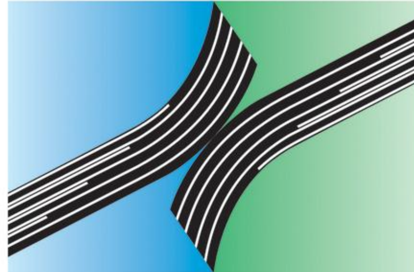
Univerza v *Ljubljani*
Fakulteta za strojništvo



Rok Vrabič, Lovro Kuščer

Osnove programskega jezika C

Gradivo za vaje pri predmetih Mehatronski sistemi in Diskretni krmilni sistemi

Ljubljana, 2014

Kazalo

0 Uvod 1
1 Prvi C program 2
1.1 Zgodovina programskih jezikov 2
1.2 Programski jezik C 2
1.3 Prvi program 3
1.4 Prevajanje 3
1.5 Povzetek 6
1.6 Naloge 7
Povezave 8
Literatura 8
2 Spremenljivke in funkcije 9
2.1 Spremenljivke in podatkovni tipi 9
2.1.1 Modfikatorji, kvalifikatorji in druge ključne besede, vezane na spremenljivke 13
2.1.2 Branje in izpisovanje spremenljivk s standardnega inputa/outputa (`printf`, `scanf`) 15
2.1.3 Aritmetične operacije 16
2.2 Funkcije 16
2.3 Povzetek 19
2.4 Naloge 20
2.5 Predloga datoteke za programe 20
3 Krmilne strukture in operatorji 22
3.1 Krmilne strukture 22
3.1.1 if 22
3.1.2 switch/case 24
3.1.3 for 26
3.1.4 while, do/while 27
3.1.5 Izhod iz zank 28
3.1.6 Skok v naslednjo iteracijo zanke 28
3.2 Relacijski in logični operatorji 29
3.3 Povzetek 31
3.4 Naloge 32
4 Bitni operatorji in algoritmi 33
4.1 Bitni operatorji 33
4.2 Algoritmi 34
4.3 Povzetek 39
4.4 Naloge 40
5 Algoritmi (nadaljevanje) 41
5.1 Primeri 41
5.2 Povzetek 48
5.3 Naloge 49

6 Polja in kazalci 50
6.1 Podatkovna polja 50
6.2 Kazalci 51
6.3 Povzetek 57
6.4 Naloge 58
7 Znakovna polja 59
7.1 Delo z znakovnimi polji 59
7.2 Delo z datotekami 61
7.3 Povzetek 69
7.4 Naloge 70
8 Strukture, spomin in predprocesor 72
8.1 Strukture 72
8.1.1 Uporaba struktur s kazalci 73
8.2 Delo s spominom 73
8.2.1 malloc 74
8.2.2 calloc 74
8.2.3 realloc 75
8.2.4 free 75
8.2.5 Primer 75
8.3 Predprocesor 76
8.4 Povzetek 78
8.5 Naloge 80
9 Zaključek oz. kako naprej? 81
9.1 Izpuščene tematike 81
9.2 Od C proti OOP 81
9.3 C na mikrokrmilnikih 81
9.4 Zaključek 81

0 Uvod

Gradivo je namenjeno uporabi na vajah pri predmetih Mehatronski sistemi, Diskretni krmilni sistemi in na Poletnih šolah mehatronike, izvajanih na Fakulteti za strojništvo Univerze v Ljubljani. V trenutni verziji je gradivo razdeljeno na 8 sklopov, pri čemer je posamezen sklop obravnavan kot celota.

Gradivo obsega osnove programskega jezika C, ki je še danes eden izmed najpomembnejših programskih jezikov, predvsem zato, ker omogoča nizkonivojsko programiranje in je posledično stanje tehnike na področju programiranja mikrokrmilnikov.

Obravnavane tematike gradijo ena na drugi, od prvega programa, do kompleksnih konceptov, kot so kazalci in strukture. Na ta način je omogočen postopen študij jezika. Vsako poglavje podaja naloge, ki služijo utrjevanju snovi in vzpodbujajo samostojen študij.

1 Prvi C program

Vsebina:

- zgodovina programskih jezikov,
- prvi program,
- prevajanje,
- demonstacija: notepad + gcc iz komandne vrstice,
- demonstracija: razvojno okolje (IDE) Code::Blocks,
- povzetek,
- naloge.

1.1 Zgodovina programskih jezikov

Prvi programski jeziki so nastali pred modernimi računalniki. Že leta 1801 so bile na statvah uporabljene luknjaste kartice, ki so vsebovale informacije za avtomatsko tkanje različnih zapletenih vzorcev. Moderni, električni računalniki so se pojavili šele v 40. letih 20. stoletja. Zaradi zelo omejenih procesnih in spominskih zmogljivosti je programiranje potekalo v zbirnem jeziku (angl. assembly language), kar se je izkazalo za naporno delo z veliko verjetnostjo napak. Zato so se kmalu pojavili programski jeziki, kot na primer Plankalkül ter ENIAC coding system, ki izvirata iz obdobja med leti 1943 in 1948.

V 50. letih so nastali prvi moderni programski jeziki, in sicer FORTRAN (1955), LISP (1958) in COBOL (1959). Iz obdobja med letoma 1967 in 1978 izvira večina programskih paradigem, ki se uporabljajo še danes, med katerimi so tudi Simula, C (1969 - 1973), Smalltalk, Prolog in ML. V 80. letih so je razvoj nadaljeval v smeri izpopolnjevanja in združevanja obstoječih paradigim, pri čemer so nastali programski jeziki C++, Objective-C, Ada, Perl, Mathematica in številni drugi.

Naslednje desetletje je zaznamovala hitra rast svetovnega spleta in s tem priložnosti za razvoj novih programskih jezikov. V tem obdobju so med drugimi nastali Python, Visual Basic, Ruby, Java, Delphi, JavaScript in C#. Evolucija programskih jezikov se danes nadaljuje v mnogih smereh med katere sodijo sočasno in distribuirano programiranje, dodajanje varnosti, preverjanje zanesljivosti, integracija s podatkovnimi bazami, vzporedno procesiranje na grafičnih karticah in procesorskih poljih.

1.2 Programski jezik C

C je srednjenivojski programski jezik, ki ga je razvil Dennis Ritchie (Bell Telephone Laboratories) med letoma 1969 in 1973. Je eden izmed najbolj razširjenih programskih jezikov in se uporablja na številnih računalniških arhitekturah. Zasnovan je bil za izdelavo sistemskega programja, vendar se množično uporablja tudi za razvoj uporabniških programov. Programski jezik C je standardiziran v večih standardih:

- 1983 – 1988 ANSI standardizacija,
- 1990: ANSI/ISO 9899:1990 (C90),
- 1999: ISO/IEC 9899:1999 (C99),
- 2011: ISO/IEC 9899:2011 (C11).

1.3 Prvi program

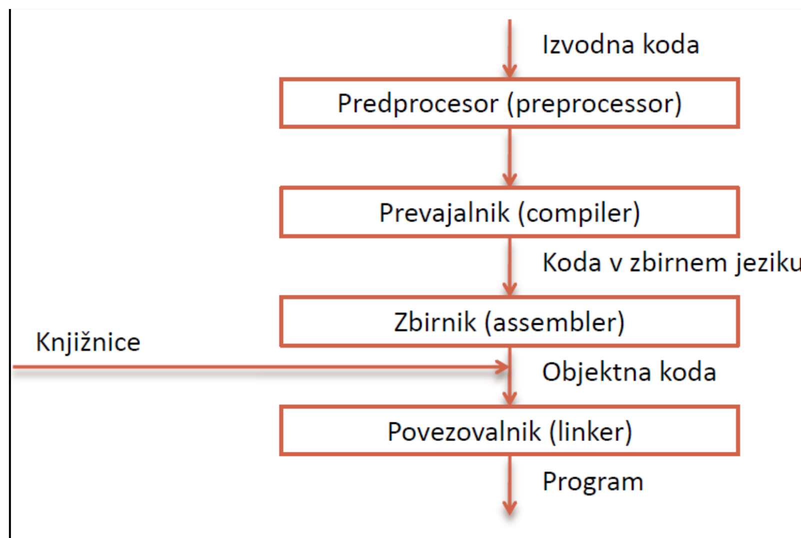
```
#include <stdio.h>

int main(void)
{
      printf("Mehatronski sistemi!");
      return 0;
}
```

- `#include` predprocesorski ukaz za vključitev knjižnice `stdio.h` v prvi program.
- `int` podatkovni tip za cela števila.
- `main()` glavna funkcija, pri kateri se program prične izvajati.
- `void` prazen podatkovni tip (pomeni, da `main` ničesar ne sprejema)
- `printf()` funkcija za formatiran izpis v konzolo (iz knjižnice `stdio.h`).
- `return` zaključek funkcije in vračanje vrednosti. Funkcija main po dogovoru vrne kodo napake programa. Običajno je, da programi, ki pravilno zaključijo svoje delovanje vrnejo 0. V DOS oknu izhodno kodo zadnjega programa izpišemo z ukazom `ECHO.%ERRORLEVEL%`
- `{...}` vse, kar je med zavitima oklepajema imenujemo *blok kode*

1.4 Prevajanje

V prejšnjem poglavju prikazan računalniški program je napisan v programskem jeziku, ki je razumljiv programerju, medtem ko računalnik za izvrševanje nalog potrebuje navodila v strojnem jeziku. Za prevajanje programa iz ljudem razumljivega programskega jezika v računalniku razumljiv strojni jezik uporabimo prevajalnik (angl. compiler). Prevajalnik je računalniški program (ali nabor več programov), ki pretvori izvorno kodo v programskem jeziku v nek drug jezik (običajno strojni jezik). Posamezni koraki prevajanja programa so prikazani na sliki 1-1.

- predprocesor poskrbi za gradnjo vmesne datoteke, ki jo zgradi na podlagi izvorne kode in v izvorni kodi zapisanih predprocesorskih ukazov. Na primer, namesto `#include <stdio.h>` se v vmesno datoteko zapiše vsebina datoteke stdio.h
- prevajalnik prevede vmesno datoteko v kodo v zbirnem jeziku (običajno pri mikrokrmilnikih), ali pa neposredno v objektno kodo. Objektna koda je po obliki enaka končnemu programu, le da vanjo še ni vključena objektna koda iz knjižnic. S predprocesorskimi ukazi so namreč vključene le definicije uporabljenih funkcij, ne pa tudi njihova objektna koda
- za gradnjo končnega programa poskrbi povezovalnik, ki združi prevedeno objektno kodo s kodo uporabljenih funkcij iz vključenih knjižnic



Slika 1-1: Princip delovanja prevajalnika

Za demonstracijo bomo uporabili odprtokodni C prevajalnik iz zbirke GCC (GNU Compiler Collection) in poljuben urejevalnik besedila. V urejevalniku napišemo izvorno kodo prvega programa in datoteko shranimo pod imenom main.c. Nato zaženemo komandno vrstico (angl. command prompt) in poiščemo mapo v katero smo shranili datoteko main.c. V komandno vrstico vpišemo naslednji ukaz:

```
gcc main.c -o main.exe
```

S tem smo prevajalniku GCC sporočili naj prevede izvorno datoteko main.c v program main.exe, ki ga lahko v nadaljevanju zaženemo s klicem iz komandne vrstice:

```
main.exe
```

Pogosto se za avtomatiziranje postopka prevajanja uporabljajo t.i. *make* datoteke.

Poleg omenjenega načina se pri razvoju programov pogosto uporabljajo integrirana razvojna okolja (angl. IDE - Integrated Development Environment), ki predstavljajo skupek programskih orodij namenjenih programiranju. Ti programski paketi običajno vsebujejo:

- urejevalnik besedil,
- prevajalnik,
- orodja za avomatizirano izgradnjo programov,
- iskalnik napak (angl. debugger).

V okviru vaj bomo uporabljali odprtokodno integrirano razvojno okolje Code::Blocks v kombinaciji s prevajalnikom GCC. Razvojno okolje je prosto dostopno na naslovu: http://www.codeblocks.org, kjer za operacijski sistem Windows najdemo istalacijsko datoteko (codeblocks-[verzija]mingw-setup.exe), ki vsebuje tudi GCC prevajalnik.

Pred ustvarjanjem prvega projekta je potrebno razvojno okolje ustrezno nastaviti. V našem primeru to pomeni zgolj izbiro ustreznega prevajalnika, ki bo kodo, napisano v programskem jeziku C, prevedel v strojno kodo za procesor osebnega računalnika.

Po zagonu razvojnega okolje Code::Blocks izvedemo izbiro prevajalnika z naslednjimi koraki:

- Settings -> Compiler and debugger...,
- Selected Compiler nastavimo na GNU GCC Compiler,
- Kliknemo na Set as default in nato OK.

Po nastavitvi ustvarimo nov projekt s klikom na File -> New -> Project..., kjer izberemo Console Application. V nadaljevanju izberemo programski jezik C in ime ter lokacijo projektne mape na disku. Razvojno okolje nam pri ustvarjanju projekta samodejno ustvari datoteko main.c, ki že vsebuje program za izpis pozdravnega besedila na zaslonu. Program prevedemo in zgradimo s klikom na Build -> Build in ga nato zaženemo s klikom na Build -> Run.

1.5 Povzetek

Predprocesorski ukazi:

- `#include <>`

Funkcije (knjižnice):

- `printf (iz stdio.h)`

Ključne besede:

- `int, void, main, return`
- (`int` in `void` sta podatovna tipa)

Razno:

- prevajanje programa iz komandne vrstice
- prevajanje z uporabo IDE

1.6 Naloge

Naloga 1

Z uporabo okolja Code::Blocks napišite program, ki bo na zaslonu izpisal naslednjo tabelo:

```
jezik       mesto       delez       letna rast
Java        1           17.05%      -1.43%
C           2           16.52%      +1.54%
C#          3           8.65%       +1.84%
C++         4           7.85%       -0.33%
```

Potrebne informacije za formatiran izpis na zaslonu poiščite na internetu.

Naloga 2

Napišite program, ki bo na zaslonu izpisal:

```
Pozor!
```

in pri tem proizvedel pisk. Uporabite zgolj funkcijo `printf`.

Naloga 3

V CodeBlocks napišite naslednji program (brez include, return, …!):

```
int main()
{
    printf("Mehatronika");
}
```

Se program prevede? Zakaj? Poskusite razvozlati, zakaj je izhodna koda tega programa, ki nima eksplicitnega klica `return`, 11, kadar ga prevajamo z gcc.

Dodatne naloge

Poglejte si, kako uporabiti *make* datoteko. Ugotovite, kaj morate naložiti na računalnik (namig: iščite npr. "win make"), kakšna je oblika make datoteke in kako avtomatizirati prevajanje programa iz naloge 1.

Preberite, kakšna je razlika med prevajanjem v C-ju in prevajanjem v Javi. Kaj pomeni izraz "just-in-time compiler"?

Poiščite in preizkusite še en IDE za razvoj v C ali C++.

Poglejte, kaj pomeni naslednja oblika funkcije main (tudi ta oblika je veljavna po standardu):

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

Poizkusite poiskati C99 standard. Kaj vse specificira?

Povezave

[1] Programski jezik C na Wikipediji:
http://en.wikipedia.org/wiki/C_(programming_language)
[2] Zbirka prevajalnikov GNU: http://gcc.gnu.org/
[3] Razvojno okolje Code::Blocks: http://www.codeblocks.org/
[4] C referenca na enem listu:
http://www.math.brown.edu/~jhs/ReferenceCards/CRefCard.v2.2.pdf

Literatura

[1] Mike Banahan, Declan Brady, Mark Doran: **The C Book**
(http://publications.gbdirect.co.uk/c_book/thecbook.pdf)
[2] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery:
Numerical Recipes in C (http://astronu.jinr.ru/wiki/upload/d/d6/NumericalRecipesinC.pdf)

2 Spremenljivke in funkcije

Vsebina:

- spremenljivke in podatkovni tipi,
- funkcije,
- demonstracija,
- povzetek,
- naloge.

2.1 Spremenljivke in podatkovni tipi

Da bi bili programi berljivi in razumljivi, C omogoča, da poimenujemo dele računalnikovega spomina z besedami. To nam omogoča koncept spremenljivk. Vsako spremenljiviko moramo na začetku funkcije, v kateri jo uporabljamo, najaviti. Spremenljivkam, ki jih najavimo znotraj funkcij pravimo lokalne spremenljivke. Osnovna oblika najave je naslednja:

```
podatkovni_tip ime_spremenljivke;
```

Npr.:

```
int hitrost;
```

Ob najavi spremenljivkam pogosto določimo začetno vrednost, čemur pravimo inicializacija:

```
int hitrost = 10;
```

Z enim ukazom lahko najavimo in/ali inicializiramo več spremenljivk, pri čemer morajo biti vse istega podatkovnega tipa:

```
int hitrost = 10, pospešek = 3, zasuk, oddaljenost = 4;
```

V C-ju obstaja 5 osnovnih podatkovnih tipov:

- `char` za podatke tipa znak (‘a’, ‘1’, ‘\n’, ‘\0’, …)
- `int` za celoštevilske spremenljivke (42, -255, 1337, …)
- `float` za števila s plavajočo vejico, enojna natančnost (3.14, -2.72, …)
- `double` za števila s plavajočo vejico, dvojna natančnost (3.1415926535897932)
- `void` za prazen podatkovni tip (posebnost, uporabno le pri funkcijah)

Podatkovni tip `char`

Spremenljvike tipa `char` zavzemajo 1 byte = 8 bitov spomina. To pomeni, da je v C-ju 2^8 različnih znakov. Znake, ki jih je možno uporabljati definira t.i. ASCII tabela oz. njene razširitve. V osnovi ima ASCII tabela 128 znakov, med katerimi je 32 t.i. kontrolnih znakov in 96 izpisljivih znakov. Nabor kontrolnih znakov vsebuje npr. ničti znak (NUL - hex koda 00), escape (ESC - koda 1B), novo vrstico (line feed - LF - koda 0A), tab (HT - koda 09), pisk (BEL - 07), …

Izpisljivi znaki so velike in male črke ter matematični in drugi simboli, npr. # (koda 23), ‘ (koda 60), …

Celoten nabor standardne ASCII tabele prikazuje slika 2-1.

ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Slika 2-1: Nabor standardnih znakov ASCII kodne tabele

Zgodovinsko ima ASCII tabela 7 bitno kodiranje, ker je 8. bit predstavljal pariteto. Danes je 8. bit uporabljen za kodiranje drugih 128 znakov, katerih oblika pa je odvisna od t.i. kodne strani (angl. code page). Za slovenske znake se uporablja kodna tabela ISO 8859-16 (Centralna, vzhodna in južna Evropa), v kateri so med drugim določene kode šumnikov. Na Windows operacijskem sistemu se uporablja nabor znakov Windows-1250.

Poglejmo, kakšen je izpis naslednjega programa:

```
#include <stdio.h>

int main(void)
{
      printf("čšž je težko izpisati...");
      return 0;
}
```

Izpis: ─ì┼í┼╛ je te┼╛ko izpisati...

Da bi prepričali konzolno okno, da izpiše šumnike, sta potrebna dva koraka:

- spremeniti Font na takega, ki podpira UTF-8 standard (Npr. Consolas na Windows 7)
- spremeniti kodno stran DOS okna na UTF-8 z ukazom:
  `chcp 65001`

Iz obravnave kodnih tabel se vrnimo na podatkovni tip `char`. Kot zapisano, dajemo znake v enojne narekovaje, npr. ‘a’, ‘A’, … Kadar vidimo tak zapis, preberemo: ASCII koda črke a oz. ASCII koda črke A. Za program ni nobene razlike med zapisom ‘a’, šestnajstiškim zapisom 0x61 ali desetiškim zapisom 97.

V C-ju obstaja kopica bližnjic za ASCII kode, npr. ‘\0’ za kodo 00, ‘\n’ za kodo 0A (nova vrstica), ‘\t’ za kodo 09 (tab), itn.

Podatkovni tip `char` lahko poleg tega, da ga uporabljamo za črke, uporabljamo tudi kot kratko, 8 bitno celo število.

Podatkovni tip `int`

Podatkovni tip `int` uporabljamo za cela števila. C standard ne predpisuje velikosti tega podatkovnega tipa. Običajno zaseda 32 bitov.

Za kodiranje predznačenih celih števil v RAMu je uporabljeno kodiranje z dvojiškim komplementom. Najpreprostejši način izračuna dvojiškega komplementa je naslednji:

- v bitni predstavitvi števila poišči prvo 1 z desne
- obrni vse bite levo od te 1

Primer za število 0x37 (55 v desetiškem, 00110111 v dvojiškem):

- `0011011`**`1`**
- **`1100100`**`1`

Dvojiški komplement se uporablja zato, da se ohranijo vsa pravila aritmetičnih operacij (seštevanje, množenje, …) tudi za negativna števila. Poleg tega je tako kodiranje maksimalno učinkovito, saj obstaja le ena ničla: 00000000.

Podatkovna tipa `float` in `double`

Podatkovna tipa `float` in `double` uporabljamo za števila z decimalno piko. Spremenljivke tipa `float` običajno zasedajo 32 bitov, tipa `double` pa 64 bitov spomina.

Kodiranje teh števil je naslednje. Pri 32 bitnih številih je:

- 1 bit za predznak,
- 8 bitov za eksponent in
- 23 bitov za mantiso.

Tak zapis je natančen na 7 signifikantnih cifer. Zapis v RAMu je predstavljen na sliki 2-2.

Slika 2-2: Zapis 32 bitnih števil z decimalno piko

Vrednost takega števila je, za predznačena 32 bitna števila z decimalno piko:

$$v = (-1)^s \cdot (1 + \sum_{i=1}^{23} b_{23-i} \cdot 2^{-i}) \cdot 2^{(e-127)}$$

Pri 64 bitnih številih je:

- 1 bit za predznak,
- 11 bitov za eksponent in
- 52 bitov za mantiso.

Tak zapis ja natančen na približno 16 signifikantnih cifer. Zapis v RAMu je predstavljen na sliki 2-3.

Slika 2-3: Zapis 64 bitnih števil z decimalno piko

Podatkovni tip `void`

Podatkovni tip `void` se uporablja le pri funkcijah, pri čemer označuje to, da funkcija ničesar ne sprejema in/ali ničesar ne vrača.

2.1.1 Modfikatorji, kvalifikatorji in druge ključne besede, vezane na spremenljivke

short/long

Velikost podatkovnih tipov ni standardizirana. Kljub temu pa so v C jasno definirani modifikatorji, s katerimi lahko njihovo velikost spreminjamo. Taka modifikatorja sta ključni besedi `short` in `long`.

S ključno besedo `short` prevajalniku povemo, da naj za shranjevanje podatkovnega tipa porabi (običajno polovico) manj prostora. Obratno, s ključno besedo `long` povemo, naj porabi (običajno polovico) več prostora, s čimer lahko shranjujemo večja števila in/ali pa jih shranjujemo bolj natančno.

Spremenljivke tipa int npr. modificiramo tako, da modifikator zapišemo pred podatkovni tip:

```
short int a = 8;
long int b = 1442349599333858823822;
```

operator sizeof

Količino spomina, ki ga porablja posamezna spremenljivka ali tip spremenljvike, lahko izpišemo s pomočjo operatorja `sizeof`:

```
printf("%zu", sizeof(char));
```

signed/unsigned

Poleg modifikatorjev za velikost spremenljivk obstajata tudi modifikatorja `signed` in `unsigned`, s katerima definiramo, da je določena spremenljivka predznačena oz. nepredznačena. Privzeto so spremenljivke predznačene, če želimo, da so nepredznačene, pa to zapišemo z naslednjo sintakso:

```
unsigned int a = 42;
```

const/volatile

Kvalifikatorja `const` in `volatile` omejujeta način dostopa in spreminjanja vrednosti spremenljivk. S `const` povemo, da vrednosti neke spremenljivke tekom programa ne bomo spreminjali, in da je vsako neposredno spreminjanje vrednosti spremenljivke napaka.

Z `volatile` povemo, da naj prevajalnik pogleda vrednost spremenljivke na vsakem mestu (vsakič) v programu. S tem preprečimo optimizacije, kot je npr. ta, da spremenljivko, ki je uporabljena znotraj ponavaljalne zanke prevajalnik prebere le na začetku izvajanja zanke. `volatile` se najpogosteje uporablja pri vgradnih elektronskih sistemih in mikrokrmilnikih, kjer lahko vrednost neke spremenljivke spreminja prekinitvena rutina. Primera najav:

```
const double pi = 3.1415;
volatile char port_3_2 = 1;
```

register

S ključno besedo register povemo, naj se ta spremenljivka, če je mogoče, shrani v register procesorja. Pogosto jo uporabljamo na mikrokrmilniških platformah, ko želimo doseči časovno gledano ponovljivo obnašanje spreminjanja vrednosti spremenljivke. Primer najave:

```
register int count = 1;
```

auto/static

Z `auto` določimo, da ima spremenljivka lokalno vidnost. Vse spremeljivke so privzeto auto, zato to ključno besedo le redko vidimo.

S ključno besedo `static` določimo, da se vrednost spremenljivke ohranja med posameznimi klici funkcije, v kateri je definirana. `static` spremenljivke so vidne le znotraj datoteke, v kateri so najavljene.

`static` spremenljivke dobijo prostor na delu spomina, imenovanem kopica. Ta spomin je na začetku izvajanja programa inicializiran na 0, zato imajo static spremenljivke privzeto vrednost 0.

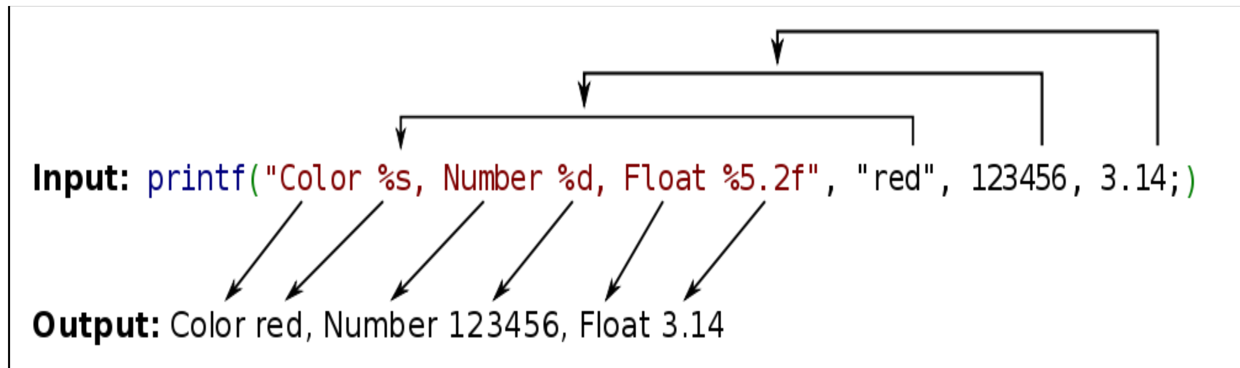
typedef

Ključno besedo typedef uporabljamo za krajšanje definicij tipov. Primer:

```
typedef unsigned int uint;
uint hitrost = 10;
```

2.1.2 Branje in izpisovanje spremenljivk s standardnega inputa/outputa (`printf, scanf`)

Za izpisovanje spremenljivk uporabljamo funkcijo `printf`. Princip prikazuje slika 2-4.



Slika 2-4: Princip izpisa konstant s printf

`printf` sprejme poljubno število argumentov, ločenih z vejico. Prvi argument je t.i. formatna beseda, v kateri so z znakom procent in črko označena mesta, kamor naj se zapišejo vrednosti vseh ostalih argumentov (drugega argumenta na prvo pojavitev %-a in črke, tretjega na drugo, …).

Na sliki 2-4 je prikazan primer, pri katerem se na prvo pojavitev `%s` zapiše “red”, na `%d` 123456 in na `%5.2f` 3.14. Najpogosteje bomo srečali:

- `%d` ali `%i` za izpis celih števil (int),
- `%f` za izpis float števil,
- `%c` za izpis znakov in
- `%s` za izpis besed.

Zapis `%5.2f` pomeni, da se float število zapiše na 5 mest, od tega 2 decimalki. V splošnem je oblika %-črka sintakse naslednja:

```
%-+ 0w.pmc
```

, pri čemer so:

- `-` leva poravnava
- `+` zapis s predznakom
- *presledek* zapis presledka, če je predznak +
- 0 zapis začetnik ničel
- w minimalna širina zapisa
- p natačnost
- m modifikator (h za short, l za long, L za long double)

- c   znak:
    - d,i integer
    - u unsigned
    - c char
    - s beseda
    - f double (printf)
    - e,E eksponent
    - f float (scanf)
    - lf double (scanf)
    - o osmiško
    - x,X šestnajstiško
    - p kazalec (pointer)
    - n number of chars written
    - g,G enako as f ali e,E v odvistnosti od eksponenta

Za branje spremenljivk s standardnega inputa uporabljamo funkcijo `scanf`. Primer uporabe:

```
int hitrost;
scanf("%d", &hitrost);
```

Znak & beremo kot *naslov od*. `scanf` namreč zahteva spominsko lokacijo, na katero naj se zapiše vpisana vrednost.

2.1.3 Aritmetične operacije

Aritmetične operacije so seštevanje +, odštevanje -, množenje *, deljenje / in ostanek pri deljenju %. Primer:

```
printf("%d %d %d", 3+2, 3/2, 3%2);
```

Izpis:

```
5 1 1
```

2.2 Funkcije

Funkcije uporabljamo za združevanje stavkov in blokov kode v zaključene celote, ki opravljajo zaključeno funkcionalnost. Obsežnejše programe nato sestavimo iz funkcij. Do sedaj je bila prikazana uporaba funkcij iz standardnih knjižnic. C pa omogoča tudi definicijo lastnih funkcij. Sintaksa za definicijo lastne funkcije je naslednja:

```
return_tip ime_funkcije (tip1 argument1, tip2 argument2, …)
{
      telo_funkcije;
}
```

Primer definicije preproste funkcije, ki izpiše novo vrstico:

```
void nova_vrstica (void)
{
      printf("\n");
      return;
}
```

Tako funkcijo lahko nato uporabimo v kodi:

```
printf("To je v prvi vrstici...");
nova_vrstica();
printf("To je v drugi vrstici...");
```

Za najavo funkcij uporabljamo t.i. prototipe. Prototipe zapišemo na vrh datoteke ali v datoteke s končnico .h (datoteke glave). Za obravnavano funkcijo ima prototip naslednjo obliko:

```
void nova_vrstica (void);
```

, torej je enak glavi funkcije (pravimo: podpisu funkcije), le da je na koncu dodano podpičje.

Spremenljivke, ki jih definiramo na začetku funkcij so vidne le znotraj funkcije. Takim spremenljivkam pravimo *lokalne* spremenljivke. Spremenljivke, ki so definirane zunaj vseh funkcij (tudi zunaj funkcije main), pa so *globalne* spremenljivke.

Pomembno je, da se spomin za lokalne spremenljivke in za argumente funkcije alocira na skladu, ki je del spomina, rezerviran za začasne definicije. Ko se funkcija zaključi, se del sklada, v katerem so bile shranjene spremenljivke te funkcije, izprazni.

Globalne spremenljivke so shranjene na t.i. kopici (heap).
Prav tako je pomembno, da se v funkcijo prenese kopija vrednosti njenega argumenta, ne pa argument sam. Princip prikazuje naslednji primer:

```
#include <stdio.h>

int main(void)
{
      int n = 5;
      add1(n);
      printf("%d", n);
}

int add1(int n)
{
```

```
      n = n+1; // enako kot n++ ali n+=1
      return n;
}
```

Izpis:

```
5
```

2.3 Povzetek

Predprocesorski ukazi:

- #include <>

Funkcije (knjižnice):

- printf (iz stdio.h), scanf (iz stdio.h)

Ključne besede:

- int, void, main, return, char, float, double, short, long, sizeof, signed, unsigned, const, volatile, register, auto, static, typedef

Operatorji:

- sizeof, aritmetični (+, -, *, /, %)

Razno:

- prevajanje programa iz komandne vrstice
- prevajanje z uporabo IDE
- najava in inicializacija spremenljivk
- kodiranje celoštevilskih in float tipov
- aritmetične operacije
- lokalne/globalne spremenljivke
- sklad (stack), kopica (heap)
- komentarji // in /* */

2.4 Naloge

Naloga 1

Napiši program, ki izpiše velikosti (v bitih) 5-ih podatkovnih tipov (z ali brez modifikatorjev).

Naloga 2

Napiši program, ki sprejme dve celi števili prek standardnega inputa (konzole) in izpiše njuno vsoto, razliko, zmnožek, ju zdeli in izpiše tudi ostanek pri deljenju.

Naloga 3

Izračunaj (zapiši tudi postopek izračuna), kakšno je največje predznačeno število, ki ga še lahko zapišemo v 64 bitno celoštevilsko spremenljivko.

Dodatne naloge

Razišči, kako je z natančnostjo, kadar izvajamo aritmetične operacije na podatkih različnih tipov (npr. množenje celega števila s float ali double vrednostjo).

Zapiši programa iz 1. in 2. naloge v obliki template-a, definiranega na naslednji strani.

2.5 Predloga datoteke za programe

Za vse programe bomo uporabljali enak način zapisa, ki je prikazan na naslednji strani.

```c
/*
 * datoteka: main.c
 * datum:    15.2.2012
 * avtor:    Ime Priimek
 * opis:     Program sesteje dve stevili tipa int in rezultat
 *           izpise na zaslonu.
 */

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

// funkcijski prototipi
int sestej(int a, int b);

// globalne spremenljivke
int stevec = 0;


int main(void)
{
	int a = 2, b = 3;
	printf("Rezultat je: %d\n", sestej(a, b));
	printf("sestej() je bila uporabljena %d-krat.\n", stevec);
	return 0;
}

/*
 * opis:   Funkcija sesteje dve stevili tipa int.
 * arg:    a: Prvo stevilo za izvedbo sestevanja.
 * arg:    b: Drugo stevilo za izvedbo sestevanja.
 * return: Vsota vhodnih argumentov a in b.
 */
int sestej(int a, int b)
{
	stevec++;
	return a + b;
}
```

3 Krmilne strukture in operatorji

Vsebina:

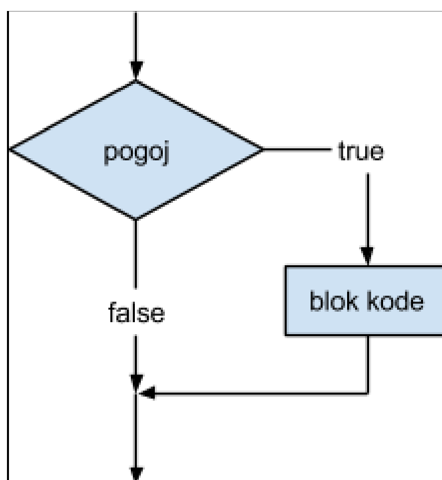
- krmilne strukture,
- relacijski in logični opreatorji,
- demonstracija,
- povzetek,
- naloge.

3.1 Krmilne strukture

Pogosto računalniški programi ne predstavljajo zgolj nespremenljivega, vnaprej določenega zaporedja ukazov, temveč med potekom izvajajo odločitve (vejitve) in ponovitve posameznih segmentov kode glede na dane okoliščine. Za ta namen so v jeziku C prisotne krmilne strukture s katerimi krmilimo potek programa.

3.1.1 if

If predstavlja osnovno odločitveno strukturo, ki določa potek izvajanja programskih stavkov ali blokov kode. Za krmiljenje poteka uporablja testni izraz (pogoj), katerega vrednost se izračuna pred izvedbo odločitve. Testni izraz je lahko kateri koli veljaven C izraz. Vrednost testnega izraza je lahko "true" (vse vrednosti, različne od nič) ali "false" (nič). Na podlagi izračunane vrednosti se izvedejo ali preskočijo določeni programski stavki ali bloki (slika 3-1).



Slika 3-1: Odločitvena struktura if.

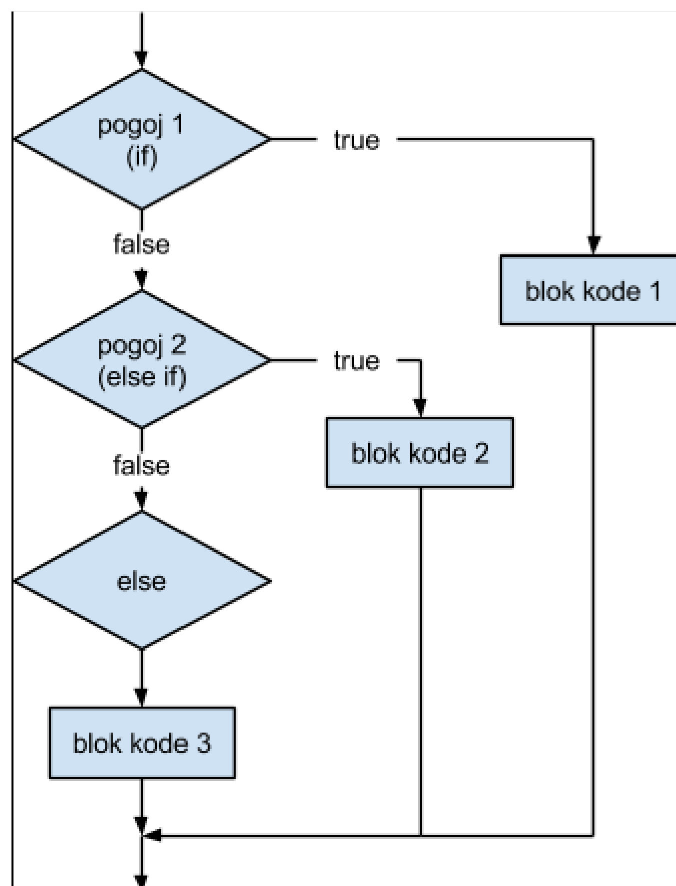
Sintaksa odločitvene strukture if je naslednja:

```
if(pogoj)
      stavek _ali_blok_kode;
```

Primer uporabe if stavka:

```
if(5 > 3)
      printf("Ta stavek se bo izvedel.");
if(5 >= 7)
      printf("Ta stavek se NE bo izvedel.");
```

Testni izraz (pogoj) pogosto vsebuje relacijske in logične opreatorje (<, >=, &&,...), ki bodo predstavljeni v naslednjem podpoglavju.
Odločitveno strukturo if lahko razširimo z uporabo ključne besede `else` s katero določimo kaj naj se zgodi, ko pogoj ni izpolnjen (testni izraz ima vrednost nič). Nadalje lahko strukturo razširimo še z dodajanjem `else if`, ki vnese dodatne pogoje za izvrševanje določenega bloka kode (slika 3-2).



Slika 3-2: Odločitvena struktura if/else.

Sintaksa odločitvene strukture if/else if/else je naslednja:

```
if(pogoj_1)
{
      blok_kode_1
}
else if(pogoj_2)
```

```
{
      blok_kode_2
}
else
{
      blok_kode_3
}
```

Odločitvena struktura lahko vsebuje večje število else if. Primer:

```
if(kolokvij >= 90)
{
      printf("odlicno”);
}
else if(kolokvij >= 70)
{
      printf("dobro");
}
else if(kolokvij >= 50)
{
      printf("zadostno");
}
else
{
      printf("nezadostno\a");
}
```

Pri odločitveni strukturi if se pogoji preverjajo od zgoraj navzdol. Ko je določen pogoj izpolnjen, se izvede pripadajoči blok kode, vsi ostali bloki pa se preskočijo. Če ni izpoljen nobeden izmed pogojev se izvede blok kode za else.

Strukturo if lahko poljubno vgnezdimo v drugo strukturo if in to spet v naslednjo. Primer:

```
if(a > 10)
{
      if(b > 10)
            printf("a in b sta večja od 10");
      else
            printf("samo a je večji od 10");
}
```

3.1.2 switch/case

Odločitvena struktura if je primerna za izbiro med dvema možnostima. Če potrebujemo izbiro med več možnostmi lahko uporabimo vgnezdene if strukture, vendar pa ob njihovi pretirani rabi program tako hitro postane nepregleden. C zato ponuja rešitev v obliki izbirne strukture

switch. Njena sintaksa je sledeča:

```
switch(spremenljivka)
{
      case konstanta_1:
            blok_kode_1
      break;
      case konstanta_2:
            blok_kode_2
      break;
      default:
            blok_kode_n
}
```

Za razliko od if strukture je `spremenljivka` v switch strukturi lahko zgolj tipa `int` ali `char`. Vrednost spremenljivke se primerja s konstantami za ključno besedo `case`. V primeru ujemanja se izvede pripadajoči blok kode do ključne besede `break`. Če v po koncu bloka kode ni besede `break`, se izvajanje programa nadaljuje v naslednji blok kode. Kadar spremenljivka ni enaka nobeni izmed konstant za ključno besedo `case`, se izvede privzeti blok kode (`default`). Primer:

```
#include <stdio.h>

int main()
{
      int num;
      printf("Vpisi stevilo!\n");
      scanf("%d", &num);
      switch(num)
      {
            case 1:
                  printf("ena\n");
            break;
            case 2:
                  printf("dve\n");
            break;
            default:
                  printf("neznano stevilo\n");
      }
      return 0;
}
```

V vseh primerih strukture switch ne moremo uporabiti. Kadar pa jo lahko uporabimo in imamo 3 ali več možnosti med katerimi želimo izbrati, je uporaba switch priporočena. Razlog za to je predvsem večja preglednost programa ter v nekaterih primerih (odvisno od prevajalnika in procesorja) tudi večja hitrost izvedbe.

3.1.3 for

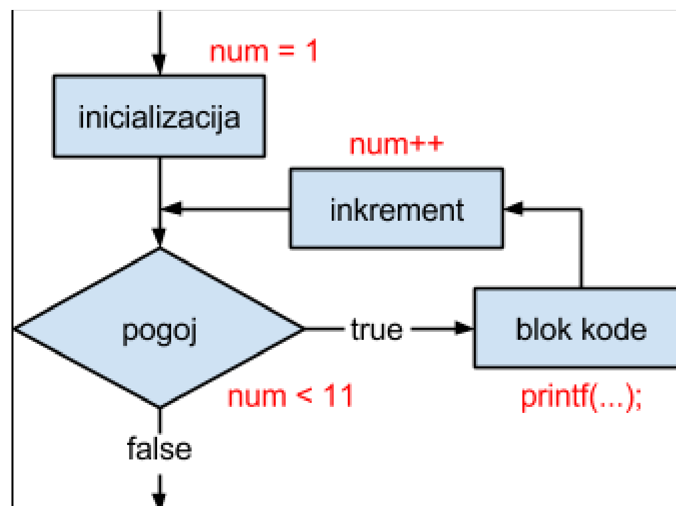
For zanka je ena izmed treh ponavljalnih struktur jezika C. Zanka omogoča ponavljanje posameznih izrazov ali blokov kode. Sintaksa for zanke je naslednja:

```
for(inicializacija; pogoj_za_ponavljanje; inkrement)
{
	blok_kode
}
```

Inicializacija pomeni določanje začetne vrednosti kontrolne spremenljivke, ki jih pogosto označimo z i, j, k, … Pogoj za ponavljanje predstavlja testiranje vrednosti kontrolne spremenljivke. Ponavljanje se nadaljuje, če je pogoj “true”. Inkrement se zgodi na koncu, za blokom kode. Velikokrat v inkrementu povečamo kontrolno spremenljivko, na primer i++, kar pomeni i = i + 1. Primer

```
#include <stdio.h>

int main()
{
	int num;
	for (num = 1; num < 11; num++)
	{
		printf("%d ", num);
	}
return 0;
}
```

Prikazani primer na zaslonu izpiše številke od 1 do 10. Na sliki 3-3 je prikazano izvajanje for zanke še v grafični obliki.



Slika 3-3: Ponavljalna struktura for.

3.1.4 while, do/while

While je druga izmed ponavljalnih struktur (zank) jezika C. While zanka ponavlja nek izraz ali blok kode toliko časa, dokler je pogoj “true”. Sintaksa je naslednja:

```
while (pogoj)
{
      blok_kode
}
```

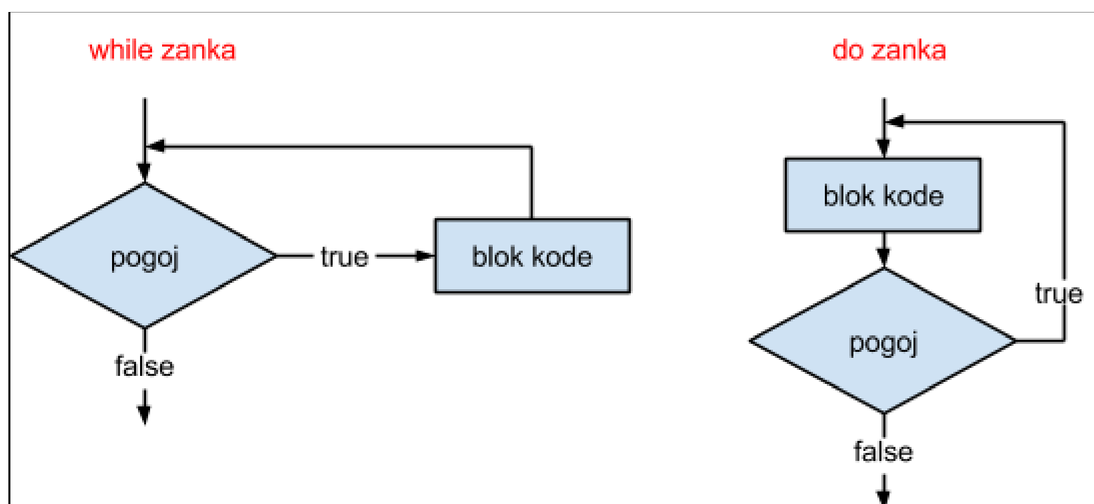
Do je zadnja izmed ponavljalnih struktur jezika C. Njena sintaksa je:

```
do
{
      blok_kode
} while (pogoj);
```

Razlika med zankama while in do je naslednja:

- while preverja pogoj na začetku zanke,
- do preverja pogoj na koncu zanke.

To pomeni, da se blok kode v do zanki v vsakem primeru izvede vsaj enkrat. Pri while zanki pa to ni nujno. Če je pogoj na začetku while zanke “false” se blok kode ne izvede niti enkrat. Razilka med zankama je vidna tudi na sliki 3-4.



Slika 3-4: While in do zanka.

Primer uporabe while zanke:

```
#include <stdio.h>

int main()
{
```

```
	int num = 1;
	while (num <= 10)
	{
		scanf("%d", &num);
		printf("Kvadrat: %d\n", num * num);
	}
	return 0;
}
```

3.1.5 Izhod iz zank

Ključna beseda `break` omogoča, da na kateremkoli mestu prekinemo izvajanje zanke (for, while ali do). Program po tem nadaljuje z izvajanjem stavka za zanko. Primer:

```
#include <stdio.h>

int main()
{
	int num;
	for (num = 1; num < 100; num++)
	{
		printf("%d ", num);
		if(num == 10)
			break;
	}
	return 0;
}
```

Program iz primera bo izpisal samo števila od 1 do 10, saj se for zanka prekine, ko je `num` enak 10.

3.1.6 Skok v naslednjo iteracijo zanke

S ključno besedo `continue` vsilimo izvajanje naslednje iteracije zanke. Vsa koda, ki je med stavkom `continue` in pogojem za izvedbo zanke se preskoči. Primer:

```
#include <stdio.h>

int main()
{
	int num;
	for (num = 1; num < 100; num++)
	{
		if(num == 10)
			continue;
		printf("%d ", num);
```

```
        }
        return 0;
}
```

Program iz primera izpiše vsa števila od 1 do 99 razen števila 10, saj se v tem primeru izvede stavek `continue` in zato zanka preskoči v naslednjo iteracijo.
Stavek `continue` se redko uporablja.

3.2 Relacijski in logični operatorji

Do sedaj smo spoznali krmilne strukture (if, switch, for, while, do), ki pogosto uporabljajo pogoje (testne izraze). Vrednost le-teh je lahko "true" ali "false". Pogoje lahko sestavimo s pomočjo relacijskih in logičnih operatorjev.

Relacijski operatorji
Relacijski operatorji primerjajo dve vrednosti in vrnejo rezultat, ki je lahko "true" (1) ali "false" (0). Relacijski opreatorji so naslednji:

- `>` večje
- `>=` večje ali enako
- `<` manjše
- `<=` manjše ali enako
- `==` enako (POZOR: `==` ni enako kot `=`)
- `!=` ni enako

Pri operatorju `==` je potrebna posebna pozornost, saj gre za primerjavo med vrednostjo na levi in desni strani. Če sta vrednosti enaki je rezultat "true", sicer "false". Operator `=` pa predstavlja prirejanje vrednosti izraza ali spremenljivke na desni strani enačaja spremenljivki na levi strani.

Logični opreatorji
Logični operatorji povezujejo vrednosti "true/false" med seboj. V C-ju poznamo naslednje logične opreatorje:

- `&&` AND
- `||` OR
- `!` NOT

Pravilnostna tabla logičnih operatorjev je naslednja (0 ustreza "false" in 1 ustreza "true"):

p	q	p && q	p || q	!p
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Z uporabo relacijskih in logičnih operatorjev lahko sestavimo kompleksne pogoje kot na primer:

Če je (a > 0 in hkrati b < 0) ali pa če je c > 12:

```
if ((a > 0 && b < 0) || c > 12)
{
      ...
}
```

Pri tem je potrebna pozornost na prioritete posameznih operatorjev. Relativne prioritete realcijskih in logičnih operatorjev so sledeče:

`!` višja prioriteta
`> >= < <=`
`== !=`
`&&`
`||` nižja prioriteta

Za določanje želenega vrstnega reda izvajanja relacijskih in logičnih operaracij uporabimo oklepaje. Primer:
`a > 0 && b > 0 || d > 0` ni enako kot `a > 0 && (b > 0 || d > 0)`

Najprej se izračunajo izrazi z relacijskimi opreatorji, ki imajo višjo prioriteto kot logični opreatorji. Če si izberemo vrednosti a = -1, b = 0, d = 1, dobimo torej:
`0 && 0 || 1` in v primeru z oklepaji `0 && (0 || 1)`
Ko nadaljujemo z izračunom dobimo:
`1` in v primeru z oklepaji `0`

Pravilno postavljanje oklepajev je bistvenega pomena. Pogosto je bolje postaviti več oklepajev s katerimi natančno določimo vrstni red opreacij in naredimo program razumljiv tudi drugim.

3.3 Povzetek

Predprocesorski ukazi:

- `#include <>`

Funkcije (knjižnice):

- `printf (iz stdio.h), scanf (iz stdio.h)`

Ključne besede:

- `int, void, main, return, char, float, double, short, long, sizeof, signed, unsigned, const, volatile, register, auto, static, typedef, if, else, switch, case, break, default, for, while, do, continue,`

Operatorji:

- `sizeof,`
- aritmetični `(+, -, *, /, %, ++ ,--)`
- relacijski operatorji (`>, >=, <, <=, ==, !=`)
- logični opreatorji (`&&, ||, !`)

Razno:

- prevajanje programa iz komandne vrstice
- prevajanje z uporabo IDE
- najava in inicializacija spremenljivk
- kodiranje celoštevilskih in float tipov
- aritmetične operacije
- lokalne/globalne spremenljivke
- sklad (stack), kopica (heap)
- komentarji // in /* */
- odločitvene in izbirne strukture
- ponavljalne strukture (zanke)
- relacijske in logične operacije

3.4 Naloge

Naloga 1

Napišite program, ki vsebuje lastno funkcijo za izračun variance. Funkcija sprejme 3 argumente tipa double in vrne njihovo varianco.

Naloga 2

Napišite program, ki od uporabnika zahteva vnos celega števila. Če je število med 1 in 4, program vnešeno število izpiše z besedo, na primer “stiri”. Če je vnešeno število manjše od 1, program izpiše “stevilo je manjse od 1”, v vseh ostalih primerih pa program izpiše “stevilo je vecje od 4”.

Naloga 3

Napišite program, ki preveri ali je vneseno celo število kvadrat nekega drugega celega števila. Celoten postopek naj se ponovi 10-krat.

Dodatne naloge

Raziščite, zakaj je v nekaterih primerih struktura switch hitrejša kot if/else.

4 Bitni operatorji in algoritmi

Vsebina:
- bitni operatorji,
- algoritmi,
- povzetek,
- naloge.

4.1 Bitni operatorji

Bitne operatorje uporabljamo za manipulacijo podatkov na nivoju bitov. V C-ju so definirani naslednji bitni operatorji:

- & and
- | or
- ^ xor
- ~ eniški komplement
- << shift levo
- >> shift desno

Pravilnostne tabele bitnih operatorjev:

p	q	p & q	p | q	p ^ q	~p
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Primer uporabe bitnega operatorja &:

```
  1010 0110
& 0011 1011
= 0010 0010
```

Primer uporabe bitnega operatorja shift levo:

```
<< 0010 0110
 = 0100 1100
```

Bitni operatorji so pomembni pri programiranju mikrokrmilnikov, saj omogočajo nastavljanje ali spreminjane posameznih bitov. Vzemimo za primer register, katerega vrednost je 1010 0110.

Postavljanje bita na 1 z bitnim operatorjem or:

```
  1010 0110 //začetna vrednost registra
| 0001 0000 //maska
= 1011 0110 //končna vrednost
```

Postavljanje bita na 0 z bitnim operatorjem and in invertirano masko:

```
  1010 0110 //začetna vrednost registra
& 1101 1111 //maska
= 1000 0110 //končna vrednost
```

Spreminjane bita z bitnim operatorjem xor:

```
  1010 0110 //začetna vrednost registra
^ 0000 0001 //maska
= 1010 0111 //končna vrednost
```

4.2 Algoritmi

Poznavanje principov gradnje algoritmov in najpogosteje uporabljenih algoritmov je zelo pomembno za vsakega programerja. V algoritmih so zakodirani postopki, ki rešujejo točno določene probleme. Pomembno je, da so postopki neodvisni od programskega jezika. Poznavanje gradnje algoritmov je zato splošneje uporabno za vso programiranje, ne samo za programiranje v C-ju.

V tem delu bo predstavljenih nekaj osnovnih algoritmov z namenom prikaza logike gradnje le-teh.

Primer 1:
Napiši program, ki pove, ali je naravno število, ki ga vnese uporabnik, kvadrat celega števila.

Ko se soočimo s takšnim problemom je najbolje, da naredimo načrt na papir. Načrt mora natančno definirati postopek, to je zaporedje operacij, ki nas bo pripeljal do rešitve. Za prvi primer bi lahko razmišljali nekako takole:

- vnešeno število korenimo,
- zaokrožimo navzdol,
- ponovno kvadriramo,
- preverimo ali je ponovno kvadrirano število enako vnešenemu.

Nato vsak posamezen korak razdelamo, dokler ne pridemo do C kode, ki posamezen korak reši. Npr:

- vnešeno število korenimo

Poiščemo funkcijo za korenjenje. Izkaže se, da ima podpis double sqrt(double n) in da se

nahaja v knjižnici math.h. Koda za ta korak bo torej:

```
#include <math.h>
…
      int vneseno_stevilo;
      double koren;
      // scanf za določanje vrednosti spremenljivke vneseno_stevilo
      …
      koren = sqrt(stevilo);
```

- zaokrožimo navzdol

Za to je mogoče uporabiti kar eksplicitno pretvorbo (cast) tipa double v tip int. Definirati bomo morali še eno spremenljivko, int stevilo;

```
      int stevilo;
      …
      stevilo = (int) koren; // cast double-a v int
```

- ponovno kvardiramo

Za to uporabimo novo spremenljivko, kvadrat.

```
      int kvadrat;
      …
      kvadrat = stevilo * stevilo;
```

- preverimo ali je ponovno kvadrirano število enako vnešenemu (in izpišemo)

```
      if (kvadrat == vneseno_stevilo)
            printf("Vneseno stevilo je kvadrat stevila %d.",
      stevilo);
      else
            printf("Vneseno stevilo ni kvadrat naravnega stevila.");
```

Preostane nam, da vse korake združimo v en program:

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

int main(void)
{
    int vneseno_stevilo, stevilo, kvadrat;
    double koren;
    printf("Vnesite stevilo:\n");
    scanf("%d", &vneseno_stevilo);
    koren = sqrt(vneseno_stevilo);
```

```
    stevilo = (int)koren;
    kvadrat = stevilo * stevilo;
    if (kvadrat == vneseno_stevilo)
        printf("Vneseno stevilo je kvadrat stevila %d.", stevilo);
    else
        printf("Vneseno stevilo ni kvadrat naravnega stevila.");
    return 0;
}
```

Pri načrtovanju algoritmov je skoraj vedno možnih več pristopov k reševanju problema. Isto nalogo bi lahko rešili tudi tako, da bi z zanko kvadrirali zaporedna naravna števila in preverjali, ali je kvadrat števila slučajno enak vnešenemu številu. Postopek bi prekinili, ko kvadrat naravnega števila preseže vrednost vnešenega števila. Ker ne vemo, kolikokrat bo potrebno postopek ponoviti je bolje, da vzamemo ponavljalno strukturo do/while, kot for. Primer rešitve bi bil npr.:

```
#include <stdio.h>

int main(void)
{
    int vneseno_stevilo, stevilo, kvadrat;
    printf("Vnesite stevilo:\n");
    scanf("%d", &vneseno_stevilo);
    stevilo = 1;
    do
    {
        kvadrat = stevilo * stevilo;
        if (kvadrat == vneseno_stevilo)
        {
            printf("Vneseno stevilo je kvadrat stevila %d", stevilo);
            return 0;
        }
        stevilo++;
    } while (kvadrat <= vneseno_stevilo);
    printf("Vneseno stevilo ni kvadrat naravnega stevila.");
    return 0;
}
```

Oba algoritma data enak rezultat, kljub temu, da je uporabljen povsem različen pristop. Sledi torej, da je izbira postopka pomembna, saj lahko z dobrimi algoritmi programe pohitrimo ali pa uspemo s programom uporabiti manj pomnilniškega prostora.

Primer 2:
Napiši program, ki izračuna vsoto vseh naravnih števil, manjših od 1000, ki so večkratniki števil 3 ali 5.

Rešitev: 233168

Postopek:

- za vsa števila med 1 in 999
  - če je število deljivo s 3 ali s 5 ga prištej spremenljivki vsote (sum)

Primer rešitve:

```
    int i;
    int sum = 0;
    for (i = 1; i < 1000; i++)
    {
        if (i%3 == 0 || i%5 == 0)
        {
            sum += i;
        }
    }
    printf("%d", sum);
```

Primer 3:
Napiši funkcijo, ki pove, ali je neko število praštevilo. Argument, ki ga funkcija sprejema naj bo tipa unsigned long long. Funkcija naj vrne int 1, če je število praštevilo in 0, če ni.

Postopek:

- za vsa števila med 2 in korenom števila
  - če ga število deli, vrni 0
- če ga nobeno izmed teh števil ne deli, vrni 1 (število je praštevilo)

Primer rešitve:

```
int is_prime(int number)
{
    int i;
    for (i = 2; i <= sqrt(number); i++)
    {
        if(number%i == 0)
            return 0;
    }
    return 1;
}
```

Dodatno: napiši program, ki to funkcijo uporabi za izračun 10001. praštevila.

Primer 4:

Napiši funkcijo, ki izračuna fakulteto podanega celega števila.

Postopek:

- za vsa števila med 1 in podanim številom
    - zmnoži število s spremenljivko produkta (factorial)

Primer rešitve:

```
int factorial(int number)
{
    int i, factorial = 1;
    for (i = 1; i <= number; i++)
    {
        factorial = factorial * i;
    }
    return factorial;
}
```

4.3 Povzetek

Predprocesorski ukazi:

- #include <>

Funkcije (knjižnice):

- printf (iz stdio.h), scanf (iz stdio.h)

Ključne besede:

- int, void, main, return, char, float, double, short, long, sizeof, signed, unsigned, const, volatile, register, auto, static, typedef, if, else, switch, case, break, default, for, while, do, continue

Operatorji:

- sizeof,
- aritmetični (+, -, *, /, %, ++ ,--)
- relacijski operatorji (>, >=, <, <=, ==, !=)
- logični opreatorji (&&, ||, !)
- bitni operatorji (&, |, ^, ~, <<, >>)

Razno:

- prevajanje programa iz komandne vrstice
- prevajanje z uporabo IDE
- najava in inicializacija spremenljivk
- kodiranje celoštevilskih in float tipov
- aritmetične operacije
- lokalne/globalne spremenljivke
- sklad (stack), kopica (heap)
- komentarji // in /* */
- odločitvene in izbirne strukture
- ponavljalne strukture (zanke)
- relacijske in logične operacije
- bitne operacije
- načrtovanje in pisanje algoritmov

4.4 Naloge

Naloga 1
Napišite program, ki bo proti vam igral igro ugibanja števila. Igra ugibanja števila poteka tako, da si zamislite poljubno število med 0 in 100, potem pa pustite programu, naj to število ugane. Za vsak poskus programu sporočite le, ali je vaše število manjše, večje ali enako poskusu.

Oblikujte strategijo, po kateri bo program ugibal (ni nujno, da uporabite optimalno).

Primer izpisa:

```
Ste si zamislili stevilo 50?
v
75?
v
87?
m
82?
v
85?
e
Zamislili ste si stevilo 85!
```

Naloga 2
Napišite program, ki izračuna konstanto pi na 10 decimalk. Za izračun uporabite naslednjo formulo:

$$\pi = 4\sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1} = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \cdots$$

5 Algoritmi (nadaljevanje)

Vsebina:
- algoritmi,
- povzetek,
- naloge.

5.1 Primeri

Primer 1:
Napišite program za pretvorbo števila iz desetiškega v dvojiški sistem.

Te naloge se lahko lotimo na več načinov. Eden izmed njih je izračunavanje ostanka pri deljenju z 2, ki ga običajno uporabljamo pri pretvorbi na papirju, drugi je uporaba bitnih operatorjev, kar bomo prikazali v nadaljevanju.

Postopek:
- preberi število,
- ustvari masko,
- izvedi bitno operacijo & med masko in številom,
- glede na rezultat operacije & izpiši “1” ali “0”,
- izvedi shift desno maske za en bit,
- ponavljaj predhodne tri korake dokler je maska večja od 0.

Pri obravnavi bitnih operatorjev smo spoznali, da lahko z njihovo uporabo izvajamo manipulacije na nivoju bitov. Sedaj pa si poglejmo kako izvedemo posamezne korake.

- preberi število

Ko število preberemo (npr. s funkcijo scanf in %d, ker beremo število), se njegova vrednost zapiše v spomin v binarni obliki. Če npr. vnesemo število 13 in ga program interpretira kot 8-bitni `unsigned char`, potem se v predel spomina, ki je rezerviran za naše število zapiše:
`00001101`

- ustvari masko

Sedaj naredimo novo spremenljivko tipa `unsigned char`, ki ji bomo rekli maska in ima vrednost 128 (v šestnajstiškem sistemu je to 0x80) kar se v spominu zapiše v binarni obliki kot:
`10000000`

- izvedi bitno operacijo & med masko in številom

```
  00001101   //stevilo (13)
& 10000000   //maska (128)
= 00000000   //rezultat
```

- glede na rezultat operacije & izpiši “1” ali “0”

V kodi to realiziramo z odločitveni strukturo if:

```
if((stevilo & maska) == 0)
      printf("0");
else
      printf("1");
```

Posebej pomembno je, da najprej izvedemo bitni & in šele nato primerjavo (==). Če izraz zapišemo brez oklepajev, se zaradi večje prioritete operatorja == le-ta izvede pred bitnim &.

- izvedi shift desno maske za en bit

Do sedaj smo prebrali samo prvi bit števila. Da preberemo še ostale, moramo spremeniti masko. Iz zapisa maske v binarni obliki 10000000 želimo dobiti 01000000 s čimer bomo prebrali drugi bit števila. To izvedemo z bitnim operatorjem shift desno:

```
maska = maska >> 1;
```

S tem stavkom dosežemo, da se vsi biti spremenljivke maska premaknejo za 1 v desno.

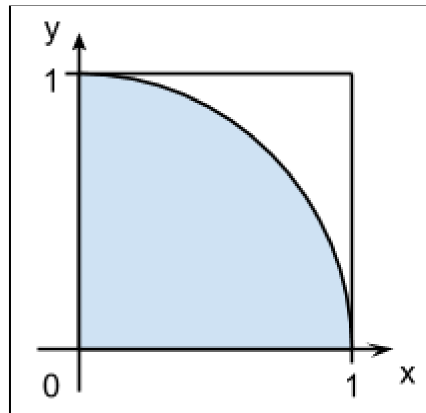
- ponavljaj predhodne tri korake dokler je maska večja od 0

Po izvedbi operacije shift desno ponovno izvedemo logično & operacijo in izpišemo 0 oz 1 glede na rezultat.
Primer rešitve:

```
#include <stdio.h>

int main(void)
{
      unsigned char stevilo, maska;
      printf("podaj stevilo: ");
      scanf("%d", &stevilo);
      maska = 128;
      printf("Binarni zapis stevila %d = ", stevilo);
      while(maska > 0)
      {
            if((stevilo&maska) == 0)
                  printf("0");
            else
                  printf("1");
            maska = maska >> 1;
      }
}
```

Primer 2:
Napišite algoritem, ki bo z uporabo generatorjia psevdo-naključnih števil izračunal približek števila pi.

Za izvedbo primera 2 bomo uporabili simulacijo Monte Carlo. Oznaka Monte Carlo združuje skupino metod, ki za določanje rezultatov uporabljajo naključno generirane razporeditve. Takšne metode se danes uporabljajo na številnih področjih npr. v ekonomiji, fiziki, kemiji, tehniki in drugje. Najbolj primerne so predvsem, kadar analitični izračuni niso izvedljivi.
Zamislimo si primer na sliki 5-1.



Slika 5-1: Del krožnice z enotskim polmerom.

Četrtina kroga s polmerom 1 enoto ima površino pi*1^2/4, medtem ko ima kvadrat površino 1^2. Če iz območja, ki ga omejuje kvadrat naključno izbiramo točke v ravnini, bodo nekatere padle v območje kroga, druge pa izven njega. Če je verjetnost izbire vseh točk v območju kvadrata enaka, je razumno pričakovati, da bo število naključno izbranih točk z območja kroga premo sorazmerno z njegovo površino. Zato lahko zapišemo:

$$\frac{\pi/4}{1} = \frac{n_{kr}}{n}$$

Pri čemer je n_{kr} število naključno izbranih točk iz območja kroga ter n število vseh naključno izbranih točk (oz. število točk z območja kvadrata). Če torej poznamo n_{kr} in n, lahko izračunamo oceno števila pi.

Postopek:
- naključno izberi točko z območja kvadrata,
- preveri, ali točka leži v krogu,
    - če da, povečaj n_{kr}
- ponovi prva dva koraka n-krat,
- izračunaj približek števila pi z uporabo n in n_{kr}.

Razdelani koraki postopka sledijo v nadaljevanju.

- naključno izberi točko z območja kvadrata

V C-ju imamo na voljo generator naključnih števil v obliki funkcije rand(), ki se nahaja v knjižnici stdlib.h. Funkcija vrne psevdo-naključno celo število z intervala [0 - RAND_MAX].

Pred prvim klicem funkcije rand() običajno kličemo funkcijo srand(), ki inicializira generator naključnih števil. Argumentu funkcije srand() pravimo seme. Različna semena rezultirajo v različnih zaporedjih psevdo-naključnih števil, ki jih generira funkcija rand(). Če želimo pri vsakem zagonu programa imeti drugo seme, lahko inicializacijo generatorja naključnih števil izvedemo tako:

```
srand ( time(NULL) );
```

Pri tem funkcija time(NULL) iz knjižnice time.h vrne število sekund od 1. januarja 1970.
Sedaj želimo generirati naključna števila med 0 in 1, s čimer naključno izbiramo koordinate točk znotraj kvadrata na sliki 5-1. To izvedemo z naslednjima vrsticama:

```
x = (double)rand()/RAND_MAX;
y = (double)rand()/RAND_MAX;
```

S tem smo dobili koordinati x in y, ki imata vrednosti tipa double med 0 in 1. Uporabili smo t.i. casting za začasno pretvorbo celega števila, ki ga vrne funkcija rand(), v tip double.

- preveri, ali točka leži v krogu,
    - če da, povečaj n_{kr}

```
if(x*x + y*y <= 1)
      nkr++;
```

- ponovi prva dva koraka n-krat

```
for(i=0; i < n; i++)
…
```

- izračunaj približek števila pi z uporabo n in n_{kr}

```
pi = 4*(double)nkr/n;
```

Primer rešitve:

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

int main(void)
{
      double x, y;
      int i, nkr = 0, n = 10000000;
      srand(time(NULL));
      for(i=0; i < n; i++)
      {
            x = (double)rand()/RAND_MAX;
```

```
            y = (double)rand()/RAND_MAX;
            if(x*x + y*y <= 1.0)
                  nkr++;
      }
      printf("Priblizek pi je: %lf\n", 4*(double)nkr/n);
      return 0;
}
```

Primer 3:
Napišite program za učenje poštevanke. Program naj izpiše 10 matematičnih izrazov z naključno generiranimi števili med 1 in 10 in pri tem šteje pravilne odgovore in meri čas.

Postopek:
- prični z merjenjem časa,
- naključno generiraj dve celi števili med 1 in 10,
- uporabi ju v matematičnem izrazu,
- preberi uporabnikov odgovor,
- če je odgovor pravilen, povečaj števec pravilnih odgovorov,
- ponovi postopek 10-krat (razen začetka merjenja časa, ki se zgodi samo enkrat),
- končaj z merjenjem časa.

V programu bomo uporabili funkcije srand(), rand() in clock() iz knjižnic stdlib.h in time.h. S funkcijo srand() bomo inicializirali generator naključnih števil, s funkcijo rand() bomo generirali naključna števila in s funkcijo clock() merili čas, ki ga je uporabnik porabil za reševanje.

- prični z merjenjem časa

Uporabili bomo funkcijo clock(), ki vrne število urnih ciklov od začetka izvajanja programa. Tip vrnjene vrednosti je clock_t.

```
cas1 = clock();
```

- naključno generiraj dve celi števili med 1 in 10

Najprej izvedemo inicializacijo generatorja naključnih števil (enako kor v prejšnjem primeru):

```
srand(time(NULL));
```

Nato uporabimo funkcijo rand(), ki vrne naključno celo število iz intervala [0 - RAND_MAX]. Ker pa želimo generirati števila na intervalu [1 - 10], uporabimo operator %:

```
stevilo1 = rand() % 10 + 1;
stevilo2 = rand() % 10 + 1;
```

Operacija izračuna ostanek pri deljenju naključnega števila z 10 in dobljeni vrednosti prišteje 1. Tako dobimo interval [1 - 10].

- uporabi ju v matematičnem izrazu

```
printf("%d * %d = ", stevilo1, stevilo2);
```

- preberi uporabnikov odgovor

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

- če je odgovor pravilen, povečaj števec pravilnih odgovorov

```
if(odgovor == stevilo1*stevilo2)
      stevec++;
```

- ponovi postopek 10-krat (razen začetka merjenja časa, ki se zgodi samo enkrat)

Uporabimo for zanko.

- končaj z merjenjem časa

```
cas2 = clock();
```

Na koncu še izračunamo razliko med cas2 in cas1 ter vrednost izpišemo.
Primer rešitve:

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

int main(void)
{
      clock_t cas1, cas2;
      int stevilo1, stevilo2, odgovor, stevec, i;
      stevec = 0;
      cas1 = clock();
      srand(time(NULL));
      for(i=0; i<10; i++)
      {
            stevilo1 = rand() % 10 + 1;
            stevilo2 = rand() % 10 + 1;
            printf("%d * %d = ", stevilo1, stevilo2);
            scanf("%d", &odgovor);
            if(odgovor == stevilo1*stevilo2)
                  stevec++;
      }
      cas2 = clock();
      printf("Pravilo si odgovoril %d-krat.\n", stevec);
      printf("Porabil si: %.3lf sekund\n",
```

```
                        (double)(cas2-cas1) / CLOCKS_PER_SEC);
      return 0;
}
```

5.2 Povzetek

Predprocesorski ukazi:

- #include <>

Funkcije (knjižnice):

- printf (iz stdio.h), scanf (iz stdio.h), srand (stdlib.h), rand (stdlib.h)

Ključne besede:

- int, void, main, return, char, float, double, short, long, sizeof, signed, unsigned, const, volatile, register, auto, static, typedef, if, else, switch, case, break, default, for, while, do, continue

Operatorji:

- sizeof,
- aritmetični (+, -, *, /, %, ++ ,--)
- relacijski operatorji (>, >=, <, <=, ==, !=)
- logični opreatorji (&&, ||, !)
- bitni operatorji (&, |, ^, ~, <<, >>)

Razno:

- prevajanje programa iz komandne vrstice
- prevajanje z uporabo IDE
- najava in inicializacija spremenljivk
- kodiranje celoštevilskih in float tipov
- aritmetične operacije
- lokalne/globalne spremenljivke
- sklad (stack), kopica (heap)
- komentarji // in /* */
- odločitvene in izbirne strukture
- ponavljalne strukture (zanke)
- relacijske in logične operacije
- bitne operacije
- načrtovanje in pisanje algoritmov

5.3 Naloge

Naloga 1
Napišite program, ki bo simuliral metanje kocke za igro Človek ne jezi se. Izvedite veliko število ponovitev in na koncu izpišite, kolikokrat se je posamezna številka ponovila.

Naloga 2
Poiščite in izpišite vse pitagorejske trojice za vrednosti števil a, b, in c od 1 do 1000. Pitagorejsko trojico sestavljajo tri cela števila (a, b, c) za katera velja: a^2 + b^2 = c^2. Pitagorejske trojice tudi preštejte in izmerite čas, ki je potreben za iskanje.

Dodatna naloga
Določite število vseh različnih pitagorejskih trojic od 1 do 1000. Pri tem upoštevajte, da sta pitagorejski trojici 3^2 + 4^2 = 5^2 in 4^2 + 3^2 = 5^2 enaki.

6 Polja in kazalci

Vsebina:
- podatkovna polja,
- kazalci,
- povzetek,
- naloge.

6.1 Podatkovna polja

Podatkovna polja združujejo podatke istega tipa v zaporednih lokacijah v spominu tako, da lahko do njih dostopamo prek skupnega imena.

Najava podatkovnega polja ima naslednjo obliko:

```
tip ime_polja[velikost];
```

Npr.:

```
double tocka[3];
```

Polje tocka je sestavljeno iz treh double vrednosti, ki predstavljajo npr. x, y in z koordinate točke v prostoru. Do posameznih elementov polja dostopamo prek imena polja in indeksom na naslednje načine:

```
tocka[0] = 3.3;
y_koordinata = tocka[1];
…
```

C ne preverja, če programer preseže najavljeno velikost polja, zato je potrebno biti pri tem zelo previden. Poglejmo si to na obravnavanem primeru. Povsem veljavna je naslednja vrstica:

```
tocka[4] = 5.0;
```

Pri prevajanju prevajalnik ne bo javil napake. Pri izvajanju vrstice, pa lahko pride do prepisa druge vrednosti, zaradi česar bo delovanje programa napačno, ali pa do preseganja spominskega naslova, ki je na voljo programu, zaradi česar se bo program sesul.

Če si predstavljamo zapis podatkovnega polja z vrednostmi (3.3, 4.0, 5.0) v RAM-u, izgleda nekako tako:

```
Naslovi:          … 0x0F01 0x0F02 0x0F03 0x0F04 0x0F05 0x0F06 …
Vrednosti:        …   ?      3.3    4.0    5.0    ?      ?    …
```

Vrednosti so shranjene v zaporednih spominskih lokacijah. Lokacija 0x0F02 ima ime tocka[0], 0x0F03 tocka[1] itn. Z zapisom na lokacijo tocka[4] zapisujemo na lokacijo 0x0F05, ki ni več del polja. Na tistem delu spomina je lahko zapisana vrednost kake druge spremenljivke, ki jo z zapisom prepišemo.

Podatkovna polja lahko ob najavi tudi incializiramo:

```
double tocka[] = {3.3, 4.0, 5.0};
```

, kar je enako kot:

```
double tocka[3] = {3.3, 4.0, 5.0};
```

Večdimenzionalna polja najavimo z:

double matrika[3][3];

Ob najavi jih lahko inicializiramo eno od naslednjih možnosti:

```
double matrika[][] = {1,0,0,
                      0,1,0,
                      0,0,1};
double matrika[3][3] = {1,0,0,
                        0,1,0,
                        0,0,1};
double matrika[][] = {{1,0,0},
                      {0,1,0},
                      {0,0,1}};
double matrika[3][3] = {{1,0,0},
                        {0,1,0},
                        {0,0,1}};
```

Seveda je možno vse izmed zgornjih inicializacij zapisati v eno vrstico (spomnite se, da C ni občutljiv na znak za novo vrstico - vsak program bi lahko zapisali v eno vrstico, pa bi vseeno deloval, le pregleden ne bi bil).

Celotnega polja ni mogoče prirediti drugemu polju:

```
char a1[10], a2[10];
a1 = a2; // tako ne gre!!!
```

Razlog za to je, da so imena polj kazalci na njihove ničte elemente!

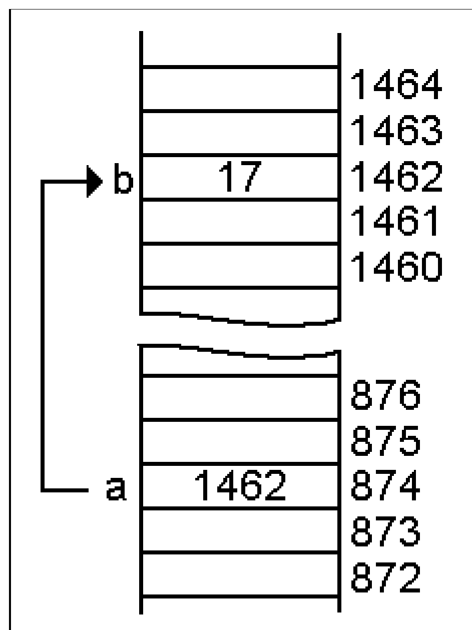
6.2 Kazalci

Kazalci so za začetnike najbolj zahteven koncept v jeziku C, zato bodite pri tem podpoglavju

posebej pozorni. Programerju omogočajo neposredno delo s spominom, kar je lahko prednost ali slabost. Prednost je popolna kontrola nad delovanjem programa, vključno s spominom. Slabost pa je, da če naredite eno samo napako, program ne dela, še več, napake vezane na kazalce so najtežje za odkriti, velikokrat pa celo zaidejo v produkcijsko kodo in povzročajo, da imajo programi t.i. memory leak-e. To pomeni, da program porablja čez čas vedno več spomina. Tak program moramo vsake toliko zapreti in ponovno odpreti, sicer porabi ves RAM. Moderni programski jeziki se zato odmikajo od tega koncepta, s čimer pa pride, na splošno, do malenkostno počasnejšega izvajanja kode. Definicija kazalca je preprosta:

Kazalec je spremenljivka, katere vrednost je spominska lokacija druge spremenljivke.

Če spremenljivka a vsebuje spominsko lokacijo spremenljivke b pravimo, da a kaže na b. Poglejmo na primeru.



Slika 6-1: Kazalci, primer.

Spremenljivka a se nahaja na spominski lokaciji 874, spremenljivka b pa na lokaciji 1462. Vrednost spremenljivke a je 1462, kar je spominska lokacija spermenljivke b. Vrednost spremenljivke b je 17.

Zaradi pomembnosti in uporabnosti v C so spremenljivke tipa kazalec pri najavi označene z *. Poleg tega sta na kazalce vezana še dva simbola: * (povsod, kjer ni uporabljena v najavi) in &:

* - vrednost, na katero kaže
& - naslov od

Če zgornji primer zapišemo v kodi:

```
int *a, b;      // najava kazalca a - simbol * - in spremenljivke b
b = 17;         // b je 17
a = &b;         // a je naslov od b
```

Če želimo prek a-ja, to je kazalca na b, dostopati do b-ja, to naredimo z naslednjo vrstico:

```
*a = 18;        // vrednost, nakatero kaže a je 18
```

S tem se vrednost b-ja spremeni iz 17 na 18!

Pomembno je naslednje:

- tip kazalca mora biti enak tipu spremenljivke, na katero kaže

Vsi kazalci porabijo enako spominskega prostora. Razlika med int *p in double *q je v tem, da se bo prvi pri inkrementu (p++) povečal za 4 (byte), drugi (q++) pa za 8. Pri različnih tipih kazalcev je različna le aritmetika z njimi!

- kazalci so tesno povezani s podatkovnimi polji

Ime polja je kazalec na njegov ničti element!

To pomeni, da sta naslednja zapisa povsem ekvivalentna:

tocka[0]   je povsem isto kot   *tocka

*tocka beremo kot vrednost, na katero kaže tocka, ki je kazalec na ničti element polja. *tocka je torej vrednost ničtega elementa polja oz. tocka[0]. Enako velja, da je tocka[1] enako *(tocka+1), tocka[2] enako *(tocka+2) itn. Od tu izhaja, da je pomembno, kakšen je tip kazalca, saj se tocka+1 v primeru tipa char poveča za 1 byte, v primeru int pa za 4.

Kazalci se v C največkrat uporabljajo za naslednje:

- s podatkovnimi polji
- s funkcijami, kadar so njihovi argumenti polja
- če želimo, da funkcija spreminja podatek, ne pa kopije podatka
- kadar želimo, da funkcija “vrne” več kot en podatek
- s pretočnimi kanali (streami)
- z datotekami
- s strukturami
- …

Poglejmo si uporabo kazalcev za prenos in vračanje podatkovnih polj. Kadar želimo v funkcijo prenesti podatkovno polje, to naredimo tako, da kot njen argument podamo:

`tip ime_polja[]`   ali pa   `tip *ime_polja`

Če bi želeli narediti funkcijo z izpis vrednost celoštevilskega polja, bi to naredili npr. z

naslednjo kodo:

```
void printa(int array[], int size)
{
    int i;
    for (i = 0; i < size; i++)
    {
        printf("%d ", array[i]);
    }
    return;
}
```

In jo uporabili v funkciji main na naslednji način:

```
int a[5] = {0,1,2,3,4};
printa(a, 5);
```

Pri vračanju je situacija drugačna. Funkcije ne morejo neposredno vračati polj. To rešimo tako, da vrnemo kazalec na njegov ničti element. Vseeno pa je potrebno biti pri vračanju polj iz funkcij posebno pazljiv. Vzemimo naslednji izsek kode, funkcijo return_array, ki vrača kazalec tipa int:

```
int *return_array(void)
{
    int array[3] = {1,2,3};
    return array;
}
```

Kljub temu, da koda na prvi pogled izgleda v redu, je izpis pri uporabi funkcije v naslednjem programu nepričakovan:

```
int main(void)
{
    int *b;
    printa(return_array(),3);
    return 0;
}
```

Razlog za to je, da funkcija return_array vrne array, to je kazalec na lokacijo array[0]. Ker je array lokalna spremenljivka, se po zaključku funkcije izbriše iz spomina (za časa izvajanja funkcije obstaja na skladu). To pomeni, da bo izpis zgornjega programa nedefiniran.

Kadar želimo, da funkcija vrača polje podatkov, moramo torej lokacijo polja podatki kot argument funkcije!

Z ničlo zaključena znakovna polja - besede

C nima posebnega podatkovnega tipa za besede (kot je npr. string v drugih programskih jezikih). Namesto tega so besede enostavno polja znakov, zaključena z znakom \0, t.j. ASCII kodo 0. To nam omogoča, da vemo, kje se polje zaključi. Zaradi pomembnosti besed jim je namenjena celotna standardna knjižnica <string.h>, ki jo bomo obravnavali kasneje.

Beseda “Mehatronika” ima v spominu naslednji zapis:

```
M | e | h | a | t | r | o | n | i | k | a | \0
```

Seveda so namesto črk v spominu zapisane njihove ASCII kode. Kadarkoli uporabimo dva narekovaja: “” ima zapis v spominu končni znak \0. To pomeni, da moramo definirati za en znak večjo spremenljivko, kot je število znakov.

`char str[12] = “Mehatronika”;` in ne `char str[11] = “Mehatronika”`

Ker končna ničla omejuje besedo, jo lahko uporabimo za izračun dolžine besede. Naslednja koda je elegantna, a zahtevna za razumevanje:

```
strcpy (char * to, char * from)
{
      while(*from)
      {
      *to++ = *from++;
      }
      *to = '\0';
}
```

Kazalci na funkcije

Poleg kazalcev na spremenljivke obstajajo tudi kazalci na funkcije. Vzemimo primer naslednjega prototipa funkcije:

```
int func(int a, float b);
```

Kazalec na funkcijo najavimo z:

```
int (*func)(int a, float b);
```

Pozor, oklepaji morajo biti prisotni, saj je naslednje funkcija, ki vrača kazalec, in ne funkcijski kazalec:

```
int *func(int a, float b);
```

Primer uporabe funkcijskega kazalca je predstavljen v naslednji kodi:

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

void func(int);

int main(void)
{
    void (*fp)(int);

    fp = func;

    (*fp)(1);
    fp(2);

    return 0;
}

void func(int arg)
{
    printf("%d\n", arg);
    return;
}
```

6.3 Povzetek

Predprocesorski ukazi:

- #include <>

Funkcije (knjižnice):

- printf (iz stdio.h), scanf (iz stdio.h), srand (stdlib.h), rand (stdlib.h), fflush (stdio.h)

Ključne besede:

- int, void, main, return, char, float, double, short, long, sizeof, signed, unsigned, const, volatile, register, auto, static, typedef, if, else, switch, case, break, default, for, while, do, continue

Operatorji:

- sizeof,
- aritmetični (+, -, *, /, %, ++, --)
- relacijski operatorji (>, >=, <, <=, ==, !=)
- logični opreatorji (&&, ||, !)
- bitni operatorji (&, |, ^, ~, <<, >>)
- za polja in kazalce ([ ], *, &)

Razno:

- prevajanje programa iz komandne vrstice
- prevajanje z uporabo IDE
- najava in inicializacija spremenljivk
- kodiranje celoštevilskih in float tipov
- aritmetične operacije
- lokalne/globalne spremenljivke
- sklad (stack), kopica (heap)
- komentarji // in /* */
- odločitvene in izbirne strukture
- ponavljalne strukture (zanke)
- relacijske in logične operacije
- bitne operacije
- načrtovanje in pisanje algoritmov
- polja, kazalci, kazalci na funkcije

6.4 Naloge

Naloga 1:
Napišite in uporabite funkcijo, ki sortira polje celih števil po velikosti. Uporabite algoritem Bubble sort.

Naloga 2:
Napišite in uporabite funkcijo, ki sortira polje celih števil po velikosti. Uporabite algoritem Bogosort.

Dodatna naloga:
Z algoritmom Bogosort sortirajte naraščajoče velika naključno generirana polja celih števil (3 elementi, 4 elementi, …). Kako narašča povprečni čas sortiranja v odvisnosti od velikosti polja?

7 Znakovna polja

Vsebina:

- ponovitev,
- znakovna polja,
- delo z datotekami,
- povzetek,
- naloge.

7.1 Delo z znakovnimi polji

C nima posebnega podatkovnega tipa za besede. Le-te shranimo v polje znakov (char[ ]), ki mora biti zaključeno z znakom \0, t.j. ASCII kodo 0. Ker se pri programiranju z besedami pogosto srečujemo, jim je v C-ju namenjena posebna standardna knjižnica <string.h>.
Pred funkcijami iz knjižnice <string.h> pa si poglejmo, kako lahko uporabnik programu posreduje nek zankovni niz (besedo). Ena izmed možnosti je uporaba že poznane funkcije scanf():

```
char beseda[101];
printf("Vnesi besedo, dolgo do 100 znakov:\n");
scanf("%s", beseda);
printf("Vnesel si: %s", beseda);
```

Funkcija scanf() bo v znakovno polje "beseda" zapisala vse preko tipkovnice vnešene znake do prvega presledka (space), tabulatorja (tab) ali nove vrstice (newline). V zgornjem primeru opazimo, da je drugi argument funkcije scanf() `beseda` in ne `&beseda`, kot smo bili navajani pri spremenljivkah int, float, itd. Razlog je v tem, da ime polja prestavlja kazalec na njegov prvi element. Namesto tega, bi lahko pisali tudi `&beseda[0]`.

Če želimo prebrati tudi presledke, lahko uporabimo funkcijo gets() iz knjižnice <stdio.h>. Funkcija prebere znakovni niz iz standardnega inputa (tipkovnice) do prvega znaka za novo vrstico (newline):

```
char beseda[101];
gets(beseda);
printf(beseda);
```

Naslednja, pogosto uporabljana funkcija je strcpy(), ki kopira vsebino enega znakovnega niza v drugega:

```
char beseda[100];
strcpy(beseda, "Pozdravljeni!");
printf(beseda);
```

Mnogokrat želimo na konec ene besede dodati drugo besedo, kar lahko storimo s funkcijo strcat(). Primer ustvarjanja stavka s funkcijama strcpy() in strcat():

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

int main()
{
      char stavek[100];
      char ime[50];

      printf("Vpisi svoje ime:\n");
      gets(ime);
      strcpy(stavek, "Tvoje ime je ");
      strcat(stavek, ime);
      strcat(stavek, ".");

      printf(stavek);
      return 0;
}
```

V zgornjem primeru s funkcijo gets() preberemo vnos iz tipkovnice in ga zapišemo v polje ime. Nato s funkcijo strcpy() v polje stavek zapišemo “Tvoje ime je “. V nadaljevanju s funkcijo strcat() na konec polja (označen je z znakom \0) prepišemo vsebino polja ime. Na koncu stavek zaključimo s piko in ga izpišemo.

Funkcija strcmp() primerja dve besedi (znakovna niza) in vrne vrednosti 0, če sta besedi enaki. Če sta besedi različni, funkcija vrne manj kot 0, če se prva beseda po abecednem redu pojavi pred drugo, in več kot 0 v nasprotnem primeru. Primer:

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

int main()
{
      char geslo[50] = {"marelicna marmelada"};
      char vnos[100];
      printf("Vnesi gelso:\n");
      gets(vnos);
      if(strcmp(vnos, geslo) == 0)
            printf("Geslo potrjeno.");
      else
            printf("Geslo zavrnjeno.");
      return 0;
}
```

Funkcija strlen() vrne dolžino besede v obliki tipa size_t, ki smo ga spoznali že pri operatorju sizeof. Primer:

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

int main()
{
	char stavek[256];
	printf("Vnesi stavek:\n");
	gets(stavek);
	printf("Vneseni stavek ima %d znakov.\n", strlen(stavek));
	return 0;
}
```

V nekaterih primerih želimo namesto izpisa na zaslon (funkcija printf()) izvesti formatiran izpis v znakovno polje. To nam omogoča funkcija sprintf() iz knjižnice <stdio.h>. Primer:

```
char stavek[256];
double hitrost = 22.4;
sprintf(stavek, "Hitrost je %.2lf", hitrost);
printf(stavek);
```

7.2 Delo z datotekami

V realnih situacijah smo mnogokrat soočeni z veliko količino podatkov, ki bi jih bilo zamudno vnašati ali prebirati preko terminala. Poleg tega smo do sedaj obravnavali zgolj primere, kjer so bili po zaključku izvajanja programa vsi podatki izgubljeni. Da zaobidemo te omejitve je v C-ju mogoče delo z datotekami, ki shranijo podatke na disku računalnika.
Preden pričnemo z datotekami je pomembno razumevanje koncepta podatkovnih tokov (angl. stream). V C-ju stream predstavlja skupen logični vmesnik do različnih naprav, ki sestavljajo računalnik (trdi disk, zaslon, tipkovnica, …). Čeprav so lastnosti naprav različne, so streami enaki, kar je ugodno za programerja, saj na enak način dostopa do različnih naprav. V C-ju se iz zgodovinskih razlogov struktura, ki predstavlja stream, imenuje FILE.

Za delo z datotekami najprej ustvarimo kazalec na podatkovni tip FILE, ki je definiran v <stdio.h>. V osnovi je FILE struktura, ki vsebuje različne informacije o datoteki, kot na primer velikost, lokacija, itd. Strukture bodo podrobneje obravnavane v naslednji vaji. Nadalje kličemo funkcijo fopen(), ki vrne kazalec na novo ustvarjeno strukturo FILE:

```
FILE *datoteka;
datoteka = fopen("ime_datoteke","nacin");
```

Če funkcija ni mogla odpreti želene datoteke, vrne NULL. Zato je pomembno, da vedno preverimo, ali je bila datoteka uspešno odprta:

```
FILE *datoteka;
datoteka = fopen("ime_datoteke","nacin");
if(datoteka == NULL)
{
	printf("Napaka pri odpiranju datoteke.\n");
}
```

Enak učinek dosežemo s spodnjo, bolj zgoščeno kodo.

```
FILE *datoteka;
if((datoteka = fopen("ime_datoteke","nacin")) == NULL)
{
	printf("Napaka pri odpiranju datoteke.\n");
}
```

Pri tem ima znakovno polje “nacin” lahko naslednje vrednosti:

nacin	**pomen**
r	odpri datoteko za branje
w	ustvari datoteko za pisanje
a	dodaj v tekstovno datoteko
rb	odpri binarno datoteko za branje
wb	ustvari binarno datoteko za pisanje
ab	dodaj v binarno datoteko
r+	odpri tekstovno datoteko za branje / pisanje
w+	ustvari praznno tekstovno datoteko za branje / pisanje
a+	dodaj ali ustvari tekstovno datoteko za branje / pisanje
r+b	odpri binarno datoteko za branje / pisanje
w+b	ustvari prazno binarno datoteko za branje / pisanje
a+b	dodaj ali ustvari binarno datoteko za branje / pisanje

Primer ustvarjanja in pisanja v tekstovno datoteko:

```
#include <stdio.h>

int main()
{
	FILE *datoteka;
	datoteka = fopen("dat.txt", "w");
	fprintf(datoteka, "Vsebina tekstovne datoteke.");
	fclose(datoteka);
	return 0;
}
```

V zgornjem programu smo uporabili funckijo fprintf(), ki jo uporabljamo na enak način kot

printf(), s to razliko, da je prvi argument funkcije fprintf() kazalec na FILE strukturo. Za zapiranje datoteke smo uporabili funkcijo fclose().

Ravno tako kot funkcija printf() ima funkcija scanf() različico za branje podatkov iz datotek. Njena uporaba je prikazana v naslednjem primeru:

```
#include <stdio.h>

int main(void)
{
	FILE *datoteka;
	int i;
	float f;
	char beseda[100];

	datoteka = fopen("c:\\dat.txt", "w+");//odpiranje datoteke
	//formatiran izpis v datoteko
	fprintf(datoteka, "%d %f %s", 2, 3.14, "besedilo");
	rewind(datoteka);
	//formatirano branje iz datoteke
	fscanf(datoteka, "%d %f %s", &i, &f, beseda);
	printf("Vsebina datoteke: %d %f %s\n", i, f, beseda);
	fclose(datoteka);                    //zapiranje datoteke
	return 0;
}
```

Program ustvari datoteko c:\dat.txt za branje in pisanje v tekstovnem načinu ter s funkcijo fprintf() v datoteko zapiše različne podatkovne tipe. V nadaljevanju uporabi funkcijo rewind() s čimer nastavi indikator trenutnega položaja v datoteki nazaj na začetek datoteke. Nato z uporabo funkcije fscanf() prebere vsebino datoteke in vrednosti zapiše v spremenljivke i, f in beseda ter jih izpiše na zaslonu.

Primer:
Program za branje tabele iz datoteke podatki.txt:

```
#include <stdio.h>

int main()
{
	FILE *datoteka;
	int i;
	double a, b;
	//odpiranje datoteke za branje in preverjanje uspesnosti
	if((datoteka = fopen("podatki.txt", "r")) != NULL)
	{
		//branje vrstic do konca datoteke
```

```
            while(fscanf(datoteka, "%d %lf %lf", &i, &b, &c) != EOF)
                  printf("%lf\t%lf\t%lf\n", a, b, c);
            fclose(datoteka);
      }
      else
            printf("Ne morem odpreti datoteke.\n");
      return 0;
}
```

Program odpre datoteko podatki.txt, ki mora imeti naslednjo obliko:

```
1     1.45  12.34
2     2.43  23.23
3     1.34  95.33
```

Podatke iz datoteke prebere in jih izpiše na zaslonu. Branje do konca datoteke dosežemo z while zanko, ki preverja vrnjeno vrednost funkcije fscanf(). Ko je vrnjena vrednost enaka EOF (anlg. end of file), se zanka zaključi.

Primer:
Napišite program, ki bo omogočal kodiranje oz. dekodiranje besedila iz datoteke z uporabo bitne operacije xor.

Postopek reševanja (kodiranje):

- preberi vsebino datoteke in jo zapiši v znakovno polje,
- izvedi bitni xor na vsakemu znaku v polju,
- rezultat zapiši v novo datoteko.

Sedaj se lotimo reševanja primera po korakih. Prva dva koraka zaradi enostavnosti združimo.

- preberi vsebino datoteke in jo zapiši v znakovno polje
- izvedi bitni xor na vsakemu znaku v polju

```
FILE *datoteka;
char koda[10001];
char kljuc = 25;
int i;
datoteka = fopen("besedilo.txt","r"); //odpri datoteko
if(datoteka != NULL)
{
      fgets(koda, 10000, datoteka);    //preberi vrstico besedila
      for(i=0; i < strlen(koda); i++) //zakodiraj vrstico
      {
            koda[i] = koda[i] ^ kljuc;
      }
      fclose(datoteka);                //zapri datoteko
```

```
}
```

Zgornja koda najprej ustvari kazalec na strukturo datoteka in nato ustvari znakovno polje s kapaciteto 10000 znakov. Nato s funkcijo fopen datoteko odpremo za branje v tekstovnem načinu. Preverimo ali je bila datoteka uspešno odprta in nato s funkcijo fgets preberemo znakovni niz iz datoteke in ga zapišemo v polje koda. Funkcija fgets() prebere znakovni niz izbrane dolžine iz datoteke. Če prej naleti na znak za novo vrstico (newline) branje zaključi. V nadaljevanju na vsakem prebranem znaku izvedemo bitni xor s spremenljivko kljuc ter nato datoteko zapremo s funkcijo fclose().

- rezultat zapiši v novo datoteko

```
FILE *izh_datoteka;
izh_datoteka = fopen("zakodirano.txt","w");
fputs(koda, izh_datoteka);
fclose(izh_datoteka);
```

Zapisana koda ni najprimernejša, saj predpostavlja, da je zakodirana datoteka tudi tekstovna, kar pa ni nujno, saj pri izvajanju operacije xor lahko dobimo vrednosti, ki jih napačno interpretiramo. Če tako dobimo vrednost 0, bo funkcija fputs() to interpretirala kot konec znakovnega polja in od tu naprej bodo vsi podatki izgubljeni. Zato bomo uporabili odpiranje datoteke v binarnem načinu in funkcijo fwrite() za zapis n znakov iz polja koda v izhodno datoteko.

```
FILE *izh_datoteka;
izh_datoteka = fopen("zakodirano.txt","wb");
fwrite(koda, 1, n, izh_datoteka);
fclose(izh_datoteka);
```

Postopek odkodiranja je identičen kodiranju, ker smo uporabili bitni xor. Uporabimo lahko celo enak program, le da zamenjamo imena datotek: besedilo.txt nadomestimo z zakodirano.txt in zakodirano.txt z odkodirano.txt. Ker pa je takšno spreminjanje imen datotek v izvorni kodi časovno potratno, bomo napisali program, ki uporablja argumente komandne vrstice za določanje imen vhodne in izhodne datoteke.
Po standaru je veljavna tudi naslednja oblika funkcije main:

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

pri čemer argument argc (angl. argument count) predstavlja število argumentov v komandni vrstici in argv (angl. argument vector) polje kazalcev na znakovne nize (znakovni nizi predstavljajo argumente komandne vrstice).
Uporabo argumentov v komandni vrstici prikazuje naslednji primer:

```
#include <stdio.h>

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

```
{
      int i;
      for(i = 0; i < argc; i++)
            printf("arg %d: %s\n", i, argv[i]);
      return 0;
}
```

Po prevajanju programa dobimo datoteko npr. program.exe. Če to datoteko zaženemo iz komandne vrstice in poleg njenega imena zapišemo še nekaj znakovnih nizov:

```
program.exe beseda1 beseda2
```

dobimo naslednji izpis:

```
arg 0: program.exe
arg 1: beseda1
arg 2: beseda2
```

Na ta način lahko pri zagonu programa iz komandne vrstice preberemo znakovna polja (besede), ki jih uporabimo v programu.

Smiselno bi bilo uporabiti naslednjo obliko klica programa iz komandne vrstice:

```
program.exe  vhodna_datoteka  -operacija  izhodna_datoteka
```

Podobno obliko smo srečali pri prevajanju iz komandne vrstice s prevajalnikom GCC. Program bo torej sprejel 3 dodatne argumente iz komandne vrstice, ki bodo določili ime vhodne datoteke, tip operacije (kodiranje ali odkodiranje) ter ime izhodne datoteke.
Poskusimo to zapisati:

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

main(int argc, char *argv[])
{
      int i;
      char ime_vh_dat[100], ime_izh_dat[100], operacija[100];
      if(argc < 4)
            printf("Vnesel si premalo argumentov.\n");
      else
      {
            //kopiranje argumentov v znakovna polja
            strcpy(ime_vh_dat, argv[1]);
            strcpy(operacija, argv[2]);
            strcpy(ime_izh_dat, argv[3]);
```

```
            if(strcmp(operacija, "-k") == 0)
                  printf("Izvedli bomo kodiranje\n");
            else if(strcmp(operacija, "-d") == 0)
                  printf("Izvedli bomo dekodiranje.\n");
            else
                  printf("Neznana operacija.\n");
      }
      return 0;
}
```

Če zgornji program prevedemo v codec.exe in ga kličemo iz komandne vrstice z naslednjim ukazom:

```
codec.exe vhodna_dat.txt -k izhodna_dat.txt
```

Bo program izpisal:

```
Izvedli bomo kodiranje.
```

Sedaj pa združimo prikazane programe v skupen kodirni / dekodirni program. Pri tem bomo zahtevali vnos zgolj imena vhodne in izhodne datoteke. Vnos tipa operacije ne bo potreben, saj sta postopka kodiranja in dekodiranja enaka.

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

main(int argc, char *argv[])
{
      FILE *vh_datoteka, *izh_datoteka;
      char ime_vh_dat[100], ime_izh_dat[100];
      unsigned char koda[10000], kljuc;
      int i, n;
      if(argc < 3)
      {
            printf("Vnesel si premalo argumentov.\n");
            return 0;  //zakljucimo funkcijo main in s tem program
      }
      //kopiranje argumentov v znakovna polja
      strcpy(ime_vh_dat, argv[1]);
      strcpy(ime_izh_dat, argv[2]);

      vh_datoteka = fopen(ime_vh_dat, "rb");//odpiranje datotek
      izh_datoteka = fopen(ime_izh_dat, "wb");    //v binarnem nacinu
      if(vh_datoteka != NULL && izh_datoteka != NULL)  //preverjanje
      {
            printf("Vnesi kljuc:\n");              //vnos kljuca
```

```
            scanf("%d", &kljuc);
            n = fread(koda, 1, 10000, vh_datoteka); //branje do max.
                                                    //10000 bytov
            printf("Stevilo prebranih znakov je %d.\n", n);
            for(i=0; i < n; i++)
            {
                  koda[i] = koda[i] ^ kljuc;     //kodiranje
            }
            fclose(vh_datoteka);               //zapiranje vh_datoteke
            fwrite(koda, 1, n, izh_datoteka); //pisanje v izh_datot.
            fclose(izh_datoteka);             //zapiranje izh_datoteke
      }
      else
            printf("Ne morem odpreti datoteke.\n");
      return 0;
}
```

Zgornji program preveri število argumentov v komandni vrstici in jih zapiše v znakovna polja ime_vh_dat in ime_izh_dat.. Nato odpre obe datoteki, od uporabnika zahteva vnos ključa in prebere največ 10000 znakov iz vh_datoteke. Dejansko število prebranih znakov izpiše na zaslonu. Nato vsak znak zakodira z btinim xor, ter rezultat zapiše v izh_datoteko. Za branje in pisanje v datoteko smo uporabili funkciji fread() in fwrite(), ki sta primerni za delo v binarnem načinu. Kadar imamo opraviti z znakovnimi polji (besedami), pa lahko uporabimo tudi fgets() in fputs().

7.3 Povzetek

Predprocesorski ukazi:

- #include <>

Funkcije (knjižnice):

- printf (iz stdio.h), scanf (iz stdio.h), srand (stdlib.h), rand (stdlib.h), fflush (stdio.h), strcpy, strcat, strcmp, strlen (string.h), sprintf, gets, fopen, fclose, fprintf, fscanf, fgets, fputs (stdio.h)

Ključne besede:

- int, void, main, return, char, float, double, short, long, sizeof, signed, unsigned, const, volatile, register, auto, static, typedef, if, else, switch, case, break, default, for, while, do, continue

Operatorji:

- sizeof,
- aritmetični (+, -, *, /, %, ++, --)
- relacijski operatorji (>, >=, <, <=, ==, !=)
- logični opreatorji (&&, ||, !)
- bitni operatorji (&, |, ^, ~, <<, >>)
- za polja in kazalce ([ ], *, &)

Razno:

- prevajanje programa iz komandne vrstice
- prevajanje z uporabo IDE
- najava in inicializacija spremenljivk
- kodiranje celoštevilskih in float tipov
- aritmetične operacije
- lokalne/globalne spremenljivke
- sklad (stack), kopica (heap)
- komentarji // in /* */
- odločitvene in izbirne strukture
- ponavljalne strukture (zanke)
- relacijske in logične operacije
- bitne operacije
- načrtovanje in pisanje algoritmov
- polja, kazalci, kazalci na funkcije
- delo z znakovnimi polji
- podakovni tokovi (streami)
- delo z datotekami

7.4 Naloge

Naloga 1:

Napišite program za igranje vislic. Program iz tekstovne datoteke naključno izbere besedo ter jo prikaže v obliki:

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

Nato uporabnik vnese znak. Če se znak pojavi v besedi, ga program izpiše na ustreznem mestu.

```
...A......A
```

Na voljo je omejeno število poskusov.
Ne pozabite na klic funkcije fflush(stdin) pred vsakim klicem funkcije scanf(). Po vnosu znaka kličite funkcijo system(“cls”), ki počisti zaslon (clear screen) in na novo izpiše delno razkrito besedo:

```
M..A......A
```

Pomoč (ki je ni potrebno upoštevati):
Ustvarite datoteko datoteka.txt z npr. 10 različnimi besedami, pri čemer je vsaka beseda v svoji vrstici. Datoteko odprite za branje. Nato v začetku programa ustvarite polje znakov, ki bo shranjevalo besedo za ugibanje:

```
char beseda[25];
FILE *datoteka;
```

Sedaj želimo prebrati naključno vrstico iz datoteke in jo prekopirati v polje beseda. Eden izmed možnih postopkov je naslednji:

- generiraj naključno število n od 0 do 9,
- kliči funkcijo fscanf() n-krat.

Pri vsakem klicu funkcije fscanf():

```
fscanf(datoteka, "%s", beseda);
```

se v polje beseda zapiše trenutna vrstica. Če klic ponovite n-krat, bo polje beseda vsebovalo n-to vrstico.

Sedaj je potrebno preveriti dolžino besede s funkcijo strlen() in izpisati potrebno število znakov ‘.’. Znake je smiselno shraniti v znakovno polje, ki ga bomo med ugibanjem spreminjali in izpisovali na zaslonu:

```
char skrita_beseda[25];
int j;
for(j=0; j < strlen(beseda); j++)
{
      skrita_beseda[j] = '.';
}
```

```
skrita_beseda[j] = 0;       //zaključimo znakovno polje z 0
```

Nato na zaslonu izpišite skrito_besedo in uporabniku sporočite naj začne z ugibanjem.
Ko uporabnik vnese znak, ga je potrebno primerjati z vsemi znaki v izžrebani besedi. To storimo s for zanko:

```
int i;
char c;
fflush(stdin);
scanf("%c", &c); //prebermo uporabnikov vnos
for(i=0; i < strlen(beseda); i++)
{
      if(c == beseda[i])
            skrita_beseda[i] = beseda[i];    //razkrijemo      uganjen
znak
}
```

Uporabnik je uganil besedo, ko sta polji `skrita_beseda` in `beseda` enaki, kar lahko ugotovimo s funkcijo strcmp().

Dodatna naloga:
Napišite program za kodiranje z xor, ki bo omogočal kodiranje z daljšim ključem (npr. 10 znakov).

8 Strukture, spomin in predprocesor

Vsebina

- strukture,
- delo s spominom,
- predprocesor,
- povzetek,
- naloge.

8.1 Strukture

Do sedaj so bili v programih uporabljani povečini le osnovni podatkovni tipi: void, char, int, float in double, ter polja osnovnih tipov. Zaradi svoje enostavnosti lahko s tem popišemo le enostavne pojme iz realnega sveta. Npr. hitrost kot double, ime osebe kot polje znakov, št. loterijske srečke kot int, … Osnovni podatkovni tipi omogočajo le poenostavljeno oz. nepopolno abstrakcijo realnega sveta.

Strukture so gradniki C programov, ki omogočajo vpeljavo lastnih podatkovnih tipov. Lastni podatkovni tipi so lahko sestavljeni iz osnovnih. Tako je možno definirati podatkovni tip za poljuben objekt iz realnega sveta, npr. za študenta, ki se udeležuje vaj iz C-ja:

```
struct student
{
      int vpisna;
      char ime[40];
      char priimek[40];
      int domaca_naloga[8];
      int kolokvij[3];
} student1;
```

S sintakso struct student definiramo nov podatkovni tip, katerega definicijo nato napišemo med { in }. Obravnavana struktura vsebuje int, dve polji char, z velikostjo 40 elementov, polje int velikosti 8 za domače naloge in polje int velikosti 3 za ocene kolokvijev. student1 v zgornjem primeru pomeni, da hkrati z definicijo novega podatkovnega tipa ustvarimo spremenljivko student1, ki je tega podatkovnega tipa, t.j. podatkovnega tipa struct student. Enako, kot je pri int a, a ime spremenljivke, je pri struct student student1, student1 ime spremenljivke. Definicija spremenljivke ob najavi novega podatkovnega tipa ni obvezna, v zgornjem primeru bi lahko student1 mirno izpustili!

Zgornjo definicijo lahko v programu postavimo ali kot globalno, izven vseh funkcij, ali kot lokalno, veljavno le znotraj funkcije.

Zaradi nerodne uporabe imen stuktur pogosto uporabimo typedef za preimenovanje novonastalega podatkovnega tipa.

```
typedef struct student student_t;
```

Za strukture, preimenovane s typedef, po konvenciji uporabljamo oznako ime_strukture_t.

Strukturni podatkovni tip najavimo v kodi povsem enako kot katerikoli drugi. Ob najavi ga lahko inicializiramo podobno kot podatkovna polja. Poglejmo si uporabo stukture na primeru:

```
int main(void)
{
    student_t student1;
    student_t student2 = {23000001, "Janez", "Novak",
      {1,1,1,1,1,0,1,1}, {10,9,10}};
    printf("vpisna1: %d, vpisna2: %d", student1.vpisna,
      student2.vpisna);
    return 0;
}
```

V zgornjem primeru sta najavljeni dve strukturi tipa student_t, student1 in student2. Struktura student2 je tudi inicializirana. S prinftf sta izpisani vrednosti vpisnih številk. Dostop do podatka znotraj stukture je dostopen prek pike '.'. V prvem primeru bo izpisana naključna vrednost, oz. vrednost na neinicializirani spominski lokaciji. Za student2 program izpiše '23000001'.

8.1.1 Uporaba struktur s kazalci

Poleg obravnavane uporabe je možno strukture uporabljati tudi s kazalci. Ta primer je pogosto uporabljan, saj omogoča, da v funkcijo prenesemo le kopijo kazalca namesto kopije celotne strukture in s tem prihranimo prostor na skladu. Poleg tega je tak način običajen, kadar dinamično alociramo spomin. Naslednji primer prikazuje najavo in incializacijo kazalca p na strukturo student s.

```
struct student s, *p;
p = &s;
p->vpisna = 23010001;
// kar je enako kot:
(*p).vpisna = 23010001;
// in enako kot:
s.vpisna = 23010001
```

Dostop do podatkov je možen na tri načine:

- neposredno z imenom strukture in piko		s.
- prek kazalca in pike, podobno kot pri poljih		(*p).
- prek kazalca in puščice, skrajšana sintaksa		p->

8.2 Delo s spominom

Da bi programi uporabili čim manj spomina, torej ravno prav spomina, ravno takrat, ko ga

potrebujejo, moramo uporabiti princip dinamične alokacije spomina. Tipičen primer je branje datoteke. Ker ne vemo, koliko znakov bo prebranih je smiselno najprej rezervirati majhen kos spomina in ga nato dinamično povečevati glede na potrebe.

Principi alokacije spomina v C-ju so naslednji:

- statična alokacija - za static in globalne spremenljivke, na začetku izvajanja programa, na kopico (heap), spomin ni nikoli sproščen,
- avtomatska alokacija - za lokalne spremenljivke, ob vstopu v blok kode, kjer je spremenljivka najavljena, na sklad (stack), spomin je sproščen ob izhodu iz bloka kode,
- dinamična alokacija - z uporabo funkcij knjižnice stdlib:
    - malloc,
    - calloc,
    - realloc,
    - free.

8.2.1 malloc

Funkcija malloc je osnovna funkcija za rezervacijo spomina. Argument funkcije je zahtevana količina spomina, funkcija pa vrne kazalec na rezervirani spomin. Podpis funkcije je naslednji:

void *malloc(size_t size);

Namesto da bi napisali:

```
int polje[10];
```

lahko spomin za polje rezerviramo s funkcijo malloc na naslednji način:

```
int *polje = malloc(10*sizeof(int));
```

malloc ne incializira spomina. To lahko naredimo npr. s funkcijo memset, ki ima naslednji podpis:

```
void * memset(void *ptr, int value, size_t num);
```

Pri tem je *ptr kazalec na spomin, ki ga želimo nastaviti, value vrednost (običajno 0), num pa velikost spomina, ki ga želimo nastaviti na vrednost.

malloc vrne NULL, kadar ni dovolj spomina na voljo. NULL je koda, rezervirana za takrat, kadar želimo povedati, da kazalec nikamor ne kaže.

8.2.2 calloc

calloc uporabljamo za rezervacijo in hkratno inicializacijo spomina. Podpis funkcije je:

```
void * calloc(size_t num, size_t size);
```

num pove koliko blokov spomina, size pa velikost posameznega bloka v bytih.

8.2.3 realloc

realloc uporabljamo za ponovno alokacijo spomina, npr. za povečanje ali zmanjšanje spomina, na voljo določenim podatkom. Podpis funkcije:

void * realloc (void * ptr, size_t size);

Velikost spominskega bloka, na katerega kaže ptr je spremenjena na size bytov, pri čemer se količina spomina v bloku poveča ali pomanjša.

Funkcija lahko premakne blok spomina na novo lokacijo, pri čemer je vrnjen kazalec nanjo. Vsebina bloka spomina je ohranjena do manjše izmed stare in nove velikosti, tudi če je blok premaknjen. Če je nova velikost večja, je vrednost na novo alociranih dodatnih spominskih prostorov nedefinirana.

V primeru, da je ptr enak NULL se funkcija obnaša enako kot malloc.

V primeru, da je size 0 je spomin, ki je bil prej alociran na lokaciji ptr sprosti, enako kot če bi klicali free. Vrnjen je NULL kazalec.

8.2.4 free

Spomin, rezerviran z malloc, calloc ali realloc mora biti na koncu sproščen z uporabo funkcije free, katere argument je kazalec na spomin, ki ga želimo sprostiti.

free mora biti klican enkrat za vsak malloc oz. calloc oz. realloc, kadar je slednji uporabljen kot malloc.

Klic funkcije free s kazalcem, ki ni bil inicializiran vodi v nedefinirano obnašanje ali sesutje programa.

8.2.5 Primer

Primer prikazuje, kako velikost polja povečujemo ob vnosu novih spremenljivk s strani uporabnika.

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

int main (void)
{
    int input,n;
    int count=0;
```

```
    int * numbers = NULL;
    int * more_numbers;

    do
    {
        printf ("Enter an integer value (0 to end): ");
        scanf ("%d", &input);
        count++;

        more_numbers = (int*) realloc (numbers, count * sizeof(int));

        if (more_numbers!=NULL)
        {
            numbers=more_numbers;
            numbers[count-1]=input;
        }
        else
        {
            free (numbers);
            puts ("Error (re)allocating memory");
            exit (1);
        }
    }
    while (input!=0);

    printf ("Numbers entered: ");
    for (n=0; n<count; n++) printf ("%d ",numbers[n]);
    free (numbers);

    return 0;
}
```

8.3 Predprocesor

Obravnavali bomo le najosnovnejša predprocesorska ukaza, #include in #define. Poleg teh obstajajo še ukazi #if, #ifdef, #ifndef, #else, #elif in #endif za pogojno prevajanje in ukaz #error za izpis napak na podatkovni tok za napake (angl. error stream).

Pri #include je lahko argument zaprt v dvojne narekovaje ali trikotne oklepaje:

```
#include “datoteka.h”
#include <datoteka.h>
```

V prvem primeru mora biti datoteka.h v isti mapi kot datoteka, ki jo z #include vključuje. S trikotnimi oklepaji označimo, da vključujemo standardno knjižnico oz. datoteko, do katere pot je specificirana globalno.

Poleg predprocesorskega ukaza #include, je pomemben še predprocesorski ukaz #define. Z #define pogosto definiramo konstante in t.i. makroje. Uporaba #define ima naslednjo obliko:

```
#define NOVO_IME staro_ime
```

npr.

```
#define PI 3.14159
```

Povsod, kjer bo v programu napisan simbol PI bo ta simbol nadomeščen s 3.14159.

#define je možno uporabiti tudi na podoben način kot funkcijo. Primer:

```
#define RADTODEG(x) ((x) * 57.29578)
```

Pri tem v oklepaje zapišemo parametre.

Poseben predprocesorski ukaz je #pragma, s katerim so označeni ukazi za specifičen prevajalnik.

8.4 Povzetek

Predprocesorski ukazi:

- #include <>, #include “”, #define

Funkcije (knjižnice):

- printf (iz stdio.h), scanf (iz stdio.h), srand (stdlib.h), rand (stdlib.h), fflush (stdio.h), strcpy, strcat, strcmp, strlen (string.h), sprintf, gets, fopen, fclose, fprintf, fscanf, fgets, fputs (stdio.h), malloc, calloc, realloc, free (stdlib.h), memset (string.h)

Ključne besede:

- int, void, main, return, char, float, double, short, long, sizeof, signed, unsigned, const, volatile, register, auto, static, typedef, if, else, switch, case, break, default, for, while, do, continue, struct

Operatorji:

- sizeof,
- aritmetični (+, -, *, /, %, ++, --)
- relacijski operatorji (>, >=, <, <=, ==, !=)
- logični opreatorji (&&, ||, !)
- bitni operatorji (&, |, ^, ~, <<, >>)
- za polja in kazalce ([ ], *, &)
- za strukture (., ->)

Razno:

- prevajanje programa iz komandne vrstice
- prevajanje z uporabo IDE
- najava in inicializacija spremenljivk
- kodiranje celoštevilskih in float tipov
- aritmetične operacije
- lokalne/globalne spremenljivke
- sklad (stack), kopica (heap)
- komentarji // in /* */
- odločitvene in izbirne strukture
- ponavljalne strukture (zanke)
- relacijske in logične operacije
- bitne operacije
- načrtovanje in pisanje algoritmov
- polja, kazalci, kazalci na funkcije
- delo z znakovnimi polji
- podakovni tokovi (streami)
- delo z datotekami
- strukture,

- kazalci na strukture,
- dinamična alokacija spomina,
- predprocesorski ukazi

8.5 Naloge

Naloga 1

Opiši, kaj dela spodnji izsek kode. Premisli, kaj predstavlja podatkovni tip list_el in kako je uporabljen. V kakšnih primerih (na kakšnem področju), bi lahko idejo, predstavljeno v spodnji kodi uporabili?

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

struct list_el {
   int val;
   struct list_el * next;
};

typedef struct list_el item;

void main() {
   item * curr, * head;
   int i;

   head = NULL;

   for(i=1;i<=10;i++) {
      curr = (item *)malloc(sizeof(item));
      curr->val = i;
      curr->next  = head;
      head = curr;
   }

   curr = head;

   while(curr) {
      printf("%d\n", curr->val);
      curr = curr->next ;
   }
}
```

9 Zaključek oz. kako naprej?

Vsebina:

- izpuščene tematike
- od C proti OOP,
- C na mikrokrmilnikih.

9.1 Izpuščene tematike

Izpuščena je obravnava unij (union) in bitnih polj (bitfield). Prav tako je narejen le osnovni pregled standardne knjižnice.

Prav tako je izpuščena uporaba zunanjih knjižnic za uporabniški vmesnik, grafiko, itd.

9.2 Od C proti OOP

C je t.i. proceduralen jezik. Pri proceduralnih jezikih je osnova pisanja programa zaporedje ukazov, ki jih mora program izvesti. Da bi zaporedja organizirali v zaključene celote je uveden koncept funkcij.

Popularni moderni programski jeziki, kot so npr. C++, Java in C#, se od paradigme proceduralnega programiranja odmikajo in uvajajo t.i. objektno orientirano programiranje (OOP).

9.3 C na mikrokrmilnikih

C je najbolj uporabljan programski jezik za programiranje mikrokrmilnikov, saj omogoča nizkonivojsko programiranje – programiranje, ki je blizu strojnem jeziku.

9.4 Zaključek

Pričujoče gradivo za vaje pri predmetih Mehatronski sistemi in Diskretni krmilni sistemi obravnavajo programski jezik C, katerega poznavanje je pomembno za vsako programersko udejstvovanje. Podana znanja je možno nadgrajevati v več smereh, od programiranja mikrokrmilnikov, do objektno orientiranega programiranja, ki pa so obravnavane v sklopu nadaljnega študija mehatronike na Fakulteti za strojništvo Univerze v Ljubljani.