Programiranje 2 (vježbe)

Vedran Šego

4. ožujka 2012.

Sadržaj

Poglavlje 1

Ponavljanje

Prije nego krenemo na novo gradivo, ponoivimo prvo što se radilo u Programiranju 1. Proći ćemo kroz osnovne izraze, petlje, nizove i funkcije. Ukoliko neki od zadataka ne znate riješiti **samostalno**, preporuča se da ponovite gradivo iz tog dijela Programiranja 1. **U ostatku ovog kolegija smatra se da je student savladao gradivo Programiranja 1**.

1.1 Osnovni programi

Zadatak 1.1.1. *Što ispisuju sljedeći isječci kôda (poštujte eventualne razmake, skokove u novi red i sl!), te koje će vrijednosti varijable poprimiti nakon što se taj kôd izvrši?*

a)
```
int a = 1, b = 2, c = 3, d = 4;
printf("%d\n%d", ((a /= 2) ? b++ : d++), --c);
```

b)
```
int a = 4, b = 3, c = 2, d = 1;
printf("%d\n%d", b++, ((c /= 6) ? --d : --a));
```

c)
```
int a = 4, b = 3, c = 2, d = 1;
printf("%d\n%d", ((b /= 6) ? a++ : d++), --c);
```

d)
```
int a = 1, b = 2, c = 3, d = 4;
printf("%d\n%d", ++a, ((d /= 8) ? c-- : b--));
```

Rješenje. Riješit ćemo podzadatak a).

Prvi redak izraza deklarira četiri varijable: `a`, `b`, `c` i `d`, sve tipa `int` (cijeli broj). Odmah po deklaraciji, varijablama se pridružuju vrijednosti: 1 (varijabli `a`), 2 (varijabli `b`), itd. Pogledajmo sada drugi redak.

Izraze treba analizirati "iznutra", kao u matematici (prvo se evaluira ono što je najdublje u zagradama, uz poštivanje prioriteta među operatorima tamo gdje zagrada nema). Izraz koji gledamo je oblika:

```
printf("A", B, C);
```

`"A"` objašnjava kako se vrijednosti ispisuju. Sve što piše između navodnika ispisuje se točno kako piše osim znakova koji se nalaze iza `%` i `\`. Da podsjetimo, izrazi oblika `%nesto` zovu se formati i zamjenjuju se vrijednostima izraza koji se nalaze iza `A` (ovdje `B` i `C`). Pri tome, `%d` označava cjelobrojni izraz. Znakovi ispred kojih se nalazi `\` su specijalni znakovi. Ovdje imamo `\n` što označava skok u novi redak.

Drugi izraz, koji smo ovdje označili s `B`, je

```
((a /= 2) ? b++ : d++)
```

Očito, vanjske zagrade ovdje nemaju svrhu (postavljene su radi preglednosti), pa je naš izraz ekvivalentan izrazu

```
(a /= 2) ? b++ : d++
```

Ovdje je riječ o uvjetnom operatoru, koji ima opći oblik

```
UVJET ? IZRAZ1 : IZRAZ2
```

Prvo se evaluira `UVJET`; kod nas je to izraz `a /= 2`. To je skraćeni zapis izraza

```
a = a / 2
```

Pridruživanje ima manji prioritet od dijeljenja, pa ovaj izraz odgovara izrazu

```
a = (a / 2)
```

Dakle, prvo se `a` dijeli s 2, a zatim se rezultat ponovno posprema u varijablu `a`. Drugim riječima, izraz `a /= 2` znači "podijeli `a` s 2".

Pošto su i `a` i 2 cijeli brojevi, dijeljenje je cjelobrojno, tj. `1/2` $= 0$. To znači da je nova vrijednost varijable `a` nula.

Povratna vrijednost svakog pridruživanja je upravo vrijednost koja je pridružena, pa će tako vrijednost cijelog izraza `a /= 2` biti nula. No, to je naš `UVJET` koji služi kao uvjet uvjetnog operatora.

Vrijednost 0 označava LAŽ, pa se evaluira `IZRAZ2`. Vrijednost cijelog izraza `B` će biti upravo ono što vrati `IZRAZ2`. Pri tome, `IZRAZ1` se uopće ne evaluira. Da je `UVJET` bio bilo što različito od nule (dakle, ISTINA), evaluirao bi se `IZRAZ1` i njegov rezultat bi bio povratna vrijednost izraza `B`.

`IZRAZ2` je `d++`. Riječ je o inkrement operatoru koji:

1. poveća `d` za 1
2. vrati **staru** vrijednost varijable `d` (onu prije povećanja), jer su plusevi iza varijable

U našem slučaju, varijabla `d` ima vrijednost 4, pa će se povećati na 5, a rezultat izraza `IZRAZ2`, pa time i cijelog izraza `B`, bit će 4.

Preostaje još evaluirati izraz `C`, tj. `--c` – dekrement operator koji:

1. smanji `c` za 1
2. vrati **novu** vrijednost varijable `c` (onu nakon povećanja), jer su minusi ispred varijable

U našem slučaju, varijabla `c` ima vrijednost 3, pa će se smanjiti na 2, što će biti i povratna vrijednost izraza `C`.

Ostaje vidjeti što će se ispisati. To vidimo uvrštavanjem vrijednosti izraza `B` i `C` u početni izraz:

`printf("A", B, C);` $\Rightarrow$ `printf("%d\n%d", 4, 2);`

Dakle, programski isječak će ispisati:

```
4
2
```

a vrijednosti varijabli će biti (čitamo posljednje vrijednosti varijabli iz prethodnog razmatranja):

`a = 0`, `b = 2` (nije se mijenjala), `c = 2`, `d = 5`.

□

Zadatak 1.1.2. *Napišite program koji:*

a) učitava dva cijela broja i, ako su oba parna ispisuje njihovu sumu; inače treba ispisati produkt

b) učitava tri realna broja i ispisuje najvećeg među njima

c) učitava tri cijela broja i ispisuje najbližeg nuli (najmanjeg po apsolutnoj vrijednosti)

Rješenje. Ponovno ćemo riješiti podzadatak a).

Da bismo računalu objasnili što treba raditi, potrebno je prvo objasniti sebi, na razini "recepta". Kako bismo mi (umjesto računala) riješili zadani problem? Po koracima:

1. pitati korisnika koje brojeve želi zadati
2. provjeriti jesu li učitani brojevi parni
3. ovisno o odgovoru na prethodno pitanje, ispisati sumu ili produkt

C ne zna općenito "pitati korisnika za brojeve", dok ne zna koliko brojeva ima i koji je njihov tip, pa se točka 1 svodi na:

"Pitaj korisnika za prvi broj. Pitaj korisnika za drugi broj."

Prevedeno na sintaksu C-a:

```
printf("Unesite prvi broj: ");
scanf("%d", &prvi_broj);
printf("Unesite drugi broj: ");
scanf("%d", &drugi_broj);
```

C nema niti funkciju koja će reći je li broj paran, ali zna izračunati ostatak djeljenja s nekim brojem, pa se drugo pitanje svodi na

"Ako je prvi broj djeljiv s 2 i ako je drugi broj djeljiv s 2... inače..."

tj.

"Ako je ostatak pri dijeljenju prvog broja s 2 jednak nuli i
ako je ostatak pri dijeljenju drugog broja s 2 jednak nuli...
inače..."

Prevedeno na sintaksu C-a:

```
if ((prvi_broj % 2 == 0) && (drugi_broj % 2 == 0))...
else...
```

Suma, odnosno produkt, se ispisuju direktno, pa treću točku možemo odmah "prevesti" u C:

Ako su brojevi parni: `printf("%d\n", prvi_broj + drugi_broj);`
Inače: `printf("%d\n", prvi_broj * drugi_broj);`

Svakom programu potrebno je dodati nekakva zaglavlja, koja se u početku uče "na pamet", a kasnije se objašnjavaju dijelovi. Uz njih, treba dodati i deklaracije varijabli: popisi varijabli koje mislimo koristiti u programu, uz navedene tipove podataka koji će se pohranjivati u tim varijablama. Ovdje, mi namjeravamo koristiti dvije cjelobrojne varijable (deklaracija se nalazi u liniji 4 rješenja).

Konačno, rješenje izgleda ovako:

```
1  #include <stdio.h>
2
3  int main(void) {
4    int prvi_broj, drugi_broj;
5
6    printf("Unesite prvi broj: ");
7    scanf("%d", &prvi_broj);
8    printf("Unesite drugi broj: ");
9    scanf("%d", &drugi_broj);
10
11   if ((prvi_broj % 2 == 0) && (drugi_broj % 2 == 0))
12     printf("%d\n", prvi_broj + drugi_broj);
13   else
14     printf("%d\n", prvi_broj * drugi_broj);
15
16   return 0;
17 }
```

□

Uputa. Podzadatak b) možemo riješiti jednostavnom provjerom sljedećih uvjeta:

1. $a \geq b$ i $a \geq c$,
2. $b \geq a$ i $b \geq c$,
3. $c \geq a$ i $c \geq b$,

te ispisom odgovarajućeg broja ako je neki uvjet zadovoljen (a bit će barem jedan). Između `if()`-ova stavite `else`-ove, da ne bi došlo do više od jednog ispisa ako je više uvjeta zadovoljeno (npr. kad su dva broja međusobno jednaki i veći od trećeg).

U podzadatku c) treba samo spretno izračunati apsolutne vrijednosti. To je najlakše napraviti upotrebom uvjetnog operatora. Na primjer, za varijablu `a`:

```
abs_a = (a < 0 ? -a : a);
```

Naravno, pomoćnu varijablu `abs_a` treba deklarirati i ona mora biti istog tipa kao varijabla `a`. □

1.2 Petlje

Zadatak 1.2.1. *Napišite dio programa koji učitava cijele brojeve dok ne učita nulu. Program treba ispisati:*

a) koliko je neparnih brojeva učitano,

b) sumu svih učitanih brojeva čija je apsolutna vrijednost prost broj,

c) koliko je učitanih brojeva strogo veće od prvog, te

d) sumu svih znamenaka svih učitanh brojeva.

Rješenje. Riješit ćemo podzadatke a) i b).

Svi zadaci, u osnovi, kažu isto:

1. učitaj broj
2. ako je broj jednak nuli, prekini učitavanje i ispiši rezultat
3. napravi nešto s brojem
4. ponovi opisano

Ovdje je najpraktičnije koristiti `while()` petlju koja kaže "dok je zadovoljen neku uvjet, radi nešto". Petlju možemo postaviti na dva načina:

1. "Učitaj prvi broj; dok je zadnji učitani broj različit od nule, napravi što treba i učitaj idući broj."

```
scanf("%d", &x);
while (x != 0) {
  napravi sto vec treba s x;
  scanf("%d", &x);
}
```

2. "Radi (bezuvjetno) sljedeće: učitaj broj, ako je učitani broj jednak nuli, prekini; ako nije, napravi što treba."

```
while (1) {
  scanf("%d", &x);
  if (x == 0) break;
  napravi sto vec treba s x;
}
```

Mi ćemo koristiti drugi pristup.

Riješimo sada onaj dio "napravi što već treba s `x`". U podzadatku a), to se svodi na "provjeri je li učitani broj neparan i, ako je, povećaj neki brojač (označit ćemo ga s `br_nep`):

```
if (x % 2 == 1) br_nep++;
```

Oprez: Na početku (prilikom deklaracije), nužno moramo zadati početnu vrijednost za `br_nep` i ona mora biti nula (što je početna vrijednost svakog brojanja, na primjer kad nešto brojimo "na prste")!

Podzadatak b) je malo složeniji: treba provjeriti da li je broj `x` prost. To radimo tako da provjerimo postoji li neki broj strogo veći od 1 i strogo manji od `x` (po apsolutnoj vrijednosti) s kojim je `x` djeljiv. Ako postoji, treba zapamtiti (npr. u varijabli `prost`) da broj nije prost. Ako nismo "ništa zapamtili", to znači da nismo mijenjali vrijednost varijable `prost`, što znači da njena početna vrijednost mora biti ISTINA (u C-u, to je 1):

```
prost = 1;
aps_vr_od_x = (x < 0 ? -x : x);
for (i = 2; i < aps_vr_od_x; i++)
  if (x % i == 0) {
    prost = 0;
    break;
  }
```

Ovdje `break` nije nužan, ali nekako je logično prekinuti provjeru čim otkrijemo da broj nije prost (brže je od nepotrebnog provjeravanja do kraja).

Primijetimo da će prikazani dio programa zaključiti da su 0 i 1 prosti brojevi. Da bismo to ispravili, najlakše je promijeniti početnu vrijednost varijable prost, tako da ne bude 1 nego da ovisi o tome je li `x` nula ili jedan (pa nije prost) ili je veći (pa **možda** je prost). Također, ako smo već zaključili da nije prost, ne treba ulaziti u petlju. Sve zajedno:

```
aps_vr_od_x = (x < 0 ? -x : x);
prost = (aps_vr_od_x <= 1 ? 0 : 1);
if (prost)
  for (i = 2; i < aps_vr_od_x; i++)
    if (x % i == 0) {
      prost = 0;
      break;
    }
```

Nakon petlje znamo je li broj `x` prost, pa možemo reći: "ako je `x` prost, povećaj sumu prostih brojeva za `x`".

```
if (prost) suma_prostih += x;
```

Konačno, zajedničko rješenje podzadataka a) i b) izgleda ovako:

```
int x, i, br_nep = 0, suma_prostih = 0, prost,
    aps_vr_od_x;

while (1) {
  scanf("%d", &x);
  if (x == 0) break;

  if (x % 2 == 1) br_nep++;

  aps_vr_od_x = (x < 0 ? -x : x);
  prost = (aps_vr_od_x <= 1 ? 0 : 1);
  if (prost)
    for (i = 2; i < aps_vr_od_x; i++)
      if (x % i == 0) {
        prost = 0;
        break;
      }
  if (prost) suma_prostih += x;
}
printf("Broj neparnih: %d\n", br_nep);
printf("Suma prostih: %d\n", suma_prostih);
```

Naravno, za napisati cijeli program (a ne samo dio programa), treba dodati i zaglavlja (kao u prethodnom zadatku). □

Napomena 1.2.1. *Izraz*

```
prost = (aps_vr_od_x <= 1 ? 0 : 1);
```

možemo zapisati i ovako:

```
prost = (aps_vr_od_x > 1);
```

Naravno, ovo skraćivanje nije nužno; dulja verzija je podjednako ispravna.

1.3 Nizovi

Zadatak 1.3.1. *Napišite programski isječak koji učitava prirodni broj $n \leq 17$, te n cijelih brojeva i zatim:*

a) sortira i ispisuje brojeve uzlazno po vrijednosti zadnje znamenke

b) ispisuje sve brojeve koji su veći ili jednaki zadnjem učitanom

c) ispisuje produkt svih brojeva koji su veći ili jednaki predzadnjem učitanom (pretpostavite da je $n > 1$)

d) pomoću Hornerovog algoritma izračunava i ispisuje vrijednost $p(a_1)$, gdje su $(a_i)_{i=0}^{n-1}$ učitani brojevi i

$$p(x) = \sum_{i=0}^{n-1} a_i^2 x^i.$$

Rješenje. Riješit ćemo podzadatke a) i b).

Nizove deklariramo kao i obične varijable, uz dodatak navođenja maksimalne duljine niza:

```
int n;
int a[17];
```

Ako maksimalna duljina niza nije poznata, zadatak treba riješiti bez upotrebe nizova ili pomoću dinamičkih nizova!

Elementima niza pristupamo kao i običnim varijablama, uz navođenje indeksa (indeksi kreću od nule!):
Ispis drugog elementa niza: `printf("%d", a[1]);`
Učitavanje tećeg elementa niza: `scanf("%d", &a[2]);`

Dakle, učitavanje niza od $n \leq 17$ realnih brojeva je rutina:

```
int n, i, a[17];
scanf("%d", &n);
for (i = 0; i < n; i++) scanf("%d", &a[i]);
```

U podzadatku a) traži se uzlazni sort niza. Postoji mnogo sortova, od kojih su neki lakši za napisati, a neki se brže izvršavaju. Mi ćemo upotrijebiti klasični (engl. *selection*) sort koji kaže:

"Za sve indekse i, j takve da je $i < j$ provjeri je li $a_i > a_j$ (tj. jesu li brojevi a_i i a_j u pogrešnom redoslijedu); ako da, onda ih zamijeni."

Pri tome treba paziti da je kriterij sorta vrijednost zadnje znamenke, pa izraz $a_i > a_j$ **ne znači** "a_i strogo veće od a_j, nego "zadnja znamenka od a_i strogo veća od zadnje znamenke od a_j".

Zadnju znamenku možemo izračunati kao ostatak pri dijeljenju apsolutne vrijednosti broja s 10. Tu apsolutnu vrijednost računamo pomoću funkcije `abs()` iz biblioteke `stdlib`

(treba na početku programa dodati `#include <stdlib.h>`) ili ju možemo samostalno napisati:

```
unsigned abs(int x) {
  return (x < 0 ? -x : x);
}
```

Naš sort izgleda ovako:

```
for (i = 0; i < n - 1; i++)
  for (j = i + 1; j < n; j++)
    if (abs(a[i]) % 10 > abs(a[j]) % 10) {
      int temp = a[i];
      a[i] = a[j];
      a[j] = temp;
    }
```

Primijetimo da ovaj dio kôda mijenja niz (tj. poredak njegovih elemenata), što narušava uvjet zadatka b) (jer se može promijeniti pozicija zadnjeg elementa). Zbog toga prvo treba riješiti podzadatak b) (ako ih rješavamo u jednom programu)!

Podzadatak b) zapravo kaže: "protrči po svim brojevima i ispiši one koji su veći ili jednaki a_{n-1}":

```
for (i = 0; i < n; i++)
  if (a[i] >= a[n-1])
    printf("%d\n", a[i]);
```

Konačno zajedničko rješenje podzadataka a) i b) je:

```
int n, i, j, a[17];

scanf("%d", &n);
for (i = 0; i < n; i++) scanf("%d", &a[i]);

printf("Svi brojevi veci ili jednaki %d:\n", a[n-1]);
for (i = 0; i < n; i++)
  if (a[i] >= a[n-1])
    printf("%d\n", a[i]);

for (i = 0; i < n - 1; i++)
  for (j = i + 1; j < n; j++)
    if (a[i] % 10 > a[j] % 10) {
      int temp = a[i];
      a[i] = a[j];
      a[j] = temp;
    }
printf("Sortirani niz:\n");
for (i = 0; i < n; i++)
```

```
20   printf("%d\n", a[i]);
```

U podzadatku d) spominje se Hornerov algoritam. Riječ je o poznatom algoritmu za brzo računanje vrijednosti polinoma u zadanoj točki (ponoviti samostalno). □

1.4 Funkcije

S funkcijama se u C-u nužno srećemo od prvog programa. Naime, famozni `main()` je obična funkcija, samo što ima specijalno ime.

Zadatak 1.4.1. *Napišite funkciju koja kao argumente uzima dva realna broja, te vraća većeg od njih. Dodatno, napišite program koji prikazuje kako se funkcija upotrebljava.*

Rješenje. Funkcije imaju standardizirano zaglavlje:

```
tip_povratne_vrijednosti ime_funkcije(argumenti)
```

gdje se argumenti navode slično kao i deklaracije varijabli. Razlika je u tome što svakom argumentu treba zadati tip, te što se argumenti odvajaju zarezima (a ne točka-zarezima). Ako argumenata nema, zagrade svejedno treba navesti, a unutar njih se navodi ključna riječ `void`.

Naša funkcija uzima dva realna broja, te vraća realnu vrijednost, pa će njena deklaracija (nazovimo funkciju `max`) biti:

```
double max(double x, double y)
```

Da bi funkcija vratila vrijednost, potrebno je u tijelu navesti:

```
return vrijednost_koja_se_vraca;
```

Ključna riječ `return` **prekida izvršavanje funkcije** i postavlja zadanu vrijednost kao povratnu vrijednost funkcije. Konačno, funkcija izgleda ovako:

```
double max(double x, double y) {
  double veci = (x > y ? x : y);
  return veci;
}
```

U `return` možemo navesti i cijeli izraz, a ne nužno samo varijablu, pa funkciju možemo zapisati i ovako:

```
double max(double x, double y) {
  return (x > y ? x : y);
}
```

Ovako definiranu funkciju pozivamo jednako kao i funkcije koje "dolaze s C-om" (npr. `printf()`), tj. kao obične izraze. Na primjer, prikazanu funkciju možemo pozvati iz glavnog programa ovako:

```
double veci, a, b;
...
veci = max(a, b);
```

Pri tome će se parametri `a` i `b` proslijediti u funkciju redom kojim su zadani. To znači da će varijabla `x` (ona u funkciji `max()`) poprimiti vrijednost koju ima varijabla `a` iz glavnog programa, dok će `y` (opet, iz funkcije `max()`) poprimiti vrijednost koju ima varijabla `b` iz glavnog programa.

Time varijable `x` i `y` postaju **kopije** varijabli `a` i `b` (dakle **NE** iste varijable, nego nove varijable s istom vrijednošću).

Prilikom poziva, funkcija se izvršava, te se njena povratna vrijednost pridružuje varijabli `veci`.

Naravno, funkcija se može pozivati i iz drugih funkcija (a ne samo iz glavnog programa). Također, prilikom poziva funkcije, kao argumente možemo zadati i izraze. Na primjer:

```
double veci, a, b, c, d;
...
veci = max(a + b * c, d);
```

Nakon izvršavanja ovog dijela kôda, u varijabli `veci` će biti veća od vrijednosti $a + b \cdot c$ i d.

Konačno rješenje zadatka ćemo napraviti direktnim ispisom povratne vrijednosti funkcije:

```
1  #include <stdio.h>
2
3  double max(double x, double y) {
4    return (x > y ? x : y);
5  }
6
7  int main(void) {
8    double a, b;
9
10   printf("a = "); scanf("%lf", &a);
11   printf("b = "); scanf("%lf", &b);
12
13   printf("Veci je %g.\n", max(a, b));
14
15   return 0;
16 }
```

□

Napomena 1.4.1. *Kako smo rekli, argumenti u funkciji su kopije onih s kojima je funkcija pozvana. To znači da promjene vrijednosti argumenata se u C-u ne odražavaju na varijable koje su zadane kao parametri prilikom poziva funkcije. Na primjer, ako imamo funkciju*

```
int f(int a) {
  a++;
  return a;
}
```

te ju pozovemo s

```
int x = 17;
printf("1: %d", x);
printf("2: %d", f(x));
printf("3: %d", x);
```

ispis će biti

```
1: 17
2: 18
3: 17
```

a ne

```
1: 17
2: 18
3: 18
```

Svojevrsna iznimka su elementi nizova. Ako elementu niza promijenimo vrijednost, promjena će se odraziti na odgovarajući element pozivnog parametra! Ovakvo ponašanje nije neobično, a zašto do njega dolazi, vidjet ćemo u kasnijim poglavljima.

Zadatak 1.4.2. *Napišite funkciju koja kao argumente uzima niz realnih brojeva i cijeli broj n (koji označava duljinu niza). Funkcija treba uzlazno sortirati niz.*
Napišite i kako se funkcija poziva (pretpostavite da imate učitan niz `a` *s* `n` *elemenata).*

Rješenje. Očito, funkcija ne vraća ništa (jer se u zadatku ne navodi što bi trebala vratiti). To znači da je tip njene povratne vrijednosti `void`.

Kad u funkciju želimo "poslati" niz, argument funkcije deklariramo na sljedeći način:

```
tip_elementa_niza ime_varijable[];
```

dakle jednako kao deklaracija nizovne varijable, ali bez navođenja najveće duljine.

Konačno rješenje zadatka:

```
1 #include <stdio.h>
2
3 void sort(double x[], int n) {
4   int i, j;
5   for (i = 0; i < n - 1; i++)
6     for (j = i + 1; j < n; j++)
7       if (x[i] > x[j]) {
8         int temp = x[i];
```

```
9           x[i] = x[j];
10          x[j] = temp;
11        }
12  }
```

Poziv funkcije: `sort(a, n)`

Dakle, funkciju pozivamo uobičajenim popisivanjem parametara, bez obzira što je jedan od njih niz. Česte **GREŠKE** su:
`sort(a[], n);` i
`sort(a[n], n);` □

Zadatak 1.4.3. *Napišite funkciju koja kao parametre prima prirodne brojeve* `a` *i* `b`, $a, b > 1$, *a kao rezultat ne vraća ništa. Funkcija treba "nacrtati" graf za sve brojeve* k *takve da je* $b < k \leq a + b$, *koji u svakom retku ispisuje broj* k, *te* $f(k)$ *"povisilica" (znak "#"), pri čemu s* $f(k)$ *označavamo broj brojeva* x *takvih da je* $a \leq x \leq b$ *i* k *djeljiv s* x *(*$f(k)$ *služi za opis zadatka, te ga* **ne morate** *definirati kao posebnu funkciju!), te preko trećeg parametra vratiti najveći broj "povisilica" ispisanih u jednom retku (ako nije ispisana niti jedna, treba vratiti nulu).*
Na primjer, za `a=3, b=9`*, treba ispisati (bez objašnjenja koja su u zagradama):*

`10: #` *(od brojeva iz skupa* $\{3, 4, \ldots, 9\}$*, broj* 10 *je djeljiv samo s brojem* 5*)*

`11:` *(broj* 11 *nije djeljiv s niti jednim brojem iz skupa* $\{3, 4, \ldots, 9\}$*)*

`12: ###` *(od brojeva iz skupa* $\{3, 4, \ldots, 9\}$*, broj* 12 *je djeljiv s brojevima* 3, 4 *i* 6*)*

te preko trećeg parametra vratiti vrijednost 3 *(*$= \max\{1, 0, 3\}$*).*
Napomene: *Nije dozvoljeno korištenje funkcija iz* `math.h` *i nizova! Ako ne znate napisati funkciju, onda napišite dio programa (koji umjesto vraćanja vrijednosti ima ispis), no takvo rješenje donosi najviše* 15 *bodova. Ako ne znate napisati funkciju koja vraća vrijednost preko parametara, napišite ju tako da vraća vrijednost pomoću* `return`*, no takvo rješenje nosi najviše* 20 *bodova.*

Rješenje. Ovdje je upotrijebljen naizgled kompliciran način ispisa: preko nekakvog "grafa". No, crtanje "grafa" u formatu
`k:` $\underbrace{\texttt{\#\#\#...\#}}_{k}$
je jednostavno:

```
printf("%d: ", k);
for (i = 0; i < k; i++) printf("#");
printf("\n");
```

Ostatak zadatka je kombiniranje već viđenih zadataka, pa je konačno rješenje:

```
1  int f(int k, int a, int b) {
2    int cnt = 0, x;
3    for (x = a; x <= b; x++)
4      if (k % x == 0) cnt++;
5    return cnt;
```

```
6  }
7
8  void zad(int a, int b, int *br) {
9    int k, i;
10   *br = 0;
11   for (k = b + 1; k <= a + b; k++) {
12     int fk = f(k, a, b);
13     if (fk > *br) *br = fk;
14     printf("%d: ", k);
15     for (i = 0; i < fk; i++) printf("#");
16     printf("\n");
17   }
18 }
```

Primjer programa za testiranje funkcije:

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

  zad(3, 9, &res);
  printf("Rezultat: %d\n", res);

  return 0;
}
```

□

1.5 Zadaci za samostalnu vježbu

Zadatak 1.5.1. *Napišite program (ne samo dio programa!) koji učitava niz cijelih brojeva* `x` *dok ne učita broj* 17 *ili ukupno* 314 *brojeva. Program treba niz* `x` *sortirati silazno prema sumi druge (s lijeva) i zadnje znamenke (ako broj nema neku od traženih znamenaka, za njenu vrijednost se uzima nula), te ispisati tako dobiveni niz.*
Napomena: *Nije dozvoljeno korištenje funkcija iz* `math.h` *i dodatnih nizova!*

Zadatak 1.5.2. *Napišite program (ne samo dio programa!) koji učitava niz realnih brojeva* `x` *dok ne učita broj* 3.25 *ili ukupno* 17 *brojeva. Program treba niz* `x` *sortirati uzlazno prema broju znamenaka cjelobrojnog dijela broja (tj. prema broju znamenaka lijevo od decimalne točke), te ispisati tako dobiveni niz.*
Napomena: *Nije dozvoljeno korištenje funkcija iz* `math.h` *i dodatnih nizova!*

Zadatak 1.5.3. *Napišite program (ne samo dio programa!) koji učitava niz cijelih brojeva* `x` *dok ne učita broj* 314 *ili ukupno* 17 *brojeva. Program treba ispisati one elemente niza* `x` *za koje je druga znamenka (s lijeva) jednaka znamenci jedinica (npr.* 12342*).*
Napomena: *Nije dozvoljeno korištenje funkcija iz* `math.h` *i dodatnih nizova! Ako broj nema neku od traženih znamenaka, za njenu vrijednost uzima se nula.*

Zadatak 1.5.4. *Napišite program (ne samo dio programa!) koji učitava niz cijelih brojeva* `x` *dok ne učita broj* 17 *ili ukupno* 314 *brojeva. Program treba ispisati one elemente niza* `x` *za koje je treća znamenka (s lijeva) manja od znamenke desetica (npr.* 12342*).*
Napomena: *Nije dozvoljeno korištenje funkcija iz* `math.h` *i dodatnih nizova! Ako broj nema neku od traženih znamenaka, za njenu vrijednost uzima se nula.*

Zadatak 1.5.5. *Napišite funkciju koja kao parametre prima prirodne brojeve* `a` *i* `b`*,* $a, b > 1$*, a kao rezultat ne vraća ništa. Funkcija treba "nacrtati" graf koji za sve brojeve* k*, takve da je* $a < k \leq b$*, u svakom retku ispisuje broj* k*, te onoliko križića (malo slovo "x") koliko* k *ima različitih prostih djelitelja, te preko trećeg parametra vratiti ukupan broj ispisanih križića (ako nije ispisana niti jedan, treba vratiti nulu).*
Na primjer, za `a=27, b=30`*, treba ispisati (bez objašnjenja koja su u zagradama):*

```
28: xx    (različiti prosti djelitelji od 28 su 2 i 7)
29: x     (29 je jedini prosti djelitelj od 29)
30: xxx   (različiti prosti djelitelji od 30 su 2, 3 i 5)
```

te preko trećeg parametra vratiti vrijednost 6 $(= 2 + 1 + 3)$*.*
Napomene: *Nije dozvoljeno korištenje funkcija iz* `math.h` *i nizova! Ako ne znate napisati funkciju, onda napišite dio programa (koji umjesto vraćanja vrijednosti ima ispis), no takvo rješenje donosi najviše* 15 *bodova. Ako ne znate napisati funkciju koja vraća vrijednost preko parametara, napišite ju tako da vraća vrijednost pomoću* `return`*, no takvo rješenje nosi najviše* 20 *bodova.*

Zadatak 1.5.6. *Napišite funkciju koja kao parametre prima prirodne brojeve* `a` *i* `b`*,* $a, b > 1$*, a kao rezultat ne vraća ništa. Funkcija treba "nacrtati" graf za sve brojeve* k *takve da je* $a < k \leq a + b$*, koji u svakom retku ispisuje broj* k*, te* $a + \binom{b}{k}$ *minusa (znak "-"), pri čemu je* $\binom{x}{y} = \frac{x!}{y!(x-y)!}$*,* a $n! = 1 \cdot 2 \cdots n$ *(po definiciji je* $0! = 1$*), te preko trećeg parametra vratiti ukupni broj minusa ispisanih u jednom retku (ako nije ispisan niti jedan, treba vratiti nulu).*
Na primjer, za `a=4, b=3`*, treba ispisati (bez objašnjenja koja su u zagradama):*

```
5: ----------------------   (21 minus, jer je (4+3 5) = 7!/5!2! = 21)
6: -------                  (7 minusa, jer je (4+3 6) = 7!/6!1! = 7)
7: -                        (1 minus, jer je (4+3 7) = 7!/7!0! = 1)
```

te preko trećeg parametra vratiti vrijednost 29 $(= 21 + 7 + 1)$*.*
Napomene: *Nije dozvoljeno korištenje funkcija iz* `math.h` *i nizova! Ako ne znate napisati funkciju, onda napišite dio programa (koji umjesto vraćanja vrijednosti ima ispis), no takvo rješenje donosi najviše* 15 *bodova. Ako ne znate napisati funkciju koja vraća vrijednost preko parametara, napišite ju tako da vraća vrijednost pomoću* `return`*, no takvo rješenje nosi najviše* 20 *bodova.*

Zadatak 1.5.7. *Napišite funkciju koja kao parametre prima prirodne brojeve* `a` *i* `b`*,* $a, b > 1$*, a kao rezultat ne vraća ništa. Funkcija treba "nacrtati" graf za sve brojeve* k *takve da je* $1 < k \leq b$*, koji u svakom retku ispisuje broj* k*, te* $M(a, k)$ *kružića (malo slovo "o"), pri čemu je* $M(a, k)$ *najveća zajednička mjera brojeva* a *i* k *(najveći prirodni*

broj $n \leq a, k$ takav da su i a i k djeljivi s n), te preko trećeg parametra vratiti najveći broj kružića ispisanih u jednom retku (ako nije ispisana niti jedan, treba vratiti nulu).
Na primjer, za `a=9, b=4`*, treba ispisati (bez objašnjenja koja su u zagradama):*

```
2: o
```
($M(9,2) = 1$ jer je 1 najveći prirodni broj s kojim su djeljivi i 9 i 2)

```
3: ooo
```
($M(9,3) = 3$ jer je 3 najveći prirodni broj s kojim su djeljivi i 9 i 3)

```
4: o
```
($M(9,4) = 1$ jer je 1 najveći prirodni broj s kojim su djeljivi i 9 i 4)

te preko trećeg parametra vratiti vrijednost 3 *(*$= \max\{1,3,1\}$*).*
Napomene: *Nije dozvoljeno korištenje funkcija iz* `math.h` *i nizova! Ako ne znate napisati funkciju, onda napišite dio programa (koji umjesto vraćanja vrijednosti ima ispis), no takvo rješenje donosi najviše* 15 *bodova. Ako ne znate napisati funkciju koja vraća vrijednost preko parametara, napišite ju tako da vraća vrijednost pomoću* `return`*, no takvo rješenje nosi najviše* 20 *bodova.*

Poglavlje 2

Rekurzije

Rekurzivne funkcije ili, kraće, rekurzije, su funkcije koje pozivaju same sebe. Proizlaze iz raznih matematičkih i drugih problema, a posebno su česte u kombinatorici. Također, važne su za razne stablaste strukture podataka (više o tome na kolegiju Strukture podataka i algoritmi).

U ovom poglavlju proći ćemo kroz evaluiranje rekurzija (kako iz gotovog kôda otkriti što on računa) te kroz pisanje jednostavnih (zadanih formulom) i složenijih (zadanih pričom) rekurzija.

2.1 Uvod

Krenimo s tipičnim matematičkim primjerom rekurzivne funkcije – Fibonaccijevi brojevi. Pri tome je bitno odmah u startu naglasiti da je primjer praktičan jer proizlazi iz jednostavne, dobro poznate formule, no zapravo Fibonaccijeve brojeve na računalu **nikad** ne smijemo realizirati pomoću rekurzije (v. napomenu 2.2.1)!

Primjer 2.1.1 (Fibonaccijevi brojevi (rekurzivno))**.** *Fibonaccijeve brojeve definiramo sljedećom formulom:*

$$F_n := \begin{cases} 0, & n = 0, \\ 1, & n = 1, \\ F_{n-2} + F_{n-1}, & n \in \mathbb{N},\ n > 1. \end{cases}$$

Napišimo funkciju za računanje Fibonaccijevih brojeva direktno prema prikazanoj definiciji:

```
1 long int fib(long int n) {
2   if (n <= 1) return n;
3   return fib(n-1) + fib(n-2);
4 }
```

Kako se evaluiraju rekurzije?

Prije nego krenemo analizirati rekurziju, potrebno je prisjetiti se gdje i kada varijable "žive". Na primjer:

```c
#include <stdio.h>

int a = 1, b = 2, c = 3, d = 4;

void f(int a) {
  int b = 12;

  printf("  a=%d, b=%d, c=%d, d=%d\n", a, b, c, d);
  a++; b++; c++;
  printf("  a=%d, b=%d, c=%d, d=%d\n", a, b, c, d);
}

int main(void) {
  int a = 21, b = 22, d = 24;

  printf("main() 1. put:\n");
  printf("  a=%d, b=%d, c=%d, d=%d\n", a, b, c, d);
  printf("Funkcija 1. put:\n");
  f(c);
  printf("Funkcija 2. put:\n");
  f(c);
  printf("main() 2. put:\n");
  printf("  a=%d, b=%d, c=%d, d=%d\n", a, b, c, d);

  return 0;
}
```

Ispis ovog programa će biti:

```
main() 1. put:
  a = 21, b = 22, c = 3, d = 24
Funkcija 1. put:
  a = 3, b = 12, c = 3, d = 4
  a = 4, b = 13, c = 4, d = 4
Funkcija 2. put:
  a = 4, b = 12, c = 4, d = 4
  a = 5, b = 13, c = 5, d = 4
main() 2. put:
  a = 21, b = 22, c = 5, d = 24
```

Primijetimo nekoliko detalja:

1. Imena varijabli ne igraju ulogu između poziva funkcije i tijela funkcije. Zbog toga, te zbog poziva `f(c)`, varijabla `a` u funkciji poprima vrijednost varijable `c` iz glavnog programa.

2. Varijabla `b` u funkciji `f()` "živi" samo dok živi i funkcija, tj. njena vrijednost se ne čuva između dva poziva funkcije.

3. Varijabla `c` pripada cijelom programu, pa njena vrijednost raste sa svakim evaluiranjem izraza `c++` (tj. vrijednost se ne gubi)

4. Postoje dvije varijable `d`: jedna pripada cijelom programu, a druga samo funkciji `main()`. Pošto funkcija `main()` ima svoju varijablu `d`, u njoj se ne "vidi" globalna varijabla `d`. Funkcija nema svoju varijablu `d`, pa pod tim imenom vidi onu globalnu.

Funkciju smo mogli napisati i na sljedeći način (koristeći različite nazive za sve različite varijable):

```
#include <stdio.h>

int glob_a = 1, glob_b = 2, glob_c = 3, glob_d = 4;

void f(int f_a) {
  int f_b = 12;

  printf("  a=%d, b=%d, c=%d, d=%d\n",
    f_a, f_b, glob_c, glob_d);
  f_a++; f_b++; glob_c++;
  printf("  a=%d, b=%d, c=%d, d=%d\n",
    f_a, f_b, glob_c, glob_d);
}

int main(void) {
  int main_a = 21, main_b = 22, main_d = 24;
  printf("main() 1. put:\n");
  printf("  a=%d, b=%d, c=%d, d=%d\n",
    main_a, main_b, glob_c, main_d);
  printf("Funkcija 1. put:\n");
  f(glob_c);
  printf("Funkcija 2. put:\n");
  f(glob_c);
  printf("main() 2. put:\n");
  printf("  a=%d, b=%d, c=%d, d=%d\n",
    main_a, main_b, glob_c, main_d);

  return 0;
}
```

Dakle, u funkciji `fib()` iz primjera 2.1.1, varijabla `n` je lokalna.

Napomena 2.1.1 (Lokalne varijable)**.** *Izuzetno je važno zapamtiti da* **svaki poziv funkcije dobija svoju varijablu** `n`*, tj. promjena varijable* `n` *u nekom pozivu rekurzije ne mijenja vrijednost varijable* `n` *u funkciji koja je taj poziv izvršila!*

2.2 Evaluiranje rekurzija

Zadatak 2.2.1. *Koju vrijednost vraća poziv funkcije* `fib(5)`*, pri čemu je funkcija* `fib()` *ona koju smo definirali u primjeru 2.1.1?*

Rješenje. Idealno je pozive zapisati "stablasto", slično prikazu direktorija u Konqueroru, Windows Exploreru i sličnim programima. Pri tome za svaki poziv neke funkcije (najčešće rekurzije) zapisujemo stanja ulaznih argumenata, a "ispod" toga (kao podstablo) zapisujemo stanja lokalnih varijabli (po kojima možemo "šarati" ako se vrijednosti mijenjaju tijekom izvođenja funkcije), te daljnje pozive funkcija.

```
main:
| n=5
| printf("...", fib(5))
| fib(5)=?
+ fib(5):
| | n=5
| | return fib(n-1)+fib(n-2)=fib(4)+fib(3)
| | fib(4)+fib(3)=?
| | fib(4)=?
| + fib(4):
| | | n=4
| | | return fib(3)+fib(2)=?
| | + fib(3):
| | | | n=3
| | | | return fib(2)+fib(1)=?
| | | + fib(2):
| | | | | n=2
| | | | | return fib(1)+fib(0)=?
| | | | | fib(1):
| | | | | | n=1
| | | | | \ return 1 (jer je 1 <= 1)
| | | | + fib(0):
| | | | | | n=0
| | | | | \ return 0 (jer je 0 <= 1)
| | | | \ return fib(1)+fib(0)=1+0=1 (=>fib(2) vraca 1)
| | | + fib(1):
| | | | | n=1
| | | | \ return 1
| | | \ return fib(2)+fib(1)=1+1=2 (=>fib(3) vraca 2)
| | + fib(2):
| | | | ovdje prepisujemo cijeli "blok" fib(2)!
| | | | n=2
| | | | return fib(1)+fib(0)=?
```

```
| | | + fib(1):
| | | | | n=1
| | | | \ return 1 (jer je 1 <= 1)
| | | | fib(0):
| | | | | n=0
| | | | \ return 0 (jer je 0 <= 1)
| | | \ return fib(1)+fib(0)=1+0=1 (=>fib(2) vraca 1)
| | \ return fib(3)+fib(2)=2+1=3 (=>fib(4) vraca 3)
| + fib(3):
| | | ovdje bismo trebali prepisati cijeli "blok" fib(3)!
| | | ...
| | \ return 2
| \ return fib(4)+fib(3)=3+2=5
+ printf("...", fib(5)) -> ispisuje 5
```

□

U ovom prikazu smo znakom "plus" ("+") označili ulazak u funkciju, dok kosa crta ("\") označava kraj funkcijskog poziva. Okomita crta ("|") označava tijek funkcijskog poziva. Tako, na primjer,

```
fib(5):
| n=5
| ...
+ fib(4):
| | n=4
| | ...
| \ return 3
| ...
\ return 5
```

označava tijek funkcijskog poziva `fib(5)` koji ima lokalnu varijablu `n` vrijednosti 5 i u kojem se (između ostalog) izvršava izraz `fib(4)`, pozivom funkcije. Taj poziv ima svoju lokalnu varijablu `n` **koja nema direktne veze s varijablom** `n` **iz poziva** `fib(5)`**!** Na kraju izvršavanja, poziv `fib(4)` vraća vrijednost 3, dok poziv `fib(5)` vraća vrijednost 5.

Slijedeća modifikacija rekurzije iz primjera 2.1.1 ispisuje stanja varijabli u svakom koraku rekurzije na sličan način:

```
1 long int fib(long int n, int dubina_rekurzije) {
2   int i;
3   long int temp;
4   for (i = 0; i < dubina_rekurzije; i++) printf("  ");
5   printf("n=%d\n", n);
6   if (n <= 1) {
7     for (i = 0; i < dubina_rekurzije; i++)
8       printf("  ");
```

```
9       printf("return %d\n", n);
10      return n;
11    }
12    temp =
13      fib(n-1, dubina_rekurzije+1) +
14      fib(n-2, dubina_rekurzije+1);
15    for (i = 0; i < dubina_rekurzije; i++) printf("  ");
16    printf("return %ld\n", temp);
17    return temp;
18  }
```

Da bismo postigli "stablastu" strukturu ispisa, potrebno je znati koliko smo duboko u rekurziji. U tu svrhu koristimo još jednu lokalnu varijablu (`dubina_rekurzije`) koju povećavamo prilikog svakog rekurzivnog poziva. Pri tom nije potrebno varijablu smanjivati kad se izlazi iz rekurzije, jer je ona **lokalna za svaki funkcijski poziv**, pa se gubi priliko kraja izvršavanja funkcijskog poziva, te nas ponovno "dočeka" stara vrijednost (tj. istoimena varijabla nadređenog poziva).

Ovu varijantu funkcije iz glavnog programa pozivamo s `fib(n, 0)`, a sličan princip ispisa možemo primijeniti na sve rekurzivne funkcije (korisno za vježbanje rekurzija)!

Napomena 2.2.1 (Fibonaccijevi brojevi (nerekurzivno))**.** *Iako se računanje Fibonaccijevih brojeva pomoću rekurzije čini prirodno (jer je i definicija rekurzivna), ono je izuzetno neefikasno! Kao što možemo vidjeti u prethodnom primjeru, vrijednosti* `fib(n)` *računamo mnogo puta za većinu vrijednosti argumenta* `n` *(što manja vrijednost od* `n`*, to je više redundantnih računanja* `fib(n)`*). Ovdje je daleko efikasniji nerekurzivni pristup:*

```
unsigned long fib(unsigned n) {
  unsigned long fib1 = 0, fib2 = 1, fib3;
  unsigned i;
  if (n <= 1) return n;
  for (i = 2; i <= n; i++) {
    fib3 = fib1 + fib2;
    fib1 = fib2;
    fib2 = fib3;
  }
  return fib2;
}
```

Na testom računalu (Pentium 4, 1.7GHz) nerekruzivna verzija programa izvršava poziv `fib(1000000)` *za jednu stotinku. Rekurzivna varijanta programa se "sruši" zbog prevelike dubine rekurzije, dok za* `n=40` *rekurzivnoj verziji programa treba* 4.5 *sekunde. Za* `n=41` *treba* 7.7 *sekundi, što je čak* 70%*-tno povećanje!*

Primijetite i da `fib(1000000)` *neće vratiti milijunti Fibonaccijev broj, nego ostatak koji taj broj daje pri dijeljenju s* 2^n *(pri čemu* $n \geq 32$ *ovisi o arhitekturi računala). Probajte napisati program koji pronalazi i ispisuje najveći Fibonaccijev broj prikaziv u varijabli tipa* `unsigned long`*.*

Naravno, ovo ne znači da su rekurzije općenito loše, nego samo da treba paziti kad se primjenjuju. Neki problemi čak niti nemaju nerekurzivno rješenje!

2.3 Kreiranje rekurzija

Prilikom kreiranja rekurzivnih funkcija treba biti oprezan, kako rekurzija ne bi završila u beskonačnoj petlji i srušila program.

Napomena 2.3.1 (Terminalni uvjeti)**.** *Svaka rekurzija mora imati i terminalne uvjete, tj. način izvršavanja koji nema rekurzivnih poziva! Kod računanja Fibonaccijevih brojeva, to je uvjet*

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

Takav uvjet je nužan kako se rekurzija ne bi pozivala u nedogled. Također, treba paziti da svaki poziv rekurzije prije ili poslije može dovesti do nekog od terminalnih uvjeta. Na primjer, da je uvjet u primjeru 2.1.1 bio samo

```
2   if (n == 0) return 0;
```

onda bismo kod evaluiranja izraza (na primjer) `fib(2)` *imali:*

```
fib(2) =          fib(1) +                fib(0)
       = fib(0) +      fib(-1) +          fib(0)
       = fib(0) +   fib(-2) + fib(-3) +   fib(0)
       = ...
```

Rekurzije zadane formulom rješavamo jednostavnim "prepisivanjem u C". U praksi treba paziti da je rekurzija korektno zadana (tj. da za svaki ulaz završava u konačnom vremenu), no ovdje ćemo se baviti samo implementacijom u C-u, bez formalne provjere korektnosti.

Zadatak 2.3.1. *Napišite rekurzivnu funkciju* `f(a,b)` *koja je definirana sljedećom formulom:*

$$f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$$

$$f(a,b) = \begin{cases} 1, & a = b = 0, \\ -f(-a,b), & a < 0, b \geq 0, \\ -f(a,-b), & a \geq 0, b < 0, \\ f(-a,-b), & a < 0, b < 0, \\ f(b, a-1) + 2, & \textit{inače.} \end{cases}$$

Rješenje. Kad je zadana rekurzivna formula, rješenje zadatka je jednostavno i svodi se na prepisivanje matematičke formule u C:

```
int f(int a, int b) {
  if (a == 0 && b == 0) return 1; else
  if (a < 0 && b >= 0) return -f(-a,b); else
  if (a >= 0 && b < 0) return -f(a,-b); else
  if (a < 0 && b < 0) return f(-a,-b); else
    return f(b, a - 1) + 2;
}
```

Očito, ovdje nam `else`–ovi zapravo ne trebaju, jer `return` prekida izvršavanje funkcije. Oni su tu samo zbog preglednosti. □

Zadatak 2.3.2. *Napišite rekurzivnu funkciju* `f(a,b)` *koja je definirana sljedećom formulom:*

$$f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$$

$$f(a,b) = \begin{cases} 1, & |a-b| < 5, \\ -f(-2a,b), & a < 0, b \geq 0, \\ -f(-2b,a), & a \geq 0, b < 0, \\ f(a+1,0) + f(0,b-3), & a < 0, b < 0, \\ f(7-b,a), & \textit{inače.} \end{cases}$$

No, gotove formule rijetko nalazimo u praksi. Često su zadaci zadani opisno ("pričom" od koje treba izgraditi rekurziju).

Zadatak 2.3.3. *Napišite rekurzivnu funkciju koja za skup cijelih brojeva (zadan pomoću niza) vraća broj podskupova čija je suma djeljiva sa* 17*. Uz niz i njegovu duljinu, funkcija smije primati i druge argumente.*

Na primjer, za skup $\{5, 7, 12\}$*, funkcija treba vratiti* 2*, jer traženi uvjet zadovoljavaju podskupovi* $\emptyset$ *(suma elemenata je nula) i* $\{5, 12\}$ *(suma elemenata je* 17*).*

Obavezno napišite i kako se funkcija poziva!

Rješenje. Ovdje je najjednostavnije isprobati sve mogućnosti (tj. proći po svim podskupovima zadanog skupa i brojati one koji imaju sumu elemenata djeljivu sa 17).

```
int rek(int niz[], int n, int suma) {
  if (n) return
    rek(niz, n-1, suma + niz[n-1]) +
    rek(niz, n-1, suma);
  return (suma % 17 ? 0 : 1);
}
```

Poziv funkcije:

```
int niz[] = {-12, -11, -5, -2, 1, 3, 5, 8};
int duljina_niza = 8;
printf("Takvih podskupova ima %d.\n",
  rek(niz, duljina_niza, 0));
```

Pogledajmo malo kako ova funkcija u stvari radi. Najprije provjeravamo da niz nije prazan (tj. da li je `n` različito od nule). Ako ima elemenata, onda gledamo zadnjeg (`niz[n-1]`): njega možemo staviti u podskup ili ga izostaviti.

- **Linija** 3**:**
  Ako zadnjeg stavimo u podskup, onda ukupna suma raste za vrijednost `niz[n-1]`, te nastavljamo (rekurzivnim pozivom) za niz **bez** zadnjeg elementa (zbog toga u rekurzivnom pozivu prosljeđujemo `n-1`), ali sa sumom uvećanom za `niz[n-1]`.

- **Linija** 4**:**
  Ako zadnjeg **ne** stavimo u podskup, onda ukupna suma ostaje nepromijenjena, te nastavljamo (rekurzivnim pozivom) za niz **bez** zadnjeg elementa (zbog toga i u ovom rekurzivnom pozivu prosljeđujemo `n-1`), s nepromijenjenom sumom.

Nakon što smo izračunali koliko podskupova imamo u oba slučaja, zbrajamo dobivene vrijednosti i rezultat vraćamo kao rezultat funkcije (linija 3).

Ako uvjet u liniji 2 nije bio zadovoljen, to znači da je `n=0`, tj. niz je prazan ("potrošili" smo sve elemente skupa). U varijabli `suma` se nalazi suma elemenata koji su u podskupu. Ako ta suma odgovara uvjetu zadatka (djeljiva je sa 17), brojimo taj podskup kao jedan (pa funkcija vraća 1); u protivnom, taj podskup ne brojimo (tj. vraćamo nulu). □

Napomena 2.3.2. *Primijetite da se nigdje ne čuvaju podskupovi, nego samo suma "pokupljenih" brojeva i oznaka do kojeg broja smo stigli (varijabla* `n`*)!*

Zadatak 2.3.4 (pismeni ispit, 28.11.2005)**.** *Napišite rekurzivnu funkciju koja uzima barem jedan argument $x \in \mathbb{N}$. Funkcija treba vratiti broj na koliko različitih načina se x može prikazati kao suma brojeva* 2*,* 3 *i* 5*, neovisno o redoslijedu sumanada (tj.* $2+3+3$ *je isto što i* $3+2+3$ *i* $3+3+2$*, pa se to broji samo kao jedan način).*

Napišite i program kojim se testira funkcija (treba samo učitati broj, pozvati funkciju i ispisati rezultat).

Napomena: *Uz argument* `x`*, funkcija smije primati dodatne pomoćne argumente, ali nije dozvoljeno korištenje polja, lista, te globalnih i static varijabli!*

Rješenje. Pogledajmo prvo rješenje koje broji sve mogućnosti (i one redundantne):

```
1  #include <stdio.h>
2  int part(int x) {
3    if (x < 0) return 0; // ovako ne ide
4    if (x == 0) return 1; // jedan uspjesan nacin
5    return part(x - 2) + part(x - 3) + part(x - 5);
6  }
7
8  int main(void) {
9    int n;
10
11   printf("Upisite broj n: "); scanf("%d", &n);
12
```

```
13    printf("Takvih rastava ima %d.\n", part(n));
14
15    return 0;
16  }
```

Kao što vidimo, ideja je identična onoj u prethodnom zadatku: isprobavamo sve mogućnosti i pratimo što nam je ostalo. Postavlja se pitanje kako osigurati da se redundantni rastavi ne ponavljaju?

Jednostavan način je primijetiti da svaki rastav među svojim redundantnim verzijama ima točno jedno uzlazno sortirano pojavljivanje. Npr. kod $2 + 3 + 3$ su brojevi poredani uzlazno, dok kod $3 + 2 + 3$ i $3 + 3 + 2$ nisu. Drugim riječima, jednom kad upotrijebimo broj 3, više ne smijemo upotrebljavati broj 2 (u tom rekurzivnom pozivu, te onima koji proizlaze iz njega).

U tu svrhu pratimo koji je prvi (tj. najmanji) broj koji smijemo koristiti. U početku, to je broj 2 (najmanji od svih koje uopće smijemo koristiti), pa je zato drugi parametar u pozivu `rek()` u liniji 20 (u idućoj varijanti rješenja) jednak 2. Jednom kad iskoristimo broj 3, on postaje najmanji (zbog toga je drugi parametar rekurzivnog poziva u liniji 10 jednak 3). Analogno, kad iskoristimo broj 5, on postaje najmanji broj koji smijemo koristiti (pa je drugi parametar rekurzivnog poziva u liniji 11 jednak 5).

Sumu računamo kumulativno, tj. prvo ju inicijaliziramo na nulu, te zatim povećavamo za one brojeve koje smijemo koristiti (to ostvarujemo `if()`–ovima u linijama 9–11).

```
1  #include <stdio.h>
2
3  int part(int x, int prvi) {
4    int cnt = 0;
5    if (x < 0) return 0; // ovako ne ide
6    if (x == 0) return 1; // jedan uspjesan nacin
7    // Kreni od zadnjeg koristenog sumanda (tj. gledamo
8    // samo uzlazne poretke, pa nema ponavljanja):
9    if (prvi <= 2) cnt += part(x - 2, 2);
10   if (prvi <= 3) cnt += part(x - 3, 3);
11   if (prvi <= 5) cnt += part(x - 5, 5);
12   return cnt;
13 }
14 int main(void) {
15   int n;
16
17   printf("Upisite broj n: "); scanf("%d", &n);
18
19   // Prvi sumand za koji testiramo je 2
20   printf("Takvih rastava ima %d.\n", part(n, 2));
21
22   return 0;
23 }
```

□

Slično prethodnom zadatku:

Zadatak 2.3.5. *Napišite rekurzivnu funkciju koja uzima barem jedan argument $x \in \mathbb{N}$. Funkcija treba ispisati sve različite načine na koje se x može prikazati kao suma brojeva 2, 3 i 5, neovisno o redoslijedu sumanada (tj. $2+3+3$ je isto što i $3+2+3$ i $3+3+2$, pa se to broji samo kao jedan način).*

Napišite i program kojim se testira funkcija (treba samo učitati broj i pozvati funkciju).

Napomena: *Uz argument* `x`*, funkcija smije primati dodatne pomoćne argumente, ali nije dozvoljeno korištenje globalnih i static varijabli!*

Uputa. Uvedite još jedan niz (u koji ćete spremati sumande) i pomoćni parametar (u kojem ćete čuvati trenutnu duljinu početnog niza). Prilikom poziva funkcije, pomoćni niz ograničite na 1000 elemenata (ovo ograničenje ćete moći ukloniti pomoću dinamičkih nizova). □

Zadatak 2.3.6. *Napišite rekurzivnu verziju Euklidovog algoritma za računanje najvećeg zajedničkog djelitelja (engl. GCD, greatest common divisor) dva prirodna broja.*

Rješenje.

```
int gcd(int a, int b) {
  if (b == 0)
    return a;
  else
    return gcd(b, a % b);
}
```

ili, kraće

```
int gcd(int a, int b) {
  return (b ? gcd(b, a % b) : a);
}
```

□

Zadatak 2.3.7 (pismeni ispit, 27.6.2005)**.** *Napišite rekurzivnu funkciju koja uzima barem jedan argument $x \in \mathbb{N}$. Funkcija treba vratiti broj na koliko različitih načina se x može prikazati kao umnožak prirodnih brojeva većih od 1. Na primjer, za $x = 12$, funkcija treba vratiti 4 ($2\cdot 2\cdot 3$, $2\cdot 6$, $3\cdot 4$ i 12), za $x = 30$ treba vratiti 5 ($2\cdot 3\cdot 5$, $2\cdot 15$, $3\cdot 10$, $5\cdot 6$, 30), a za $x = 24$ treba vratiti 7 ($2\cdot 2\cdot 2\cdot 3$, $2\cdot 2\cdot 6$, $2\cdot 12$, $2\cdot 3\cdot 4$, $3\cdot 8$, $4\cdot 6$, 24).*

Napišite i program kojim se testira funkcija (treba samo učitati broj, pozvati funkciju i ispisati rezultat).

Napomena: *Uz argument* `x`*, funkcija smije primati dodatne pomoćne argumente, ali nije dozvoljeno korištenje polja, lista, te globalnih i static varijabli!*

Zadatak 2.3.8. *Napišite funkciju koja kao argument prima niz cijelih brojeva duljine $n = 3^k$ za neki $k \in \mathbb{N}$ (ne treba provjeravati da je duljina zaista potencija broja 3), te eventualne pomoćne argumente (koje možete sami odabrati). Funkcija treba srednju*

trećinu niza ispuniti nulama, a rubne jedinicama. Zatim za rubne trećine treba primijeni isti postupak (dok te trećine ne postanu jednočlane). Vrijednosti niza koje treba dobiti za neke vrijednosti n su:

n	*Niz*
3	$1,0,1$
9	$\underbrace{1,0,1}_{3},\underbrace{0,0,0}_{3},\underbrace{1,0,1}_{3}$
27	$\underbrace{1,0,1,0,0,0,1,0,1}_{9},\underbrace{0,0,0,0,0,0,0,0,0}_{9},\underbrace{1,0,1,0,0,0,1,0,1}_{9}$

Napišite i kako se funkcija poziva.

Rješenje.

```
void niz(int x[], int from, int to) {
  int i;
  if (from == to) {
    x[from] = 1;
    return;
  }
  for (i = from; i <= to; i++) x[i] = 0;
  niz(x, from, (2*from+to)/3);
  niz(x, (from+2*to)/3 + 1, to);
}
```

Poziv funkcije (za $n = 3^4 = 81$):

```
int x[81];
niz(x, 0, 80);
```

□

Zadatak 2.3.9. *Napišite program koji testira funkciju iz prethodnog zadatka za vrijednosti* $3^1, 3^2, \cdots, 3^5$.

Rješenje.

```
int main(void) {
  int x[243], i, n;

  n = 3;
  while (n <= 243) {
    niz(x, 0, n - 1);
    printf("%3d: %d", n, x[0]);
    for (i = 1; i < n; i++) printf(" %d", x[i]);
    printf("\n");
    n *= 3;
```

```
11    }
12
13    return 0;
14  }
```

□

Zadatak 2.3.10. *U funkciji iz zadatka 2.3.8 isti se elementi niza nepotrebno mnogo puta postavljaju na nulu. Pokušajte ispraviti taj nedostatak.*

Uputa. Uvedite jednu nerekurzivnu funkciju koja će "pripremiti" niz (postaviti mu sve elemente na vrijednost nula) i zatim pozvati rekurzivnu funkciju koja će postavljati samo jedinice.

Alternativno, moguće je riješiti zadatak tako da se i vrijednost 0 i vrijednost 1 postavljaju samo ako je zadovoljen terminalni uvjet, ali onda je potreban pomoćni parametar (v. uputu za idući zadatak).

No, vjerojatno najjednostavniji način je izmjena `for()`-petlje u liniji 7 funkcije `niz()` na način da postavlja na nulu samo vrijednosti niza `x` na indeksima od `(2*from+to)/3+1` do `(from+2*to)/3`. □

Zadatak 2.3.11. *Riješite zadatak 2.3.8 tako da funkcija ne kreira niz, nego ispisuje tražene elemente na ekran,* **bez upotrebe nizova**.

Uputa. Rekurzija je, u osnovi, slična onoj iz originalnog zadatka. Potreban je pomoćni parametar (vrijednost 0 ili 1) koji označava u kojoj smo trećini, a ispis se smije raditi samo u slučaju da je zadovoljen terminalni uvjet. □

Zadatak 2.3.12 (pismeni ispit, 1.9.2004)**.** *Žaba želi prijeći rijeku skačući preko n listova lopoča. To radi u skokovima po dva ili tri lista prema naprijed ili prema natrag. Povratka na kopno nema (dakle, ne može otići "ispred" prvog lista). Također, ne može skočiti niti iza zadnjeg lista (npr. ne može skočiti s lista $n-1$ na list $n+1$). Skakutanje je gotovo kad žaba dođe na n-ti list.*

Napišite (rekurzivnu) funkciju koja za zadani n (jedan od funkcijskih argumenata) vraća broj načina kojima žaba može izvesti opisano skakanje u najviše 17 koraka.

Treba napisati i kako se funkcija poziva.

2.4 Statičke varijable

Iako nisu direktno povezane s rekurzijama, statičke varijable često ovdje nalaze svoju primjenu. Opisno, riječ je o globalnim varijablama koje se "ne vide izvan funkcije".

Primjer 2.4.1. *Varijabla* `a` *je globalna:*

```
1  #include <stdio.h>
2
3  int a = 0;
```

```
4
5  void f(void) {
6    a++;
7    printf("a = %d\n", a);
8  }
9
10 int main(void) {
11   f();
12   f();
13   return 0;
14 }
```

Ispis:

```
a = 1
a = 2
```

Varijabla `a` *je lokalna:*

```
1  #include <stdio.h>
2
3  void f(void) {
4    int a = 0;
5    a++;
6    printf("a = %d\n", a);
7  }
8
9  int main(void) {
10   f();
11   f();
12   return 0;
13 }
```

Ispis:

```
a = 1
a = 1
```

Varijabla `a` *je lokalna, statička:*

```
1  #include <stdio.h>
2
3  void f(void) {
4    static int a = 0;
5    a++;
6    printf("a = %d\n", a);
7  }
8
9  int main(void) {
```

```
10    f();
11    f();
12    return 0;
13  }
```

Ispis:

```
a = 1
a = 2
```

Varijabla `a` *je lokalna, statička, ali ju pozivamo i u glavnom programu:*

```
1  #include <stdio.h>
2
3  void f(void) {
4    static int a = 0;
5    a++;
6    printf("a = %d\n", a);
7  }
8
9  int main(void) {
10   f();
11   f();
12   printf("a = %d\n", a);
13   return 0;
14 }
```

Ispisa nema, jer se prilikom compiliranja javlja greška poput ove:

```
t.c: In function main:
t.c:12: error: a undeclared (first use in this function)
t.c:12: error: (Each undeclared identifier is reported
t.c:12: error: only once for each function it appears in.)
```

Pokušajmo modificirati program s globalnom varijablom:

```
1  #include <stdio.h>
2
3  int a = 0;
4
5  void f(void) {
6    a++;
7    printf("a = %d\n", a);
8  }
9
10 int main(void) {
11   f();
12   f();
13   printf("a = %d\n", a);
```

```
14   return 0;
15 }
```

Ispis:

```
a = 1
a = 2
a = 2
```

Dakle, statička varijabla (deklarirana dodavanjem ključne riječi `static` *ispred tipa varijable):*

- *ne gubi vrijednost prilikom izlaska iz funkcije (slično globalnim varijablama), ali*
- *nije vidljiva izvan bloka (ovdje je to funkcija* `f()`*) u kojem je deklarirana*

Globalne varijable se općenito smatraju lošima i treba ih izbjegavati, jer u primjeni često dolazi do kolizije imena (upotrijebimo jednu varijablu u dvije funkcije i time u jednoj funkciji "pokvarimo" vrijednost koju je postavila druga funkcija). Statičke varijable omogućuju funkcionalnost globalnih varijabli (ne "zaboravljaju vrijednost"), ali bez opasnosti da se "potuku" varijable iz različitih funkcija.

Pri tome, inicijalizacija varijable (pridruživanje vrijednosti prilikom same deklaracije) kod statičkih varijabli izvodi se samo jednom. Zbog toga se u prethodnom primjeru, u programu sa statičkom varijablom, vrijednost varijable `a` ne vraća na nulu prilikom drugog poziva funkcije.

Vratimo se na rekurzivnu (**izuzetno lošu!**) implementaciju Fibonnacijevih brojeva.

Primjer 2.4.2. *Uvedimo ograničenje:* $n \leq 100000$*. Uz takvo ograničenje, možemo dodati spremanje izračunatih vrijednosti u neki niz (cacheiranje), kako ne bismo više puta računali istu vrijednost. Tako prilikom računanja* F_n *provjeravamo je li* F_n *već izračunat. Ako je, vraćamo tu vrijednost; ako nije, računamo ga, novu vrijednost pospremamo u niz i vraćamo ju kao rezultat funkcije.*

Za početak, niz može biti globalna varijabla:

```
1  long int fib_cache[100001] = {0};
2
3  long int fib(long int n) {
4    if (n <= 1) return n;
5    if (fib_cache[n]) return fib_cache[n];
6    return fib_cache[n] = fib(n-1) + fib(n-2);
7  }
```

Niz `fib_cache` *inicijaliziramo na vrijednosti nula (za sve elemente!) prilikom deklaracije.*

Poziv ovakve funkcije daje rješenje za $n = 100000$ *za* 11 *stotinki, ali se i dalje ruši za* $n = 1000000$ *(naravno, uz prilagodbu maksimalne duljine niza). Rušenje se događa zbog prevelike dubine rekurzije i to nikakvo cacheiranje ne može riješiti.*

Primijetimo da nam varijabla `fib_cache` *nije potrebna izvan funkcije. Dapače, ovakvo rješenje omogućuje da se negdje izvrši, na primjer, naredba* `fib_cache[17] = 19;`*:*

```
int main(void) {
  printf("fib(17) = %ld\n", fib(17));
  printf("fib(19) = %ld\n", fib(19));
  fib_cache[17] = 19;
  printf("fib(17) = %ld\n", fib(17));
  printf("fib(19) = %ld\n", fib(19));
  return 0;
}
```

Nakon takve narebe, rezultati koje vraća ova funkcija ostaju trajno pogrešni:

```
fib(17) = 1597
fib(19) = 4181
fib(17) = 19
fib(19) = 4181
```

Primijetimo da je "pokvarena" samo vrijednost od F_{17}. Na žalost, to nije uvijek slučaj. Na primjer, glavni program

```
int main(void) {
  printf("fib(17) = %ld\n", fib(17));
  fib_cache[17] = 19;
  printf("fib(17) = %ld\n", fib(17));
  printf("fib(19) = %ld\n", fib(19));
  return 0;
}
```

ispisat će

```
fib(17) = 1597
fib(17) = 19
fib(19) = 1025
```

*Kao što vidimo, vrijednost od F_{19} je također pogrešna (*1025 *umjesto* 4181*). Zašto?*

Naravno, nitko neće sam sebi ovako podmetati u kôdu, ali moguće je da do ovakvih problema dođe nenamjerno. Zbog toga je idealno niz `fib_cache` *smjestiti unutar same funkcije. No, on mora zadržavati vrijednost između dva funkcijska poziva, pa ne smije biti obična lokalna varijabla, nego mora biti statička:*

```
long int fib(long int n) {
  static long int fib_cache[100001] = {0};

  if (n <= 1) return n;
  if (fib_cache[n]) return fib_cache[n];
  return fib_cache[n] = fib(n-1) + fib(n-2);
}
```

Isprobajte ovu funkciju i uvjerite se da radi ispravno i (puno) brže od klasičnog rekurzivnog rješenja.

Napomena 2.4.1. *Prethodni primjer pokazuje poboljšanje rekurzivnog algoritma za računanje Fibonaccijevih brojeva. Ipak, to rješenje je još uvijek neusporedivo lošije od nerekurzivnog rješenja jer uvodimo ograničenje na n, a i troši puno više memorije (na 32-bitnom računalu, niz* `fib_cache` *troši gotovo $400kB$ memorije).*

Zadatak 2.4.1. *Napišite rekurzivnu varijantu algoritma za računanje Fibonaccijevih brojeva tako da neposredno prije izlaska iz funkcije program ispisuje poruke poput*
`Povratna vrijednost i-tog poziva je...`
Pozivi ne moraju biti nabrojani po redu. Npr. za program

```
20 int main(void) {
21   printf("fib(3) = %ld\n", fib(3));
22   printf("fib(5) = %ld\n", fib(5));
23   return 0;
24 }
```

ispis bi mogao biti:

```
Povratna vrijednost 3. poziva je 1
Povratna vrijednost 4. poziva je 0
Povratna vrijednost 2. poziva je 1
Povratna vrijednost 5. poziva je 1
Povratna vrijednost 1. poziva je 2
fib(3) = 2
Povratna vrijednost 8. poziva je 2
Povratna vrijednost 9. poziva je 1
Povratna vrijednost 7. poziva je 3
Povratna vrijednost 10. poziva je 2
Povratna vrijednost 6. poziva je 5
fib(5) = 5
```

Napomena 2.4.2. *Prethodni zadatak nije jednostavan! Sitne modifikacije rješenja mogu dovesti do toga da se neki koraci ispisuju više puta i/ili da se neki koraci ne ispišu niti jednom. Pokušajte dobiti da se svaki korak ispiše točno jednom!*

Poglavlje 3

Višedimenzionalna polja

U C-u ne postoji posebna sintaksa za višedimenzionalna polja. Za simuliranje matrica koriste se obični nizovi. Na primjer:

```
int x[10];
```

Ovdje je `x` niz od 10 cijelih brojeva. Slično:

```
int y[10][20];
```

Ovo je ekvivalentno

```
int (y[10])[20];
```

pa je `y` niz od 10 elemenata od kojih je svaki niz od 20 cijelih brojeva. To možemo provjeriti i sljedećim programskim isječkom:

```
  int a[10][20], b[20][10];
  printf("a: %u, a[0]: %u, a[0][0]: %u\n",
    sizeof(a), sizeof(a[0]), sizeof(a[0][0]));
  printf("b: %u, b[0]: %u, b[0][0]: %u\n",
    sizeof(b), sizeof(b[0]), sizeof(b[0][0]));
```

Operator `sizeof()` ćemo kasnije detaljnije obraditi, a ovdje ćemo samo reći da on vraća veličinu tipa ili izraza (ovo oprezno koristiti!) u byteovima. Ispis ovog programskog isječka bit će sličan ovome:

```
a: 800, a[0]: 80, a[0][0]: 4
b: 800, b[0]: 40, b[0][0]: 4
```

Vidimo da je veličina `a[0][0]` jednaka veličini `b[0][0]` i iznosi 4 bytea (što je jedan cijeli broj na 32-bitnim računalima). Ukupno, i varijabla `a` i varijabla `b` zauzimaju 800 bytea (što je $10 \cdot 20$ cijelih brojeva po 4 bytea). No, prvi element niza `a` zauzima 80 byteova (što je 20 `int`-ova), dok prvi element niza `b` zauzima 40 byteova (što je 10 `int`-ova). Dakle, niz `a` sadrži 20-eročlane nizove cijelih brojeva, dok niz `b` sadrži 10-eročlane nizove cijelih brojeva.

Napomena 3.1.3 (Česta greška)**.** *Prilikom deklaracije višedimenzionalnih polja potrebno je zadati sve dimenzije, dakle*

`int x[17][19];`,

a ne

`int x[17][];` *ili* `int x[][19];` *ili* `int x[][];`.

Jednostavnije deklariranje višedimenzionalnih polja možete vidjeti u poglavlju 5.5.

Elementima višedimenzionalnih polja pristupamo navođenjem indeksa – svakog u svojim uglatim zagradama! Na primjer,

```
printf("%d", a[0][1]);
```

ispisuje drugi element (zbog `[1]`) prvog niza (zbog `[0]`) od nizova cijelih brojeva sadržanih u `a`.

Zadatak 3.1.2. *Napišite program koji učitava dva prirodna broja $m, n \leq 10$, te matrice $a, b \in \mathbb{N}^{m \times n}$. Program treba izračunati sumu matrica $c := a + b$ i ispisati ju (tablično; možete pretpostaviti da će svi učitani brojevi imati najviše 5 znamenaka).*

Rješenje. Matrice se zbrajaju po elementima: $c_{ij} = a_{ij} + b_{ij}$.

```
#include <stdio.h>

void ucit_mat(int x[][10], int m, int n, char ime) {
  int i, j;
  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++) {
      printf("%c[%d][%d] = ", ime, i, j);
      scanf("%d", &x[i][j]);
    }
}

int main(void) {
  int a[10][10], b[10][10], c[10][10], m, n, i, j;

  printf("m = "); scanf("%d", &m);
  printf("n = "); scanf("%d", &n);

  ucit_mat(a, m, n, 'a');
  ucit_mat(b, m, n, 'b');

  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
      c[i][j] = a[i][j] + b[i][j];

  printf("Rezultat c =\n");
  for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++)
```

```
28        printf("%6d", c[i][j]);
29      printf("\n");
30    }
31
32    return 0;
33  }
```

Primijetite da kod proslijeđivanja višedimenzionalnih nizova u funkcije, u zaglavlju funkcije moramo navesti njihove dimenzije (sve osim, eventualno, prve), što kod običnih nizova nije bilo nužno (tj. bilo je, ali tamo je prva dimenzija ujedno i jedina, pa je to manje vidljivo). Razlog tome je raspored elemenata u memoriji. □

Ispis je u rješenju prethodnog zadatka mogao biti i ljepši:

```
25  for (i = 0; i < m; i++) {
26    for (j = 0; j < n; j++)
27      printf("%6d", a[i][j]);
28    printf("  %c  ", i == m / 2 ? '+' : ' ');
29    for (j = 0; j < n; j++)
30      printf("%6d", b[i][j]);
31    printf("  %c  ", i == m / 2 ? '=' : ' ');
32    for (j = 0; j < n; j++)
33      printf("%6d", c[i][j]);
34    printf("\n");
35  }
```

Zadatak 3.1.3. *Napišite program koji učitava dva prirodna broja $m, n \leq 10$, te matrice $a, b \in \mathbb{R}^{m \times n}$. Program treba izračunati sumu matrica $c := a + b$ i ispisati ju (tablično; možete pretpostaviti da će svi učitani brojevi imati najviše 7 znakova (znamenke, točke, predznak), a od toga najviše dvije decimalne).*

Zadatak 3.1.4. *Neka je u varijablu* `x` *učitana kvadratna matrica realnih brojeva reda* `n`*. Napišite dio programa koji računa i ispisuje trag te matrice.*

Rješenje. Trag matrice je suma elemenata glavne dijagonale.

```
1  int i;
2  double sum = 0;
3
4  for (i = 0; i < n; i++) sum += a[i][i];
5  printf("tr a = %g\n", sum);
```

Ovo rješenje ima n zbrajanja, što znači linearnu složenost (u ovisnosti o dimenziji matrice n). □

Zadatak je moguće riješiti i na sljedeći način:

```
int i, j;
double sum = 0;

for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    if (i == j) sum += a[i][j];

printf("tr a = %g\n", sum);
```

Iako daje točan rezultat, ovo rješenje nije dobro. Ono ima kvadratnu složenost, a ne postoji nikakav razlog zašto se ne bi koristila ista varijabla na mjestu oba indeksa.

Zadatak 3.1.5. *Napišite program koji učitava $n \in \mathbb{N}$ i kvadratnu matricu reda n, te ispisuje produkt elemenata na njenoj sporednoj dijagonali. Pokušajte postići da program ima linearnu složenost.*

Uputa. Za indekse elemenata a_{ij} na sporednoj dijagonali kvadratne matrice reda n vrijedi $i + j = n - 1$ (ako elemente indeksiramo od nule, kao sto se to radi u C-u). □

Zadatak 3.1.6. *Zadano je trodimenzionalno polje nula i jedinica dimenzije n (reprezentacija diskretizirane kocke u trodimenzionalnom prostoru). Napišite dio programa koji će ispisati koliko ima jedinica*

- *u vrhovima kocke*
- *na bridovima kocke (bez vrhova)*
- *na stranama kocke (bez vrhova i bridova)*
- *u unutrašnjosti kocke (bez vrhova, bridova i strana)*
- *na glavnoj dijagonali kocke*

Uputa. Ako je matrica pohranjena u varijablu `a`, vrhovi kocke se nalaze u
`a[0][0][0]`, `a[0][0][n-1]`, `a[0][n-1][0]`, `a[0][n-1][n-1]`,
`a[n-1][0][0]`, `a[n-1][0][n-1]`, `a[n-1][n-1][0]`,
`a[n-1][n-1][n-1]`
tj.

`a[i][j][k]`, za `i`, `j`, `k` $\in \{0, $`n-1`$\}$.

Dakle, broj jedinica u vrhovima `cnt_vrhovi` možemo dobiti ovako:

```
int i, j, k;
int cnt_vrhovi = 0;

for (i = 0; i < n; i += n-1)
  for (j = 0; j < n; j += n-1)
    for (k = 0; k < n; k += n-1)
      cnt_vrhovi += a[i][j][k];
```

ili

```
int i, j, k;
int cnt_vrhovi = 0;

for (i = 0; i < 2; i++)
  for (j = 0; j < 2; j++)
    for (k = 0; k < 2; k++)
      cnt_vrhovi += a[i*(n-1)][j*(n-1)][k*(n-1)];
```

Analogno za ostale tražene objekte. □

Zadatak 3.1.7. *Riješite prethodni zadatak tako da matrica sadrži* **znakove** *(umjesto cijelih brojeva), a brojati treba zvjezdice.*

Zadatak 3.1.8. *Napišite dio programa koji za učitanu kvadratnu matricu realnih brojeva $a \in \mathbb{R}^{n \times n}$ provjerava je li ona donje trokutasta.*

Kad kažemo da "program nešto provjerava", to znači da treba ispisati da li traženo svojstvo vrijedi.

Rješenje. Matrica a je donje trokutasta ako za sve i, j vrijedi:

$$a_{i,j} \neq 0 \Rightarrow i \geq j,$$

tj. ako joj se svi elementi različiti od nule nalaze na glavnoj dijagonali ili ispod nje.

Zadatak rješavamo tako da pretpostavimo da matrica **je** donje trokutasta, a zatim provjeravamo da li se iznad glavne dijagonale (dakle, za $i < j$) nalaze samo nule.

```
int d_trokut = 1, i, j;

for (i = 0; i < n; i++)
  for (j = i + 1; j < n; j++)
    if (a[i][j]) {
      d_trokut = 0;
      break;
    }

printf("Matrica %sje donje trokutasta.\n",
  d_trokut ? "" : "ni");
```

□

Primjer 3.1.3 (Testiranje rješenja koja traže učitavanja mnogo podataka)**.** *Jednostavan program za testiranje prethodnog rješenja (bez učitavanja matrice), može izgledati ovako:*

```
#include <stdio.h>

int main(void) {
  int a[10][10] = {{1,0,0},{1,1,1},{0,1,1}}, n = 3;
```

```
5   int d_trokut = 1, i, j;
6
7   for (i = 0; i < n; i++)
8     for (j = i + 1; j < n; j++)
9       if (a[i][j]) {
10        d_trokut = 0;
11        break;
12      }
13
14  printf("Matrica %sje donje trokutasta.\n",
15    d_trokut ? "" : "ni");
16
17  return 0;
18 }
```

Program ispituje da li je matrica

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

donje trokutasta (nije). Ako liniju 4 *zamijenimo s*

```
4   int a[10][10] = {{1,0,0},{1,1,0},{0,1,1}}, n = 3;
```

dobijamo matricu

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

koja je donje trokutasta.

Ovakav pristup nema praktičnih vrijednosti za konačna rješenja, jer ovakvi programi rade samo za jedan upisani podatak, ali je jako koristan za testiranje programa na računalu i ispravljanje grešaka, kada je dobro izbjeći upisivanja velikih količina podataka prilikom svakog testiranja programa.

Zadatak 3.1.9. *Napišite dio programa koji za učitanu kvadratnu matricu realnih brojeva $a \in \mathbb{R}^{n\times n}$ provjerava je li ona identiteta, dijagonalna, donje trokutasta, gornje trokutasta ili ni jedno od navedenog.*

Ako je identiteta, ne treba ispisivati da je i dijagonalna; ako je dijagonalna, ne treba ispisivati da je i gornje i donje trokutasta.

Uputa. Matrica a je identiteta ako za sve i, j vrijedi:

$$a_{i,j} = \begin{cases} 1, & i = j, \\ 0, & i \neq j. \end{cases}$$

Matrica a je dijagonalna ako za sve i, j vrijedi:

$$a_{i,j} \neq 0 \Rightarrow i = j.$$

Matrica a je gornje trokutasta ako za sve i, j vrijedi:

$$a_{i,j} \neq 0 \Rightarrow i \leq j,$$

tj. ako joj se svi elementi različiti od nule nalaze na glavnoj dijagonali ili iznad nje. □

Zadatak 3.1.10. *Napišite program koji učitava cijele brojeve $i, j \in \mathbb{N}_0$, $i, j < 10$, te kreira i ispisuje tablicu m s 10×10 znakova koja na svim mjestima ima točkice, osim na horizontalnoj i vertikalnoj liniji koje prolaze elementom $m_{i,j}$ (na te linije treba staviti zvjezdice).*

Na primjer, za $i = 1, j = 2$, tablica treba izgledati ovako:

```
. . * . . . . . . .
* * * * * * * * * *
. . * . . . . . . .
. . * . . . . . . .
. . * . . . . . . .
. . * . . . . . . .
. . * . . . . . . .
. . * . . . . . . .
. . * . . . . . . .
. . * . . . . . . .
```

Rješenje.

```
1  #include <stdio.h>
2
3  int main(void) {
4    char a[10][10];
5    int i, j, k, l;
6
7    printf("i = "); scanf("%d", &i);
8    printf("j = "); scanf("%d", &j);
9
10   for (k = 0; k < 10; k++)
11     for (l = 0; l < 10; l++)
12       a[k][l] = '.';
13
14   for (k = 0; k < 10; k++) {
15     a[i][k] = '*';
16     a[k][j] = '*';
17   }
18
```

```
19    for (k = 0; k < 10; k++) {
20      for (l = 0; l < 10; l++)
21        printf("%2c", a[k][l]);
22      printf("\n");
23    }
24
25    return 0;
26  }
```

□

Zadatak 3.1.11. *Napišite program sličan onom u prethodnom zadatku, ali umjesto horizontalne i vertikalne linije napravite "križ" od linija paralelnih s glavnom i sporednom dijagonalom.*

Zadatak 3.1.12. *Napišite dio programa koji ispisuje koliko se prostih brojeva nalazi u učitanoj matrici* $a \in \mathbb{N}^{m \times n}$.

Rješenje.

```
1  int i, j, k, cnt = 0;
2
3  for (i = 0; i < m; i++)
4    for (j = 0; j < n; j++) {
5      for (k = 2; k < a[i][j]; k++)
6        if (!(a[i][j] % k)) break;
7      if (k == a[i][j]) cnt++;
8    }
9
10 printf("Broj prostih brojeva u matrici: %d\n", cnt);
```

□

Zadatak 3.1.13. *Napišite program koji ispisuje koliko se složenih brojeva nalazi na sporednoj dijagonali učitane matrice* $x \in \mathbb{N}^{n \times n}$.

Razlika između baratanja recima i stupcima nije velika, ali potrebno je dobro pripaziti što se i kako radi.

Zadatak 3.1.14. *Neka je učitana matrica* $x \in \mathbb{R}^{m \times n}$. *Napišite dio programa koji ispisuje indeks* **retka** *s najvećom sumom elemenata. Ako takvih ima više, dovoljno je ispisati indeks jednog od njih.*

Rješenje.

```
1  int i, j, maxi;
2  double max;
3
4  for (i = 0; i < m; i++) {
```

```
5     double sum = 0;
6     for (j = 0; j < n; j++) sum += x[i][j];
7     if (i == 0 || sum > max) {
8       max = sum;
9       maxi = i;
10    }
11  }
12
13  printf("Index retka s najvecom sumom: %d\n", maxi);
```

□

Zadatak 3.1.15. *Neka je učitana matrica $x \in \mathbb{R}^{m \times n}$. Napišite dio programa koji ispisuje indeks* **stupca** *s najvećom sumom elemenata. Ako takvih ima više, dovoljno je ispisati indeks jednog od njih.*

Rješenje.

```
1  int i, j, maxi;
2  double max;
3
4  for (j = 0; j < n; j++) {
5    double sum = 0;
6    for (i = 0; i < m; i++) sum += x[i][j];
7    if (j == 0 || sum > max) {
8      max = sum;
9      maxj = j;
10   }
11 }
12
13 printf("Index retka s najvecom sumom: %d\n", maxi);
```

□

Napomena 3.1.4 (Česta greška)**.** *Česta greška kod programa koji traže stupce je da se zamijeni i poredak petlji i indeksiranje elemenata. Na primjer, ako kod prethodnog zadatka zamijenimo indekse* `i` *i* `j` *u liniji* 5*:*

```
3  for (j = 0; j < n; j++) {
4    int sum = 0;
5    for (i = 0; i < m; i++) sum += x[j][i];
6    if (j == 0 || sum > max) {
7      max = sum;
8      maxj = j;
9    }
10 }
```

dobit ćemo rješenje "po recima". Naime, računalu je svejedno kako se varijable zovu, pa možemo i zamijeniti varijable `i` *i* `j` *u programskom isječku, što će nas dovesti upravo do rješenja zadatka "po recima" (umjesto ovog zadnjeg, "po stupcima").*

Dodatno, indeks retka (ovdje `j`*) se "kreće" od* 0 *do* $n-1$*, što je pogrešno jer je broj redaka* m*, a ne* n*.*

Ako je potrebno ispisati indekse svih redaka/stupaca koji zadovoljavaju neki kriterij, potrebno je prvo izračunati kriterij (ovdje najveća suma elemenata), te još jednom "proći" kroz matricu i ispisati sve indekse koji zadovoljavaju kriterij:

Zadatak 3.1.16. *Neka je učitana matrica* $x \in \mathbb{R}^{m\times n}$*. Napišite dio programa koji ispisuje* **indekse redaka**[1] *s najvećom sumom elemenata.*

rješenje.

```
int i, j, maxi;
double max;

for (i = 0; i < m; i++) {
  double sum = 0;
  for (j = 0; j < n; j++) sum += x[i][j];
  if (i == 0 || sum > max) max = sum;
}

printf("Indexi redaka s najvecom sumom:\n");

for (i = 0; i < m; i++) {
  double sum = 0;
  for (j = 0; j < n; j++) sum += x[i][j];
  if (sum == max) printf("%d\n", i);
}
```

□

Zadatak 3.1.17. *Neka je učitana matrica* $x \in \mathbb{R}^{m\times n}$*. Ispišite* **elemente** *onog stupca koji ima najmanji produkt onih elemenata koji su različiti od nule.*

Zadatak 3.1.18. *Neka je učitana matrica* $x \in \mathbb{Z}^{m\times n}$*. Ispišite indekse onih stupaca koji imaju prostu sumu pozitivnih elemenata.*

Zadatak 3.1.19. *Napišite dio programa koji transponira elemente učitane matrice* $x \in \mathbb{R}^{m\times n}$ *(pri tome je potrebno prilagoditi i dimenzije matrice tako da odgovaraju novonastaloj matrici).*

Napomena: *Nije dozvoljeno korištenje pomoćnih nizova.*

Rješenje. Transponiramo kvadratnu matricu reda $M \times M$, gdje je $M = \max\{m, n\}$. Pri tome ćemo čitati i neinicijalizirane elemente, no oni nisu bitni jer ćemo ih samo premjestiti na mjesta koja se kasnije i tako neće čitati (zbog zamjene varijabli m i n). Uz dodatne uvjete, moguće je napraviti transponiranje bez čitanja neinicijaliziranih elemenata.

[1] Kad piše u množini, to znači da treba ispisati indekse svih redaka/stupaca koji zadovoljavaju kriterij (a ne samo jednog od njih).

```
int max = (m > n ? m : n), tmp;
for (i = 0; i < max; i++) {
  for (j = i + 1; j < max; j++) {
    double tmp = x[i][j];
    x[i][j] = x[j][i];
    x[j][i] = tmp;
  }
}
tmp = m; m = n; n = tmp;
```

Primijetimo da program neće dobro raditi ako matrica nije deklarirana na odgovarajući način (tako da su obje dimenzije veće ili jednake *max*). No, u tom slučaju niti transponiranje nije dobro definirano, tj. ne može se provesti na korektan način. □

Napomena 3.1.5. *U prethodnom rješenju postoje* **dvije različite** *varijable* `tmp`*. Jedna je cjelobrojna i dostupna u cijelom programskom isječku, a druga je realna i dostupna samo u unutrašnjoj petlji programa (gdje se, zbog realne varijable* `tmp`*, ne "vidi" ona cjelobrojna).*

Ovdje je to samo ilustracija mogućnosti koje su spominjane pri početku poglavlju o varijabilnim argumentima funkcija. Inače, dozvoljeno je (čak i poželjno) različitim varijablama davati različita imena.

Zadatak 3.1.20. *Napišite dio programa koji transponira elemente učitane matrice* $x \in \mathbb{R}^{n\times n}$*, te kreira novu matricu* $y \in \mathbb{R}^{n\times n}$ *takvu da je* $y = x^\tau$*. Pretpostavite da je varijabla* `y` *deklarirana.*

Zadatak 3.1.21. *Pretpostavimo da je učitana matrica* $x \in \mathbb{R}^{n\times n}$*. Napišite dio programa koji kreira nove matrice* $a, b \in \mathbb{R}^{n\times n}$ *(pretpostavite da su već deklarirane) takve da je*

$$x = a + b,$$

te da je matrica a *simetrična, a matrica* b *antisimetrična.*

Uputa. Matrica a je simetrična ako vrijedi: $a = a^\tau$, a matrica b je antisimetrična ako je $b = -b^\tau$.

Traženi rastav postiže se formulama:

$$a = \frac{1}{2}(x + x^\tau),$$
$$b = \frac{1}{2}(x - x^\tau).$$

□

Zadatak 3.1.22 (Šlag na kraju)**.** *Napišite tekstualnu verziju poznate igre Minesweeper. Više o igri možete pronaći na*
http://acm.uva.es/problemset/v101/10189.html

Uputa. Potrebno je složiti program koji generira matricu reda $m \times n$ za učitane $m, n \in \mathbb{N}$ (uzmite da je $m, n \leq 20$). Pri tome, matrica treba sadržavati brojeve koji označavaju mine i slobodna polja. Na primjer, -1 može označavati slobodno polje, a -2 minu. Matrica treba sadržavati -2 na točno $k \in \mathbb{N}$ mjesta (k također učitati).

Korisnik upisuje koordinate polja na koje želi "stati". Program provjerava da li se na tom položaju nalazi mina i, ako da, ispisuje matricu ("mine" označiti sa zvjezdicom, a prazna nekliknuta polja s točkicom), te prekida igru uz odgovarajuću poruku (nemojte da bude baš jako okrutna prema igraču).

Ako korisnik nije "stao" na minu, potrebno je pronaći koliko polja oko tog polja sadrži minu, te taj broj upisati u matricu na odabranoj lokaciji. Zatim matricu treba ispisati, na način da se umjesto negativnih brojeva ispisuje neki znak (npr. točkica). Ako je korisnik odabrao posljednje slobodno polje, treba ispisati matricu (kao i u slučaju poraza), pohvaliti igrača i prekinuti izvršavanje programa.

Dodatna komplikacija: ako na susjednim poljima nema mina, upisati nulu i automatski otvoriti sva susjedna polja. Za ovo je idealno upotrijebiti rekurziju, iako je moguće i bez nje.

Slučajne nenegativne cijele brojeve vraća funkcija `rand()` (između `0` i `RAND_MAX`, gdje je `RAND_MAX` konstanta deklarirana u `stdlib.h`), a nalazi se u biblioteci `stdlib`. Slučajni broj `x` od `0` do `n-1` (uključivo) možemo dobiti izrazom

```
x = rand() % n;
```

Prije **prvog** poziva funkcije `rand()` u programu, potrebno je izvršiti

```
srand(time(0));
```

□

Poglavlje 4

Dinamičke varijable

U praksi, često ne znamo unaprijed (dok pišemo program) koje su gornje ograde dimenzija ulaznih nizova ili matrica. Tada ne možemo koristiti statički alocirana jednodimenzionalna (Programiranje 1) i višedimenzionalna polja (poglavlje 3), nego je takve strukture potrebno dinamički alocirati.

Dinamičku alokaciju koristit ćemo i kasnije, sa stringovima i vezanim listama, što će biti važno u kasnijim kolegijima.

U ovom poglavlju proći ćemo kroz osnovne pojmove i alokaciju pojedinih varijabli, a zatim i kroz dinamičku alokaciju nizova i razne načine dinamičke alokacije višedimenzionalnih polja. Na kraju obrađujemo realokaciju memorije, koja se koristi u slučaju da niti ne znamo veličinu potrebnog polja prije nego ga počnemo koristiti, nego je količinu alocirane memorije potrebno naknadno mijenjati.

4.1 Uvod

Što će ispisati sljedeći dio koda?

```
1 int x;
2 printf("%d\n", x);
```

Memorijske lokacije uvijek imaju neku vrijednost. Kad deklariramo varijablu, njoj se dodjeljuje lokacija u memoriji, te varijabla "poprima" vrijednost zatečenu na toj lokaciji. No, ta vrijednost je nastala bez kontrole korisnika (ostala od prijašnje varijable nekog programa ili nekako slično, gotovo slučajno). Zbog toga, gornji dio koda će ispisati neki "slučajni" cijeli broj.

Na sličan način, ako deklariramo pokazivač na cijeli broj

```
1 int *x;
```

njegova vrijednost će biti slučajna. Pri tome, nužno je razlikovati **pokazivač** i **sadržaj ćelije na koju on pokazuje**.

Ovdje smo deklarirali varijablu `x` i ona, kao i maloprije, ima slučajnu vrijednost. No, ta varijabla je tipa `int*` (pointer na nešto tipa `int`), pa se njena vrijednost interpretira kao adresa memorijske lokacije u kojoj se nalazi cijeli broj.

Ako pokušamo pristupiti toj memorijskoj lokaciji, npr.

```
1 *x = 17;
```

time efektivno pristupamo memorijskoj lokaciji čiju adresu sadrži varijabla `x`. No, ta adresa uopće ne mora postojati, pa će se program srušiti!

Ako nekim čudom adresa i postoji (tj. u `x` se slučajno zatekla "legalna" adresa), ona pripada nekoj varijabli, no mi ne možemo znati kojoj. Da stvar bude gora, ta varijabla (ona na koju pokazuje `x`) gotovo sigurno ne pripada našem programu, nego nekom drugom programu ili čak samom operacijskom sustavu (iako neki sustavi sprječavaju ovakvo ponašanje, što onda opet dovodi do rušenja našeg programa).

Zbog toga je izraz `*x = 17` uvijek problematičan: ili sruši program ili dovodi do nekontrolirane promjene neke varijable koju ne smijemo mijenjati.

Znači, da bismo koristili varijablu `x`, ona mora sadržavati adresu memorijske lokacije po kojoj smijemo pisati. Jedan način kako to postići vidjeli smo u poglavlju o varijablinim argumentima funkcija, a svodi se na:

```
1 int broj;
2 int *pokazivac;
3 pokazivac = &broj;
4 *pokazivac = 17;
```

Dakle, varijabla `pokazivac` sadrži adresu varijable `broj` (zbog linije 3), što znači da izraz `*pokazivac = 17` radi isto što bi radio i izraz `broj = 17`.

S obzirom da ti izrazi rade istu stvar, postavlja se logično pitanje: čemu nam služe pointeri (osim za varijabilne parametre funkcija)?

Odgovor leži u činjenici da je moguće zauzeti dio memorije "u hodu", tj. dok se program izvršava (bez deklariranja posebne varijable). **Opisno** (dakle, ovo nije C–kod!), to izgleda ovako:

```
pokazivac = alociraj_memoriju_za_int();
*pokazivac = 17;
```

Imamo funkciju (ovdje smo ju nazvali `alociraj_memoriju_za_int()`) koja računalu kaže da treba prostor za jedan cijeli broj. Računalo nađe takav prostor (ako postoji dovoljno veliki blok slobodne memorije), dodijeli ga programu te kao povratnu vrijednost funkcije `alociraj_memoriju_za_int()` vrati adresu alociranog prostora. Mi tu adresu pohranjujemo u varijablu `pokazivac` te pomoću nje možemo pristupati alociranom dijelu memorije.

Ova skica je dobra, ali sintaktički nije ispravna. Naime, različiti tipovi podataka zauzimaju različite količine memorije. Umjesto definiranja alokacijskih funkcija za sve moguće tipove podataka (što je nemoguće, jer programer može definirati i svoje tipove podataka), u C-u postoje općenite funkcije za alokaciju memorije. Mi ćemo koristiti funkciju `malloc()`.

Funkcija `malloc()` prima jedan argument: količinu potrebne memorije. Znači, ako želimo alocirati memoriju za jedan cijeli broj, moramo pozvati funkciju `malloc()` i reći joj da želimo onoliko memorije koliko zauzima cijeli broj. Ta količina se, na žalost, razlikuje od računala do računala, pa nije dovoljno napisati konkretan broj, nego je potrebno

"saznati" tu veličinu. U tu svrhu koristimo operator (iako izgleda i poziva se kao funkcija) `sizeof()` kojem, kao parametar, zadajemo tip (ili izraz, ali tu treba biti oprezan!) čija veličina nas zanima.

Primjer 4.1.1 (Veličine nekih varijabli)**.**

```
#include <stdio.h>

int main(void) {
  printf("Velicina int: %u\n", sizeof(int));
  printf("Velicina pokazivaca na int: %u\n",
    sizeof(int*));
  printf("Velicina double: %u\n", sizeof(double));
  printf("Velicina pokazivaca na double: %u\n",
    sizeof(double*));
  printf("Velicina char: %u\n", sizeof(char));
  printf("Velicina pokazivaca na char: %u\n",
    sizeof(char*));
  return 0;
}
```

Ovaj program će ispisati nešto poput:

```
Velicina int: 4
Velicina pokazivaca na int: 4
Velicina double: 8
Velicina pokazivaca na double: 4
Velicina char: 1
Velicina pokazivaca na char: 4
```

Konkretni brojevi se mogu razlikovati od računala do računala. Ovdje prikazani ispis potječe s već spomenutog Pentiuma 4 (32-bitno računalo). Na 64-bitnom računalu (Athlon64), ispis izgleda ovako:

```
Velicina int: 4
Velicina pokazivaca na int: 8
Velicina double: 8
Velicina pokazivaca na double: 8
Velicina char: 1
Velicina pokazivaca na char: 8
```

Možete računati da `sizeof()` *vraća veličinu tipa (ili izraza) u byteovima*[1]*.*

Primijetimo da se razlike veličina osnovnih tipova razlikuju. Na primjer, na 32-bitnom računalu, `char` *zauzima* 1 *byte,* `int` *zauzima* 4, `double` 8,*... Ti brojevi mogu biti i drugačiji (npr. na 64-bitnom računalu), ali – bez obzira na arhitekturu računala – veličina*

[1] `sizeof()` zapravo vraća broj `char`–ova koji stanu u jednu varijablu traženog tipa, no veličina `char` obično odgovara jednom byteu. Istu mjernu jedinicu ("jedan `char`") koriste i ostale C-ovske funkcije/operatori za baratanje memorijom.

pokazivača je konstantna, neovisno o tome na koji tip podatka pokazivač pokazuje. To je i logično, jer pokazivač sadrži adresu memorijske lokacije koja, naravno, ima fiksnu veličinu, neovisnu o tome što se na toj adresi nalazi.

Upravo ta veličina pointera označava "bitnost" u arhitekturi. Zbog toga adresa na 32*-bitnom računalu zauzima* 4 *bytea (što je* $4 \cdot 8 = 32$ *bita), dok na* 64*-bitnom računalu adrese zauzimaju po* 8 *byteova (što je* $8 \cdot 8 = 64$ *bita).*

Uz sve opisano, sada možemo alocirati jednu varijablu i baratati s njom:

Primjer 4.1.2 (Alokacija i dealokacija memorije)**.**

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void) {
5    int *pokazivac;
6    pokazivac = (int*)malloc(sizeof(int));
7    *pokazivac = 17;
8    printf("%d\n", *pokazivac);
9    (*pokazivac)++;
10   printf("%d\n", *pokazivac);
11   free(pokazivac);
12   return 0;
13 }
```

Analizirajmo program:

Linija 2 *Funkcija* `malloc()` *nalazi se u biblioteci* `stdlib`*, pa je njenu upotrebu potrebno najaviti:* `#include <stdlib.h>`*.*
Kod nestandardnih compilera, moguće je da funkcija `malloc()` *bude definirana u biblioteci* `malloc`*. U tom slucaju, compiler će prijaviti grešku (da ne prepoznaje funkciju* `malloc()`*), pa onda treba dodati i* `#include <malloc.h>`

Linija 5 *Pointer deklariramo kao i "obične" varijable, uz dodatak zvjezdice ("*`*`*"). Time zapravo kažemo: "deklariraj varijablu* `x` *na način da* `*x` *bude tipa* `int`*", što onda implicitno povlači da je varijabla* `x` *pokazivač na nešto tipa* `int`*.*

Linija 6 **Alokacija memorije***:*
Alokaciju vršimo pozivom funkcije `malloc()` *kojoj smo rekli koliko nam memorije treba (*`sizeof(int)` *byteova; dakle, onoliko memorije koliko treba za jedan* `int`*). Funkcija zauzima memoriju i vraća njenu adresu.*

Povratni tip je samo "memorijska adresa", no ne i "adresa čega". Pošto je varijabla `pokazivac` *"adresa* `int`*-a", programu treba reći da tu "općenitu" adresu "pretvori" u adresu* `int`*-a (dakle, u* `int*`*). Prilikom te pretvorbe se, u stvari, ništa ne događa i ona služi samo zato da compiler može ispravno prevesti program.*

Na kraju, "prilagođena" adresa upravo alocirane memorije pohranjuje se u varijablu `pokazivac` *(***NE** *u* `*pokazivac`*!).*

Linije 7–10 *Izraz* `*pokazivac` *koristimo kao da je obična varijabla tipa* `int`*: na jednak način mijenjamo tu vrijednost i čitamo ju.*

Linija 11 *Kad nam memorija dodijeljena u liniji 5 više nije potrebna, potrebno je računalu reći da ju može "uzeti natrag". To radi funkcija* `free()`*. Time ta memorija prestaje pripadati programu, te je može zauzeti tko god želi.*

Napomena 4.1.1. *U prethodnom primjeru, nakon linije* 11 *u varijabli* `pokazivac` *će ostati adresa memorije na koju je pokazivala tijekom izvođenja programa i kojoj smo pristupali u linijama* 7–10 *(jer poziv* `free(pokazivac)` *ne mijenja niti vrijednost varijable* `pokazivac` *niti memorije na koju on pokazuje – on samo označi tu memoriju kao slobodnu), ali toj memoriji* **više ne smijemo pristupati***, jer ne znamo da li ju je (i kad) počeo koristiti neki drugi program (ili sam operacijski sustav).*

Ako želimo ponovno koristiti `*pokazivac`*, potrebno je ponovno alocirati memoriju (u osnovi, ponoviti poziv iz linije* 6*).*

Zadatak 4.1.1. *Napišite dio programa koji učitava* 2 *cijela broja, te ispisuje koji je veći. Jedna varijabla neka bude tipa* `int`*, a druga pointer na* `int`*.*

Rješenje.

```
1 int *x, y;
2 x = (int*)malloc(sizeof(int));
3 scanf("%d", x);
4 scanf("%d", &y);
5 printf("Veci je %d\n", *x > y ? *x : y);
6 free(x);
```

□

Napomena 4.1.2. *Primijetimo da u pozivu* `scanf()` *u liniji* 4 *nema operatora* `&` *ispred varijable* `x`*, dok ga ima u liniji* 5 *ispred varijable* `y`*. To je zbog toga što funkcija* `scanf()` *prima memorijske adrese na koje treba pospremiti učitane vrijednost.*

Ovdje, varijabla `x` *sadrži upravo memorijsku adresu na koju želimo pospremiti vrijednost, dok varijabla* `y` **JE** *memorijska lokacija na koju želimo pospremiti vrijednost, pa je potrebno funkciji proslijediti njenu adresu (tj.* `&y`*).*

Ova razlika je izuzetno bitna za razumijevanje pointera u C-u.

Zadatak 4.1.2. *Napišite program koji učitava* 7 *cijelih brojeva te ispisuje sumu najmanjeg i najvećeg među njima. Pri tome smijete koristiti najviše* 6 *varijabli tipa* `int`*, dok ostale moraju biti pointeri na* `int`*!*

Zadatak 4.1.3. *Riješite prethodni zadatak s realnim brojevima (umjesto cijelih).*

4.2 Dinamička jednodimenzionalna polja

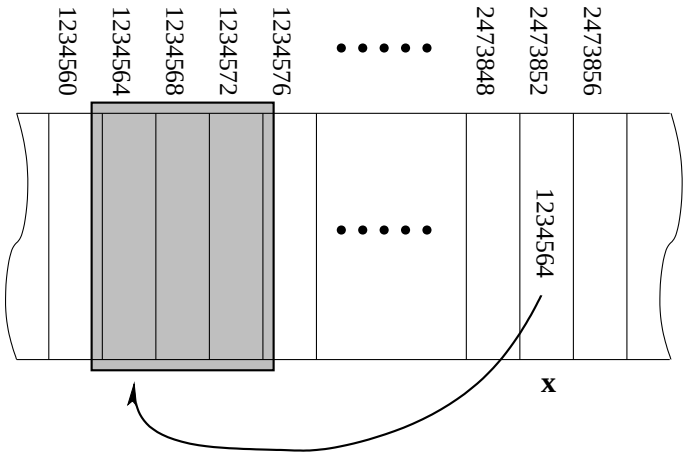
Vidjeli smo kako se može alocirati memorija za jednu varijablu. No, što bi se dogodilo kad bismo alocirali prostor nekoliko puta veći od potrebnog?

Kao prvo, funkcija `malloc()` uvijek vraća adresu alocirane memorije. Ako memorija sadrži više od jedne riječi[2], funkcija vraća adresu prve riječi. Tako ćemo pozivom `malloc(sizeof(double))` alocirati 8 byteova, što su na 32-bitnoj arhitekturi dvije riječi (jedna riječ je 32 bita, tj. 4 bytea). Funkcijski poziv će vratiti adresu prve od dvije riječi.

Promotrimo sljedeći dio koda:

```
1 int *x;
2 x = (int*)malloc(3 * sizeof(int));
```

Dobiveno stanje memorije prikazano je na slici 4.1, gdje jedna ćelija predstavlja prostor za 1 `int` (dakle, 4 bytea na prije spomenutim računalima).



Slika 4.1: Alokacija memorije za 3 cijela broja

Funkcija `malloc()` se ponaša točno kako smo prije opisali:

1. Alocira onoliko bytea koliko kaže funkcijski parametar (ovdje je to `3*sizeof(int)`)
2. Vraća adresu alocirane memorije (preciznije: prve ćelije alocirane memorije)

Nakon pospremanja vraćene adrese u varijablu `x`, ona pokazuje na tu prvu ćeliju.

Primijetimo da se alocirane ćelije nalaze točno jedna iza druge – **baš kao u nizu!** Prvoj ćeliji (onoj na koju pokazuje `x`) znamo pristupiti: pomoću izraza `*x`. Kako pristupiti, na primjer, drugoj ćeliji?

Na slici smo prvoj ćeliji dodijelili adresu 1234564, pa iduća ima adresu 1234568, što je točno za 4 (= `1` · `sizeof(int)`) više od adrese prve ćelije. No, adresa prve ćelije je pohranjena u varijabli `x`. Ako uvrstimo tu vrijednost u izraz `x+1`, dobit ćemo da je

[2]Podsjetnik: riječ je najmanja jedinica memorije koju računalo može alocirati

`x + 1 = 1234564 + sizeof(*x) = 1234568`,

što je adresa druge ćelije. Dakle, izraz

`*(x + 1)`

zapravo označava drugu ćeliju!

Napomena 4.2.1 (Pointerska aritmetika)**.** *Pointeri se u C-u mogu upotrebljavati točno kako je opisano! Pri tome, sam compiler pazi na taj detalj upravo pomoću tipova podataka.*

```
1  int *x;
2  double *y
3  x = (int*)malloc(3 * sizeof(int));
4  y = (int*)malloc(3 * sizeof(int));
5  *(x+1) = 17;
6  *(y+1) = 17.19;
```

Compiler zna da je `x` *pokazivač na* `int`*, pa će operator* `+` *u liniji* 5 *povećati adresu od* `x` *za točno* `sizeof(int)` *byteova (dakle, za* 4 *bytea). S druge strane,* `y` *je pokazivač na* `double`*, pa će operator* `+` *u liniji* 6 *povećati adresu od* `y` *za točno* `sizeof(double)` *byteova, što je* 8 *bytea!*

Upravo zbog te "svijesti" compilera o pointerskim tipovima, ovakav pristup je izuzetno praktičan, ali treba paziti da se tipovi ispravno pridjeljuju. To znači da nije preporučljivo u varijablu tipa `int*` *pospremiti adresu neke varijable tipa* `double`*, iako će to sintaktički "proći" (jer sve adrese zauzimaju jednako mnogo prostora).*

Za ilustriraciju prethodne napomene, pogledajmo sljedeći primjer:

Primjer 4.2.1.

```
1  int *x;
2  double *y;
3  x = (int*)malloc(3 * sizeof(int));
4  y = (double*)malloc(3 * sizeof(double));
5  printf("x = %p; x+1 = %p; sizeof(int) = %u\n",
6    x, x + 1, sizeof(int));
7  printf("y = %p; y+1 = %p; sizeof(double) = %u\n",
8    y, y + 1, sizeof(double));
9  free(x);
10 free(y);
```

Ispis ovog dijela koda (na spomenutom Pentiumu 4) bit će:

```
x = 0x1963010; x+1 = 0x1963014; sizeof(int) = 4
y = 0x1963030; y+1 = 0x1963038; sizeof(double) = 8
```

Kao što vidimo, za svaku pointersku varijablu `a` *tipa* `A`*, razlika između* `a` *i* `a+1` *je točno* `sizeof(A)` *i zbog toga možemo ovako pristupati pojedinim ćelijama.*

Napomena 4.2.2 (Ekvivalentnost pointera i nizova)**. VAŽNO!** *U C-u, za pointersku varijablu* `p` *i cijeli broj* `i`*, vrijedi sljedeća ekvivalencija:*

$$\texttt{*(p + i)} \Leftrightarrow \texttt{p[i]}$$

Zbog toga dinamički alociranom nizu možemo pristupati na identičan način na koji smo pristupali statičkim nizovima!

Jedina razlika dinamičkih i statičkih nizova je u tome što dinamičkim nizovima prije upotrebe **moramo** *alocirati memoriju, dok statičkima ne možemo. Zato su statički nizovi fiksne duljine i ona se kroz program ne može mijenjati, dok kod dinamičkih može.*

Napomena 4.2.3 (Česta greška)**.** *Kod višedimenzionalnih polja treba biti oprezan:*

$$\texttt{*(p + i + j)} \not\Leftrightarrow \texttt{p[i][j]} \quad i \quad \texttt{**(p + i + j)} \not\Leftrightarrow \texttt{p[i][j]}$$

Ispravno je:

$$\texttt{*(*(p + i) + j)} \Leftrightarrow \texttt{p[i][j]}$$

Zadatak 4.2.1. *Napišite program koji učitava broj $n \in \mathbb{N}_0$, te n realnih brojeva. Brojeve treba ispisati unatrag.*

Rješenje. Ovako zadan zadatak ne postavlja nikakve ograde za maksimalnu duljinu niza, pa zbog toga brojeve ne možemo učitati u statički niz! Pribjegavamo dinamičkoj alokaciji.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void) {
5    double *x;
6    int n, i;
7
8    printf("Upisite n: ");
9    scanf("%d", &n);
10
11   if (n > 0) {
12     x = (double*)malloc(n * sizeof(double));
13
14     if (x == NULL) {
15       printf("Greska: Nema dovoljno memorije!\n");
16       exit(1);
17     }
18
19     for (i = 0; i < n; i++) {
20       printf("Upisite x[%d]: ", i);
21       scanf("%lg", &x[i]);
22     }
23
```

```
24      printf("Brojevi unatrag: %g", x[n-1]);
25      for (i = n-2; i >= 0; i--)
26        printf(", %g", x[i]);
27      printf("\n");
28
29      free(x);
30    }
31
32    return 0;
33  }
```

ili

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main(void) {
5     double *x;
6     int n, i;
7
8     printf("Upisite n: ");
9     scanf("%d", &n);
10
11    if (n > 0) {
12      x = (double*)malloc(n * sizeof(double));
13
14      if (x == NULL) {
15        printf("Greska: Nema dovoljno memorije!\n");
16        exit(1);
17      }
18
19      for (i = 0; i < n; i++) {
20        printf("Upisite x[%d]: ", i);
21        scanf("%lg", x + i);
22      }
23
24      printf("Brojevi unatrag: %g", *(x + n-1));
25      for (i = n-2; i >= 0; i--)
26        printf(", %g", *(x + i));
27      printf("\n");
28
29      free(x);
30    }
31
32    return 0;
33  }
```

Analizirajmo program:

Linije 5 **i** 6 Deklaracije potrebnih varijabli. Potreban nam je niz realnih brojeva, ali ne znamo koje duljine, pa moramo deklarirati pointer.
Deklaracija `double x[]` **NIJE ISPRAVNA**, osim u argumentima funkcija!

Linije 12–14 Alokacija memorije za niz. Ukoliko `malloc()` ne uspije alocirati potrebnu memoriju (npr. vrijednost učitana u `n` je prevelika), vratit će posebnu adresu 0 (tzv. null–pointer). U tom slučaju program treba javiti grešku i prekinuti s izvođenjem jer upotreba nealociranog niza, naravno, dovodi do rušenja programa.

Linija 16 Prekidanje izvršavanja programa. Funkcija `exit()` definirana je u biblioteci `stdlib`, a prima jedan argument koji označava povratnu vrijednost programa. Njena uloga je jednaka ulozi `return` na kraju funkcije `main()`, ali ju je moguće pozvati bilo gdje (a ne samo u `main()`-u).

Linije 19–22 Učitavanje elemenata niza. Primijetite razliku između dvije verzije rješenja u `scanf()`-u u liniji 22. Kad nam treba adresa nekog elementa niza, praktičnije je upotrijebiti sintaksu `x+i` (iako je i `&x[i]` apsolutno ispravno, ali pazite da ne izostavite `&`).

Linije 24–27 Ispis niza (unatrag). Posljednji element niza ispisujemo odvojeno jednostavno zato da na kraju ispisa ne ostane jedan "usamljeni" zarez. Ispis je uredno mogao biti:

```
25      printf("Brojevi unatrag:\n");
26      for (i = n-1; i >= 0; i--)
27        printf("%g\n", x[i]);
```

I u ispisu možemo vidjeti razliku između dvije verzije programa. U ovom slučaju, praktičnije je pisati `x[i]` nego `*(x+i)` (iako je, opet, oboje točno).

Linija 29 Oslobađanje memorije. Ovo se ne smije zaboraviti, iako moderni operacijski sustavi u pravilu to sami obave kad program završi s izvršavanjem.

□

Zadatak 4.2.2. *Napišite program koji učitava broj $n \in \mathbb{N}$, te dva niza a i b realnih brojeva duljine n. Program ispisati nizove na sljedeći način:*

$$a_0, b_0, a_1, b_1, a_2, b_2, \ldots, a_{n-1}, b_{n-1}.$$

Za ispis nizova definirajte funkciju koja će kao argumente primati nizove i duljinu n.

Rješenje. Ovdje imamo alokacije dva niza, pa je elegantnije napisati funkciju koja će brinuti o alokaciji i prijavi greške u slučaju nedostatka memorije.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
```

```
4  double *alociraj_i_ucitaj(int n, char ime) {
5    double *x;
6    int i;
7
8    x = (double*)malloc(n * sizeof(double));
9
10   if (x == NULL) {
11     printf("Greska: Nema dovoljno memorije!\n");
12     exit(1);
13   }
14
15   for (i = 0; i < n; i++) {
16     printf("Upisite %c[%d]: ", ime, i);
17     scanf("%lg", &x[i]);
18   }
19
20   return x;
21 }
22
23 void ispis(double *a, double *b, int n) {
24   int i;
25
26   printf("Ispis: %g, %g", a[0], b[0]);
27   for (i = 1; i < n; i++)
28     printf(", %g, %g", a[i], b[i]);
29   printf("\n");
30 }
31
32 int main(void) {
33   double *a, *b;
34   int n;
35
36   printf("Upisite n: ");
37   scanf("%d", &n);
38
39   if (n > 0) {
40     a = alociraj_i_ucitaj(n, 'a');
41     b = alociraj_i_ucitaj(n, 'b');
42     ispis(a, b, n);
43     free(a);
44     free(b);
45   }
46
47   return 0;
48 }
```

□

U funkciji `ispis()` u prethodnom zadatku, prvi `printf()` smo mogli napisati i na sljedeći način:

```
27   printf("Ispis: %g, %g", *a, *b);
```

Zaglavlje same funkcije mogli smo napisati i ovako:

```
24 void ispis(double a[], double b[], int n) {
```

ili ovako:

```
24 void ispis(double *a, double b[], int n) {
```

ili ovako:

```
24 void ispis(double a[], double *b, int n) {
```

Potpuno je svejedno!

Zadatak 4.2.3. *Napišite dio programa koji učitava brojeve $m, n \in \mathbb{N}$, te dva niza realnih brojeva, prvi duljine m, a drugi duljine n. Program treba uzlazno sortirati i ispisati onaj niz koji ima veću sumu elemenata.*

Zadatak 4.2.4. *Napišite dio programa koji učitava broj $n \in \mathbb{N}$, te niz $(a_i)_{i=0}^n$. Program treba ispisati sve vrijednosti polinoma*

$$p(x) = \sum_{i=0}^{n} a_i \cdot x^i$$

za $x = a_0, a_1, \ldots, a_n$. Računanje vrijednosti polinoma izvedite preko Hornerovog algoritma i implementirajte kao funkciju.

Zadatak 4.2.5. *Napišite program koji učitava broj $n \in \mathbb{N}$, te niz x sa $3n$ cijelih brojeva. Program ispisati niz na sljedeći način:*

$$x_0, x_3, x_6, \ldots, x_{3n-3}, x_1, x_4, \ldots, x_{3n-2}, x_2, x_5, \ldots, x_{3n-1}.$$

Za ispis nizova definirajte funkciju koja će kao argumente primati nizove i duljinu n.

Rješenje. Napisat ćemo samo funkciju za ispis (ostatak ide po uzoru na zadatak 4.2.2). Pazite na razliku u tipu elemenata niza (ovdje su to cijeli brojevi, a u zadatku 4.2.2 su realni)!

```
1  void ispis(int *x, int n) {
2    int i, j;
3
4    for (i = 0; i < 3; i++)
5      for (j = i; j < 3 * n; j += 3)
6        printf("%d\n", x[j]);
7  }
```

Primijetimo da će ova funkcija ispisati brojeve jedan ispod drugog. Ako želimo ispis pomoću zareza (kako se u zadatku i traži), moramo prvoga ispisati posebno, ali onda i pripaziti da se on ne ispiše ponovno i u petlji. To možemo na više načina, npr. ovako:

```
void ispis(int *x, int n) {
  int i, j;

  printf("%d", x[0]);
  for (i = 0; i < 3; i++)
    for (j = i; j < 3 * n; j += 3)
      if (j) printf(", %d", x[j]);
  printf("\n");
}
```

ili ovako:

```
void ispis(int *x, int n) {
  int i, j;

  printf("%d", x[0]);
  for (i = 0; i < 3; i++)
    for (j = (i ? 3 : i); j < 3 * n; j += 3)
      printf(", %d", x[j]);
  printf("\n");
}
```

□

Zadatak 4.2.6. *Napišite program koji učitava $n \in \mathbb{N}$, te niz a od n cijelih brojeva. Program zatim treba učitati niz b od $\sum_i a_i$ brojeva, te ispisati sve proste elemente tog niza.*

Zadatak 4.2.7. *Napišite program koji učitava broj $x \in \mathbb{N}$, te ispisuje njegove znamenke u silazno sortiranom redoslijedu. Smijete koristiti jedan pomoćni niz, ali za njega morate alocirati točno onoliko memorije koliko je nužno.*

Zadatak 4.2.8. *Riješite zadatak 2.3.5 bez postavljanja ograničenja na pomoćni niz.*

Uputa. Najveći broj sumanada postižemo ako su svi oni najmanji mogući. U našem slučaju, najmanji sumand je 2, pa pomoćni niz treba imati $\lceil n/2 \rceil$ elemenata (u praksi, najpraktičnije je alocirati niz od `n/2+1` elemenata).

Moguće je napraviti i s `n/2` elemenata, ali treba paziti da suma ne postane strogo veća od početnog broja (npr. da za broj 5 ne kreiramo sumu $2+2+2$ jer ona ima 3 sumanda, a `5/2` je jednako 2). □

4.3 Dinamička višedimenzionalna polja

Naravno, moguće je dinamički alocirati memoriju i za višedimenzionalna polja. Podsjetimo se i da nizu pristupamo i njime manipuliramo pomoću pointera na prvi element. No,

kod višedimenzionalnih polja se bitno **razlikuje** da li polje samo koristimo preko pointera ili zapravo spremamo neke pointere, iako elementima i dalje pristupamo na jednak način.

Pogledajmo na koji način možemo u memoriju spremiti dvodimenzionalno raspoređene podatke (matricu, niz stringova (svaki string je niz znakova, a obrađujemo ih u poglavlju 5) i sl). Kao osnovni tip pojedine ćelije koristit ćemo `char` (kojeg, naravno, lako možemo zamijeniti s `int`, `double` ili bilo kojim drugim tipom).

1. **Automatsko dvodimenzionalno polje**: `char x[17][19];`

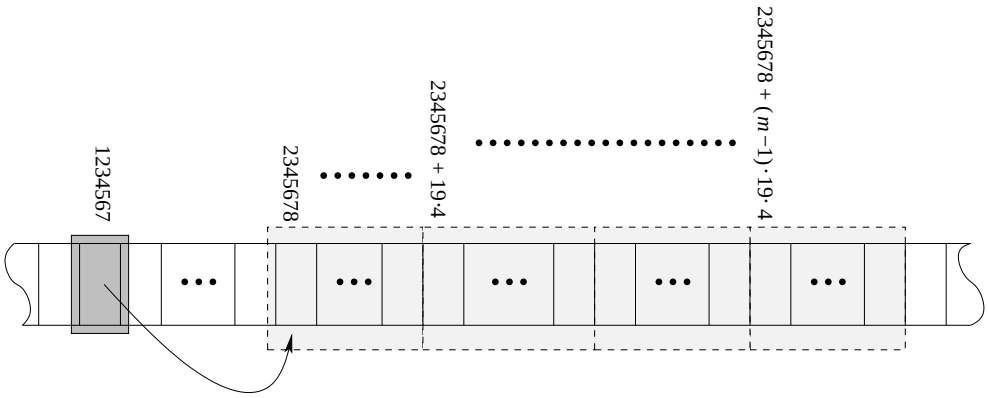
   Varijabla `x` je dvodimenzionalno polje (matrica sa 17×19 znakova) poput onih koja smo obradili u poglavlju 3. Memorija je automatski rezervirana i na kraju programa se automatski oslobađa, no maksimalne dimenzije polja su ograničene (17 redaka i 19 stupaca) te se ne mogu mijenjati (osim ponovnim prevođenjem i pokretanjem programa).

   Deklaracija matrice 17×19 znakova:

   ```
   char x[17][19];
   ```

2. **(Dinamički alocirano) 2D polje**: `char (*x)[19];`

   Varijabla `x` je pointer na blok memorije koji sadrži (jednog za drugim!) nizove od po 19 znakova, što u memoriji izgleda poput slike 4.2. Podaci nisu rasuti (nalaze se u jednom bloku, do na sam pointer koji je negdje odvojeno), ali je duljina "malih nizova" ograničena na najviše 19 znakova (svaki). Za razliku od njihove duljine, broj tih malih nizova nije ograničen (do na raspoloživu memoriju). Memoriju za cijelu strukturu treba alocirati prije upotrebe te ju na kraju osloboditi.



Slika 4.2: Dinamički alocirano dvodimenzionalno polje

Deklaracija i alokacija matrice $m \times 19$ znakova:

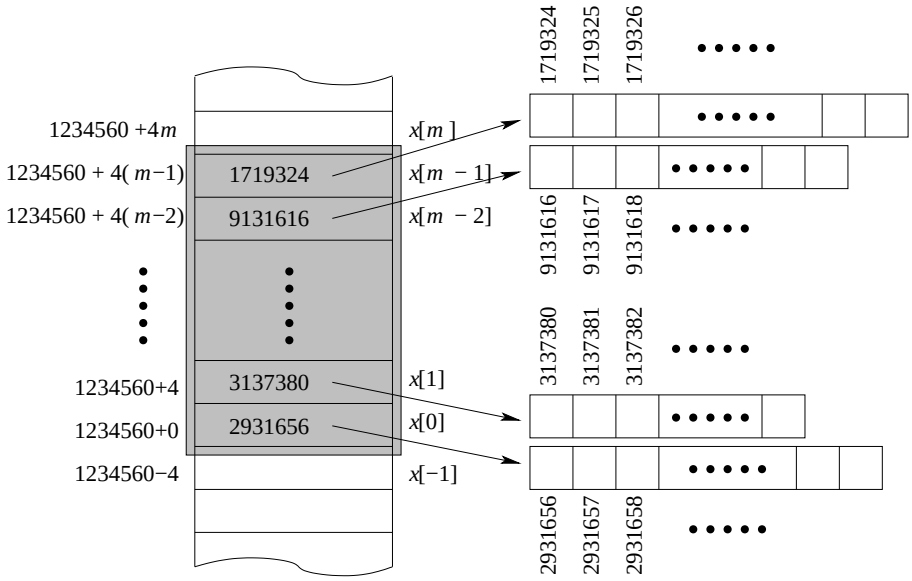
```
char (*x)[19];
x = (char(*)[19])malloc(n * sizeof(char[19]));
```

Dealokacija zauzete memorije:

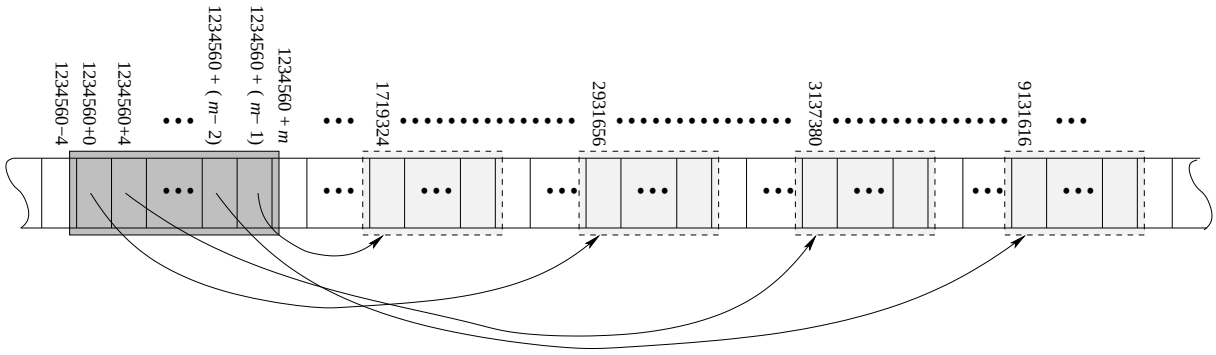
```
free(x);
```

3. **Niz (dinamički alociranih) nizova**: `char *(x[17]);` ili `char *x[17];`

   Varijabla `x` je jednodimenzionalno polje 17 pointera, od kojih svaki pokazuje na neki niz. U memoriji to izgleda kao na slikama 4.3 i 4.4. "Mali nizovi" su "rasuti" po memoriji (nisu nužno svi u jednom bloku) i mogu biti međusobno različitih duljina.



Slika 4.3: Prikaz dijela memorije 32-bitnog računala u kojem je pohranjen niz nizova



Slika 4.4: Niz nizova – stvarno stanje

Memoriju za "male nizove" treba alocirati prije prvog korištenja, te ju nakon zadnje upotrebe treba osloboditi. Memorija za "veliki niz" se automatski alocira odmah kod deklaracije i oslobađa na kraju bloka u kojem je varijabla deklarirana.

Koristimo li ovakvu deklaraciju za implementaciju matrice, ograničeni smo na najviše 17 redaka, dok broj stupaca nema gornje ograde (osim raspoložive memorije),

no takve matrice – zbog "razbacanosti" u memoriji ne možemo slati raznim bibliotekama poput LAPACKa i BLASa.

Deklaracija i alokacija matrice $17 \times n$ znakova:

```
char *x[17];
int i;
for (i = 0; i < 17; ++i)
  x[i] = (char*)malloc(n * sizeof(char));
```

Dealokacija zauzete memorije:

```
for (i = 0; i < 17; ++i) free(x[i]);
```

4. **Dinamički niz dinamičkih nizova**: `char **x;`

   Varijabla `x` je pointer na niz pointera na znakove. U memoriji to izgleda slično slikama 4.3 i 4.4, uz tu razliku da je i "veliki niz" dinamički alociran (dakle, i na njega pokazuje neki pointer). Dakle, možemo dinamički alocirati i "veliki" i "mali niz", a oni su svi "razbacani" po memoriji.

   Svaki od nizova (i "veliki" i sve "male") treba posebno alocirati i pažljivo dealocirati, a "mali nizovi" mogu biti međusobno različite duljine (dakle, podatak ne mora nužno biti matrica, nego može biti i – na primjer – rječnik).

   Koristimo li ovakvu deklaraciju za implementaciju matrice, nemamo ograničnja niti na broj redaka niti na broj stupaca (osim raspoložive memorije), no takve matrice – zbog "razbacanosti" u memoriji ne možemo slati raznim bibliotekama poput LAPACKa i BLASa.

   Deklaracija i alokacija matrice $m \times n$ znakova (**Oprez!** Prvo "veliki niz", pa tek onda "mali" – u rješenju zadatka 4.3.1 ćemo objasniti zašto tako):

```
char **x;
int i;
x = (char**)malloc(m * sizeof(char*));
for (i = 0; i < m; ++i)
  x[i] = (char*)malloc(n * sizeof(char));
```

   Dealokacija zauzete memorije (**Oprez!** Prvo "mali nizovi", pa tek onda "veliki"):

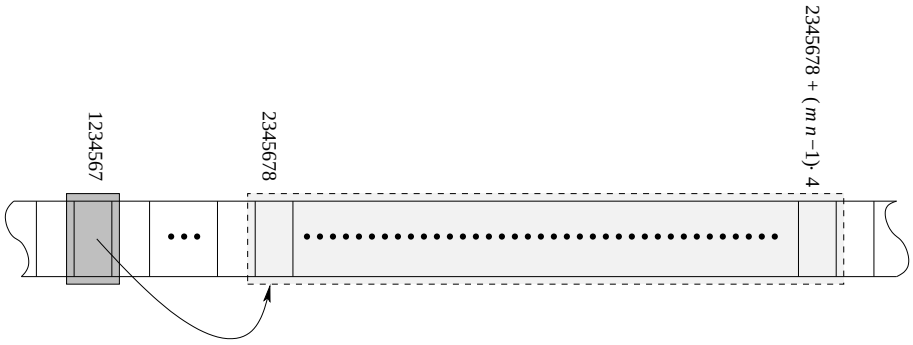
```
for (i = 0; i < m; ++i) free(x[i]);
free(x);
```

5. **Dinamički alocirani niz**: `char *x;`

   Ovdje je riječ o klasičnom nizu, kakve smo prikazali u poglavlju 4.2, a razlika je u upotrebi. Naime, ako u takav niz želimo spremiti matricu, a pojedinim podacima niza se pristupa preko jednog indeksa, njega je potrebno izračunati na osnovu dimenzija matrice (preciznije, broja stupaca u matrici) i para indeksa elementa.

   Svi elementi su grupirani u memoriji (osim, naravno, samog pointera `x`), a memorija izgleda slično kao na slici 4.5.

   Deklaracija i alokacija matrice $m \times n$ znakova:



Slika 4.5: Prikaz dijela memorije 32-bitnog računala u kojem je pohranjen dinamički alocirani niz

```
char *x;
x = (char*)malloc(m * n * sizeof(char));
```

Dealokacija zauzete memorije:

```
free(x);
```

Ispis elementa $x_{i,j}$:

```
printf("%c\n", x[i * n + j]);
```

Obično je dobro napisati pomoćnu funkciju koja par indeksa u matrici pretvara u indeks u nizu:

```
int xy2i(int x, int y, int n) {
  return i * n + j;
}
```

Tako korišteni indeksi simuliraju ono što C program i inače radi kad pristupa dvodimenzionalnom polju spremljenom po retcima. Alternativno, indekse možemo pretvarati i ovako:

```
int xy2i(int x, int y, int n) {
  return i + j * m;
}
```

Tada elemente spremamo po stupcima, što očekuju biblioteke pisane u Fortranu, a to je većina numeričkih biblioteka, uključivo i spomenute LAPACK i BLAS. Dapače, čak i njihove C-ovske varijante donose probleme s ovim (treba paziti kod compiliranja).

Ukoliko nam nije bitno da su podaci grupirani u jednom bloku memorije (a na ovom kolegiju nam to neće predstavljati prednost), pristup 4 je najopćenitiji: nema ograničenja na dimenzije, ne treba konvertirati indekse i pojedini "mali nizovi" mogu biti međusobno različitih duljina. Zbog toga ćemo u nastavku upravo tako realizirati dinamički alocirane podatke pohranjene u više dimenzija. Dakle, **matrice ćemo simulirati** preko dinamički alociranih nizova pointera na dinamički alocirane nizove, no imajte na umu da su prave matrice (u matematičkom smislu) kompaktne, tj. realiziraju se prvim, drugim ili petim od pet prije navedenih pristupa (v. napomenu 4.3.1).

Napomena 4.3.1 (Dvodimenzionalna polja i nizovi nizova)**.** *Pristup* 1 *je najjednostavniji za realizaciju i najpraktičniji za baratanje s podacima, ali je dobar samo ako znamo maksimalne duljine svih nizova (npr. ako radimo s matricom i imamo zadane gornje ograde na dimenzije matrice). Dodatno, treba paziti i da su numeričke biblioteke obično pisane u Fortranu koji matrice u dvodimenzionalnim poljima pamti po stupcima (C ih pamti po recima), pa kod pozivanja funkcija s matričnim argumentima dolazi do implicitnog transponiranja matrica, na što treba posebno paziti.*

Alternativa prvom pristupu, uz čuvanje podataka u jednom bloku memorije, je posljednji opisani pristup. Iako nespretan zbog stalnog preračunavanja indeksa, on dozvoljava punu slobodu u dimenzijama matrica te je relativno spretan način izbjegavanja diskrepancije kod različitog pohranjivanja matrica po recima odnosno stupcima (izborom ispravne funkcije konverzije indeksa).

Ovakav pristup koristi se na nekim kolegijima na diplomskim studijima Primjenjena matematika i Računarstvo i bit će važno znati te razlike. Na ovom kolegiju koristit ćemo nizove nizova, pa za sada samo ***budite svjesni da zapravo ne radimo s "pravim" matricama (u matematičkom smislu) nego ih simuliramo C-ovskim trikom****. Također, zapamtite i da većina numeričkih biblioteka zamjenjuje (u odnosu na C) ulogu redaka i stupaca.*

Pristupi 2 *i* 3 *su mješavina automatskih i dinamičkih polja, te kao takvi zadržavaju ograničenje na jednu od dimenzija polja. Dodatno, lako je zbuniti se u sintaksi, a treći pristup donosi i "razbacanost" podataka što stvara nekompatibilnost s pojedinim bibliotekama.*

Novije verzije C-a dozvoljavaju deklaracije poput `niz[m][n]`*, ali to na ovom kolegiju* ***ne dozvoljavamo****, jer je jedan od ciljeva ovog poglavlja i učenje dinamičke alokacije memorije za primjene koje nisu samo matematičke. Također, takav pristup ne omogućuje naknadnu realokaciju memorije, što je nerijetko važno u primjenama.*

Napomena 4.3.2 (Kako otkriti tip?)**.** *Ponekad su varijable deklarirane dosta kompliciranо. Na primjer, za* `char (*x)[19];` *nije sasvim očito zašto se castanje kod dinamičke alokacije radi s* `(char(*)[19]))`*, tj. zašto zvjezdica mora biti u zagradama. Ako ne možete pronaći kako u castanju navesti tip, možete se poslužiti trikom:*

```
char (*x)[19];
char t[17];
t = x;
```

Ovaj kod je pogrešan *(jer automatskom nizu* `t` *ne možemo pridružiti nikakvu vrijednost), no koristan je upravo zbog greške koju compiler prijavi. Kod* `gcc` *compilera (uobičajen na Linuxu i drugim Unixoidima, a koristi ga i Code::Blocks) ta greška je: "incompatible types when assigning to type 'char[17]' from type 'char (*)[19]'" iz čega lako iščitavamo kako se zapisuje tip varijable* `x` *(samo tip, bez same varijable, što nam treba za castanje).*

Zadatak 4.3.1. *Napišite program koji učitava prirodni broj $n \in \mathbb{N}$ i matricu $x \in \mathbb{R}^{n \times n}$. Program treba ispisati trag matrice (sumu elemenata na glavnoj dijagonali).*

Rješenje. Ovakav zadatak smo već vidjeli (zadatak 3.1.4). Rješenje ovog zadatka je identično, jednom kad riješimo učitavanje matrice.

Matrica nema zadanih ograničenja niti po visini niti po širini, pa ćemo ju realizirati na četvrti način (od gore prezentiranih), što znači da ćemo matricu odsimulirati kao dinamički alocirani niz pointera na dinamički alocirane nizove realnih brojeva (kraće: dinamički niz nizova realnih brojeva). Dakle, deklaracija će biti:

```
double **x;
```

Kako alocirati memoriju za varijablu `x`?

Alokacija memorije se uvijek može obaviti po "kuharici":

```
x = (tip od x)malloc(
  dimenzija *
  sizeof(tip od onoga na sto x pokazuje)
);
```

odnosno, u našem slučaju:

```
x = (double**)malloc(n * sizeof(double*));
```

Na taj način dobili smo niz od n pokazivača na realne brojeve. Ako bismo sada izveli

```
x[0] = (double*)malloc(n * sizeof(double));
```

onda bi `x` bio niz od n pokazivača na realne brojeve, pri čemu bi `x[0]` bio pokazivač na upravo alocirani niz od n realnih brojeva, dok bi ostali `x[i]` bili pokazivači na nealocirane dijelove memorije. Dakle, na sličan način je potrebno alocirati memoriju za **SVAKI** `x[i]`. Konačno, alokacija memorije za (simuliranu) matricu `x` izgleda ovako:

```
10  x = (double**)malloc(n * sizeof(double*));
11  for (i = 0; i < n; i++)
12    x[i] = (double*)malloc(n * sizeof(double));
```

Nakon alokacije memorije, varijablu `x` koristimo jednako kao da je riječ o statički alociranoj matrici (zbog ekvivalentnosti nizova i pointera). Na kraju je nužno osloboditi memoriju. Pri tome treba paziti da prvo oslobađamo memoriju za sve `x[i]`, a tek nakon toga za `x`. Dakle:

```
25  for (i = 0; i < n; i++)
26    free(x[i]);
27  free(x);
```

Razlog tome je što nakon `free(x)` više ne smijemo dereferencirati varijablu `x` (tj. ne smijemo raditi sa `*x` i `x[i]`), pa više ne bismo mogli osloboditi memoriju za pojedine `x[i]`. Drugim riječima, ovakvo rješenje je **POGREŠNO**:

```
free(x);
for (i = 0; i < n; i++)
  free(x[i]);
```

Konačno, rješenje zadatka izgleda ovako:

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

int main(void) {
  double **x, tr = 0;
  int i, j, n;

  printf("n = "); scanf("%d", &n);

  x = (double**)malloc(n * sizeof(double*));
  for (i = 0; i < n; i++)
    x[i] = (double*)malloc(n * sizeof(double));

  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++) {
      printf("x[%d][%d] = ", i, j);
      scanf("%lf", &x[i][j]);
    }

  for (i = 0; i < n; i++)
    tr += x[i][i];

  printf("tr(x) = %g\n", tr);

  for (i = 0; i < n; i++)
    free(x[i]);
  free(x);

  return 0;
}
```

□

Zadatak 4.3.2. *Napišite program koji učitava brojeve $a, b, c \in \mathbb{N}$, te matrice $x \in \mathbb{R}^{a\times b}$ i $y \in \mathbb{R}^{b\times c}$, te računa i ispisuje njihov umnožak $z = x \cdot y$.*

Rješenje. U ovom zadatku je važno razlikovati retke i stupce jer nije riječ o kvadratnim matricama! Također, potrebno je odrediti dimenzije matrice z. Prema definiciji matričnog množenja:

$$z_{ik} = \sum_{j=1}^{b} x_{ij} \cdot y_{jk},$$

vidimo da je matrica z dimenzije $a \times c$.

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

```
3
4  int main(void) {
5    double **x, **y, **z;
6    int a, b, c, i, j, k;
7
8    // Ucitavanje dimenzija matrica:
9    printf("a = "); scanf("%d", &a);
10   printf("b = "); scanf("%d", &b);
11   printf("c = "); scanf("%d", &c);
12
13   // Alokacija memorije za (simulirane) matrice:
14   x = (double**)malloc(a * sizeof(double*));
15   for (i = 0; i < a; i++)
16     x[i] = (double*)malloc(b * sizeof(double));
17   y = (double**)malloc(b * sizeof(double*));
18   for (i = 0; i < b; i++)
19     y[i] = (double*)malloc(c * sizeof(double));
20   z = (double**)malloc(a * sizeof(double*));
21   for (i = 0; i < a; i++)
22     z[i] = (double*)malloc(c * sizeof(double));
23
24   // Ucitavanje matrica:
25   for (i = 0; i < a; i++)
26     for (j = 0; j < b; j++) {
27       printf("x[%d][%d] = ", i, j);
28       scanf("%lf", &x[i][j]);
29     }
30   for (i = 0; i < b; i++)
31     for (j = 0; j < c; j++) {
32       printf("y[%d][%d] = ", i, j);
33       scanf("%lf", &y[i][j]);
34     }
35
36   // Racunanje produkta matrica:
37   for (i = 0; i < a; i++)
38     for (k = 0; k < c; k++) {
39       z[i][k] = 0;
40       for (j = 0; j < b; j++)
41         z[i][k] += x[i][j] * y[j][k];
42     }
43
44   // Ispis matrice z
45   printf("z = x*y =\n");
46   for (i = 0; i < a; i++) {
47     for (j = 0; j < c; j++)
48       printf("%10.2g", z[i][j]);
```

```
49      printf("\n");
50    }
51
52    // Dealokacija memorije:
53    for (i = 0; i < a; i++) free(x[i]);
54    for (i = 0; i < b; i++) free(y[i]);
55    for (i = 0; i < a; i++) free(z[i]);
56    free(x);
57    free(y);
58    free(z);
59
60    return 0;
61  }
```

□

Napomena 4.3.3. *Prethodni zadatak je izuzetno bitan za razumijevanje dinamičke alokacije. Pokušajte ga riješiti samostalno, te provjerite granice u petljama i vrijednosti u pozivima* `malloc()` *i* `free()`.

Složenost ovog algoritma je, očito, $O(n^3)$. Postoje i brži algoritmi za množenje velikih matrica: najčešće korišteni, Strassenov, sa složenošću $O(n^{2.807})$ i, trenutno teoretski najbrži poznati, Coppersmith-Winogradov sa složenošću $O(n^{2.376})$. Potonji se ne koristi u praksi zbog velikih konstanti skrivenih iza $O(\cdot)$ notacije, pa se ubrzanje postiže samo za izuzetno velike matrice.

Zadatak 4.3.3 (Za samostalnu vježbu)**.** *Riješite zadatke iz poglavlja 3 ignorirajući ograničenja na duljine polja i dimenzije matrica.*

Pokušajte barem neke od tih zadataka riješiti i alociranjem "pravih" matrica, u smislu napomene 4.3.1.

Zadatak 4.3.4 (Minesweeper)**.** *Riješite zadatak 3.1.22 bez postavljanja ograničenja na dimenzije polja, tj. na veličinu parametara m i n.*

Dinamički alocirana dvodimenzionalna polja ne moraju nužno biti pravokutnog oblika (ono što obično podrazumijevamo pod pojmom "matrice"), nego reci mogu biti različitih duljina.

Zadatak 4.3.5. *Održava se nekakvo natjecanje u kojem su igrači označeni brojevima od 0 do $n-1$. Svaka dva igrača igraju točno jednom i točno jedan od njih mora pobijediti. Napišite program koji učitava prirodni broj n, te za svakog igrača učitava koje je igrače pobijedio. Program treba provjeriti da li su uneseni podaci ispravni (tj. da li za svaki par (i, j), $i \neq j$, postoji točno jedan zapis o pobjedi, te da niti jedan igrač nije pobijedio sam sebe), te ispisati igrače (njihove brojeve) padajuće prema broju pobjeda. Uz svakog igrača treba napisati koliko pobjeda ima i koga je sve pobijedio.*

Ovaj zadatak moguće je riješiti na dva načina. Mi ćemo ga riješiti pomoću nepravokutnog dvodimenzionalnog polja, te ćemo dati uputu za rješenje pomoću dvodimenzionalnog bit-polja.

Rješenje (nepravokutno polje). U ovom programu trebamo držati popis igrača. Njihova imena, u početku, odgovaraju njihovim indeksima. No, zadatak traži sortiranje, pa se njihov redoslijed može i promijeniti. Zbog toga imena moramo pamtiti u nizu (zvat ćemo ga `igraci`).

Želimo pamtiti i pobjede, što je po jedan niz za svakog igrača, pa nam treba niz nizova, tj. dvodimenzionalno polje (npr. `pob`). No, nema svaki igrač jednak broj pobjeda, pa to polje neće biti pravokutno.

Dodatno, za svakog igrača moramo znati broj pobjeda, odnosno duljinu niza `pob[i]`. Te podatke ćemo čuvati u polju `br_pob`.

Nizove sortiramo prema vrijednosti u odgovarajućim elementima polja `br_pob`. Pri tome treba paziti da se zamjene vrše **na sva tri niza**, kako bi podaci ostali usklađeni. Elementi niza `pob` su pokazivači, pa na njima smijemo vršiti jednaku zamjenu kao na "običnim" poljima cijelih brojeva (naravno, uz poštivanje tipa pomoćne varijable).

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

int main(void) {
  int *igraci, **pob, *br_pob, n, i, j, k;

  printf("Broj igraca: "); scanf("%d", &n);

  // Alokacija memorije i ucitavanje pobjeda:
  igraci = (int*)malloc(n * sizeof(int));
  for (i = 0; i < n; i++) igraci[i] = i;
  br_pob = (int*)malloc(n * sizeof(int));
  pob = (int**)malloc(n * sizeof(int*));
  for (i = 0; i < n; i++) {
    printf("Koliko pobjeda ima igrac %d? ", i);
    scanf("%d", &br_pob[i]);
    pob[i] = (int*)malloc(br_pob[i] * sizeof(int));
    for (j = 0; j < br_pob[i]; j++) {
      printf(
        "Koji je %d. igrac kojeg je %d pobijedio? ",
        j+1, i
      );
      scanf("%d", &pob[i][j]);
    }
  }

  // Provjera ulaza:
  // 1. Provjera da nitko nije sam sebe pobijedio
```

```
29  for (i = 0; i < n; i++)
30    for (j = 0; j < br_pob[i]; j++)
31      if (i == pob[i][j]) {
32        printf("Greska: igrac %d je pobijedio ", i);
33        printf("sam sebe!\n");
34        exit(1);
35      }
36  // 2. Provjera da je svaki par odigrao s tocno
37  //      jednom pobjedom
38  for (i = 0; i < n; i++)
39    for (j = i + 1; j < n; j++) {
40      int pobjeda = 0;
41      for (k = 0; k < br_pob[i]; k++)
42        if (pob[i][k] == j)
43          if (pobjeda) {
44            printf("Greska: barem dvije igre ");
45            printf("igraca %d i %d!\n", i, j);
46            exit(1);
47          } else
48            pobjeda = 1;
49      for (k = 0; k < br_pob[j]; k++)
50        if (pob[j][k] == i)
51          if (pobjeda) {
52            printf("Greska: barem dvije igre ");
53            printf("igraca %d i %d!\n", i, j);
54            exit(1);
55          } else
56            pobjeda = 1;
57      if (!pobjeda) {
58        printf("Greska: nema ni jedne igre igraca ");
59        printf("%d i %d!\n", i, j);
60        exit(1);
61      }
62    }
63
64  // Sort po broju pobjeda
65  for (i = 0; i < n; i++)
66    for (j = i + 1; j < n; j++)
67      if (br_pob[i] < br_pob[j]) {
68        int t, *tp;
69        t = igraci[i];
70        igraci[i] = igraci[j];
71        igraci[j] = t;
72        t = br_pob[i];
73        br_pob[i] = br_pob[j];
74        br_pob[j] = t;
```

```
75          tp = pob[i];
76          pob[i] = pob[j];
77          pob[j] = tp;
78        }
79
80    // Ispis
81    for (i = 0; i < n; i++) {
82      printf("Igrac %d ima %d pobjeda", igraci[i],
83        br_pob[i]);
84      if (br_pob[i]) {
85        printf(": %d", pob[i][0]);
86        for (j = 1; j < br_pob[i]; j++)
87          printf(", %d", pob[i][j]);
88      }
89      printf("\n");
90    }
91
92    // Dealokacija memorije:
93    for (i = 0; i < n; i++) free(pob[i]);
94    free(pob);
95    free(igraci);
96    free(br_pob);
97
98    return 0;
99  }
```

□

Uputa (bit-polje). Bit-polje je polje (jednodimenzionalno ili višedimenzionalno) koje sadrži samo nule i jedinice (dakle vrijednosti koje se mogu pohraniti u jednom bitu). Ovdje možemo deklarirati matricu `pobjede` $\in \{0,1\}^{n\times n}$. Na mjesto (i,j) stavljamo 1 ako je igrač i pobijedio igrača j; inače stavljamo nulu.

Provjera ispravnosti podataka se svodi na provjeru da za svaki par (i,j), $i \neq j$, imamo nulu na mjestu (i,j) i jedinicu na mjestu (j,i) ili obrnuto; na dijagonali matrice moraju biti nule.

Kriterij za sort su sume redaka. □

Nepravokutna polja dolaze do puno jačeg izražaja prilikom korištenja stringova (poglavlje 5)..

4.4 Realokacija memorije

Zadatak 4.4.1. *Napišite dio programa koji učitava prirodni broj n i niz* `x` *od n realnih brojeva. Nakon toga, program treba učitati broj m, te još m realnih brojeva koje treba dodati na kraj niza* `x`.

Rješenje (`malloc()`). Zadatak možemo riješiti alokacijom novog, duljeg niza. Pri tome, podatke iz starog niza treba kopirati u novi niz, a zatim treba osloboditi memoriju koja je pripadala starom nizu:

```
1  int i, n, m;
2  double *x, *y;
3
4  printf("n = "); scanf("%d", &n);
5  x = (double*)malloc(n * sizeof(double));
6  for (i = 0; i < n; i++) {
7    printf("x[%d] = ", i);
8    scanf("%d", &x[i]);
9  }
10 printf("m = "); scanf("%d", &m);
11 y = (double*)malloc((n + m) * sizeof(double));
12 for (i = 0; i < n; i++) y[i] = x[i];
13 free(x);
14 x = y;
15 for (i = n; i < n + m; i++) {
16   printf("x[%d] = ", i);
17   scanf("%d", &x[i]);
18 }
19
20 for (i = 0; i < n + m; i++)
21   printf("%d\n", x[i]);
22
23 free(x);
```

Izraz `x=y` **NIJE** pridruživanje nizova, nego samo pointera: nakon tog izraza, `x` i `y` pokazuju na isti niz (istu memorijsku lookaciju)! No, pri tome gubimo adresu memorije koja je prije bila alocirana kao niz `x` (iako sama memorija ostaje zauzeta), pa ju zbog toga prvo treba osloboditi. □

Dodavanje (ili oduzimanje) memorije dinamički alociranom nizu je normalan zahtjev i nema potrebe da rješenje bude komplicirano, kao u prethodnom programu (linije 11–14). U C-u, za takve zahvate postoji funkcija `realloc()` koja prima dva argumenta:

```
niz = (tip_elementa*)realloc(niz, nova_velicina_niza);
```

Prvi parametar funkcije `realloc()` je sam `niz`, a drugi je nova veličina tog niza (smije biti i veća i manja od stare veličine). Funkcija će pokušati alocirati potrebnu memoriju. Ako to uspije (tj. ako je nova veličina manja od stare ili ako ima dovoljno prostora "u komadu" iza postojećeg niza), funkcija vraća originalnu adresu niza.

Ako funkcija ne može alocirati potrebnu memoriju (tj. ako ne uspije produljiti niz), alocirat će skroz novu memoriju (na novoj lokaciji), prebaciti stare podatke u novi niz, osloboditi memoriju u kojoj se niz prije nalazio i vratiti adresu upravo alocirane memorije (dakle točno ono što u prethodnom programu radimo u linijama 11–14).

Rješenje (`realloc()`*).*

```
1   int i, n, m;
2   double *x, *y;
3
4   printf("n = "); scanf("%d", &n);
5   x = (double*)malloc(n * sizeof(double));
6   for (i = 0; i < n; i++) {
7     printf("x[%d] = ", i);
8     scanf("%d", &x[i]);
9   }
10  printf("m = "); scanf("%d", &m);
11
12  x = (double*)realloc(x, (n + m) * sizeof(double));
13
14
15  for (i = n; i < n + m; i++) {
16    printf("x[%d] = ", i);
17    scanf("%d", &x[i]);
18  }
19
20  for (i = 0; i < n + m; i++)
21    printf("%d\n", x[i]);
22
23  free(x);
```

□

Zadatak 4.4.2. *Napišite program koji učitava nenegativne realne brojeve dok ne učita nulu. Program treba ispisati one učitane brojeve koji su strogo veći od geometrijske sredine učitanih.*

Rješenje. Očito, podatke moramo pohraniti u niz (ili listu, ali o tome u poglavlju 7), no ne možemo *a priori* alocirati dovoljno memorije. Jedini način za učitati ovakav niz je realociranje memorije kad alocirana memorija postane premala. Pri tome, dobro je znati da je poziv
`realloc(NULL, size);`
ekvivalentan pozivu `malloc(size);`
Dakle, učitavamo brojeve, te za svaki broj koji želimo pohraniti u niz alociramo dodatnu memoriju:

```
6   double *niz = NULL, gs = 1;
7   int n = 0, i;
8
9   while (1) {
10    double x;
11    printf("Ucitajte nenegativni broj: ");
12    scanf("%lf", &x);
```

```
13    if (!x) break;
14    niz = (double*)realloc(niz, ++n * sizeof(double));
15    niz[n - 1] = x;
16  }
```

Ovdje se `if(...) break;` nalazi odmah iza učitavanja broja jer ne želimo da upisana nula bude element niza. Kad bismo i nju željeli u nizu, onda bi `if(...) break;` išlo na kraj `while(1)` petlje (iza `niz[n - 1] = x;`).

Podsjetimo se što radi `realloc()` ako nema dovoljno memorije: alocira novu memoriju, prebacuje sve podatke, otpušta staru memoriju i vraća adresu nove memorije. Ovo je jako spor postupak, pa je dobro provoditi ga što rjeđe. Jedan način da se to postigne je alociranje nekoliko mjesta odjednom. Kad se sva alocirana memorija popuni, alociramo nekoliko dodatnih mjesta. Dakle, jednako kao i u prikazanom isječku kôda, samo što ne alociramo jedno po jedno mjesto nego, na primjer, deset po deset mjesta:

```
6   double *niz = NULL, gs = 1;
7   int n = 0, length = 0, i;
8
9   while (1) {
10    double x;
11    printf("Duljina niza: %d; ", n);
12    printf("alocirana memorija: %d\n", length);
13    printf("Ucitajte nenegativni broj: ");
14    scanf("%lf", &x);
15    if (!x) break;
16    if (length < ++n)
17      niz = (double*)realloc(
18        niz,
19        (length += 10) * sizeof(double)
20      );
21    niz[n - 1] = x;
22  }
```

Napomenimo da su oba načina unosa točni. Razlika je u tome što je drugi način bolji, u smislu efikasnijeg izvođenja.

Potrebno je još izračunati geometrijsku sredinu. To radimo slično aritmetičkoj sredini: izračunamo produkt svih elemenata niza (npr. u varijabli `gs`), te vadimo n-ti korijen iz `gs`. Kod vađenja korijena treba biti oprezan. Na primjer, ovo je **POGREŠAN** način:
`pow(gs, 1/n);`
Problem s ovim izrazom je to što su i 1 i n cjelobrojne vrijednosti, pa je `1/n` **cjelobrojno dijeljenje**, čiji je rezultat, naravno, nula (osim za $n = 1$). To znači da izraz neće računati $\sqrt[n]{\texttt{gs}}$, nego $\texttt{gs}^0 = 1$. Ispravni izraz je
`pow(gs, 1./n);`, `pow(gs, 1.0/n);` ili `pow(gs, 1/(double)n);`
ili neki slični. Konačno, rješenje zadatka (uz pomoćne ispise o duljini niza, alociranoj memoriji i geometrijskoj sredini) izgleda ovako:

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

```c
#include <stdlib.h>
#include <math.h>

int main(void) {
  double *niz = NULL, gs = 1;
  int n = 0, length = 0, i;

  while (1) {
    double x;
    printf("Duljina niza: %d; ", n);
    printf("alocirana memorija: %d\n", length);
    printf("Ucitajte nenegativni broj: ");
    scanf("%lf", &x);
    if (!x) break;
    if (length < ++n)
      niz = (double*)realloc(
        niz,
        (length += 10) * sizeof(double)
      );
    niz[n - 1] = x;
  }

  for (i = 0; i < n; i++) gs *= niz[i];
  gs = pow(gs, 1./n);
  printf("Geometrijska sredina: %g\n", gs);

  for (i = 0; i < n; i++)
    if (niz[i] > gs)
      printf("%g\n", niz[i]);

  free(niz);

  return 0;
}
```

□

Zadatak 4.4.3. *Pokušajte riješiti prethodni zadatak bez upotrebe funkcija iz biblioteke* `math` *(dakle, bez* `pow()`*).*

Zadatak 4.4.4. *Napišite program koji učitava realne brojeve dok ne učita* 17.19 *(koji ne smije postati element niza). Program treba invertirati niz i ispisati one njegove elemente kojima je cjelobrojni dio paran (nije dovoljno samo unatrag ispisati tražene elemente niza!).*

Podsjetnik: *Ako je* `x` *tipa* `double`*, njegov cjelobrojni dio možete dobiti castanjem na* `int`*:*

```
(int)x
```

ili pridruživanjem cjelobrojnoj varijabli (implicitna konverzija):
`int y = x;`

Zadatak 4.4.5. *Napišite program koji učitava cijele brojeve dok ne učita nulu, te ispisuje aritmetičku sredinu svih učitanih prostih brojeva, a zatim ispisuje i sve elemente niza koji su strogo manji od te artimetičke sredine.*

Zadatak 4.4.6. *Napišite program koji učitava realne brojeve dok ne učita negativni broj (koji također mora postati element niza). Program treba sortirati učitane brojeve prema apsolutnoj vrijednosti, te ih ispisati.*

Zadatak 4.4.7. *Napišite program koji učitava cijele brojeve dok ne učita negativni broj (koji ne smije postati element niza). Program treba unatrag ispisati niz i to samo one elemente koji su strogo veći od prosječne sume znamenaka svih elemenata niza.*

Zadatak 4.4.8. *Napišite program koji učitava prirodni broj $n \in \mathbb{N}$, te niz $(a_i)_{i=1}^{n-1}$ od n prirodnih brojeva. Nakon učitavanja, program treba pronaći proste faktore svih učitanih brojeva, te – za svaki učitani broj posebno – treba ispisati one koji su strogo veći od aritmetičke sredine svih dobivenih prostih faktora.*

Rješenje. Proste faktore držat ćemo u polju `pf`. Kako nam treba niz prostih faktora za svaki `a[i]`, polje `pf` mora biti dvodimenzionalno. No, prostih faktora nema jednako mnogo za sve `a[i]`, pa će nizovi `pf[i]` biti različite duljine. Za svaki od tih nizova treba negdje pamtiti njegovu duljinu; mi ćemo za tu svrhu koristiti nulti element svakog niza. Dakle:

- `pf[i][0]` bit će broj prostih faktora broja `a[i]`
- `pf[i][j]`, za `j` $\in \{1, 2, \ldots,$ `pf[i][0]`$\}$, bit će prosti faktori broja `a[i]`.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void) {
5    int *a, **pf, i, j, n, pf_cnt = 0;
6    double as = 0;
7
8    // Ucitavanje duljine niza:
9    printf("n = "); scanf("%d", &n);
10
11   // Alokacija memorije i ucitavanje niza a:
12   a = (int*)malloc(n * sizeof(int));
13   for (i = 0; i < n; i++) {
14     printf("a[%d] = ", i);
15     scanf("%d", &a[i]);
16   }
17
```

```
18  // Racunanje prostih faktora:
19  pf = (int**)malloc(n * sizeof(int*));
20  for (i = 0; i < n; i++) {
21    int x = a[i], k = 2;
22    pf[i] = (int*)malloc(sizeof(int));
23    pf[i][0] = 0;
24    while (x > 1) {
25      if (!(x % k)) {
26        while (!(x % k)) x /= k;
27        pf[i][0]++;
28        pf[i] = (int*)realloc(
29          pf[i],
30          (pf[i][0] + 1) * sizeof(int)
31        );
32        pf[i][pf[i][0]] = k;
33      }
34      k++;
35    }
36  }
37
38  // Aritmeticka sredina prostih faktora:
39  for (i = 0; i < n; i++) {
40    pf_cnt += pf[i][0];
41    for (j = 1; j <= pf[i][0]; j++)
42      as += pf[i][j];
43  }
44  as /= pf_cnt;
45
46  // Pomocni ispis:
47  printf("Aritmeticka sredina: %g\n", as);
48
49  // Ispis prostih faktora vecih od as:
50  for (i = 0; i < n; i++) {
51    int prvi = 1;
52    printf("Faktori od %d: ", a[i]);
53    for (j = 1; j <= pf[i][0]; j++)
54      if (pf[i][j] > as) {
55        if (prvi)
56          prvi = 0;
57        else
58          printf(", ");
59        printf("%d", pf[i][j]);
60      }
61    if (prvi) printf("nema");
62    printf("\n");
63  }
```

```
64
65    // Dealokacija memorije:
66    for (i = 0; i < n; i++) free(pf[i]);
67    free(pf);
68    free(a);
69
70    return 0;
71  }
```

□

Zadatak 4.4.9. *Riješite prethodni zadatak tako da program svaki prosti faktor uzima u obzir onoliko puta kolika mu je kratnost (umjesto samo jednom).*

Zadatak 4.4.10 (Mozgalica)**.** *Da li je sljedeći program ispravan? Ako da, što će se desiti prilikom njegovog izvršavanja; ako ne, zašto?*

```
1  #include <stdio.h>
2
3  int *f(void) {
4    int a[3] = {1, 2, 3};
5    return a;
6  }
7
8  int main(void) {
9    int a[3];
10
11   a = f();
12   printf("%d, %d, %d\n", a[0], a[1], a[2]);
13
14   return 0;
15 }
```

Zadatak 4.4.11 (Mozgalica)**.** *Isto kao u prethodnom zadatku, samo uz promjenu:*

```
9    int *a;
```

Napomena 4.4.1. *Program iz druge "mozgalice" će se* **vjerojatno** *ispravno ponašati, ali on ipak nije dobar (zašto?).*

Poglavlje 5

Stringovi

U C-u postoji tip podataka `char` koji služi za pohranu jednog znaka. No, želimo li pospremiti riječ, rečenicu ili čak veći tekst, u C-u ne postoji odgovarajući tip. U tu svrhu koriste se nizovi znakova za koje vrijedi sve što smo vidjeli u poglavljima 3 i 4, kao i u "Programiranju 1" gdje smo obrađivali statičke nizove. Kao dodatak postojećim funkcionalnostima nizova u C-u, postoji i niz funkcija specifično napisanih za nizove znakova (stringove).

5.1 Osnovne operacije

Stringovne konstante navodimo kao nizove znamenaka između **dvostrukih** navodnika. Na primjer:
`"Pero Sapun"`
je string od 10 znakova (9 slova i 1 razmak). Za razliku od stringova, znakovne konstante se navode između **jednostrukih** navodnika. Tako je `"A"` string, a `'A'` znak. Ova razlika je izuzetno bitna, jer ne možemo koristiti znakove umjesto stringova (niti obrnuto). To bi bilo kao da pokušamo koristiti `int` umjesto niza brojeva (ili obrnuto).

Napomena 5.1.1 (Koliko string zauzima memorije?)**.** *String "*`Pero Sapun`*" ima* 10 *slova, ali u memoriji zauzima* 11 *mjesta! Prisjetimo "običnih" nizova: postoji memorija dodijeljena nekom nizu (statički ili dinamički), ali nigdje nije pohranjeno koliko elemenata se stvarno nalazi u nizu (tj. koji dio niza zaista koristimo). U tu svrhu uvodili smo pomoćnu varijablu (najčešće smo ju zvali* `n`*) u kojoj smo čuvali broj elemenata niza.*

Kod stringova, kao kraj niza se koristi (nevidljivi) znak, tzv. null-character (`'\0'`*), čija je ASCII vrijednost nula. Sve funkcije koje barataju sa stringovima upotrebljavaju taj znak kao oznaku za kraj stringa. Ako stringu pristupamo direktno, kao nizu znakova (umjesto pomoću specijalnih stringovnih funkcija), moramo paziti na označavanje završetka!*

Kod ispisa stringova naredbom `printf()` koristi se format `%s`.

Učitavanje je nešto složenije: format `%s` označava učitavanje **jedne riječi**. Želimo li pročitati cijeli redak (tj. string do prvog skoka u novi red), treba upotrijebiti format `%[^\n]` ili funkciju `gets()`.

Općenito, ako želimo učitavati dijelove teksta odvojene znakovima, na primjer, `'a'`, `'b'` i `'c'` onda navodimo format
`%[^abc]`
Prilikom takvog učitavanja, na primjer, teksta "Ovaj kolegij je baš zanimljiv!", C će razlikovati "riječi": "Ov", "j kolegij je ", "š z", "nimljiv!". Dakle, format `%s` je samo pokrata za `%[^ \t\n]` (jer kao separatore riječi uzima razmake, `TAB`-ove i skokove u novi red.

Nakon čitanja riječi, razmak kojim riječ završava ostaje nepročitan. No, iduće učitavanje (s istim formatom) će taj razmak zanemariti, te se on neće naći niti u jednoj od učitanih riječi.

Zadatak 5.1.1. *Napišite program koji učitava jednu riječ duljine najviše* 17 *znakova, te ispisuje:*

a) tu riječ

b) tu riječ bez prvog znaka

c) treće slovo te riječi (pretpostavite da je riječ dovoljno dugačka)

Rješenje. Prisjetimo se da je niz isto što i pokazivač na prvi element (dakle na početak) niza! Kako naredba `scanf()` mora primati adrese na koje se spremaju podaci, ispred stringovnih varijabli **NE STAVLJAMO "&"!** Također, ako imamo stringovnu varijablu `rijec` (a ona je, kako smo već rekli, ekvivalentna `&rijec[0]`), onda `&rijec[1]` predstavlja adresu drugog znaka. Nastavimo li čitati znakove od drugog znaka do `'\0'`, dobit ćemo upravo riječ bez prvog znaka (rješenje podzadatka b)). Treće (ili bilo koje drugo) slovo je znak, dakle tipa `char`, i ispisuje se pomoću formata `%c`.

Prilikom deklaracije stringa fiksne duljine, kao i kod dinamičke alokacije stringa, moramo uvijek ostaviti jedno mjesto "viška" u kojem će biti pospremljen završni znak `'\0'`. U našem zadatku, riječ može imati najviše 17 znakova, što znači da nam za pohranu te riječi u memoriji treba niz od $17 + 1 = 18$ znakova.

```
#include <stdio.h>

int main(void) {
  char rijec[18];

  printf("Upisite rijec: "); scanf("%s", rijec);

  printf("a) rijec: %s\n", rijec);
  printf("b) rijec bez prvog znaka: %s\n", &rijec[1]);
  printf("c) treci znak: %c\n", rijec[2]);

  return 0;
}
```

Izvedite program na računalu tako da upišete dvije riječi (npr. "Pero Sapun"). Pod a), program će ispisati samo prvu riječ, dok druga (bez razmaka) ostaje nepročitana! Ako programski kod (tj. linije 6–10) kopirate tako da se izvrši dva puta, onda će drugi `scanf()` učitati drugu riječ, bez čekanja na unos. □

5.2 Dinamički alocirani stringovi

Kao i kod običnih nizova, i stringovi se jednako koriste bez obzira na to da li su statički ili dinamički alocirani. Pogledajmo kako se radi s dinamičkim stringovima, a u nastavku ćemo obraditi string-specifične funkcije bez pravljenja razlika između statičke ili dinamičke alokacije.

Da bismo mogli učitati string, potrebno je **unaprijed** alocirati memoriju za njega, na primjer ovako:

```
1 char *string;
2 int len;
3 printf("Duljinja strin\-ga: ");
4 scanf("%d", &len);
5 string = (char*)malloc((len+1)*sizeof(char));
6 scanf("%s", string);
```

Potrebno je alocirati `len` + 1 mjesto zbog dodatnog znaka `'\0'` na kraju (slično prethodnom zadatku gdje smo deklarirali duljinu 18 za riječ od najviše 17 slova).

Napomena 5.2.1 (Česta greška)**.** *Prilikom učitavanja stringova, studenti ponekad je zamijene* `scanf()`*-a i alokaciju memorije (funkcija* `strlen()` *vraća duljinu stringa, kako ćemo naknadno vidjeti):*

```
scanf("%s", string);
string = (char*)malloc((strlen(string)+1)*sizeof(char));
```

Ovo je pogrešno jer `scanf()` *piše po nekoj memoriji koja ne pripada stringu (čak vjerojatno ne pripada niti samom programu), a* `malloc()` *će naknadno alocirati memoriju, i to gotovo sigurno ne onu gdje je string zapisan (ako uopće program uopće dođe do* `malloc()`*, tj. ako se ne sruši na* `scanf()`*).*

Dinamičko učitavanje stringa čiju duljinu ne znamo unaprijed je izuzetno složeno i u praksi se rijetko radi. Na primjer, ovaj komad kôda

```
1 char *big = NULL, *old_big;
2 char s[11];
3 int len = 0, old_len;
4
5 do {
6   old_len = len;
7   old_big = big;
8   scanf("%10[^\n]", s); s[10] = '\0';
```

```
9   if (!(big = realloc(big, len += strlen(s)))) {
10    free(old_big);
11    printf("Out of memory!\n");
12    exit(1);
13  }
14  strcpy(big + old_len, s);
15 } while (len - old_len == 10);
```

će učitati jednu liniju teksta proizvoljne duljine u varijablu `big`.

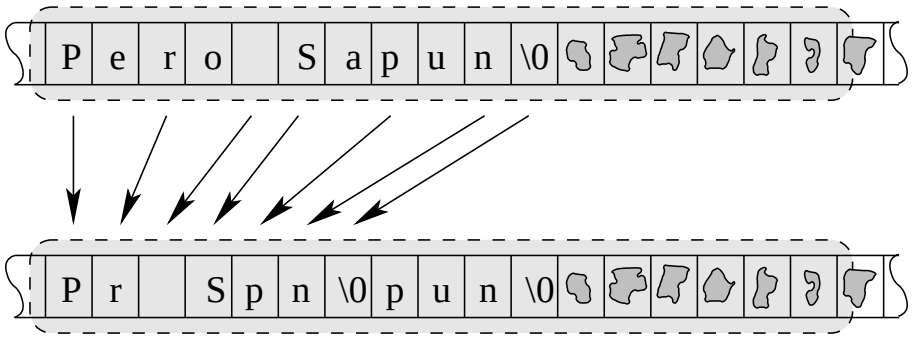
5.3 Direktno baratanje sa stringovima

String možemo tretirati upravo kao niz znakova, bez upotrebe posebnih funkcija za rad sa stringovima. Takav pristup je često potreban jer nemamo funkcije za svaki zahvat koji nam padne na pamet.

Zadatak 5.3.1. *Napišite funkciju koja kao argument uzima string, te iz njega briše sve samoglasnike. Dodatno, napišite i program koji pokazuje kako se funkcija upotrebljava.*

Rješenje. U ovom zadatku, riječ je o običnom brisanju elemenata iz niza. Pri tome, svi elementi koji se nalaze iza obrisanog, pomiču se za jedno mjesto u lijevo.
Oprez: Pomicanje elemenata treba napraviti do znaka `'\0'`, **uključujući i njega** jer on označava kraj stringa.

Na slici 5.1 vidimo promjenu u stringu (uz pretpostavku da je deklariran kao niz od 17 znakova). "Mrlje" na kraju označavaju memoriju u kojoj se nešto nalazi (u memoriji se uvijek nešto nalazi), ali ne znamo što. Također, u funkciji ne možemo znati koliko je prostora alocirano za string, te zbog toga moramo pretpostaviti da je string ispravno alociran i napunjen s podacima, te da ima dovoljno memorije za rezultat (npr. u slučaju da string treba produljiti).



Slika 5.1: Brisanje samoglasnika iz stringa

Kao što vidimo na slici, iza `'\0'` se može nalaziti bilo što. Zbog toga treba paziti da prilikom baratanja sa stringom ne "promašimo" `'\0'`, jer ćemo onda mijenjati nešto što ne pripada našem tekstu, a možda čak niti samom stringu (na slici, "mrlja" iza sivog područja).

```
1  #include <stdio.h>
2  #include <ctype.h>
3
4  void brisi_samoglasnike(char s[]) {
5    int i, j = 0;
6    for (i = 0; s[i] != '\0'; i++) {
7      char c = tolower(s[i]);
8      if (!(c == 'a' || c == 'e' || c == 'i' ||
9            c == 'o' || c == 'u'))
10       s[j++] = s[i];
11   }
12   s[j] = '\0';
13 }
14
15 int main (void) {
16   char s[17];
17
18   printf("Upisite string: ");
19   scanf("%[^\n]", s);
20   brisi_samoglasnike(s);
21   printf("Rezultat: \"%s\"\n", s);
22
23   return 0;
24 }
```

Ovdje vidimo i jednu funkciju koja nema veze sa stringovima: `char tolower(char c)` vraća malo slovo koje odgovara slovu u varijabli `c` (ako je u njoj slovo; za druge znakove vraća nepromijenjeni znak `c`). Slično, postoji funkcija `toupper()` koja vraća veliko slovo. Obje funkcije nalaze se u biblioteci `ctype`.

Funkciju smo iskoristili da bismo smanjili broj usporedbi u `if()`, jer su samoglasnici: "A", "a", "E", "e", itd. (ukupno 10 znakova). Ovako, slovo pretvaramo u malo, pa je dovoljno provjeriti samo male samoglasnike (njih 5). Naravno, ovo služi samo za usporedbu, dok pridruživaje (linija 10) treba raditi s originalnim znakom `s[i]`. □

Napomena 5.3.1 (Česta greška)**.** *Funkcije* `tolower()` *i* `toupper()` *rade sa znakovima, a ne sa stringovima. Drugim riječima, pogrešno je napisati:*

```
char s[] = "Pero";
s = toupper(s);
printf(s);
```

Umjesto toga, treba upotrijebit neku (npr. `for()`*) petlju:*

```
char s[] = "Pero";
int i;
for (i = 0; s[i]; i++) s[i] = toupper(s[i]);
printf(s);
```

Funkciju iz prethodnog zadatka možemo i puno "zapetljanije" napisati. Opkušajte otkriti kako (i zašto) sljedeća modifikacija također radi ono što se traži:

```
4  void brisi_samoglasnike(char s[]) {
5    char *i = s;
6    for (; *i; i++) {
7      char c = tolower(*i);
8      if (c - 'a' && c - 'e' && c - 'i' &&
9          c - 'o' && c - 'u')
10       *(s++) = *i;
11   }
12   *s = 0;
13 }
```

Zadatak 5.3.2. *Napišite funkciju koja kao argument prima jedan string, te iz njega briše svaki treći znak. Napišite i program koji pokazuje kako se funkcija upotrebljava.*

U biblioteci `ctype` postoje i sljedeće funkcije za provjeru pripadnosti nekog **znaka** (**NE cijelih stringova!**) nekoj klasi znakova:

`int isalnum(int c)` je li `c` slovo ili broj?

`int isalpha(int c)` je li `c` slovo?

`int isblank(int c)` je li `c` razmak ili `'\t'` (`TAB`)?

`int isdigit(int c)` je li `c` znamenka?

`int isgraph(int c)` je li `c` znak koji se može ispisati na ekranu (osim razmaka)?

`int islower(int c)` je li `c` malo slovo?

`int isprint(int c)` je li `c` znak koji se može ispisati na ekranu (uključujući i razmak)?

`int ispunct(int c)` je li `c` znak koji se može ispisati na ekranu, ali nije slovo?

`int isspace(int c)` je li `c` razmak, `'\t'`, `'\n'` (skok u novi red), `'\f'`, `'\r'` ili `'\v'`?

`int isupper(int c)` je li `c` veliko slovo?

`int isxdigit(int c)` je li `c` heksadecimalna znamenka (`0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `a`, `b`, `c`, `d`, `e`, `f`, `A`, `B`, `C`, `D`, `E`, `F`)?

Ove funkcije odgovaraju na pitanja s "da" (vrijednost različita od nule) ili "ne" (vrijednost 0).

Zadatak 5.3.3. *Napišite funkciju koja kao argument prima jedan string, te iz njega briše svako treće slovo. Napišite i program koji pokazuje kako se funkcija upotrebljava.*
Uputa: *Upotrijebite funkciju* `int isalpha(int c)` *koja vraća* 1 *ako se u vrarijabli* `c` *nalazi znak (vrši se automatska konverzija između* `int` *i* `char`*), odnosno* 0 *ako u* `c` *nije slovo (nego neki drugi znak).*

Zadatak 5.3.4. *Napišite funkciju koja kao argument prima jedan string, te iz njega briše sve znamenke. Napišite i program koji pokazuje kako se funkcija upotrebljava.*
Uputa: *Upotrijebite funkciju* `isdigit(int c)` *koja provjerava nalazi li se u* `c` *znamenka.*

Zadatak 5.3.5. *Napišite funkciju koja kao argument prima jedan string i jedan znak. Funkcija treba duplicirati svako pojavljivanje znaka u stringu. Na primjer, ako su zadani string* `"Popokatepetl"` *i znak* `'p'`*, promijenjeni string treba biti* `"Poppokateppetl"`*. Napišite i program kojim testirate funkciju.*
Pretpostavite da je za string alocirano dovoljno memorije da se promjena izvede.

Rješenje. Ovdje treba biti oprezan: po nizu treba "trčati" s desna na lijevo. Pokušamo li to raditi s lijeva na desno, "pregazit" ćemo neke vrijednosti!

No, ne znamo koliko je string dugačak, niti koliko znakova treba dodati. To dvoje tražimo u prvoj petlji. Nakon nje, u varijabli `i` se nalazi indeks znaka `'\0'` (tj. duljina stringa `s`), a u varijabli `j` nalazi se broj znakova koje treba dodati (što je jednako broju pojavljivanja znaka `c` u stringu).

U drugoj petlji idemo s desna na lijevo i kopiramo znakove na potrebna mjesta. Skicirajte string u memoriji da biste lakše vidjeli kako funkcija radi.

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

void dupliciraj_c(char s[], char c) {
  int i, j = 0;
  for (i = 0; s[i] != '\0'; i++)
    if (s[i] == c) j++;
  for (; i >= 0; i--) {
    s[i + j] = s[i];
    if (s[i] == c) s[i + (--j)] = s[i];
  }
}

int main (void) {
  char s[17];

  printf("Upisite string: ");
  scanf("%[^\n]", s);
  dupliciraj_c(s, 'p');
  printf("Rezultat: \"%s\"\n", s);

  return 0;
}
```

□

Zadatak 5.3.6. *Modificirajte rješenje prethodnog zadatka tako da se znak* `c` *ne udvostručuje, nego utrostručuje.*

Uputa: *Potrebno je promijeniti promjenu brojača* `j` *u prvoj petlji, te malo doraditi liniju* 10.

Zadatak 5.3.7. *Napišite funkciju koja kao argument prima jedan string* `s`*, jedno slovo* `c` *(deklarira se kao* `char`*, ali garantiramo da će korisnik zadati slovo) i jedan broj* `n`*. Funkcija treba n-terostručiti svako pojavljivanje slova* `c` *u stringu* `s`*, neovisno o tome je li riječ o malom ili velikom slovu. Na primjer, ako su zadani string* `"Popokatepetl"`*, znak* `'p'` *i broj* `4`*, promijenjeni string treba biti* `"PPPPopppppokatepppppetl"`*. Napišite i program kojim testirate funkciju.*
Pretpostavite da je za string alocirano dovoljno memorije da se promjena izvede.

Zadatak 5.3.8. *Napišite funkciju*
`void tr(char s[], const char f[], const char t[])`
koja u stringu `s` *zamjenjuje svaki znak iz* `f` *s odgovarajućim znakom iz* `t`*. Na primjer, ako su dani stringovi* `"Pero Sapun"`*,* `"pas"` *i* `"mir"`*, onda prvi string treba postati* `"Pero Simun"` *(jer* `'p'` *prelazi u* `'m'`*,* `'a'` *u* `'i'` *i* `'s'` *u* `'r'`*; pri tome se* **znakovi** `'S'` *i* `'s'` *razlikuju).*
Možete pretpostaviti da su stringovi `f` *i* `t` *jednake duljine.*

Rješenje. Iako naizgled kompliciran, ovaj zadatak traži jedno "trčanje" po stringu, te zamjenu jednog znaka drugim za svaki znak (što je još jedna petlja). Pri tome izvršavanje unutrašnje petlje moramo prekinuti ako je izvršena zamjena, kako ne bi došlo do nekoliko zamjena istog znaka. Ovdje je `break` gotovo nezamjenjiv.

```
1 void tr(char s[], const char f[], const char t[]) {
2   int i, j = 0;
3   for (i = 0; s[i] != '\0'; i++)
4     for (j = 0; f[j] != '\0'; j++)
5       if (s[i] == f[j]) {
6         s[i] = t[j];
7         break;
8       }
9 }
```

Kad ne bismo imali garanciju jednake duljine stringova `f` i `t`, rješenje bi moralo raditi tako da u obzir uzima samo prvih m znakova, pri čemu je m duljina kraćeg od ta dva stringa. No, m ne moramo direktno računati; dovoljna je modifikacija uvjeta u unutrašnjoj petlji:

```
3     for (j = 0; f[j] != '\0' && t[j] != '\0'; j++)
```

ili, kraće:

```
3     for (j = 0; f[j] && t[j]; j++)
```

□

Napomena 5.3.2. *U deklaraciji funkcije, varijable* `f` *i* `t` *navedene su kao* `const char[]`*. Ključna riječ* `const` *compileru govori da se stringovi* `f` *i* `t` *u funkciji neće mijenjati, što nam efektivno omogućuje pozive poput*

```
tr(s, "pas", "mir");
```

Zadatak 5.3.9. *Riješite prethodni zadatak upotrebom binarnog traženja, uz pretpostavku da je string* `f` *uzlazno sortiran.*

Zadatak 5.3.10. *Napišite funkciju koja uzima tri argumenta: string* `s`*, te znakove* `c1` *i* `c2`*. Funkcija treba obrisati sva pojavljivanja znaka* `c1` *i dupli*cirati *sva pojavljivanja znaka* `c2` *u stringu* `s`*. Smijete pretpostaviti da je* `c1` $\neq$ `c2`*.*

Zadatak 5.3.11. *Napišite funkciju koja uzima četiri argumenta: niz stringova* `s`*, broj stringova u nizu* `n`*, te znakove* `c1` *i* `c2`*. Funkcija treba obrisati sva pojavljivanja znaka* `c1` *i dupli*cirati *sva pojavljivanja znaka* `c2` *u stringovima u nizu* `s`*. Smijete pretpostaviti da je* `c1` $\neq$ `c2`*.*

Uputa. Funkciju možete jednostavno realizirati pomoću rješenja prethodnog zadatka. Dovoljna je jedna `for()`-petlja koja za svaki element niza poziva funkciju iz prethodnog zadatka. □

Zadatak 5.3.12. *Napišite funkcije*
`strtolower(char *s)` *i* `strtoupper(char *s)`
koje uzimaju po jedan argument (string `s`*), te sva njegova slova prebacuju u mala, odnosno velika; ostale znakove ne mijenjaju.*
Uputa: *Upotrijebite* `tolower()` *i* `toupper()` *iz biblioteke* `ctype`*, kako je prikazano u napomeni 5.3.1.*

Zadatak 5.3.13. *Napišite funkciju* `int palindrom(char *s)` *koja vraća* 1 *ako je string* `s` *palindrom (niz znakova koji se jednako čitaju s lijeva na desno i s desna na lijevo; npr.* `"anavolimilovana"`*,* `"aba"` *i* `"abba"`*, ali ne i* `"ana voli milovana"`*); inače treba vratiti* 0*.*

Zadatak 5.3.14 (Šlag na kraju)**.** *Bez korištenja string-specifičnih funkcija, napišite funkciju koja prima jedan niz znakova i jedan string, te vraća* 1 *ako se string može dobiti permutacijom nekih elemenata niza; u protivnom treba vratiti nulu.*

Rješenje. Niz znakova (dakle, ne string) pamtimo pomoću dvije varijable: sam niz (tipa `char[]`) i duljina (tipa `int`). Riječ je string, pa nju pamtimo samo kao jednu varijablu (tipa `char[]`) u kojoj je duljina implicitno sadržana terminalnim znakom `'\0'`.

```
1 int is_perm(const char niz[], int n,
2              const char rijec[]) {
3   int i;
4   char *niz2;
5
6   niz2 = (char*)malloc(n * sizeof(char));
7   for (i = 0; i < n; i++)
8     niz2[i] = niz[i];
9
```

```
10    while (*rijec) {
11      for (i = 0; i < n; i++)
12        if (niz2[i] == *rijec) {
13          niz2[i] = '\0';
14          rijec++;
15          break;
16        }
17      if (i == n) break;
18    }
19
20    free(niz2);
21
22    return !*rijec;
23  }
```

Pomoćni niz `niz2` potreban nam je jer želimo mijenjati ulazni niz (pomoću pridruživanja u liniji 12 brišemo znak iz niza, da bismo spriječili višestruku upotrebu istog znaka), ali ne želimo da promjena afektira glavni program. Dapače, zbog ključne riječi `const`, prilikom poziva možemo navesti i konstantne stringove umjesto varijabli, na primjer: `is_perm("baobab", 6, "baba")`.

Funkcija radi tako da prolazi cijelim stringom `rijec`, te svaki njen znak traži u nizu `niz2`. Ako nađe znak, briše ga iz niza i "skače" na idući znak riječi, te prekida izvršavanje `for()` petlje.

U liniji 16 provjeravamo razlog prekidanja `for()`-petlje. Ako je `i == n`, to znači da smo iz `for()`-petlje izašli bez izvršavanja `break`-a u liniji 14, tj. bez da je uvjet u liniji 11 bio istinit. To znači da znak `*rijec` nismo našli u nizu `niz2`, pa prekidamo daljnje traženje.

Na kraju oslobađamo memoriju privremeno zauzetu kopijom niza, te vraćamo 1 ako je `*rijec` nul-karakter (`'\0'`; označava da smo uspješno prošli kroz cijelu riječ), a 0 ako je `*rijec` neki drugi znak (tj. ako smo traženje prekinuli pomoću `break` u liniji 16, što znači da znak `*rijec` nismo našli u nizu `niz2`).

U rješenju koristimo ekvivalentnost nizova i pointera. Tako naredba `rijec++` povećava pointer `rijec` nakon čega on pokazuje na znak iza znaka na koji je pokazivao prije povećavanja, dok `*rijec` daje znak na koji `rijec` pokazuje. Drugim riječima, stalno baratamo sa znakom `rijec[0]`, ali pomičemo početak stringa prema desno i na taj način (praktički, stalnim odbacivanjem prvog znaka stringa) prolazimo cijelim stringom.

Sama varijabla `rijec` (ne i `*rijec`!) je lokalna kopija istovrsnog pointera iz glavnog programa, pa ta promjena neće afektirati pozivnu varijablu. Kad bismo mijenjali `*rijec` (ili `rijec[j]` za neki j), onda bi se promijenio pozivni string (i ne bismo mogli koristiti ključnu riječ `const` kod deklaracije argumenta `rijec`). □

Zadatak 5.3.15. *Riješite prethodni zadatak tako da funkcija provjerava je li riječ* `rijec` *permutacija* **svih** *znakova iz niza* `niz`.

Uputa. Jedan je način provjerom na kraju jesu li svi znakovi niza `niz2` obrisani, ali može

i daleko jednostavnije. Što znate o odnosu duljine riječi i niza ako se traži da se iskoriste svi znakovi i to svaki točno jednom (ovo drugo je uvjet originalnog zadatka)?

Drugi način je sortiranje oba stringa, te usporedba jesu li oni jednaki. Takvo rješenje bi se moglo primijeniti i u prethodnom zadatku, uz manje modifikacije (ne traži se jednakost stringova, nego da je sortirana `rijec` sadržana u sortiranom `niz`-u. □

Zadatak 5.3.16 (Šlag na kraju)**.** *Napišite funkciju*
`int palindrom(char *s)`
koja vraća 1 *ako je string* `s` *palindrom; inače treba vratiti* 0*. Funkcija treba biti case-insensitive (tj. ne smije raditi razliku između velikih i malih slova), te mora ignorirati sve znakove koji nisu slovo. Pri tome ne smije mijenjati string* `s`*.*
Na primjer, string `"Ana voli: Milovana!"` *treba prepoznati kao palindrom.*

5.4 String-specifične funkcije

Većina funkcija specijaliziranih za rad sa stringovima nalaze se u biblioteci `string`. Nazivi tih funkcija počinju sa "str":

`strlen()` vraća duljinu stringa (engl. *string length*)

`strcpy()` kopira jedan string u drugi (engl. *string copy*)

`strcat()` lijepi jedan string na kraj drugog (engl. *string concatenate*)

`strcmp()`, `strcasecmp()` uspoređuju dva stringa (engl. *string compare*; "case" znači *case insensitive*)

Neke funkcije nalaze se i u drugim bibliotekama. Na primjer, `gets()` i `fgets()` se nalaze u biblioteci `stdio`.

Zadatak 5.4.1. *Napišite svoje verzije pobrojanih stringovnih funkcija, bez korištenja funkcija iz biblioteke* `string`*.*

Zadatak 5.4.2. *Napišite program koji učitava jednu riječ* `s` *duljine najviše* 19 *znakova i prirodni broj* `n`*. Program treba u varijabli* `s2` *kreirati najkraću riječ koja se sastoji od kopija riječi* `s` *"lijepljenih" jedna iza druge, tako da duljina stringa* `s2` *bude barem* `n`*. Na primjer, za riječ* `"Pero"` *i* `n=17`*,* `s2` *treba biti* `"PeroPeroPeroPeroPero"` *(string duljine* 20 *znakova, jer s jednom kopijom manje –* `"PeroPeroPeroPero"` *– ima samo* $16 <$ `n` *znakova).*

Rješenje. Maksimalna duljina stringa `s` je zadana i iznosi 19, pa njega možemo odmah deklarirati kao niz od 20 znakova (jedan više od 19 zbog završnog `'\0'`).

Kako nije zadana najveća moguća vrijednost za `n`, string `s2` je potrebno dinamički alocirati. Pri tome ne znamo unaprijed kolika je njegova duljina, ali ju možemo izračunati.

Naime, ako je n djeljiv s duljinom stringa `s`, onda je jasno da je n buduća duljina stringa `s2`. No, ako nije, onda je duljina stringa `s2` jednaka

$$\texttt{strlen(s)} \cdot \left\lceil \frac{n}{\texttt{strlen(s)}} \right\rceil,$$

što je jednako

$$\texttt{strlen(s)} \cdot \left\lfloor \frac{n}{\texttt{strlen(s)}} + 1 \right\rfloor$$

tj.

$$\texttt{strlen(s)} \cdot \left(\left\lfloor \frac{n}{\texttt{strlen(s)}} \right\rfloor + 1 \right)$$

(jer n nije djeljiv sa `(strlen(s))`). Dijeljenje je cjelobrojno, pa će rezultat odmah biti "najveće cijelo", što znači da je rezultatu dijeljenja dovoljno dodati 1 i pomnožiti ga sa `strlen(s)`.

Nakon računanja buduće duljine stringa `s2`, možemo alocirati prostor (jedno mjesto više od te duljine, zbog završnog znaka `'\0'`) i kreirati string `s2`. Prvo u njega pohranjujemo kopiju stringa `s` naredbom `strcpy(s2, s)`. Redoslijed argumenata lako možete zapamtiti jer je isti kao u notaciji `s2 = s` (koja je pograšna jer radi izjednačavanje pointera, a ne kopiranje niza znakova!).

Nakon toga "lijepimo" kopije stringa `s` na kraj stringa `s2` dok `s2` ne postigne željenu duljinu. To radimo naredbom `strcat(s2, s)`. Redoslijed argumenata funkcije je, kao i kod funkcije `strcpy(s2, s)`, prirodan jer podsjeća na `s2 += s` (što nije točno, ali intuitivno daje naslutiti što želimo).

Na kraju je potrebno osloboditi dinamički alociranu memoriju.

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4
5  int main (void) {
6    char s[20], *s2;
7    int n;
8
9    // Ucitavanje:
10   printf("Unesite rijec: "); scanf("%s", s);
11   printf("Unesite broj: "); scanf("%d", &n);
12
13   // Racunanje duljine strin\-ga s2:
14   if (n % strlen(s))
15     n = (n / strlen(s) + 1) * strlen(s);
16
17   // Alokacija memorije za s2:
18   s2 = (char*)malloc((n + 1) * sizeof(char));
19
```

```
20    // Kopiranje strin\-ga s u s2:
21    strcpy(s2, s);
22
23    // Lijepljenje strin\-ga s na kraj s2:
24    while (strlen(s2) < n) strcat(s2, s);
25
26    // Ispis rezultata:
27    printf("s2 = \"%s\"\n", s2);
28
29    // Oslobadjanje alocirane memorije:
30    free(s2);
31
32    return 0;
33  }
```

□

Zadatak 5.4.3. *Napišite program koji učitava string* `s` *(najviše* 17 *znakova), te kreira string* `s2` *koji se sastoji od kopija dijelova stringa* `s`*. Prva kopija treba biti cijela, druga bez prvog znaka, treća bez drugog znaka, itd. Na primjer, od stringa* `"Pero"` *treba proizvesti string* `"Peroeroroo"`*.*

Rješenje. Primijetimo da je `s` u ovom zadatku **string, a ne riječ**, što znači da ga ne možemo učitati pomoću formata `%s`, nego nam treba `%[^\n]`.

Ovaj zadatak ćemo riješiti slično prethodnom. Ako je duljina stringa `s` d, onda je duljina stringa `s2` jednaka

$$\sum_{i=1}^{d} i = \frac{d}{2}(d+1).$$

Duljinu možemo računati ili kao sumu ili preko prikazane formule. Ako računamo preko formule, treba paziti na redoslijed operacija, jer **NIJE isto** `n/2*(n+1)` i `n*(n+1)/2` (zašto?).

Kako dobiti string bez prvih i znakova (gdje je $i \in \{0, 1, \ldots, d-1\}$)? Prisjetimo se da je string niz znakova koji završavaju s `'\0'`, te da je niz ekvivalentan pointeru na prvi element niza. To znači da je string bez prvih i znamenaka ekvivalentan pointeru na znak s indeksom i, tj. `&s[i]`.

Duljinu stringa `s` držimo u varijabli `n` kako ne bismo stalno pozivali funkciju `strlen()` koja je relativno spora; duljinu stringa `s2` moramo izračunati i držati u varijabli `m` (nju ne možemo dobiti pomoću `strlen()` dok string `s2` nije do kraja kreiran!).

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4
5  int main (void) {
6    char s[18], *s2;
```

```
7   int m, n, i;
8
9   // Ucitavanje:
10  printf("Unesite string: "); scanf("%[^\n]", s);
11
12  // Racunanje duljine strin\-ga s2:
13  n = strlen(s);
14  m = n * (n + 1) / 2;
15  printf("Duljina s2: %d\n", m);
16
17  // Alokacija memorije za s2:
18  s2 = (char*)malloc((m + 1) * sizeof(char));
19
20  // Postavljanje vrijednosti strin\-ga s na prazni
21  // string:
22  *s2 = '\0'; // ili s2[0] = '\0';
23
24  // Lijepljenje strin\-ga s na kraj s2:
25  for (i = 0; i < n; i++) strcat(s2, &s[i]);
26
27  // Ispis rezultata:
28  printf("s2 = \"%s\"\n", s2);
29
30  // Oslobadjanje alocirane memorije:
31  free(s2);
32
33  return 0;
34 }
```

Umjesto "bacanja" stringa `s2` na prazni string (linija 22), mogli smo u njega kopirati vrijednost od `s`, ali bi onda `for()`-petlja u liniji 25 morala krenuti od `i=1`. □

Napomena 5.4.1. *Razlikujte prazni string i* `NULL`*-pointer! Prazni string je niz od barem jednog znaka kojem je prvi znak jednak* `'\0'`*, dok* `NULL`*-pointer možemo promatrati kao niz duljine nula. Pristupanje bilo kojem elementu tog "niza" će srušiti program! Kako elementima niza pristupaju i osnovne stringovne funkcije, to će i njihova upotreba na* `NULL`*-pointeru također srušiti program.*

Zadatak 5.4.4. *Riješite prethodni zadatak tako da kopije koje lijepite na kraj budu jednake stringu* `s` *bez prvih i znakova* **za parne** *i. Na primjer, od stringa* `"Perica"` *treba dobiti riječ* `"Pericaricaca"`*.*

Zadatak 5.4.5. *Napišite program koji učitava riječ* `s` *od najviše* 20 *znakova, te stvara novu riječ* `s2` *koja se sastoji od invertirane riječi* `s` *iza koje nalijepite originalnu riječ* `s`*. Na primjer, od riječi* `"Pero"` *treba dobiti riječ* `"orePPero"`*.*
Napomena: *String* `s` *na kraju izvršavanja programa mora biti nepromijenjen!*
Uputa: *Napišite funkciju za invertiranje stringa.*

Rješenje. Ovdje će string `s2` biti dvostruko veći od stringa `s`, što znači da ga možemo deklarirati kao string od $2 \cdot 20 = 40$ znakova (tj. niz od $2 \cdot 20 + 1 = 41$ znaka).

String `s` ćemo prvo kopirati u pomoćni string `s2`, pa tek njega invertirati. Na taj način imat ćemo i originalni (`s`) i invertirani (`s2`) string, bez mijenjanja originalnog stringa `s`.

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

void invertiraj(char *s) {
  int i, n = strlen(s);
  for (i = 0; i < n/2; i++) {
    char c = s[i];
    s[i] = s[n - 1 - i];
    s[n - 1 - i] = c;
  }
}

int main (void) {
  char s[21], s2[41];

  printf("Unesite rijec: "); scanf("%s", s);
  strcpy(s2, s);
  invertiraj(s2);
  strcat(s2, s);
  printf("s2 = \"%s\"\n", s2);

  return 0;
}
```

□

Zadatak 5.4.6. *Riješite prethodni zadatak tako da za* `s2` *alocirate najmanju potrebnu količinu memorije i bez korištenja funkcija iz biblioteke* `string`*.*

Zadatak 5.4.7. *Napišite funkciju koja kao argumente uzima stringove* `s1`*,* `s2` *i* `s3`*. Funkcija treba slijepiti stringove* `s2` *i* `s3` *tako da prvo ide leksikografski manji od njih (treba razlikovati velika i mala slova), te rezultat treba pospremiti u string* `s1` *(dakle, stringovi* `s2` *i* `s3` *moraju ostati nepromijenjeni). Pretpostavite da je za string* `s1` *alocirano dovoljno memorije.*
Napišite i program kojim testirate funkciju.

Rješenje. Novost u ovom zadatku je usporedba stringova, za koju se koristi funkcija `strcmp()` (ili `strcasecmp()` ako želite zanemariti razliku između velikih i malih slova). Jednostavan način za upamtiti ponašanje funkcije:

$$\texttt{strcmp(s1, s2)} \begin{matrix} < \\ = \\ > \end{matrix} 0 \Leftrightarrow s1 \begin{matrix} < \\ = \\ > \end{matrix} s2.$$

Naravno, desna strana ekvivalencije je samo simbolički zapis; stringove ne možemo uspoređivati relacijskim operatorima jer bi to bila usproedba pointera, a ne nizova znakova na koje oni pokazuju.

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

void min(char *s1, char *s2, char*s3) {
  if (strcmp(s2, s3) < 0) {
    strcpy(s1, s2);
    strcat(s1, s3);
  } else {
    strcpy(s1, s3);
    strcat(s1, s2);
  }
}

int main (void) {
  char s1[41], s2[21], s3[21];

  printf("Unesite dvije rijeci: ");
  scanf("%s %s", s2, s3);
  min(s1, s2, s3);

  // Ispis rezultata:
  printf("s1 = \"%s\"\n", s1);
  printf("s2 = \"%s\"\n", s2);
  printf("s3 = \"%s\"\n", s3);

  return 0;
}
```

□

Napomena 5.4.2. *Funkcija* `strcasecmp()` *nije dostupna u Microsoftovom Visual C++. Umjesto nje, tamo se koriste nestandardne funkcije* `strcmpi()` *i* `stricmp()`.

Zadatak 5.4.8. *Prepravite rješenje prethodnog zadatka tako da prilikom usporedbe ignorira razliku velikih i malih slova (te provjerite ispravnost rješenja izvršavanjem na računalu).*

5.5 Nizovi stringova

U osnovi, niz stringova je dvodimenzionalno polje znakova. Ipak, deklaracije mogu ispadati relativno zbunjujuće i komplicirane (slično poglavlju 3). Zbog toga je ponekad praktično definirati novi tip podataka. Na primjer,

`int x;`
deklarira **varijablu** `x` tipa `int`, dok
`typedef int x;`
definira **tip** `x` koji je isto što i `int`. Želimo li definirati poseban tip za stringove duljine, na primjer, najviše dvadeset znakova, možemo definirati:
`typedef char moj_string[21];`
Nakon definicije tipa, uredno možemo deklarirati varijable novog tipa:
`moj_string s;`
što je ekvivalentno
`char s[21];`
Ovo je posebno korisno kod nekih zbunjujućih deklaracija, kao i kod deklaracija koje se često ponavljaju (npr. kod lista, u poglavlju 7).

Niz od, na primjer, 10 stringova s po najviše 20 znakova deklariramo na sljedeći način:
`char s[10][21];`
Ovo može zbuniti, jer je lako pomiješati redoslijede. Ako niste sigurni kako deklarirati takvu varijablu, pomozite si upravo pomoćnim tipom:

- `typedef char mojstring[21]`
  tip za jedan string od najviše 20 znakova; obično se navodi prije svih funkcija i globalnih varijabli

- `moj_string s[10]`
  varijabla `s` je niz od 10 varijabli tipa `moj_string`, tj. od 10 stringova s po najviše 20 znakova

Zadatak 5.5.1. *Napišite program koji učitava prirodni broj $n \leq 17$, te niz od n riječi s po najviše 19 znakova. Niz treba sortirati uzlazno (neovisno o velikim/malim slovima) i ispisati.*

Rješenje. Primjenjujemo klasični sort, uz usporedbu stringova pomoću `strcasecmp()`.

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

typedef char moj_string[20];

int main (void) {
  moj_string s[17];
  int n, i, j;

  // Ucitavanje niza rijeci
  printf("Koliko rijeci? "); scanf("%d", &n);
  for (i = 0; i < n; i++) {
    printf("Rijec %d: ", i + 1);
    scanf("%s", s[i]);
  }
```

```
16
17    // Sortiranje niza stringova uzlazno
18    for (i = 0; i < n - 1; i++)
19      for (j = i + 1; j < n; j++)
20        if (strcasecmp(s[i], s[j]) > 0) {
21          moj_string t;
22          strcpy(t, s[i]);
23          strcpy(s[i], s[j]);
24          strcpy(s[j], t);
25        }
26
27    // Ispis niza stringova
28    for (i = 0; i < n; i++)
29      printf("%s ", s[i]);
30    printf("\n");
31
32    return 0;
33  }
```

Umjesto uvođenja pomoćnog tipa `moj_string`, mogli smo i direktno deklarirati niz stringova:

```
7     char s[17][20];
```

...

```
21          char t[20];
```

□

Napomena 5.5.1. *U rješenju prethodnog zadatka, stringove zamjenjujemo kopiranjem sadržaja. Zamjenu smo mogli napraviti i zamjenom pointera, što je prikazano na predavanjima, no pri tome bi elementi niza morali biti pokazivači na znakove (*`char *`*), a ne nizovi znakova (slično kao u zadatku 4.4.10).*

Zadatak 5.5.2. *Napišite program koji učitava prirodni broj n, te niz od n riječi sa po strogo manje od* 17 *znakova. Niz treba sortirati silazno (poštujući razliku velikih i malih slova), usporedbom stringova bez prvog slova (dakle, string* `"Pero"` *dolazi iza stringa* `"Ivan"` *jer je* `"van"` *leksikografski veće od* `"ero"`*, a niz sortiramo silazno). Na kraju je potrebno ispisati sortirani niz.*

Rješenje. Zadatak rješavamo kombinacijom već viđenih rješenja (deklaracija i učitavanje niza riječi, upotreba stringa bez prvih k znakova, sort i ispis). Niz ćemo deklarirati preko pomoćnog tipa, zbog lakšeg baratanja s funkcijom `malloc()`.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4
```

```
5  typedef char moj_string[17];
6
7  int main (void) {
8    moj_string *s;
9    int n, i, j;
10
11   // Ucitavanje n
12   printf("Koliko rijeci? "); scanf("%d", &n);
13
14   // Alociranje memorije za n stringova od
15   // najvise 16 (strogo manje od 17) znakova
16   s = (moj_string*)malloc(n * sizeof(moj_string));
17
18   // Ucitavanje rijeci
19   for (i = 0; i < n; i++) {
20     printf("Rijec %d: ", i + 1);
21     scanf("%s", s[i]);
22   }
23
24   // Sortiranje niza stringova uzlazno
25   for (i = 0; i < n - 1; i++)
26     for (j = i + 1; j < n; j++)
27       if (strcmp(&s[i][1], &s[j][1]) < 0) {
28         moj_string t;
29         strcpy(t, s[i]);
30         strcpy(s[i], s[j]);
31         strcpy(s[j], t);
32       }
33
34   // Ispis niza stringova
35   for (i = 0; i < n; i++)
36     printf("%s ", s[i]);
37   printf("\n");
38
39   free(s);
40   return 0;
41 }
```

Usporedbu smo mogli izvesti i ovako:

```
27       if (strcmp(s[i] + 1, s[j] + 1) < 0) {
```

Pokušajte objasniti zašto! □

Zadatak 5.5.3. *Napišite program koji učitava prirodni broj n, te niz od n riječi sa po strogo manje od* 20 *znakova. Niz treba sortirati silazno (poštujući razliku velikih i malih slova), usporedbom stringova prema zadnjem znaku (dakle, string* `"Pero"` *dolazi prije*

stringa `"Ivan"` *jer je znak* `'o'` *leksikografski veći od znaka* `'n'`*, a niz sortiramo silazno). Na kraju je potrebno ispisati sortirani niz.*

Uputa. Kako, pomoću funkcije `strlen()`, možemo dobiti posljednji znak stringa (onaj koji je neposredno prije `'\0'`)? □

Zadatak 5.5.4. *Riješite prethodni zadatak dinamičkom alokacijom svakog pojedinog stringa, uz upotrebu minimalno potrebne memorije.*

Uputa. Riječi je potrebno učitavati u pomoćnu varijablu (niz od 21 znaka), zatim alocirati dovoljno memorije (jedno mjesto više nego je učitana riječ dugačka), te na kraju u alociranu memoriju kopirati učitanu riječ. Deklariranje, alociranje i delociranje nepravokutnih višedimenzionalnih polja objašnjeno je u poglavlju 4.3. □

Zadatak 5.5.5. *Na popločanoj cesti nalazi se skočimiš koji se kreće s lijeva na desno. U svakom skoku, on može preskočiti neki (cjelobrojni) broj ploča. Prilikom doskoka on stavlja oznaku (jedan* `char`*) na polje na koje je sletio (prvo polje je označeno sa "X"). Napišite program koji učitava cijele brojeve $k \in \mathbb{N}$ dok ne učita nepozitivni broj ($x \leq 0$), te za svaki x učitava jedan znak i njime označava cestu. Na kraju treba nacrtati cestu.*

Rješenje. Očito, naša cesta je niz znakova koji ima unaprijed neodređeni broj ploča. To znači da taj niz treba produljivati u svakom koraku, te na predzadnje mjesto upisivati traženi znak (a prije njega treba popuniti razmacima koji predstavljaju prazne ploče). Nakon što smo gotovi, na zadnje mjesto u nizu upisujemo znak `'\0'`, kako bi niz znakova postao regularni string i ispravno se ispisao.

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

int main (void) {
  char *s;
  int n = 0, i, x;
  char znak;

  // Alociramo dva mjesta: za 'X' s kojeg
  // skocimis krece i za '\0' kao kraj strin\-ga
  s = (char*)malloc(2 * sizeof(char));
  strcpy(s, "X"); // ili s[0]='X'; s[1]='\0';

  while (1) {
    // Ucitavanje podataka o skoku:
    printf("Za koliko cemo skociti? ");
    scanf("%d", &x);
    if (x <= 0) break;
    printf("Koji znak cemo ostaviti? ");
    scanf(" %c", &znak);
```

```
22      // Realociranje niza znakova
23      s = (char*)realloc(s, (n + x + 2) * sizeof(char));
24      // Oznacavanje praznih ploca
25      for (i = n+1; i < n + x; i++)
26        s[i] = ' ';
27      // Oznacavanje doskocne ploce
28      s[n+=x] = znak;
29    }
30    // "Zatvaranje" strin\-ga
31    s[n+1] = '\0';
32
33    printf("Cesta:\n'%s'\n", s);
34
35    free(s);
36
37    return 0;
38  }
```

Primijetimo da tijekom izvršavanja `while`-petlje niz znakova `s` nije ispravan string (jer nije terminiran znakom `'\0'`). To je u redu, jer ga i koristimo isključivo kao niz znakova. Tek nakon petlje, kad želimo "nacrtati cestu", potrebno nam je da `s` bude pravi string, pa zato i terminiramo string iza `while`-petlje (u liniji 31). □

Zadatak 5.5.6. *Napišite funkciju koja kao argument uzima prirodni broj n, te kreira string (uz potrebnu alokaciju memorije) u koji će pospremiti niz sličan onome iz zadatka 2.3.8, s time da umjesto nula treba staviti točkicu, a umjesto jedinica treba staviti zvjezdicu. Funkcija treba vratiti string (tj. pointer na prvi znak stringa).*
Napišite i program kojim testirate funkciju.

Zadatak 5.5.7 (Šlag na kraju)**.** *Modificirajte rješenje zadatka 5.5.5 tako da skočimiš može skakati i prema lijevo ($x < 0$) i prema desno ($x > 0$), a upis se prekida kad ne skoči nigdje ($x = 0$). Pri tome pazite da skočimiš može otići i ljevije od početne ploče, pri čemu – uz rekolaciju – treba postojeće elemente niza pomaknuti u desno.*

Ako skočimiš skoči dva (ili više) puta na istu ploču, novi znak stavlja preko starog, tj. stari znak možete "zaboraviti".

Poglavlje 6

Strukture

U nizove možemo pospremiti više vrijednosti istog tipa. Slično tome, u jednu strukturu možemo pospremiti više vrijednosti različitog tipa. Na primjer, možemo sastaviti strukturu koja sadrži ime, inicijal prezimena, starost osobe i broj cipela koje nosi. Smjestimo li te podatke u niz, imamo popis traženih podataka za određenu populaciju. Kombinirano sa snimanjem u datoteku, dobit ćemo pravu malu bazu podataka.

Strukture definiramo:

1. kao nove tipove, pomoću `typedef`:

   ```
   typedef struct {
     char ime[20];
     char inicijal;
     int starost;
     int br_cipela;
   } osoba;
   ```

   Varijablu tipa `osoba` možemo deklarirati ovako:

   ```
   osoba x;
   ```

2. kao nove tipove, bez `typedef`:

   ```
   struct osoba {
     char ime[20];
     char inicijal;
     int starost;
     int br_cipela;
   };
   ```

   ali onda deklaracije varijabli treba raditi ovako:

```
struct osoba x;
```

3. bez deklaracije tipa:

```
struct {
  char ime[20];
  char inicijal;
  int starost;
  int br_cipela;
} x;
```

Moguća je i kombinacija načina 1 i 2 koja će biti posebno korisna u poglavlju 7:

```
typedef struct _osoba {
  char ime[20];
  char inicijal;
  int starost;
  int br_cipela;
} osoba;
```

Varijable deklariramo ili sa

```
struct _osoba x;
```

ili sa

```
osoba x;
```

Napomena 6.1.2 (Česta greška)**.** *Nazivi "*`_osoba`*" i "*`osoba`*" ne moraju biti u nikakvoj vezi, ali* **moraju biti međusobno različiti** *(kao i svi ostali identifikatori (tipovi, varijable, funkcije,...) u programu)!*

Običaj je koristiti sličan naziv, uz dodatak "`_`*" ispred naziva, zbog preglednosti: čovjeku je jasno da su to skoro iste stvari, a računalo ipak dobija različite nazive.*

Elementima strukture pristupamo upotrebom operatora "točka" (`.`), te ih tretiramo kao obične varijable:

```
osoba x;
x.br_cipela = 17;
scanf("%s", x.ime);
scanf("%c", &x.inicijal);
```

Zadatak 6.1.8. *Napišite program koji učitava podatke o dvije osobe (deklarirane kao u prethodnom paragrafu), zamjenjuje te podatke (klasični "swap") i ispisuje ih.*

Rješenje. Napisat ćemo pomoćne funkcije za učitavanje i ispis podataka. Pri tome treba paziti da `struct` sadrži varijable, što znači da promjena nekog polja strukture znači i promjenu same strukture, pa funkcija za učitavanje mora primati pointer na varijablu tipa `osoba`.

Podsjetnik: prilikom učitavanja znaka, ispred formata `%c` potrebno je staviti razmak (zašto?).

```
#include <stdio.h>

typedef struct _osoba {
  char ime[20];
  char inicijal;
  int starost;
  int br_cipela;
} osoba;

void ucitaj(osoba *o) {
  printf("  Ime: ");
  scanf("%s", (*o).ime);
  printf("  Inicijal prezimena: ");
  scanf(" %c", &((*o).inicijal));
  printf("  Starost: ");
  scanf("%d", &((*o).starost));
  printf("  Broj cipela: ");
  scanf("%d", &((*o).br_cipela));
}

void ispisi(osoba o) {
  printf("  Ime: %s\n", o.ime);
  printf("  Inicijal prezimena: %c\n", o.inicijal);
  printf("  Starost: %d\n", o.starost);
  printf("  Broj cipela: %d\n", o.br_cipela);
}

int main (void) {
  osoba osoba1, osoba2, temp;

  // Ucitavanje:
  printf("Osoba 1:\n");
  ucitaj(&osoba1);
  printf("Osoba 2:\n");
  ucitaj(&osoba2);

  // Swap:
  temp = osoba1;
  osoba1 = osoba2;
```

```
40    osoba2 = temp;
41
42    // Ispis:
43    printf("Osoba 1:\n");
44    ispisi(osoba1);
45    printf("Osoba 2:\n");
46    ispisi(osoba2);
47
48    return 0;
49  }
```

□

Napomena 6.1.3. *Pointeri na strukture su jako često korišteni u C-u. Pri tome može doći i do ugnježdavanja, što lako može dovesti do sintaktičkih zavrzlama poput* `(*(*(*a).b).c).d).e`
U tu svrhu definiran je operator `->` *koji služi kao zamjena za kombinaciju dereferenciranja (operator* `*`*) i pristupanja elementu strukture (operator* `.`*):*

$$\texttt{(*x).y} \Leftrightarrow \texttt{x->y}$$

Na ovaj način, prethodno spomenutu "zavrzlamu" možemo napisati ovako: `a->b->c->d->e`

Primjenom operatora `->`, funkcija `ucitaj()` iz rješenja prethodnog zadatka postaje puno preglednija:

```
10  void ucitaj(osoba *o) {
11    printf("  Ime: ");
12    scanf("%s", o->ime);
13    printf("  Inicijal prezimena: ");
14    scanf(" %c", &o->inicijal);
15    printf("  Starost: ");
16    scanf("%d", &o->starost);
17    printf("  Broj cipela: ");
18    scanf("%d", &o->br_cipela);
19  }
```

Kod učitavanja pojedinih polja, operator `&` se odnosi na samo polje (npr. `o->inicijal`), a ne na samu varijablu `o`! Zbog toga u prvom `scanf()` nema operatora `&` (jer je `o->ime` string).

Napomena 6.1.4 (Česta greška)**.** *Operatori* `->` *i* `.` *nisu ekvivalentni i ne mogu upotrebljavati kao zamjena jedan drugome. Preciznije, operator* `->` *se može primjenjivati isključivo na pointerima na strukture, dok se operator* `.` *može primjenjivati isključivo na strukture!*

Na primjer, sljedeće je **pogrešno***:*

```
osoba x, *y;
...
x->broj_cipela = 19;
y.starost = 17;
```

Ispravno je:

```
osoba x, *y;
...
x.broj_cipela = 19;
y->starost = 17;
```

Zadatak 6.1.9. *Deklarirajte tip (strukturu) za pohranu jednog kompleksnog broja, te napišite funkcije za zbrajanje, množenje, konjugiranje i ispis kompleksnih brojeva. Napišite i program za testiranje napisanih funkcija.*

Rješenje. Napisat ćemo samo funkcije za množenje i ispis; zbrajanje i konjugiranje ($x + iy \mapsto x - iy$) napišite samostalno.

```
#include <stdio.h>

typedef struct {
  double x, y;
} complex;

complex complex_multiply(complex x, complex y) {
  complex res = {
    x.x * y.x - x.y * y.y,
    x.y * y.x + x.x * y.y
  };
  return res;
}

void complex_print(complex x) {
  if (x.y < 0)
    printf("%g + i * (%g)", x.x, x.y);
  else
    printf("%g + i * %g", x.x, x.y);
}

int main (void) {
  complex a = {1, 2}, b = {-3.1, 2.7};

  printf("(");
  complex_print(a);
  printf(") * (");
  complex_print(b);
```

```
29    printf(") = ");
30    complex_print(complex_multiply(a, b));
31    printf("\n");
32
33    return 0;
34  }
```

U linijama 8 i 23 vidimo inicijalizacije slične onima koje se provode na nizovima. Za razliku od nizova, kod struktura moramo navesti sve vrijednosti. Takvo pridruživanje radi isključivo prilikom deklaracije varijabli, ali ne i kasnije u programu. Dakle, ovo

```
return {x.x * y.x - x.y * y.y, x.y * y.x + x.x * y.y};
```

i

```
complex res;
res = {x.x * y.x - x.y * y.y, x.y * y.x + x.x * y.y};
```

bi bilo **POGREŠNO** i compiler bi javio grešku poput:

```
t.c: In function complex_multiply:
t.c:11: error: expected expression before { token
```

Također, iza "`}`" treba biti točka-zarez jer ovdje to nije oznaka kraja bloka, nego dio vrijednosti varijable! □

Zadatak 6.1.10. *Riješite prethodni zadatak bez upotrebe* `return` *(tj. uz vraćanje vrijednosti iz funkcije preko varijabilnih argumenata).*

Zadatak 6.1.11. *Deklarirajte tip podatka u kojem ćete držati naziv jednog automobila (najviše 30 znakova) i njegovu cijenu (cijeli broj). Napišite program koji učitava prirodni broj $n \in \mathbb{N}$, te podatke o n automobila. Program treba sortirati niz automobila padajuće po cijeni, te ispisati tako sortirane automobile i njihove cijene.*

Rješenje. U zadatku nije zadan najveći mogući broj automobila, pa je potrebno dinamički alocirati niz, što se sa strukturama radi na jednak način kao sa cijelim brojevima i znakovima.

Ovaj put, prilikom učitavanja naziva, potrebno je ispred formata `%[^\n]` staviti razmak (zašto?).

Sort je jednak kao sort cijelih brojeva "po nekom kriteriju". Pri tome treba paziti da u usporedbi (`if()` u liniji 27) uspoređujemo ono po čemu sortiramo (ovdje je to cijena automobila: `auti[·].cijena`), ali zamjenjujemo **cijele** strukture (**NE samo cijene!**).

Ispis ćemo malo "ukrasiti", da izgleda tablično. Riječ je o običnom igranju s formatima (broj znači u koliko mjesta želimo ispisati vrijednost, poravnato na desno; negativni broj označava lijevo poravnavanje).

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
```

```
4  typedef struct {
5    char naziv[31];
6    int cijena;
7  } automobil;
8
9  int main (void) {
10   automobil *auti;
11   int n, i, j;
12
13   // Ucitavanje (s alokacijom memorije):
14   printf("Koliko automobila? ");
15   scanf("%d", &n);
16   auti = (automobil*)malloc(n * sizeof(automobil));
17   for (i = 0; i < n; i++) {
18     printf("Automobil %d:\n  Naziv: ", i + 1);
19     scanf(" %[^\n]", auti[i].naziv);
20     printf("  Cijena: ");
21     scanf("%d", &auti[i].cijena);
22   }
23
24   // Sort:
25   for (i = 0; i < n - 1; i++)
26     for (j = i + 1; j < n; j++)
27       if (auti[i].cijena < auti[j].cijena) {
28         automobil temp = auti[i];
29         auti[i] = auti[j];
30         auti[j] = temp;
31       }
32
33   // Ispis:
34   printf("%-30s Cijena\n", "Naziv");
35   for (i = 0; i < 37; i++) printf("-");
36   printf("\n");
37   for (i = 0; i < n; i++)
38     printf(
39       "%-30s %6d\n",
40       auti[i].naziv,
41       auti[i].cijena
42     );
43
44   // Oslobadjanje memorije
45   free(auti);
46
47   return 0;
48 }
```

□

Napomena 6.1.5 (Česte **greške**). *Kod sortiranja struktura, česte su sljedeće greške:*

- *Uspoređivanje samih struktura:*

```
27        if (auti[i] < auti[j]) {
```

Ovo je **POGREŠNO** *jer nije definiran uređaj između dva* `struct`*-a i računalo ne može pogoditi što mi želimo. Mogli smo, na primjer, tražiti sort po nazivima, a ne po cijeni.*
Strukture ne možemo direktno uspoređivati čak niti ako sadrže samo jedno polje!

- *Zamjena dijelova struktura:*

```
28            int temp = auti[i].cijena;
29            auti[i].cijena = auti[j].cijena;
30            auti[j].cijena = temp;
```

Ovo je **POGREŠNO** *jer zamjenjujemo samo cijene automobila, dok nazivi ostaju gdje su bili. Rezultat toga bi bio da cijene budu navedene uz automobile kojima ne pripadaju.*

- *Razne* **BESMISLENE** *sintakse:*
`auti.cijena[i]`
`cijena.auti[i]`
`auti[i]->cijena`
...
Ako se želi pristupiti polju `cijena` *strukture* `auti[i]`*, onda je jedini ispravni način za to napraviti "ulazak" u strukturu pomoću operatora* `.` *te zatim navođenje imena polja kojem se pristupa:* `auti[i].cijena`*!*

Zadatak 6.1.12. *Riješite prethodni zadatak tako da nazivi automobila imaju najviše* 37 *znakova, te da se sort vrši uzlazno prema nazivu.*

Zadatak 6.1.13. *Napišite program koji ima učitavanje po uzoru na zadatak 6.1.11, ali ispisuje (nesortirano!) sve automobile (s cijenom) koji imaju cijenu manju od prosječne.*

Zadatak 6.1.14. *Riješite zadatak 5.5.5 tako da ne pamtite "cestu" (koja može biti jako dugačka, pogotovo za dugačke skokove), nego da pamtite "povijest skokova" (dakle, niz koji "raste" pomoću* `realloc()`*, a u kojem su sadržane pozicije na cesti i znakovi).*

Zadatak 6.1.15 (Šlag na kraju). *Riješite zadatak 5.5.7 tako da ne pamtite "cestu" (koja može biti jako dugačka, pogotovo za dugačke skokove), nego da pamtite "povijest skokova" (dakle, niz koji "raste" pomoću* `realloc()`*, a u kojem su sadržane pozicije na cesti i znakovi).*

Uputa. Ovdje je rješenje jednostavnije nego u slučaju zadatka 5.5.7. Naime, učitavanje se radi jednako kao i u zadatku 6.1.14, a nakon učitavanja treba sortirati niz skokova prema položaju i onda složiti ispis.

Pripazite na situacije u kojima različiti skokovi vode na istu ploču! □

Poglavlje 7

Vezane liste

Niz struktura možemo jednostavno zamijeniti s više nizova. Na primjer, umjesto niza od 20 automobila (definiranih u zadacima u prethodnom poglavlju):

```
automobil auti[20];
```

možemo definirati nizove naziva i cijena:

```
int cijene[20];
char nazivi[20][31];
```

Prednosti struktura u ovakvim zadacima su organizacijske (preglednost, manje kôda kod nekih radnji), ali tu strukture nisu neophodne. Njihova prava primjena dolazi kod raznih dinamičkih struktura poput jednostruko i dvostruko vezanih lista, raznih stabala i sl.

Lista služi za pohranu istovrsnih elemenata u nekakav niz (**NE** u C-ovskom smislu riječi "niz"!), najčešće unaprijed neodređene duljine. Iako za tu svrhu možemo upotrijebiti i dinamičke nizove (poglavlje 4.2), liste imaju nekoliko prednosti:

disperziranost u memoriji Niz, bez obzira na to kako je zadan, zauzima blok memorije, što znači da ga nije moguće alocirati ako je memorija jako fragmentirana

brze operacije Ubacivanje novih i brisanje starih elemenata izvršava se u vremenu koje ne ovisi o duljini liste[1] (kod nizova je to vrijeme linearno ovisno o duljini niza).

Jedan element liste je struktura koja sadrži tzv. "korisni podatak" (ili podatke), te adresu idućeg elementa u listi. Na primjeru automobila, jedan element liste bi izgledao ovako:

```
typedef struct _automobil {
  int cijena;
```

[1]Da bi ubacivanje i brisanje zaista imali konstantnu složenost, potrebno je imati pokazivač na element koji prethodi onom kojeg brišemo ili ispred kojeg dodajemo novi element. Ovo je tehnički detalj, koji se u praksi rješava (ovisno o potrebama programa) ili upotrebom pomoćne varijable ili implementacijom dvostruko vezane liste.

```
  char naziv[31];
  struct _automobil *next;
} automobil;
```

Ovdje je nužno zadati pomoćno ime tipa `struct _automobil`, jer se unutar `struct`-a ne vidi tip `automobil` (koji je definiran **nakon** samog `struct`).

Zadatak 7.1.16. *Napišite program koji učitava listu automobila (za svakoga treba učitati naziv (string do* 30 *znakova) i cijenu (cijeli broj)). Program treba ispisati one automobile (naziv i cijenu) koji su skuplji od prosjeka.*

Rješenje. Učitavanje liste bitno se razlikuje od učitavanja niza. Za početak, ne postoji tip podataka "lista", nego listu sami kreiramo kao sturkture povezane pointerima. Zbog toga alociramo svaki zasebni element (za razliku od niza gdje su se sve alokacije i realokacije vršile na cijelom nizu).

Nadalje, potrebno je razlikovati učitavanje (preciznije: dodavanje u listu) prvog elementa i svih ostalih elemenata. Naime, ako je lista prazna (tj. njen prvi element ne postoji, pa u varijabli `first` imamo `NULL`), onda je potrebno upravo novi element proglasiti prvim:

```
first = new;
```

U protivnom, novi element je sljedbenik dosadašnjeg zadnjeg:

```
pom->next = new;
```

U oba slučaja, novi element liste postaje novi zadnji element, pa je zato dodano "`pom =`" u linije 25 i 27.

Ne postoji jednostavan način da se utvrdi ukupni broj elemenata liste, no moguće ga je izračunati, "trčanjem" po cijeloj listi. Krećemo od prvog elementa (`first`) i skačemo na idućeg dok ne dođemo do kraja (`NULL`): linije 37–40.

Ispis radimo slično kao i brojanje elemenata: linije 50–52.

Na kraju, listu treba obrisati element po element: linije 55–58.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <ctype.h>
4
5  typedef struct _automobil {
6    char naziv[31];
7    int cijena;
8    struct _automobil *next;
9  } automobil;
10
11 int main (void) {
12   automobil *first = NULL, *pom, *pom2;
13   int prosjek = 0, n = 0;
14   char c;
15
16   // Ucitavanje liste
```

```
17    do {
18      automobil *new;
19      new = (automobil*)malloc(sizeof(automobil));
20      printf("Naziv automobila: ");
21      scanf(" %[^\n]", new->naziv);
22      printf("Cijena automobila: ");
23      scanf("%d", &new->cijena);
24      if (first)
25        pom = pom->next = new;
26      else
27        pom = first = new;
28      do {
29        printf("Zelite li nastaviti ucitavanje ");
30        printf("(d/n)? ");
31        scanf(" %c", &c);
32        c = tolower(c);
33      } while (c != 'd' && c != 'n');
34    } while (c == 'd');
35    pom->next = NULL;
36
37    // Racunanje prosjecne cijene i duljine liste
38    for (pom = first; pom; pom = pom->next) {
39      prosjek += pom->cijena;
40      n++;
41    }
42    prosjek /= n;
43
44    // Pomocni ispis
45    printf("Ukupno automobila: %d\n", n);
46    printf("Prosjecna cijena: %d\n", prosjek);
47
48    // Ispis auta koji su skuplji od prosjeka
49    printf("Natprosjecno skupi automobili:\n");
50    for (pom = first; pom; pom = pom->next)
51      if (pom->cijena > prosjek)
52        printf("  %s (%d)\n", pom->naziv, pom->cijena);
53
54     // Brisanje cijele liste
55    for (pom = first; pom; pom = pom2) {
56      pom2 = pom->next;
57      free(pom);
58    }
59
60    return 0;
61  }
```

□

Napomena 7.1.6 (Česta greška)**.** *Brisanje* **cijele** *liste treba izvesti oprezno. Potrebno je brisati element po element, no pri tome ne smijemo pristupati obrisanim elementima. Dakle, ovo bi bilo* **pogrešno***:*

```
for (pom = first; pom != NULL; pom = pom->next)
  free(pom);
```

jer u trenutku kad izvršavamo inkrement (`pom = pom->next`*), element* `pom` *je već obrisan (tj. njegova memorija je oslobođena i više joj ne smijemo pristupati).*

Napomena 7.1.7. *Čak i da je broj elemenata liste bio poznat unaprijed (npr. da prvo pitamo koliko će automobila biti), memoriju moramo alocirati za svaki element posebno (dakle, n poziva funkcije* `malloc()`*, a nikako ne samo jedan, kao kod nizova čiju duljinu unaprijed znamo)!*

Zadatak 7.1.17. *Definirajte tip podatka* `razlomak` *za pohranu jednog razlomka u listu (tako bude moguće deklarirati varijablu* `razlomak r;`*), te napišite funkciju koja dodaje jedan razlomak (određen parametrima* `x` *i* `y`*) na početak liste razlomaka. Napišite i kako se funkcija poziva.*

Rješenje. Pošto se traži mogućnost deklaracije `razlomak r;`, moramo koristiti `typedef` (što se i inače preporuča). Bez toga, jedini mogući način deklaracije bio bi upotrebom ključne riječi `struct` ispred tipa `razlomak`, ovako: `struct razlomak r;`.

Prilikom dodavanja elementa na početak liste, mijenja se adresa prvog elementa u listi, pa funkcija mora vratiti novu adresu početka liste (tj. adresu novog elementa liste). Tu adresu treba pospremiti u istu varijablu u kojoj se prije poziva funkcije nalazi početak liste.

```
1  typedef struct _razlomak {
2    int br, naz;
3    struct _razlomak *next;
4  } razlomak;
5
6  razlomak *insert(razlomak *first, int x, int y) {
7    razlomak *new = (razlomak*)malloc(sizeof(razlomak));
8    new->br = x;
9    new->naz = y;
10   new->next = first;
11   return new;
12 }
```

Poziv:

```
first = insert(first, br, naz);
```

gdje je `br` brojnik, a `naz` nazivnik novog razlomka. U početku, varijablu `first` treba inicijalizirati

`razlomak *first = NULL;`

kako bi predstavljala praznu listu. □

Zadatak 7.1.18. *Napišite funkciju* `delete()` *koja kao argumente uzima pointer na početak liste razlomaka (definiranih kao u zadatku 7.1.17), te brojnik i nazivnik razlomka. Funkcija treba iz liste obrisati* **prvi** *razlomak koji ima jednaku vrijednost kao razlomak definiran brojnikom i nazivnikom (argumenti funkcije).*

Rješenje. Moguće je da funkcija za brisanje obriše prvi element liste. U tom slučaju, promijenit će se početak liste, pa funkcija za brisanje također treba vraćati novi (ili stari) početak liste. Zbog toga je potrebno i razlikovati slučajeve "obriši prvi element liste" i "nađi element liste (koji nije prvi) i obriši ga".

```
1  razlomak *delete(razlomak *first, int x, int y) {
2    razlomak *t;
3    if (!first) return NULL;
4    if (first->br * y == first->naz * x) {
5      t = first->next;
6      free(first);
7      return t;
8    } else {
9      razlomak *del;
10     t = first;
11     while (
12       t->next &&
13       t->next->br * y != t->next->naz * x
14     ) t = t->next;
15     if (t->next) {
16       del = t->next;
17       t->next = del->next;
18       free(del);
19     }
20     return first;
21   }
22 }
```

□

Napomena 7.1.8. *U prethodnom zadatku je trebalo provjeriti jednakost razlomaka. Iako se svaki razlomak* $\frac{x}{y}$ *može direktno evaluirati kao realan broj (*`(double)x/y`*), zbog grešaka u računu daleko je bolje usporedbu razlomaka* $\frac{x_1}{y_1}$ *i* $\frac{x_2}{y_2}$ *provesti svođenjem na zajednički nazivnik i provjerom jednakosti*

$$x_1 \cdot y_2 \stackrel{?}{=} x_2 \cdot y_1.$$

Zadatak 7.1.19. *Napišite funkciju* `deleteAll()` *koja kao argumente uzima pointer na početak liste razlomaka (definiranih kao u zadatku 7.1.17), te brojnik i nazivnik razlomka. Funkcija treba iz liste obrisati* **sve** *razlomke koji imaju jednaku vrijednost kao razlomak definiran brojnikom i nazivnikom (argumenti funkcije).*

Rješenje. Rješenje je slično prethodnom. Potrebno je samo prepraviti grananja u petlji i malo "doraditi" rješenje:

```
razlomak *deleteAll(razlomak *first, int x, int y) {
  razlomak *t, *del;
  while (
    first &&
    first->br * y == first->naz * x
  ) {
    t = first;
    first = t->next;
    free(t);
  }
  if (!first) return NULL;
  t = first;
  while (t->next) {
    if (t->next->br * y == t->next->naz * x) {
      del = t->next;
      t->next = del->next;
      free(del);
    } else
      t = t->next;
  }
  return first;
}
```

□

Zadatak 7.1.20. *Napišite funkciju* `deleten()` *koja kao argumente uzima pointer na početak liste razlomaka (definiranih kao u zadatku 7.1.17), te prirodni broj* $n \in \mathbb{N}$. *Funkcija treba iz liste obrisati* **svaki** n**-ti** *razlomak (počevši od prvog). Za* $n = 1$, *funkcija treba obrisati sve elemente liste.*

Zadatak 7.1.21. *Napišite funkciju* `deleteNeg()` *koja kao argument uzima pointer na početak liste razlomaka (definiranih kao u zadatku 7.1.17) i iz nje briše sve negativne razlomke.*
Podsjetnik: *Lista nije ni na koji način "uređena", pa negativni mogu biti i brojnici i nazivnici!*

Zadatak 7.1.22. *Napišite funkciju* `deleteGeom()` *koja kao argument uzima pointer na početak liste razlomaka (definiranih kao u zadatku 7.1.17). Funkcija treba iz liste obrisati one razlomke koji su strogo manji od geometrijske sredine svih elemenata. Pri tome nije dozvoljeno koristiti realne brojeve (dakle, niti funkciju* `pow()` *i slične).*
Pretpostavite da su i brojnici i nazivnici svih elemenata liste pozitivni brojevi.

Uputa. Ako elemente liste označimo kao niz, tj. s $(x_i)_{i=1}^n$, onda se u zadatku traži brisanje

svih elemenata koji zadovoljavaju uvjet

$$x_i < \sqrt[n]{\prod_{j=1}^{n} x_j}.$$

No, taj uvjet je ekvivalentan uvjetu

$$x_i^n < \prod_{j=1}^{n} x_j.$$

Ako uvedemo oznaku:

$$x_i := \frac{a_i}{b_i},$$

onda je traženi uvjet

$$a_i^n \cdot \prod_{j=1}^{n} b_j < b_i^n \cdot \prod_{j=1}^{n} a_j.$$

Tražene produkte i broj n je jednostavno izračunati jednim prolazom kroz listu (v. računanje prosjeka u zadatku 6.1.11). □

Napomena 7.1.9. *Elemente liste* **ne možemo** *indeksirati! U prethodnoj uputi, indeksiranje je upotrijebljeno zbog matematičkog zapisa, no u programu ga ne možemo koristiti! Kako se pristupa n-tom elementu liste (za zadani n), pogledajte u zadatku 7.1.26.*

Zadatak 7.1.23. *Napišite funkciju koja kao argument uzima pointer na početak liste razlomaka (definiranih kao u zadatku 7.1.17), te vraća pointer na invertiranu verziju te liste. Invertiranje treba postići razmještanjem postojećih elemenata,* **BEZ** *upotrebe funkcije* `malloc()`.

Uputa. Potrebno je elemente vaditi s početka liste (slično prvom bloku funkcije `delete()` u rješenju zadatka 7.1.18) i dodavati ih na početak nove liste (poput funkcije `insert()` u zadatku 7.1.17). Pri tome, kod brisanja ne treba pozivati `free()` niti kod dodavanja ne treba pozivati `malloc()`, nego treba samo "popraviti strelice". □

Zadatak 7.1.24. *Napišite funkciju koja kao argument uzima pointer na početak liste razlomaka (definiranih kao u zadatku 7.1.17). Funkcija treba "pokratiti" sve elemente liste.*

Rješenje. Ovdje je potrebno "protrčati" svim elementima liste i podijeliti svaki brojnik i nazivnik s njihovom najvećom zajedničkom mjerom (koju ćemo izračunati pomoću Euklidovog algoritma).

Primijetimo da je varijabla `first` u funkciji lokalna. Zbog toga mijenjanje varijable `first` neće afektirati varijablu `first` u glavnom programu (ovo ne vrijedi i za ono na što `first` pokazuje!), pa ju smijemo u funkciji iskoristiti za "trčanje" po listi.

Očito, funkcija ne mora vraćati nikakvu vrijednost.

```
int gcd(int a, int b) {
  while (b > 0) {
    int t = a % b;
    a = b;
    b = t;
  }
  return a;
}
void skrati(razlomak *first) {
  while (first) {
    int g = gcd(first->br, first->naz);
    first->br /= g;
    first->naz /= g;
    first = first->next;
  }
}
```

□

Zadatak 7.1.25. *Napišite funkciju koja kao argument prima pointer na početak liste razlomaka (definiranih kao u zadatku 7.1.17), te razlomke koji su po apsolutnoj vrijednosti manji od* 1 *zamjenjuje s njihovim recipročnim vrijednostima.*

Zadatak 7.1.26. *Napišite funkciju koja kao argumente prima pointer na početak liste razlomaka (definiranih kao u zadatku 7.1.17) i cijeli broj* n*. Funkcija treba* n*-tom elementu liste zadati vrijednost* $1/n$ *(pri tome elemente brojimo od* 1*, a ne od nule).*

Rješenje. Ovdje se pojavljuje problem lociranja n-tog elementa liste. Za razliku od nizova, kod liste nema načina da tom elementu pristupimo direktno; potrebno ga je "pronaći".

```
void promijeni(razlomak *first, int n) {
  int k = n;
  while (--k) first = first->next;
  first->br = 1;
  first->naz = n;
}
```

□

Zadatak 7.1.27. *Napišite funkciju koja prima jedan argument, prirodni broj* $n \in \mathbb{N}$*. Funkcija treba kreirati listu razlomaka (definiranih kao u zadatku 7.1.17) oblika*

$$\frac{1}{k},\ k = 1, 2, \ldots, n$$

i vratiti pointer na početak liste.

Rješenje. Jedan način rješavanja ovog zadatka je dodavanje razlomaka $1/k$ na kraj liste za sve k od 1 do n:

```
razlomak *kreiraj1n(int n) {
  razlomak *first = NULL, *last, *new;
  int k;
  for (k = 1; k <= n; k++) {
    new = (razlomak*)malloc(sizeof(razlomak));
    if (first)
      last = last->next = new;
    else
      last = first = new;
    new->br = 1;
    new->naz = k;
  }
  if (first) last->next = NULL;
  return first;
}
```

No, dodavanje na početak liste je jednostavnije, a ovdje imamo mogućnost listu kreirati "naopako", na način da prvo ubacimo $1/n$, zatim $1/(n-1)$ i tako dalje sve do $1/1$. Naravno, u ovom slučaju, razlomke ćemo dodavati na početak:

```
razlomak *kreiraj1n(int n) {
  razlomak *first = NULL, *new;
  while (n) {
    new = (razlomak*)malloc(sizeof(razlomak));
    new->br = 1;
    new->naz = n--;
    new->next = first;
    first = new;
  };
  return first;
}
```

Upotrebom funkcije `insert()` iz rješenja zadatka 7.1.17, ovaj problem možemo riješiti još kraće:

```
razlomak *kreiraj1n(int n) {
  razlomak *first = NULL;
  while (n) first = insert(first, 1, n--);
  return first;
}
```

□

Zadatak 7.1.28. *Napišite funkciju koja prima dva cjelobrojna argumenta: $a, b \in \mathbb{N}$. Funkcija treba kreirati listu svih razlomaka (definiranih kao u zadatku 7.1.17) oblika*

$$\frac{i}{j},\ i \in \{1, 2, \ldots, a\}, j \in \{1, 2, \ldots, b\},$$

te vratiti pointer na početak stvorene liste.

Zadatak 7.1.29. *Napišite funkciju koja prima dva cjelobrojna argumenta:* $a, b \in \mathbb{N}$. *Funkcija treba kreirati listu svih* **potpuno skraćenih** *razlomaka (definiranih kao u zadatku 7.1.17) oblika*

$$\frac{i}{j},\ i \in \{1, 2, \ldots, a\}, j \in \{1, 2, \ldots, b\},$$

te vratiti pointer na početak stvorene liste. Pri tome, svaki razlomak smije se pojavljivati točno jednom!

Uputa. Ovaj zadatak sličan je prethodnom. Za postizanje "potpune skraćenosti" razlomaka i jedinstvenosti svakog elementa liste, dovoljno je provjeriti: $\mathrm{GCD}(i, j) = 1$. Ako je taj uvjet ispunjen, potrebno je kreirati element liste; inače ga preskačemo. □

Zadatak 7.1.30. *Napišite funkciju* `presjek` *koja kao argumente uzima pointere na početke dvije liste razlomaka (definiranih kao u zadatku 7.1.17). Liste predstavljaju skupove racionalnih brojeva (niti jedan razlomak se ne pojavljuje više od jednom u istoj listi), a razlomci u njima su sortirani prema veličini (uzlazno).*

Funkcija treba kreirati novu listu koja sadrži one razlomke koji se nalaze u obje liste, te vratiti pointer na početak nove liste. Pri tome, ulazne liste moraju ostati nepromijenjene!

Rješenje. Modificiramo Merge sort na način da elemente dodajemo u novu listu samo ako se nalaze na početku obje ulazne liste.

```
1  razlomak *presjek(razlomak *a, razlomak *b) {
2    razlomak *first = NULL, *new, *last;
3    while (a && b) {
4      int exp1 = a->br * b->naz, exp2 = a->naz * b->br;
5      if (exp1 == exp2) {
6        new = (razlomak*)malloc(sizeof(razlomak));
7        if (first)
8          last = last->next = new;
9        else
10         last = first = new;
11       new->br = a->br;
12       new->naz = a->naz;
13       a = a->next;
14       b = b->next;
15     } else if (exp1 < exp2)
16       a = a->next;
17     else
18       b = b->next;
19   }
20   if (first) last->next = NULL;
21   return first;
22 }
```

□

Zadatak 7.1.31. *Napišite funkciju* `presjek2` *koja kao argumente uzima pointere na početke dvije liste razlomaka (definiranih kao u zadatku 7.1.17). Liste predstavljaju skupove racionalnih brojeva (niti jedan razlomak se ne pojavljuje više od jednom u istoj listi), a razlomci u njima su sortirani prema veličini (uzlazno).*

Funkcija treba preslagivanjem elemenata dobivenih listi (dakle, bez upotrebe funkcije `malloc()`*) kreirati novu listu u kojoj se nalaze oni elementi koji se nalaze u obje liste. Na kraju, funkcija treba vratiti pointer na početak nove liste. "Višak" elemenata (one koji ne završe u novoj listi) treba obrisati iz memorije!*

Rješenje. Modificiramo Merge sort. Ako se neki razlomak nalazi na početku obje ulazne liste, element jedne prebacujemo u novu listu, a element druge brišemo. U protivnom, brišemo manji početak dvije liste.

U prethodnom rješenju smo se "vrtili" u petlji samo dok nismo "potrošili" elemente jedne od listi. U ovom rješenju, na kraju moramo obrisati do kraja onu listu koju još nismo "potrošili" (posljednje dvije `while()`-petlje).

```
1  razlomak *presjek2(razlomak *a, razlomak *b) {
2    razlomak *first = NULL, *last;
3    while (a && b) {
4      int exp1 = a->br * b->naz, exp2 = a->naz * b->br;
5      if (exp1 == exp2) {
6        razlomak *tmp = b;
7        if (first)
8          last = last->next = a;
9        else
10         last = first = a;
11       a = a->next;
12       b = b->next;
13       free(tmp);
14     } else if (exp1 < exp2) {
15       razlomak *tmp = a;
16       a = a->next;
17       free(tmp);
18     } else {
19       razlomak *tmp = b;
20       b = b->next;
21       free(tmp);
22     }
23   }
24   last->next = NULL;
25   while (a) {
26     razlomak *tmp = a;
27     a = a->next;
28     free(tmp);
29   }
30   while (b) {
```

```
31      razlomak *tmp = b;
32      b = b->next;
33      free(tmp);
34    }
35    return first;
36  }
```

□

Zadatak 7.1.32. *Napišite funkciju* `unija` *koja kao argumente uzima pointere na početke dvije liste razlomaka (definiranih kao u zadatku 7.1.17). Liste predstavljaju skupove racionalnih brojeva (niti jedan razlomak se ne pojavljuje više od jednom u istoj listi), a razlomci u njima su sortirani prema veličini (uzlazno).*

Funkcija treba kreirati novu listu koja sadrži one razlomke koji se nalaze u barem jednoj od dvije liste (bez ponavljanja elemenata u novoj listi), te vratiti pointer na početak nove liste. Pri tome, ulazne liste moraju ostati nepromijenjene!

Uputa. Ovdje je riječ o još jednoj modificikaciji klasičnog Merge sorta: ako su početni elementi lista jednaki (u smislu da sadrže jednake razlomke), pomak se radi u obje liste (iako se u novu listu dodaje samo jedan element). □

Zadatak 7.1.33. *Napišite funkciju* `unija2` *koja kao argumente uzima pointere na početke dvije liste razlomaka (definiranih kao u zadatku 7.1.17). Liste predstavljaju skupove racionalnih brojeva (niti jedan razlomak se ne pojavljuje više od jednom u istoj listi), a razlomci u njima su sortirani prema veličini (uzlazno).*

Funkcija treba preslagivanjem elemenata dobivenih listi (dakle, bez upotrebe funkcije `malloc()`*) kreirati novu listu u kojoj se nalaze oni elementi koji se nalaze u barem jednoj od dvije liste (bez ponavljanja elemenata u novoj listi). Na kraju, funkcija treba vratiti pointer na početak nove liste. "Višak" elemenata (one koji ne završe u novoj listi) treba obrisati iz memorije!*

Uputa. Ovdje je riječ o još jednoj modificikaciji klasičnog Merge sorta: ako su početni elementi lista jednaki (u smislu da sadrže jednake razlomke), u novu listu se dodaje samo jedan, dok je drugog potrebno obrisati. □

Zadatak 7.1.34. *Riješite zadatak 6.1.15 pomoću liste skokova.*

Uputa. Skokove je najbolje dodavati u sortiranu listu (dakle, odmah ubacivati na pravo mjesto). Na taj način lako provjerimo postoji li element liste koji odgovara skoku na novu poziciju. Ako postoji, potrebno je samo promijeniti znak koji je zapisan na toj poziciji; inače, dodaje se novi element u listu. □

Poglavlje 8

Datoteke

Nije praktično stalno unositi podatke. Ponekad ih treba pospremiti na disk i kasnije ponovno pročitati. U tu svrhu koristimo datoteke: tekstualne i binarne. Mi ćemo se baviti samo tekstualnima.

Tekstualne datoteke je najlakše promatrati kao standardne ulaze i izlaze. Za rad s njima upotrebljavaju se slične funkcije kao i bez datoteke: `fscanf()` i `fprintf()`. Jedina razlika u odnosu na "obične" `scanf()` i `printf()` je dodatak prvog parametra (pokazivač na **otvorenu** datoteku).

Datoteke otvaramo korištenjem funkcije `fopen()`, a zatvaramo pomoću `fclose()`.

Zadatak 8.1.35. *Napišite program koji učitava nazive dviju tekstualnih datoteka, te prepisuje sadržaj jedne u drugu, na način da na početak svake linije doda broj linije. Na primjer:*

`ulaz.txt`*:*		`izlaz.txt`*:*
Iš'o Pero u dućan	$\Rightarrow$	*1: Iš'o Pero u dućan*
Nije rek'o "Dobar dan".		*2: Nije rek'o "Dobar dan".*

Pretpostavite da su imena datoteka duga najviše 255 *znakova.*

Rješenje.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main (void) {
5    FILE *in, *out;
6    char iname[256], oname[256], c;
7    int i = 1;
8
9    printf("Ime ulazne datoteke: ");
10   scanf("%[^\n]", iname);
11   scanf("%*c");
12   printf("Ime izlazne datoteke: ");
13   scanf("%[^\n]", oname);
```

```
14
15    if ((in = fopen(iname, "rt")) == NULL) {
16      printf("Greska prilikom otvaranja ");
17      printf("datoteke \"%s\".", iname);
18      exit(1);
19    }
20
21    if ((out = fopen(oname, "wt")) == NULL) {
22      fclose(in);
23      printf("Greska prilikom otvaranja ");
24      printf("datoteke \"%s\".", oname);
25      exit(1);
26    }
27
28    fprintf(out, "1: ");
29    while (fscanf(in, "%c", &c) > 0)
30      if (c == '\n')
31        fprintf(out, "\n%d: ", ++i);
32      else
33        fprintf(out, "%c", c);
34
35    fclose(in); fclose(out);
36
37    return 0;
38  }
```

□

Funkcija `fscanf()` vraća broj polja (formata) koji su prepoznati i kojima je pridana vrijednost. Ukoliko je program došao do kraja datoteke koju čita, povratna vrijednost će biti `EOF` (naziv iza kojeg se "skriva" vrijednost -1).

Za naše potrebe, dovoljno je provjeriti jesu li sva polja pročitana (u gornjem primjeru, uvjet `> 0`). U praksi, dobro je provjeriti zašto je čitanje stalo: kraj datoteke, pogrešan/neočekivan raspored podataka u datoteci, oštećen medij,...

Zadatak 8.1.36. *Napišite funkciju koja kao jedini argument prima file-pointer koji pokazuje na datoteku otvorenu za čitanje, te iz nje čita kvadratnu matricu realnih brojeva i vraća trag pročitane matrice. Matrica je zapisana tako da je prvi broj u datoteci cijeli i označava red matrice. Nakon tog broja pobrojani su svi elementi matrice, redak po redak, odvojeni razmacima i/ili skokovima u novi red.*

Rješenje. Maksimalni red matrice nije zadan, no ovdje nam nije niti potrebno da matricu pospremamo u memoriju. Dovoljno je pratiti koji element trenutno čitamo.

Zadatak kaže da je datoteka već otvorena, što znači da ne trebamo ponovno otvarati datoteku (dapače, ne smijemo, jer ne znamo ime datoteke).

Rješenje se svodi na rješavanje zadatka bez datoteke (učitavanje s tipkovnice), te zamjenu svih poziva `scanf()` s adekvatnim `fscanf()`, uz dodatak prvog parametra (`in`).

```
double tr(FILE *in) {
  int n, i, j;
  double x, tr = 0.0;

  fscanf(in, "%d", &n);

  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++) {
      fscanf(in, "%lf", &x);
      if (i == j) tr += x;
    }

  return tr;
}
```

Ovdje je poznato na koji način su zapisani podaci u datoteci, pa nije nužno provjeravati povratnu vrijednost funkcije `fscanf()`. U praksi, tu vrijednost je uvijek dobro provjeriti jer gotovo nikada nemamo garanciju da su podaci ispravno zapisani. □

Zadatak 8.1.37. *Napišite funkciju koja kao jedini argument prima file-pointer koji pokazuje na datoteku otvorenu za čitanje, te iz nje čita kvadratnu matricu realnih brojeva i vraća vrijednost* 1 *ako je matrica simetrična, a* 0 *ako nije. Matrica je zapisana kao u zadatku 8.1.36.*

Rješenje. Za provjeru simetričnosti matrice, potrebno ju je cijelu pospremiti u memoriju. Kako nije zadan najveći red matrice, moramo pribjeći dinamičkoj alokaciji dvodimenzionalnog polja. Nakon provjere simetričnosti, potrebno je osloboditi zauzetu memoriju!

```
int isSymmetric(FILE *in) {
  int n, i, j, sym = 1;
  double **m;

  fscanf(in, "%d", &n);
  m = (double**)malloc(n * sizeof(double*));
  for (i = 0; i < n; i++)
    m[i] = (double*)malloc(n * sizeof(double));

  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      fscanf(in, "%lf", &m[i][j]);

  for (i = 0; i < n; i++)
    for (j = i + 1; j < n; j++)
      if (m[i][j] - m[j][i]) {
        sym = 0;
        i = n;
        break;
```

```
20        }
21
22    for (i = 0; i < n; i++) free(m[i]);
23    free(m);
24
25    return sym;
26  }
```

□

Rješenja prethodna dva zadatka možemo testirati sljedećim programom:

```
1   int main (void) {
2     FILE *in;
3
4     if ((in = fopen("t.in", "rt")) == NULL) {
5       printf("Greska prilikom otvaranja datoteke!\n");
6       exit(1);
7     }
8
9     printf("tr(A) = %g\n", tr(in));
10    rewind(in);
11    printf(
12      "A %sje simetricna.\n",
13      isSymmetric(in) ? "" : "ni"
14    );
15
16    fclose(in);
17
18    return 0;
19  }
```

Funkcija `rewind(·)` vraća čitanje na početak datoteke na koju pokazuje argument (kod nas `in`).

Zadatak 8.1.38. *Napišite funkciju koja kao jedini argument prima file-pointer koji pokazuje na datoteku otvorenu za čitanje, te iz nje čita kvadratnu matricu realnih brojeva i vraća produkt svih elemenata koji nisu na niti jednoj od dijagonala. Matrica je zapisana kao u zadatku 8.1.36.*

Zadatak 8.1.39. *Napišite funkciju koja kao jedini argument prima file-pointer koji pokazuje na datoteku otvorenu za čitanje, te iz nje čita kvadratnu matricu realnih brojeva i vraća produkt suma svih redaka, tj.* $\prod_{i=0}^{n-1} \sum_{j=0}^{n-1} a_{ij}$. *Matrica je zapisana kao u zadatku 8.1.36.*

Zadatak 8.1.40. *Napišite funkciju koja kao jedini argument prima file-pointer koji pokazuje na datoteku otvorenu za čitanje, te iz nje čita kvadratnu matricu realnih brojeva i vraća* 3 *ako je matrica dijagonalna. Ako nije dijagonalna, treba vratiti* 1 *ako je gornje*

trokutasta, 2 ako je donje trokutasta, a 0 ako nije niti jedno od nabrojanog. Matrica je zapisana kao u zadatku 8.1.36.

Zadatak 8.1.41. *Napišite funkciju koja kao jedini argument prima file-pointer koji pokazuje na datoteku otvorenu za čitanje, te iz nje čita kvadratnu matricu realnih brojeva i vraća najveću sumu stupca u matrici. Matrica je zapisana kao u zadatku 8.1.36.*

Rješenje. Matrica je zapisana po recima, pa za čitanje po stupcima treba alocirati niz u kojem ćemo čuvati sume svih stupaca.

```
1  double colSum(FILE *in) {
2    int n, i, j;
3    double *sum, x, max;
4
5    fscanf(in, "%d", &n);
6    sum = (double*)malloc(n * sizeof(double));
7
8    for (i = 0; i < n; i++) sum[i] = 0;
9    for (i = 0; i < n; i++)
10     for (j = 0; j < n; j++) {
11       fscanf(in, "%lf", &x);
12       sum[j] += x;
13     }
14
15   max = sum[0];
16   for (i = 1; i < n; i++)
17     if (max < sum[i]) max = sum[i];
18
19   free(sum);
20
21   return max;
22 }
```

□

Zadatak 8.1.42. *Napišite funkciju koja kao argumente prima sljedeće file-pointere:* `in` *koji pokazuje na datoteku otvorenu za čitanje i* `out` *koji pokazuje na datoteku otvorenu za pisanje. Funkcija treba iz datoteke* `in` *čitati kvadratnu matricu realnih brojeva, te u datoteku* `out` *zapisati transponiranu matricu. Matrica je u datoteci* `in` *zapisana kao u zadatku 8.1.36, te na isti način treba biti zapisana i u datoteci* `out`*.*

Rješenje. Rješenje ovog zadatka je, ponovno, ekvivalentno rješenju koje bismo imali i bez datoteka:

```
1  void transpose(FILE *in, FILE *out) {
2    int n, i, j;
3    double **m;
4
```

```
5    fscanf(in, "%d", &n);
6    m = (double**)malloc(n * sizeof(double*));
7    for (i = 0; i < n; i++)
8      m[i] = (double*)malloc(n * sizeof(double));
9
10   for (i = 0; i < n; i++)
11     for (j = 0; j < n; j++)
12       fscanf(in, "%lf", &m[i][j]);
13
14   fprintf(out, "%d\n", n);
15   for (i = 0; i < n; i++) {
16     for (j = 0; j < n; j++)
17       fprintf(out, "%7g", m[j][i]);
18     fprintf(out, "\n");
19   }
20
21   for (i = 0; i < n; i++) free(m[i]);
22   free(m);
23 }
```

□

Zadatak 8.1.43. *Napišite funkciju koja kao argumente prima sljedeće file-pointere:* `in1` *i* `in2` *koji pokazuju na datoteku otvorenu za čitanje i* `out` *koji pokazuje na datoteku otvorenu za pisanje. Funkcija treba iz datoteka* `in1` *i* `in2` *čitati kvadratne matrice realnih brojeva, te u datoteku* `out` *zapisati njihov umnožak. Matrice su u datotekama* `in1` *i* `in2` *zapisane kao u zadatku 8.1.36, te na isti način treba biti zapisana i matrica u datoteci* `out`*.*

Ako množenje nije moguće provesti (različite dimenzije matrica), potrebno je prijaviti grešku.

Zadatak 8.1.44. *U datoteci su popisane nogometne utakmice, u svakom retku po jedna i to u sljedećem formatu:*

```
Klub 1:Klub 2=a:b
```

Brojevi a i b označavaju broj golova koje je dao "Klub 1" odnosno "Klub 2". Pobjeda donosi 3 *boda, izjednačenje* 1 *bod, a poraz* 0 *bodova. Napišite funkciju koja kao argumente prima nazive ulazne i izlazne datoteke, te učitava podatke o utakmicama (snimljene na opisani način) iz ulazne datoteke. U izlaznu datoteku funckija treba ispisati konačnu rang-listu klubova sortiranu silazno prema broju bodova, u formatu pogodnom za čitanje pomoću tabličnih kalkulatora (OpenOffice.org Calc, Quattro Pro, Microsoft Excel i sl) tako da u prvom stupcu piše redni broj, u drugom naziv momčadi, a u trećem broj bodova.*

Možete pretpostaviti da je ukupni broj klubova najviše 20*, te da je naziv svakog kluba najviše* 50 *znakova.*

Rješenje. Podatke možemo organizirati na dva načina. Jedan je da konstruiramo dva niza: jedan s nazivima klubova i jedan s brojevima bodova koje su skupili. Drugi način

je slaganje jednog niza `struct`-ova, pri čemu u svaki `struct` pohranjujemo naziv kluba i broj bodova koje je klub skupio. Mi ćemo upotrijebiti drugi način.

Od formata pogodnih za čitanje pomoću tabličnih kalkulatora, najlakše je složiti tzv. CSV (engl. *comma separated values*): jedan redak tablice odgovara retku u datoteci, a pojedina polja ("ćelije" u tablici) odvajaju se nekim separatorom, najčešće `TAB`-om ili točka-zarezom.

Za traženje momčadi po listi, upotrijebit ćemo pomoćnu funkciju (jer se traženje radi na dva mjesta u kodu).

```
typedef struct {
  char name[51];
  int score;
} team;

int teamIndex(char name[], team list[], int *n) {
  int i;
  for (i = 0; i < *n; i++)
    if (!strcmp(name, list[i].name)) return i;
  strcpy(list[i].name, name);
  list[(*n)++].score = 0;
  return i;
}

void football(char iname[], char oname[]) {
  FILE *in, *out;
  team list[20];
  int n = 0, s1, s2, i1, i2;
  char t1[51], t2[51];

  if ((in = fopen(iname, "rt")) == NULL) {
    printf("Greska prilikom otvaranja ");
    printf("datoteke \"%s\".", iname);
    exit(1);
  }

  if ((out = fopen(oname, "wt")) == NULL) {
    fclose(in);
    printf("Greska prilikom otvaranja ");
    printf("datoteke \"%s\".", oname);
    exit(1);
  }

  while (fscanf(in,
    "%[^:]:%[^=]=%d:%d ", t1, t2, &s1, &s2
  ) == 4) {
    i1 = teamIndex(t1, list, &n);
```

```
38      i2 = teamIndex(t2, list, &n);
39      if (s1 < s2)
40        list[i2].score += 3;
41      else if (s1 > s2)
42        list[i1].score += 3;
43      else {
44        list[i1].score++;
45        list[i2].score++;
46      }
47    }
48
49    fclose(in);
50
51    for (i1 = 0; i1 < n; i1++)
52      for (i2 = i1 + 1; i2 < n; i2++)
53        if (list[i1].score < list[i2].score) {
54          team tmp = list[i1];
55          list[i1] = list[i2];
56          list[i2] = tmp;
57        }
58
59    for (i1 = 0; i1 < n; i1++)
60      fprintf(out, "%d. %s (%d)\n",
61        i1 + 1,
62        list[i1].name,
63        list[i1].score
64      );
65
66    fclose(out);
67  }
```

□

CSV datoteku možete otvoriti u tabličnom kalkulatoru, te dodatno ukrasiti i/ili dodati formule, grafove i slično, kao da ste tablicu složili "na ruke".

Zadatak 8.1.45. *Riješite prethodni zadatak bez ograničenja na najveći mogući broj klubova.*

Upute. Izmijenite funkciju `teamIndex()` tako da po potrebi realocira memoriju za nove elemente polja. Pri tome argument `list` mora biti tipa `team**`. Također, potrebno je prilagoditi deklaraciju liste momčadi u funkciji `football()`.

Umjesto realokacije polja, zadatak možete riješiti pomoću vezane liste. Kod takvog pristupa je dodavanje novog elementa jednostavnije, no onda je sort složeniji. □

Napomena 8.1.10 (Snimanje vezanih lista)**.** *Prilikom snimanja vezane liste u datoteku ne smijete snimati pointer na sljedeći element liste (format* `%p` *ili* `%u`*), te ga kod učitavanja*

koristiti kao vezu na idući član, jer ne znamo gdje će `malloc()` *kreirati novi element liste. U datoteku smijemo snimati samo "korisne podatke" (bez pointera), a pointere treba rekreirati prilikom čitanja liste iz datoteke.*

Zadatak 8.1.46. *Napišite funkciju koja kao argumente uzima ime datoteke, te realne parametre a, b i $d > 0$. Neka je $m := \min\{a, b\}$ i $M := \max\{a, b\}$. Funkcija treba u datoteku zapisati tablicu s dva stupca: u prvom stupcu trebaju biti sve vrijednosti*

$$x \in \{m + k \cdot d \leq M : \ k \in \mathbb{N}_0\}$$

u uzlaznom poretku, a u drugom stupcu trebaju biti odgovarajuće vrijednosti funkcije

$$f(x) := \frac{\sin x}{\cos x + \log x}.$$

Izlazna datoteka treba biti u CSV formatu, s točka-zarezima kao separatorima. Odgovarajuće matematičke funkcije nalaze se u biblioteci `math`*.*

Datoteku stvorenu pomoću funkcije iz prethodnog zadatka možete učitati u tablični kalkulator, te s nekoliko klikova složiti graf zadane funkcije.

Zadatak 8.1.47. *Napišite funkciju koja kao argumente prima sljedeće file-pointere:* `in` *koji pokazuje na datoteku otvorenu za čitanje i* `out` *koji pokazuje na datoteku otvorenu za pisanje. Funkcija treba pročitati datoteku* `in` *i njen sadržaj prepisati u datoteku* `out` *na način da linije idu obrnutim redoslijedom. Na primjer:*

`in`*:*		`out`*:*
Iš'o Pero u dućan	$\Longrightarrow$	*Nije rek'o "Dobar dan".*
Nije rek'o "Dobar dan".		*Iš'o Pero u dućan*

Nemojte postavljati ograničenja na duljine linija, ali možete pretpostaviti da su sve riječi u datoteci duge najviše 30 *znakova i odvojene točno jednim razmakom ili skokom u novi red.*

Uputa. Linije čitajte riječ po riječ (format `%s`), te ih spremajte u string koji će, po potrebi, "rasti" (pomoću `realloc()`). Takve stringove spremajte u vezanu listu, dodavanjem na početak liste. Na kraju je potrebno samo ispisati stringove u izlaznu datoteku i osloboditi alociranu memoriju (i onu za stringove i onu za čvorove liste).

Zadatak je moguće riješiti i bez liste, dinamički alociranim (i relociranim) poljem. □

Zadatak 8.1.48 (Šlag na kraju)**.** *Dane su dvije datoteke:* `ma.txt` *i* `la.txt`*. U svakoj se nalaze ocjene studenata, i to po jedan redak za svakog studenta, u formatu:*

```
ime studenta;prezime studenta;ocjena
```

Napišite program koji spaja te dvije datoteke u jednu, `sve.txt`*, na način da za svakog studenta zapiše obje ocjene (ili "–" ako se student ne pojavljuje na listi). Format zapisa izlazne datoteke treba također biti CSV, a podaci trebaju biti sortirani uzlazno prema*

prezimenu. Na primjer, od datoteka:

`ma.txt`*:*

```
Pero;Sapun;5
Ana Marija;Prekoplotic;1
Djuro;Pajser;2
```

`la.txt`*:*

```
Ana;Prekoplotic;2
Pero;Sapun;5
Djuro;Pajser;3
```

treba dobiti datoteku `sve.txt`*:*

```
Djuro;Pajser;2;3
Ana;Prekoplotic;-;2
Ana Marija;Prekoplotic;1;-
Pero;Sapun;5;5
```

Duljinu imena i prezimena studenta ograničite na 50 *znakova.*

Upute. Ovdje čitamo dvije datoteke (općenitiji zadatak bi radio s nizom datoteka) u kojima su podaci zapisani na isti način. Najjednostavniji način čitanja je pomoću funkcije slične funkciji `teamIndex()` iz rješenja zadatka 8.1.44. Funkcija može raditi s globalnim poljem koje, po potrebi, produljuje realokacijom memorije, pa su joj dovoljna samo dva argumenta: `ime` i `prezime` (studenta).

Alternativno, može se složiti funkcija koja će obavljati cijelo čitanje jedne datoteke (od `fopen()` do `fclose()`, uključivo).

Podatke o studentu najbolje je pamtiti u nizu `struct`-ova s tri polja: `ime`, `prezime` (stringovi duljine do 50 znakova) i `ocjene` (niz od onoliko `int`-ova koliko ima ulaznih datoteka). Ako želite složiti za neodređeni broj datoteka, niz ocjena mora se dinamički alocirati (malo više posla, no ne naročito teško; dobro za vježbu).

Sort je rutina (pokušajte složiti tako da osobe s jednakim prezimenom sortira prema imenu), kao i ispis.

Primijetite i da pojedina osoba može imati više imena i/ili prezimena, odvojenih razmakom, no znamo da će ime(na) i prezime(na) biti odvojena točka-zarecom. Stoga je idealni format za učitavanje imena i prezimena `[^;]`. □

Za vježbu možete bilo koji zadatak iz prethodnih poglavlja modificirati tako da podatke čita iz datoteke i/ili ih zapisuje u datoteku.

Indeks