Zapiski iz pouka Osnove programiranja II
Programiranje Mikrokontrolerjev

Matej Blagšič

5. april 2018

Kazalo

1 Programiranje v C

1.1 Osnovno

Pri temu predmetu bomo obravnavali jezik C. Za uporabo lahko preneseš okolje Codeblocks z MinGW inštalacijo ali posebej MinGW compiler in poljubno okolje(Jetbrains).

Pomembno je, da imaš predznanje iz prejšnjega polletja pri Javascriptu, saj so tipi spremenljivk, sintaksa in drugo zelo podobno, tako da v detajle o stvarih, ki so enake ne bom šel.

Vsak dokument začnemo z `@#include <stdio.h>@` za standardne vhodne in izhodne ukaze.

Vsa koda se izvaja znotraj main funkcije:

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

Prav tako je pomembno uporabiti PODPIČJE za vsakim ukazom/vrstico!!!

Če začnemo na začetku, opazimo `#include` ukaz. Ta se izvrši, preden se karkoli drugega. V temu primeru lahko vnesemo knjižnice. Te nam olajšajo programiranje s tem, da nam en ukaz izvede več ukazov, ki bi jih morali tipkati na roke. To datoteko/knjižnico navedemo lahko z "datoteka"navednicam. Če pa damo v <datoteka>, potem pa išče datoteke v sistemskih mapah okolja. Te datoteke so vrste **header** s končnico **.h**. V našem primeru je knjižnica za pisat in brat podatke - vhodne in izhodne podatke.
To je podobno kot v javascriptu: `<script src="datoteka">`

1.2 Podatki

Poglejmo si zgled:

Program 1: Branje in pisanje podatkov

```
int main(){
    int a;
    float b; //spremenljivka

    printf("Vprisi prvo vrednost");
    scanf("%d", &a);
    printf("Vpisi drugo vrednos");
    scanf("%f", &b);
    printf("%d + %f = %f\n", a, b, a+b);
    return 0;
}
```

C je občutljiv na tip podatkov. Pravimo tudi, da je C statično tipiziran jezik. To pomeni, da moramo vrsto podatka navesti. To pomeni, da se moramo sami odločiti, kakšen tip

podatka bo nosila spremenljivka.

Vemo, da v Javascriptu nismo rabili napisati tipa spremenljivke, le `var`, torej je Javascript dinamično tipiziran jezik.

1.2.1 Branje podatkov

Da nam program prebere podatek, uporabimo funkcijo:

```
scanf("formatno_dolocilo", &spremenljivka);
```

Vidimo, da moramo najprej deklarirati tip podatka, ki ga pričakuje operator Scanf. Potem pa določimo naslovni operator & in nato za njim spremenljivko, ki naj sprejme podatek.

Še eno "pravilo scanf funkcije. Funkcija dejansko vrne število. To kar vrne je odvisno od "uspeška"funkcije. Vrne lahko 2, če so bili vhodni podatki pravi. Vrne 1, če je bil en podatek mal napačen. Vrne 0, če je bil en podatek čisto napačen.

1.2.2 Pisanje podatkov

Za pisanje podatkov uporabimo funkcijo:

```
printf("formatni_niz", izrazi)
```

Pomembne so tudi ubežne sekvence. To so `\r \n \t`, ki povejo, kaj se zgodi, ko se text izpiše. `\n` naredi novo vrstico(new line) po besedilu, `\t` je tabulator...

Tako v našem primeru, se a izpiše tam, kjer je njegov `%d` in b, kjer je `%f` ter vsota $a + b$ tam, kjer je `%f` (glej izsek programske kode).

1.2.3 Spremenljivke:

V C-ju Boolov tip ne obstaja, tako da primerjalni operatorji delujejo enako, le da vračajo 0 za false in 1 za vse, kar je različno od nič. **Ne obstaja TRUE ali FALSE**.

Spoznali bomo tudi, da je pri celoštevilskem tipu pomembna omejitev območja, pri realnem tipu pa natačnost!

Velikokrat bomo srečali izraz `unsigned`. ta nam območje podatkovnega tipa prestavi od 0 do 2× maksimum. Če je char od -128 dp 127, potem je unsigned char od 0 do 255;

1.2.4 Celoštevilski tip (n bitov)

Obstaja nepredznačen, ki je od 0 do 2^{32}

TIP	DOLŽINA(bitov)	FORMATNO DOLOČILO	OBMOČJE
char	8	`%d %c`	-128 do 127
short, int	16	`%d`	-65536 do $+65535$
	32		-32768 do $+23767$
long	vsaj 32	`%ld`	-2.1×10^9 do $+2.1 \times 10^9$
float	običajno 32	`%f`	-2.1×10^9 do $+2.1 \times 10^9$
double	običajno 64	`%lf`	-9.2×10^{18} do $+9.2 \times 10^{18}$
void	0		

Tabela 1: Tipi spremenljivk v c-ju

1.2.5 Realni tip (IEEE floating point)

p	eksp. (e)	mantisa (m)
1 bit	8 bitov	23 bitov

Ta ima enojno natančnost (single precision) ali ***float*** in so števila zapisana z 32 bitno velikostjo. Tako so v desetiškem sistemu števila natančna do 7,22 signifikantnih mest, sepravi 7 mest je natančnih, od 8. števila naprej pa je že vprašljivo. Torej, signifikantno pomeni pomembno, tisto, kar je natančno.

Če hočemo večjo natančnost, uporabimo ***double*** oz. dvojna natačnost (double precision). Ima kapaciteto 64 bitov, torej v desetiškem do 15,95 mest natančno. Po 15. mestu je že vprašljivo natančno.

1.2.6 Modifikatorji formatna določila

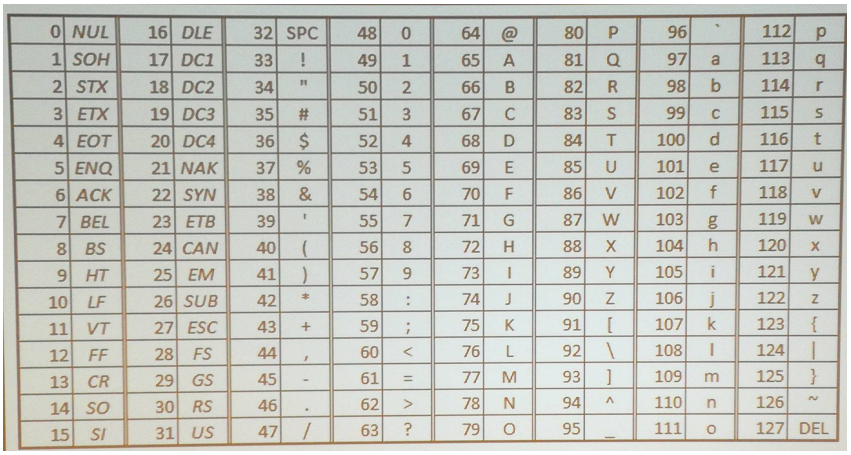
`%d` vemo, da stoji za cela števila. Če vrinemo neko število "N" → `"%Nd"`, potem povemo, na koliko mest se izpiše število, deluje na desno poravnavo.
Če vrinemo ničlo → `"%0Nd"`, potem zapolne prazna mesta z ničlam.
Če vrinemo "N.Mf"→ `"%N.Mf"`, potem izpiše N mest število z M mesti za decimalno piko.

```
int x = 15;
float y = 3.141592;

printf("%5d",x); --> Izpise _ _ _1 5
printf("%.2f",y); --> Izpise 3.14
printf("%05d",x); --> Izpise 00015
```

1.2.7 Znaki

Imamo več standardov znakov. Najbolj osnoven in razširjen je `ASCII` (American standard code for information Interchange). Ta zapis je 8-biten. Lahko najdemo tabelco, ki nam pokaže kodo za vsak znak.



0	NUL	16	DLE	32	SPC	48	0	64	@	80	P	96	`	112	p
1	SOH	17	DC1	33	!	49	1	65	A	81	Q	97	a	113	q
2	STX	18	DC2	34	"	50	2	66	B	82	R	98	b	114	r
3	ETX	19	DC3	35	#	51	3	67	C	83	S	99	c	115	s
4	EOT	20	DC4	36	$	52	4	68	D	84	T	100	d	116	t
5	ENQ	21	NAK	37	%	53	5	69	E	85	U	101	e	117	u
6	ACK	22	SYN	38	&	54	6	70	F	86	V	102	f	118	v
7	BEL	23	ETB	39	'	55	7	71	G	87	W	103	g	119	w
8	BS	24	CAN	40	(	56	8	72	H	88	X	104	h	120	x
9	HT	25	EM	41	)	57	9	73	I	89	Y	105	i	121	y
10	LF	26	SUB	42	*	58	:	74	J	90	Z	106	j	122	z
11	VT	27	ESC	43	+	59	;	75	K	91	[	107	k	123	{
12	FF	28	FS	44	,	60	<	76	L	92	\	108	l	124	|
13	CR	29	GS	45	-	61	=	77	M	93	]	109	m	125	}
14	SO	30	RS	46	.	62	>	78	N	94	^	110	n	126	~
15	SI	31	US	47	/	63	?	79	O	95	_	111	o	127	DEL

Slika 1: ASCII tabela

V C-ju je pomembno, da damo en znak v enojne navednice. S tem pomeni, da program zaznava ASCII kodo. Torej, če izpišemo `printf("%d", '0');`, nam program izpiše 48. Pri znakih uporabimo torej spremenljivko `char`.

1.2.8 Psevdonaključna števila

Ena metoda za pridobivanje naključnih števil je metoda srednjih kvadratov. Pri tem kvadriramo dvomestna števila. Pri tem je statistično gledano naključnost zelo podobna realni naključnosti, kot da bi metali kocke.

1.2.9 Statične in dinamične spremenljivke

Glej priloženo kodo spodaj!

V temu programu imamo funkcijo tipa void, ki kot vemo ne vrne nič, pred funkcijo main. prav tako imamo definirano spremenljivko ga, ki je zunaj kode. Ta je zato globalna in je uporabna v vseh funkcijah. Te globalne spremenljivke so **STATIČNE**. To pomeni da so te spremenljivke vedno na voljo in hranijo vrednost ves čas, kajti prostor za njih je že rezerviran v začetku.

Za kontrast, vse lokalne spremenljivke so **DINAMIČNE**, razen če so definirane kot statične z ukazom `static` pred definicijo spremenljivke(glej kodo, vrstica 6). Če poženemo kodo vidimo, kako deluje ukaz static. Ker smo spremenljivko sa definirali le enkrat in ji dodali vrednost 12, potem se ne definira ponovno vsakič, ko pridemo v to funkcijo foo() še enkrat. Zato se vrednost te spremenljivke povečuje. Za kontrast, spremenljivka a se ne povečuje, saj se vsakič, ko pridemo v funkcijo foo() ponovno definira.

Prav tako velja, da statične neinicializirane spremenljivke dobijo vrednost 0. To ne velja za dinamične, tako da če ne zapišemo začetne vrednosti spremenljivke a, potem vidimo, da program meče za vrednosti a kar nekaj.

Program 2: Statične in dinamične spremenljivke

```c
#include <stdio.h>
int ga; //globalna spremenljivka

void foo(){
    int a = 12; //lokalni spremenljivki
    static int sa = 12; //spremenljivka je staticna
    a++;
    sa++;
    ga++;
    printf("%d, %d, %d\n", ga, a, sa);
}
int main(){
    for(int i=0; i < 5; i++){
        foo();
    }
    return 0;
}
```

1.2.10 Zbirka(array)

Pomnilnik je razdeljen na pomnilniške celice v velikosti 8 bitov na celico. Ko naslavljamo celico, je vedno naslednja celica za 1 večja od prejšne po vrstni številki.

```
tipElementa imeZbirke[dim] = {element1, element2, element3, ...}
```

V temu primeru, je „dim" dimenzija zbirke. Naslavljamo jih lahko enako, kot v javascriptu, to je, da napišemo ime zbirke in število, ki predstavlja mesto želenega elementa. Ne moremo izpisati celotne zbirke naenkrat, temveč le po en element naenkrat. Pomembno je tudi, da ena zbirka določenega tipa lahko vsebuje elemente le tega tipa, torej vsi enaki. Ne moremo mešati različnih tipov, kot v javascriptu.

Poglejmo si zanimiv primer:

```c
#include <stdio.h>

int a[] = {2, 4};
int b[] = {1, 3};

int main(){

    a[2] = 42;
    b[-1] = 42;

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

    return 0;
}
```

V tem primeru, imamo globalna arraya a in b. Imata dva elementa, torej je njuna velikost 2. Nato pa v main funkciji elementu 2 v arrayu a in elementu -1 v arrayu b priredimo

vrednost 42. In potem, ko izpišemo drugi člen arraya a in prvi člen arraya b, se nam izpišeta točno te dve števili. Zelo je čudno, saj ne mormo iti v negativno mesto arraya, drugo mesto arraya pa ne obstaja, saj pri obeh gre le 0 in 1. mesto.

Žal na predavanjih še nismo obdelali kazalcev. Zakaj se to zgodi pojasnijo kazalci v naslednjem poglavju. Spremenljivki a in b sta v resnici kazalca na začetek seznamov, ki sta jima prirejena. a torej kaže na prvi element v seznamu in ko kličemo `a[0]` v resnici zahtevamo element, ki se nahaja na mestu, na katerega kaže a, plus 0. Torej element na začetku seznama. Ko kličemo `a[1]` tako zahtevamo element, ki se nahaja na mestu na katerega kaže a plus 1. Torej naslednji element. Več o kazalcih in seznamih si lahko prebereš `na tej povezavi`.

Seznama `a` in `b` pa imata v tem primeru še eno lastnost – deklarirana sta en za drugim ter vsebujeta zelo malo elemntov. Zaradi tega se seznam `b` na RAM-u nahaja takoj za seznamom `a`. Ko pokličemo `a[2]` gremo v resnici na prvi element v seznamu `b`, ko pokličemo `b[-1]`, pa gremo v resnici na zadnji element seznama `a`. Ko kasneje izpišemo zadnji element seznama `a` in prvi element seznama `b`, se izpišeti točno te števili.

Ker v obeh primerih nastavimo število v seznamu na 42, lahko dobimo lažen občutek, da se v vrstici `a[2]` števec obrne in gremo nazaj na prvo mesto v seznamu `a`, v vrstici `b[-1]`, pa da gre števec od zadaj in pišemo na zadnje mesto v seznamu `b`. Da se to v resnici ne zgodi, lahko preverimo na zelo preprost način – vsakemu seznamu nastavimo drugačno število. Poglejmo si naslednjo kodo.

Program 3: Zbirke primer

```c
#include <stdio.h>

int a[] = {2, 4};
int b[] = {1, 3};

int main(){
    a[2] = 42;
    b[-1] = 69;

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

    return 0;
}
```

Ta program bo sedaj izpisal `69 42`, torej smo res z `b[-1]` nastavili zadnji element v `a`, z `a[2]`, pa smo nastavili prvi element v `b`.

1.2.11 Znakovni niz(string)

V C-ju ne ovstaja string kot samostojen tip. Ampak vemo, če poznamo zbirke in char, da je string nič drugega kot zbirka char elementov. Zato lahko definiramo string kot v prvem ali drugem okvirčku. Prvi je namreč bolj kot zbirka, a težje za vnašanje:

```c
char niz[] = 'a','b','c','d'..... 'n'
```

```c
char niz[] = "abcd....n"
```

Lahko definiramo tudi drugače, kot je zapisano v drugem okvirčku.

Poglejmo primer. Imamo program, ki nam izpiše string kot posamezne elemente zbirke txt v for zanki. For zanka je napisana malo (niste) drugače. Namesto primerjave v srednjem delu, imamo primerjavo i-tega člena zbirke txt in če je (niste) ta različna od nič oz. če obstaja, zato lahko tam napišemo ali: `txt[i] != 0` ali pa `txt[i]`, potem jo izpiše. Tako se pomika po zbirki navzgor. Na koncu vsake zbirke, je "nevidna"ničla in ko jo najde, jo več ne izpiše in se for zanka konča. Jaz sem uporabil drugo metodo.

Program 4: Znakovni niz: navajanje 1

```c
#include <stdio.h>

int main(){
    char txt[] = "Tole izpisi";
    for (int i = 0; txt[i]; i++){
        printf("%c", txt[i]);
    }
    return 0;
}
```

Ker se string rabi pogosto, obstaja formatno določilo %s. Tako lahko napišemo program kot kaže koda. Pomni, da ko izpišeš, uporabi le ime char array-a brez oglatih oklepajev.

Program 5: Znakovni niz: navajanje 2

```c
#include <stdio.h>

int main(){
    char txt[] = "Tole izpisi";
    printf("%s", txt); // <-- tu txt, ne txt[] ter formatno dolocilo %s
    return 0;
}
```

1.2.12 Kazalci

Kazalce sem omenil že pri zbirkah in da so ti razlog, da se zbirke obnašajo tako, kot se. Rekli smo, da je spremenljivka a[] zares kazalec na začetek zbirke in s številom v oklepajih povemo na katero mesto stran od kazalca naj beremo. Vrednost kazalca je torej pomnilniški naslov.

Kazalec lahko definiramo kot: `tip *p;`. S tem definiramo, da je p kazalec in hrani le naslov v pomnilniku. s sledečim ukazom mu povemo, naj kaže na premenljivko: `p = &x;`. če želimo na mesto te spremenljivke, na katero kaže prirediti vrednost, potem izpišemo: `*p = 42`. Sedaj, če izpišemo vrednost spremenljivke x, nam vrne vrednost 42. Tip kazalca mora biti enak tipu spremenljivke, na katero kaže!

Kaj je torej ta zvezdica. To je operator indirekcije, ki dobi naslov podatka in gre direktnon na ta naslov. Torej zgoraj v zadnjem okvirčku torej pomeni, da z operacijo *p ne operiram s p-jem, teveč s podatkom, kamor kaže p, torej na spremenljivko x. Zato zgoraj v okvirčkih vidimo, da ko izvedemo ukaz `*p = 42`, nam program manipulira s prostorom,

kjer se nahaja spremenljivka x, na katero kaže kazalec p.

Torej ukazi ki jih imamo na voljo so prikazani v tabeli primera kode:

operacija	opis	vrne
`int sprem = 20;`	v spremenljivko sprem shrani vrednost 20	
`int *p;`	ustvari kazalec p	
`p = &sprem;`	shrani naslov spremenljivke v kazalec p	
`printf("%x", &sprem);`	izpiše naslov spremenljivke sprem	`6af36e8c`
`printf("%x", p);`	izpiše vrednost p / naslov mesta, kamor kaže	`6af36e8c`
`printf("%d", *p);`	izpiše vrednost mesta, kamor kaže p / sprem	`20`

Program 6: Kazalci: Mesta v zbirki

```c
#include <stdio.h>

int main(){
    int zb[5];
    int *p1, *p2;
    p1 = &zb[1];
    p2 = &zb[3];
    printf("%d", p2-p1);
    return 0;
}
```

Sledeča koda najprej naredi zbirko 5-ih elementov. Nato ustvarimo dva kazalca z imeni p1 in p2. Ta nastavimo, da kažeta p1 na 2. člen zbirke ter p2 na 4. člen zbirke. Nato odštejemo kazalca med seboj, in nam program vrne 2, torej razliko mest med njima.

Program 7: Kazalci: Izpis podatkov s kazalci in zbirke

```c
#include <stdio.h>

int main(){
    int zb[5] = {11, 22, 33, 44, 55};
    printf("%d", *(zb+2));
    return 0;
}
```

V temu primeru izpišemo vrednost elementa, ki se nahaja na naslovu za 2 stran od naslova zb. Torej lahko naslavljamo kot: `&zb[0] = zb`. Torej, če napišemo zb, s tem kažemo na naslov, kjer se začne zbirka. Z zvezdico beremo podatek, iz tega naslova. Torej velja tudi `*zb = zb[0]`

Program 8: Menjava vrednosti elementov s kazalci

```c
#include <stdio.h>

void menjaj(int *a, int *b){
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main(){
    int x = 10, y = 20;
    menjaj(&x, &y);
    printf("%d, %d", x, y);
    return 0;
}
```

Če želimo zamenjati vrednosti dveh spremenljivk, moramo najprej eno odložiti na začasno mesto, drugo preslikati na njeno mesti in skopirati vrednost iz začasnega mesta na prvotno mesto druge. S tem moramo upravljati z globalnimi spremenjivkami ..... Lahko pa s kazalci. Rečemo, da sta v void funkciji, ki nič ne vrne, kažeta elementa a in b na x in y vhodna podatka. tako operiramo z originalnimi podatki v main funkciji, ne da bi potrebovali funkcijo, ki bi vračala podatke ali jih pisala...

Program 9: Izpis stringa s kazalci

```c
#include <stdio.h>

int main (){
    char a[] = "neki";
    printf("%s",a+2);
    return 0;
}
```

Zgornja koda nam izpiše del stringa, ki je od 2. člena naprej. Torej nam izpiše `"ki"`.

Program 10: Izpis stringa in upravljanje s stringi

```c
#include <stdio.h>

void vVelikeCrke(char *s){
for (int i = 0; s[i]; ++i){
    if(s[i] >= 'a' && s[i] <= 'z'){
        s[i] += 'A' - 'a';
        }
    }
}

int main (){

    char a[] = "neki Cudnga 123$";

    vVelikeCrke(a);
    printf("%s",a);
    return 0;
}
```

Ta koda nam pretvori vse črke v stringu na velike. Opazimo enko naslavljene, ko izpisujemo string.

Naslednji primer nam pride prav pri avditorni vaji oz. pri bubble sort, kjer je neučinkovito prestavljati celotne strukture, temveč le kazalce. (poglej si strukture tu)

Program 11: Strukture in kazalci

```c
#include <stdio.h>

struct oseba{
    char ime[30];
    int starost;
};

int main (){

    struct oseba os1, os2 = {"Joze", 42};
    struct oseba *p1, *p2, *tmp;

    p1 = &os1;
    p2 = &os2;

    printf("%d\n", p2->starost);//(*p2).starost

    tmp = p1;
    p1 = p2;
    p2 = tmp;

    printf("%s\n", p1->ime);
    return 0;
}
```

Imamo operator `p1 -> ime`, ki nam zahteva element ime iz strukture, na katero kaže kazalec p1. Poglej si v zapiskih avditornih vaj za razlago v poglavju 6, 2. naloga.

1.3 Operatorji

Pri C-ju so enaki operatorji, kot v JS, le da z nekimi izjemami: Operator === in !=== ne obstajata.
Prav tako operator za deljenje ne zapišemo kot "/" ne deluje enako. Problem prihaja iz tipa spremenljivk. Če obsoječo spremenljivko x, ki je tipa int, deljimo ali spreminjamo tako, da bi postala ta spremenljivka kateregakoli drugega tipa, kot prvotni int, potem vrne program 0. Primer:

```c
int maint(){
    x = 7;
    x = x / 8 * 8
    printf("%d", x);
    return 0;
}
```

Če pa spremenimo prvo 8 z 8.0, potem bo program jo vzel za realno število in deljil in

nato nazaj množil z 8, tako se te pokrajšata in program vrne 7.

1.4 Funkcije

Funkcije deklariramo tako:

```
tip_funkcije imeFunkcije(){/*telo funkcije*/return 0;}
```

Opazimo, da funkcijo deklariramo kot float oz. funkcijo, ki vrne realno število. V resnici lahko funkcije definiramo kot karkoli hočemo, glede na to, kaj naj bi vrnila.

Prav tako vidimo, da glavna zanka, v kateri se koda izvaja, je `main`. v tej kodi se izvajajo vsi programi in funkcije. Tako se koda, ki je napisana tu notri, se prevede in spremeni v izvršilno kodo(executable).

Primer funkcije je iz poglavja Psevdonaključna števila. Tam smo spoznali definiranje funkcije: `int dogodek(float verjetnost);` Vidimo, da moramo za razliko od JS definirati vrsto spremenljivke, ki gre v vhodne podatke, tj. verjetnost. Poleg tega, ker se konča z podpičjem, imenujemo ta del prototip. Nič ne naredi. Nato definiramo šele funkcijo.

Program 12: Naključnost s funkcijo srand in brez

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

int dogodek(float verjetnost){
    if((float)rand() / RAND_MAX <= verjetnost){
        return 1;
    }
    return 0;
}

int main(){

    int x, stevec = 0;

    srand(time(NULL));

    for(x=0; x < 100000;x++){
    stevec += dogodek(0.5);
    }
    printf("%d", stevec);

    return 0;
}
```

V funkciji je pomembno, da pretvorimo rand() v float tip spremenljivke, ker drugače gre za celoštevilsko deljenje, kar potem pomeni le 0 ali 1. Problem je, da nam potem vsakič vrne enako vrednost okoli 50 000, ker je ta random le psevdonaključna. Zato srednjo verednost definiramo z ukazom `srand`(oz seme) in vanj vnesemo čas, ki pa nikoli ni enak. Zato tako vsakič generira zares naključno število.

Naredimo primer na bolezni. Izračunajmo, koliko % ljudi, ki so bolani zares, zanje test pokaže, da so res bolani.

Testiramo 100 000 ljudi in vemo, da bolezen ubije 0.5% ljudi. Prav tako vemo, da test pokaže z natančnostjo 99%, da je oseba bolana. 1%, da je oseba zdrava. Ampak ali je res, da je 1% zdravih ali napačno diagnosticirano.

Program 13: Primer naključnosti z bolanimi ljudmi

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

int dogodek(float verjetnost){
    if((float)rand() / RAND_MAX <= verjetnost){
        return 1;
    }
    return 0;
}

int main(){

    unsigned long i;
    float pBolan = 0.005; //verjetnost da ubije
    float pPozitBolan = 0.99; //resnicno bolan verjetnost
    float pPozitZdrav = 0.01;//verjetnost da je bolan, ceprav je zdrav
    unsigned long pozit = 0;
    unsigned long pozitBolan = 0;

    srand(time(NULL));

    for(i = 0; i<100000;i++){
        if(dogodek(pBolan)){//vemo da je bolan
            if(dogodek(pPozitBolan)){//testiramo kako dobro izmerimo,ce je bolan
                pozitBolan++;//dodamo ga med bolane in pozitivno testirane
                pozit++;
            }
        }
        else{//testiramo zdravega
            if(dogodek(pPozitZdrav)){//tu se znajde zdrav in pozitivno testiran
                pozit++;
            }
        }
    }

    printf("%f", (float)pozitBolan/pozit);// rezultat je bolni/testirane pozitivno
    return 0;
}
```

1.5 Strukture

V C-ju, za razliko od C++ in JS ni objektov. Zato imamo strukture, ki so nekako podobna zadeva. Definiramo z :

```
struct ime{
    tip1 ime1;
    tip2 ime2;
    ...
};
```

Struct s spremenljivko ime je nov podatkovni tip. In ko definiramo komponente tega novega podatkovnega tipa, definiramo novo spremenljivko kot: `struct ime sprem;`

Če hočemo klicati spremenljivko, definiramo kot `sprem.ime1`; Glej kodo spodaj za referenco:

Program 14: Kopiranje struktur

```
#include <stdio.h>

struct vektor {
    float x, y;
};

int main() {
    struct vektor v1, v2;

    v1.x = 1.4;
    v1.y = -0.7;

    v2 = v1; // ustvari se cista kopija, ne samo povezava do spremenljivke, kot v JS
    v2.x = 13; // ce odstranimo vrstico, potem nam program vrne 1, drugace 0
    printf("%d\n", v1.x == v2.x); // testiramo, da vidimo, ali je enakost, %d, ker
        nam vraca 1 ali 0(boolean)

    return 0;
}
```

Primer uporabe je sledeča koda. Napišemo program, ki nam izračuna vektorski produkt dveh vektorjev. Vektorski produkt dveh vektorjev vemo, da je nov vektor, ki je pravokoten na prvotna dva $\vec{a}$ in $\vec{b}$. Vektorski produkt lahko rešimo z determinanto 3×3 matrike vektorjev. Po definiciji sledi:

$$\vec{a} \times \vec{b} = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix} = \begin{bmatrix} a_y \cdot b_z - a_z \cdot b_y \\ a_z \cdot b_x - a_x \cdot b_z \\ a_x \cdot b_y - a_y \cdot b_x \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Program 15: Strukture: vektorski produkt

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

struct vektor{
    int x, y, z;
};

int main(){

    srand(time(NULL));

    struct vektor v1, v2, r;

    v1.x = random()%11;
    v1.y = random()%11;
    v1.z = random()%11;

    v2.x = random()%10;
    v2.y = random()%10;
    v2.z = random()%10;

    r.x = v1.y*v2.z - v1.z*v2.y;
    r.y = v1.z*v2.x - v1.x*v2.z;
    r.z = v1.x*v2.y - v1.y*v2.x;

    printf("Nov vektor je: (%d, %d, %d)\n", r.x, r.y, r.z);

    return 0;
}
```

Program nam torej reši vektorski produkt dveh naključno izbranih vektorjev. Uporablja se struktura za vektorje, naključna funkcija `random()` ter formula za vektorski produkt.

1.5.1 Podrobnejša razlaga struktur

Slišal sem, da nekateri ne razumejo principa delovanja struktur, zato sem dodal sledečo razlago, ki naj bi pomagala k razumevanju. Dele razlage bom obarval za lažje sledenje razlagi.

Za to kar bomo potrebovali si lahko predstavljate strukturo kot mašina za izdelovanje svojih vrst spremenljivk. Pomislite na vse tipe spremenljivk, ki so v preglednici, torej `int`, `short`, `char`, `long`, `float` ... Vse te vrste spremenljivk imajo svoja pravila delovanja, uporabe prostora v pomnilniku ipd. Enako struktura. Definiramo svoj tip spremenljivke in z njim naprej upravljamo. Dajmo najprej za začetek definirati naš nov tip spremenljivke.

```c
struct tip{
};
```

S to " zanko " oz ukazom smo naredili novo spremenljivko z imenom `tip`. Sedaj so

lastnosti/komponente te spremenljivke še nedefinirane, zato ji podajmo lastnosti:

```
struct tip{
    int vred1;
    int vred2;
};
```

Kaj smo torej naredili. Tipu spremenljivke `tip` smo dodali dvoje vrednosti z imeni `vred1` in `vred2`. Ti sta tipa `int` ampak pri razlagi to nima veze.
Torej, ustvarili smo tip spremenljivke, ki ima dvoje vrednosti. Kako sedaj kreiramo dejansko spremenljivko tega tipa `tip`?

```
struct tip spremenljivka; // kreacija nove spremenljivke
tip spremenljivka;        // deluje tako, ce bi bila TIP uveden
/* enak nacin definiranja nove spremenljivke,kot ze poznamo: */
int spremenljivka;
```

V zgornji kodi smo sedaj ustvarili novo spremenljivko tipa `tip`. To si lahko predstavljamo kot evivalent drugi vrstici, če odmislimo ukaz `struct`. `tip` je "enak "`int`

Sedaj želimo dostopati do podatkov naše spremenljivke. Ker smo naredili spremenljivko, ki ima več vrednosti, moramo do teh dostopati sledeče:

```
spremenljivka.vred1 = 10;
spremenljivka.vred2 = 20;
```

V zgornji kodi smo priredili spremenljivki določene vrednosti. A ker ima naša spremenljivka, ki je tipa `tip` več vrednosti, moramo te naslavljati posamično. Zato uporabimo piko in za njo ime vrednosti, na katero naslavljamo. Tako v kodi priredimo vrednost 10 `vred1` ter vrednost 20 `vred2`. Sedaj ima naša spremenljivka `spremenljivka` dodeljene določene vrednosti svojim komponentam. Če želimo izpisati vsebino spremenljivke `spremenljivka`, deluje enako:

```
printf("%d %d", spremenljivka.vred1, spremenljivka.vred2);
```

Enako, kot pri prirejanju vrednosti, moramo pri izpisovanju naslavljati posamične komponente/vrednosti naše spremenljivke tipa `tip` po komponentah.

Sedaj, zakaj bi hoteli imeti neke svoje vrste spremenljivke? Potreba pride iz neke enostavnosti in čistoče kode. Primer pride iz baz podatkov. Tam imam n oseb in vsaka ima več karakteristik; ime, priimek, spol, starost, teža ipd. In da se ognemo vsem teh spremenljivkah za n oseb, kjer bi imeli n spremenljivk za imena, priimke ..., naredimo škupino"oz kategorijo z imenom OSEBA, ki vsebuje vse te podkategorije/lastnosti in nato lahko recimo sortiramo osebe po teh komponentah. To smo delali pri avditornih vajah, zato si lahko pogledaš primer tam.

Celoten postopek, ki bi ga uporabil je:

```
#include <stdio.h>

struct tip{   //definiramo nov tip spremenljivke z imenom TIP
    int vred1;
    int vred2;
```

```
};

int main(){

    struct tip spremenljivka; // definiramo novo spremenljivko tipa TIP

    spremenljivka.vred1 = 10; // priredimo vrednosti komponentam spremenljivke
    spremenljivka.vred2 = 20;

    printf("%d, %d", spremenljivka.vred1, spremenljivka.vred2); // izpis komponent
    //izpise: 10, 20

    return 0;
}
```

2 Mikrokrmilniki

Uvod Sedaj bom ozačeli s programiranjem mikrokrmilnikov. Sedaj bo lahko naša koda naredila kaj drugega, kot samo izpisovanje podatkov na ekran.

Kako Arduino deluje se mi po pravici ne da pisati na dolgo in široko, zato ti proporočam, da odpreš Youtube in vpišeš Arduino tutorial in si pogledaš toliko videjev, dokler ti oči ne padejo iz jamic. Ko bomo delali vezja in programe, bom seveda zraven razlagal, zakaj dogaja to, kar dogaja, zraven, ampak je vseeno dobro, da si pogledaš celotno zadevo.

2.1 Osnovno

Sledeč del je zasnova za vsak program. Ker vemo, da se program začne od zgoraj, se prvo izvede zanka setup. To je zanka, ki se izvede le 1x. Ko se ukazi znotraj setup izvedejo, gre program na loop. Ta se izvaja v zanki, zato se imenuje loop. To pomeni, da bo mikrokontrler ves čas izvajal kodo, ki se nahaja v loop zanki.

```
void setup(){
}

void loop(){
}
```

Imamo za začetek simpel program. Želimo blinkati vgrajeno LED.

```
void setup(){
    pinMode(LED_BUILTIN, OUTPUT)
}
void loop(){
    digitalWrite(LED; BUILTIN, HIGH)
    delay(1000);
    digitalWrite(LED; BUILTIN, LOW)
    delay(1000);
}
```

Imamo v setup funkciji definicijo `pinMode(LED_BUILTIN, HIGH)` Ta funkcija, sprejme za prvi vhodni podatek številko pin-a oz. nogice, katero želimo manipulirati. Drugi pogoj pove ali se bo ta uporabljal kot izhodni ali vhodni. Tako definiramo, da je izhodni in želimo, da to deluje za digitalni pin 13, ki je tudi isti pin, na katerega je priklopljena LED.

Nato gremo na loop, kjer z ukazom `(LED_BUILTIN, HIGH)` manipuliramo pin, da ga dvigne na visoko napetost, HIGH ali 1. Tako se LED prižge. Potem zahtevamo LOW ali 0, da se ugasne.