0. Sadržaj

1. Uvod

Ovo je uvod u programiranje u programskom jeziku C, namijenjen za studente FER-a u Zagrebu, ali se može koristiti i dijeliti bilo gdje bez znanja i dopuštenja autora.

Osnovne funkcije C-a su objašnjene, no samo one. Drugim riječima, ovdje se neće spominjati razne postojeće metode skraćivanja izraza, rijetko korištene funkcije, itd. Naglasak je na logici programiranja u svrhu razumijevanja, nakon čega se lako nauče ostale ne-osnovne stvari.

Ovaj tekst postoji jer je autor imao dovoljno volje i znanja.

Ovaj je tekst teško pratiti, no zato je direktan i kratak.

Ime autora je namjerno izostavljeno.

2. Osnovni program

Slijedi jednostavan program koji je najčešći prvi primjer u programiranju: Hello, world! Nakon toga je posebno objašnjen svaki redak.

```
#include<stdio.h>
int main(){
	printf("Hello, world!");	//funkcija za ispis
	return 0;			/*funkcija za vraćanje vrijednosti*/
}
```

je sustavni operator; slijedi naredba sustavu **include**
include se koristi za uključivanje setova naredbi i funkcija koje koristimo u programu.
stdio je standard input output -> nužan u svakom programu s obzirom da ~svaki program treba nekakve funkcije za unos i ispis.
int je kratica za integer, označava cijele brojeve. Koristi se pri definiranju funkcija i varijabli. U ovom slučaju pomoćno definira funkciju **main**. Za sada se to može ignorirati, tj. staviti u svaki program. Objašnjenje je u poglavlju 'Funkcije'.
main je glavna funkcija koja se uvijek prva pokreće, druge se funkcije pozivaju iz nje. Više o funkcijama je objašnjeno u poglavlju 'Funkcije'.
() služi da bismo funkciji (u ovom slučaju **main**) poslali argumente. U kontekstu programiranja za početnike, zagrade uz funkciju **main** će uvijek biti prazne.
{ označava početak funkcije.

printf je naredba koja ispisuje sve što se nalazi između navodnih znakova.
(i **)** su dio sintakse funkcije **printf**, kao i **"**...**"**.
; se stavlja na kraju svake naredbe (ne na kraju svakog retka).
return je naredba koja vraća neku vrijednost (u ovom slučaju **0**) pozivatelju. Više o tome u poglavlju 'Funkcije'.
0 je vrijednost koja se obično vraća kada je funkcija obavila zadatak koji je trebala; u ovom slučaju, funkcija **main** je obavila sve naredbe (samo **printf**) koje je trebala s obzirom da je došla do naredbe **return**.
; je ovdje jer označava kraj neke naredbe, kao i nakon **printf** naredbe.
} označava kraj funkcije.

Ispis ovog programa je: Hello, world!

Primjetimo da postoje komentari nakon funkcija **printf** i **return**. Prikazana su dva načina komentiranja; **//** i **/***...***/**. Kompajler ignorira sve napisano nakon **//** (do kraja reda) i unutar **/***...***/**; komentari služe da programerima olakšaju čitanje složenih kôdova tako da se objasni što se gdje događa.

Sada znamo napraviti program koji ispisuje što god želimo. Očito, takav program nije pretjerano koristan. Treba nam program koji pruža interakciju s korisnikom...

3. Varijable

Da bi program imao drugačiji rezultat koji ovisi o korisniku, mora imati funkcije koje će primiti input od korisnika i varijable u koje će spremiti vrijednosti koje korisnik upiše, i te će se varijable onda koristiti za što ih već trebamo.
Prva funkcija za prihvaćanje inputa je scanf, no počnimo s varijablama.

Postoje četiri glavna tipa varijabli u C-u (ostali tipovi se neće ovdje spominjati):
- int -> cjelobrojna varijabla
- float -> realnobrojna varijabla sa 7 decimala
- double -> realnobrojna varijabla s 15 decimala
- char -> znakovna varijabla

Prije korištenja, svaka se varijabla mora deklarirati; na taj se način zauzme memorija za nju. Taj dio memorije također ima svoju adresu. Više o adresama i memoriji je u poglavlju 'Pokazivači'.
Varijabli se može pristupiti direktno preko imena, i indirektno preko memorije.

Funkcija **scanf** traži unos podatka od korisnika i taj podatak sprema u neku varijablu, za to joj treba memorijska adresa te varijable.

Pogledajmo primjer programa. Program uzima dva cijela broja od korisnika, zbraja ih i ispisuje rezultat.

```
#include<stdio.h>
int main(){
	int a, b, c;
	printf("Upiši dva broja: ");
	scanf("%d,%d", &a, &b);
	c=a+b;
	printf("\nZbroj je %d.", c);
	return 0;
```

}

Prva dva retka su kao i u 'Hello, world!' programu.
int a, b, c je naredba za deklariranje tri cjelobrojne varijable nazvane **a**, **b** i **c**. Varijable se, dakle, odvajaju zarezima, i na kraju dolazi **;**.
scanf je naredba za primanje unosa od korisnika.
(i **)** su dio sintakse funkcije **scanf**, kao i **"..."**.
%d je operator za primanje cijelog broja. Dva su takva operatora odvojena zarezom (**,**) -> u ovom se slučaju varijable (pribrojnici) moraju upisati razmaknuti zarezom. U slučaju float broja, operator je %f, u slučaju double broja, operator je %lf, a su slučaju znaka (char), operator je %c.
, nakon završnih navodnika je dio sintakse funkcije **scanf**.
& je operator za traženje adrese varijable. **a** i **b** su u ovom trenutku deklarirane varijable, i funkcija **scanf** će u njih spremiti vrijednost na način da unesene brojeve spremi na njihovu memorijsku adresu. Nakon ove naredbe varijable **a** i **b** su definirane.
c=a+b je naredba koja operatorom **=** u varijablu **c** sprema vrijednost operacije **a+b**. Sada je varijabla **c** definirana i sadrži zbroj varijabli **a** i **b**. Oduzimanje se postiže upotrebom **-**, moženje upotrebom *****, dijeljenje upotrebom **/** i ostatak dijeljenja upotrebom **%** (**%** ima i drugu svrhu, pogledajmo dalje).
**** je operator u funkciji **printf** koji služi za manipuliranje teksta koji se ispisuje. U ovom slučaju potpuni operator je **\n** -> ispisat će se 'new line' znak, tj. novi red prije ostatka teksta.
% je operator za ispis vrijednosti varijable. Potpuni operator je u ovom slučaju **%d** jer se ispisuje vrijednost cjelobrojne varijable. Isto kao i u funkciji **scanf**, koristi se **%f** i **%c** za ispisivanje float broja, odnosno znaka. Iznimka je varijabla tipa double koja za **scanf** treba imati %lf, a za **printf** treba %f.
, nakon navodnika je dio sintakse funkcije **printf** kada se ispisuje vrijednost varijable; ako ispisujemo više varijabli odvajamo ih zarezima.
U funkciji **printf** nam ne treba adresa varijable da bismo ju ispisali, možemo joj direktno pristupiti nazivom: **c**.

Važno je objasniti sintaksu za rad sa znakovima.
Možemo imati slijedeću deklaraciju znakovne varijable.
char c;
Kako ćemo joj pridružiti neki znak?
Ako napišemo **char c=a;**, kompajler će nam javiti grešku; **a** je nedefinirana varijabla, stoga se njezina vrijednost ne može spremiti u neku drugu varijablu, u našem slučaju **c**.
Ispravna deklaracija izgleda ovako.
char c='a';
Dakle, koristimo apostrofe (jednostruke navodnike) da bismo definirali da je riječ o znaku **a**, a ne o varijabli **a**.

Ako imamo, s druge strane, varijablu **a** koja ima vrijednost 20, pa napišemo
char c=a;,
ili jednostavno napišemo
char c=20;,
u varijablu **c** će se spremiti upravo vrijednost **20**. Kakva je onda razlika između cjelobrojne varijable i znakovne varijable?
Kada ispisujemo znak, koristimo **%c** u funkciji **printf**, i na kraju **printf**-a pišemo ime varijable.
Pogledajmo dalje.

Smijemo imati slijedeće naredbe u programu.
int a=20;
char c='b';
printf("%d, %c“, a, c);

Ispisat će se broj **20**, i nakon toga znak **b**.
Smijemo također imati i ovakvu situaciju.
int a=20;
char c='b';
printf("%d, %c“, c, a);
U ovom slučaju ispis će biti cijeli broj koji odgovara malom slovu **b** u ASCII tablici, i znak koji odgovara broju **20** u ASCII tablici.

ASCII tablica je ključ koji koristi C programski jezik. Svaki znak ima svoj cjelobrojni ekvivalent. Zbog toga, ako znakovnu varijablu **c** želimo ispisati kao cijeli broj (koristeći **%d**), dobit ćemo cjelobrojni ekvivalent malog slova **b**, a ako cjelobrojnu varijablu **a** želimo ispisati kao znak (koristeći **%c**), dobit ćemo znakovni ekvivalent tog broja (**20**).

Vraćamo se na pitanje: 'Kakva je razlika između znakovne i cjelobrojne varijable?'
Točno je da smijemo napisati
int a='a';,
i time dati varijabli **a** brojčanu vrijednost koja odgovara malom slovu **a** u ASCII tablici.
Uostalom, kada napišemo
char c='c';,
varijabla **c** isto ima brojčanu vrijednost u sebi.
Na ovaj način za korištenje uopće nema razlike između znakovne i cjelobrojne varijable.
Razlika je slijedeća. Znakovna varijabla u memoriji zauzima jedan bajt, a cjelobrojna varijabla zauzima četiri.
Kada imamo male programe, ta nam nama razlika ne smeta, no ako imamo program s tisućama linija kôda i isto toliko varijabli, razlika nije zanemariva, tako da je dobro imati naviku korištenja znakovne varijable kada ju koristimo za znakove, a cjelobrojne kada koristimo brojeve.

Napomena: primijetimo da se vrijednost varijabli može pridružiti u trenutku deklaracije. Drugim riječima, možemo pisati
int a;
a=5;
ili
int a=5;.

4. Uvjetovanje

Ako želimo da program ne radi svaki put istu stvar, trebamo ga uvjetovati operatorom **if**.
Pogledajmo primjer: program koji dijeli dva unesena realna broja i ispisuje rezultat.

```
#include<stdio.h>
int main(){
	float a, b, c;
	printf("Upiši dva broja: ");
	scanf("%f %f", &a, &b);
	if(b==0){
		printf("\nNazivnik ne može biti jednak nuli.");
	}
	else{
		c=a/b;
		printf("\nOmjer je %f.", c);
	}
	return 0;
```

}

float a, b, c je naredba za definiranje tri realnobrojne varijable: **a**, **b** i **c**.
%f %f su dva operatora koja služe za unos **float** brojeva, u ovom se slučaju unose odvojeni razmakom.
if je ključna riječ za operator **if**.
(i **)** su dio sintakse za operator **if**. Unutar tih zagrada se nalazi uvjet: b==0. Znači, ako je **b** jednak nuli, dijeljenje je nemoguće i izvršava se niz naredbi unutar **{** i **}** (ispis poruke da nazivnik ne može biti jednak nuli).
Kod uvjeta, za jednakost koristimo **==**, za nejednakost koristimo **<**, **>**, **<=** i **>=**, a za različitost koristimo **!=**.
else je dio operatora **if**: u slučaju da uvjet operatora **if** nije zadovoljen (jer vrijedi **b!=0**), izvršavaju se naredbe unutar **{** i **}** od **else**-a (dijeljenje i ispis rezultata).

5. Petlje

Petlje se u C-u koriste kako ne bismo morali veliki broj puta pisati iste stvari: napisat ćemo petlju tako da program sâm zna što koliko puta treba ponoviti.

Pogledajmo primjer programa koji ispisuje kvadrate nekoliko prvih prirodnih brojeva. Koliko? Ovisi o unosu.

```
#include<stdio.h>
int main(){
	int n, i;
	printf("Upiši broj: ");
	scanf("%d", &n);
	for(i=1; i<=n; i++){
		printf("\n%d^2=%d", i, i*i);
	}
	return 0;
}
```

Definirali smo cjelobrojne varijable **n** i **i**. **n** koristimo kao granični prirodan broj, a **i** koristimo kao brojač u petlji, tj. koliko kvadrata prirodnih brojeva treba ispisati? **n**.
for je ključna riječ za **for** petlju.
Sintaksa **for** petlje je slijedeća:

- unutar zagrada se upisuju svi parametri odvojeni s **;**
- prvi parametar je početna vrijednost brojača (u ovom slučaju počinjemo od vrijednosti **1**, pa varijabli **i** pridružujemo tu vrijednost)
- drugi parametar je uvjet; petlja će izvoditi zadane naredbe sve dok je uvjet zadovoljen, tj., u ovom slučaju, dok god vrijedi **i<=n**
- treći parametar je korak petlje; situacija se mora mijenjati, inače bi uvjet uvijek bio zadovoljen (u ovom slučaju, vrijednost varijable **i** raste* nakon svakog prolaza petlje)
- unutar vitičastih zagrada se nalaze naredbe koje **for** petlja treba izvršiti svakim prolazom

Funkcijom **printf** ispisujemo kvadrat vrijednosti brojača** za svaki prolaz petlje:

- za i==1 ispisuje se 1^2=1
- za i==2 ispisuje se 2^2=4
- za i==3 ispisuje se 3^2=9
- itd.

Pojašnjenja:

* Napisali smo **i++** kada smo htjeli da se brojač **i** svaki put povećava za 1. Ovo je dopuštena sintaksa C-a. Isto tako možemo pisati **i--** ako želimo smanjiti vrijednost varijable za 1.
Dakle, **i++** radi isto što i **i=i+1**, tj. **i--** radi isto što i **i=i-1**.
** U ovom primjeru, postoji račun **i*i** unutar funkcije **printf**. Ovakav je način pisanja dozvoljen, tj. nije potrebno svaki put rezultat operacije spremati u posebnu varijablu da bismo ga ispisali. Dakako, to znači da smo u prethodna dva primjera sa zbrajanjem i dijeljenjem mogli napraviti istu stvar, tj. zadatak riješiti pomoću samo dvije varijable.
Funkcije koje petlja izvršava u ovom primjeru su, dakle, mogle biti napisane ovako:
kvadrat=i*i;
printf("\n%d^2=%d", i, kvadrat);
Prije toga bi, naravno, varijabla **kvadrat** trebala biti deklarirana.

Općenito, **for** petlja radi slijedećim redoslijedom:

1. postavi početnu vrijednost brojača (**i=1**)
2. provjeri je li uvjet zadovoljen (**i<=n**)
    a. ako je, izvrši naredbe
    b. ako nije, petlji je kraj
3. izvrši korak petlje (**i++**)
4. vrati se na 2. korak

for petlja je dobra kada znamo koliko puta se treba nešto ponoviti. No što ako ne znamo? Zato imamo **while** petlju. **while** se koristi na način da ponavlja niz naredbi sve **dok** vrijedi uvjet.
Pogledajmo primjer: program treba računati korijene cijelih brojeva sve dok se ne upiše negativan broj.

```
#include<stdio.h>
#include<math.h>
int main(){
	int broj;
	float korijen;
	printf("Upiši broj: ");
	scanf("%d", &broj);
	while(broj>=0){
		korijen=(float)sqrt((float)broj);
		printf("\nKorijen od %d je %f.", broj, korijen);
		printf("\nUpiši broj: ");
		scanf("%d", &broj);
	}
	printf("\nUpisan je negativan broj.");
	return 0;
}
```

math.h je ovaj put također potrebno uključiti jer nam je potrebna jedna matematička funkcija: **sqrt**.
printf i **scanf** funkcije ispred **while** petlje služe za unos, odnosno provjeru prvog broja pri ulasku u **while** petlju.
Petlja provjerava je li uvjet unutar **(** i **)** zadovoljen, u ovom slučaju **broj>=0**, tj. ako je broj nenegativan, petlja počinje izvršavati svoje naredbe unutar **{** i **}**.
Računa se korijen unesenog prvog broja, i potom se ispisuje. **sqrt*** (square root) funkcija vraća drugi korijen.
Potom se učitava idući broj. Petlja je došla do kraja pa se vraća na uvjet. Opet provjerava uvjet: ako je novi broj negativan, program ispada iz petlje i ide na kraj, tj. ispisuje da je upisan negativan broj.

Ako je novi broj nenegativan, petlja kreće ispočetka - računa i ispisuje korijen i traži novi broj. Petlja će se, dakle, izvršavati sve **dok** se ne upiše negativan broj.

Pojašnjenje:
* funkcija **sqrt** uzima vrijednost tipa **double** ili **float** i vraća vrijednost tipa **double**, a mi želimo da uzme vrijednost tipa **int** i vrati vrijednost tipa **float**. Zato smo unutar zagrada gdje se piše argument funkcije napisali **(float)broj** umjesto samo **broj**; operator **(float)** pretvara slijedeću varijablu u vrijednost tipa **float** koju funkcija **sqrt** prihvaća. Na sličan način smo umjesto **sqrt((float)broj)** napisali **(float)sqrt((float)broj)**; funkcija **sqrt** vraća vrijednost **double**, a u varijablu **korijen** želimo spremiti vrijednost **float**, pa koristimo operator **(float)** koji pretvara tu vrijednost u vrijednost tipa **float**. Ovo se zove type casting, tj. mijenjanje tipa.
Primjetimo: **double** je tip realnobrojne varijable s velikom preciznošću. Ako želimo učitati **double** vrijednost, u funkciji **scanf** pišemo **%lf**, dok pri ispisivanju **double** vrijednosti u funkciji **printf** koristimo **%f**, kao što je navedeno u poglavlju 'Varijable'.

6. Polja

Općenito, čim korisnik upiše neku vrijednost u varijablu, možemo s tom vrijednošću u varijabli napraviti što želimo i ispisati tu varijablu kao rezultat. Pri tome se vrijednost varijable mijenja. Što ako, s druge strane, želimo s tom početnom vrijednošću još nešto napraviti? Da bismo ju više no jedanput koristili, moramo ispisati nekakav rezultat bez mijenjanja vrijednosti varijable.
To nije problem, jednostavno ćemo svugdje pisati konačnu formulu koja ovisi o našoj varijabli, dakle nećemo mijenjati vrijednost varijable u svrhu nekakvih međukoraka.

Novi problem: što ako želimo napraviti jednu stvar s velikim brojem vrijednosti, i tek onda drugu stvar s istim tim vrijednostima? To znači da ne smijemo izgubiti početno upisane vrijednosti. Trebat ćemo deklarirati mnogo varijabli za to. Stotine? ....
Tu dolaze polja. Polje je varijabla koja u sebi može spremiti jako puno vrijednosti. Koliko? Koliko god želimo, ovisno o tome kako ju deklariramo. Naravno da u stvarnosti postoji granica broja vrijednosti, no time se ne treba zamarati za sada.
Pogledajmo primjer deklaracije polja.
int a[20];
Na ovaj smo način deklarirali polje s 20 cjelobrojnih vrijednosti (od 0 do 19) kojima pristupamo pomoću naziva polja: **a**.
Recimo da želimo zbrojiti prva tri elementa ovog polja i spremiti ih u četvrti. To radimo na slijedeći način.
a[3]=a[0]+a[1]+a[2];
Prisjetimo se: prvi element polja je a[0], drugi je a[1], i tako dalje, dvadeseti je a[19].
Pogledajmo primjer zadatka s poljima.
Trebamo napisati program koji učitava **n** vrijednosti, najviše 200 njih, i prvo ispisuje sve kvadrate, a potom sve korijene tih brojeva.

```
#include<stdio.h>
#include<math.h>
int main(){
	int i, n, a[200];
	printf("Koliko brojeva? ");
	scanf("%d", &n);
	for(i=0; i<n; i++){
		printf("\nUpiši %d. broj: ", i+1);
		scanf("%d", &a[i]);
		printf("\n%d", a[i]*a[i]);
```

```
	}
	for(i=0; i<n; i++)
		printf("\n%f", sqrt((float)a[i]));
	return 0;
}
```

Dopustili smo da korisnik izabere broj brojeva s kojima radimo: **n**.
Imamo **for** petlju koja ide od **0** do **n** (ne uključujući **n**), učitava svih **n** brojeva i sprema ih redom u polje. U svakom koraku petlje se, vidimo, učitava broj i ispisuje njegov kvadrat.
Imamo **i+1** u **printf** funkciji. Zašto? Zato što **i** ide od **0** do **n**-1 (jer je uvjet u **for**-u **i<n**), a mi želimo korisniku reći da upisuje brojeve od **1** do **n**, tako da svaki **i** povećavamo za **1**.
a[i] je način pristupanja polju **a**. No kojem elementu? Onom koji je unutar **[** i **]**, tj. **i**.
Nakon prve **for** petlje gdje se učitavaju brojevi i ispisuju svi kvadrati imamo drugu **for** petlju koja ispisuje sve korijene tih brojeva.
Primjetimo da za drugu **for** petlju ne trebamo pisati vitičaste zagrade jer se ponavlja samo jedna naredba. Ista stvar vrijedi i za uvjetovanje **if**. No da smo i pisali te zagrade, nije greška; to smo radili u prethodnim primjerima.
Također primjetimo da nismo mogli ispisivati korijene u prvoj **for** petlji. Tj. mogli smo, no zadatak je zadan tako da prvo ispišemo sve kvadrate, a tek onda sve korijene, a ne kvadrate i korijene naizmjenično.

7. Pokazivači

Pokazivači ili pokazivačke varijable su varijable koje sadžavaju adresu neke druge varijable. Pomoću pokazivača varijablama se može pristupiti i mijenjati vrijednost bez korištenja naziva samih varijabli. 'Čemu služe pokazivači?' je dobro pitanje. ~Jedina svrha pokazivača je mogućnost vraćanja više varijabli (ili polja) iz funkcije. Štoviše, jedini način da se u funkciji uopće koristi polje definirano u nekoj drugoj funkciji je preko pokazivača. Više o funkcijama u poglavlju 'Funkcije'. Sada ćemo samo vidjeti kako se koriste pokazivači tamo gdje nisu nužni u svrhu primjera.

U kontekstu pokazivača imamo dva operatora: ***** i **&**.
***** je operator koji:
- u slučaju deklaracije definira da je varijabla pokazivačka
- u bilo kojem drugom slučaju traži vrijednost s neke adrese

& je operator koji:
- traži adresu neke varijable

Pogledajmo deklaraciju pokazivača.
Recimo da deklariramo cjelobrojnu varijablu:
int a;
Sada kad **a** postoji, možemo deklarirati pokazivač na tu varijablu.
int *pok_a;
pok_a je pokazivačka varijabla koja za sada ne pokazuje ni na što. Pridružimo joj vrijednost:
pok_a=&a;

Pogledajmo što se sve ovdje dogodilo.
int a; je standardna deklaracija cjelobrojne varijable **a**.
int *pok_a je deklaracija pokazivača naziva **pok_a**; zbog ključne riječi **int**, **pok_a** smo deklarirali kao pokazivač na cjelobrojnu varijablu, a ***** označava da je ova varijabla pokazivačka varijabla.
& je operator koji traži adresu neke varijable, u našem slučaju varijable **a**.
Linijom **pok_a=&a;** smo, dakle, pokazivaču **pok_a** pridružili adresu varijable **a**.

Primjetimo da smo ova tri reda mogli napisati na slijedeći način:
int a, *pok_a=&a;

Napomena: korištenje varijable u naredbi u kojoj je deklarirana (osim za pridruživanje vrijednosti dotičnoj) je opasno jer nekad nije dopušteno. Dakle prethodni jedan red je uglavnom bolje pisati u dva reda;
int a;
int *pok_a=&a;

Nadalje, varijabli **a** možemo pridružiti vrijednost na dva načina sada kada imamo njezin pokazivač:
a=5;
ili
*pok_a=5;.
Rekli smo, ***** je operator koji ima dvije svrhe. U ovom slučaju ***** traži vrijednost na nekoj adresi. Na kojoj adresi? Na adresi **pok_a**. Ali **pok_a** je adresa varijable **a**! Upravo tako!
S obzirom da je **pok_a** zapravo adresa varijable **a**, linija ***pok_a=5;** će naći vrijednost na adresi od **a** i tamo spremiti vrijednost **5**.
Dakle te dvije linije rade istu stvar. Vraćamo se na pitanje: 'Čemu služe pokazivači?' Odgovor je isti: služe ~samo za rad s više varijabli u funkciji deklariranih drugdje. Stvar će se razjasniti u poglavlju 'Funkcije'. Za sada pogledajmo primjer programa s pokazivačima.
Ovaj program učitava cijeli broj od korisnika i ispisuje kub tog broja koristeći samo pokazivače.

```
#include<stdio.h>
int main(){
	int n;
	int *pok_n=&n;
	printf("Upiši broj: ");
	scanf("%d", pok_n);
	printf("\n%d", (*pok_n)*(*pok_n)*(*pok_n));
	return 0;
}
```

int n i **int *pok_n=&n;** su standardne deklaracije varijabli.
Vidimo da u funkciji **scanf** trebamo adresu varijable **n**, tako da možemo pisati **&n** ili **pok_n**.
Na kraju, pri ispisivanju kuba u funkciji **printf** trebamo vrijednost varijable **n**, odnosno trebamo vrijednost na adresi **pok_n**. Do te vrijednosti možemo doći na dva načina; preko pokazivača **pok_n** ili preko same varijable **n**. Pa, zadatak je zadan tako da smijemo koristiti samo pokazivač na varijablu, a ne i samu varijablu, tako da do broja dolazimo upisivanjem ***pok_n** u naš program, i onda množeći te vrijednosti same sa sobom da bismo došli do kuba.
Napomena: jedina svrha pokazivača je rad s više varijabli u funkciji deklariranih drugdje Za Sada, kasnije će se koristiti i za druge stvari poput rada s datotekama. No za početak to nije potrebno znati.

8. Funkcije

Na funkcije možemo gledati kao na zasebne programe. Možemo im dati vrijednosti s kojima će nešto raditi i vratiti rezultat, mogu one same ispisivati nešto, a mogu i napraviti nešto bez vraćanja vrijednosti.
Postoji nekoliko tipova funkcija; tip funkcije ovisi samo o tipu vrijednosti koju funkcija vraća. Ako funkcija ne vraća vrijednost, onda je tipa **void**.
Deklaracija funkcije izgleda ovako:

```
int funkcija(int a){
	neke naredbe;
```

```
	return a;
}
```

int na početku označava tip funkcije.
funkcija je ime funkcije i može biti bilo koje osim **main**; **main** se zove samo glavna funkcija i ni jedna druga. Naravno, i općenito dvije funkcije ne mogu dijeliti jedno ime.
Unutar zagrada popisujemo argumente funkcije. U ovom primjeru, funkcija **funkcija** prima jedan cjelobrojni argument kojem ćemo se u ovoj funkciji obraćati s **a**.
{ označava početak funkcije.
return a; je naredba koja vraća vrijednost varijable **a** pozivnoj funkciji, tj. onoj funkciji koja je pozvala ovu funkciju. Tome služi i **return 0;** u **main**-u u svakom programu do sada.
} označava kraj funkcije.

Pogledajmo primjer zadatka:
ovaj program funkciji predaje cjelobrojni argument, a ona vraća realni kvadrat tog broja.

```
#include<stdio.h>
float kvadrat(int broj){
	float kv;
	kv=(float)broj*broj;
	return kv;
}
int main(){
	int a;
	printf("Upiši broj: ");
	scanf("%d", &a);
	printf("\n%.2f", kvadrat(a));
	return 0;
}
```

float je tip funkcije jer ona vraća **float** vrijednost (**kv**).
kvadrat je naziv funkcije.
int je tip varijable koju funkcija prima.
broj je naziv varijable koji smo u ovom trenutku deklarirali u funkciji **kvadrat**, i dali mu vrijednost koju je funkcija **main** poslala. Drugim riječima, to je vrijednost koju funkcija **kvadrat** prima od **main**-a. Ako funkcija prima više argumenata istog tipa, nije dovoljno napisati **tip varijabla 1, varijabla 2**, nego **tip varijabla 1, tip varijabla 2**. Tj. nije dovoljno napisati, npr., **int a, b** nego **int a, int b**.
Unutar **{** i **}** je tijelo funkcije.
float kv; je linija koja na standardan način deklarira realnobrojnu varijablu **kv**.
U liniji **kv=(float)broj*broj;** smo varijabli **kv** pridružili kvadrat vrijednosti primljene iz **main** funkcije. Operator **(float)** ovdje koristimo jer želimo u **kv** spremiti realnu vrijednost; da bi rezultat operacije bio realan broj barem jedan od operanada mora biti realan broj, zato smo prvi **broj** pretvorili iz cijele u realnu vrijednost. Mogli smo ovu liniju napisati i ovako: **kv=broj*(float)broj;**
return kv; vraća pozivnoj funkciji (**main**) realnobrojni rezultat.
U funkciji **main** imamo sve standardne naredbe, i poziv funkcije unutar **printf**-a.
Vidimo da poziv funkcije izgleda tako da nakon imena funkcije u zagrade popisujemo varijable koje joj šaljemo, inače odvojene zarezom, i funkcija ih tim redoslijedom prima.
Mogli smo u **main**-u imati posebnu varijablu, npr., **kvadrat** i napisati
kvadrat=kvadrat(a);
printf("\n%.2f", kvadrat);,
no to nije potrebno jer na mjestu poziva funkcije unutar **printf**-a treba ići broj (tj. varijabla), a ta

funkcija vraća baš taj broj koji nam treba, tako da smijemo pisati na ovakav način.
%.2f pišemo umjesto **%f** da označimo da želimo ispisati samo dvije decimale tog broja, a ne svih sedam.

Za kraj prvog dijela potrebno je reći da smo mogli napisati program s funkcijom i na drugačiji način. Cijelo tijelo funkcije smo mogli napisati iza (ispod) **main** funkcije, ali bismo trebali napisati njezin prototip iznad. Prototip funkcije je deklaracija funkcije sa svim vrijednostima koje prima, ali bez naredbi.
Program bi, dakle, mogao izgledati ovako:

```
#include<stdio.h>
float kvadrat(int broj);
int main(){
	int a;
	printf("Upiši broj: ");
	scanf("%d", &a);
	printf("\n%.2f", kvadrat(a));
	return 0;
}
float kvadrat(int broj){
	float kv;
	kv=(float)broj*broj;
	return kv;
}
```

Vidimo da je **float kvadrat(int broj);** prototip funkcije kvadrat. Ovime smo kompajleru rekli da postoji ta funkcija, samo ju još nismo definirali. Na kraju prototipa funkcije se također piše **;**.
Radi se o tome da kompajler čita program od gore prema dolje tako da ne smijemo imati funkciju ispod **main** funkcije; kako će kompajler moći ući u neku funkciju, odnosno izvršiti ju, ako ju još nije registrirao? Zato se sve funkcije pišu iznad **main** funkcije.
Jedini način da zaobiđemo ovo pravilo je, dakle, da prototip funkcije napišemo iznad mjesta gdje se koristi, dakle iznad **main**-a, time smo deklarirali funkciju i kompajler zna da postoji, a da je definirana drugdje. Izvanredno će otići do mjesta gdje je definirana i izvršiti ju normalno, i potom se vratiti na liniju u **main**-u ispod poziva te funkcije.

U drugom dijelu ovog poglavlja ćemo pogledati što se događa s poljima u funkciji.
Prije toga moramo definirati jednu važnu stvar: pokazivač na polje.
Pokazivač na varijablu funkcionira jednostavno: u sebi sadrži vrijednost adrese te varijable. Problem kod polja je što ono ima puno varijabli u sebi.
Jedan način da pokazujemo na sve te varijable je da napravimo polje pokazivača, npr.
int *pok[20];,
koje će imati 20 različitih pokazivača na 20 različitih varijabli. Da bismo pridružili sve te adrese svim tim pokazivačima bismo trebali **for** petlju, a i ne bismo ništa posebno mogli s tim kasnije napraviti. Zato se ovaj način ne koristi.
Dogovorni način je ovakav: pokazivač na prvi element polja možemo tretirati kao pokazivač na polje.
Pročitajmo to još jedanput: pokazivač na prvi element polja možemo tretirati kao pokazivač na polje.

Recimo da imamo polje:
int a[20];.
Sada deklariram pokazivač na to polje na slijedeći način.
int *pok=&a[0];

pok je sada pokazivač na prvi cjelobrojni element tog polja, i s tim pokazivačem mogu pristupiti bilo kojoj vrijednosti u polju, npr.
*(pok+6)=7;
Ovom sam naredbom postavio vrijednost **7** u sedmi element polja.
Prvo imamo ***** koja traži vrijednost na adresi. **pok** je adresa prvog elementa polja (odnosno **&a[0]**), a **pok+6** je ta adresa pomaknuta za još 6 mjesta (odnosno &a[6]), dakle sedmi element polja. **=7**, naravno, samo sprema vrijednost **7** u taj sedmi element polja.
Pogledajmo primjer programa koji 20 učitanih brojeva sprema u polje koristeći pokazivače.

```
#include<stdio.h>
int main(){
	int i, a[20];
	int *pok=&a[0];
	for(i=0; i<20; i++){
		printf("Upiši %d. broj: ", i+1);
		scanf("%d", pok+i);
	}
	return 0
}
```

Rekli smo, funkcija **scanf** treba adresu na koju će spremiti vrijednost, a **pok+i** joj upravo to daje; prvi će broj spremiti na adresu **pok+0**, dakle **&a[0]**, drugi na **pok+1**, dakle **&a[1]**, itd. Naravno da smo umjesto **pok+i** mogli pisati **&a[i]**, ali zadatak smo htjeli rješiti koristeći pokazivače.

Još jedna važna napomena: u trenutku obične deklaracije polja se deklarira i pokazivač na prvi element polja, i dijeli ime s poljem.
Recimo da deklariramo polje.
int a[20];
U ovom trenutku, **a** je pokazivač na prvi element tog polja, odnosno **a** je jednako **&a[0]**.
To znači da u prethodnom zadatku nismo morali imati pokazivačku varijablu **pok** koja će ispunjavati tu svrhu. Drugim riječima, zamijenimo li svaki **pok** s **a**, program će raditi na isti način uz manje zauzete memorije jer imamo varijablu manje.
Naravno da ne smijemo napraviti slijedeće.
int a[20];
int a;
Kompajler neće znati što je **a**: pokazivač na polje (točnije, na prvi element polja) ili obična cjelobrojna varijabla.

Sada kada znamo kako funkcioniraju polja, pokazivači, funkcije i pokazivači na polja, vraćamo se zadnji put na pitanje: 'Čemu služe pokazivači?'
Odgovor je ponovno da su pokazivači jedini način da u funkciji radimo s više varijabli koje su deklarirane drugdje. (Točnije, pokazivači su nužni (1) da bi funkcija vraćala više od jedne vrijednosti i (2) da bismo mogli poslati polje funkciji. Inače funkcija normalno može primiti više varijabli i s njima raditi, no samo jednu vrijednost može vratiti.)
Pogledajmo kako šaljemo polje funkciji.
To ćemo napraviti na način da u pozivu funkcije jedna od popisanih varijabli (odnosno jedna od varijabli čije vrijednosti šaljemo polju) bude pokazivač na polje, tj. na prvi element polja, i funkcija dalje s tim može normalno raditi.
Napišimo program koji učitava polje s 20 elemenata, i potom u funkciji ispisuje sve elemente tog polja odvojene zarezom.

```
#include<stdio.h>
void ispis(int *a){
	int i;
	for(i=0; i<20; i++)
		printf("%d, ", a[i]);
	return;
}
int main (){
	int i, a[20];
	for(i=0; i<20; i++){
		printf("\nUpiši %d. broj: ", i+1);
		scanf("%d", &a[i]);
	}
	printf("\n");
	ispis(a);
	return 0;
}
```

Funkcija **ispis** je tipa **void** jer ne vraća vrijednost pozivnom programu, tj. **main**-u.
int *a unutar zagrada u deklaraciji funkcije znači da, kao što smo rekli, funkcija prima pokazivač na cjelobrojnu varijablu. U ovom slučaju ta cjelobrojna varijabla je prvi element polja koje ima i druge elemente, i zato što postoje drugi elementi pomoću tog pokazivača **a** možemo svima njima pristupiti. Da je **a** obična cjelobrojna varijabla pomoću pokazivača bismo mogli pristupiti samo njoj.
U **printf** funkciji imamo **a[i]**. Na tom smo mjestu mogli napisati i ***(a+i)** ili ***(&a[i])**, jer operator * traži vrijednost na adresi, a u našem slučaju adresa je **a+i**, odnosno **&a[i]**.
return naredba ne vraća vrijednost jer naša funkcija ne treba vratiti nikakvu vrijednost **main**-u, samo treba ispisati neke brojeve. Uostalom, zato funkcija **ispis** i je tipa **void**.
U glavnoj funkciji imamo naredbu **printf("\n");** koja samo stavlja novi red nakon učitavanja brojeva zbog urednosti u ispisu.
Poziv funkcije se u ovom slučaju sastoji samo od **ispis(a)** jer ta funkcija ne vraća vrijednost; inače bismo imali **varijabla=ispis(a)**.
a unutar zagrada u pozivu funkcije je argument funkciji. Rekli smo, funkciji trebamo predati polje, odnosno pokazivač na polje, tj. pokazivač na prvi element polja, a to je upravo **a**. Tu smo mogli, naravno, napisati i **&a[0]**.
Još jedna stvar koju je dobro spomenuti je da smo u funkciji mogli raditi s bilo kakvim imenom varijable. Mogli smo, dakle, taj pokazivač nazvati **pok** da smo htjeli, i dalje normalno raditi s tim, dakle svugdje gdje je **a** bi bio **pok**. No logično ga je nazvati upravo **a** jer taj pokazivač sadrži adresu baš polja **a**.

Napomena: imamo varijablu **i** unutar funkcije **ispis** i unutar funkcije **main**. Jedna nema veze s drugom iako se isto zovu jer nisu u istoj funkciji. Dakle, teoretski, da u **main**-u imamo nekakvu for petlju s brojačem **i**, i da u njoj nekoliko puta pozivamo neku funkciju koja isto ima for petlju s brojačem **i**, sve bi radilo kako spada, tj. varijabla **i** u for-u u toj funkciji bi imala neku vrijednost koja nema veze s varijablom **i** u **main**-u.

9. Stringovi

O stringovima smo mogli pričati i ranije, no nije bilo potrebe za komplikacijama.
Jednostavno rečeno, string je znakovno polje, odnosno polje znakova.
Deklaracija stringa, dakle, izgleda ovako.
char a[20];
Postoji posebno poglavlje samo za stringove jer ima dosta manjih stvari koje je dobro znati s obzirom

da se stringovi stalno koriste.
Pogledajmo kako bismo učitali string.
Možemo imati **for** petlju koja učitava znak po znak, no mi želimo korisniku olakšati korištenje što više tako da želimo da on upiše cijeli string odjednom. To se radi funkcijom **gets**.

Imamo deklarirano znakovno polje **a**.
gets(a);
Ovo je funkcija koja čeka upis znakova, i sprema ih u polje **a**, dakle funkciji **gets** predajemo pokazivač na polje.
Na sličan način možemo ispisati sve elemente tog polja, odnosno cijeli string.
puts(a);
puts je, dakle, funkcija koja prima pokazivač na polje i ispisuje to polje.

Kako funkcija **puts** zna gdje je kraj polja? Odnosno što ako sam deklarirao znakovno polje za 20 znakova, a korisnik upiše samo 10? Kako onda **puts** zna gdje treba stati?
Događa se slijedeće.
Pri učitavanju stringa funkcijom **gets** na kraj polja se stavlja znak **\0**, a njegov cjelobrojni ekvivalent u ASCII tablici je **0**. Taj znak označava kraj stringa.
Funkcija **puts** jednostavno ispisuje znak po znak dok ne dođe do znaka **\0** i onda stane.
Pogledajmo jednostavan primjer.

```
#include<stdio.h>
int main(){
	char a[20];
	printf("Upiši string: ");
	gets(a);
	printf("\n");
	puts(a);
	return 0;
}
```

Ovaj program učita string od najviše 19 znakova (jer je potrebno jedno mjesto za **\0**)
i ispiše ga u novom redu. Primjetimo, dakle, da ako tražimo određeni string s nekim brojem znakova polje mora imati jedno mjesto više za terminirajući (terminating) nul-znak (nul-character).
Ima još jedan način ispisivanja stringova.
printf("%s", a);
Ova naredba radi istu stvar kao i **puts(a);**, dakle i s **printf**-om se može ispisati string.

gets i **puts** su dvije naredbe koje služe za rad sa stringovima, a dostupne su u library-ju **stdio.h**.
Općenito, funkcije za rad sa stringovima se nalaze u library-ju **string.h**. Ovdje nećemo nabrajati sve te funkcije i njihove svrhe, pogledajmo samo jedan primjer.

Napišimo program koji ovisno o učitanom stringu pokreće jednu od tri funkcije na slijedeći način.
Ako je upisan 'Pariz', pokreće se funkcija **Pariz**, ako je upisan 'London', pokreće se funkcija **London**, a ako je upisan 'Zagreb', pokreće se funkcija **Zagreb**.

```
#include<stdio.h>
#include<string.h>
void Pariz(){
	printf("\nNalaziš se u Parizu.");
	return;
```

```
}
void London(){
	printf("\nNalaziš se u Londonu.");
	return;
}
void Zagreb(){
	printf("\nNalaziš se u hrvatskom gradu, Zagrebu.");
	return;
}
int main(){
	char mjesto[20];
	printf("Kamo želiš ići? ");
	gets(mjesto);
	if(strcmp(mjesto, "Pariz")==0)
		Pariz();
	else if(strcmp(mjesto, "London")==0)
		London();
	else if(strcmp(mjesto, "Zagreb")==0)
		Zagreb();
	else
		printf("\nUpis nije dobar.");
	return 0;
}
```

Sve tri funkcije su tipa **void**, ne vraćaju nikakvu vrijednost.
strcmp je funkcija unutar library-ja **string.h** koja služi za uspoređivanje stringova (**strcmp** = string comparison). Ta funkcija prima dva stringa, vraća **0** ako su stringovi identični, ili nešto različito od **0** ako su različiti.
Drugim riječima, ako korisnik u polje **mjesto** upiše upravo, npr., 'London', usporedba sa stringom **London** će vratiti **0**, tako da ćemo zapravo imati **if(0==0) London();**
S obzirom da je **0** stvarno jednako **0**, idemo u London.

Još jedna nova stvar ovdje je **else if**. Imamo slijedeće.
Ako je upisan **Pariz**, idemo u **Pariz**. Nakon toga imamo još jedno uvjetovanje, ovaj put s ključnom riječju (ključnim riječima) **else if** za **London**, i nakon toga još jednom tako za **Zagreb**. **else if** označava da se **else** na kraju odnosi na svaki od tih **if**-ova. Drugim riječima, ako ni jednom **if**-u nije ispunjen uvjet izvršavaju se naredbe od **else**-a, to je u ovom slučaju samo naredba **printf**. Da je svugdje gdje imamo **else if** samo **if**, **else** bi se odnosio samo na zadnji **if**. U tom bi se slučaju skup naredbi **else**-a izvodio kada god nije upisan 'Zagreb', a naravno da to ne želimo.

Vidimo da dok se pojedinačan znak piše s apostrofima ('a'), string se piše s navodnicima ("...").
Na kraju, ako nije upisano ni jedno mjesto pravilno korisnik dobiva poruku da mu upis nije dobar. Da smo htjeli, mogli smo napraviti **while** petlju koja čeka pravilan upis, i kad upis bude dobar pozove funkciju, a dok upis nije dobar ponavlja istu poruku: npr. "Upis nije dobar. " ili "Ponovi upis: "
Ovaj zadatak je ujedno i osnova tekstualne video igre.

10. Završno slovo

'Je li programiranje teško?' je vječno pitanje sa složenim odgovorom. Radi se o slijedećem. S obzirom da smo različiti ljudi, na različite načine razmišljamo. Konkretno za kontekst programiranja, netko je racionalniji, netko iracionalniji; netko doslovnije razmišlja, netko manje doslovno; netko voli objašnjavati sve na najtočniji način, netko prihvaća stvari onakve kakvima se

čine. Za svaku od ove tri usporedbe vrijedi da je prvi slučaj bolji za programiranje (no ne i za život općenito :).
Često kažemo 'Računalo je glupo.' Misli se na činjenicu da računalo ne razumije implikacije, sve mu se mora točno definirati. Ako nismo navikli na takvo razmišljanje teško ćemo moći pretočiti misli u programski jezik. Ako, s druge strane, inače objašnjavamo sve što više i što je točnije moguće i samo se služimo logikom programiranje će biti samo drugi način govora, a učenje programiranja kao učenje drugog narječja.

Ovaj je tutorial napravila lijena osoba za druge lijene ljude; ovaj tekst je dovoljan da se napravi većina zadataka kolegija "Programiranje i Programsko Inženjerstvo" na FER-u u Zagrebu, barem u prvom ciklusu, uz, naravno, vježbanje zadataka koji nisu dostupni u ovom tekstu.
Ovaj je tekst dobro proći s razumijevanjem dva puta da bi se potpuno shvatilo gradivo. Nakon toga zadaci se lako mogu riješiti počevši od jednostavnih, pa sve do složenih.

Nevezano za studij ovaj tekst je dovoljan da se počne raditi jednostavan koristan software za pravu uporabu pod uvjetom da se zna kompajlirati kôd i napraviti .exe datoteku iz njega. Može se, npr., isprogramirati svaki algoritam od kalkulatora do Rubic's cube solvera.

Na internetu se može saznati doslovno sve vezano za programiranje. Štoviše, ako se nema pristup internetu, ne treba se ni započinjati projekt.

11. Literatura

Internet.