

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Tomaž Dobravec

ab**C**

Ljubljana, januar 2007

Zapiski abC, ki obsegajo del snovi predmeta Osnove programiranja 2, niso samostojno gradivo. Namenjeni so študentom prvega letnika visoko strokovnega študija Fakultete za računalništvo in informatiko Univerze v Ljubljani, da bodo lažje spremljali predavanja omenjenega predmeta. V zapiskih so zbrani povzetki nekaterih osnov jezika C, ki jih podrobneje obravnavam na predavanjih. Zapiski niso priročnik niti ne *hiter vodič v C*, saj v njih manjka tisto vezno besedilo, ki ga ob mahanju z rokami in neposrednem stiku s študenti lažje posredujem v predavalnici.

Zapiski poleg bolj ali manj skope razlage posameznih delov snovi vsebujejo dve pomembni komponenti: naloge (🔗) in izzive (🎯). Z nalogami se bomo ukvarjali na predavanjih (in jih tam večinoma tudi rešili), izzivi pa so mišljeni kot domača naloga. Študentom priporočam, da pred reševanjem izzivov še enkrat samostojno rešijo naloge s predavanj. Čeprav se včasih morda zdi, da je rešitev preprosta in "že osvojena", se pogosto šele ob ponovnem reševanju pokaže, kje so šibke točke v znanju.

Naloge in nekateri izzivi so v celoti rešeni in predstavljeni v teh zapiskih, v elektronski obliki pa jih lahko najdete na domači strani predmeta OP2.

Tomaž Dobravec

Zavedam se, da zapiski niso brez napak in pomanjkljivosti. Prosim bralce, da me nanje opozorijo po elektronski pošti na naslov tomaz.dobravec@fri.uni-lj.si.

Kazalo

1	Uvod	9
1.1	Osnovno o jeziku C	9
1.2	Primerjava z jezikom Java	9
1.3	Priporočeni prevajalniki za C	11
1.4	Pomoč	12
1.5	Prvi program v C	12
1.6	Rezervirane besede jezika C	13
2	Spoznavanje osnovnih ukazov preko primerov	15
2.1	Večkratno ponavljanje ukaza	15
2.2	Preprost formatiran izpis	17
2.3	Malo zahtevnejši formatiran izpis	18
2.4	Argumenti programa	20
2.5	Tabela Fahrenheit - Celsius	23
2.6	Naključna števila	25
2.7	Branje podatkov	28
3	Izrazi in operatorji	31
3.1	Aritmetični operatorji	31
3.2	Relacijski operatorji	31
3.3	Logični operatorji	32
3.4	Operatorja ++ in --	32
3.5	Bitni operatorji	32
3.6	Operator ,	34
3.7	Operator ?:	34
3.8	Prireditveni stavki	35
3.9	Prioriteta in asociativnost operatorjev	36
4	Strukturiranje kode	37
4.1	Ukaz if-else	37
4.2	Zanke do, while in for	38
4.3	Ukaz goto	38
4.4	Ukaz switch	38

4.5	Funkcije	39
4.6	Program v več datotekah	42
5	Podatkovni tipi in konstante	45
5.1	Osnovni številski podatkovni tipi	46
5.2	Znaki	48
5.3	Tabele (polja)	49
5.4	Večdimenzionalne tabele	55
5.5	Nizi	60
5.6	Strukture	64
6	Spremenljivke	69
6.1	Ime spremenljivke	69
6.2	Vidljivost (scope)	70
6.2.1	Bločna vidljivost	71
6.2.2	Programska vidljivost	72
6.2.3	Datotečna vidljivost	73
6.2.4	Avtomatske in statične lokalne spremenljivke	74
6.3	Organizacija pomnilnika	75
6.3.1	Inicializacija spremenljivk	77
7	Kazalci	79
7.1	Kaj so kazalci?	79
7.2	Kazalci in tabele	81
7.3	Dinamično delo s pomnilnikom	83
7.4	Prenos parametrov v funkcije	85
7.4.1	Prenos reference po vrednosti	87
7.4.2	Kazalec na kazalec (<code>int **x</code>)	88
7.4.3	Tabela kot parameter	88
7.4.4	Uporaba funkcije <code>scanf</code>	90
7.5	Kazalci in strukture	90
7.6	Linearni seznam	91
8	Delo z datotekami	101
8.1	Branje tekstovnih datotek po znakih	104
8.2	Branje tekstovnih datotek po besedah	104
8.3	Branje podatkov v tekstovnih datotekah	105
8.4	Branje tekstovnih datotek po vrsticah	108
8.5	Kopiranje tekstovne datoteke	109
8.6	Pisanje podatkov v tekstovno datoteko	110
8.7	Dodajanje v tekstovno datoteko	110
8.8	Kopiranje binarne datoteke	111
8.9	Branje in pisanje binarne datoteke po blokih	111

9	Rešitve nekaterih izzivov	113
9.1	Izziv 2-I	113
9.2	Izziv 2-II	114
9.3	Izziv 2-III	114
9.4	Izziv 2-IV	115
9.5	Izziv 5-I	116
9.6	Izziv 5-VII	117

Poglavje 1

Uvod

1.1 Osnovno o jeziku C

Programski jezik C je splošno namenski programski jezik, v osnovi podoben jezikoma Fortran in Pascal (in še mnogim drugim sodobnim jezikom). Prvotno je bil namenjen uporabi v Unix okolju, zelo kmalu pa se je razširil in postal eden najbolj uporabljenih jezikov v skoraj vseh sistemih. Prva različica jezika C je nastala leta 1970 (isto leto je nastala tudi prva različica jezika Pascal; precej starejši so Fortran (1954), Lisp (1958) in Cobol (1959)).

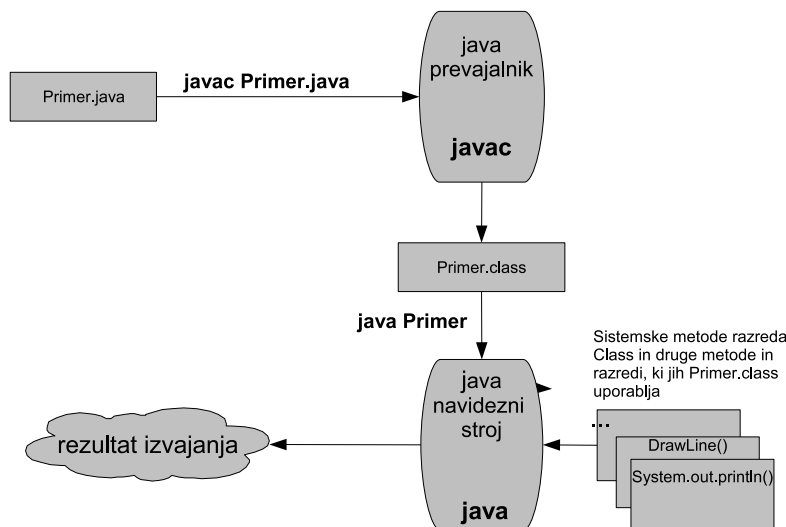
Različice jezika C:

- danes najbolj razširjena standardizacija jezika C (**Ansi C**) je nastala leta 1988; potrditev standarda: 1989; popravki: 1994 in 1996; različica znana tudi kot C89;
- nova izdaja ISO C standarda je bila objavljena leta 1999; popravki: 2001, 2004; ime različice: **C99**; različica še ni povsem zaživela, je pa to stvar bližnje prihodnosti;
- objektna različica jezika C (C++) je nastala leta 1983.

Čeprav C na področju pisanja uporabniških programov zmeraj bolj izpodrivajo sodobni jeziki (kot na primer Java in C#) je C še vedno nepogrešljiv, saj je, poleg mnogih drugih programov, tudi velik del sodobnih operacijskih sistemov (Unix, Windows, ...) pisan v njem.

1.2 Primerjava z jezikom Java

Čeprav sta sintaksi jezikov Java in C (vsaj na prvi pogled) precej podobni, se jezika med seboj močno razlikujeta. Omenimo le najpomembnejši razliki: objektna orien-



Slika 1.1: Primer prevajanja in izvajanja v Javi

tiranost in način prevajanja.

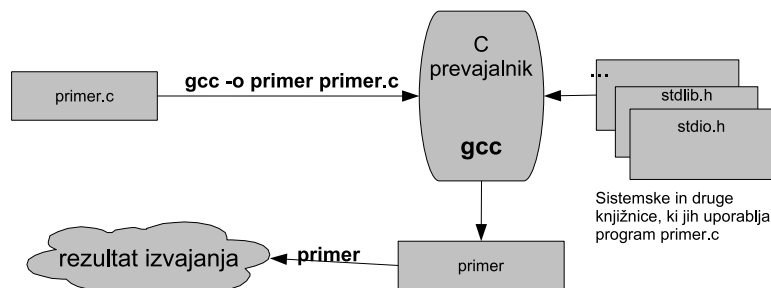
Objektna orientiranost. C ni objektno orientiran jezik. Če je programiranje v Javi podobno delegiranju opravil množici robotov (vsak robot predstavlja en objekt, opravilo je klic metode), lahko programiranje v C primerjamo z ukazovanjem enemu samemu robotu. Program je v tem primeru spisek ukazov, ki jih robot izvrši enega za drugim.

Prevajanje in izvajanje v Javi. V Javi program (datoteko s končnico .java) z ukazom *javac* prevedemo v t.i. bitno kodo (datoteka s končnico .class). Ta datoteka je izvršljiva v vseh Java navideznih strojih (v operacijskih sistemih Windows, Unix, MacOS, ...). Rečemo, da so javanski programi *prenosljivi* (kodo napišeš in prevedeš na enem računalniku, izvedeš jo kjerkoli). Glej sliko 1.1).

Prevajanje in izvajanje v Javi (Linux okolje)

```
[ user@localhost]# javac Primer.java
[ user@localhost]# java Primer
```

Prevajanje in izvajanje v C. Prevajanje v C poteka v enem samem koraku. Napisan program (navadno je to datoteka s končnico .c) prevedemo v izvršljivo datoteko (v Windowsih ima taka datoteka končnico .exe, v Linuxu nima predpisane končnice). Za izvrševanje prevedene datoteke ne potrebujemo virtualnega stroja (datoteka je izvršljiva sama zase). Prevedeno datoteko lahko izvajamo le v okolju, v katerem smo jo prevedli (primer: datoteke program.exe, ki



Slika 1.2: Primer prevajanja in izvajanja v C

je nastala v Windows okolju kot rezultat prevajanja datoteke `program.c`, ne moremo uporabiti v Linux okolju; če želimo program izvajati tudi v Linuxu, ga moramo tam ponovno prevesti!).

Prevajanje in izvajanje v C (Linux okolje)

```
[ user@localhost ]# gcc -o primer primer.c
[ user@localhost ]# ./ primer
```

S prvim ukazom pokličemo prevajalnik `gcc` in zahtevamo prevod datoteke `primer.c`. Če datoteka `primer.c` ne vsebuje napak, bo `gcc` program prevedel v datoteko `primer` (stikalo `-o`). Z drugim ukazom zahtevamo izvršitev programa `primer`.

1.3 Priporočeni prevajalniki za C

Priporočam uporabo prevajalnika `gcc`, ki obstaja v Linux (<http://gcc.gnu.org/>) kot tudi v Windows (glej <http://www.mingw.org/>) različici.

Kot integrirano programsko okolje lahko uporabljate Eclipse (brezplačno ga dobite na <http://www.eclipse.org/>), Visual Studio (ta je študentom brezplačno na voljo na FRI MSDNAA straneh) ali KDevelop (<http://www.kdevelop.org>)

GCC podpira več standardov jezika C. Privzet standard: `gnu89` (razširjena verzija ANSI C standarda iz leta 1989). Standard izberemo s stikalom `-std` (primer: `-std=gnu89`, `-std=c89`, `-std=c99`). Popolno skladnost z Ansi C standardom dosežemo s stikaloma `-ansi` ter `-pedantic`.

1.4 Pomoč

- Google (iskane besede: C + ukaz, ki ga iščete)
- Uporabniki Linux okolja boste pomoč za posamezen C ukaz našli z ukazom `man 3 ime_ukaza` (primer `man 3 printf`). Včasih 3 ni najboljša izbira, takrat poskusite brez številke (`man ime_ukaza`) ali s stikalom `-a` (`man -a ime_ukaza`).
- povezave na <http://kepa.fri.uni-lj.si/edu/op2>

1.5 Prvi program v C

V skladu s tradicijo bo naš prvi program na zaslon izpisal pozdravne besede.

```
1 #include <stdio.h>
2 main() {
3     printf("Pozdravljen , svet\n");
4 }
```

Pozdravljen svet v C (*pozdrav.c*)

Za osvežitev spomina pogledjmo še, kako je ta isti program videti v jeziku Java.

```
1 import java.lang.*;
2 class Pozdrav {
3     public static void main(String[] args) {
4         System.out.println("Pozdravljen , svet");
5     }
6 }
```

Pozdravljen svet v Javi (*Pozdrav.Java*)

Podobnosti med Javo in C

- sintaktične podobnosti
 - blok funkcije se določi z zaviranimi oklepaji (`{ in }`),
 - niz se piše med dvojnimi narekovaji,
 - vsak ukaz se konča s podpičjem;
- funkcija, ki se zažene ob izvajanju programa, se imenuje `main`;
- ukaz `#include` v C uporabimo podobno kot ukaz `import` v Javi.

Razlike med Javo in C

- v Javi namesto izraza **funkcija** uporabljamo izraz **metoda**;
- glava metode `main` v Javi je oblike

```
public static void Main(String[] args),
```

v C pa preprosto

```
main().
```

(Opomba: tudi v C lahko glavo funkcije `main` pišemo tako, da v njej deklariramo argumente ukazne vrstice; glej poglavje 2.4);

- namesto metode `System.out.println` smo uporabili `printf`;
- v C smo na koncu niza za izpis dodali znak `\n` (znak za novo vrstico), v Javi za prehod v novo vrstico poskrbi metoda `println()`;
- funkcija `main` ni del razreda (ker v C razredi ne obstajajo).

1.6 Rezervirane besede jezika C

Programski jezik C pozna 32 rezerviranih besed, kot prikazuje spodnje tabela.

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


- Imena spremenljivk in funkcij se morajo razlikovati od rezerviranih besed.
- Podčrtane besede v tabeli imajo podoben (v večini primerov enak) pomen kot v Javi.
- Drugi ukazi jezika C so deklarirani in definirani v knjižnicah.

Poglavje 2

Spoznavanje osnovnih ukazov preko primerov

V tem poglavju bomo spoznavali osnove jezika C skozi posamezne preproste naloge.

2.1 Večkratno ponavljanje ukaza

Naloga 2-I. Napiši program, ki 5-krat izpiše pozdravno besedilo "Pozdravljen, svet". 

Brez dodatnega znanja lahko program *pozdrav.c* predelamo tako, da ukaz `printf` napišemo 5-krat. Tako popravljen program je videti nekako takole:

```
#include <stdio.h>
main() {
    printf("Pozdravljen , svet\n");
    printf("Pozdravljen , svet\n");
    printf("Pozdravljen , svet\n");
    printf("Pozdravljen , svet\n");
    printf("Pozdravljen , svet\n");
}
```

5-krat pozdravljen svet v C (prvič) (*pozdrav5a.c*)

Zgornja rešitev je sicer pravilna, ni pa optimalna. Če bi, na primer, naloga zahtevala 100-kratni izpis, bi koda postala zelo dolga, nepregledna in neobvladljiva (vsak morebitni popravek bi bilo treba vnesti 100-krat). Podobno kot Java (in večina drugih jezikov), programski jezik C ponuja orodja za reševanje tovrstnih problemov: *zanke*.

- zanka pomeni ponavljanje;

- C pozna tri različne zanke (`for`, `while` in `do`; glej poglavje ??);
- ukaz, ki ga želimo ponavljati, zapremo v blok zanke;
- število ponovitev `for` zanke običajno nadziramo s pomočjo števca (določimo začetno in končno vrednost ter korak); števec lahko uporabimo tudi v ukazih;
- `while` in `do` zanko nadziramo z logičnim pogojem (ki se lahko nanaša tudi na števec).

Nalogo 2-I lahko rešimo tako s pomočjo `for`

```
#include <stdio.h>
main() {
    int i;
    for (i=1; i<=5; i++) {
        printf("Pozdravljen , svet\n");
    }
}
```

Rešitev naloge 2-I.: 5-krat pozdravljen svet v C (drugič) (*pozdrav5b.c*)

kot tudi `while` zanke

```
1 #include <stdio.h>
2 main() {
3     int i=1;
4     while (i<=5) {
5         printf("Pozdravljen , svet\n");
6         i++;
7     }
8 }
```

Rešitev naloge 2-I.: 5-krat pozdravljen svet v C (tretjič) (*pozdrav5c.c*)

Medtem ko `for` zanka števec nadzira samostojno, moramo pri uporabi `while` sami paziti na:

- inicializacijo števca pred zanko,
- pravilen izstopni pogoj,
- povečevanje števca v zanki.

2.2 Preprost formatiran izpis



Naloga 2-II. Popravi program *pozdrav5b.c* tako, da bodo posamezne vrstice izpisa oštevilčene. Izpis naj bo naslednje oblike:

```
1. Pozdravljen , svet
2. Pozdravljen , svet
3. Pozdravljen , svet
4. Pozdravljen , svet
5. Pozdravljen , svet
```

Glede na dejstvo, da že sama for zanka zahteva števec, je rešitev omenjene naloge dokaj preprosta. Isti števec namreč lahko uporabimo tudi pri izpisu. Vprašanje je samo, kako (s katerim ukazom) izpišemo “živo” besedilo (za razliko od besedila “Pozdravljen, svet”, ki se ne spreminja, bo imel števec v vsaki iteraciji zanke drugo vrednost). Odgovor: z ukazom `printf`, ki ga uporabljamo za formatiran izpis.

Oglejmo si najprej preprost primer uporabe ukaza `printf`. Če želimo izpisati vsoto spremenljivk x z vrednostjo 7 in y z vrednostjo 5 v obliki $7 + 5 = 12$, to storimo tako, kot prikazuje spodnji program.

```
#include <stdio.h>
main() {
    int x = 5;
    int y = 7;
    printf("%d + %d = %d \n", x, y, x+y);
}
```

Izpis vsote spremenljivk x in y (*vsotaXY.c*)

Ukaz *printf* v programu *vsotaXY.c* vsa pojavljanja zaporedja znakov `%d` zamenja z vrednostmi spremenljivk x in y ter z vsoto $x+y$ (glej sliko 2.1). Pri klicu ukaza `printf` s prvim parametrom določimo, kakšna bo oblika (format), drugi parametri pa določijo vsebino izpisa. Število parametrov pri klicu ukaza `printf` je odvisno od formata (število parametrov, ki sledijo prvemu, je enako številu znakov `%` v prvem parametru).

Rešitev naloge 2-II. Za rešitev naloge 2-II moramo spremeniti le peto vrstico programa *pozdrav5b.c*, in se zdaj pravilno glasi takole (glej program *pozdrav5plus.c*):

```
5 | printf("%d. Pozdravljen , svet\n", i);
```

```
printf("%d + %d = %d \n", x, y, x+y);
```

Slika 2.1: Formatiran izpis z uporabo ukaza printf

2.3 Malo zahtevnejši formatiran izpis



Naloga 2-III. Napiši program, ki bo izpisal tabelo za pretvorbo prvih 32 naravnih števil med desetiškim, osmiškim in šestnajstiškim številskim sistemom. Začetek tabele naj bo takle:

Desetisko	Osmisko	Šestnajstisko
0	0	0
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7
8	10	8
9	11	9
10	12	A

Slika 2.2: Tabela za pretvorbo 10-8-16

Da bomo znali rešiti to nalogo, moramo spoznati še nekatere možnosti, ki jih ponuja ukaz `printf`. Ta namreč poleg izpisa celega števila (z zaporedjem `%d`) omogoča tudi izpis drugih tipov, med drugim: izpis realnega števila (`%f`), posameznega znaka (`%c`), niza znakov (`%s`) ter formatiranega celega števila v osmiškem (`%o`) in šestnajstiškem (`%X`) sistemu.

Ukaz `printf` uporabljamo tudi za poravnavo izpisa: zahtevamo lahko tak izpis podatkov, ki bo zasedel natančno določeno število mest (če je podatek prekratek, bo `printf` dodal presledke). Tako, na primer, z ukazom

```
|| printf("x=%5d", 10)
```

dosežemo naslednji izpis:

```
x=   10
```

(pred številom 10 so izpisani trije presledki, saj smo z `%5d` zahtevali, naj izpis cel-ega števila zasede 5 mest; ker število 10 zasede le dve mesti, ukaz `printf` doda tri presledke).

Pri izpisu realnih števil lahko določimo tudi število cifer za decimalno piko (primer: izpis števila na 3 decimalna mesta dosežemo z `%.3f`). Primer uporabe ukaza `printf` prikazuje naslednji program.

```
#include <stdio.h>

main() {
    printf ("Znaki: %c %c \n", 'a', 65);
    printf ("Cela stevila: %d %ld\n", 1977, 650000);
    printf ("Presledki pred številom: %10d \n", 1977);
    printf ("Znak '0' pred številom: %010d \n", 1977);
    printf ("Številski sistemi: %d %x %o %#x %#o,%x,%X\n",
        100, 100, 100, 100, 100, 10, 10);
    printf ("Realna stevila: %4.2f %+0e %E \n",
        3.1416, 3.1416, 3.1416);
    printf ("Spremenljiva sirina: %*d \n", 5, 10);
    printf ("Niz: %s \n", "besedilo");
}
```

Primeri izpisa s pomočjo ukaza `printf` (*printf.c*)

Ko poženemo program `printf.c`, dobimo naslednji izpis:

```
Znaki : a A
Cela  stevila : 1977 650000
Presledki pred številom :          1977
Znak '0' pred številom : 0000001977
Številski sistemi : 100 64 144 0x64 0144 , a , A
Realna stevila : 3.14 +3e+00 3.141600E+00
Spremenljiva sirina :      10
Niz : besedilo
```

Rešitev naloge 2-III. Za izpis 32 vrstic tabele bomo napisali `for` zanko, v kateri bo števec i tekkel od 1 do 32. V posamezni *iteraciji* zanke bomo i izpisali na tri načine: kot celo število (`%d`), kot osmiško celo število (`%o`) in kot šestnajstiško celo število (`%x`). Ker želimo, da so števila desno poravnana pod črko 's' v besedah Desetisko, Osmisko in Sestnajstisko (glej izpis na sliki 2.2), moramo pošteti, koliko črk vsebuje posamezna beseda. Izpis formatiramo tudi tako, da določimo število mest, ki jih posamezna beseda zasede (desetisko - 9, osmisko - 7 in sestnajstisko - 13). Pravilen izpis (števila so poravnana dva znaka od konca posamezne besede) dosežemo z ukazom `printf` v peti vrstici spodnjega programa.

```

1 #include <stdio.h>
2
3 main() {
4     int i;
5     printf("Desetisko | Osmisko | Sestnajstisko\n");
6     for (i=1; i<=32; i++)
7         printf("%7d    | %5o    | %11X\n", i, i, i);
8 }

```

Rešitev naloge 2-III.: Izpis tabele za pretvorbo (*izpis10-8-16.c*)



Izziv 2-I. Napiši program za izpis tabele s poštevanke, kot prikazuje spodnja slika.

	1	2	3	4	5	6	7	8	9	10
1	1	2	3	4	5	6	7	8	9	10
2	2	4	6	8	10	12	14	16	18	20
3	3	6	9	12	15	18	21	24	27	30
4	4	8	12	16	20	24	28	32	36	40
5	5	10	15	20	25	30	35	40	45	50
6	6	12	18	24	30	36	42	48	54	60
7	7	14	21	28	35	42	49	56	63	70
8	8	16	24	32	40	48	56	64	72	80
9	9	18	27	36	45	54	63	72	81	90
10	10	20	30	40	50	60	70	80	90	100

2.4 Argumenti programa



Naloga 2-IV. Napiši program, ki izpiše svoje ime (ukaz, s katerim je bil pognan)

in vse argumente, ki so bili podani ob klicu.

Ob klicu programa iz lupine lahko uporabnik poda *argumente*, s katerimi določa način delovanja programa. Pri nekaterih programih so argumenti obvezni (program *man*, klican brez argumentov, ne ve, katero stran naj prikaže), nekateri programi argumentov ne uporabljajo (na primer program *clear*), nekateri programi pa lahko delajo z argumenti ali brez njih - program *ls*, klican brez argumentov, izpiše seznam vseh datotek trenutnega direktorija, ukaz *ls *.c* pa izpiše vse datoteke s končnico *.c* (običajno so to programi, pisani v jeziku C). V zadnjem primeru (*ls *.c*) nizu **.c* pravimo *prvi argument* (vsem morebitnim nadaljnjim pa drugi, tretji, argument).

Primer klica brez (*clear*), z enim (*ls*) in z dvema (*cp*) argumentoma

```
clear
ls *.c
cp program.c nov.c
```

Če želimo v programu, pisanem v jeziku C, prebrati argumente, ki so bili podani ob klicu, moramo primerno spremeniti glavo funkcije *main*:

```
main() ==> main(int argc, char *argv[])
```

Če je funkcija *main* deklarirana na tak način, se pred začetkom izvajanja programa v celoštevilsko spremenljivko *argc* zapiše število argumentov (+1), v tabelo *argv* pa se shranijo vsi argumenti.

Pozor: spremenljivka *argc* ima vedno vrednost ≥ 1 (če ima vrednost 1, pomeni, da ob klicu ni bilo argumentov, če ima vrednost 2, pomeni, da je bil podan en argument, vrednost 3 pomeni, da sta bila podana dva argumenta, in tako naprej).

```
argc == 1 ... 0 - ni argumentov
argc == 2 ... 1 - en argument
argc == 3 ... 2 - dva argumenta
```

V *argv[0]* (prvi element tabele *args*) je vedno zapisano ime programa, v drugih elementih tabele, to je v

```
args[1], args[2], ..., argv[argc-1],
```

pa so zapisani argumenti.

Upoštevajoč predstavljena dejstva lahko sedaj zapišemo rešitev naloge 2-IV. Bodite pozorni: števec v *for* zanki (vrstica 8) teče od 1 do *argc-1*.

```

1 #include <stdio.h>
2
3 main(int argc, char *argv[]) {
4     int i;
5
6     printf("Ukaz: %s \n", argv[0]);
7     printf("Stevilo podanih argumentov: %d \n", argc - 1);
8     for (i=1; i<=argc-1; i++)
9         printf("Argument %d: %s\n", i, argv[i]);
10 }

```

Rešitev naloge 2-IV.: Izpis imena in argumentov (*args.c*)



Naloga 2-V. Napiši program *racunalo*, ki izračuna in izpiše vsoto prvih dveh argumentov (primer: ob klicu `racunalo 5 7` naj program izpiše `5 + 7 = 12`).

Program potrebuje dva argumenta, zato moramo pred začetkom izvajanja preveriti, ali jih je uporabnik podal. Če jih ni podal, ga moramo podučiti, kako se program uporablja (podobno delajo vsi sistemski ukazi - ob napačnem klicu se izpiše navodilo za uporabo) in končati. To storimo z naslednjimi ukazi.

```

    if (argc != 3) {
        printf("Program za izpis vsote dveh števil. \n");
        printf("Uporaba: %s x y \n", argv[0]);
        exit(1);
    }

```

Ukaz `exit(1)` konča izvajanje programa. Izhodna koda programa (prvi parameter pri klicu funkcije `exit`) se sporoči klicatelju (na primer lupini). Običajno koda 0 pomeni, da se je program končal brez napake, koda, ki je večja od 0, pa pomeni številko napake. Funkcija `exit` je deklarirana v `stdlib.h`, kar prevajalniku sporočimo z ukazom `#include <stdlib.h>`.

Drugi del programa (ki se izvrši le v primeru, da je uporabnik podal dovolj argumentov) vzame prva dva argumenta in ju sešteje. Pri tem se pojavi naslednji problem: argumenti so tipa *niz* (v jeziku C je to tabela znakov, glej poglavje 5.5), seštevamo pa lahko le števila. Pred seštevanjem moramo torej niza *pretvoriti* v cela števila. V ta namen uporabimo funkcijo `atoi`, ki je v `stdlib.h` deklarirana takole:

```
int atoi(const char *s);
```

Funkcija `atoi` torej prejme en parameter (*niz*) in vrne število.

```


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

main(int argc, char *argv[]) {
    if (argc != 3) {
        printf("Program za izpis vsote dveh števil. \n");
        printf("Uporaba: %s x y \n", argv[0]);
        exit(1);
    }


    int x = atoi(argv[1]);
    int y = atoi(argv[2]);
    printf("%d + %d = %d\n", x, y, x+y);
}

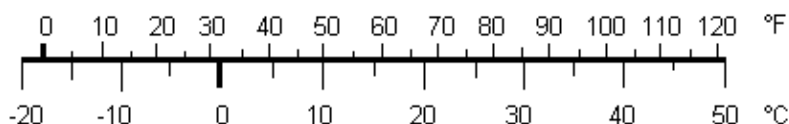
```

Rešitev naloge 2-V.: Vsota prvih dveh argumentov (*racunalo.c*)

Izziv 2-II. Popravi program *izpis-10-8-16.c* iz poglavja 2.3 tako, da bo uporabnik velikost tabele določil s prvim argumentom pri klicu programa. 

2.5 Tabela Farenheit - Celsius

Naloga 2-VI. Napiši program, ki izpiše tabelo za pretvorbo med stopinjami Farenheit in stopinjami Celzija. 



Slika 2.3: Pretvorba med °F in °C

Preden se lotimi reševanja naloge, pogledjmo, kaj sploh je stopinja Farenheit. Wikipedia nas podučí, da v Ameriki za merjenje temperature namesto stopinj Celzija (kot je to v navadi v Evropi in marsikje drugod) uporabljajo stopinje Farenheit. Ena stopinja Celzija (1 °C) je 1.8 stopinj Farenheit (°F), formula za pretvorbo pa se glasi takole:

$$^{\circ}C = (^{\circ}F - 32)/1.8.$$

Sedaj, ko poznamo formulo, se moramo odločiti, kakšna naj bo tabela (navodilo naloge nam pri tem pušča precej svobode). Če bomo tabelo uporabljali za pretvorbo

iz stopinj Celzija v stopinje Farenheit, potem potrebujemo drugačno, kot če bi pretvarjali v obratni smeri. Prav tako se moramo odločiti za zgornjo in spodnjo mejo tabele ter za korak. Naša odločitev: izpisali bomo tabelo za pretvorbo iz stopinj Farenheit v stopinje Celzija. Začeli bomo pri 0°F končali pri 120°F, korak bo 10°F (glej sliko 2.3). Študent lahko za vajo poljubno spreminja zgornjo in spodnjo mejo tabele ter korak.

Nalogo bomo rešili s pomočjo zanke, katere števec `f` bo začel pri 0 in se bo povečeval za 10, dokler ne bo prišel do 120. Spremenljivka `f`, ki jo bomo uporabili za števec, bo imela le celoštevilске vrednosti, zato bo tipa `int`.

```
int f;
for (f=0; f<=120; f=f+10)
```

Na vsakem koraku zanke bomo vrednost števca `f` (to bodo stopinje Farenheit) pretvorili v stopinje Celzija po formuli $c=(f - 32)/1.8$. Spremenljivka `c` pa bo imela realne vrednosti, zato bo tipa `double`.

Da bo izpis posamezne vrstice tabele pregleden, bomo uporabili ukaz `printf` - vrednosti `f` in `c` bomo izpisovali tako, da bosta na zaslonu zasedla vsak po 5 znakov, vrednost v stopinjah Celzija bomo zaokroževali na 2 decimalki natančno.

```
printf("%5d | %5.2f\n", f, c);
```

Vse ugotovitve zdaj lahko združimo v naslednji program.

```
#include <stdio.h>


main() {
    int f;
    double c;


    // izpis glave tabele
    printf("%6c | %6c\n", 'F', 'C');
    printf("-----\n");

    // glavna zanka programa
    for (f=0; f<=120; f=f+10) {
        c=(f - 32)/1.8;

        // izpis posamezne vrstice tabele
        printf("%6d | %6.2f\n", f, c);
    }
}
```


Rešitev naloge 2-VI.: Pretvorba med stopinjami Farenheit in Celzija (*fc.c*)

Izziv 2-III. Popravi program *fc.c* tako, da bo velikost tabele (spodnjo mejo, zgornjo mejo in korak) določil uporabnik s prvimi tremi argumenti. 

Izziv 2-IV. Popravi program *fc.c* tako, da bo uporabnik preko argumenta podal število decimalnih mest izpisa (v programu *fc.c* smo izpisali dve decimalni mesti). 

Izziv 2-V. 

Napiši program *sit2eur.c* za pretvorbo med valutama SIT in EUR (1 EUR = 239,64 SIT). Program lahko uporabljamo na dva načina:

- ob klicu brez argumentov program izpiše tabelo za pretvorbo
- ob klicu z dvema argumentoma program preveri vrednost drugega argumenta; če je ta enak 'SIT', program opravi pretvorbo iz SIT v EUR, sicer iz EUR v SIT.

Primer klicev in izpisa:

```
sit2eur 1 EUR -> izpis: 1 EUR = 239,64 SIT
sit2eur 250 SIT -> izpis: 250 SIT = 1,04 EUR
```

2.6 Naključna števila

Naloga 2-VII. Napiši program za naključno izpolnjevanje osnovnega loto listka. 

Preden se lotimo reševanja katerekoli naloge, moramo natančno razumeti, kaj naloga od nas sploh zahteva! Ne moremo namreč napisati programa (navodil za delo), če sploh ne vemo, kaj je cilj. Poskusimo torej ugotoviti, kaj je cilj naloge 2-VII.

Osnovni loto listek je listek, na katerem prekrizamo 7 števil med 1 in 39. Naključno izpolnjevanje listka pomeni, da števila prekrizamo brez posebne logike (zapremo oči in križamo kar nam pride pod svinčnik). Računalnik bi nam pri tem lahko pomagal tako, da bi na zaslon izpisal 7 naključnih števil med 1 in 39, mi pa bi potem ta števila na listku prekrizali.

Za preprosto rešitev naloge potrebujemo `for` zanko (števec bo tekkel od 1 do 7), v posamezni iteraciji te zanke pa moramo izpisati eno naključno število. Kako pa pridemo do teh naključnih števil? Oglejmo si način razmišljanja, ki privede do rešitve:

- Naključnim številom se po angleško reče *random numbers*, verjetno se bo ukaz za generiranje naključnih števil imenoval podobno; morda `rand`?
- Poskusimo srečo: v lupini vtipkamo **man 3 rand**
- Iz opisa ukaza `rand`, ki ga dobimo z zgornjim ukazom razberemo
 - funkcija `rand()` je deklarirana v `stdlib.h`,
 - razpon naključnih števil, ki jih vrne `rand()`, je med 0 in `MAX RAND`.
- Iz radovednosti preverimo, kako veliko je število `MAX RAND`

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

main() {
    printf("RANDMAX = %d\n", RANDMAX);
}
```

Odgovor, ki ga dobimo, se glasi: `MAX RAND = 2147483647`,

- Funkcija `rand()` torej vrača naključno število med 0 in 2147483647. Kako bomo ta obseg zmanjšali na obseg, ki bo zadostil našim potrebam (1..39)? Odgovor je preprost: z uporabo ostanka pri deljenju s 40. Namreč, če je x naključno število med 0 in 2147483647, bo $x\%39$ naključno število med 0 in 38, $(x\%39) + 1$ pa naključno število med 1 in 39.
- Sedaj se lahko lotimo prvotnega problema in napišemo prvo različico programa `loto.c`:

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

main() {
    int i;
    for (i=1; i<=7; i++) {
        printf("%d ", (rand() % 39 + 1));
    }
}
```

- Če program poženemo večkrat, ugotovimo, da vedno vrne ISTI rezultat (14 23 25 5 15 2 26). Je to naključje? Ne, to je lastnost funkcije `rand()`.
- Za rešitev problema znova pogledamo man stran ukaza `rand()`, kjer opazimo še opis funkcije `srand()`. Generator naključnih števil potrebuje t.i. *seme*, ki mu ga zagotovimo s klicem funkcije `srand()`. Pri istem semenu vedno vrača ista “naključna” števila. Privzeto seme je 1.
- Kako bi zagotovili seme, ki bi bilo ob vsakem klicu našega programa različno? Odgovor: uporabimo čas!
- Vtipkajmo **man 2 time** in ugotovimo, da ukaz `time` vrne število sekund, ki so pretekle od 1. januarja 1970. Z uporabo tega ukaza bomo očitno res vedno dobili različno seme!

Napišimo sedaj program za izpolnjevanje loto listka.

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

main() {
    int i;

    srand(time(NULL));
    for (i=1; i<=7; i++) {
        printf("%d ", (rand() % 39 + 1));
    }
    printf("\n");
}
```

Rešitev naloge 2-VII.: Naključno izpolnjevanje loto listka (*loto.c*)

Če večkrat poženemo program `loto.c`, ugotovimo naslednje:

- Če med dvema zaporednima zagonoma programa mine manj kot sekunda, se bo rezultat ponovil. Razlog je seveda v semenu, ki se (v našem programu) spreminja “le” vsako sekundo.
- Program ne izpiše vedno 7 različnih števil - med izpisanimi števili sta včasih dve (ali več) enaki.
- Izpis števil ni urejen.

Zadnjih dveh hib programa se bomo lotili v naslednjih poglavjih (glej nalogo 5-IV in izziv 5-I).

2.7 Branje podatkov



Naloga 2-VIII. Napiši program za računanje povprečja vpisanih ocen. Podrobneje: program naj na zaslon izpiše besedilo `Vpiši oceno:` in počaka, da uporabnik vpiše oceno in pritisne Enter. Takrat naj program vpisano oceno prebere in jo prišteje vsoti vseh prebranih ocen. Ta postopek naj se ponavlja, dokler uporabnik ne vpiše ocene 0 (signal za konec vpisovanja). Takrat naj program izračuna povprečje vpisanih ocen (skupna vsota deljeno s številom vpisanih ocen) in ga izpiše.

- nalogo bomo rešili s pomočjo `do` zanke – izstopni pogoj bo vezan na vpisano številko,
- potrebovali bomo tri spremenljivke: `vsota` (vsota do sedaj vpisanih ocen), `stevilo_ocen` (število do sedaj vpisanih ocen) `ocena` (pomožna spremenljivka, v katero bomo prebrali posamezno oceno),
- **pozor:** spremenljivki `vsota` in `stevilo_ocen` moramo pred uporabo **OBVEZNO** inicializirati (postaviti na nič)
- pri računanju povprečja moramo biti pozorni, saj uporabimo deljenje (preprečiti moramo možnost deljenja z nič),
- za branje podatkov bomo uporabili funkcijo `scanf`;
- **zapomnimo si:** pri uporabi funkcije `scanf`, moramo pred parametri pisati znak `&`:

<code>// NAROBE!!!</code>	<code>// PRAVILNO</code>
<code>scanf("%d", i);</code>	<code>scanf("%d", &i);</code>

(za razlago glej poglavje 7.4).

```
main() {
    int vsota=0;
    int stevilo_ocen=0;
    int ocena;

    do {
        printf("Vpiši oceno: ");
        scanf("%d", &ocena);
        if (ocena != 0) {
```

```
        stevilo_ocen++;
        vsota=vsota+ocena;
    }
} while (ocena != 0);
if (stevilo_ocen != 0) {
    printf("Povprecje: %.2f \n",
        (1.0 * vsota / stevilo_ocen));
} else {
    printf("Povprecje: 0 \n");
}
}
```

Rešitev naloge 2-VIII.: Izpis povprečne ocene (*ocene.c*)

Pozor: pri ukazu

```
    printf("Povprecje: %.2f \n",
        (1.0 * vsota / stevilo_ocen));
```

smo vsoto množili z 1.0, da smo prevajalnik prisilili, da izvaja realno aritmetiko (sicer bi bil rezultat celoštevilski, torej brez decimalk)!

Poglavje 3

Izrazi in operatorji

3.1 Aritmetični operatorji

- Unarni: -, +
- Binarni: *, / ,

```
int a=8, b=2, c;  
  
c = (a + b) * 3; // c = 30;  
c = a / b;      // 4 (celo"stevilsko deljenje)  
c = 33 \% 5;    // 3 (ostanek po deljenju s 6)
```

3.2 Relacijski operatorji

- < : manjše
- <= : manjše ali enako
- > : večje
- >= : večje ali enako
- == : enako
- != : različno

Aritmetični operatorji imajo prednost pred relacijskimi (na primer, $i < a-1$ pomeni $i < (a-1)$).

3.3 Logični operatorji

- `!` : negacija
- `&&` : konjunkcija (logini in)
- `||` : disjunkcija (logini ali)

Logične operatorje uporabljamo za sestavljanje logičnih izrazov. Logični izraz ima vrednost 0, kadar je neresničen ter > 0 , kadar je resničen.

```

if (a)      ... isto kot ...   if (a != 0)
if (!a)    ... isto kot ...   if (a == 0)

printf("%d, %d", (4 < 5), (13 == 42)); // 1, 0

```

Izračun logičnih izrazov se konča takoj, ko je znana končna vrednost ($A1 \ \&\& \ .. \ \&\& \ A_n$ ima končno vrednost 0 takoj, ko se najde prvi izraz A_i z vrednostjo 0).

3.4 Operatorja ++ in --

Argumentu prištej (odštej) ena. Obstajata v pre- in post-fiksni obliki.

```

... isto kot ...

i++;                               i = i + 1;

x = ++n;                            n = n + 1; x = n;
x = n++;                             x = n; n = n + 1;

t[i++] = a;                          t[i] = a; i = i + 1;

```

3.5 Bitni operatorji

- `&` : in (ali sta prižgana oba bita)
- `|` : ali (ali je prižgan vsaj en bit)
- `^` : ekskluzivni ali (ali je prižgan natanko en bit)

- `<<`: pomik v levo (isto kot množenje z dva)
- `>>`: pomik v desno (isto kot deljenj z dva)
- `~` : eniški komplement (spremeni vrednost vseh bitov)

Vsi operatorji delujejo po istoležnih bitih.

Primer bitnih operacij (glej po stolpcih)

01011	01011	01011	01011	~01011
+00101	&00101	00101	^00101	
=10000	=00001	=01111	=01110	=10100

Naloga 3-I. Napiši funkcijo, ki vrne število prižganih bitov v podanem parametru. 

V zanki bomo preverili, če je prižgan zadnji bit podanega števila (`(x & 1)`), nato bomo število pomaknili v desno (`x = x >> 1`). Zanko bomo ponavljali, dokler ne bo število enako 0.

```

1 | main(int argv, char *args[]) {
2 |     // stevilo, v katerem stojemo bite (prvi argument)
3 |     int x=atoi(args[1]);
4 |
5 |     // stevilo prizganih bitov
6 |     int i=0;
7 |
8 |     while (x > 0) {
9 |         // ali je prizgan zadnji bit?
10 |         if (x & 1) i++;
11 |
12 |         // x pomaknem v desno
13 |         x = x >> 1;
14 |     }
15 |
16 |     printf("Število bitov v %d je %d", atoi(args[1]), i);
17 | }
```

Program prešteje število bitov v podanem številu (*biti.c*)

3.6 Operator ,

Z operatorjem , (vejica) lahko združimo več izrazov v en izraz. Natančneje, če sta *i1* in *i2* dva izraza, potem se izraz

i1, *i2*

izvrši tako, da se najprej izvrši *i1*, nato *i2*, rezultat celotnega izraza pa je enak rezultatu izraza *i2*.

Nekaj primerov uporabe operatorja , je prikazanih v spodnjem programu.

```

1 main() {
2     int i, j;
3
4     // dva stevca v zanki (i raste, j pada)
5     for (i=1, j=10; i<=10; ++i, --j) {
6         printf("i = %02d  j = %02d\n", i, j);
7     }
8
9     // nenavadno, vendar pravilno prirejanje
10    i=(j=2,4);
11    printf("%d, %d \n", i, j);
12
13    // dva stavka v zanki brez bloka
14    for(i=1; i<=10; i++)
15        printf("Prvi stavek, ..."),
16        printf("... pa se drugi\n");
17 }
```

Primer uporabe operatorja , (*vejica.c*)

3.7 Operator ?:

Namesto

```

if (pogoj)
    izraz1;
else
    izraz2;
```

skrajšano zapišemo

```
|| pogoj ? izraz1 : izraz2;
```

Primer: ukaz

```
|| printf("%s", x < 37 ? "OK" : "VROCINA!");
```

v primeru, da ima spremenljivka *x* vrednost manjšo od 37 izpise OK, sicer izpise VROCINA.

3.8 Prireditveni stavki

Prireditveni stavek je izraz, ki ima vrednost. Primer: vrednost prirejanja

```
|| b = a;
```

je *a*. To dejstvo uporabimo v večkratnem prirejanju

```
|| c = b = a;
```

ali, na primer, takole:

```
|| while ((c=getchar()) != '\n') {  
||     ...  
|| }
```

Razlaga: prirejanje `c = getchar()` ima rezultat (vrednost funkcije `getchar()`); pogoj bi sicer lahko napisali tudi brez prirejanja (`while (getchar() != '\n')`) toda v tem primeru v telesu zanke ne bi vedeli, kateri znak smo prebrali.

3.9 Prioriteta in asociativnost operatorjev

Se bo v $(d = a + b * c)$ najprej izračunala vsota ali produkt?

Odgovor: Najprej se bo izračunal produkt, saj ima operator $*$ večjo prioriteto kot operator $+$.

V kakšnem vrstnem redu se bo izračunal izraz $(d = a / 4 / 2)$?

Odgovor: ker je operator $/$ levo-asociativen, je zgornji izraz ekvivalenten izrazu $d = (a / 4) / 2$.

Operatorji	Asociativnost
<code>() [] -> .</code>	L
<code>! ~ ++ -- + - * & sizeof</code>	D
<code>* / %</code>	L
<code>+ -</code>	L
<code><< >></code>	L
<code><, <=, >=, ></code>	L
<code>== !=</code>	L
<code>&</code>	L
<code>^</code>	L
<code> </code>	L
<code>&&</code>	L
<code> </code>	L
<code>?:</code>	D
<code>=, +=, -=, *=, /=, %=, &=, =, <<=, >>=</code>	D
<code>,</code>	L

Poglavje 4

Strukturiranje kode

Med izvajanjem programa se ukazi izvajajo v točno določenem vrstnem redu. Ta je odvisen od poteka izvajanja, od vrednosti prebranih podatkov, od uporabnikovih ukazov in podobno. Pri večjih programih je število izvršenih ukazov zelo veliko (tudi do več 100 tisoč). Zelo težko (oziroma skoraj nemogoče) bi bilo vse ukaze večjega programa napisati preprosto enega za drugim - zaporedje bi bilo zelo dolgo, nepregledno, možnost napake bi bila zelo velika, vzdrževanje kode zahtevno.

Kodo večjih programov obvladamo s t.i. *strukturiranim programiranjem*. Pri načrtovanju programa iščemo enake (ali podobne) dele kode, ki se jih da združiti v večkrat uporabno celoto. Pri tem si pomagamo z vejitvami (**if-else** ter **switch** stavek in pogojni izraz **E1 ? E2 : E3**), skoki (ukaz **goto**), zankami (**for**, **while** ter **do**) ter funkcijami. Pri večjih programih kodo razdelimo v več datotek. Dele programa, ki logično spadaja skupaj, pišemo v svojo datoteko (na primer: vse kar se tiče branja podatkov pišemo v eni, vse v zvezi z izpisom podatkov v drugi, vse matematične funkcije pa v tretji datoteki).

4.1 Ukaz **if-else**

```
if (a==1)          if (x==5) {          if (q<5) {
    b=1;           c=1;d=3;           d=3;
                  } else {           } else if (q<2) {
                  c=4;              d=2;
                  }                 } else {
                                      d=1;
                                      }

```

4.2 Zanke do, while in for

<code>int i=0;</code>	<code>int i=0;</code>	<code>int i;</code>
<code>do {</code>	<code>while (i<=10) {</code>	<code>for (i=1;i<=10;i++) {</code>
<code>i=i+1;</code>	<code>i=i+1;</code>	<code>....</code>
<code>....</code>	<code>....</code>	<code>}</code>
<code>} while (i<10)</code>	<code>}</code>	

Z ukazom `continue` dosežemo začetek izvajanja naslednje iteracije zanke. Ukaz `break` prekine izvajanje zanke, program se nadaljuje s prvim ukazim, ki zanki sledi.

4.3 Ukaz goto

Ukaz `goto` se uporablja za skok na označeno mesto v programu (labela). Ukazu `goto` se je dobro izogibati (nepregledni programi).

```
main() {
  zacetek:
  printf("Dobro mi gre! \n");
  goto zacetek;
}
```

4.4 Ukaz switch



Naloga 4-I. Napiši program, ki prebere oceno in izpiše "Odlicno" za 10, "Prav dobro" za 9,

Nalogo lahko rešimo vsaj na dva načina: z večkratno uporabo `if-else` stavkov (bralec naj nalogo na ta način reši za vajo) ali z uporabo `switch` stavka.

```
1 #include <string.h>
2 #include <stdlib.h>
3
4
5 main() {
6   int ocena;
7   printf("Vpisi oceno: "); scanf("%d", &ocena);
```

```
8
9  switch( ocena ) {
10     case 10 : printf( "Odlicno\n" );
11     case 9  : printf( "Zelo dobro\n" );
12     case 8  : printf( "Dobro\n" );
13     case 7  : printf( "Se kar\n" );
14     case 6  : printf( "Se bo treba bolj potruditi\n" );
15     default : printf( "Vpisana ocena ni prava\n" );
16 }
17 }
```

Izpis opisa za oceno (*switch.c*)

Zgornji program ne dela povsem pravilno – če, na primer, uporabnik vpiše oceno 10, mu program izpiše vsa besedila: Odlicno, Zelo dobro, Dobro, ... Ko namreč program enkrat “vstopi” v izvajalni del `switch` ukaza, ga samodejno ne zapusti več. Prekinitev izvajanja programer doseže z ukazom `break`. Pravilno napisan `switch` ukaz, se glasi takole:

```
switch( ocena ) {
    case 10 : printf( "Odlicno\n" );    break;
    case 9  : printf( "Zelo dobro\n" ); break;
    case 8  : printf( "Dobro\n" );     break;
    case 7  : printf( "Se kar\n" );     break;
    case 6  : printf( "Se bo treba bolj potruditi\n" );
                break;
    default : printf( "Vpisana ocena ni prava\n" );
}
}
```

4.5 Funkcije

- program v C je sestavljen iz funkcij,
- preprosti programi imajo eno samo funkcijo (`main()`), zahtevnejši jih imajo več,
- čeprav je teoretično možno vsak program napisati v eni sami funkciji, je priporočljivo, da se ga razbije na več manjših,
- nekaj majhnih delov programa je lažje vzdrževati kot enega velikega, poleg tega lahko posamezne manjše dele uporabim večkrat ali celo v različnih programih,

- funkcija v C je bolj splošen pojem kot matematična funkcija; funkcijo v C lahko obravnavamo tudi kot samostojen manjši program,
- poleg rezultata, ki ga funkcija vrne, ima funkcija lahko tudi stranski učinek (izpis na zaslon, sprememba katere od zunanjih spremenljivk, ...).
- v podpis funkcije spadajo: ime, spisek in tip parametrov ter tip rezultata,
- imena funkcij v C so enolična (ne glede na tip in količino parametrov)

Primer funkcije:

```
double sredina(double x, double y) {
    return (x+y)/2;
}
```



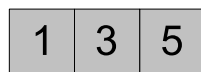
Naloga 4-II. Napiši program, ki izpiše vsoto prvih dveh argumentov (glej nalogo 2-V). Program naj NE uporablja vgrajene funkcije za pretvorbo niza v število.

Postopek za pretvorbo bomo morali napisati sami: v zanki se bomo “sprehodili” čez številke podanega števila in na vsakem koraku izračunali novo delno vsoto (prejšnjo delno vsoto bomo pomnožili z deset in prišteli novo številko), kot prikazuje slika.



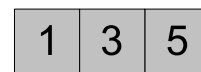
$$x = 10 * x + 1 = 1$$

(0)



$$x = 10 * x + 3 = 13$$

(10)



$$x = 10 * x + 5 = 135$$

(130)

Upoštevati moramo še, da je `argv[1]` niz, torej zaporedje znakov. Posamezen znak moramo zato pretvoriti v številko. Uporabili bomo dejstvo, da v C z znaki lahko računamo. Velja, na primer, `'3'-'0' == 3`, saj je v tabeli znakov znak `'3'` za tri mesta oddaljen od znaka `'0'`. V splošnem torej velikost številke dobimo tako, da od njene znakovne predstavitve odštejemo znak `'0'`.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 main(int argc, char *argv[]) {
5     int i;
```



```

6   char *s;
7
8   // namesto int x = atoi(argv[1]);
9   int x=0;
10  s = argv[1];
11  i = 0;
12  while (i<strlen(s)) {
13      x = 10*x + (s[i++]-'0');
14  }
15
16  // namesto int y = atoi(argv[2]);
17  int y=0;
18  s = argv[2];
19  i = 0;
20  while (i<strlen(s)) {
21      y = 10*y + (s[i++]-'0');
22  }
23
24  printf("%d + %d = %d\n", x, y, x+y);
25 }

```

Rešitev naloge 4-II.: Računanje vsote (brez uporabe `atoi()`) (*racunalo2.c*)

V programu smo morali pretvorbo med nizom in številom izvesti dvakrat, zato smo tudi kodo za pretvorbo napisali dvakrat. Z uporabo funkcij se takemu podvajanju lahko ognemo. Pred funkcijo `main()` deklarirajmo funkcijo `int stevilka(char *s)`, v kateri napišemo podvojeno kodo. V funkciji `main()` funkcijo `stevilka` kličemo podobno, kot smo v *racunalo.c* klicali funkcijo `atoi`.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int stevilka(char *s) {
5      int i = 0, x=0;
6      while (i<strlen(s)) {
7          x = 10*x + (s[i++]-'0');
8      }
9      return x;
10 }
11
12 main(int argc, char *argv[]) {
13     // namesto int x = atoi(argv[1]);
14     int x=stevilka(argv[1]);
15 }

```

```

16 // namesto int y = atoi(argv[2]);
17 int y=stevilka(argv[2]);
18
19 printf("%d + %d = %d\n", x, y, x+y);
20 }

```

pretvorbo izvedemo v funkciji `stevilka` (*racunalo3.c*)



Izziv 4-I. Popravi program `fc.c` (glej nalogo 2-VI na strani 23) tako, da bo pretvorba potekala v funkciji (napiši funkcijo `c2f` z enim parametrom (temperaturo v stopinjah Celzija) in vrne temperaturo v stopinjah Fahrenheit; v `for` zanki potem namesto formule uporabi funkcijo).

4.6 Program v več datotekah

- program v C je lahko napisan v več datotekah
- pri prevajanju je potrebno vse datoteke povezati v celoto
- funkcija ali spremenljivka se imenuje *zunanja*, če je vidna v vseh datotekah programa
- vse C funkcije so zunanje (C ne dovoljuje pisanja funkcij znotraj funkcij)
- vse spremenljivke, ki so deklarirane izven vsake funkcije, so zunanje
- program v C je množica zunanjih spremenljivk in funkcij



Naloga 4-III. Napiši program za preprosto kodiranje besedila. Natančneje: napiši funkciji `kodiraj` in `odkodiraj`, ki zakodirata in odkodirata podano besedilo ter preprost program za preizkus delovanja teh funkcij. Funkciji naj bosta napisani v eni datoteki, glavni program pa v drugi.

Kodiranja se lahko lotimo na več načinov, mi bomo izbrali najpreprostejšo možnost. Besedilo bomo zakodirali tako, da bomo kodirali posamezne znake po spodnji tabelci:

Originalna crka: A B C D E F G H ... V W X Y Z
 Zakodirana crka: D E F G H I J K ... Y Z A B C

Kodiramo torej tako, da posamezni črki prištejemo konstanten zamik (v zgornjem primeru 3). Omenjeno operacijo izvajamo po modulu 26 (toliko je črk angleške abecede).

```

|| niz[i] = 'A' + (niz[i] - 'A' + zamik) % 26;

```

Odkodiranje potem poteka v obratni smeri in sicer tako, da črkam odštevamo isti konstantni zamik.

```
niz[i] = 'A' + (26 + niz[i] - 'A' - zamik) % 26;
```

V datoteki *kodiranje.c* napišemo funkcije *kodiraj* in *odkodiraj*.

```
1 // Preprosto kodiranje besedila
2
3 // Kodiranje: vsakemu znaku pristevamo konstanto
4 // Odkodiranje: vsakemu znaku odstejemo konstanto
5 // odstejem konstanto
6
7 // Primer: zamik=3
8 // Original: A B C D E F G H ... V W X Y Z
9 // Kodirano: D E F G H I J K ... Y Z A B C
10
11 // Operaciji pristevanja in odstevanja delamo
12 // po modulu 26 (26 crk abecede)
13
14 static int zamik = 3;
15
16 void kodiraj(char *niz) {
17     int i;
18     for(i=0; i<strlen(niz); i++)
19         niz[i] = 'A' + (niz[i] - 'A' + zamik) % 26;
20 }
21
22 void odkodiraj(char *niz) {
23     int i;
24     for(i=0; i<strlen(niz); i++)
25         niz[i] = 'A' + (26 + niz[i] - 'A' - zamik) % 26;
26 }
```

Funkciji za kodiranje in dekodiranje (*kodiraj.c*)

Glavni program (*tajno.c*) prebere niz in ga zakodira, nato pa še odkodira.

```
1 main() {
2     char niz[100];
3     while(1) {
4         // preberem niz ...
5         printf("Vpisi niz: "); scanf("%s", niz);
6
7         // in ga zakodiram
8         kodiraj(niz);
9         printf("Zakodiran niz: %s\n", niz);
10
11        // niz odkodiram v prvotno obliko
12        odkodiraj(niz);
13        printf("Originalni niz: %s\n", niz);
14    }
15 }
```

Rešitev naloge 4-III.: Kodiranje in dekodiranje besedila (*tajno.c*)

Ker je program pisan v dveh datotekah, moramo prevajalniku povedati, naj datoteki *tajno.c* in *kodiranje.c* prevaja skupaj.

Prevajanje in izvajanje programa za kodiranje in dekodiranje

```
[user@localhost]# gcc -o tajno tajno.c kodiranje.c
[user@localhost]# tajno
Vpisi niz: TEST
Zakodiran niz: WHW
Originalni niz: TEST
```

Poglavje 5

Podatkovni tipi in konstante

C pozna 5 osnovnih podatkovnih tipov:

Tip	Pomen	Predvideno število bajtov
void	nedoločeno	0
char	znak	1
int	celo število	4
float	realno število (enojna natančnost)	4
double	realno število (dvojna natančnost)	8

Za razliko od Java, kjer za vsak tip natančno poznamo njegovo velikost (koliko bajtov spomina zasede spremenljivka tega tipa), velikost posameznih podatkovnih tipov v C ni natančno določena. Odvisna je od prevajalnika, s katerim program prevajamo, in od računalnika, na katerem prevajalnik poženemo. V zadnjem stolpcu zgornje tabele je prikazano najpogostejša možnost za 32-bitne računalnike, na katero pa se ne smemo povsem zanašati. Kadar je velikost podatkovnega tipa ključnega pomena za naš program, velikost preverimo z operatorjem `sizeof()`:

```
printf("Velikost tipa int je: %d\n", sizeof(int));
```

Tip `char` uporabljamo za znakovne vrednosti ('a', 'X', ...), tip `int` za celoštevilске vrednosti (15, -300), tip `float` za realne vrednosti (3.14, 239.640, ...), tip `double` pa za realne vrednosti dvojne natančnosti. Spodnji primer (program `types.c`) prikazuje uporabo štirih osnovnih tipov. Bodite pozorni na izpis števila pi - v prvem primeru smo za izredno natančno vrednost tega števila uporabili tip `float`, v drugem pa `double`. Poženite program in primerjajte oba izpisa - kateri je bolj pravilen?

```
#include <stdio.h>
main() {
    char znak='A';
```

```

int  stevilo_studentov = 250;
float pi_f=3.141592653589793238462643383;
double pi_d=3.141592653589793238462643383;

printf("Prvi znak abecede je '%c'\n", znak);
printf("Stevilo studentov: %d\n", stevilo_studentov);
printf("Pi = %.16f\n", pi_f);
printf("Pi = %.16f\n", pi_d);
}

```

Uporaba osnovnih podatkovnih tipov (*types.c*)

5.1 Osnovni številski podatkovni tipi

Osnovnim številskim tipom v C lahko spremenimo (razširimo) pomen s pomočjo enega od naslednjih določil: `short`, `long`, `unsigned`, `signed`. Vsi možni številski podatkovni tipi so tako:

- `int`, `unsigned int`, `signed int`, `short int`, `unsigned short int`, `signed short int`, `long int`, `signed long int`, `unsigned long int`,
- `float`, `double`, `long double`.

Pri uporabi teh določil je `int` privzet tip, zato, na primer, `long int` pomeni isto kot samo `long`.

Z `short` in `long` prevajalniku povemo, da želimo zmanjšati oziroma povečati velikost posameznega tipa. Kaj točno za posamezen prevajalnik pomeni, na primer, `short int`, ni natančno določeno, predpisani so le okviri, ki se jih morajo prevajalniki držati:

- tip `short int` mora biti vsaj 16-biten,
- tip `long int` mora biti vsaj 32-biten,
- `sizeof(short int) <= sizeof(int) <= sizeof(long int)`,
- vsi realni podatkovni tipi vsebujejo vsaj števila v območjih $[-10^{+38}, -10^{-38}]$ in $[10^{-38}, 10^{+38}]$ ter število 0,
- števila tipa `float` imajo natančnost vsaj 6, števila `double` in `long double` pa vsaj 10 decimalnih mest,

- obseg in natančnost tipa `double` sta vsaj taka kot pri tipu `float` (podobno velja za `double` in `long double`).
- `sizeof(float) <= sizeof(double) <= sizeof(long double)`.

Z določiloma `signed` in `unsigned` povemo, ali želimo delati s predznačenim ali z ne-predznačenim tipom. Če je, na primer, `short int` dvo-bajtni tip, potem v spremenljivki tipa `signed short int` hranimo vrednosti od -32768 do 32767, v spremenljivki tipa `unsigned short int` pa vrednosti od 0 do 65535.

Spodnja tabela prikazuje najverjetnejše nastavitve za posamezne osnovne tipe, če uporabljamo računalnik z 32-bitnim procesorjem.

Tip	Bajti	Obseg: od	do	
<code>short int</code>	2	-32.768	+32.767	(32kb)
<code>unsigned short int</code>	2	0	+65.535	(64kb)
<code>unsigned int</code>	4	0	+4.294.967.295	(4Gb)
<code>int</code>	4	-2.147.483.648	+2.147.483.647	(2Gb)
<code>long int</code>	4	-2.147.483.648	+2.147.483.647	(2Gb)
<code>float</code>	4			
<code>double</code>	8			
<code>long double</code>	12			

Natančni podatki o obsegih in ostale konstante so zapisani v `limits.h` (v tej knjižnici so v obliki konstant in makrojev navedene vse omejitve vezane na konkretno implementacijo jezika C) in `floats.h` (konstante in makroji vezani na implementacijo realnih tipov).

Naloga 5-I. Napiši program, ki izpiše velikost in obseg celoštevilskih tipov `short`, `int` in `long` (obravnavaj obe možnosti – `signed` in `unsigned`). 

Za določitev velikosti bomo uporabili funkcijo `sizeof()`, za določitev obsega pa vnaprej definirane konstante `INT_MAX`, `INT_MIN`, ... (glej `limits.h`).

```
#include <limits.h>
#include <float.h>

main() {
    printf("Tip                | St. bajtov    "
           " | OD                | DO\n");
    printf("_____ "
           "_____ \n");
}
```

```

printf("%15s | %5d | %14d | %14d\n",
    "signed int", sizeof(signed int), INT_MIN, INT_MAX);
printf("%15s | %5d | %14u | %14u\n",
    "unsigned int", sizeof(unsigned int), 0, UINT_MAX);
printf("%15s | %5d | %14hd | %14hd\n",
    "signed short", sizeof(signed short), SHRT_MIN, SHRT_MAX);
printf("%15s | %5d | %14hu | %14hu\n",
    "unsigned short", sizeof(unsigned short), 0, USHRT_MAX);
printf("%15s | %5d | %14ld | %14ld\n",
    "signed long", sizeof(signed long), LONG_MIN, LONG_MAX);
printf("%15s | %5d | %14lu | %14lu\n",
    "unsigned long", sizeof(unsigned long), 0, ULONG_MAX);
}

```

Rešitev naloge 5-I.: Velikost in obseg celoštevilskih tipov (*sizerange.c*)

Številске konstante

```

int a= 45;           // a = 45
unsigned short b=-1; // b = 65535
int c=012;          // c = 10 (osmisko)
int d=0xFF;         // d = 255 (sestnajstisko)

float x = 3.14;     // 3.14
float y = 3e4;      // 3x10^4 = 30000

```

5.2 Znaki

Znake hranimo v spremenljivkah tipa `char`. Za razliko od Jave, kjer je `char` 16-bitni tip (65536 različnih znakov), v C `char` običajno zavzame le 8 bitov (256 različnih znakov).

Znaki in števila so medseboj zelo povezani. Vsakemu znaku pripada eno število in obratno. Za pretvorbo med znaki in števili uporabljamo ASCII tabelo, v kateri so definirani pomeni za prvih 127 števil in sicer:

- 0-31: kontrolni znaki
- 32: presledek
- 48..57: števke (od '0' do '9')

- 65..90: velike črke (od 'A' do 'Z')
- 97..122: male črke (od 'a' do 'z')

Povezava med števili in znaki je tako velika, da pri delu s števili in znaki lahko izmenično uporabljamo ene in druge, C pa bo sam poskrbel za pravilno pretvorbo. Z znaki lahko tudi računamo.

```
char a = 65; // a = 'A' (ASCII tabela)

char b = 'A', c = 'D';
int i = c - b; // i = 3 ('D' - 'A' = 3)

char p = ' ';
printf("ASCII koda presledka je: %d", p);
```


Znakovne konstante

Znakovne konstante lahko določimo tudi s pomočjo t.i. *ubežnih sekvenc* – zaporedja dveh ali več znakov, ki imajo poseben pomen:

Znak	Pomen	Znak	Pomen
<code>\n</code>	prehod v novo vrstico	<code>\t</code>	tabulator
<code>\b</code>	pomik nazaj	<code>\r</code>	pomik na začetek vrstice
<code>\f</code>	nova stran	<code>\a</code>	zvočni signal
<code>\\</code>	znak <code>\</code>	<code>\?</code>	znak <code>?</code>
<code>\'</code>	znak <code>'</code>	<code>\"</code>	znak <code>"</code>
<code>\0xx</code>	znak v osmiškem zapisu	<code>\xhh</code>	znak v šestnajstiškem zapisu

```
char a = 'A', // znak 'A'
      b = '\n', // nova vrsta
      c = '\a', // zvonček
      d = '\040', // ' ' - 40(OCT) = 32(DEC)
      e = '\x30'; // '0' - 30(HEX) = 48(DEC)
```

5.3 Tabele (polja)

Naloga 5-II. Iz standardnega vhoda preberi tri števila in izpiši njihovo povprečje  in standardni odklon.

Da bomo lažje razumeli navodila naloge, si najprej oglejmo definicijo matematičnih pojmov *povprečje* in *odklon*. Recimo, da imamo zaporedje $x_1, x_2, x_3, \dots, x_N$, potem sta povprečje \bar{x} in standardni odklon σ definirana takole:

$$\bar{x} = \frac{\sum_{i=1}^N x_i}{N}, \quad \sigma = \sqrt{\frac{\sum_{i=1}^N (\bar{x} - x_i)^2}{N}}.$$

(povprečje pove povprečno vrednost členov zaporedja, standardni odklon pa povprečno odstopanje posameznega člena zaporedja od povprečja).

Računanja povprečja se lahko lotimo tako, da enkrat preberemo zaporedje, sproti seštevamo njegove člene, na koncu pa dobljeno vsoto delimo s številom prebranih členov (glej nalogo 2-VIII na strani 28). Podoben postopek bi radi uporabili tudi za računanje standardnega odklona, toda pri tem naletimo na težavo: zaporedje moramo prebrati dvakrat - prvič zato, da izračunamo povprečje, drugič zato, da posamezne člene zaporedja odštevamo od povprečja, kvadriramo in kvadrate seštevamo. Dvojno branje v praksi ni zaželeno (predstavljajte si, da mora uporabnik, če želi izračunati standardni odklon, dvakrat vtipkati celotno zaporedje), zato bomo nalogo rešili tako, da si bomo prebrane podatke zapomnili.

Prebrane člena zaporedja bomo shranili v spremenljivke `x1`, `x2`, `x3`. Uvedli bomo še štiri spremenljivke: `vsota1` (vsota vseh členov zaporedja - vrednost vsote iz prve formule), `vsota2` (vsota kvadratov razlik povpračja in posameznih členov - vrednost vsote iz druge formule) ter `pov` (povprečje) in `odk` (standardni odklon). Za računanje kvadrata bomo uporabili funkcijo `pow(base, exp)`, za računanje korena pa `sqrt(num)` (obe sta deklarirani v knjižnici `math.m`).

```

1 | main() {
2 |     float x1, x2, x3;
3 |     float vsota1, vsota2, pov, odk;
4 |
5 |     printf("Vpisi 1. stevilo: "); scanf("%f",&x1);
6 |     printf("Vpisi 2. stevilo: "); scanf("%f",&x2);
7 |     printf("Vpisi 3. stevilo: "); scanf("%f",&x3);
8 |
9 |     // racunanje povpracja
10 |    vsota1 = x1 + x2 + x3;
11 |    pov    = vsota1 / 3;
12 |
13 |    // racunanje standardnega odklona
14 |    vsota2 = pow((pov - x1),2)
15 |              + pow((pov - x2),2)
16 |              + pow((pov - x3),2);

```


```

17 |     odk = sqrt(vsota2 / 3);
18 |
19 |     printf("Povprecje: %.2f, Odklon: %.2f", pov, odk);
20 | }

```

Rešitev naloge 5-II.: Povprecje in standardni odklon (prvič) (*odklon1.c*)

Nalogo 5-II smo sicer rešili, toda rešitev ni najboljša, saj bi potrebovali kar precej časa, če bi želeli program spremeniti tako, da bi nam računal povpračje in odklon, na primer, desetih prebranih števil: namesto treh vrstic za branje (vrstice 5,6,7) bi imeli 10 takih vrstic, namesto vsote treh spremenljivk v 10. in 14. - 16. vrstici bi imeli vsoto desetih spremenljivk, namesto s tri bi v 11. in 17. vrstici delili z deset. Spremembe sicer niso zelo zahtevne, obstaja pa velika verjetnost, da bi se programer, ki bi program popravljaj, zmotil (verjetnost napake bi se seveda povečala, če bi želeli program prilagoditi za delo s 100 številkami). Poleg tega je program neuporaben, če ga moramo spremeniti vsakič, ko se spremeni sorazmerno majhna stvar (število vhodnih podatkov). Omenjeno slabost bomo odpravili v naslednji nalogi.

Naloga 5-III. Popravi program *odklon1.c* tako, da bo število vpisanih podatkov (med 1 in 50) lahko določil uporabnik (in ne programer, kot je bilo to v *odklon1.c*). 

Pri reševanju naloge 5-III bomo potrebovali več spremenljivk - če uporabniku dovolimo vpis do 50 števil, potrebujemo 50 spremenljivk: $x_1, x_2, \dots, x_{49}, x_{50}$. Toda, že deklaracija petdesetih spremenljivk (vrstico 2, v kateri smo deklarirali tri spremenljivke moramo razširiti tako, da bodo v njej deklarirane vse omenjene spremenljivke) je duhamorno opravilo, rezultat pa je nepregleden:

```

float x1 , x2 , x3 , x4 , x5 , x6 , x7 , x8 , x9 , x10 , x11 , x12 , x13 ,
      x14 , x15 , x16 , x17 , x18 , x19 , x20 , x21 , x22 , x23 , x24 ,
      x25 , x26 , x27 , x28 , x29 , x30 , x31 , x32 , x33 , x34 , x35 ,
      x36 , x37 , x38 , x39 , x40 , x41 , x42 , x43 , x44 , x45 , x46 ,
      x47 , x48 , x49 , x50 ;

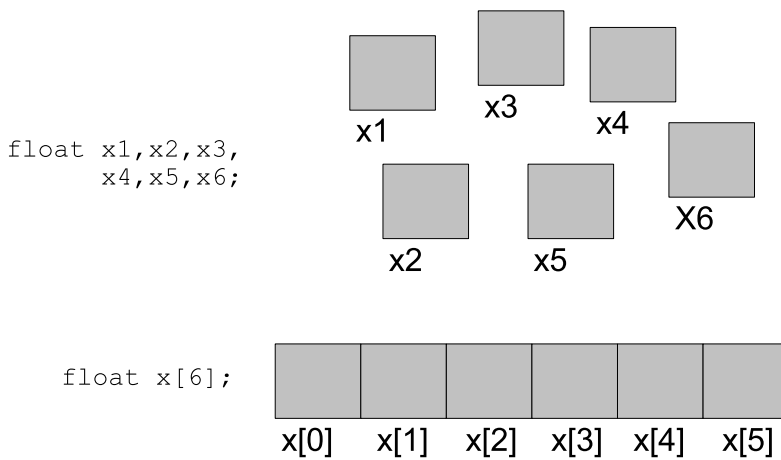
```

Tak pristop k reševanju zato ni najboljši. Za 50 spremenljivk potrebujemo **kompaktnější** (in s tem tudi bolj učinkovit) zapis. C nam v ta namen ponuja *tabele*, ki jih lahko razumemo tudi kot orodje za združevanje spremenljivk. Namesto petdesetih spremenljivk bomo imeli eno samo spremenljivko tipa tabela, ki bo združevala vse spremenljivke.

```

float x [ 50 ] ;

```



Slika 5.1: Deklaracija spremenljivk - brez in z uporabo tabele (shematski prikaz)

Z enim samim ukazom smo deklarirali tabelo, ki združuje 50 spremenljivk: `x[0]`, `x[1]`, `x[2]`, ..., `x[49]`. Pri uporabi “spremenljivk iz tabele” je treba biti pozorni na dvoje:

- indeksi se pišejo v oglatem oklepaju:

namesto `x1` pišemo `x[1]`

- indeksi se začnejo pri 0:

namesto `x1, x2, ..., x50` pišemo `x[0], x[1], ..., x[49]`

Združevanje spremenljivk v tabeli je shematsko prikazano na sliki 5.1.

Program `odklon1.c` lahko sedaj popravimo tako, da bo omogočal vpis do 50 števil (glej program `odklon2.c`):

- vrstico 2 popravimo tako, da namesto treh spremenljivk deklariramo tabelo s petdesetimi polji,
- dodatno deklariramo tri spremenljivke: `i` (indeks elementov v tabeli `x`), `n` (število prebranih elementov) in `f` (pomožna spremenljivka),
- vrstice 5, 6, in 7 zamenjamo z zanko, v kateri preberemo števila; pogoj zanke (`i < 50`) je izpolnjen, dokler uporabnik ne vpiše 50 števil; uporabnik lahko ustavi branje tudi tako, da vpiše število 0 (v tem primeru se bo izvršil ukaz `break` v 18. vrstici),

- v 21. vrstici v **n** shranimo število prebranih števil (to je enako vrednosti indeksa **i**, saj smo ga po vsakem branju v 16. vrstici povečali za ena),
- popravimo še vrstice 10-11 in 14-17: računanje povprečja in standardnega odklona izvedemo v zanki, v kateri namesto treh upoštevamo vseh **n** elementov.

```
1 main() {
2     float x[50];
3     float vsota1, vsota2, pov, odk;
4
5     int i=0;    // stevec za indekse v tabeli x
6     int n;     // stevilo prebranih števil
7     float tx;  // pomocna spremenljivka za branje
8
9     // beremo stevila, dokler uporabnik ne vpise 0
10    while (i<50) { // preberemo največ 50 števil
11        printf("Vpisi %d. stevilo: ", i+1); scanf("%f",&tx);
12        // ce je prebrano stevilo razlicno od 0,
13        // ga vpisemo v tabelo x na i-to mesto, ...
14        if (tx != 0) {
15            x[i] = tx;
16            i++;
17        } else // ... sicer končamo z branjem!
18            break;
19    }
20    // zapomnimo si, koliko števil smo prebrali
21    n = i;
22
23    // ce nismo prebrali niti enega števila, končamo!
24    if (n==0)
25        exit(0);
26
27    // racunanje povpracija
28    vsota1=0;
29    for (i=0; i<n; i++)
30        vsota1=vsota1+x[i];
31    pov    = vsota1 / n;
32
33    // racunanje standardnega odklona
34    vsota2=0;
35    for (i=0; i<n; i++)
36        vsota2=vsota2+pow((pov - x[i]),2);
37    odk = sqrt(vsota2 / n);
```

```

38 |
39 |     printf("Povprecje: %.2f, Odklon: %.2f", pov, odk);
40 | }

```

Rešitev naloge 5-III.: Povprecje in standardni odklon (drugič) (*odklon2.c*)



Naloga 5-IV. Popravi program *loto.c* (glej nalogo 2-VII na strani 25) tako, da se izpisana števila ne bodo ponavljala.

Nalogo bomo rešili takole: najprej bomo izbrali prvo naključno število in si ga zapomnili; potem bomo izbrali še 6 števil in sicer tako, da bomo *i*-to število izbirali toliko časa, dokler ne bo različno od vseh prejšnjih izbranih števil. Izbrana števila bomo najlažje hranili v tabeli.

```

1 | #include <stdio.h>
2 | #include <stdlib.h>
3 | #include <time.h>
4 |
5 | main() {
6 |     // pomocna stevca
7 |     int i, j;
8 |
9 |     // ali izbrano stevilo ze obstaja
10 |    // 0 ... se ne obstaja, 1 ... ze obstaja
11 |    int ze_obstaja;
12 |
13 |    // tabela izbranih števil
14 |    int tabela[7];
15 |
16 |    // nastavimo seme
17 |    srand(time(NULL));
18 |
19 |    // izberemo 7 števil
20 |    for (i=0; i<=6; i++) {
21 |        do {
22 |            tabela[i]=(rand() % 39 + 1);
23 |
24 |            // izbrano stevilo primerjam z ze izbranimi
25 |            ze_obstaja=0;
26 |            for(j=0; j<i; j++)
27 |                if (tabela[i]==tabela[j])
28 |                    ze_obstaja=1;
29 |
30 |        } while (ze_obstaja==1);


```


```

31 |     printf("%d ", tabela[i]);
32 | }
33 |
34 |
35 |     printf("\n");
36 | }

```

Rešitev naloge 5-IV.: Izpis števil za loto (brez ponavljanja) (*loto2.c*)

Izziv 5-I. Popravi program *loto2.c* tako, da bodo izpisana števila urejena po velikosti. Program naj bo sestavljen iz dveh datotek: datoteka *loto3.c* z glavnim programom in datoteka *urejanje.c*, v kateri je napisana funkcija `uredi` za urejanje tabele. 

Izziv 5-II. S pomočjo tabele implementiraj podatkovno strukturo sklad (funkcije `init`, `push`, `pop`, `je_prazen`). 

Tabelarične konstante

Tabelarične konstante podami v zavutih oklepajih.

```

short meseci [] =
    { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

```

5.4 Večdimenzionalne tabele

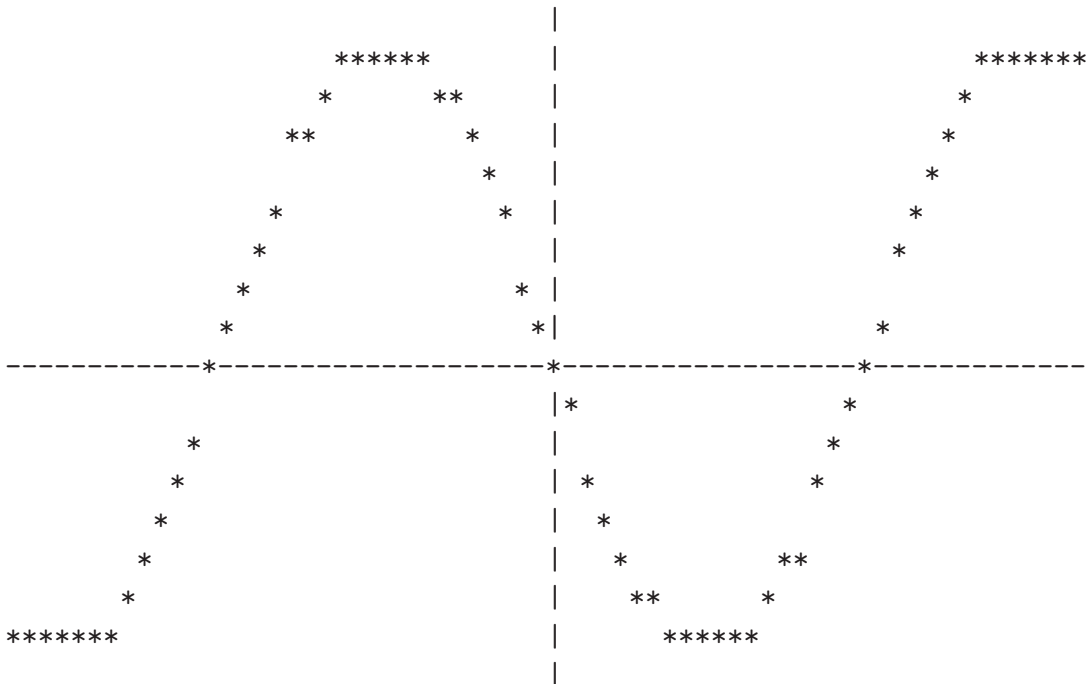
V prejšnjem poglavju smo predstavili eno-dimenzionalno tabelo. Do elementov take tabele dostopamo z enojnim indeksom: `a[0]`, `a[1]`, ... V programskem jeziku C lahko deklariramo tudi dvo ali več dimenzionalno tabelo.

3
5
1

eno-dimenzionalna
tabela (vektor)

1	0	1
0	2	2
2	0	1

dvo-dimenzionalna
tabela (matrika)



Slika 5.2: Primer izhoda programa graf.c (funkcija: $\sin(x)$)

Do elementov dvo-dimenzionalne tabele dostopamo z dvojnimi indeksom: $a[0][0]$, $a[0][1]$, $a[1][0]$, ... Celoštevilsko dvodimenzionalno tabelo velikosti 3x3 deklariramo z

```
|| int a [ 3 ] [ 3 ] ;
```



Naloga 5-V. Napiši program za risanje grafov matematičnih funkcij na tekstovnem zaslonu. Primer izhoda je prikazan na sliki 5.2.

Dejstva in način razmišljanja pri reševanju zastavljene naloge:

- pri risanju smo omejeni z zaslonom z majhno ločljivostjo, zato tudi rezultat ne bo zelo natančen,
- običajna velikost tekstovnega zaslona je 80x25 (t.j. 25 vrstic po 80 znakov),
- zaslon si bomo predstavili v spominu kot dvodimenzionalno tabelo znakov dimenzije 80x25,

```
|| char zaslon [ 80 ] [ 25 ] ;
```


- na začetku bomo vsa polja tabele “spraznili” (napolnili s presledki),

```

for (i=0; i<80; i++)
    for (j=0; j<25; j++)
        zaslون [i][j]= ' ';

```

- če bomo želeli, da se na zaslon na (i,j)-to mesto izriše nek znak, bomo ta znak zapisali v [i][j]-to polje tabele,
- na koncu bomo vsebino tabele izpisali na zaslon

```

for (j=0; j<25; j++) {
    for (i=0; i<80; i++)
        printf("%c", zaslون [i][j]);
    printf("\n");
}

```

- za bolj pregleden izpis bomo grafu dodali koordinatni sistem (abscisno in ordinatno os),

```

for (i=0; i<80; i++) zaslون [i][12]= '-';
for (j=0; j<25; j++) zaslون [40][j]= '|';

```

- glavni del programa predstavlja zanka, v kateri števec i teče od 0 do 79; za vsako vrednost i se izračuna pripadajoč j (vrednost funkcije, ki jo rišemo, v točki i); v tabeli `zaslون` se na mestu `[i][j]` nariše nek znak (na primer '*'),
- za pravilno delovanje programa moramo poskrbeti še za preslikavo med realnimi in zaslonskimi koordinatami; če bomo, na primer, risali graf funkcije $\sin(x)$ na območju od $-\pi$ do π , moramo poskrbeti, da se bo najprej i iz `[0..79]` preslikal v x iz `[- π .. π]`, in nato še, da se bo $y = \sin(x)$ iz `[-1..1]` preslikal v j iz `[0..24]`,

- če rišemo funkcijo $f(x) : [x1..x2] \mapsto [y1..y2]$, potem za preslikavo med i in x ter y in j poskrbita naslednji funkciji:

$$x = \frac{x2 - x1}{79} * i + x1, \quad j = 24 * \frac{y - y1}{y2 - y1}$$

Opomba 1: da sta navedeni funkciji res pravi, se bralec prepriča tako, da v prvo funkcijo vstavi 0 in 79 in dobi rezultat $x1$ in $x2$; podobno lahko stori z drugo funkcijo;

Opomba 2: funkciji lahko izpelje bralec tudi sam in sicer z nastavkom $x = a * i + b$, v katerega vstavi obe znani vrednosti in reši sistem dveh enačb z dvema neznankama; za izračun vrednosti j uporabi nastavek $j = c * y + d$

Oboroženi s prevkar pridobljivim znanjem lahko sedaj program za risanje grafov napišemo v celoti.

```

1 #define pi 3.14
2
3 main() {
4     int i, j;
5     float x, y;
6
7     float x1=-2*pi, x2=2*pi,
8           y1=-1.5, y2=1.5;
9
10    // deklaracija in inicializacija tabele
11    char zaslon[80][25];
12    for(i=0; i<80; i++)
13        for(j=0; j<25; j++)
14            zaslon[i][j]=' ';
15
16    // koordinatni sistem
17    for(i=0; i<80; i++) zaslon[i][12]='-';
18    for(j=0; j<25; j++) zaslon[40][j]='|';
19
20    // izracun funkcije
21    for(i=0; i<80; i++) {
22        x = (x2-x1)/79 * i + x1;
23        y = sin(x);
24        j=24*(y-y1)/(y2-y1);
25        if ((j>=0) && (j <25))
26            zaslon[i][j]='*';
27    }


```

```

28
29 // izpis rezultata
30 for (j=0; j<25; j++) {
31     for (i=0; i<80; i++)
32         printf("%c", zaslon[i][j]);
33     printf("\n");
34 }
35 }

```

Rešitev naloge 5-V.: Program za risanje matematičnih funkcij (*graf.c*)


Izziv 5-III. Popravi program *graf.c* tako, da bo omogočal interaktivno uporabo (parametre x_1 , x_2 , y_1 , y_2 in funkcijo bo lahko določil uporabnik med samim izvajanjem programa), kot prikazuje spodnja slika. 

Interaktivni del programa za risanje grafov matematičnih funkcij

```

[user@localhost]# ./ graf
Vpisi x1: -3.14
Vpisi x2: 3.14
Vpisi y1: -1
Vpisi y2: 1
Izberi funkcijo:
0 .. sin , 1 ... cos , 2 ... tan , 3 ... e^x

```

Izziv 5-IV. Napiši program, ki bo dvema igralcema omogočal igranje igre Tic-Tac-Toe. Igralec, ki je na vrsti, vpiše x in y koordinati svoje poteze. Program posamezno potezo vpisuje v celoštevilsko tabelo velikosti 3×3 (0 ... polje je prazno, 1 ... na polju je krogec, 2 ... na polju je križec). Po vsaki vpisani potezi naj program izpiše trenutno stanje na igralni plošči. Preprostejša različica programa naj odločanje o zmagi prepusti igralcema, zahtevnejša različica pa naj po vsaki potezi preveri, ali je bila to zmagovalna poteza in primerno ukrepa. 

Več-dimenzionalne tabelarične konstante

n -dimenzionalno tabelo inicializiramo kot tabelo $(n - 1)$ -dimenzionalnih tabel.

```

int y[3][3] = {
    {7, 2, 1}, /* y[0][0], y[0][1], y[0][2] */
    {3, 9, 4}, /* y[1][0], y[1][1], y[1][2] */
    {5, 8, 6} /* y[2][0], y[2][1], y[2][2] */
};

```

5.5 Nizi

Nekaj dejstev o nizih:

- C ne pozna posebnega podatkovnega tipa za delo z nizi; niz v C je tabela znakov,
- niz v C se konča z znakom `'\0'`,
- spremenljivko, v kateri lahko hranim nize dolžine $< N$ (N je neka konkretna številka), deklariram z

```
|| char niz[N];
```

- ob deklaraciji lahko niz tudi inicializiram

```
|| char niz[50]="Besedilo, ki je krajše od 50 znakov";
|| char niz[]="Poljubno dolg niz";
```

Razlika med zgornjima deklaracijama je v velikosti rezerviranega prostora; v prvem primeru se rezervira prostor za 50 znakov in v prve celice tabele vpiše podani niz (zadnjih nekaj celic tabele je "praznih"), v drugem primeru pa se rezervira natanko toliko prostora, kot je dolžina podanega niza. Program

```
|| char niz1[10]="ABC";
|| char niz2[]="DEF";
|| printf("%d, %d\n", sizeof(niz1), sizeof(niz2));
```


zato izpiše 10, 4 (10 zato, ker smo tako zahtevali, 4 pa zato, ker je dolžina niza "ABC" enaka 4 - trije znaki + znak `'\0'`),

- primer deklaracije niza dolžine do 5 znakov, inicializacije in rezultata je prikazan na spodnji sliki

```
char niz[6]="TEST";
```

T	E	S	T	\0	
0	1	2	3	4	5

- i -ti znak niza dobimo kot i -ti element tabele (prvi znak niza `niz` je `niz[0]`, drugi `niz[1]`, zadnji pa `niz[strlen(niz)-1]`).

Naloga 5-VI. Napiši program, ki izpiše dolžino niza (pri tem naj ne uporabi funkcije `strlen`). 

Niz, katerega dolžino bomo računali, deklariramo in inicializiramo z

```
|| char niz[100] = "Ugotovi mojo dolzino!";
```

S števcem i se bomo z `while` zanko "sprehodili" po tabeli `niz` in iskali znak `'\0'`. Ko ga bomo našli, bo vrednost spremenljivke i enaka dolžini niza.

```
1 | main(int argc, char *args[]) {
2 |     // niz, katerega dolzino bomo ugotavljali
3 |     char niz[100] = "Ugotovi mojo dolzino!";
4 |
5 |     // indeks po znakih niza zacne na prvem znaku
6 |     int i=0;
7 |
8 |     // i povecujem, dokler ne najdem znaka '\0'
9 |     while ((i<100) && (niz[i] != '\0')) i++;
10 |
11 |     printf("Dolzina niza '%s' je %d", niz, i);
12 | }
```

Rešitev naloge 5-VI.: Izpis dolžine podanega niza (*dolzina.c*)

Znakovne konstante

Niz inicializiramo kot tabelo znakov ali (preprosteje) z uporabo dvojnih narekovajev.

```
|| char imeA[] = {'M', 'I', 'H', 'A', 0};
|| char imeB[] = "Micka";
```

V znakovnih konstantah lahko uporabljamo tudi ubežne sekvence.

```
|| char niz[] = "Prva stevka: \x30, zvoncek: \a";
```

Funkcije za delo z nizi

Za delo z nizi C med drugimi ponuja tudi naslednje funkcije (knjižnici `stdlib.h` in `string.h`):

atof() pretvori niz v double

```
char stevilo [] = "3.1415926";
double pi = atof(stevilo);
printf("2*pi = %f", 2 * pi);
```

(izpis: 2*pi = 6.283185)

atoi() pretvori niz v int

atol() pretvori niz v long

strcat() združi (stakne) dva niza

strchr() poišče prvo pojavitev znaka v nizu

```
char niz []="To je niz";
char c = 'j';
printf("Prva pojavitev '%c' je pri '%s'",
      c, strchr(niz, c));
```

(izpis: Prva pojavitev 'j' je pri 'je niz')

strstr() poišče prvo pojavitev niza v nizu

strcmp() primerja dva niza

strcpy() skopira vsebino enega niza v drug niz

strlen() vrne dolžino niza

strncpy() skopira določeno število znakov enega niza v drug niz (podobno: `strncat()`, `strncmp()`).

```
char niz1 [10]="ABCDEF";
char niz2 [10]="abcdef";
strncpy(niz2, niz1, 3);
printf("%s", niz2);
```

(izpis: ABCdef)



Izziv 5-V. Program *dolzina.c* predelaj tako, da bo namesto for zanke uporabljal funkcijo `strlen()`.



Izziv 5-VI. Uporabi vsako od zgoraj omenjenih funkcij v primeru.



Naloga 5-VII. Napiši program, ki prebere dva niza (ime in priimek uporabnika) in z enim samim klicom funkcije `strlen()` izračuna dolžino obeh.

Potrebujemo dve spremenljivki - `niz1` in `niz2` za hranjenje imena in priimka. Podatka preberemo s funkcijo `gets()`, ki prebere eno vrstico iz standardnega vhoda.


```
char ime [100];
gets (ime);
```

Pozor: pri uporabi ukaza `gets` je treba biti izredno previden, saj je spremenljivka, v katero beremo, vedno omejene velikosti, prebrani podatki pa so načeloma lahko poljubno dolgi (v našem primeru bi lahko naleteli na hude težave, če bi bilo vpisano ime daljše od 100 znakov).

Ker lahko funkcijo `strlen()` uporabimo le enkrat, je najbolje, če prebrana niza združimo v en niz. Z ukazom `strcat(ime, priimek);` nizu `ime` pripnemo niz `priimek`. Dolžino obeh nizov sedaj dobimo z ukazom `strlen(ime)`.

```
1 #include <string.h>
2 #include <stdio.h>
3 main() {
4     char ime[100], priimek[100];
5
6     printf("Vpisi ime:      "); gets(ime);
7     printf("Vpisi priimek: "); gets(priimek);
8
9     // imenu pritaknem priimek
10    strcat(ime, priimek);
11    printf("Skupna dolzina: %d", strlen(ime));
12 }
```

Rešitev naloge 5-VII.: Skupno število črk v imenu in priimku (*nizi1.c*)

Naloga 5-VIII. Napiši program, ki s standardnega vhoda bere vrstice, dokler uporabnik ne vpiše prazne vrstice. Med branjem si program zapomni najdaljšo prebrano vrstico in jo na koncu izpiše. 

Nalogo bomo rešili s pomočjo **do** zanke, ki se bo končala, ko bo uporabnik vpisal prazno vrstico. V posamezni iteraciji zanke bomo prebrali naslednjo vrstico in jo primerjali s prejšnjo najdaljšo. Če bo trenutna vrstica daljša od prejšnje najdaljše, si jo bomo zapomnili.

```

1 #include <string.h>
2 #include <stdio.h>
3 main() {
4     char najdaljsi[100]="";
5     char trenutni[100];
6
7     do {
8         gets(trenutni);
9         if (strlen(trenutni) > strlen(najdaljsi))
10            strcpy(najdaljsi, trenutni);
11    } while (strlen(trenutni)!=0);
12    printf("%s", najdaljsi);
13 }
```

Rešitev naloge 5-VIII.: Izpis najdaljše vrstice vhoda (*nizi2.c*)

Opombi k rešitvi:

- v programu smo se omejili na vrstice dolžine 100; če bo uporabnik vpisal daljše vrstice, program ne deloval pravilno (verjetno bo prišlo do napake in program se bo končal),
- S pomočjo lupininih ukazov za preusmeritev izhoda enega programa na vhod drugega programa, lahko program *nizi2.c* uporabljamo tudi za izpis najdaljše vrstice neke datoteke.

Uporaba programa *nizi2.c* za izpis najdaljše vrstice datoteke *datoteka.txt*

```
[user@localhost]# cat datoteke.txt | ./nizi2
```

Paziti je treba le na to, da datoteka na koncu vsebuje prazno vrstico (prej pa prazne vrstice ne sme biti).

5.6 Strukture

 **Naloga 5-IX.** Napiši program, ki prebere ime, priimek in starost petih oseb.

Za shranjevanje prebranih podatkov (pet trojk (ime, priimek, starost)) imamo več možnosti. Naštejmo nekatere od njih (po vrstnem redu, od najslabše do najboljše možnosti).

Uporaba 15 spremenljivk. Za vsak podatek imamo svojo spremenljivko.

```
char ime1[10], ime2[10], ime3[10],  
     ime4[10], ime5[10];  
char priimek1[20], priimek2[20], priimek3[20],  
     priimek4[20], priimek5[20];  
int  starost1, starost2, starost3, starost4, starost5;
```

Ta rešitev je najslabša zato, ker onemogoča branje podatkov v `for` zanki in ker je zelo slabo razširljiva (kaj vse bi bilo treba storiti, če bi nalogo želeli razširiti iz 5 na 6).

Uporaba treh tabel V prvi tabeli hranimo vsa imena, v drugi priimke, v tretji starosti.

```
char ime[5][10];  
char priimek[5][20];  
int  starost[5];
```

Slabost te rešitve je, da so podatki, ki logično spadajo skupaj, fizično ločeni v tri tabele. Težave bi se pojavile, na primer, če bi želeli podatke urediti po nekem vrstnem redu (na primer, po abecedi). V tem primeru bi morali simultano urejati tri tabele (avtomatska urejanja s pomočjo vgrajenih funkcij s tem odpadejo).

Uporaba struktur Podatke o osebi, ki logično spadajo skupaj, lahko s pomočjo struktur združimo v celoto.

```
struct oseba {  
    char ime[10];  
    char priimek[20];  
    int  starost;  
};
```

Če imamo eno spremenljivko tipa `oseba`, do posameznih komponent dostopamo s pomočjo pike.

```
struct oseba o;

scanf("%s", o.ime);
scanf("%s", o.priimek);
scanf("%d", &o.starost);

printf("%s, %s, %d", o.ime, o.priimek, o.starost);
```

Za vodenje podatkov o petih osebah potrebujemo tabelo **oseb**.

```
struct osebe o[5];
```

Rešitev naloge je potem preprosta in lahko razširljiva.

```
1 struct oseba {
2     char ime[10];
3     char priimek[20];
4     int starost;
5 };
6
7
8 main(){
9     struct oseba o[5];
10    int i;
11
12    // branje podatkov
13    for (i=0; i<5; i++) {
14        printf("Vpisi ime: ");
15        scanf("%s", o[i].ime);
16        printf("Vpisi priimek: ");
17        scanf("%s", o[i].priimek);
18        printf("Vpisi starost: ");
19        scanf("%d", &o[i].starost);
20    }
21
22    // izpis podatkov
23    for (i=0; i<5; i++) {
24        printf("%s, %s, %d \n",
25            o[i].ime, o[i].priimek, o[i].starost);
26    }
27 }
```

Rešitev naloge 5-IX.: Branje podatkov o petih osebah (*struct.c*)

Strukture pogosto uporabljamo v kombinaciji z ukazom `typedef`.

```
struct kompleksno {
    double re;
    double im;
};


typedef struct kompleksno cplx;
```

ali skrajšano

```
typedef struct kompleksno {
    double re;
    double im;
} cplx;
```

Spremenljivki `w` in `z` tipa `struct kompleksno` potem deklariramo z

```
cplx w, z;
```

Izziv 5-VII. Napiši program, ki omogoča delo s kompleksnimi števili. Deklariraj strukturo `struct kompleksno` (glej zgoraj) ter funkciji `vsota` in `produkt` za računanje vsote in produkta dveh kompleksnih števil. 

```
cplx vsota (cplx x, cplx y);
cplx produkt(cplx x, cplx y);
```

Inicializacija struktur

Strukture inicializiram na dva načina: kot tabelo vrednosti ali po posameznih komponentah.

```
struct tocka {
    int x;
    int y;
};

// inicializacija s tabelo vrednosti
struct tocka b = {7,4};

// inicializacija po komponentah
struct tocka a = {.y=5, .x=2};
```

Poglavje 6

Spremenljivke

Programerja pri delu s spremenljivkami zanimajo naslednji podatki::

Podatek	Pomen
ime	do spremenljivke dostopamo preko njenega imena
tip	kakšne podatke lahko spremenljivka hrani
doseg	kje je spremenljivka vidna
velikost	količina pomnilnika rezervirana za spremenljivko
naslov	naslov v pomnilniku, kjer je rezerviran prostor za spremenljivko
vrednost	trenutna vrednost spremenljivke

O spremenljivki `x` s spodnjega primera lahko povemo naslednje: ime: `x`, tip: `int`, `doseg=znotraj` funkcije `blabla`, `naslov`: naslov na skladu, kjer je spremenljivki dodeljen prostor, `velikost`: 32 bitov (odvisno od računalnika in prevajalnika), `vrednost`: nedefinirana (vrednost `x` je lahko karkoli).

```
void blabla () {  
    int x;  
}
```

O tipih spremenljivk in količini rezerviranega prostora smo govorili v prejšnjem poglavju, zato se bomo v tem poglavju posvetili preostalim štirim podatkom.

6.1 Ime spremenljivke

Imena spremenljivk so sestavljena iz poljubnega zaporedja črk (a-z in A-Z), števk (0-9) in podčrtaja (`_`). Ime se ne sme začeti s števkjo in se mora razlikovati od rezerviranih besed.

Poleg omenjenih pravil, mimo katerih ne moremo, pa obstajajo tudi *priporočila* (Naming Conventions) – spisek pravil, ki sicer niso obvezna, so pa programi, če se programerji teh priporočil držijo, lepši, preglednejši in bolj “standardni”. Naštejmo nekatera od njih:

- imena naj ne bodo prekratka in ne predolga; dobro je, če se iz imena vidi, čemu ta spremenljivka služi,
- imena, ki se začnejo ali končajo z podčrtajem, so rezervirana za sistemska spremenljivke, zato naj bi se programerji “običajnih” programov takih imen izogibali,
- v imenih spremenljivk ne uporabljamo velikih črk (velik črke uporabljamo za imena konstant in v naštevnikih tipih),
- izogibamo se podobnim imenom, to je imenom, ki se razlikujejo samo v velikosti čke (bla in Bla), imenom, ki se razlikujejo samo v podčrtaju (blabla in bla_bla) ter imenom, ki imajo podoben vizualen izgled (črka 'l' in številka '1' sta zelo podobni),

6.2 Vidljivost (scope)

Kaj se zgodi, ko uporabnik deklarira neko spremenljivko, na primer,

```
|| int i;
```

Kje je ta spremenljivka vidna? V katerih delih programa jo lahko programer uporablja? Jo lahko uporablja tudi v drugih datotekah istega programa? Kako je z vrednostjo te spremenljivke – ko jo enkrat nastaviš, se lahko potem vedno zaneses na njeno vrednost ali spremenljivka svojo vrednosti kdaj tudi izgubi? Odgovori na ta in podobna vprašanja niso preprosti, saj je vidljivost (*angl. scope*) in življenjska doba spremenljivk odvisna od mesta in načina deklaracija spremenljivke. C preko mehanizma vidljivosti in s pomočjo rezervirane besede *static* pozna take spremenljivke, ki so vidne v vseh delih programa, kot tudi take, ki so vidne samo v omejenem delu, take, katerih vrednost se ohranja, kot tudi take, ki vrednost v določenih primerih izgubijo.

Omejevanje vidljivosti spremenljivk preprečuje konflikte v zvezi z imenovanjem (primer: velik program, več programerjev; če bi vsi videli vse spremenljivke, bi kmalu prišlo do tega, da bi dva programerja dve različni stvari poimenovala z istim imenom, in nastala bi zmeda). Po drugi strani pa je preveliko omejevanje vidljivosti problematično predvsem v primerih, ko se preko spremenljivk med deli programa

prenašajo “globalni” podatki.

Poznamo štiri nivoje vidljivosti: bločna (lokalne spremenljivke), funkcijska (omejena samo na oznake za ukaz `goto`), programska (globalne spremenljivke) in datotečna (statične globalne spremenljivke).

6.2.1 Bločna vidljivost

V primeru na prvi strani tega poglavja je v funkciji `blabla` deklarirana spremenljivka `x`. Ta spremenljivka je vidna in dosegljiva SAMO znotraj funkcije `blabla`, natančneje, znotraj bloka ki določa telo funkcije. V splošno velja, da so spremenljivke, ki so deklarirane znotraj bloka (med zavitima oklepajema `{ in }`), vidne samo znotraj tega bloka. Takim spremenljivka pravimo **lokalne spremenljivke**.

Bloki so lahko tudi gnezdeni (blok znotraj bloka). Tudi v tem primeru velja, da je spremenljivka vidna samo znotraj svojega bloka (in ne tudi v “nadbloku”), kar lahko vidimo na naslednjem primeru.

```
1 | main() { /* zacetek bloka 1 */
2 |     int i = 42;
3 |     printf("V zunanjem bloku: i=%d\n", i);
4 |
5 |     { /* zacetek bloka 2 */
6 |         int i=15; /* znotraj bloka 2 vidim samo ta i */
7 |         printf("V notranjem bloku: i=%d\n", i);
8 |     } /* konec bloka 2 */
9 |
10 |     printf("Ponovno v zunanjem bloku: i=%d\n", i);
11 | } /* konec bloka 1 */
```

Gnezdeni bloki (*gnezdeniblok.c*)

Ko smo znotraj drugega bloka redeklarirali spremenljivko `i`, smo s tem skrili zunanjo spremenljivko z istim imenom. Notranji spremenljivki `i` smo nastavili vrednost na 15, toda s tem nismo izgubili vrednosti zunanje spremenljivke. Ko tok programa zapusti notranji blok, se notranja spremenljivka “izgubi” in ponovno dosežemo zunanji `i`. Program bo zato izpisal:

Izpis programa `gnezdenibloki.c`

```
[user@localhost]# ./gnezdenibloki
V zunanjem bloku: i=42
V notranjem bloku: i=15
Ponovno v zunanjem bloku: i=42
```

6.2.2 Programska vidljivost

Spremenljivka, ki je deklarirana izven funkcij, ima programsko vidljivost. Takim spremenljivka pravimo **globalne spremenljivke**, saj so dosegljive iz vseh funkcij programa. S pomočjo globalnih spremenljivk zmanjšamo število parametrov, ki se prenašajo s klici posameznih funkcij (če ne bi uporabljali globalnih spremenljivk, bi morali vsaki funkciji, preko parametrov posredovati vse “globalne” podatke, ki jih potrebuje).

```

1 | int velikost=0;
2 |
3 | void povecaj () {
4 |     velikost++;
5 | }
6 |
7 | void pomanjsaj () {
8 |     velikost--;
9 | }
10 |
11 | main () {
12 |     povecaj (); povecaj (); povecaj ();
13 |     pomanjsaj ();
14 |     printf("Vrednost: %d", velikost);
15 | }
```

Primer uporabe globalne spremenljivke (*globalna.c*)

V zgornjem primeru je spremenljivka `velikost` deklarirana izven funkcij `povecaj`, `pomanjsaj` in `main`, zato je vidna v vseh treh funkcijah (povsod jo lahko beremo in/ali spreminjamo). Globalna spremenljivka je vidna tudi v funkcijah, ki so napisane v drugih datotekah istega programa. Če uporabljamo globalne spremenljivke, deklarirane v drugi datoteki, moramo prevajalniku to na nek način sporočiti. To storimo z uporabo rezervirane besede **extern**.



Naloga 6-I. Napiši program za štetje števila študentov, ki pridejo na izpit. Program naj sestavljata dve datoteki. V prvi (pomožna datoteka *orodja.c*) naj bo deklarirana spremenljivka `stevilo`, ki pomeni število do sedaj prispelih študentov ter funkcija `nov()`, ki jo bomo klicali vsakič, ko bo prišel nov študent. V glavni datoteki (*stetje.c*) v zanki beremo tipkovnico. Če uporabnik pritisne 'n' zabeležimo prihod novega študenta, če uporabnik pritisne 'i', izpišemo število študentov, če uporabnik pritisne 'h', končamo.

Prva datoteka (*orodja.c*) vsebuje globalno spremenljivko `stevilo` in (globalno) funkcijo `nov()`, ki poveča vrednost te spremenljivke.


```
// globalna spremenljivka
int stevilo=0;

// povecam vrednost spremenljivke stevilo
void nov() {
    stevilo++;
}
```

“Orodja” programa *stetje.c* (*orodja.c*)

V glavnem delu programa (datoteka *stetje.c*) deklariramo spremenljivko `stevilo` kot zunanje spremenljivko. Branje izvedemo v `while` zanki.

```
// spremenljivka je deklarirana v orodja.c
extern int stevilo;

main() {
    char c;

    // neskoncna zanka (koncamo z exit(0))
    while (1) {
        printf("Pritisni (N)ov, (I)zpis ali iz(h)od: ");
        do { // beremo, dokler ni vpis pravi (n, i ali h)
            c=getchar();
        } while ((c!='n') && (c!='i') && (c!='h'));

        // kaj se bo zgodilo, je odvisno od vpisa
        if (c=='n') {
            nov(); // nov student
        } else if (c=='i') {
            printf("Stevilo studentov: %d\n", stevilo);
        } else if (c=='h')
            exit(0); // koncamo z izvajanjem programa
    }
}
```

Program za štetje števila študentov (*stetje.c*)

Program prevedem z “gcc orodja.c stetje.c”.

6.2.3 Datotečna vidljivost

Pri uporabi globalnih spremenljivk moramo biti previdni, še posebej, kadar smo del večje ekipe, ki skupaj razvija nek program. Ker so globalne spremenljivke vidne povsod, lahko pride do težav, če dva programerja deklarirata vsak svojo globalno

spremenljivko z istim imenom. Takim težavam se lahko ognemo z omejevanjem uporabe globalnih spremenljivk, vendar to ni vedno rešitev. V določenih primerih uporaba globalne spremenljivke močno olajša programiranje, zato bi bilo popolna prepoved nesmiselna. Druga možnost, da se ognemo težavam, pa je z omejevanjem vidljivosti globalnih spremenljivk. Z uporabo določila `static` pri globalnih spremenljivkah (in funkcijah) vidljivost omejimo le na datoteko, v kateri so deklarirane (datotečna vidljivost). Spremenljivka, ki je kot globalna (torej izven vseh funkcij) deklarirana z

```
|| static int vrednost;
```

bo vidna v le vseh funkcijah iste datoteke, v ostalih funkcijah istega programa pa ne.

6.2.4 Avtomatske in statične lokalne spremenljivke

Lokalne spremenljivke (spremenljivke, deklarirane znotraj funkcij) “zaživijo” vsakič, ko tok programa pride v funkcijo, v kateri so deklarirane in “umrejo” vsakič, kot tok programa to funkcijo zapusti. Takim spremenljivkam zato pravimo tudi *avtomatske* spremenljivke. Njihova vrednost se med posameznimi klici funkcije NE ohranja. Če je ob deklaraciji avtomatske spremenljivke napisana tudi inicializacija (na primer: `int i=0`), se ta inicializacija izvrši vsakič, ko tok programa pride v funkcijo.

Opisano obnašanje avtomatskih spremenljivk je včasih moteče, saj bi bilo v določenih primerih bolje, če bi se vrednost lokalne spremenljivke ohranila (ob naslednjem klicu funkcije bi imela spremenljivka enako vrednost, kot jo je imela, ko se je prejšnji klic te funkcije končal). Kadar potrebujemo lokalno spremenljivko s tako lastnostjo, jo znotraj funkcije deklariramo z rezervirano besedo `static`

```
|| static int i=0;
```

Statična lokalna spremenljivka se inicializira samo enkrat in sicer pred začetkom izvajanja programa. Zato mora biti vrednost, v katero se inicializira, konstanta.

Pozor: rezervirana beseda *static* ima dvojen pomen: globalnim spremenljivkam (in funkcijam) z njo omejimo vidljivost na datotečno, lokalnim spremenljivkam pa z njo podaljšamo “življensko dobo”.

Uporabo statične lokalne spremenljivke si oglejmo na naslednjem primeru.

```

1 | bla () {
2 |     static int prvic=1; /* first is true */
3 |     if (prvic==1) {
4 |         printf("Prvi klic funkcije bla()\n");
5 |         prvic = 0;
6 |     } else {
7 |         printf("Naslednji klic funkcije bla()\n");
8 |     }
9 | }
10 |
11 |
12 | main () {
13 |     bla ();
14 |     bla ();
15 | }

```

Uporaba statične lokalne spremenljivke (*static.c*)

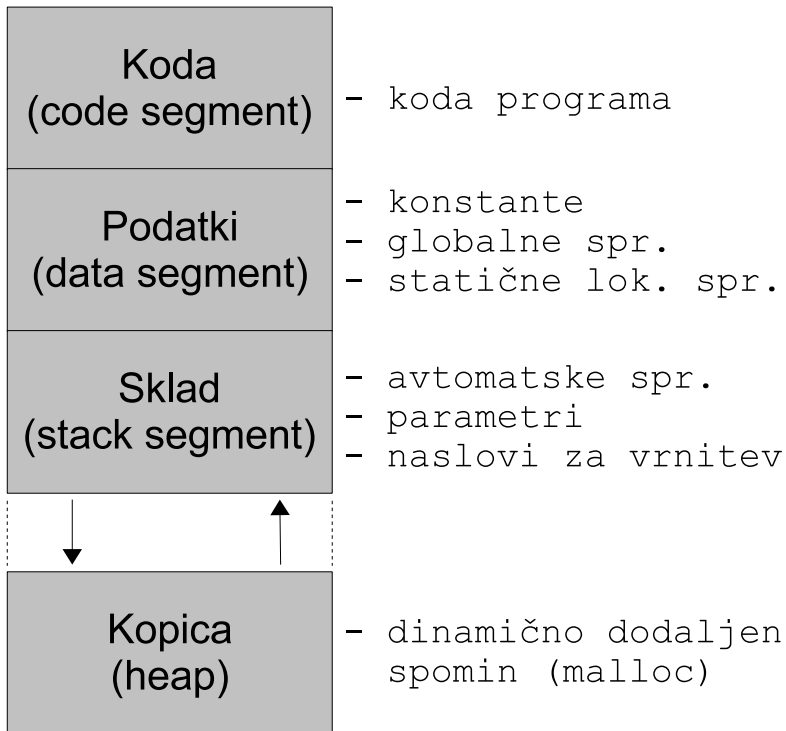
6.3 Organizacija pomnilnika

Ko požemo program, ki je napisan v programskem jeziku C, mu operacijski sistem dodeli določeno količino pomnilnika, ki je razdeljen v tri segmente (glej sliko 6.1):

Segment	Opis
Koda (Code segment)	koda programa, prevedena v zbirni jezik
Podatki (Data segment)	globalni podatki - konstante in statične spremenljivke
Sklad (Stack segment)	lokalne spremenljivke in ostali začasni podatki

Poleg omenjenega, je programu preko operacijskega sistema na razpolago dodaten spomin, ki se imenuje *kopica* (heap).

- Kodni segment včasih imenujemo tudi text segment, v njem je zapisana koda programa, ki se izvaja (prevedena v zbirni jezik),
- podatkovni segment hrani vse konstante ter globalne in statične lokalne spremenljivke,
- sklad se uporablja za lokalne (avtomatske) spremenljivke, prenos parametrov pri klicu funkcij, za shranjevanje naslovov za vrnitev iz funkci in podobno; sklad je dobro organiziran spomin brez praznih prostorov; dodajamo z ukazom `push`, brišemo s `pop`; z nepravilno uporabo (predvsem ob slabo napisanih rekurzivnih programih) lahko na skladu zmanjka prostora (takrat program ne more več nadaljevati z delom),



Slika 6.1: Organizacija spomina po segmentih

- kopica je običajno največji del pomnilnika, ki je na razpolago za dinamično uporabo (razviramo z ukazim `malloc`, sprostimo s `free`; zaradi dinamičnega dodeljevanja so lahko v kopici tudi luknje (nezaseden del spomina, ki pa je zaradi majhnosti neuporaben).



Naloga 6-II. Povej, v katerem delu pomnilnika je rezerviran prostor za posamezne dele spodnjega programa.

```

1 | const char msg[] = "Napaka!";
2 | int stevec=0;
3 |
4 | void bla(int a) {
5 |
6 |     static int i=0;
7 | }
8 |
9 | main() {
10 |     int x;
11 |     bla(5);
12 | }
```

Program za vajo (*memory.c*)

Rešitev naloge 6-II.

- prevedeni funkciji `bla` in `main` se nahajata na kodnem (text) segmentu,
- konstanta `msg`, globalna spremenljivka `stevec` ter statična lokalna spremenljivka `i` so v podatkovnem segmentu,
- lokalna spremenljivka `x` je na skladu,
- številka 5 (parameter `a` ob klicu funkcije `bla`) se zapiše na sklad.

6.3.1 Inicializacija spremenljivk

Pomembno vprašanje pri pisanju programov je, kakšno vrednost ima posamezna spremenljivka pred inicializacijo. Nekateri programski jeziki avtomatsko poskrbijo, da imajo “še neuporabljene spremenljivke” vrednost 0 (ali ekvivalent, glede na tip spremenljivke), nekateri pa tega ne storijo. Če nismo prepričani, kako je s programskim jezikom, ki ga uporabljamo, je vsekakor najbolje, da inicializacijo vedno opravimo sami. Bolje je namreč, da je inicializacija opravljena dvakrat, kot da sploh ni (v drugem primeru ne vemo, kakšno vrednost ima spremenljivka, s katero delamo). Slaba stran dvojne inicializacije pa je v počasnosti delovanja pri velikih spremenljivkah (na primer pri velikih tabelah). Le s poznavanje pravil posameznega programskega jezika dosežemo optimalno delovanje programov.

V programskem jeziku C je pravilo glede inicializacije preprosto: C poskrbi za avtomatsko inicializacijo vseh globalnih in statičnih lokalnih spremenljivk, torej vseh spremenljivk, ki so v podatkovnem segmentu. Za inicializacijo ostalih spremenljivk (tistih, ki se nahajajo na skladu in kopici), pa mora poskrbeti programer.

```
int x;          // avtomatska inicializacija (x=0)

static int y;  // avtomatska inicializacija (y=0)

int t[100];    // avtomatska inicializacija
              // (v vseh poljih tabele t so nicle)

void bla() {
    static int i; // avtomatska inicializacija (i=0)

    int j;       // NI avtomatske inicializacije
              // (v j je lahko karkoli)
```

```
int a[10];      // NI avtomstske inicializacije
                // (v poljih tabele a je lahko karkoli)

int *p;

// NI avtomstske inicializacije
// v delu spomina, ki ga rezerviramo z malloc, (torej
// v poljih p[0], p[1], ...), je lahko karkoli
p = (int *) malloc (100 * sizeof(int));
}
```

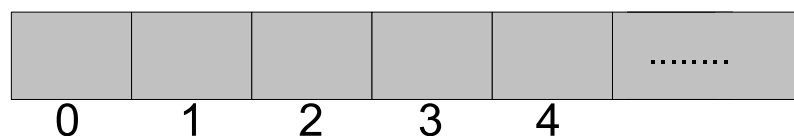
Poglavje 7

Kazalci

Kazalci (*angl. pointers*) so ena najpomembnejših komponent jezika C. Čeprav je njihova napačna uporaba vir večine napak v programih, so nepogrešljivi, saj marsikaterega problema brez kazalcev sploh ne moremo rešiti. Kazalci so tako pomemben del jezika C, da brez slabe vesti lahko zapišemo, da **brez kazalcev v C ni mogoče uspešno programirati**.

7.1 Kaj so kazalci?

Da bomo znali odgovoriti na to vprašanje, pogledjmo najprej, kako je organiziran računalniški spomin. Z malo poenostavitve lahko rečemo, da je računalniški spomin tabela spominskih celic, kot prikazuje spodnja slika.



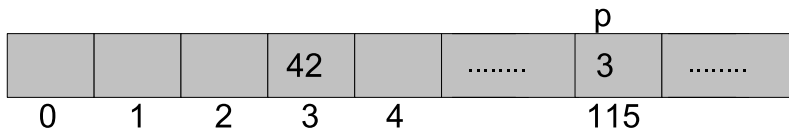
Slika 7.1: Shematska slika spominskih celic

Vsaka celica ima svoj *naslov* (na zgornji sliki: 0, 1, 2, 3, ...)

V določenih okoliščinah želimo v točno določeno spominsko celico vpisati neko vrednost. Takrat uporabimo kazalce, saj so kazalci, kot pravi uradna definicija, **spremenljivke, ki hranijo naslove spominskih lokacij**. Če bi, na primer želeli v spominsko celico številka 3 vpisati število 42, bi v programskem jeziku C postopali takole:

- najprej bi deklarirali spremenljivko `p` kot kazalec na tip `int` (tja, kamor bo kazal kazalec `p`, bomo vpisali številko tipa `int`)

- v `p` bi vpisali 3 (kazalec `p` bo s tem “pokazal” na celico številka 3)
- v celico, na katero kaže `p`, bi vpisali 42.



Prevedeno v jezik C se to glasi takole:

```
int *p; // naj bo p kazalec
p = 3; // p naj pokaze na celico številka 3
*p = 42; // v celico, na katero kaže p, vpišemo 42
```

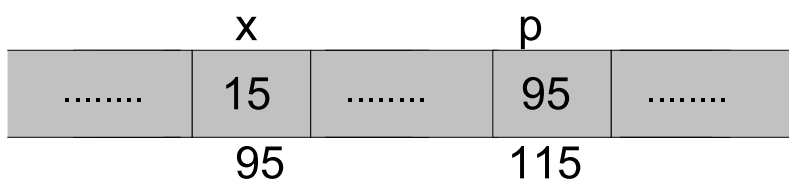
Kazalec torej deklariramo z uporabo znaka `*`:

```
int *p;
```

Tip kazalca (v zgornjem primeru `int`) pove, kakšnega tipa je celica, na katero kazalec kaže.

Z deklaracijo `int *p` še nismo povedali, kam naj `p` kaže, zato mora deklaraciji obvezno slediti inicializacija (`p=3`), šele nato lahko v celico, na katero kaže `p`, nekaj vpišemo (`*p=42`).

Direktno naslavljanje celic z absolutnim naslovom (kot v zgornjem primeru – celica številka 3) ni običajen postopek, saj absolutnih naslovov praviloma ne poznamo, oziroma niso pomembni (absolutno naslavljanje v zgornjem primeru smo uporabili zgolj za lažjo predstavbo). Bolj običajen postopek je uporaba relativnih naslovov, kot na primer: kazalec `p` naj kaže na celico, ki pripada spremenljivki `x`, ali pa, kazalec `p` naj kaže na tretjo celico tabele `a`, in podobno. Tako relativno naslavljanje izvedemo s pomočjo operatorja `&` ki vrača naslove.




```

int x;      // spremenljivka tipa int
int *p;     // kazalec na int
p = &x;     // p pokazuje na celico, ki pripada x
*p = 15;    // kamor kaže p (torej v x) vpišemo 15

```

Ko smo deklarirali `x`, se je nekje v pomnilniku rezerviral prostor (na zgornji shematski sliki se je rezervirala celica številka 95), prav tako se je rezerviral prostor za spremenljivko `p` (na sliki: celica številka 115). Z ukazom `p = &x` zahtevamo, da se v `p` vpiše “naslov” spremenljivke `x`, torej naj `p` pokaže na celico, ki pripada `x` (na sliki: v celico `p` se vpiše 95). Z `*p=15` v celico, na katero kaže `p`, vpišemo 15 (isto kot `x=15`).

7.2 Kazalci in tabele

- Kazalci in tabele so medseboj zelo povezani (površno: tabela je kazalec; kazalec je tabela),
- ime tabele je sinonim za naslov prvega elementa te tabele (če je `a` tabela, potem je `a == &a[0]`),
- velja:

```

int a[10];
int *pa;

// isto kot pa=a
pa = &a[0]; // pa pokazuje na prvi element tabele a

*pa      = 15; // isto kot a[0] = 15;
*(pa+5) = 5;  // isto kot a[5] = 5;

pa = pa + 3; // pa pomaknemo za tri polja naprej
*pa      = 7; // isto kot a[3] = 7;

```

- “`char *a`” torej lahko razumemo tudi kot tabelo znakov “`char s[]`”
- funkcijo `strlen` lahko napišemo na dva načina:

```

// strlen s tabelo
int strlen(char s[]) {
    int n=0;
    while (s[n] != '\0') {
        n++;
    }
    return n;
}

// strlen s kazalcem
int strlen(char *s) {
    int n=0;
    while (*s != '\0') {
        n++; s++;
    }
    return n;
}

```

Če imamo tabelo, jo lahko nadomestimo s kazalcem. To je sorazmerno preprosto opravilo – s kazalcem pokažem na tabelo (`pa = a;`) in potem namesto tabele uporabljam kazalec (namesto `a[i]` pisemo `*(pa+i)`).

```

int a[10];
int *pa;
pa = a;
// ... od tu naprej namesto z a delamo s pa
// ... namesto a[i] pisemo *(pa+i)

```

Kaj pa obratno: če imamo kazalec, pa bi ga radi “uporabili” kot tabelo? To je zahtevnejše opravilo, saj tabele nimamo in jo moramo šele ustvariti. Tabela “ustvarim” tako, da na kopici rezerviram dovolj prostora zanjo.

```

int *tp;

// rezerviram prostor za tabelo velikosti 100
// (tip podatkov: int)
tp = (int *) malloc (100 * sizeof(int));

tp[0] = 15; // pisem v polja tp[0] .. tp[99]
*(tp+2) = 10; // uporabim lahko tudi ta nacin

tp[150] = 13; // !!! NAPAKA !!!

```

Z ukazom `malloc(n)` rezerviram `n` bajtov prostora na kopici. Rezultat klica je kazalec na rezerviran prostor (ker je `malloc` univerzalen ukaz, s katerim rezerviram prostor za različne tipe kazalcev (`int *`, `char *`, ...), je rezultat tipa `void *`; ta rezultat je treba s pravilno pretvorbo (`(int *)` pred ukazom `malloc`) pretvoriti v pravi tip). Ukaz

```
tp = (int *) malloc (100 * sizeof(int));
```

rezervira prostor za tabelo velikost 100 (vsaka celica tabele je dovolj velika za en `int`) in v `tp` zapiše naslov rezerviranega prostora (kazalec `tp` pokaže na začetek tabele). Vrednosti v tabeli lahko sedaj nastavljam z ukazi kot sta `tp[0]=15` in `*(tp+2)=10`.

Pri tovrstnem delu s tabelami moramo biti zelo previdni. Če namreč napišemo `tp[150]=13` prevajalnik ne bo javil napake, ob izvajanju kode pa se bodo dogajale čudne stvari. S tem ukazom smo namreč pisali v del pomnilnika, ki ni namenjen nam. Rezervirali smo prostor za tabela `tp` velikosti 100, zato spominska celica na naslovu `tp+150` ni več del tabele. Ta celica je bila morda dodaljena komu drugemu (ki je zahteval spomin z ukazom `malloc`) in sedaj pišemo po njegovih podatkih. Posledice takega početja so nepredvidljive.

7.3 Dinamično delo s pomnilnikom

Ukaz `malloc` se uporablja za t.i. *dinamično* delo s pomnilnikom. Velikost tabele, ki je deklariramo z `int a[100]`, določimo v času programiranja (torej statično), velikost tabele, ki jo ustvarim z `malloc` pa določimo v času izvajanja programa (torej dinamično). Slednje je uporabno takrat, kadar v času programiranja ne vemo, kako veliko tabelo bomo potrebovali (prostor rezerviram šele takrat, ko izvem, kako velika mora biti).

Za res dinamično delo s pomnilnikom poleg ukaza, s katerim rezerviramo prostor, potrebujemo tudi ukaz, s katerim rezerviran prostor sprostim. Ukaz se imenuje `free`, uporabljamo pa ga takole:

```
int *p;  
// rezerviram prostor na kopici  
p = (int *) malloc(100 * sizeof(int));  
  
// ... uporabljam tabelo p  
  
// sprostim prostor na kopici  
free(p);
```

Če kličemo `free` s kazalcem, za katerega prej nismo rezervirali prostora, pride do napake.

Naloga 7-I. Napiši program, ki bo izračunal povprečje in standardni odklon poljub- 

nega števila vhodnih podatkov (prvi podatek, ki ga uporabnik vpiše, naj bo število podatkov, ki sledijo).

Za osvežitev spomina: nalogo 5-II, v kateri smo prvotno računali povprečje in odklon treh prebranih števil, smo razširili tako, da je uporabnik lahko vpisal do 50 števil (naloga 5-III). Na 50 podatkov smo se omejili zato, ker smo tabelo ustvarili statično. V rešitvi trenutne naloge bomo tabelo ustvarili dinamično, zato se bomo tej omejitvi lahko izognili. Najprej bomo prebrali število podatkov, šele nato bomo rezervirali prostor za tabelo.

Nalogo 7-I rešimo tako, da za osnovo vzamemo rešitev naloge 5-III (program *odklon2.c*) in jo na treh mestih spremenimo (glej program *odklon2.c*):

- namesto `int x[50]` deklariramo `int *x`
- preberemo število podatkov (`n`) in rezerviramo prostor na kopici

```
// preberemo stevilo podatkov ...
printf("Vpisi stevilo podatkov: "); scanf("%d",&n);

// ... in rezerviramo prostor za tabelo
x = (float *) malloc(n * sizeof(float));
```

- spremenimo postopek branja podatkov (sedaj lahko beremo s `for` zanko, saj natančno vemo, koliko podatkov še pričakujemo)

```
for (i=0; i<n; i++) {
    printf("Vpisi %d. stevilo: ", i+1);
    scanf("%f",&tx);
    x[i] = tx;
}
```

Čeprav predstavlja rešitev naloge 7-I bistveno izboljšavo v primerjavi s prvima dvema rešitvama (saj dovoljuje neomejen vpis podatkov), z rezultatom še ne moremo biti zadovoljni. Rešitev namreč od uporabnika zahteva, da število podatkov pozna že ob začetku vpisovanja, kar pa je lahko huda ovira (morda uporabnik tega podatka še nima (podatki med vpisovanjem še prihajajo) ali pa je podatkov veliko in je štetje zamudno). Program bi radi popravili tako, da bi se znebili tudi te omejitve. To lahko storimo (vsaj) na dva načina: z uporabo lineranih seznamov (glej poglavje 7.6) in z

dinamičnim povečevanjem velikosti tabele. Slednja rešitev pomeni naslednje: na začetku rezerviramo prostor za dinamično tabelo 50 števil. Uporabniku dovolimo vpisovanje, dokler ne vpiše ničle. Vsakič, ko število vpisanih števil preseže velikost tabele, tabelo povečamo (rezerviramo prostor za novo dinamično tabelo, ki vsebuje 50 prostih mest več kot prejšnja tabela, podatke iz prejšnje tabele prepisemo v novo tabelo, staro tabelo pobrišemo in nadaljujemo z vpisom v novo tabelo).

Izziv 7-I. Popravi program *odklon2.c* z dinamičnim povečevanjem velikosti tabele.



7.4 Prenos parametrov v funkcije

Nekateri programski jeziki omogočajo več načinov prenosa parametrov v funkcije. Najbolj znana sta

- prenos parametrov po vrednosti in
- prenos parametrov po referenci.

Oglejmo si razliko med omenjenima načinoma na naslednjem primeru (gre za program v “nekem” programskem jeziku).

```
bla(x) {
    x=10;
}

main() {
    a=5;
    bla(a);
    izpisi(a);
}
```

Prenos po vrednosti

V funkcijo `bla` se prenese vrednost spremenljivke `a` – spremenljivka `x` dobi vrednost 5. Ker gre za prenos po vrednosti, med spremenljivkama `a` in `x` ne obstaja nobena povezava. Sprememba vrednosti spremenljivke `x` v funkciji NE vpliva na vrednost spremenljivke `a` izven funkcije. Funkcija `izpisi` na koncu programa izpiše število 5.

Prenos po referenci

V funkcijo `bla` se prenese referenca na spremenljivko `a` – spremenljivka `x` se na nek način poveže s spremenljivko `a`. Vsaka sprememba `x` se odraža v spremembi `a`.

Funkcija `izpis` na koncu programa izpiše število 10.

Prenos parametrov po referenci torej omogoča spreminjanje vrednosti spremenljivk, ki “živijo” zunaj klicanih funkcij. To je v mnogih primerih zelo koristna lastnost, z nepravilno uporabo pa lahko tudi nevarna (funkcija lahko spremeni tudi vrednosti tistih spremenljivk, ki naj se ne bi spremenile).

Kako pa je s prenosom parametrov v C? Preden odgovorimo na to vprašanje, si oglejmo naslednji primer. Recimo, da potrebujemo funkcijo `zamenjaj`, ki bo zamenjala vrednost svojih parametrov. Poskusimo napisati program in preverimo, kaj se bo zgodilo.

```
1  zamenjaj(int x, int y) {
2      int t = x;
3      x = y;
4      y = t;
5  }
6
7  main() {
8      int a=10;
9      int b=5;
10
11     printf("a=%d, b=%d\n", a, b);
12     zamenjaj(a,b);
13     printf("a=%d, b=%d\n", a, b);
14 }
```

Zamenjava vrednosti parametrov? (*zamenjaj1.c*)

Ko program poženemo, dobimo naslednji izpis:

Izpis programa `zamenjaj1.c`

```
[user@localhost]# ./zamenjaj1.c
a=10, b=5
a=10, b=5
```

Sprememba vrednosti parametrov v funkciji `zamenjaj` torej NI vplivala na zunanje spremenljivke. To pa je lastnost prenosa po vrednosti. Res, saj se **v programskem jeziku C vsi parametri prenašajo po vrednosti!!!**

7.4.1 Prenos reference po vrednosti

Se torej funkcije `zamenjaj` v C sploh ne da napisati? Odgovor: seveda se da – funkcijo `zamenjaj` lahko napišemo z uporabo kazalcev. Namesto vrednosti spremenljivk `a` in `b` bomo v funkcijo `zamenjaj` prenesli vrednosti naslovov teh spremenljivk (prenos reference po vrednosti). Namesto *zamenjaj 5 in 10* bomo v funkciji `zamenjaj` rekli *zamenjaj vsebini spominskih celic s podanima naslovoma*. Program `zamenjaj1.c` popravimo v treh korakih:

1. spremenimo glavo funkcije `zamenjaj`

```
|| zamenjaj(int x, int y) -> zamenjaj(int *x, int *y)
```

2. popravimo klic funkcije

```
|| zamenjaj(a, b); -> zamenjaj(&x, &y);
```

3. vse pojavitve `x` v funkciji `zamenjaj` zamenjamo z `*x` ter `y` z `*y`.

```
1 | zamenjaj(int *x, int *y) {
2 |     int t = *x;
3 |     *x = *y;
4 |     *y = t;
5 | }
6 |
7 | main() {
8 |     int a=10;
9 |     int b=5;
10 |
11 |     printf("a=%d, b=%d\n", a, b);
12 |     zamenjaj(&a,&b);
13 |     printf("a=%d, b=%d\n", a, b);
14 | }
```

Zamenjava vrednosti parametrov (*zamenjaj2.c*)

Izpis programa `zamenjaj2.c`

```
[user@localhost]# ./zamenjaj2.c
a=10, b=5
a=5, b=10
```

7.4.2 Kazalec na kazalec (`int **x`)

Pravkar opisani postopek (trije koraki za spremembo načina prenosa parametrov) je univerzalen – uporabimo ga na vseh tipih parametrov. Če je parameter tipa `int` (kot v zgornjem primeru), ga v funkciji zamenjamo s kazalcem na `int`. Podobno, če je parameter tipa kazalec na `int` (`int *x`), ga v funkciji zamenjamo s kazalcem na kazalec na `int` (`int **x`).

```
// koda, ki ne dela           // delujoca koda
rezerviraj(int *x) {         rezerviraj(int **x) {
    x = malloc(100);         *x = malloc(100);
}                             }

main() {                     main() {
    int *a;                  int *a;
    rezerviraj(a);          rezerviraj(&a)
}                             }
```

Razlaga leve kode: funkcija `rezerviraj` prejme parameter `x` tipa “kazalec na `int`” in za kazalec `x` rezervira 100 bajtov prostora. Klic funkcije `rezerviraj` v `main` je sicer pravilen (saj je spremenljivka `a` istega tipa kot parameter funkcije `zamenjaj`) toda rezultat klica ni tak, kot bi morda pričakovali (vrednost kazalca `a` se ne spremeni). Problem je podoben kot pri funkciji `zamenjaj`: ker gre za klic po vrednosti, sprememba vrednosti `x` v funkciji ne vpliva na spremembo vrednosti `a` zunaj funkcije. Problem rešimo na enak način, kot smo to storili pri funkciji `zamenjaj` (glej opisane tri korake in primerjaj rezultat z desno kodo).

7.4.3 Tabela kot parameter

Na zgornjih primerih smo videli, da pri klicu funkcij v C velja: vrednost spremenljivke se lahko spremeni le v primeru, če pri klicu podamo njen naslov:

```
// x se ne more spremeniti   // x se lahko spremeni
bla(x);                       bla(&x);
```

Pa ima to pravilo kakšno izjemo? Oglejmo si, kaj izpiše spodnji program.


```

1 | zamenjaj(int a[], int i, int j) {
2 |     int t = a[i];
3 |     a[i] = a[j];
4 |     a[j] = t;
5 | }
6 |
7 | main() {
8 |     int tab[5] = {0,1,2,3,4};
9 |
10 |    // izpisem drugi in cetrtei element
11 |    printf("tab[2]=%d, tab[4]=%d \n", tab[2], tab[4]);
12 |
13 |    // zamenjam vrednost?
14 |    zamenjaj(tab, 2, 4);
15 |
16 |    // izpisem drugi in cetrtei element
17 |    printf("tab[2]=%d, tab[4]=%d \n", tab[2], tab[4]);
18 | }

```

Zamenjava vrednosti elementov v tabeli (*zamenjajTabela.c*)

Ker smo funkcijo `zamenjaj` klicali z `zamenjaj(tab, 2, 4)`, bi (vsaj glede na pravilo, ki smo ga pravkar navedli) pričakovali, da se elementi tabele ne bodo zamenjali. Toda, če program poženemo, dobimo naslednji izpis:

Izpis programa *zamenjajTabela.c*

```

[user@localhost]# ./zamenjajTabela
tab[2]=2, tab[4]=4
tab[2]=4, tab[4]=2

```

Elementi tabele so se zamenjali! Je to izjema zgornjega pravila? Ne, ne gre za izjemo, gre le za skrajšan zapis. V poglavju 7.2 smo namreč zapisali, da je ime tabele sinonim za naslov prvega elementa. Klic `zamenjaj(tab, 2, 4)` je torej ekvivalenten klicu `zamenjaj(&tab[0], 2, 4)`. Če upoštevamo še, da je zapis `int a[]` ekvivalenten zapisu `int *a` ter da je `a[i]` ekvivalentno `*(a+i)`, lahko funkcijo `zamenjaj` ekvivalentno zapišemo tudi takole:

```

zamenjaj(int *a, int i, int j) {
    int t = *(a+i);
    *(a+i) = *(a+j);
    *(a+j) = t;
}

```

Ta zapis pa skupaj s klicem `zamenjaj(&tab[0], 2, 4)` v vseh podrobnostih odgovarja tistemu, čemur smo prej rekli “prenos po vrednosti reference”. Ko torej prenašamo tabele kot parametre v funkcije, se moramo zavedati, da se vsebina tabel pri tem lahko spremeni.

7.4.4 Uporaba funkcije `scanf`

V uvodnem poglavju smo pri reševanju naloge 2-VIII opozorili na “posebnost” pri klicu funkcije `scanf` – pred vsakim parametrom moramo obvezno napisati znak `&`.

```

|| // NAROBE!!!
|| scanf("%d", i);
||
|| // PRAVILNO
|| scanf("%d", &i);

```

Glede na vse zapisano je sedaj razumevanje te “posebnosti” proprosta naloga: ker funkcija `scanf` spremeni vrednost spremenljivk, moramo ob klicu podati njihov naslov. Funkcija `scanf` pa je napisana tako, da prejema parametre po vrednosti reference.

Prav tako nas ne sme več presenetiti dejstvo, da v primeru, ko beremo niz, funkcijo `scanf` lahko kličemo takole:

```

|| scanf("%s", niz);

```

Spremenljivka `niz` je tabela znakov – za tabele pa smo malo prej ugotovili, da se jih pri klicu po vrednosti reference lahko uporablja brez znaka `&`.

7.5 Kazalci in strukture

Naj bo kompleksno struktura, ki vsebuje komponenti `re` in `im` (realni in imaginarni del kompleksnega števila).

```

|| struct kompleksno {
||     // realna in ...
||     double re;
||
||     // ... imaginarna komponenta
||     double im;
|| };

```

in naj bo `w` kompleksno število deklarirano z

```
|| struct kompleksno w;
```

Spomnimo se: do posameznih komponent števila `w` dostopamo s pomočjo pike:

```
|| w.re = 5;  
|| w.im = 1;
```

Deklarirajmo še `z` kot kazalec na strukturo `komplesno` in pokažimo `z` na `w`.

```
|| struct kompleksno *z;  
|| z = &w;
```

Kako sedaj preko kazalca `z` pridemo do komponent števila `w`? Možna sta dva načina: klasičen z uporabo operatorja `*`

```
|| (*z).re = 3;  
|| (*z).im = 2;
```

ali skrajšan z uporabo operatorja `->` (glej program *kazstruct.c*).

```
|| printf("w = %f + %f i", z->re, z->im);
```

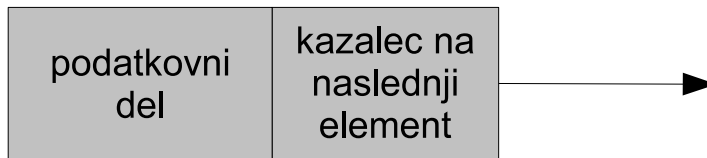
7.6 Linearni seznam

Kazalce lahko uporabljamo tudi za implementacijo dinamičnih podatkovnih struktur, kot je, na primer, linearni seznam. Ta je običajno sestavljen iz posameznih elementov, ki so medseboj povezani v verigo.

Za razliko od tabele, v kateri lahko hranimo le omejeno (vnaprej določeno) število podatkov, linearni seznam lahko poljubno širimo – če želimo vanj dodati nov element, preprosto podaljšamo verigo. Elemente linearnega seznama si lahko predstavljamo kot prikazuje spodnja slika.




Slika 7.2: Linearnega seznama (veriga elementov)



Slika 7.3: Element linearnega seznama

V podatkovnem delu so shranjeni podatki, kazalec na naslednji element pa kaže na naslednji člen verige. Uporabo linearnih seznamov bomo prikazali v rešitvi naslednje naloge.

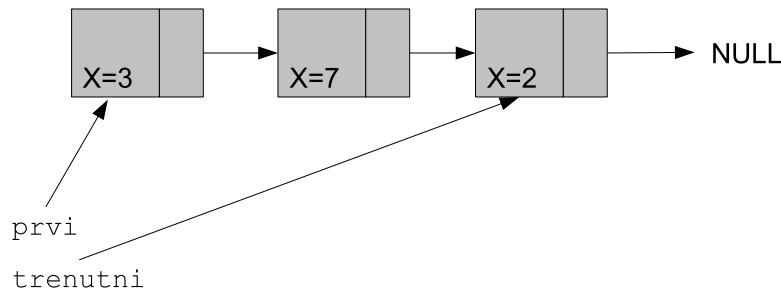
Naloga 7-II. Napiši program za računanje povprečja in standardnega odklona poljubnega števila vpisanih števil. Uporabnik vpisovanje zaključí tako, da vpiše ničlo. 

Za osvežitev spomina: program smo napisali že v več različicah, vsaka od njih je imela kakšno pomanjkljivost. Najprej smo napisali program za računanje povprečja treh števil (naloga 5-II), potem smo ta program razširili tako, da je uporabnik lahko vpisal do 50 števil (naloga 5-III). Omejitev (50 števil) smo poskušali odpraviti v nalogi 7-I, kjer je bilo število vpisanih števil poljubno, toda uporabnik je moral vnaprej povedati, koliko jih bo. Rešitev naloge 7-II bo predstavljala najboljšo možno rešitev tega problema: uporabnik bo lahko vpisal poljubno število podatkov, ko bo želel zaključiti z vpisom, bo vpisal število 0.

Ker potrebujemo popolnoma dinamično strukturo, katere velikost se bo spreminjala po potrebi (brez omejitev), bomo uporabili linearni seznam. Posamezen element tega seznama bo sestavljen iz dveh delov: spremenljivka `x`, ki bo hranila vrednost trenutnega elementa ter spremenljivka `naslednji`, ki bo kazalec na naslednji zapis v seznamu. Slika 7.4 prikazuje tak seznam, ko bodo v njem trije elementi (3, 7 in 2).

Strukturo za element našega seznama deklariramo z

```
struct element {
    float x;
    struct element *naslednji;
};
```



Slika 7.4: Linerani seznam števil, v katerem so števila 3, 7 in 2

Poleg posameznih elementov linearnega seznama potrebujemo tudi kazalec, ki bo označeval njegov začetek. Imenoval se bo **zacetek**, deklariran pa bo z

```
struct element *zacetek;
```

Ko bo seznam prazen, bo **zacetek** kazal na **NULL**, sicer pa na prvi element seznama. Ker je vrstni red shranjevanja vpisanih števil nepomemben, lahko seznam gradimo tako, da nove elemente dodajamo na začetek (in s tem poenostavimo celoten program).

Ko prispe novo število, ki ga moramo vstaviti v seznam, najprej rezerviramo prostor za en element

```
struct element *tmp;

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

nato elementu **tmp** nastavimo vrednosti: v **x** shranimo vpisano število, s kazalcem **naslednji** pa pokazemo na trenutni začetek (ki po vstavljanju ne bo več začetek ampak drugi člen v verigi).

```
tmp->x = s;
tmp->naslednji=zacetek;
```

Ostane samo še sprememba kazalca, ki označuje začetek seznama. Po novem bo ta kazal na pravkar vstavljeni element.

```
zacetek=tmp;
```

Nalogo 7-II rešimo tako, da za osnovo vzamemo program *odklon2.c*, v katerem poleg drobnarij popravimo še:

- namesto tabele `float x[50]` deklariramo kazalec na začetek seznama `struct element *prvi`,
- implementiramo funkcijo `dodaj(float s)`, kot je opisano zgoraj,
- pri računanju povprečja in standardnega odklona namesto `for` zanke uporabimo sprehod čez linearni seznam.

```

1 #include <stdlib.h>
2
3 struct element {
4     float x;
5     struct element *naslednji;
6 };
7
8 struct element *zacetek;
9
10 dodaj(float s) {
11     struct element *tmp;
12
13     // za nov element rezerviramo prostor
14     tmp = (struct element *)
15         malloc(sizeof(struct element));
16
17     // v x shranimo stevilo
18     tmp->x = s;
19
20     // pokazemo na prejsnji zacetek ...
21     tmp->naslednji=zacetek;
22
23     // ... z zacetkom pa na tmp
24     zacetek = tmp;
25 }
26
27
28 main() {
29     float vsota1, vsota2, pov, odk;
30

```

```
31  int i=0;    // stevec za indekse v tabeli x
32  int n;     // stevilo prebranih števil
33  float tx;  // pomožna spremenljivka za branje
34
35  // kazalec za sprehod po seznamu
36  struct element *trenutni;
37
38
39  // s kazalcem za začetek pokazemo na NULL
40  zacetek = NULL;
41
42  // beremo števila, dokler uporabnik ne vpise 0
43  while (1) { // beremo, dokler uporabnik ne vpise 0
44      printf("Vpisi %d. stevilo: ", i+1); scanf("%f",&tx);
45      // ce je prebrano stevilo različno od 0,
46      // ga vpisemo v linearni seznam
47      if (tx != 0) {
48          dodaj(tx);
49          i++;
50      } else // ... sicer končamo z branjem!
51          break;
52  }
53  // zapomnimo si, koliko števil smo prebrali
54  n = i;
55
56  // ce nismo prebrali niti enega števila, končamo!
57  if (n==0)
58      exit(0);
59
60  // racunanje povprečja
61  vsota1=0;
62  trenutni=zacetek;
63  while (trenutni != NULL) {
64      vsota1=vsota1+trenutni->x;
65      trenutni = trenutni -> naslednji;
66  }
67  pov      = vsota1 / n;
68
69  // racunanje standardnega odklona
70  vsota2=0;
71  trenutni=zacetek;
72  while (trenutni != NULL) {
73      vsota2=vsota2+pow((pov - trenutni->x),2);
```

```

74     trenutni = trenutni -> naslednji;
75     }
76     odk = sqrt(vsota2 / n);
77
78     printf("Povprecje: %.2f, Odklon: %.2f", pov, odk);
79 }

```

Rešitev naloge 7-II.: Povprečje in standardni odklon (*odklon4.c*)



Naloga 7-III. Implementiraj funkcije **dodaj**, **brisi** in **izpisi** za delo z linearnim seznamom (podatkovni del elementov seznama naj vsebuje podatke tipa float).

Koraki k rešitvi naloge 7-III:

- elemente seznama deklariramo na enak način kot pri prejšnji nalogi
- funkcija **dodaj**:
 - elemente bomo dodajali na konec seznama,
 - pri dodajanju moramo ločiti dva primera – vstavljanje v prazen ali v že obstoječ seznam,
 - pri vstavljanju v prazen seznam (**zacetek==NULL**) je potrebno popraviti le vrednost kazalca **zacetek** (vrstica 26),
 - pri vstavljanju v že obstoječi seznam se s pomožno spremenljivko **r** “sprehodimo” do konca seznama (**while** zanka v vrstici 29) in tam pripnemo nov element (vrstica 31),
- funkcija **brisi**:
 - ločimo dva primera – element, ki ga brišemo, je prvi element seznama in element, ki ga brišemo, se nahaja na sredini seznama,
 - če brišemo prvi element, moram pravilno popraviti kazalec **zacetek**,
 - če brišem sredinski element, ga moramo najprej poiskati (**while** zanka v 46. vrstici), nato ga pobrišemo tako, da “prevežemo” kazalce sosednjih elementov (vrstice 49–51),
 - v vsakem primeru je potrebno sprostiti spomin (ukaz **free** v 43 in 52 vrstici),
- funkcija **izpisi**: to je najpreprostejša funkcija tega programa; s pomožnim kazalcem **r** se sprehodimo po seznamu od začetka do konca in spotoma izpisujemo posamezen element.


```
1 #include <stdio.h>
2
3 // element seznama
4 struct element {
5     int vrednost;
6     struct element *naslednji;
7 };
8
9 // zacetek sezname
10 struct element *zacetek;
11
12 // dodajanje elementa v seznam
13 void dodaj(int e) {
14     struct element *nov, *r;
15
16     // rezerviram prostor in ...
17     nov = (struct element*)
18         malloc(sizeof(struct element));
19
20     // ... nastavim vrednosti
21     nov->vrednost = e;
22     nov->naslednji = NULL;
23
24     // ali je seznam prazen?
25     if (zacetek == NULL)
26         zacetek = nov;
27     else { // ce ni, vstavim na konec
28         r = zacetek;
29         while ( r->naslednji != NULL )
30             r = r->naslednji;
31         r->naslednji = nov;
32     }
33 }
34
35 // zbrisemo prvi element z vrednostjo e v seznamu
36 void brisi(int e) {
37     struct element *r, *zbrisan;
38
39     // Ali ima ze prvi element vrednost e?
40     if (zacetek->vrednost == e) {
41         zbrisan = zacetek;
42         zacetek = zacetek -> naslednji;
43         free(zbrisan);
```

```

44 } else { // poiscem element in ga zbrisem
45     r=zacetek;
46     while ((r->naslednji != NULL) &&
47             (r->naslednji->vrednost != e))
48         r = r->naslednji;
49     if (r->naslednji != NULL) {
50         zbrisan = r->naslednji;
51         r->naslednji = r->naslednji->naslednji;
52         free(zbrisan);
53     }
54 }
55 }
56
57 // izpisemo element seznama
58 void izpisi() {
59     struct element *r;
60
61     r = zacetek;
62     // sprehodimo se po celotnem seznamu
63     while ( r != NULL ) {
64         printf("%d  ", r->vrednost);
65         r = r->naslednji;
66     }
67     printf("\n");
68 }
69
70
71 main() {
72     // inicializacija seznama; seznam je na zacetku prazen
73     zacetek = NULL;
74
75     dodaj(4); dodaj(7); dodaj(3);
76     izpisi();
77
78     dodaj(6); brisi(7);
79     izpisi();
80 }

```

Rešitev naloge 7-III.: Delo z linearnim seznamom (*linsez.c*)





Izziv 7-II. Dopolni program *linsez.c* tako, da bo omogočal interaktivno uporabo. Uporabniku naj ponudi naslednje možnosti:

Opis

```
Izberi :
0 ... izhod iz programa
1 ... sprazni seznam
2 ... dodajanje novega elementa
3 ... brisanje elementa
4 ... izpis vseh elementov seznama
```

Operacija “sprazni seznam” v *linsez.c* ni implementirana. Čeprav bi program na videz delal pravilno, če bi to operacijo implementirali preprosto z `zacetek=NULL`, taka implementacija ne bi bila pravilna. Zakaj? Kako bi pravilno implementirali to operacijo?

Izziv 7-III. Popravi program *linsez.c* tako, da bosta funkciji `dodaj` in `brisi` dobili še en parameter – kazalec na začetek seznama (pozor: ker se v nekaterih funkcijah vrednost tega parametra spremeni, ga je treba deklarirati kot kazalec na kazalec). Tako popravljen program dopolni še z interaktivnim delom (glej izziv 7-II), ki bo omogočal delo z dvema seznamoma. 

Izziv 7-IV. S pomočjo linearnega seznama implementiraj podatkovno strukturo sklad (funkcije `init`, `push`, `pop`, `je_prazen`). 

Poglavje 8

Delo z datotekami

Ločimo dve vrsti datotek: tekstovne in binarne datoteke. V **tekstovnih datotekah** so zapisani le “berljivi” znaki (črke, številke, presledek, znaki !"#\\$\%&/()= in podobni znaki) ter znak za konec vrste in konec datoteke. Tekstovno datoteko lahko odprem v vsakem urejevalniku (na primer, v Notepadu) ali jo izpišemo na zaslona (ukaz `type` (DOS) oziroma `cat` (Linux)). **Binarne datoteke** vsebujejo podatke v binarnem zapisu. Najmanjša enota zapisa v binarni datoteki je podatek tipa `int`. Binarne datoteke niso “berljive” (urejevalniki besedil običajno ne znajo prikazati posebnih znakov, namesto njih prikazujejo “kvadratke”).

Da bi si lažje predstavljali razliko med tekstovno in binarno datoteko, si oglejmo, kaj se zgodi, če v prvo ali v drugo zapišemo neko številko, na primer, petinšestdeset. Če to število zapišemo v tekstovno datoteko, se bosta v datoteki znašla znaka 6 in 5 (torej niz “65”), če pa jo zapišemo v binarno datoteko, se bo tam nahajalo število 65. Če slednjo datoteko odpremo v urejevalniku besedil, bomo v njej našli znak ‘A’, saj ima ta ASCII kodo 65.

Pri delu z datotekami se srečamo z naslednjimi pojmi:

Ročica (*file handle*). Ročica predstavlja povezavo med datoteko in programsko spremenljivko. Do datotek dostopamo izključno s pomočjo ročic. Ročica je tipa `FILE *`, to je kazalec na tip `FILE`.

Odpiranje (*open*). Z odpiranjem datoteke vzpostavimo povezavo med datoteko in ročico. Funkcija za odpiranje se imenuje `fopen`, uporabljamo pa jo, na primer, takole:

```
FILE *dat ;
dat = fopen (" datoteka .txt " , " r " );
```

Prvi parameter je ime datoteke, drugi pa način, v katerem bomo datoteko uporabljali. Poleg `r`, ki pomeni branje, najpogosteje uporabljamo še `w` (pisanje) ali `a` (dodajanje).

Branje (*read*). Iz datoteke lahko preberemo le tisti podatek, ki je trenutno na bralnem položaju. Če bi po odpiranju datoteke radi prebrali deseti znak, moramo prej prebrati prvega, drugega, ..., osmega in devetega. Skakanje po datoteki (to je premikanje bralnega položaja) je sicer možno, vendar je to draga operacija. Branja datoteke zato ne moremo primerjati s pregledovanjem tabele, kjer lahko "bralno mesto" poljubno izbiramo (RAM). Za branje iz datoteke uporabljamo ukaze `fgetc`, `fgets`, `fscanf`, `fread`,

Pisanje (*write*). Ko datoteko odpremo za pisanje (način *w*), avtomatsko pobrišemo vse podatke, ki so se pred tem nahajali v njej. Podatke v datoteki pišemo le na trenutnem pisalnem položaju (običajno je to na koncu datoteke). Za pisanje v datoteko uporabljamo ukaze `fputc`, `fprintf`, `fwrite`, ...

Dodajanje (*append*). Datoteko lahko odpremo v načinu *a*; stari podatki se bodo ohranili, novi se bodo dodali na konec datoteke. Za dodajanje v datoteko uporabljamo iste ukaze kot za pisanje.

Mesto v datoteki (*file position*). Celoštevilska vrednost, ki označuje mesto v datoteki, kjer se podatki trenutno pišejo oziroma berejo. Vrednost mesta v datoteki nam vrne funkcija `fgetpos`. Po odpiranju datoteke, ali ko kličemo funkcijo `rewind`, se mesto postavi na 0. Po datoteki se lahko premikamo tudi tako, da spreminjamo trenutno bralno/pisalno mesto (funkcija `fseek`).

Konec datoteke (*end of file*). Podatke v datoteki lahko beremo le toliko časa, dokler so ti dosegljivi. Funkcija `feof` pove, ali smo že prišli do konca datoteke (v tem primeru moramo z branjem zaključiti), ali ne (v tem primeru pa lahko nadaljujemo z branjem).

Splakovanje (*flush*). Običajno se za izhodne datoteke uporablja t.i. medpomnilnik (*buffer*), v katerega se zapišejo posamezni podatki, preden se zapišejo v samo datoteko (uporaba medpomnilnika zelo pohitri izvajanje pisanja). Slaba lastnost uporabe medpomnilnika je časovni zamik med samo izvedbo ukaza za zapis podatka in dejanskim zapisom v datoteko. Postopek praznjenja medpomnilnika (takošen zapis podatkov v datoteko) imenujemo splakovanje, dosežemo pa ga s klicem funkcije `fflush`. Medpomnilnik tekstovnih datotek se splakne tudi v primeru, da v datoteko zapíšemo znak za novo vrstico.

Zapiranje (*close*). Ko končamo delo z datotekami, moramo porušiti povezavo med njimi in ročico. To storimo s klicem funkcije `fclose`. Zapiranje datotek je obvezno, saj s tem zagotovimo, da bodo v njih res vsi tisti podatki, ki smo jih vanje zapisali, poleg tega pa z zaprtjem datotek omogočimo, da bodo ostali uporabniki in programi do njih nemoteno dostopali.

Programski jezik C uporabniku pripravi tri standardne ročice: `stdin` (standardni vhod; branje z uporabo te ročice običajno pomeni branje iz tipkovnice), `stdout`

(standardni izhod, običajno zaslon) in `stderr` (standardna napaka, običajno zaslon). Čeprav sta obe ročici, `stdout` in `stderr`, običajno povezani z zaslonom, ima vsaka svoj pomen: na prvo izpisujemo rezultate izvajanja, na drugo pa le napake, do katerih je prišlo do izvajanja. Uporabnik programov, ki pravilno uporabljajo `stdout` in `stderr`, lahko s preusmeritvami prestreže le tisti del izpisa, ki ga zanima (rezultate izvajanja, napake ali oboje).

```
1 | #include <stdio.h>
2 | main() {
3 |     fprintf(stdout, "To je obicajno sporocilo \n");
4 |     fprintf(stderr, "To je sporocilo o napaki \n");
5 | }
```

Opis (*outerr.c*)

Zgornji primer izpiše dve sporočili - enega na `stdout`, drugega na `stderr`. Oglejmo si nekaj možnih načinov uporabe tega programa.

Načini uporabe programa `outerr.c`

```
[user@localhost]# ./outerr
To je obicajno sporocilo
To je sporocilo o napaki

[user@localhost]# ./outerr > sporocilo.txt
To je sporocilo o napaki

[user@localhost]# ./outerr 2> napaka.txt
To je obicajno sporocilo
```

Ko poženemo program `outerr`, izpiše obe sporočili (to je najbolj običajen način uporabe programov; uporabnik se ponavadi sploh ne zaveda, da programi lahko izpisujejo dve različni vrsti sporočil). V druge primeru smo ves tekst, ki je bil izpisan na `stdout` preusmeri v datoteko `sporocilo.txt` (če po končanju programa pogledamo vsebino te datoteke, bomo v njej našli besedilo "To je obicajno sporocilo"), na zaslon pa se izpiše samo tekst, ki je bil izpisan na `stderr`. Podobno lahko z `2>` v datoteko preusmerimo vse, kar je bilo izpisano na `stderr`, na zaslon pa se nam izpise samo besedilo, izpisano na `stdout`.

Uporabo posameznih ukazov za delo z datotekami si bomo ogledali na nekaj primerih. Najprej se bomo osredotočili na tekstovne datoteke, nato na binarne.


8.1 Branje tekstovnih datotek po znakih

Naloga 8-I. Napiši program, ki prebere tekstovno datoteko, in jo izpiše na zaslon.  Datoteko beri po znakih (funkcija `getc`).

```
1 #include <stdio.h>
2
3 main(int argc, char *args[]) {
4     if (argc != 2) exit(1);
5
6     FILE *d;
7     if ((d = fopen(args[1], "r")) == NULL)
8         exit(2);
9
10    int c;
11    while ((c = getc(d)) != EOF)
12        putchar(c);
13
14    fclose(d);
15 }
```

Branje tekstovne datoteke po znakih (*getc.c*)

8.2 Branje tekstovnih datotek po besedah

 **Naloga 8-II.** Napiši program, ki prebere datoteko, in izpiše najdaljšo besedo, ki se v njej pojavi. Za branje uporabi funkcijo `fscanf` in format `%s`. Predpostaviš lahko, da nobena beseda ni daljša od 100 znakov.

```
1 #include <stdio.h>
2 #include <string.h>
3
4 main(int argc, char *args[]) {
5     if (argc != 2) exit(1);
6
7     FILE *d;
8     if ((d = fopen(args[1], "r")) == NULL)
9         exit(2);
10
11    char trenutna[100];
```




```

12 | char najdaljsa[100]="";
13 | while (!feof(d)) {
14 |     fscanf(d, "%s", trenutna);
15 |     if (strlen(najdaljsa) < strlen(trenutna))
16 |         strcpy(najdaljsa, trenutna);
17 | }
18 | fclose(d);
19 |
20 | printf("Najdaljsa beseda datoteke '%s' je '%s'",
21 |        args[1], najdaljsa);
22 | }

```

Izpis najdaljše besede dane datoteke (*fscanf.c*)

8.3 Branje podatkov v tekstovnih datotekah

Naloga 8-III. V datoteki "ocene.txt" so zapisani podatki o ocenah pri predmetu  OP2. Vsaka vrstica datoteke pripada enemu študentu. Posamezna vrstica je oblike

STUD_ID OCENA_VAJ OCENA_IZPITA

Napiši program, ki prebere datoteko "ocene.txt", in izračuna ter izpiše povprečno oceno vaj in predavanj. Za branje uporabi funkcijo `fscanf` in format `%d %d %d`.

```

1 | #include <stdio.h>
2 | #include <string.h>
3 |
4 | main(int argc, char *args[]) {
5 |     FILE *d;
6 |     if ((d = fopen("ocene.txt", "r")) == NULL)
7 |         exit(2);
8 |
9 |     // prebrane vrednosti trenutnega studenta
10 |    int tov, toi, tsid;
11 |
12 |    // vsota vseh ocen in stevilo studentov
13 |    int ov = 0, oi = 0, n = 0;
14 |
15 |    int stp; // koliko podatkov je v trenutni vrstici
16 |
17 |    while (!feof(d)) {
18 |        stp = fscanf(d, "%d %d %d\n", &tsid, &tov, &toi);

```

```

19     if (stp !=3) {
20         printf("Napacen format podatkov v vrstici %d", n+1);
21         exit(1);
22     } else {
23         n++; ov += tov; oi += toi;
24     }
25 }
26 fclose(d);
27
28 if (n==0)
29     printf("Datoteka %s je prazna", args[1]);
30 else
31     printf("Vaje: %.2f, Izpit: %.2f\n", 1.0*ov/n, 1.0*oi/n);
32 }

```

Branje podatkov z ukazom `fscanf` (*fscanf1.c*)



Naloga 8-IV. Preberi datoteko "ocene.txt" (glej opis pri nalogi 8-III) in izpiši podatke o študentih, urejene po povprečni oceni. Za hranjenje podatkov uporabi linearni seznam s funkcijo `dodaj`, ki podatek doda na pravo mesto v seznamu (glede na povprečno oceno).

```

1 #include <stdio.h>
2 #include <string.h>
3 #include <memory.h>
4
5 typedef struct student {
6     int SID;
7     int ocena_vaj;
8     int ocena_izpita;
9     float povprecje;
10
11     struct student *nasl;
12 } stip;
13
14 stip *zacetek = NULL;
15
16 void dodaj(int id, int ov, int oi) {
17     stip *nov, *tr;
18     nov = (stip *) malloc(sizeof(stip));
19     nov->SID = id;
20     nov->ocena_izpita = oi;
21     nov->ocena_vaj = ov;

```

```
22     nov->povprecje    = (ov + oi) /2;
23
24     if (zacetek == NULL) {
25         zacetek = nov;
26         nov ->nasl = NULL;
27     } else {
28         if (zacetek->povprecje < nov->povprecje) {
29             nov ->nasl = zacetek;
30             zacetek = nov;
31         } else {
32             tr = zacetek;
33             while ((tr->nasl != NULL) &&
34                 (tr->nasl->povprecje > nov->povprecje))
35                 tr = tr ->nasl;
36             nov -> nasl = tr ->nasl;
37             tr -> nasl = nov;
38         }
39     }
40 }
41
42
43 void izpisi() {
44     stip *tr;
45     tr = zacetek;
46     while (tr != NULL) {
47         printf("%d: %d, %d, %.2f\n", tr->SID,
48             tr -> ocena_vaj, tr->ocena_izpita, tr->povprecje);
49         tr = tr -> nasl;
50     }
51 }
52
53 void main(int argc, char *args []) {
54
55     FILE *d;
56     if ((d = fopen("ocene.txt", "r")) == NULL)
57         exit(2);
58
59     int tsid, tov, toi;
60     int n=0;
61     int stp; // koliko podatkov je v trenutni vrstici
62
63     while (!feof(d)) {
64         stp = fscanf(d, "%d %d %d\n", &tsid, &tov, &toi);
```

```

65     if (stp !=3) {
66         printf("Napacen format podatkov v vrstici %d", n+1);
67         exit(1);
68     } else {
69         dodaj(tsid ,tov ,toi );
70     }
71 }
72 fclose(d);
73
74 izpisi ();
75 }

```

Branje podatkov v linearni seznam (*fscanf2.c*)

8.4 Branje tekstovnih datotek po vrsticah



Naloga 8-V. Preberi dano tekstovno datoteko in izpiši dolžino najdaljše vrstice v njej. Uporabi funkcijo `fgets`.

```


1 #include <stdio.h>
2
3 #define N 10
4
5 main(int argc , char *args []) {
6     if (argc != 2) exit(1);
7
8     FILE *d;
9     if ((d = fopen(args[1] , "r")) == NULL)
10         exit(2);
11
12     char vrstica[N]; // prebrana vrstica
13
14     // dolzina najdaljse prebrane vrstice
15     int najdaljsa = 0;
16
17     // dolzina trenutne vrstice
18     int trenutna = 0;
19
20     while (!feof(d)) {
21         fgets(vrstica ,N,d);
22         trenutna += strlen(vrstica);

```

```
23
24     // ali smo vrstico prebrali do konca?
25     if (vrstica[strlen(vrstica) - 1] == '\\n') {
26         if (najdaljsa < trenutna) najdaljsa = trenutna;
27         trenutna = 0;
28     }
29 }
30 fclose(d);
31
32 printf("Dolzina najdaljse vrstice je %d\\n", najdaljsa - 1);
33 }
```

Izpis dolžine najdaljše vrstice v datoteki (*fgets.c*)

8.5 Kopiranje tekstovne datoteke

Naloga 8-VI. Napiši program, ki vhodno datoteko prepíše v izhodno datoteko, in  pri tem vse male črke spremeni v velike.

```
1 #include<stdio.h>
2
3 main(int argc, char *args[]) {
4     FILE *vhod, *izhod;
5     int c;
6
7     vhod = fopen(args[1], "r");
8     izhod = fopen(args[2], "w");
9     if ((vhod != NULL) && (izhod != NULL)) {
10
11         while (!feof(vhod)) {
12             c = getc(vhod);
13             c = toupper(c);
14             putc(c, izhod);
15         }
16
17         fclose(izhod);
18         fclose(vhod);
19
20         printf("OK \\n");
21     } else
22         printf("Napaka!\\n");
23 }
```

Prepis (kopiranje) datoteke znak po znaku (*getputc.c*)

8.6 Pisanje podatkov v tekstovno datoteko



Naloga 8-VII. Napiši program, ki 10 naključnih števil zapiše v tekstovno datoteko (funkcija `fprintf`).

```
1 #include<stdio.h>
2
3 main() {
4     FILE *d;
5     d = fopen("rand.txt", "w");
6     if (d == NULL) exit(1);
7
8     int i;
9     for (i=0; i<10; i++)
10         fprintf(d, "%d\n", rand() % 100);
11
12     fclose(d);
13 }
```

Zapis naključnih števil v datoteko (*fprintf.c*)

8.7 Dodajanje v tekstovno datoteko



Naloga 8-VIII. Napiši program, ki na konec tekstovne datoteke doda besedilo "—konec—".

```
1 #include<stdio.h>
2
3 main() {
4     FILE *d;
5     if ((d = fopen("rand.txt", "a")) == NULL)
6         exit(1);
7
8     fprintf(d, "— konec —");
9
10    fclose(d);
11 }
```

Dodajanje besedila na konec datoteke (*append.c*)

8.8 Kopiranje binarne datoteke


Naloga 8-IX. Napiši program, ki naredi kopijo vhodne binarne datoteke. 

Rešitev te naloge je zelo podobna rešitvi naloge 8-VI - edina razlika je v tem, da program *getputc.c* kopira tekstovno datoteko, program *getputbin.c* pa binarno. Če bi program *getputc.c* uporabili za kopiranje binarnih datotek, kopija ne bi bila nujno enaka originalu. Namreč, v tekstovnih datotekah znak z ASCII kodo 26 označuje konec datoteke, v binarnih pa ta znak nima posebnega pomena. Kopija binarne datoteke, ki bi jo naredili s programom *getputc.c*, bi zato vsebovala le podatke do prvega znaka s kodo 26.

Če želimo delati z binarnimi datotekami, moramo to povedati ob klicu funkcije `fopen`; namesto `vhod = fopen(args[1], "r")` moramo reči `vhod = fopen(args[1], "rb")`, podobno za `izhod`.

<code>// getputc.c</code>	<code>// getputbin.c</code>
<code>vhod=fopen (args [1] , " r");</code>	<code>vhod=fopen (args [1] , " rb");</code>
<code>izhod=fopen (args [2] , " w");</code>	<code>izhod=fopen (args [2] , " wb");</code>

8.9 Branje in pisanje binarne datoteke po blokih

Naloga 8-X. Napiši program, ki prebere vsebino vhodne datoteke in jo zakodira. Za kodiranje uporabi preprosto XOR metodo (*i*-ti znak zakodiraš tako, da izvedeš operacijo XOR z *i*-tim znakom ključa). 

Datoteko bomo brali z ukazom `fread` (branje bloka podatkov), pisali pa z ukazom `fwrite` (pisanje bloka podatkov). Nad blokom bomo izvedli operacijo kodiranja. Ker je ključ obiajno krajši od bloka, ki ga kodiramo, uporabimo večkratno ponovitev ključa, kot prikazuje spodnja slika.

Uporaba ključa `test` (4 črke) na nizu s 24 znaki

<pre>niz: to je niz , ki ga kodiram kljuc: testtesttesttesttesttest</pre>

Ker rezultat operacije XOR ni nujno "berljiv" znak (na primer, rezultat operacije 't' xor 't' (prvi črki v zgornjem primeru) je znak z ASCII kodo 0), bo izhodna datoteka binarna. Ker je algoritem za kodiranje enak algoritmu za dekodiranje, je smiselno, da tudi vhodno datoteko odpremo kot binarno – na ta način bomo lahko isti program uporabljali za kodiranje in dekodiranje datotek.

```
1 #include<stdio.h>
2
3 main(int argc, char *args[]) {
4     FILE *vhod, *izhod;
5     int c, i;
6
7     char kljuc[100];
8     char podatki[512];
9     int koliko;
10
11     vhod = fopen(args[1], "rb");
12     izhod = fopen(args[2], "wb");
13
14     printf("Vpisi kljuc do 100 zankov: ");
15     scanf("%s", kljuc);
16
17     while (!feof(vhod)) {
18         koliko = fread(podatki, 1, 512, vhod);
19
20         for(i=0; i<koliko; i++)
21             podatki[i] ^= kljuc[i % strlen(kljuc)];
22
23         fwrite(podatki, 1, koliko, izhod);
24     }
25
26     fclose(vhod);
27     fclose(izhod);
28 }
```

Rešitev naloge 8-X.: Kodiranje datoteke s preprosto metodo XOR (*codefile.c*)

Poglavje 9

Rešitve nekaterih izzivov

9.1 Izziv 2-I

najprej izpišemo “glavo” (vrstice 7–10) ter črto pod njo (vrstici 13 in 14),

- zanka v 16. vrstici je zanka “po vrsticah” – v vsaki iteraciji izpišemo eno vrstico,
- druga (notranja) zanka (vrstica 18) je zanka za izpis posamezne vrstice (10 števil v vsaki vrsti)

```
      |  1  2  3  4  5  6  7  8  9 10
-----
```

```
1 #include <stdio.h>
2
3 main() {
4     int i, j;
5
6     // prva vrstica (glava izpisa)
7     printf("      | ");
8     for (i=1; i<=10; i++)
9         printf("%4d", i);
10    printf("\n");
11
12    // crta pod prvo vrstico
13    for (i=1; i<=47; i++) printf("-");
14    printf("\n");
15
16    for (i=1; i<=10; i++) {
17        printf("%4d | ", i);
```

```

18     for (j=1; j<=10; j++)
19         printf("%4d", i*j);
20     printf("\n");
21 }
22 }

```

Rešitev izziva 2-I: Izpis poštevanka (*postevanka.c*)

9.2 Izziv 2-II

- spremeniti moramo deklaracijo funkcije `main` (vrstica 3)
- popraviti je potrebno še zgornjo mejo za števec v `for` zanki: namesto 32 vpišimo vrednost prvega argumenta (`atoi(args[1])`),
- dobro napisan program bi moral preveriti, ali je uporabnik podal vsaj en argument; bralec naj to preverjanje doda za vajo (pomaga si lahko z rešitvijo naloge 2-V)

```

1 #include <stdio.h>
2
3 main(int argv, char **args) {
4     int i;
5     printf("Desetisko | Osmisko | Sestnajstisko\n");
6     for (i=1; i<=atoi(args[1]); i++)
7         printf("%7d | %5o | %11X\n", i, i, i);
8 }

```

Tabele za pretvorbo med številskimi sistemi (*izpis10-8-16-argv.c*)

9.3 Izziv 2-III

programu *fc.c* dodamo preverjanje podanih argumentov (vrstice 7–11)

- dodamo se pretvorbo argumentov v tip `int` (vrstice 17–19)
- spremenimo glavo `for` zanke (vrstica 22)

```

1 #include <stdio.h>
2
3 main(int argc, char *argv[]) {
4     int f;
5     double c;

```

```

6
7  if (argc!=4) {
8      printf("Uporaba programa: %s sMeja zMeja korak",
9          argv[0]);
10     exit(0);
11 }
12
13 // izpis glave tabele
14 printf("%6c | %6c\n", 'F', 'C');
15 printf("-----\n");
16
17 int sMeja=atoi(argv[1]);
18 int zMeja=atoi(argv[2]);
19 int korak=atoi(argv[3]);
20
21 // glavna zanka programa
22 for(f=sMeja; f<=zMeja; f=f+korak) {
23     c=(f - 32)/1.8;
24
25     // izpis posamezne vrstice tabele
26     printf("%6d | %6.2f\n", f, c);
27 }
28 }

```

Izpis tabele F/C z argumenti (*fc1.c*)

9.4 Izziv 2-IV

- glavna sprememba programa *fc.c* je v vrstici za izpis:

```

|| printf("%6d | %6.*f\n", f, dMest, c);

```

- *dMest* je spremenljivka tipa `int` prebrana iz argumentov, znak `*` v `%6.*f` pa pove, da je število decimalk določeno dinamično preko parametra,
- celotna rešitev naloge je v datoteki *fc2.c*

9.5 Izziv 5-I

- v datoteki *urejanje.c* napišemo dve funkciji za delo z datotekami: *urejanje*, ki tabelo uredi in *izpis*, ki tabelo izpiše na zaslon

```

1 // DELO TABELAMI
2
3
4 // funkcija uredi stevila s[0], ..., s[n-1]
5 // po velikosti z uporabo bubble-sort metode
6 void uredi(int n, int s[]) {
7     int i, j, t;
8     for (i=0; i<n; i++)
9         for (j=0; j<(n-i-1); j++)
10            if (s[j] > s[j+1]) {
11                t = s[j];
12                s[j] = s[j+1];
13                s[j+1] = t;
14            }
15 }
16
17 // funkcija izpise tabelo na zaslon
18 void izpisiTabelo(int n, int s[]) {
19     int i;
20     for (i=0; i<n; i++)
21         printf("%d ", s[i]);
22     printf("\n");
23 }

```

Orodja za delo z datotekiam (*urejanje.c*)

- datoteko *loto2.c* skopiramo v *loto3.c* in popravimo: namesto izpisovanja posameznih števil, kličemo funkciji za urejanje in izpis

```

    uredi(7, tabela);
    izpisiTabelo(7, tabela);

```

- celotni program prevedemo z `gcc urejanje.c loto3.c`

9.6 Izziv 5-VII

```
1 typedef struct kompleksno {
2     double re;
3     double im;
4 } cplx;
5
6 cplx vsota(cplx x, cplx y) {
7
8     cplx rezultat;
9
10    rezultat.re = x.re + y.re;
11    rezultat.im = x.im + y.im;
12    return rezultat;
13 }
14
15 cplx produkt(cplx x, cplx y) {
16
17    cplx rezultat;
18
19    rezultat.re = x.re * y.re - x.im * y.im;
20    rezultat.im = x.re * y.im + x.im * y.re;
21
22    return rezultat;
23 }
24
25 main() {
26
27    cplx w, z, p, v;
28
29    w.re = 3; w.im=2;
30    z.re = 5; z.im=7;
31
32    v = vsota(w,z);
33    p = produkt(w,z);
34
35    printf("(%.f + %.f i) + (%.f + %.f i) = %.f + %.f i\n",
36           w.re, w.im, z.re, z.im, v.re, v.im);
37
38    printf("(%.f + %.f i) * (%.f + %.f i) = %.f + %.f i\n",
39           w.re, w.im, z.re, z.im, p.re, p.im);
```

40 ||
41 || }

Vsota in produkt kompleksnih števil (*kompleksno.c*)