Srpski, Školski C tutrial

Autor: Nemanja Todić

Sadržaj

Uvod

UVOD

Programski jezik C je konzolni sto ce reci da preko njega ne mozete(bez koriscenja nekih „dodataka“) napraviti „prozolike“ aplikacije, no ovo ne treba da vas razocara u dogledno vreme se mozete i time pozabaviti.
C je i strukturni jezik dakle koristi strukture kao primaran vid izvedenih tipova podataka. Ovaj sistem je zastareo i jezici nove generacije su objektno orijentisani(OOP) no i takvim jezicima strukture nisu nepoznate. Ne ocekujem da ako ste pocetnik ovo sada shvatite, shvaticete polako kako budete napredovali....
Treba da znate da je svaki pocetak suvoparan i da je za ucenje bilo kog programskog jezika potrebno dosta vremena truda i pomalo pameti. Ne daj te se obeshrabrii!
Pa da pocnemo.

1.0 Brojevni sistemi

Postoji nekoliko brojevnih sistema a „najpoznatiji“su: decimalni, oktalni, binarni i hekasadekadni. Ono sto vi treba da znate je kako se iz jednog brojevnog sistema konvertuje broj u drugi brojevni sistem.

1.1 Decimalni brojevni sistem

Decimalni BS je skup koji ima 10 cifara i to su sledece : 0,1,2,3,4,5,6,7,8,9. Pravljenjem razlicitih kombinacija mozete dobiti brojeve kao sto su -789, 0 , 1, 49, 32757 idr. Evo primera kako se na drugaciji nacn moze predstaviti broj npr 385:
$3 * 10^2 + 8 * 10^1 + 5 * 10^0 = 3 * 100 + 8 * 10 + 5*1 = 300 + 80 + 5 = 385$.
Provezbajte sa jos par brojeva.

1.2 Oktalni brojevni sistem

Oktalni BS je skup koji ima 8 cifara i to su : 0,1,2,3,4,5,6,7. Bitno je znati kako da brojeve ovog sistema prevedete u decimalne brojeve. Evo primera: oktalno 142 je decimalno 98 :
$|142|_{8} = 1 * 8^2 + 4 * 8^1 + 2 * 8^0 = 1*64 + 4*8 + 2*1 = 64 + 32 + 2 = |98|_{10}$.
Poprilicno jednostavan sistem.

1.3 Binarni brojevni sistem

Binarni BS je skup koji ima 2 cifre i to su : 0, 1 , te ce binarni broj 101100 u decimalnom zapisu biti:
$|101100|_2 = 1*2^5 + 0*2^4 + 1*2^3 + 1*2^2 + 0*2^1 + 0*2^0 = |44|_{10}$
Vrlo slicno prethodnim primerima.

1.4 Hekasadekadni brojevni sistem

Ima cak 16 simbola : 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F. Bice vam potrebno da znate koje vrednosti imaju navedena slova u decimalnom zapisu:
A = 10
B = 11
C = 12
D = 13
E = 14
F = 15

Primer:
$|A2F|_{16} = 10 * 16^2 + 2 * 16^1 + 15 * 16^0 = 2560 + 32 + 15 = |2607|_{10}$
Ovo je malo komplikovanije ali bitno je uvideti da se slova zamenjuju odgovarajucim vrednostima.

1.5 Konverzija iz decimalno u binarni zapis

Bicete u prilici da vrsite ovakvu konverziju. Obajsnicu na sledecem primeru: Broj 44 prevesti u binarni zapis. Resenje:

$\|44\|_{10}$ =	44/2	22/2	11/2	5/2	2/2	1/2
	0	0	1	1	0	1

Zadati broj 44 delimo sa dva, ostatak zapisujemo ispod a rezultat deljena pisemo levo i taj rezultat delimo sa dva i tako dalje dok god se ne dobije nula kao kolicnik(ne kao ostatak). Sada obratite paznju na dobijeni od ostataka i procitaj te ga odpozadi:
101100. Dosli smo do resenja! Pokusajete kao proveru da uradite konvertovanje ovog binarnog broja u decimalni.

2.0 Jezicki tipovi podataka

Postoji nekoliko „ugradjenih“ tipova podataka u jeziku C medjutim ostavljena je mogucnost da se na vise nacina naprave novi tipovi podataka, njih konstruise programer. Izvedeni tipovi su krajnje korisni ali isto tako i malo kompleksniji za osmisljavanje pa ce mo se za sada zadrzati na nekim osnovnim tipovima podataka.

2.1 Promenljive i konstante

Recimo da zelite da napisete program koji ce da sabere dva broja koje ce korisnik uneti. Proces rada programa bi isao ovako: korisnik unosi prvi broj, program taj broj sacuva, zatim korisnik unosi drugi broj i program taj broj takodje sacuva. Postavlja se pitanje gde taj broj sacuvati? Odgovor je u Random Access Memory(RAM) odakle cemo broj po potrebi iscitavati. Ovo se realizuje tako sto cemo „napraviti“ promenljivu(takodje se nazivaju i variable) koja ce zauzeti odgovarajuce parce memorije, a velicina tog parceta zavisi od tipa promenljive. Oko tipova podataka i njihovim karakteristikama(a jedna od njih je i kolicina zauzete memorije) cemo se pozabaviti kasnije.
Bitno je da uvidite da se u promenljivu mogu upisivati i iscitavati podaci sve dok ta promenljiva postoji u memoriji. Vratimo se zadatku, dakle ucitali smo dva broja koja valja sabrati. Sledece, treba nam mesto u memoriji racunara u kojem ce mo snimiti rezultat

sabiranja, dakle treba nam jos jedna promenljiva. Njoj cemo dodeliti vrednost zbira prve dve promenljive koje smo ucitali. To je to. Trebalo bi da vam je jasno sta su promenljive i cemu sluze. Sada cemo objasniti i kako se koriste.
Dve su bitne stvari kod kreiranja(pravilnije receno „instancovanja“) promenljivih: njihova deklaracija i inicijalizacija. Recimo da zelimo da napravimo promenljivu u koju cemo smestiti prvi broj iz gore navedenog zadatka, to cemo uraditi ovako:

```
int a;
```

Ovim smo deklarisali promenljivu koja ima tip „int“ (sto ce reci da moze cuvati samo cele brojeve, ovim cemo se kasnije pozabaviti) i naglasili da se ona zove „a“. Pogledajmo sledeci primer:

```
int a = 157;
```

dodali smo „ = 157 “. To je inicijalizacija iliti dodavanje vrednosti promenljivoj u isto vreme kada je i deklarisemo(napravimo). Ukoliko neinicializujemo promenljivu ona dobija neku bezveznu vrednost, no u svim drugim pogledima ponasanje inicijalizovane i neinicijalizovane promeljive je potpuno isto. Postoje i globalne promeljive kojima se deklaracijom automatski dodeljuje vrednost 0, no otom potom.

-Konstante su takodje vid promenljivih ali se njima moze vrednost dodeliti samo jednom, prilikom inicijalizacije. U daljem toku programa njihova vrednost se samo moze iscitavati. Deluje beskorisno? Pa i nije bas, recimo da u programu koristite broj Pi, on ima konstantnu vrednost 3.14 te ce mo promeljivu Pi definisati tako da se njena vrednost ne moze menjati. Ovako:

```
const int Pi = 3.14;
```

Bitno je uvideti da konstante moraju biti inicijalizovane, u suprotnom nema svrhe koristiti ih.

2.2 Pravila za dodeljivanja imena promeljivima

Promenljiva moze sadrzati proizvoljno ime ali ono treba da opisuje namenu promenljive, dakle ako treba da cuva zbir nekih brojeva verovatno ce te je nazvati „zbir“ ili „rezultat“. Imena promenljivih smeju da sadrze slova, brojeve i znak donje crte(_). Na pocetku imena se mora naci ili slovo ili donja crta, broj ne sme!

2.3 Decimalni tip – INT

Ovaj tip je najkorisceniji tip podatka. Promenljive koje su tipa int(skracenica od integer) u 32bit-nom sistemu zauzimaju 4 bajta i u takvim promenljivim mozete cuvati decimalne cele brojeve kao sto su 0, 789, 4000, -1, -4569 isl. Primer deklaracije ovakve promenljive :

```
int broj1;
int broj2 = 150;
```

2.4 Realni tip – FLOAT I DOUBLE

Nedostatak tipa int je u tome sto ne moze cuvati vrednosti koje imaju decimalnu tacku odnosno „zarez“(dakle realne brojeve) vec INT sve cifre „iza zareza“ odbacuje. Da bi cuvala ovakve brojeve promenljiva mora biti tipa float ili ,ako je broj enormno veliki, tipa double. Primer deklaracije ovakvih promenljivih:

```
float x;
double y;
```

2.5 Kljucne reci SHORT i LONG

Ove kljucne reci mogu se primeniti samo na do sada opisane tipove podataka. Ako deklarisemo promenljivu X kao „short int“ ona ce sadrzati (u 32bit-nom) sistemu 2 bajta, dakle duplo manje od regularnog int-a. Medjutim ona moze sadrzati brojeve ciji su opstezi takodje duplo manji. Kljucna rec long omogucava da promenljiva ima veci opseg od onog koji bi ta promenljiva imala bez ove kljucne reci.

2.6 Znak – CHAR

Gore pomenuti tipovi su cuvali brojeve a promenljive koje su tipa char cuvaju simbole. Svaka promenljiva tipa char zauzima svega jedan bajt i moze da cuva samo jedan simbol. Simbola ima ukupno 255 i tu spadaju kompletna engleska abeceda, brojevi i razni drugi simboli i znakovi. O ovom tipu podatka cemo detaljnije pricati kasnije.

3.0 Operatori

Verovatno vam je poznat termin „operator“, dakle to je npr. +, -, / itd... Jezik C ima ogroman skup operatora ali to ga uopste ne cini konfuznim, kao sto ce te i videti. Postoji par bitnih stvari : prvenstvo operatora, asocijativnost, znacenje itd...
Operatore poredjenja ovde necemo objasnjavati, to cemo uraditi kasnije, priliko objasnjavanja „grananja koda“.

3.1 Izrazi i naredbe

Ranije smo naveli primer u kojem smo sabirali dva broja : a + b. Rezultat smo smestali u promenljivu rez, ako od ovih podataka formiramo izraz u jeziku C dobija ovo:

```
rez = a + b;
```

Izvedimo konstataciju da je izraz kombinacija operanada i operatora. U navedenom primeru imamo 3 operanda(rez, a, b) i dva operatora(=, +). Takodje postoje i dva izraza, prvi je sabiranje promenljive a i b, drugi je dodeljivanje te vrednosti promenljivoj rez. Sve ovo zajedno cini jednu naredbu. Naredba je dakle skup izraza(ciji broj moze biti ne ograniceno veliki ali takodje moze biti i 0- prazna naredba) koji se obavezno okoncava znakom „tacka-zarez“(;).

3.2 Operatori u jeziku C

Vazan faktor je svakako prvenstvo operatora, sto znaci da ako napisete naredbu :

```
rez = a -  b * 2
```

prvo ce se izvrsiti izraz b * 2 pa ce se od a oduzeti rezultat mnozenja i na kraju se ta razlika dodeljuje promenljivoj rez. Uvidjamo da operator mnozenja ima najvece prvenstvo, zatim operator oduzimanja i da najmanje prvenstvo ima operator dodele vrednosti. Sledi tabela prvenstva operatora.

OPERATOR	OPIS
()	Poziv funkcije
[]	Index niza
->	Pokazivac na strukturu
.	Clan strukture
-	Unarni minus
+	Unarni plus
++	Inkrementiranje
--	Dekrementiranje
!	Logicka negacija

~	Komplemant nad bitovima
*	Posredan pristup
&	Adresa-od
sizeof	Velicina objekta
(type cast)	Eksplicitna konverzija
*	Mnozenje
/	Deljenja
%	Ostatak(moduo)
+	Sabiranje
-	Oduzimanje
<<	Pomeranje bitova ulevo
>>	Pomeranje bitova udesno
<	Manje od
<=	Manje ili jednako
>	Vece od

OPERATOR	OPIS
>=	Vece ili jedanko
==	Jednako
!=	Razlicito
=	*
+=	*
-=	razni
*=	operatori dodele
/=	vrednosti
%=	*
&=	*
sizeof	*
(type cast)	*
, (zarez)	Redosle

Ako u navedenom izrazu zelite da prvo odradite sabiranje morate upotrebiti zagrade:

```
rez = (a + b) * 2;
```

Ili mozete da napisete u vise naredbi:

```
rez = a + b;
rez = rez * 2;
```

ovo je prosto i jasno. Medjutim postoji „skracenica“ koju mozemo primeniti na izraz rez = rez * 2, umesto da dva puta pisemo „rez“ napisacemo izraz:

```
rez *= 2;
```

sto ce imati isti efekat. U tabeli mozete videti gomili operatora koju se zasnivaju na ovom principu.

Mozda niste upoznati sa znacenjem operatora ostatka(moduo)- %. Ukoliko bi imali izraz:

```
rez = 10 / 2;
```

rezultat bi bio 5 i taj broj bi se dodelio promenljivoj rez. Ostatka pri deljenju 10 sa 2 nema pa je on nula. No, ako zelimo da rez sadrzi ostatak od 10 / 2 koristicemo operator moduo:

```
rez = 10 % 2;
```

rez ce imati vrednost 0.

3.3 Gubitak preciznosti

U poglavlju 2 receno je koji tipovi promenljivih mogu cuvati koje podatke. Ako imamo promenljivu X tipa int i promenljivu Y tipa float te napisemo sledece:

```
int X;
float Y = 3.14;
X = Y;
```

vrednost koju cuva X nece biti 3.14 jer je X tipa int a taj tip ne moze da cuva brojeve sa decimalnom tackom(realne brojeve). X ce imati vrednost od 3, dakle deo posle decimalne tacke se odpacuje i dolazi do gubitka preciznosti.
Postoji veoma bitna stvar koja moze biti vrlo ne zgodna. Pogledajmo primer:*;*

```
float Y;
Y = 10 / 4;
```

ocekivano je vrednost Y 2.5 ali nije! Vrednost Y je u stvari 2! Zasto? Analizirajmo naredbu Y=10/4. Prvo se obavlja deljenje pa tek onda dodeljivanje vrednosti. Kada se deljenje obavlja sistemu je potrebna memorija u koju ce privremeno smestiti vrednost deljenja pre nego sto tu vrednosti dodeli promenljivoj Y, dakle sistem ce kreirati privremenu promenljivu.Treba se zapitati kojeg je tipa ta promenljiva. Radi optimizacije iskoriscenja memorije, sistem ce kreirati promenljivu onog tipa u koji ce mocu da stane rezultat deljenja(sabiranja,mnozenja...) drugim recima ako su oba operanda tipa int(sto je ovde slucaj) sistem pravi promenljivu tipa int i dodeljuje joj odgovarajucu vrednost, a posto int ne moze cuvati brojeve iza decimalno zareza(bez obzira sto je Y tipa float jer se njemu vrednost tek kasnije dodeljuje) dolazi do gubitka preciznosti.
Dva su nacina za izbegavanje ovakvog ponasanja. Prvi, mozemo uraditi ovo:

```
Y = 10 / 4 * 1.0;
```

dodali smo „ * 1.0 “ izrazu i time smo postigli da postoje dve promenljive tipa int (10, 4) i jedna tipa float(1.0). Pomocna promenljiva koju je za sebe kreirao sistem ce samim tim biti tipa float pa nece biti gubitka preciznosti. Sve pomocne promenljive sistem automatski brise po zavrsetku naredbe i tako vraca zauzetu memoriju.
Drugi nacin je elegantniji, koristicemo operator eksplicitne konverzije(gore pomenuta je bila inplicitna) takodje zvanog „cast operator“. Koristeci ovaj nacin dobicemo naredbu:

```
Y = (float) 10 / 4;
```

Uz pomoc (float) smo promenili tip broja 10 iz int u float. Unutar zagrada mozete stavljati razlicite tipove i na taj nacin dolaziti do razlicitih konverzija.
Sve ove ce te moci da probate na delu, kada budemo poceli da pisemo programe a to ce biti od sledecih poglavlja.

3.4 Operatori inkrementiranja(++) i dekrementiranja(--)

Imamo promenljivu X kojoj smo dodelili vrednost 15.

```
int X = 15;
```

Zelimo da joj povecamo vrednost za 1(mozda deluje sumanuto ali veoma cesto ce te se sretati sa ovakvom potrebom). To mozemo uraditi na nekoliko nacina:

```
X = X + 1;
X +=1;
X++;
```

Prva dva izraza su vam poznata ali treci izraz je novi, to je takozvana inkrementacija. Ona znaci da ce se vrednosti promenljive X povecati za 1, da stoji X-- vrednost bi se smanjila za 1. Uvedena je jer je mnogo lakse/brze napisati X++ nego X = X + 1, zar ne? Medjutim operator ++ moze stojati i kako pre tako i posle imena promenljive, ovako:

```
Y = 10 + X++;
Y = 10 + ++X;
```

Razlika nije zanemarljiva. Uzmimo da je X = 15. U prvom slucaju vrednost Y ce biti 10 + 15. U drugom Y ce biti 10 + 16. Razlika je u tome sto se u prvom slucaju vrednost X inkrementira tek po zavrsetku naredbe(takozvana „postfiksna notacija“) dok se vrednost X u drugom slucaju prvo izvrsava inkrementacija promenljive X pa se tek onda odradjuje sabiranje, pa u tom trenutku promenljiva X ima vrednost 16. Ovo se naziva „prefiksna notacija“.

3.5 Napomena

Kod pisanja realnih brojeva npr. 3.14 od esencijalne je vaznosti da greskom ne upotrebite operator zareza(,) umesto operatora tacka(.). U sintaksi jezika C ovi operatori imaju potpuno razlicito znacenje!

4.0 Naredbe. Struktura programa.

Od ovog poglavlja ce mo pisati programe koji se mogu izvrsavati na racunaru. Verovatno se neko zapitao gde je standardni „HelloWorld!“ program, uz malo kasnjenja stize i on. Veci deo prethodnog texta je bio vise-manje teoriski dok ce se od sada veci deo posla svoditi na prakticno pisanje programa.

4.1 Hello World, Osnovne stvari vezane za jezik C

Napisacemo jedan program koji trebate da kopirate u text editor vaseg kompajlera, on je vrlo jednostavan i ne radi nista korisno ali ce nam veoma dobro posluziti da upoznate osnovne „komponente“ jezika.

```
/*  PROGRAM HELLO_WORLD */
#include <stdio.h>

void main()
{
      // prikazi poruku
      printf("Da nije ove poruke ekran bi bio prazan!\n");
}
```

Analizirajmo red po red.

4.2 Komentari

Prva linija koda(/* PROGRAM HELLO_WORLD */) ne radi nista tacnije kompajler je ignorise. Primeticete da se text nalazi izmedju znakova /* i */, sto znaci da je u pitanju komentar. Cemu on sluzi? Pomocu komentara program postaje citljiviji za coveka, prevodiocu(kompajleru) je potpuno sve jedno dali postoji sto hiljada linija komentara ili ni jedna jedina. Pretpostavite da ste napisali program koji ima oko 1000 linija koda i posle dva meseca hocete da otkolnite neke greske u programu ili da ga unapredite(prosirite). Ako ste redovno pisali komentare koji su opisivali sta koja grupa naredbi radi, posao bi vam bio olaksan jer ne biste morali da „desifrujete“ kod vec samo da procitate komentar.
Jos jedan je nacin za pisanje komentara- pomocu dve kose crte(// - vidi primer). Razlika je u tome sto ce se komentarom smatrati sve izmedju /* ... */ odnosno ako koristite ove simbole mozete pisati komentare u vise redova, dok se pomocu simbola // komentarom smatra sve od tog simbola do kraja tekuceg reda.

4.3 Naredba #include. Header fajlovi. Standardna biblioteka.

Postoje naredbe koje se izvrsavaju u toku rada programa i one koje se izvrsavaju samo jednom, u toku prevodjenja(kompajliranja) programa- preprocesorske naredbe. Recimo da pisete neki program sa svojim kolegom ili u ekipi od veceg broja ljudi. Svaki od programera ce pisati svoj deo programa u posebnim fajlovima. Kada svi zavrse svoj deo posla imacete nekoliko fajlova u kojima se nalazi kod za program. Imate dve mogucnosti, da tekstove spojite u jedan fajl(sto nije dobra praksa) ili da ih povezete pomocu naredbe #include (sto je dobra praksa). U drugom slucaju, standardizovano je da ekstenzija fajlova koji se povezuju bude .H (header fajlovi). Na taj nacin imacete vise malih fajlova pa ce vam sam kod biti pregledniji.
Zasto smo izmedju < > uneli „stdio.h“? stdio.h je samo jedan od gomile header fajlova koji sadrze veoma korisne kodove koje cete upotrebljavati u svojim programima. Autori ovih fajlova su programeri koji su radili na izradi programskog jezika C. Svi header fajlovi koji standardno dolaze uz svaki valjan C kompajler se zajedno nazivaju „C Standard Library“(Standardna Biblioteka jezika C). O jednom delu funkcija(ovaj pojam cemo uskoro objasniti) SL ce mo se upoznati u ovom tutrialu medjutim dosta toga cemo zaobici.
Konkretno, header „stdio.h“ smo ukljucili jer cemo koristiti funkciju printf koja je definisana u ovom zaglavlju. U sustini, mozemo ukljuciti i sve header fajlove SL ali ce to bespotrebno povecati kolicinu memorije na HD koju zauzima program. Dakle, ukljucujemo samo one header fajlove koji sadrze funkcije koje su nam potrebne, ostale cemo izostaviti.

4.4 Osnovne stvari vezane za funkcije. Blok naredbi.

Svaki program jedino sto zaista mora da ima je bar jedna funkcija, i ta osnovna funkcija se mora zvati „main“. Sve ostale mogu imati proizvoljna imena. Naime, kada se izvrsava program, naredbe tog programa se izvrsavaju onim redom kojim smo ih pisali. Medjutim prvo treba odrediti pocetnu tacku svakog programa, mesto od kojega ce se poceti sa izvrsavanjem naredbi. To mesto je funkcija „main“.
Svaka funkcija ima nekoliko elemenata: tip podatka koji ce vratiti, ime, argumente, telo.
Pogledajmo f-ju main iz gornjeg primera:

```
int main()
{
      // prikazi poruku
      printf("Da nije ove poruke ekran bi bio prazan!\n");
```

```
}
```

-void je tip promenljive koju ce funkcija vratiti. Na ovom mestu moze stajati bilo koji tip podatka. Primetite da se ne zadaje ime promenljive koja se vraca kao rezultat f-je, ovo je logicno a u to ce te se i sami uveriti.
-main je ime funkcije.
-(), lista parametara funkcije. Ona je prazna zato sto polaznoj funkciji programa main ne prosledjujemo ni jednu promenljivu(parametar). Takodje unutar zagrada mozemo napisati i „void“, ovo je univerzalni tip podatka koji moze da zameni bilo koji tip medjutim takodje kaze i da u stvari on nije ni jedan od tipova podataka pa funkcija i ne uzima nijedan parametar. Nemojte mnogo lupati glavu oko ovog tipa podataka.
-// prikazi poruku. Komentar.
-printf(„...“). Ovo je jos jedna funkcija. To je funkcija standardne biblioteke i sluzi za ispisivanje texta i vrednosti promenljivih na ekran. Mi smo ovom naredbom pozvali funkciju printf() i kao parametar joj prosledili niz znakova "Da nije ove poruke ekran bi bio prazan!\n". To kako ce ova funkcija i uz pomoc cega prikazati text na ekran nas ne interesuje jer mi je samo koristimo, neko drugi ju je vec napisao(definisao) za nas. Mi smo u gornjem primeru definisali samo funkciju main().
-{ }. Otvorena i zatvorena viticasta zagrada, oznacavaju jedan blok naredbi. Ujedno u ovom slucaju one predstavljaju pocetak i kraj tela funkcije main(). Dakle, kada pozovete funkciju main() sve naredbe koje se nalaze unutar tela funkcije ce biti izvrsene.

4.4.1 Petlja while

Programom HelloWorld smo napravili vrlo jednostavnu aplikaciju. Sada cemo je malo zakomplikovati. Zelimo da se poruka ispise 200 puta. To bi znacilo da treba 200 puta napisati printf() f-ju, na srecu postoji daleko elegantnije resenje- koriscenje petlji odnosno naredbi ponavljanja. Ovim naredbama se postize da se odredjeni blok naredbi(ili samo jedna naredba) ponavljaju nekoliko puta. Tri su naredbe ponavljanja : while, do while i for. Sve su to rezervisane kljucne reci pa se ne mogu koristiti kao imena promenljivih. Idemo redom.

Evo kako bi izgledalo resenje problema ako ga uradimo preko petlje while.

```
/*  PROGRAM HELLO_WORLD_200 */
#include <stdio.h>

void main(void)
{
      int i=200;

      //petlja while
      while( i > 0 )
      {
            printf("Da nije ove poruke ekran bi bio prazan!\n");
            --i;
      } //kraj petlje while
}
```

Iza kljucne reci while, unutar zagrade stoji uslov „i > 0“. Ovo znaci da ce se telo petlje izvrsavati dokle god je ispunjen uslov da je „i“ vece od nule. Telo petlje je sve ono sto se nalazi izmedju viticastih zagrada, slicno kao kod tela funkcija. Unutar tela f-ja izvrsavaju se dve naredbe f-ja printf() i dekrementacija promenljive i(--i). Ukoliko ne bi bilo drugog izraza

vrednost promenljive „i“ bi bila stalno veca od nule(tj. 200) sto bi znacilo da ce se petlja izvrsavati u ne dogled. To bi bio veoma ozbiljan propust i program se ne bi valjano izvrsavao. Vrlo je bitno proveriti da li je petlja valjano napisana, dakle da ne bude beskonacna.

4.4.2 Operatori poredjenja

Prvo treba da shvatite kako funkcionisu operatori poredjenja. Ako zelite da proverite da li je 1 vece od 2 izraz bi izgledao ovako : X = 1 > 2 . Ovo nije tacno pa je rezultat koji ce biti smesten u promenljivu X jednak nuli. Medjutim da smo napisali X = 5 > 2 , uslov bi bio tacan i vrednost u promenljivoj X bi bila jedan. Dakle ako je iskaz poredjenja tacan dobija se broj jedan, ako je rezultat poredjenja ne tacan dobija se broj nula. Evo liste operatora poredjenja :

OPERATOR	Znacenje
<	Manje od
<=	Manje ili jednako
>	Vece od
>=	Vece ili jednako
!=	Razlicito
==	Jednakost

U odeljku 4.4.1 smo koristili naredbu „while(i > 0) ...“ , mogli smo takodje napisati „while(i)“ jer ce se petlja izvrsavati sve dok je rezultat izraza koji se nalazi izmedju zagrada razlicit od nule. Razlog ovom je sto se sa nulom predstavlja neispunjavanje nekog uslova a sa jednicom(ili bilo kojim brojem koji je razlicit od nule) tacnost.Prema tome petlja while, kao i sve druge petlje i naredbe grananja, ce se izvrsavati kada god je rezultat uslova koji smo zadali razlicit od nule.
Treba zapaziti razliku izmedju operatora dodele vrednost = i operatora jednakosti ==. Ako imamo ovakvu situaciju :

```
int X, Y, Z;
   X = 10;
   Y = 23;
Z = (Y == X);
```

Vrednost u Z ce biti 0, jer Y nije jednako X pa uslov nije ispunjen. Ako bi umesto poslednje naredbe stojala ova:

Z = (Y != X);

vrednost u Z ce biti 1 jer je Y zaista razlicito od X pa je samim tim uslov zadovoljen. Ovde su zagrade napisane zbog bolje preglednosti inace prvenstvo operatora != je vece od operatora jednako(=) pa i nema potrebe za zagradama ali dakle ne skodi staviti ih.

4.4.3 Petlja do while

Ova je petlja veoma slicna petlji while stim sto je kod while petlje moguce da se telo petlje ne izvrsi ni jednom jer se prvo proverava uslov pa ako je uslov tacan telo se izvrsava i tako u krug, dok kod do while petlje prvo se izvrsava telo a zatim se proverava uslov i tako u krug dok je god rezultat uslova tacan. Primer, napisimo program koji ce od korisnika traziti da unese broj veci od 500 ako korisnik unese manji broj program ce mu traziti da ponovo unese broj. Evo kako bi izgledao primer ako koristimo while petlju.

```
/*  PROGRAM broj_veci_od_500_while_nacin */
```

```
#include <stdio.h>

void main(void)
{
	//deklaracija promenljive
	int n;

	//uzmi broj od korisnika
	printf("Unesite broj veci od 500 -> ");
	scanf("%d", &n);

	//dok je god vrednost N-a manja od 500 trazi novi broj
	while( n <= 500 )
		scanf("%d", &n);

	//prikazi poruku o uspehu
	printf("Uneli ste ispravan broj! Svaka cast.\n");

}
```

Sve bi trebalo da vam je jasno sem funkcije scanf(). Ona sluzi da se od korisnika uzme neki broj ili tekst. Kako ona radi za sada nije bitno, objasnicemo je kasnije.
Jos jedino treba primetiti da kod petlje while nema viticastih zagrada koje oznacavaju telo petlje. Razlog tome je sto telo petlje sacinjava samo jedna naredba pa upotreba viticastih zagrada nije ne ophodna, no ne skodi. Treba napomenuti da telo funkcija mora sadrzati par viticastih zagrada bez obzira na broj naredbi unutar njih.
Evo kao bi resenje istog zadatka izgledalo ako koristimo petlju do while:

```
/*  PROGRAM broj_veci_od_500_do_while_nacin */
#include <stdio.h>
void main(void)
{
	//deklaracija promenljive
	int n;

	//uzmi broj od korisnika preko do while petlje
	printf("Unesite broj veci od 500 -> ");
	do {
		scanf("%d", &n);
	} while( n <= 500 );

	//prikazi poruku o uspehu
	printf("Uneli ste ispravan broj! Svaka cast.\n");
}
```

Oba nacina su ispravna ali drugi je bolji jer je kraci i preglednij. Demonstrirana je upotreba do while petlje. Dakle prvo se pise kljucna rec „do“ zatim ide telo petlje(u ovom slucaju viticaste zagrade smo mogli da izostavimo jer se izvrsava samo jedna naredba) i na kraju se proverava uslov unutar zagrada.

4.4.4 For petlja

Sintaksa for petlje je takodje jednostavna i krajnje funkcionalna iako deluje malo slozenije. Ona izgleda ovako:
for(*izraz1; uslov_petlje; izraz2) { telo petlje }*

Umesto „izraz1“ i „izraz2“ mozete kucati bilo koju naredbu, dok „uslov_petlje“ pretstavlja bas to dakle dok je god rezultat naredbe koju napisete tu razlicit od nule telo petlje ce se izvrsavati. Evo kako bi izgledao program „HelloWorld_200“ koriscenjem for naredbe:

```
/*  PROGRAM HELLO_WORLD_200_for */
#include <stdio.h>

void main(void)
{
      int i;

      for( i=200; i > 0; i++)
            printf("Da nije ove poruke ekran bi bio prazan!\n");
}
```

Dakle, naredba i = 200; ce se izvrsiti samo jednom a provera uslova i naredba i++ sa svakim ponavljanjem petlje...
Ostalo je sve jasno.

4.5 Naredbe grananja

Pomocu ovih naredbi mozemo upravljati tokom izvrsavanja koda. Zadatak, od korisnika uzeti dva cela broja i od veceg oduzeti manji. Logicno prvo treba uzeti brojeve, zatim proveriti koji je od njih veci i na kraju izracunati razliku te prikazati rezultat. Jedino sto za sada ne znate da uradite je kako odrediti koji je broj veci. To odredjujemo preko naredbe grananja. Uradicemo ovaj zadatak koristeci nekoliko razlicitih nacina grananja koda.

4.5.1 Naredba grananja – if

Resenje zadatka pomocu if-a:

```
/*  PROGRAM IF_NAREDBA */
#include <stdio.h>

void main(void)
{
      //deklaracija promenljivih
      int x, y, rez;

      //uzimanje podataka
      printf("Unesite prvi pa drugi broj -> ");
      scanf("%d%d", &x, &y);

      //proveri koji je broj manji i racunaj
      if( x > y ){
            rez = x-y; }
      else {
            rez = y-x; }

      //prikazi rezultat
      printf("Rezlutat je = %d\n", rez);
}
```

Evo izgleda sintakse if naredbe:

```
if ( uslov ) { telo }
```

If je kljucna rec, uslov mora dati rezultat razlicit od nule da bi se telo naredbe if izvrsilo. Ukoliko uslov nije ispunjen sve naredbe koje se nalaze u telu naredbe if bice preskocene. Vrlo jednostavno. U daljem kodu primecujemo kljucnu rec „else“ iza koje sledi telo te

naredbe. Njom kazemo da ako uslov if-a nije ispunjen program izvrsi blok naredbi(telo) koji sledi neposredno iza ove kljucne reci. Ukoliko je uslov if petlje tacan blok naredbi iza else ce biti preskocen. Else ne mora obavezno da stoji uz svaku if komandu, ona je opciona i ubacuje se ako ima potrebe za tim kao sto je to bio slucaj u ovom primeru. Takodje, else mora stojati neposredno po zavrsetku bloka naredbi if-a.

Jedino sto jos treba da znate je znacenja keyword-a break i continue.
-Break sluzi da se izadje iz petlje. Recimo da ste napisali ovako petlju:

```
int i = 0;
while( 1 )
    {
if( i++ == 200) break;
else  printf(„recenica\n");
    }
```

primecujemo da ce uslov petlje while biti uvek ispunjen te ce petlja biti beskonacna. Medjutim u njenom telu postoji if naredba kojom proveravamo da li je promenljiva „i“ jednaka broju 200. Ako nije ispisuje se poruka ali ako jeste, na scenu stupa komanda „break“ kojom se „iskace“ iz tela petlje i prelazi se na deo koda koji sledi iza tela petlje.
-Da bi objasnili continue napisacemo for petlju kojoj ce mo ispisati svaki drugi broj iz intervala od 0 do 200.

```
int i;
for( i = 0; i < 200; i++ )  {
 if( (i % 2) != 0) continue;
    printf( „%d\n", i);
    }
```

Prve dve linije treba da su vam vec jasne. U trecoj liniji proveravamo da li je ostatak pri deljenju promenljive „i“ sa 2 razlicit od nule. Ako nije onda je broj paran i prikazujemo ga na ekran ako pak jeste to znaci da broj nije paran te naredbom „continue“ kazemo da se pocne sa novim prolazenjem kroz telo petlje.

4.5.2 Naredba grananja „switch“

Uz pomoc ove naredbe grananja moze se postici da se, u zavisnosti od vrednosti neke promenljive, izvrsava odredjeni blok naredbi. Zadatak : uzeti od korisnika broj 1, 2,3, 4 ili 5 i za svaki broj ispisati neku razlicitu poruku. Ovo se moze uraditi preko „if“ naredbe ali ce izgledati nezgrapno, no kao vezbu pokusajte da ovo uradite na taj nacin. Evo kako bi resenje izgledalo ako koristimo switch naredbu.

```
/*  PROGRAM SWITCH_NAREDBA */
#include <stdio.h>

void main(void)
{
      int broj;

      printf("Unesite broj 1 2 3 4 ili 5 -> ");
      scanf("%d", &broj);

      switch( broj ) {
      case 1:
            printf("Uneli ste broj jedan!\n");
            break;
      case 2:
            printf("Broj dva\n");
            break;
```

```
        case 3:
              printf("Ovaj mi se broj ne dopada!\n");
              break;
        case 4:
              printf("Broj 4");
              break;
        case 5:
              printf("...bip...bip...bip..\n");
              break;
        default:
              printf("Niste lepo uneli broj!\n");
        }
}
```

Iza naredbe switch, unutar zagrada treba uneti ime neke promenljive koju na osnovu cije vrednosti ce se pozivati odredjen blok naredbi. „case 1:“ oznacava da ako je vrednost promenljve „broj“ 1 da ce se poceti sa izvrsavanjem sledecih linija koda dok se god ne naidje na naredbu break ili na kraj tela switch naredbe. To znaci da ako je vrednost promenljive „broj“ npr.5 izvrsice se samo naredba `printf("...bip...bip...bip..\n"); i break;` Sve naredbe pre njih bice preskocene. Da nema naredbe break izvrsavanje naredbi bi islo sve dok se ne dodje do kraja tela petlje, ovo ponasanje se naziva „drop-down“ ili „propadanje“. Naredbe koje slede iza kljucne reci „default“ bice izvrsene ukoliko vrednost koju ima promenljiva broj nije ni 1 ni 2 ni 3 ni 4 ni 5 vec neka druga, ne predvidjena vrednost tj. ako za vrednost promenljive broj ne postoji ni jedan odgovarajuci „case“.

Postoji jos i naredba bezuslovnog skoka, goto: . Ovom naredbom mozete se „prebaciti“ tj. skociti sa bilo kojeg mesta u funkciji na bilo koje drugo mesto u toj funkciji. Medjutim logicno je da lokaciju na koju zelite da izvrsavanje koda „skoci“ treba da oznacite nekako. To se radi tako sto na pocetku reda koji predstavlja destinaciju skoka upisujete proizvoljno ime i simbol dvotacke(:).

4.5.3 Uslovni izraz

Ovo je jedini ternarni operator u C jeziku koji obezbadjuje dvosmerno grananje. Na primer:

```
int x = 13,y = 786, max;
  max = x > y ? x : y;
```

Vrlo je prosto- poredimo da li je vrednost promenljive X veca od vrednosti Y. Ukoliko je uslov ispunjen(dakle X jeste vece od Y), vrednost koju ce dobiti promenljiva MAX je ona koja se nalazi levo od znaka dve tacke(u ovom slucaju to je vrednost koju ima X). U suprotnom vrednost koju ce dobiti MAX bice ona koja se nalazi levo od znaka dve tacke.

5. Funkcije

Funkcije su poprilicno lako „svarljive“ medjutim i nezaobilazne i jako korisne stoga je dosta bitno znati ih.

5.1 Funkcije

Do sada su svi napisani programi u ovom tutrialu koristili samo jednu funkciju – main(). Program nema ogranicenja u vidu broja funkcija koje sadrzi ali svaki program napisan na

jeziku C mora imati najmanje jednu takozvanu polaznu funkciju, dakle mesto od kojeg izvrsavanje programa pocinje. Ta se funkcija mora zvati main().
Pozeljno je da svoj kod pisete organizujuci ga u funkcije. Razlozi su mnogobrojni a samo jedan od njih je i taj sto program postaje mnogo citljiviji, funkcionalniji, pregledniji i lakse je locirati eventualne greske te iste ispraviti.
Dakle funkcije predstavljaju blok naredbi koje ce odraditi neko izracunavanje i dati rezultat tog svog rada.

Svaka funkcija ima par osnovnih elemenata a evo kako to sematski izgleda:
povratni_tip ime_funkcije(lista_parametara)
{
.... telo_funkcije(blok_naredbi)....
}

- *Povratni tip* je tip promenljive koju ce funkcija vratiti. Na primer, ako funkcija vraca ceo broj ovde ce stojati int. Ako pak funkcija ne vraca nikakvu vrednost ovde treba upisati „void“.
- *Ime_funkcije* je dakle ime sto znaci da ako zelite da pozovete ovu funkciju da bi ona odradila odredjeni zadatak upisacete njeno ime. S'toga ime funkcije trebalo bi da vam da neku sliku o tome sta ta funkcija radi.
- *lista_parametara* predstavlja listu promenljivih koje su potrebne funkciji da odradi odredjeni posao. Recimo da imate funkciju koja sabira dva broja koja joj prosledite. Logicno je da funkcija nece moci da odradi svoj zadatak ako joj vi ne predate dva broja koja ce ona onda da sabere i vrati vam njihov zbir.
- *telo_funkcije* je blok naredbi koje ce izvrsiti zadatak te funkcije. Samim tim telo funkcije cini njenu srz i najbitniji deo. Kao i svaki drugi blok naredbi i ovaj mora stojati izmedju viticastih zagrada koje pretstavljaju pocetak i kraj bloka.
Ako vam nije bas jasno gore navedeno sledi primer sa detaljnijim objasnjenjem.

5.2 Funkcija Obim()

Napisacemo program koji ce pored funkcije main() imati i jps jednu funkciju. Tu cemo funkciju nazvati Obim() i ona ce imati zadatak da izracuna obim pravougaonika cije stranice joj prosledimo. Ovaj program ce biti dosta menjan i nadogradjivan kroz dalji tok tutrijala.

```
/*  PROGRAM Obim */
#include <stdio.h>

//prototip(ili deklaracija) funkcije
double Obim( double a, double b);

//funkcija main
void main(void)
{
      //deklaracija promenljivih
      double x,y, rez;

      //uzmi podatke
      printf("Unesite duzinu stranice A i B -> ");
      scanf("%lf %lf", &x, &y);

      //pozovi funkciju Obim da bih izacunao obim tela zadanih
      //dimenzija
      rez = Obim(x, y);
```

```
	//prikazi rezultat
	printf("Obim pravougaonika stranica A i B je %.2lf! \n", rez);

}


//definicija funkcije Obim()
double Obim( double a, double b)
{
	double ret;

	ret = a * b;

	return ret;
}
```

Pomocu naredbe #include<stdio.h> smo ukljucili fajl stdio.h. On nam je potreban jer su u njemu definisane funkcije koje ce mo koristiti u programu.
Zatim sledi deklaracija funkcije Obim(). Verovatno vam nije jasno zasto se ovo radi. Predpostavimo da deklaraciju nismo napisali. Pokusacemo da prevedemo program i dobiti gresku u kojoj se kaze da nije nadjena funkcija Obim() iako je ona definisana posle funkcije main. Razlog ovome je taj sto se unutar main() funkcije nalazi naredba kojom se poziva funkcija Obim():

```
rez = Obim(x, y);
```

Medjutim prevodioc program prevodi liniju po liniju koda pa posto mu ranije nismo rekli da postoji funkcija Obim() on ce se buniti i obavestiti nas o tome. Dakle funkcija Obim() jeste definisana kasnije u programu medjutim prevodilac toga jos uvek nije svestan jer jos nije stigao da „prevede“ tu funkciju a mi smo pokusali da je upotrebimo. Da bi se ovo izbeglo pribegava se pisanju prototipova funkcije. Prototipovi funkcija sadrze povratni tip funkcije, ime funkcije i listu argumenata. Telo se pise kasnije i taj se „proces“ naziva definisanje funkcije. Prototipovi trebaju da stoje pri pocetku koda programa jer se na taj nacin blagovremeno obavestava prevodioc da postoji odredjena funkcija i da ce ona kasnije biti i definisana.
Alternativa ovome postoj ali je ona losije resenje. Naime vi mozete deklarisati i definisati funkciju, u ovom slucaju Obim(), pre nego sto napisete neku naredbu koja ce je pozivati, u ovom slucaju je to naredba u funkciji main(). Ovo radi ali je losa praksa jer kod postaje ne pregledan plus sto ovaj metod nije 100% pouzdan.

Sledi definicija funkcije main() i posle nje par jasnih naredbi u kojima deklarisemo promenljive i pozivajuci funkcije printf() i scanf()(koje su definisane u zaglavlju stdio.h, detaljnije ce mo kasnije reci) vrsimo komunikaciju sa korisnikom i dobavljamo potrebne podatke. Nakon toga racunamo obim pozivajuci funkciju Obim() i smestajuci rezultat u promenljivu rez. Funkcija obim zahteva da joj se proslede dva argumenta tj.promenljive koje su tipa double.
Ovde treba razjasniti sta je argument a sta parametar. Kada definisete funkciju vi definisite i koje ce parametre(dakle promenljive) ta funkcija da zahteva od pozivaoca funkcije. Kada pak pozivate tu funkciju, promenljive koje joj se prosledjuju se nazivaju argumenti. Malo cudno i glupavo ali...

Dolazimo do definicije funkcije Obim(). Uglavnom je sve jasno sem zadnje naredbe:

```
return ret;
```

Telo funkcije se moze zavrsiti na dva nacina: kada se dodje do kraja tela funkcije tj. do viticaste zagrade (}) ili kada se dodje do naredbe return.

Iza naredbe return sledi promenljiva koja ce biti vracena kao rezultat funkcije. Jasno je da se tip promenljive i tip vrednosti koju ce funkcija vratiti moraju poklapati odnosno da u ovom slucaju promenljiva ret mora biti tipa double.
Vrlo je bitno uvideti da ako imate slucaj:

```
int x = 5;
return x;
 x = 10;
```

promenljivoj x se nikada nece dodeliti vrednost 10 jer se izvrsavanje tela ove funkcije prekida naredbom return. Ako funkcija ne vraca vrednost(void) onda se return pise bez ikakve promenljive posle njega.

Pogledajmo jos jednom listu parametara. U njoj smo definisali da funkciji moraju biti prosledjene dve promenljive tipa double. Njima smo dodelili imena a i b. Dakle bez obzira na sve, kada mi prosledimo nekoj funkciji promenljivu, funkcija nece biti u stanju da menja vrednost prosledjene promenljive. Naime, kada se promenljiva prosledi nekoj funkciji, na posebnom mestu u memoriji- steku formira se jos jedna poromenljiva sa karakteristikama(tip, ime) koje smo joj zadali prilikom definicije te funkcije koju pozivamo i dodeljuje joj se vrednost promenljive koju smo prosledili. Dakle za svaki prosledjenu promenljivu se na steku pravi nova promenljiva i inicijalizuje se vrednoscu prosledjene promenljive. Da pojasnimo na gornjem primeru. Naredbom Obim(x, y) smo pozvali funkciju Obim() i prosledili joj dva argumenta x i y. Ovo daje „direktivu“ programu da treba da „skoci“ na mesto na kome je definisana funkcija Obim() i da odatle nastavi sa izvrsavanjem koda. Kada se telo funkcije Obim() zavrsi, program nastavlja sa izvrsavanjem od mesta sa kojeg je Obim() bio pozvan, u ovom slucaju to je naredba: `rez = Obim(x, y);` koja se nalazi u main(). Vrednost koju ce funkcija Obim() vratiti(pomocu naredbe return) bice dodeljena promenljivoj rez...

Dolazimo do dela koji sada necete mozda razumeti jer treba da znate sta je to opseg vaznosti promenljive(objasnicemo u sledecem podpoglavlju) ali valja napomenuti. Funkcija Obim() ne moze da pristupi promnljivim koje ste deklarisali unutar funkcije main(), i uopste unutar neke druge funkcije(zbog opsega vaznosti), pa zato se moraju kreirati dve nove promenljive unutar funkcije Obim() u kojima ce mo cuvati vrednosti prosledjenih argumenata. Imena tih promenljivih koje smo definisali u listi parametara funkcije Obim() mogu da imaju ista imena kao i promenljive koje smo definisali u telu funkcije main() ali naravno mogu biti i razlicita. Ovo je opet zbog opsega vaznosti, a sledi i objasnjenje sta je to.

5.3 Opsezi vaznosti

Postoje 3 opsega vaznosti: lokalni, globalni i eksterni. Zadnji je retko koriscen pa ga necemo izeti u razmatranje.
Opseg vaznosti(eng. „scope“) je svojstvo svake promenljive. Promenljive sa lokalnim opsegom vaznosti(krace receno „lokalne promenljive“) su „vidljive“ samo unutar jednog bloka naredbi(telo funkcije, telo petlje) u kojem su definisane. U predhodnom primeru promenljive a i b su bile definisane u telu funkcije main() pa njima ne mozete da pristupite iz tela funkcije Obim(), zbog toga sto su one „vidljive“ samo unutar tela u kojem su definisane u ovom slucaju to je telo funkcije main(). Pogledajmo sledeci primer:

```
int main(void)
{
	//deklaracija lokalne promenljive S
	int S;
	S = 10;
```

```
	if(1) {
		//deklaracija promenljive U unutar tela naredbe if
		int U = 20;
		//dodela vrednosti promenljivoj S
		S = 20;
	} //kraj tela if naredbe
	//greska
	U = 13;

	return 0;
}
```

Ovaj se kod ne moze kompajlirati. Pogledajmo liniju kod ispod komentara „//greska“. Promenljiva U je definisana unutar bloka naredbi if–a . Ako pazljivo proanalizirate ovaj kod shvaticete da je, kao sto je gore objasnjeno, promenljiva U vidljiva samo unutar blok u kojem je definisana tj.unutar bloka naredbi if-a. Vazno je uvideti i da je promenljiva S vidljiva unutar if bloka... Jos jedna razlika izmedju tela funkcija i tela naredbi grananja/petlji...

Postoje i globalne promenljive. To su sve promenljive koje se definisu van tela funkcija. Ove promenljive su vidljive u svim delovima delovima programa dakle dostupne su svim funkcijama. Prednost ovih promenljivih a ujedno i velika mana(zbog cega ih treba zaobilaziti u sirokom luku) je to sto njihove vrednosti mogu menjati svi delovi programa, sve funkcije. Ovo deluje na prvi pogled dobro ali sta ako imate neki veci program od nekoliko stotina ili hiljada linija koda i kada ga budete testirali vi recimo naidjete na gresku da neka globalna promenljiva ima ne odgovarajucu vrednost. Kako ce te locirati sta pravi problem? Neka vam je bog u pomoci. No da ste definisali tu promenljivu kao lokalnu i po potrebi je predavali kao argument nekoj funkciji imali bi ste mnogo(ali bas mnogo) manje posla.
Primer programa koji racuna obim koristeci globalne promenljive ce izgledati ovako:

```
/*  PROGRAM Obim_Glob */
#include <stdio.h>

//prototip(ili deklaracija) funkcije
void Obim( void);

//DEKLARACIJA GLOBALNE PROMENLJIVE
double x, y, rez;

//funkcija main
void main(void)
{
	//uzmi podatke
	printf("Unesite duzinu stranice A i B -> ");
	scanf("%lf %lf", &x, &y);

	//pozovi funkciju Obim da bih izacunao obim tela zadanih
dimenzija
	Obim();

	//prikazi rezultat
	printf("Obim pravougaonika stranica A i B je %.2lf! \n", rez);

}
```

```
//definicija funkcije Obim()
void Obim( void)
{
      rez = x * y;
}
```

Uporedite detaljno razliku izmedju ova dva nacina izvodjenja ovog programa, bice vam to dobra vezba.
Program je skracen ali isto tako pretstavlja veoma losu programersku praksu. Zato drzite se lokalnih promenljivih! Sada bi trebalo da vam je jasno kako rade funkcije. Treba da se ucite da sve organizujete po funkcijama, prednosti ce te sami uvideti.
Za kraj napomenimo da ipak postoji mogucnost da jedna funkcija menja vrenosti lokalnih promenljivih deklarisanih unutar druge funkcije, pomocu pokazivaca ali cemo njih objasniti nesto kasnije da bi vam prvo predstavili kako mozete da „komunicirate“ sa korisnikom vaseg programa, sto ce uciniti da vam programiranje postane mnogo zanimljivije.

5.4 Preopterecivanje funkcija

Svaka funkcija ima dve osobine koje je cine jedinstvenom: ime i listu argumenata. Iako deluje nelogicno vise funkcija moze imati potpuno isto ime, ali onda im se mora razlikovati lista argumenata. Takve funkcije su preopterecene. Jedina funkcija koja nemoze biti preopterecena je funkcija main().
Svrha preopterecenih funkcija je da u zavisnosti od tipova argumenata izvrsavaju razlicite naredbe. Recimo da imate dve funkcije koje treba razlicito da rade u zavisnosti od tipa argumenta koji joj prosledite. Postoje dva resenja: ili da funkcije imaju razlicito ime ili da budu preopterecene te da im tip parametra koji uzimaju bude razlicit. Drugo resenje je obicno bolje.

6. Osnove prikupljanja podataka

Objasnicemo neke od funkcija stadardne biblioteke, prevashodno one pomocu kojih ce te prikupljati podatke od korisnika i prikazivati rezultate rada vasih programa. Ono sto ste do sada ucili bilo je vezano iskljucivo za manipulaciju podacima tj. koriscenje naredbi i njihova organizacija da bi ste obavili odredjeno racunanje ili sta vec. Posle ovog poglavlja znacete kako da uzimate podatke -> manipulisete njima -> prikazujete rezultate, te ce te jedan vrlo sirok krug zatvoriti i spoznati poprilican deo sintakse jezika C.

6.1 Napomena o razlici izmedju texta i znaka

U C jeziku svaki text se tretira kao „niz znakova“ a sta je to objasnicemo u poglavlju o pokazivacima, za sada treba da znate da svaki „slobodan“ text treba da stavite pod navodnike jer ako to ne uradite kompajler ce smatrati da je taj text neka naredba ili ime promenljive idr.
Mnogi jezici imaju tip promenljive „string“ koji sluzi za smestanje texta. Kod C jezika ne postoji tip string ali u C-u se isti efekat moze postici koristeci se tipom char. Char znaci znak i u promenljivu tipa char ne mozete staviti text vec samo jedan znak, na primer takva promenljiva moze da cuva vrednosti ’T’ ili ’?’ ali ne i „neki_text“. Sve ce ovo biti detaljno objasnjeno u odeljku o pokazivacima i nizovima ali vec sad treba da uvidite razliku izmedju

znaka(tacnije simbola) i texta. Sav se tekst stavlja pod navodnike – „text“, a simboli pod znacima „gornjeg zareza“ npr. ’a’ , ’e’, ’+’ ...

6.2. Funkcija printf()

Sa ovom smo se funkcijom ranije sreli ali nismo objasnili kako ona radi. Printf() funkcija sluzi za prikazivanje texta i vrednosti na monitor(tacnije na konzolu). Ova je funkcija komplikovanija ali sve postaje lako i „prirodno“ uz malo vezbe. Evo kako se pomocu ove funkcije ispisuje neki text na ekran:

```
printf("Ovo je primer");
```

Funkciji smo kao argument predali text i on ce biti ispisan na ekran.
Medjutim printf() sluzi za formatiran prikaz texta na ekran, sto znaci da koristeci ovu funkciju mozete vrsiti tabulaciju(horizontalnu i verikalnu), prelaziti u novi red idr. Sve se ovo postize koriscenjem komandnih sekvanci.

Svaka komandna sekvenca je u stvari simbol iako se sastoji od dva znaka, pa bi po recenom u predhodnom odeljku trebalo da se tretira kao text ali to nije slucaj(shvati ce te kasnije zasto). Komandna sekvenca se sastoji od znaka „beksleš“(\ ) i jos nekog dodatnog znaka, evo tabele:

K . S	OPIS
\n	New line (prelazak u novi red)
\t	Horizontalna tabulacija(HT)
\v	Vertikalna tabulacija(VT)
K . S	OPIS
\b	Backspace(vraca jedno mesto nazad
\r	Vraca na pocetak reda
\a	Alarm(beep zvuk)
\’	Prikazuje na ekran simbol ’
\“	Prikazuje na ekran simbol “
\?	Prikazuje na ekran simbol ?
\\	Prikazuje na ekran simbol \
\Oktalno	Prikazivanje ceo broj oktalno
\Xheksa	Prikazivanje ceo broj heksadecimalno

Objasnimo najkorisnije sekvence. Ako zelite da predjete u novi red kuca ce te sekvencu \n unutar texta koji se predaje kao parametar funkciji printf(), ovako:

```
printf("Prvi red\nDrugi red \n\n\nSesti red...");
```

Mozete izvrsiti na slican nacin i tabulaciju:

```
printf("Pocetak\tpa razmak\n Pa novi red...\n");
```

Razlog sto morate kucati beksleš da bi ste odstampali navodnike i gornji zarez a i sam beksleš ce te i sami uvideti ako malo razmislite. Recimo ako hocete da prikazete neki text pod navodnicima na ekran a ako ne koristite beksleš na odredjenom mestu dobili bi ste gresku prilikom kompajliranja.

```
printf("ovo "nece" raditi...");
```

Pogledajte prosledjeni argument. On zapravo nije string(text) vec se sastoji od stringa “ovo “, ne definisane reci – nece, i jos jednog stringa “ raditi...“. Da li smo to hteli? Ispravna forma ovoga ce izgledati ovako:

```
printf("ovo \"nece\" raditi...");
```

Ovim ce mo i mi i kompajler biti zadovoljni. Provezbajte ove naredbe pa krenite dalje sa ucenjem.

Ovo prikazivanje text na ekran je jednostavno uraditi ali stvari postaju komplikovanije ako hocemo da prikazemo i vrednosti nekih promenljivih.
Sintaksa funkcije printf je sledeca:

printf(upravljacki_niz_znakova, parametar1,prarametar2,parametarN)

Objasni ce mo na sledecem primeru:

```
/*  PROGRAM printf1 */
#include <stdio.h>

void main(void)
{
      int A = 10;
      float B = -789.4612;

      printf("Vrednost A je %d, dok je vrednost B %f \n", A, B);

}
```

Vidljivo je da smo sada funkciji predali vise argumenata, ukupno 3. Unutar upravljacnog niza se nalaze izmedju ostalog i upravljacki znaci, %d i %f.

Ovi znaci „govore“ funkciji da na njihovom mestu treba da se nadje vrednost neke promenljive, a koja je to promenljiva navodimo kao dodatni argument, u ovom slucaju to su promenljive A i B. Dakle upravljacki znaci su slicni komandnim sekvencama s tom razlikom sto komandne sekvence sluze za formatiranje texta a upravljacki znaci za prikazivanje vrednosti promenljivih.
Verovatno ste se zapitali zasto smo za prikazivanje vrednosti promenljive A koristili upravljacki znak %d, a za za prikazivanje vrednosti promenljive B %f. Razlog je taj sto se ove promenljive razlikuju u tipu(A je tipa int dok je B float) a za svaki tip promenljive postoje razliciti upravljacki znaci, kao sto je slucaj i sa komandnim sekvencama. Evo tabele:

U. Z	Opis
%c	Za tip podatka CHAR
%d, %i	Za tip podatka INT
%f	Za tip podatka FLOAT
%lf	Za tip podatka DOUBLE
%e	Prikazuje broj u eksponencionalnom obliku

Upravljacki znaci se pravilnije zovu „specifikatori konverzije celih brojeva“. Rekli smo jos pri pocetku da postoji eksplicitna i implicitna konverzija jednog tipa promenljive u drugi. Evo kako moze doci do inplicitne konverzije(malo izmenjen kod prethodnog primera):

```
/*  PROGRAM printf2 */
#include <stdio.h>

void main(void)
{
      int A = 10;
      float B = -789.4612;

      printf("Vrednost A je %d, dok je vrednost B %d \n", A, B);

}
```

Promena je izvrsena nad samo jednim znakom, umesto %f sada stoji %d u upravljackom nizu koji smo predali printf()-u. Ovim smo naveli da je tip druge promenljive INT a on je u stvari FLOAT pa ce doci do gubitka preciznosti. A zbog cega ce se to dogoditi vec smo rekli u poglavlju 3.3.
I to je cela nauka vezana za funkciju printf(), od esencijalne je vaznosti da ovo savladate kako treba jer je ovo veoma koriscena funkcija a i kasnije ce vam biti mnogo lakse da naucite i funkcije za uzimanje podataka(scanf()) od korisnika programa.

6.3 Funkcija scanf()

Ako ste shvatili kako i zasto funkcionise printf(), funkciju za formatirano uzimanje podataka ce te shvatiti bez i najmanje poteskoca. Sintaksa ove funkcije je:

```
 scanf( upravljacki_niz_znakova, parametar1,prarametar2,parametarN )
```

Dakle sintaksa je potpuno ista kao i kod printf(), tako da je sve sto treba reci vec receno. Ipak postoji jedna mala razlika, evo primera:

```
/*  PROGRAM scanf()1 */
#include <stdio.h>

void main(void)
{
```

```
    int A;
    float B;

    //uzmi podatke
    printf("Unesite vrednost A a zatim i promenljive B: ");
    scanf("%d %f", &A, &B);

    //prikazi rezultat
    printf("Vrednost A je %d, dok je vrednost B %f \n", A, B);

}
```

Ceo kod bi trebalo da vam je jasan sem naredbe u kojoj pozivamo funkciju scanf(). Funkciji smo prosledili upravljacki niz `"%d %f"` sto ce reci da korisnik treba da unese vrednost tipa INT a zatim i vrednost koja ima tip FLOAT. Unete vrednosti ce biti dodeljene promenljivim A i B. Primeti ce te da pored imena promenljive stoji znak ampersend(&) koji u stvari predstavlja operator „adresa-od“. Deteljnije ce mo ga objasniti u poglavlju o pokazivacima, za sada treba da znate da smo ga stavili tu da bi omogucli funkciji scanf() da upise vrednost koju uzme od korisnika u promenljive A i B. Kako, sta, zasto? Kada smo govorili o funkcijama rekli smo da se lokalnim promenljivim moze pristupati samo unutar bloka naredbi u kojem su iste deklarisane na primer samo u funkciji u kojoj su te promenljive deklarisane i to je uradjeno bas zato da druge funkcije ne mogu pristupati ovim promenljivim i menjati njihovu vrednost. No, funkcija scanf() mora da upise vrednost u promenljive koje smo joj predali kao argumente pa se mora nekako zaobici gore navedeno ponasanje. To se ostvaruje pomocu operatora & jer mi kada koristio ovaj operator, u stvari ne prosledjujemo funkciji promenljivu vec njenu adresu u memoriji racunara pa ce se u tu adresu upisivati podaci i samim tip menjati vrednost prosledjenog argumenta. Ako niste razumeli o cemu je rec ne mari jer ce biti vise reci u poglavlju o pokazivacima, za sada morate zapamtiti da kod funkcije scanf() morate koristiti operator &(uz svako ime argumenata, naravno).

6.4 Zakljucak

Sada znate kako da od korisnika uzimate i kako da korisniku prikazujete podatke koristeci printf() i scanf(). Druga funkcija je jako „pipava“ tj.u slucaju da vrednosti koje korisnik unosi nisu onakve kakve mi ocekujemo desice se vrlo ne prijatne stvari. Zato se scanf() koristi za ucitavanje pazljivo sortiranih podataka, recimo iz nekog fajla. Primeticete da nismo ni u jednom primeru nit uzimali nit prikazivali vrednosti koje su tipa CHAR ili string, razlog je sto je prvo potrebno da ovladate pokazivacima i nizovima.
Postoje jos mnoge funkcije za upis i ispis ali ce mo njih razmotriti tek posle sledeceg poglavlja.

7.0 Pokazivaci i Nizovi

Pokazivaci su jedna od glavnih prednosti jezika C. Oni sluze za direktan pristup memoriji racunara. Vec smo rekli da kada deklarisemo neku promenljivu, ma koji bio njen tip, sistem mora da rezervise potrebnu kolicinu memorije u koju ce smestati odredjene vrednosti. Da bi davali vrednost promenljivoj ili da bi smo njenu vrednost „citali“ koristimo ime te promenljive. Ako pak deklarisemo pokazivac on ce „pokazivati“ na lokaciju neke promenljive unutar

memorije i dakle sadrzace adresu promenljive na koju pokazuje, a ne vrednost koja se nalazi na toj adresi. Mozda vam ovo nije jasno za sada, mozda vam izgleda bespotrebno ali sledi objasnjenje kroz jedan primer.

7.1 Kako rade pokazivaci i adrese.

Svaka promenljiva ima adresu na kojoj se nalazi u memoriji. Posto je i pokazivac vrsta promenljive i on ima svoju adresu u memoriji. Postavlja se pitanje kako vi u programu mozete da saznate na kojoj se adresi nalazi neka promenljiva. Odgovor je- pomocu operatora & („ampersend“ ili operator „adresa-od“). Evo kako to izgleda na delu:

```
int BROJ;
BROJ = 45;
printf("Vrednost promenljive je %d \n", BROJ);
printf("Adresa promenljive je %d\n", &BROJ);
```

Prvo deklarisemo promenljivu koja ima tip INT a ime joj je BROJ. To znaci da je sistem na nekoj lokaciji u memoriji rezervisao dovoljno mesta da stane svaka vrednost koju podrzava tip INT(dakle za svaki celi broj). Zatim naredbom `BROJ = 45` vrsimo dodelu vrednosti promenljivoj BROJ. Dakle menja se vrednost promenljive ali njena adresa je ista. Adrese svih promenljivih su staticne i ne mogu se menjati. Zatim pomocu funkcije printf() prikazujemo na ekran vrednost koju ime promenljiva BROJ, u ovom slucaju je to 45.
U sledecem printf()-u smo ispred imena BROJ stavili operator & sto znaci da necemo dobiti vrednost promenljive BROJ vec njenu adresu u memoriji. Takodje se iz primera moze zakljuciti da su adrese celi brojevi(INT). Ovaj primer nije narocito korista ali je posluzio da objasnimo kako se dobijaju adrese promenljivih, na dalje ce mo objasniti kako da ih koristite za nesto pametno.

Kao sto smo rekli, pokazivac ne moze da sadrzi vrednost vec moze samo da sadrzi adresu neke promenljive. I to ga cini posebnim tipom podatak jer ni jedan drugi tip nemoze cuvati adrese. Bitno je da uvidite da za svaki tip promenljive(int, float, double...) postoji i pokazivac na taj tip. Dakle pokazivac na tip INT ne moze sadrzati adresu promenljive koja je tipa FLOAT. Evo primera deklaracije pokazivaca:

```
int *POK;
```

ovo je pokazivac. Primecujete operator *(operator posrednog pristupa) on govori da zelimo da deklarisemo pokazivac na tip INT. Da nema operatora * mi bi deklarisali klasicnu promenljivu tipa INT. E sad imamo pokazivac, evo kako se on moze koristiti:

```
int BROJ;
int *POK;

POK = &BROJ;
printf("Vrednosti BROJ je: %d", *POK)
```

prve dve linije su jasne. Trecom linijom smo pokazivacu POK dodelili adresu BROJ-a, te sad preko pokazivaca POK mozemo iscitavati i upisivati neke vrednosti direktno u to mesto na memoriji. Cetvrtom linijom smo iscitavali vrednost. Primecujete da smo stavili `*POK` , dakle opet smo koristili operator * kao i kod deklaracije pokazivaca. Medjutim znacenja ovog operatora je dvojako, u zavisnosti u kom se kontekstu upotrebljava. Razmotrimo na sledeci nacim.
Ako napisemo samo ime pokazivaca(samo – POK) , dobicemo adresu na koju on pokazuje. Ako hocemo da pokazivacu dodelimo neku drugu adresu pisacemo `POK = &necega` i to bi trebalo da je jasno sada. E, kada smo pokazivacu dodelili da pokazuje na adresu neke promenljive mi vrednosti na toj adresi mozemo da pristupamo. Pa ako hocemo da dobijemo vrednost na toj adresi na koju pokazivac pokazuje stavicemo operator * ispred imena pokazivaca: `*POK` . To je upravo ono sto smo uradili u poslednjoj liniji koda gornjeg

primera. Medjutim vrednost se moze i menjati ne samo citati pomocu pokazivaca, princip je slican. Da bi ste dodelili vrednost onome na sta pokazivac pokazuje napisacete :

```
*POK = 1989;
```

Ako i dalje ne shvatate ovo evo jos jednog primera:

```
/*  PROGRAM POKAZIVACI_1 */
#include <stdio.h>

void main(void)
{
      //deklarise promenljivu tipa int
      int A;
      //deklarise promenljivu tipa pokazivac na int
      int *P;
      //dodeljuje vrednost promenljivi A
      A = 10;
      //Dodeljuje pokazivacu P adresu od A
      P = &A;
      //neposredno menja vrednost promenljive A
      A = 456;
      //Posredno menja vrednost promenljive A
      *P = 123;
      //neposredno cita vrednost promenljive A
      printf("A je %d \n", A);
      //posredno cita vrednost promenljive A
      printf("A je %d \n", *P);

}
```

U ovoj liniji `*P = 123;` smo posredno(preko pokazivaca) smestili vrednost 123 u memoriju na koju je pokaziva P. Ovo znaci da ce se i vrednost koju ima A promeniti u 123 iz logicnog razloga, pomocu pokazivaca smo i bez upotrebe promenljve A mi njenu vrednost promenili tako sto smo upisali novu vrednost u adresu na memoriji koju koristi promenljiva A. Ovo je bitno uvideti da bi ste razumeli kako funkcionisu pokazivaci.

Jos jedna bitna stvar, kako se ponasaju operatori ++ i – kada se koriste nad pokazivacima. Pa ponasaju se potpuno isto. S' tim sto operatori ++ i – imaju vece prvenstvo od operatora *. Sto ce reci da ako imate komandu `*P++;` vi ce te inkrementirati(povecati za jedan) adresu na koju pokazuje pokazivac P, dok ako napisete `(*P)++;` vrednost u memoriji na koju pokazuje pokazivac P ce biti inkrementirana a adresa na koju pokazuje P ostaje naravno ista.

7.1.2 Pokazivaci i funkcije

Rekli smo dovoljno o „dometu“ promenljivih i izveli zakljucak su promenljive vidljive samo unutar bloka naredbi unutar kojih su definisane npr. unutar tela neke funkcije.
Sa druge strane imamo ogranicenje u tome sto funkcije ne mogu da vrate vise od jedne vrednosti a takodje i ne mogu da menjaju vrednosti promenljivih koje su deklarisane unutar drugih funkcija jer ih „ne vide“. Ovo se moze promeniti koriscenjem pokazivaca za parametre funkcija. Evo primera jedne takve funkcije:

```
int test( int *a, int *b )
      {
      *a = 789;
      *b = 456;
      }
```

A evo kako bi izgledala funkcija koja poziva funkciju test :

```
void main(void)
{
int e = 23,r = 10;
test( &e, &r);

//prikazi vrednost
printf("E je %d, R je %d \n", e,r);
}
```

Na ekran ce se ispisati „E je 789, R je 456 " sto znaci da smo iz funkcije test() uspeli da promenimo vrednosti promenljivim koje su definisane unutar funkcije main(). To smo uspeli jer smo funkciji test prosledili kao argument ne vrednosti koje imaju e i r vec njihove adrese. Zatim smo u telu funkcije test() pomocu pokazivaca promenili vrednost koja se nalazi u delu memorije koju koriste promenljive e i r . Ove dve promenljive i dalje nisu vidljive unutar funkcije test() ali smo obezbedili njihove adrese i pomocu njih smo u mogucnosti da menjamo njihove vrednosti.
Po ovom principu radi i funkcija scanf() tako da sada znate zasto smo joj prosledjivali adrese promenljivih a ne same promenljive dakle da bi joj omogucili da podatke koje uzme od korisnika upise u te promenljive koje smo joj prosledili.

7.2. Nizovi

Nizovi su niz promenljivih istog tipa koje su kontinualno napravljene u memoriji. Ako hocemo da napravimo jednu promenljivu tipa INT pisemo:

```
int broj;
```

A sta ako hocemo da napravimo pet promenljivih, da u njih smestimo odredjene vrednosti i da te vrednosti medjusobno saberemo? Dosadasnja metoda deklarisanja promenljvih bi zahtevala da promenljive imaju razlicita imena i kod bi bio dosta dosadan i nepotrebno dug, medjutim stvari postaju vidno lakse sa koriscenjem nizova. Evo kako se moze deklarisati niz od 10 promenljivih tipa INT:

```
int broj[10];
```

Uglaste zagrade([]) ukazuju da se radi o nizu a ne o samo jednoj promenljivoj. Tako sada imamo promenljivu `broj[5], broj[2], broj[7]` ... dakle broj mora imati index. Index je broj unutar uglastih zagrada.
Ipak navedenom deklaracijom nismo napravili promenljivu broj sa indexom 10 (dakle ne postoji broj[10]). Razlog ovome je taj sto index prvog broja nije 1, vec 0(nula) . Tako da mi imamo sada `broj[0] broj[1] broj[3] broj[4] .... broj[9]`. Jos samo jednu stvar treba da znate, ako napisemo samo `broj` dakle bez ikakvog indexa to ce imati isti efekat kao da smo napisali `broj[0],` dakle pristupamo prvoj promenljivoj niza.
Kao sto mozete da inicijalizujete vrednosti „obicnih" promenljivih tako mozete inicializovati i vrednosti nizovskih promenljivih. Razmotrimo sledci primer:

```
int broj[4] = { 4679, 1, -3555, -8 };
```

Tako se inicializuju nizovske promenljive. `broj[0]` ce imati vrednost 4679, `broj[2]` ce biti -3555 idt... Obavezni ste da stavite viticaste zagrade({}).
Recimo da imate niz od 10 elemenata i svaki element zelite da inicijalizujete vrednoscu 0. Nema potrebe da navodite deset nula unutar viticastih zagrada, ovaj sistem je mnogo elegantniji:

```
int broj[10] = { 0 };
```

Svi elementi niza bice inicializovani vrednoscu nula.

Sledi program koji resava problem koji smo naveli na pocetku poglavlja:

```
/*  PROGRAM NIZOVI_1 */
#include <stdio.h>

void main(void)
{
      //deklaracija niza
      float broj[5];
      double rez = 0;
      int i;  //pomocna promenljiva

      //uzmi podatke i smesti ih u broj
      for(i=0; i < 5; i++)
      {
            printf("Unesite broj %d : ", i+1);
            scanf("%f", &broj[i] );
      }

      //saberi
      for(i = 4; i+1 ; i--)
            rez += broj[i];

      //prikazi rezultat
      printf("Zbir unesenih brojeva je %lf \n", rez);
}
```

Iako bi trebalo sve da je jasno ovaj cemo primer detaljno objasniti. No ako vam je jasno sledeci deo texta mozete preskociti.

Prva linija koda je komentar.

U drugoj liniji smo pomocu direktive `#include` ukljucili `stdio.h` , header fajl standardne biblioteke jezika C. Ovo smo uradili jer su u tom headeru definisane funkcije scanf() i printf() koje su nam potrebe za izradu programa.

Zatim sledi funkcija main(). Ona predstavlja polaznu tacku svakog programa, te je svaki programa mora imati.

Prva stvar koju radimo unutar tela main()-a je deklaracija promenljivih koje ce nam kasnije trebati. Bitno je uvideti da promenljive moraju biti definisane na samom pocetku bloka naredbi, dakle pre bilo kojih drugih naredbi. Promenljiva broj je tipa float i ona je niz od ukupno 5 elemenata(prvi element se nalazi na indexu 0 a posledji na indexu 4).

Nailazimo na petlju for. Prvom naredbom(i = 0) postavljamo vrednost promenljive i na nulu. Druga naredba pretstavlja uslov da bi se petlja ponavljala. Treca naredba se kao i druga, izvrsava pri svakom ponavljanju tela petlje i njom inkrementiramo „i“. U telu petlje nalaze se dve naredbe, prvom kazujemo korisniku sta treba da unese dok drugom prihvatamo podatke koje ce korisnik uneti. Razmotrimo ovu naredbu:

```
printf("Unesite broj %d : ", i+1);
```

Kao drugi argument smo predali `i + 1`. Mozda cete pomisliti da smo na ovaj nacin povecali vrednost promenljive i ali u tom slucaju cete se prevariti. Sistem ce kreirati pomocnu promenljivu i dodeliti joj vrednosti zbira ovih operanada a zatim ce tu promenljivu predati funkciji, dok vrednost promenljive „i“ ostaje ista(pogledajte odeljak 3.3 ako vam ovo nije jasno).

Sledi jos jedna petlja for, ovoga puta njena uloga je da sabere sve elemente niza bromenljive broj. Naredbe unutar malih zagrada su namerno malo zakomplikovane. Naime promenljivoj „i“ dodeljena je vrednost 4, dok je u prvoj petlji dobila vrednost 0. Oba su nacina ispravna ali pogledajmo sledecu naredbu, naredbu uslova. U njoj stoji samo „ i + 1 “. Podsetimo se da je vrednost nula znaci da uslov petlje nije ispunjen a da svaka druga vrednost znaci ispunjenost uslova. Samim tim ovo ce raditi. Medjutim ono „ + 1“ stoji zato sto ,da ga nema petlja bi se izvrsila 4 puta. Malo je teze pratiti ovakve stvari zato je bolje pridrzavati se „standarda“ iz

prve for petlje. Uvidjate da druga petlja for nema viticaste zagrade(telo). Nema ih jer joj nisu potrebne posto celo telo petlje ima samo jednu naredbu.
Na kraju, ispisujemo rezultat koristeci funkciju printf().

7.3 Promenljive tipa char

(napomena: u ovom odeljku su stvari prikazivane malo slikovito zbog lakseg razumevanja)

Char je tip promenljive koji moze da cuva samo jedan simbol(znak, slova, broj...). 'n' je simbol, preciznije slovo. Na drugoj strani „n" je niz simbola. Zasto? Uskoro cete saznati, za sada je bitno da uvidite da se simbol stavlja pod jednostruke navodnike(' ') a niz simbola(string) pod regularne navodnike(" ").
Razmotrimo inicializaciju:

```
char slovo = 'a';
```

Na prvi pogled, slovo ce sadrzati vrednost 'a'. Nije bas tako. Promenljive koje su tipa char mogu da cuvaju samo brojeve, kao i int, ali posto char ima samo jedan bajt najveca vrednost koju moze cuvati je 255. Kako onda promenljivoj char mozemo dodeliti neki simbol kada unau stvari cuva brojeve? Postoji „tabela" u kojoj se nalaze simboli(ukupno 255 njih) i svaki simbol ima svoj redni broj. Mi mozemo proslediti decimalni broj(a on se nalazi u rasponu od 0 do 255) tabeli i kao rezultat cemo dobiti simbol koji je pod tim rednim brojem. Primera radi, simbol 'a' ima vrednost 97 u toj tabeli pa ce promenljiva slovo iz gornjeg primera cuvati vrednost od 97 a ne sam simbol. Odavde se da zakljuciti da je slovo moglo biti i tipa int. Sledi program koji ce na ekran(tacnije na konzolu) ispisati sve simbole i njihove redne brojeve:

```
/*  PROGRAM TABELA_SIMBOLA */
#include <stdio.h>
void main(void)
{
      int simb;

      for(simb = 0; simb < 255; simb++)
            printf("%d %c \n", simb, (char)simb);
}
```

Obratimo paznju na red koji sadrzi funkciju prontf(). Prvo se prikazuje decimalna vrednost promenljive simb, i tu je sve jasno. Zatim smo koristeci upravljackog znaka %c rekli da na tom mestu treba da se ispise neki znak. To govori funkciji da umesto decimalne vrednosti koju bude dobila treba da ispise odgovarajuci simbol iz tabele. Kao treci argument smo naveli `(char)simb` .Ovde smo izvrsili eksplicitnu konverziju(objasnjeno u odeljku 3.3) ali to nam i nije bilo potrebno jer ce funkcija printf() sama izvrsiti konverziju(inplicitnu) iz razloga sto ona kao treci argument ocekuje promenljivu tipa char a ne tipa int.
Mali detalj: Do sada ste za prelazak u novi red koristili '\n'. Pokrenite ponovo gornji program i obratite paznju na simbol pod rednim brojem deset. Nema nista, a ispod njega je prazan red. Razlog ovome je taj sto se pod rednim brojem 10 nalazi upravo znak za novi red- '\n'.

7.4 Niz promenljivih tipa char

Pogledajmo nacine na koje mozemo deklarisati i inicializovati ovu vrstu promenljivih:

```
char TEST[] = "toxi";
char TEST[] = { 't', 'o', 'x', 'i' };
```

Bitno je znati da samo ime nizovske promenljive pretstavlja konstantni pokazivac na prvi element tog niza. Sto ce reci da cemo ako napisemo: `TEST;` ili `&test[0];` dobiti istu stvar- adresu prvog elementa niza.

Sta mislite koliko TEST, u gornjem primeru ima elemenata? Ako ih prebroite naicice te na broj 4. Ipak, TEST ima 5 elemenata. Da bi prikazali vrednost TEST-a napisacemo ovu naredbu:

```
printf("%s ", TEST);
```

Koristili smo upravljacki znak %s kojim signaliziramo da zelimo prikazati string(niz char-ova). Funkciji printf() smo predali adresu prvog elementa niza(jer smo jos predali: TEST , sto je adresa prvog elementa tog niza). Printf() prikazuje element po element... Kako ce znati kada je kraj niza? Da bi to omogucili, na kraj svakog stringa se dodaje jos jedan znak – '\0' (kosa crta unazad i nula) koji dakle signalizira kraj stringa. Sada treba da vam je jasno zasto u gornjem primeru TEST ima 5 a ne 4 elementa. I to je veoma bitna stvar!
Da bi utvrdili nauceno, napisacemo dva mala programa:

```
/*  PROGRAM VEZBA_1 */
#include <stdio.h>

void func(int a)
{
      a = 123;
}

void main(void)
{
      int broj = 7;

      printf("Vrednosti BROJ pre poziva func() je %d \n", broj);
      func(broj);
      printf("Vrednosti BROJ posle poziva func() je %d \n", broj);
}
```

Trebalo bi da vam je jasno zasto ce vrednost promenljive broj ostati ista i posle poziva funkcije func().
Drugi program je ovaj:

```
/*  PROGRAM VEZBA_2 */
#include <stdio.h>
#include<string.h>

void func(char a[])
{
      strcpy(a, "nova_vrednost");
}

void main(void)
{
      char broj[20] = "todic";

      printf("Vrednosti BROJ pre poziva func() je %s \n", broj);
      func(broj);
      printf("Vrednosti BROJ posle poziva func() je %s \n", broj);
}
```

Vrednost je promenjena, zato sto smo funkcije func() predali samo ime promenljive broj dakle adresu prvog elementa. Samim tim func() je u mogucnosti da menja vrednost broj-a (zasto je to tako vec smo govorili tako da bi trebalo da vam je to jasno).

Obratimo paznju na to da smo ukljucili jos jedan header fajl- string.h . Ovo smo uradili jer nam je bila potrebna funkcija strcpy() – ona sluzi za dodeljivanje jednog stringa drugom. Pašće vam na pamet da smo mogli da napišemo:

```
a = "nova_vrednost";
```

Medjutim to nije omoguceno pa se za dodavanje nove vrednosti stringu koristi funkcija strcpy(). O ovim funkcijama cemo govoriti u narednim poglavljima.
Za kraj cemo jos jednom napomenuti da je veoma vazno da za niz char-ova izdvojite dovoljno mesta u memoriji, dakle ogromna je greska napisati:

```
char ime[5];
strcpy(ime, "Pera Detlic");
```

Uvek obratite paznju na ovo!

8. Neke od funkcija standardne biblioteke

U prethodnom poglavlju smo pominjali funkciju strcpy() koja je definisana u string.h. U ovom poglavlju cemo objasniti kako se koriste i cemu sluze neke od funkcija(ne sve) koje su definisane u sledecim header fajlovima standardne biblioteke: string.h, ctype.h, math.h, stdlib.h. Svi ovi header fajlovi se isporučuju uz svaki valjan C kompajler tako da sa te strane nemate o cemu da brinete.

8.1 STRING.H

Naredbom `#include<string.h>` u program ukljucujemo fajl string.h u kojem su definisane razne funkcije za naratanje sa stringovima.

8.1.2 Funkcije za kopiranje stringova

strcpy():

```
char* strcpy(char *, const char *);
```

Ova je funkcija ranije pominjana i sluzi za dodeljivanje jednog stringa drugom. Kao sto se iz date deklaracije vidi, ona uzima dva argumenta: pokazivac na tip char i constantni pokazivac na tip char. Povratni tip ove funkcije je takodje pokazivac na tip char.
Da biste utvrdili znanje o tipu podataka char i string(niz char-ova), zadrzacemo se malo vise na ovoj funkciji- napisacemo kako izgleda njena definicija i zatim je proanalizirati.
Izvedba funkcije strcpy():

```
/*  PROGRAM STRCPY()1 */
#include <stdio.h>

void kopiraj(char *odr, const char *izv)
{
      while(*izv != '\0' )
            *(odr++) = *(izv++);
```

```
	*odr = '\0';
}



void main(void)
{
	//deklaracija promenljivih
	char rec1[50], rec2[50];

	//uzmi string od korisnika
	printf("Unesi recenicu-> ");
	scanf("%s", rec1);


	//kopiraj
	kopiraj(rec2, rec1);


	//prikazi rezultat kopiranja
	printf("Rec2 ima vrednost-> %s.\n", rec2);

}
```

Prvo, da bi smo koliko-toliko uprostili funkciju strcpy() stavili smo da njen povratni tip bude void a ne char*. Nasu funkciju za kopiranje smo nazvali kopiraj() a parametri su ostali isti kao i kod strcpy().
Mozda neuvidjate razlog zasto je drugi argument constantan. Odgovor lezi u tome cemu je funkcija namenjena- kopiranju drugog argumenta ,koji joj prosledimo, u prvi. To znaci da ce se vrednost prvog argumenta, koji smo nazvali ODR, menjati dok vrednost drugog argumenta treba da ostane ista. Da bi smo se „obavezali“ da necemo menjati vrednosti drugog argumenta stavili smo kljucnu rec const i zbog toga necemo moci da menjamo vrednost drugom argumentu(koji smo u gornjem primeru nazvali IZV).
Prva naredba u telu funkcije je petlja while. Ona ce se „vrteti“ dok god znak na koji pokazuje pokazivac IZV ne bude ’\0’. Kao sto smo ranije rekli simbol ’\0’ pretstavlja simbol za kraj stringa. Telo petlje se sastoji od samo jedne naredbe: `*(odr++) = *(izv++);` pa samim tim viticaste zagrade nije potrebno stavljati, ali naravno ne skodi staviti ih.
Sta radi naredba `*(odr++) = *(izv++); ?` Njom znak(element niza) na koji pokazuje IZV dodeljujemo onom elementu niza na koji pokazuje ODR, i tako vrsimo kopiranje. Takodje smo koristili operator inkrementiranja(++) da bi, po zavrsetku naredbe, podesili pokazivace da pokazuju na sledeci element niza. U jednom trenutku IZV ce pokazivati na ’\0’ sto znaci da smo dosli do kraja stringa koji treba da kopiramo pa samim tim uslov `*izv != ’\0’` nece biti ispunjen i petlja se nece vise izvrsavati. Zatim nam ostaje samo da dodelimo ’\0’ elementu na koji pokazuje ODR jer isti nije bio dodat u toku izvrsavanja petlje, a zasto je to tako trebalo bi da vam je jasno.

```
strncpy()
char* strncpy( char*, const char*, size_t )
```

Uloga ove funkcije je da u prvi argument kopira onoliko znakova drugog argumenta koliko je navedeno trecim argumentom. Drugim recima strncpy() kopira samo jedan, pocetni deo stringa(znakovnog niza). Primer koriscenja ove funkcije:

```
/*  PROGRAM STRNCPY1 */
#include <stdio.h>
#include <string.h>

void main(void)
{
      //deklaracija promenljivih
      char rec1[50], rec2[50];

      //dodeli vrednost promenljivoj rec1
      strcpy(rec1, "trivijalni tekst");

      //kopiraj prvih 8 elemenata stringa rec1 u string rec2
      strncpy(rec2, rec1, 8);
      rec2[8] = '\0';

      printf("Rec2 je -> %s.\n", rec2);
}
```

Obratite paznju da smo napisali `rec2[8] = '\0';` razlog je taj sto funkcija strncpy(), za razliku od strcpy(), ne dodaje znak za kraj stringa pa ga moramo izricito dodati.
Jedna napomena, treci argument ove funkcije je tipa size_t. Ovaj tip podatka ne moze cuvati negativne brojeve samo pozitivne(isto ponasanje ispoljava i tip unsigned int). To je ovde vrlo logicno koristiti jer kako moze da broj elemenata bude negativan? Nemoze, naravno.

8.1.3 Funkcije nadovezivanja

```
strcat()
char* strcat( char*, const char*);

strncat()
char* strncat( char*, const char*, size_t);
```

Prva funkcija sluzi za nadovezivanje jednog stringa na drugi. Recimo da imate jedan string „todic“ i jos jedan string „ nemanja“, ako hocete da ih spojite u jedan string koristicete funkciju strcat(). Sledi primer za koriscenje ove funkcije:

```
/*  PROGRAM STRCAT1 */
#include <stdio.h>
#include <string.h>

void main(void)
{
      //deklaracija promenljivih
      char rec1[50];

      //dodeli vrednost promenljivoj rec1
      strcpy(rec1, "trivijalni tekst.");

      //dodaj jos teksta promenljivoj rec1
      strncat(rec1, " I jos jedan!");

      printf("Rec1 je -> %s\n", rec1);
}
```

Ovde je sve jasno.

Koriscenje funkcije strncat() je slicno kao kod strncpy(), dakle prvi argument se „povecava“ za jedan, pocetni deo drugog argumenta. Koliki je taj drugi deo odredjuje se trecim argumentom. Takodje cete morati sami da dodate simbol '\0' prvom argumentu.

8.1.4 funkcije poredjenja

```
strcmp()
int strcmp( const char*, const char*);

strncmp()
int strncmp( const char*, const char*);
```

Pomocu ovih funkcija mozete proveriti da li su stringovi indenticni. Ako stringovi jesu isti funkcija vraca vrednost nulu, a ako su razliciti jedinicu ili neki drugi broj razlicit od nule. Primer:

```
/*  PROGRAM STRCMP1 */
#include <stdio.h>
#include <string.h>

void main(void)
{
      //deklaracija promenljivih
      char rec1[50], rec2[50];

      //dodeli vrednost promenljivim
      printf("Unesite dve reci-> ");
      scanf("%s %s", rec1, rec2);

      //uporedi tekst
      if( strcmp(rec2, rec2) )
            printf("Reci su razlicite!");
      else
            printf("Reci su potpuno iste!");
}
```

Funkcijom strncmp() moguce je porediti samo par elemenata sa pocetka nizova. Koliko elemenata ce biti poredjeno se odredjuje trecim argumentom.

Jos jednu funkciju iz string.h cemo objasniti. To je strlen(). Ova funkcija vraca duzinu znakovnog niza koji joj prosledimo. Ako imamo:

```
char rec[50] = "neki string...";
printf( "Duzina ovog stringa je %d. \n",  strlen( rec));
```

Na ekran(pravilnije receno na konzolu) cese ispisati koliko elemenata ima ovaj string.

8.2 MATH.H

U ovom header fajlu definisane su razne funkcije za rad sa brojevima kao sto su korenovanje, dobijanje sinusa, kosinusa idr. Objasnicemo samo par najosnovnijih jer zaista matematika smara.

8.2.1 Trigonometrijske funkcije

Za dobijanje sinusa koristite funkciju sin(). Za dobijanje kosinusa cos() a za dobijanje takgensa tan(). Sve ove funkcije kao jedini argument uzimaju promenljive tipa double.

8.2.2 Stepenovanje vrednosti

Za stepenovanje vrednosti koristite funkciju `double pow( double, double)`. Kao sto se vidi pow() uzima dva argumenta od cega je prvi broj koji treba stepenovati a drugi argument je eksponent tj.broj kojim se stepenuje. Ako napisemo pow(10,3) kao povratnu vrednost dobicemo 1000.
Za dobijanje kvadratnog korena nekog broja koristi se sqrt(). Ona vraca vrednost tipa double a mana joj je sto nemoze vaditi na primer treci ili cetvrti koren vec samo drugi(kvadratni).

8.2.3 Zaokruzivanje na celobrojnu vrednost

Objasnicemo 4 funkcije:

`double fabs( double);` Sluzi za dobijanje apsolutne vrednosti broja(dakle vraca uvek pozitivnu vrednost)

`double fmod(double, double);` Operator ostatka(%) nemoze se primenjivati na realne vec samo na cele brojeve. Pomocu ove funkcije to se ogranicenje prevazilazi.

`double ceil( double);` Prosledjeni argument ova funkcija zaokruzuje na najblizu vecu celobrojnu vrednost. Na primer ako prosledite broj 10.01 funkciji ceil(), ona ce vam vratiti broj 11.

`double floor( double);` Radi potpuno isto kao i ceil() s'tom razlikom sto prosledjeni argument zaokruzuje na najblizu manju vrednosti. Tako da ako napisete:

```
int x;
double Y = 4.764458;
x = floor(Y);
```

X ce imati vrednost 4.

Primecuje se nedostatak funkcije koja bi zaokruzivala vrednosti promenljivih na najblizu celobrojnu vrednost. Dakle ako imate 4.7 ili 4.5 ili 4.9, da broj bude zaokruzen automatski na 5 a ako imamo 4.2 ili 4.4999 ili 4.001, da broj bude zaokruzen na 4 stim da u oba slucaja pozivate jednu funkciju posto neznate dali treba pozvati floor() ili ceil() (recimo da korisnik zadaje broj koji treba zaokruziti- kako ce te onda znati koju funkciju da pozovete?). Sa ovim problemom sam se susreo prilikom pisanja programa koji treba da od prosecne ocene koju ucenik ima izvede zakljucnu ocenu(koja dakle mora biti zaokruzena vrednost). Evo kako izgleda „interesantno“ resenje:

```
int Ceo(double x)
{
      if( (x - floor(x)) < 0.5 )
            return (int)floor(x);
      return (int)ceil(x);
}
```

Preporucujem da analizirate kod i shvatite kako radi. Naravno, postoji jos nacina da se ovaj problem resi.

8.3 CTYPE.H

U ovom header fajlu su funkcije za baratanje sa simbolima(char-ovima), za razliku od string.h u kom se nalaze funkcije za nizove char-ova(stringova). Glavna primena ovih funkcija je kod obrade podataka koji su zapisani u fajlovima a o tome ce kasnije biti reci.

8.3.1 Pripadnost simbola grupi

Simbol moze biti broj, slovo, znak razmaka ili neki drugi znak. Sledece funkcije vracaju vrednost 1 da bi signalizirale tacnost a nulu za netacnost:

`int isalpha( int);` Funkcija proverava da li je prosledjeni znak slovo i ako jeste vraca vrednost 1. Evo jednog primera:

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


void main(void)
{
      char a = 'r';

      if( isalpha(a) )
            printf("%c je slovo!\n", a);
      else
            printf("%c nije slovo!\n", a);
}
```

U ovom slucaju poruka koja ce biti ispisana je da 'r' jeste slovo. I to je logicno ali sta bi bilo ispisano da umesto `char a = 'r';` stoji `char a = 144; ?` Ako ste pomislili da ce se ispisati da „nije slovo“ grdno ste se presli. Setite se poglavlja 7.3 u kojem smo rekli da niti jedan tip podatka nemoze sadrzati znakove vec samo broja a da postoji tabela u kojoj svaki simbol(znak) ima svoj redni broj preko koga mu se pristupa. Mi bi smo dakle promenljivoj A dodelili vrednost 144 sto znaci da ce A sadrzati taj broj. U poglavlju 7.3 smo napisali program koji nam ispisuje tabelu simbola i pod kojim se rednim brojem ti simboli nalaze, pogledajte sta je pod rednim brojem 144. Slovo 'r'! Ako vam razlog ovom ponasanju nije jasan procitajte poglavlje 7.3 jos jednom.

`int isdigit( int);` Funkcija radi potpuno isto kao i isalpha() s' tom razlikom sto proverava dali je prosledjeni argument broj ili ne. Ako jeste broj funkcija vraca vrednost 1, u suprotnom vraca 0.

`int isalnum( int);` Ukoliko je prosledjeni argument slovo ili broj, funkcija vraca vrednost 1 a u suprotnom 0.

`int isspace( int);` Pomocu isspace() vrsimo proveravanje dali je prosledjeni argument neki od znakova razmaka. Znakom razmaka se smatraju: novi red('\n'), razmak(' ') i tabulacija('\t'). Ako argument jeste znak razmaka vraca se vrednost 1 u suprotnom dobicemo nulu.

8.3.2 Odredjivanje i menjanje velicine slova

Cetri su bite funkcije:

`int islower( int);` Proverava dali je prosledjeni argument malo slovo, ako jeste vraca rezultat 1 ako nije vraca nulu.

`int isupper( int);` Proverava dali je prosledjeni argument veliko slovo. Vraca vrednost 1 ako jeste i 0 ako nije.

`int tolower( int);` Vrednost koju vraca ova funkcija je „smanjeno“ slovo. Primer koriscenja:

```
char slovo = 'T';
slovo = tolower(slovo);
```

sada ce promenljiva SLOVO sadrzati znak 't'(pravilije bi bilo reci “ SLOVO ce sadrzati redni broj znaka 't' “ ali ovo zvuci glupavo i glomazno pa se i ne koristi).

`int toupper( int);` Ista stvar kao i tolower() uz razliko sto slovo biva „povecano“.

8.4 Jos funkcija za komuniciranje sa korisnikom

Pored printf(), tekst mozete prikazivati na ekran(konzolu) i pomocu sledecih funkcija(sve su definisane u stdio.h):

`int putchar( int);` Pomocu ove funkcije mozete prikazati samo jedan simbol na ekran, ali ne i niz znakova. Na primer napisite `putchar('\a');` i ćućete „beep“ zvuk na vasem racunaru.

`int puts( const char*);` Sluzi za prikazivanje teksta na ekran. Nesto slicno poput printf() uz razliku sto nemoze da prikazuje vrednosti promenljivih vec iskljucivo tekst.

Rekli smo ranije da je scanf() vrlo „pipava“ funkcija u tom pogledu sto zahteva da podaci budu uneseni korektno a korisnik programa moze slucajno(ili namerno) da unese pogresan podatak, tada ceo program pada u vodu.Na primer umesto da otkuca 2 korisnik pritisne E.... Da bi se ovaj rizik izbegao koriste se neke druge funkcije za uzimanje podataka jedna od njih je:
`char* gets( char*);` koja sluzi za uzimanje reda teksta od korisnika. Naime, mozda ste primetili da koristeci scanf() ne mozete ucitati ceo red teksta vec najvise samo jednu rec(sve do prvog znaka razmaka). Za razliku od nje, gets() uzima ceo tekst koji je unesen nebitno dali on sadrzi brojeve, slova, razmake ili neke druge znake.
Napisacemo program koji od korisnika trazi ime i prezime a zatim menja velicinu slovima imena i prezimena, koristeci pritom neke od funkcija pomenutih u ovom poglavlju.

```
/*  PROGRAM IME_I_PREZIME */
#include <stdio.h>
#include <ctype.h>
#include<string.h>

//prototip funkcije
int MalaSlova(char *text);

void main(void)
```

```
{
	//deklaracija
	int duzina = 0;
	int i=0, n;
	static char pom[50],ime[25], prezime[25];

	//trazi ime i prezime dok god nebudu unesena malim slovima
	do {
		printf("Unesite vase ime i prezime malim slovima-> ");
		gets(pom);
	}while(MalaSlova(pom));


	/* Podeli POM na IME i PREZIME */
	//preskoci razmake sa pocetka ako ih ima
	while( isspace( pom[i]) ) ++i;

	//sada POM[i] pokazuje na prvo slovo imena, "uvecaj" slovo
	pom[i] = toupper(pom[i]);

	//trazi kraj imena( do prvog razmaka) i kopiraj vrednost u IME
	while( !isspace(pom[i]) ) ++i;
	strncpy(ime, pom, i);  ime[i] = '\0'; //propust1

	//preskoci sve razmake izmedju imena i prezimena
	while( isspace( pom[i]) ) i++;

	//povecaj prvo slovo prezimena
	pom[i] = toupper(pom[i]);

	//kopiraj prezime u promenljivu PREZIME
	for(n=0; !isspace(pom[i]) && pom[i] != '\0'; ++i)
		prezime[n++] = pom[i];
	prezime[n] = '\0';

	//prikazi rezultat
	printf("Vase ime je: %s, a prezime %s.\n", ime, prezime);

}

//definicija f-je MalaSlova()
int MalaSlova(char *text)
{
	for(; *text != '\0'; ++text )
		if( !islower(*text) && !isspace(*text) ) return 1;
	return 0;
}
```

Ovaj kod namerno necemo komentarisati i objasnjavati. Reci cemo samo da sve sto treba da znate da bi razumeli ovaj program ste vec imali priliku da naucite u ovom tutrialu, i jos jednu stvar: Kod je namerno „neusavrsen“, na vama je da eventualne propuste ispravite i da ceo kod poboljsate koliko je to moguce. Bice vam odlicna vezba. Pri tom obratite paznju na liniju koda u kojoj stoji komentar „`//propust1`“ , u promenljivu IME bice upisani i eventualni razmaci(ako ih je korisnik uneo). Napisite funkciju koja ce se pozivati u sledecoj liniji koda da ispraviti ovaj propust „brisuci“ nepotrebne znake razmaka. Srecno!

8.5 Funkcije za dinamicko upravljanje memorijom

Sve funkcije koje cemo spominjati u ovom odeljku se nalaze u header fajlu STDLIB.H.
Do sada ste naucili sta su pokazivaci i rekli smo da oni mogu pokazivati na neku drugu promenljivu, tacnije na mestu u memoriji koje je odvojeno(rezervisano) za tu drugu promenljivu. Medjutim, koristeci funkciju malloc() iz stdlib.h mi mozemo da rezervisemo odredjeno „parce“ memorije- memorija ce biti zauzeta ali je ni jedna promenljiva nece koristiti. Pa ste ce nam onda? Naime, pomenuta funkcija malloc() kao povratnu vrednost daje adresu memorije koja je upravo zauzeta, a koji tip podatka jedini moze cuvati adrese? Naravno, pokazivaci. Pomocu ovog mehanizma u mogucnosti smo dinamicki, u toku izvrsavanja programa, da alociramo(zauzimamo) odredjene kolicine memorije i na njih ce pokazivaci pokazivati pa cemo moci da u tu memoriju skladistimo podatke. Sledi primer za dinamicku alokaciju memorije.

```
/*  PROGRAM MEMORIJA_1 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void main(void)
{
      //deklaracija
      char *text;

      //alociranje memorije
      text = malloc( sizeof(char) * 50);

      //upisi neku vrednost u memoriju na koju pokazuje TEXT
      strcpy(text, "jos jedna trivijalna poruka!");

      //prikazi vrednost
      printf("TEXT-> %s\n", text);

}
```

Sve sem naredbe u kojoj pozivamo malloc() treba da vam je jasno.
Funkciji malloc() smo kao argument predali jednu pomalo cudno vrednost::

```
sizeof(char) * 50
```

sizeof je operator C jezika(kao sto su i recimo + ili / operatori) a njegova uloga je da odredi koliko memorije zauzima neki tip podatka. U ovom slucaju trazili smo koliko bajtova zauzima tip podatka char i tu vrednost pomnozili sa 50. Time cemo dobiti broj bajtova koji je potreban da bi se smestilo 50 char-ova u memoriju. Drugim recima, ako zelimo da rezervisemo memoriju za npr. 10 promenljivih tipa INT, ne mozemo funkciji malloc() predati vrednost 10. Razlog je taj sto bi smo na taj nacin trazili od funkcije da rezervise 10 bajtova a ne 10 puta po onaj broj bajtova koji zauzima tip INT. Pa zato moramo da napisemo:

```
sizeof(int) * 10
```

Dakle ovo govori prevodiocu(kompajleru) „izracunaj koliko bajtova zauzima tip INT a zatim taj broj pomnozi sa 4“ i time cemo dobiti broj bajtova koji nam je potreban da bi smestili 10 INTova u memoriju racunara.
Kada je potrebna kolicina memorije rezervisana(alocirana) funkcija malloc() nam vraca adresu te memorije i mi je u gornjem primeru dodeljujemo pokazivacu TEXT. Zatim upisujemo vrednosti u alociranu memoriju pomocu funkcije strcpy(), da bi na kraju i iscutavali vrednost iz te memorije. Zakljucak je da se sa dinamicki zauzetom memorijom moze baratati

kao i sa „normalno“ zauzetom memorijom(memorija koja je zauzeta pravljenjem obicnih promenljivih).

Sad bi trebalo da vam je jasno zasto i kako ovo radi. Ostaje da se vidi zasto bi uopste koristili dinamicko alociranje. Razloga je vise a ako se budete posvetili programiranju nesto „dublje“ i sami cete ih zakljuciti. Ja cu reci samo jedan od njih- kompaktnije iskoriscenje memorije. Imacemo jedan primer sa ovakvim pristupom ali pre toga morate da znate jos (samo) sta je realokacija i „curenje memorije“.

Curnje memorije je jedina losa stvar kod dinamickog upravljanja memorijom. Sledi primer o ovome.

```
/*  PROGRAM MEMORIJA_2 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void main(void)
{
      //deklaracija
      char *text;

      //alociranje memorije
      text = malloc( sizeof(char) * 50);

      //upisi neku vrednost u memoriju na koju pokazuje TEXT
      strcpy(text, "jos jedna trivijalna poruka!");

      //sad cemo se malo-mnogo prevariti
      text = malloc( sizeof(char) * 20);

      //prikazi vrednost
      printf("TEXT-> %s\n", text);

}
```

U odnosu na prosli program, ovaj smo povecali za samo dve linije koda, i to ove:

```
//sad cemo se malo-mnogo prevariti
text = malloc( sizeof(char) * 20);
```

Napasalis smo na curenje memorije! Evo i zasto: prvo smo pri pocetku programa alocirali jedan blok memorije i na taj blok memorije je pokazivao samo pokazivac TEXT. Sada smo alocirali jos jedan blok memorije(koji je nesto manji od prethodnog zauzima mesto za 20 char-ova, ali to uopste nije bitno) i adresu tog bloka dodelili opet pokazivacu TEXT. Ovo radi i sa te strane je sve OK. Medjutim, na prvi blok memorije koju smo zauzeli sada vise ni jedan pokazivac ne pokazuje, tako da joj vise nemozemo pristupiti. Takva memorija je „procurela“ memorija i ona je stetna jer zauzima jedan deo memorije racunara a nicemu ne sluzi. U ovom primeru to i nije tako opasno jer je to mala kolicina memorije ali sta da je do „curenja“ doslo u nekoj funkciji koja se u toku rada programa poziva nekoliko hiljda puta? Program bi „pojeo“ svu raspolozivu memoriju racunara!
Resenje ovoga problema lezu u pravovremenom oslobadjanju memorije koja nam vise nece biti potrebna. Oslobadjanje memorije vrsi se pomocu funkcije free(). Njeno koriscenje je krajnje jednostavno- dovoljno je da joj predate pokazivac na memoriju koju treba osloboditi(dealocirati). Sledi ispravljena verzija predjasnjeg programa.

```
/*  PROGRAM MEMORIJA_3 */
#include <stdio.h>
#include <stdlib.h>
#include<string.h>

void main(void)
{
      //deklaracija
      char *text;

      //alociranje memorije
      text = malloc( sizeof(char) * 50);

      //upisi neku vrednost u memoriju na koju pokazuje TEXT
      strcpy(text, "jos jedna trivijalna poruka!");

     //dealociramo memoriju koju smo zauzeli pa nema curenja memorije
      free(text);

      //sad je sasvim korektno dodeliti novu adresu pokazivacu TEXT
      text = malloc( sizeof(char) * 20);

      //uradi nesto glupo
      text[0] = 'T';
      text[1] = '\0';

      //prikazi vrednost
      printf("TEXT-> %s\n", text);
}
```

Prosto i jednostavno.
Ostaje jos da shvatite sta je realokacija zauzete memorije.

Predpostavimo da smo dinamicki zauzeli jednu povecu kolicinu memorije. Zatim smo u tu memoriju smestili podatke koje je na primer uneo korisnim programa. Mi mozemo izracunati koliko nam memorije stvarno treba za cuvanje dobijenih podataka,a onu ostalu memoriju koja pretstavlja visak, pa je samim tim bespotrebna, mozemo osloboditi. Za to koristimo funkciju realloc() iz stdlib.h. Primer za realokaciju dinamicki zauzete memorije:

```
/*  PROGRAM MEMORIJA_4 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void main(void)
{
      //deklaracija
      char *text;

      //alociranje memorije
      text = malloc( sizeof(char) * 50);

      //uzmi ime i prezime od korisnika
      printf("Unesi ime i prezime-> ");
      gets(text);
```

```
	//prikazi koliko memorije sada zauzima text(u bajtovima)
	printf("Kolicina sada zauzete memorije je %d bajtova.\n",
		sizeof(char) * 50);

	//"START". realociraj memoriju
	text = realloc( text, sizeof(char) * (strlen(text) + 1));

	//nova kolicina memorije je
	printf("Posle realociranja memorije ona zauzima %d bajtova!\n",
		sizeof(char) * (strlen(text)+1));
}
```

Kod iznad komentara koji pocinje sa „START“ treba da vam je jasan, pa cemo objasniti samo kod koji sledi posle njega.
Prvo koristeci funkciji realloc realociramo zauzetu memoriju. Ova funkcija uzima dva parametra: pokazivac i decimalni broj. Kao prvi argument trebamo da predamo pokazivac na memoriju kojoj zelimo da promenimo velicinu(da je realociramo). U ovom slucaju smo predali TEXT zato sto zelimo da promenimo broj bajtova(velicinu) bloku memorije na koji on pokazuje, konkretno ovde-da taj blok smanjimo. Naravno, pomocu realloc() moze se i povecavati i smanjiti zauzeti blok memorije, dok za potpuno oslobadjanje zauzetog bloka memorije moramo da koristimo free() funkciju. Kao drugi argument smo predali „ `sizeof(char) * (strlen(text) + 1)` “. Pomocu funkcije strlen() cemo dobiti koliko znakova ima u TEXTu, ali posto svaki string ima i zavrsni znak '\0'(a taj znak strlen() ne broji) moramo dodati i ono „ +1 “ . `(strlen() + 1)` smo stavili izmedju zagrada jer bi se u suprotnom prvo izvrsilo mnozenje a tek onda sabiranje.
Funkcija realoc(), poput malloc(), vraca adresu realocirane memorije, pa smo mi tu adresu dodelili pokazivacu TEXT.
Na kraju, pomocu printf() prikazujemo novu, „optimizovanu“ kolicinu zauzete memorije. Na ovaj nacin smo ustedeli malo memorije. Kao sto smo vec rekli, razloga za koriscenje dinamickog alociranja ima vise a ovo je bio samo jedan od njih....

8.6 Generisanje random broja

Naci cete se u situaciji da vam zatreba nasumicni(random) broj a u ovom odeljku cemo vam reci sve sto treba da znate o tome. Sledi najjednostavniji primer generisanja random broja:

```
/*Program RAND1*/
#include <stdio.h>
#include <stdlib.h>

void main(void)
{
	int broj;
	broj = rand();
	printf("Dobijeni broj je %d\n", broj);
}
```

Prvo smo deklarisali promenljivu BROJ koja je tipa INT. Zatim smo joj dodelili vrednost koju vraca funkcija rand(), da bi na kraju prikazali koji smo broj dobili. Kao sto se moze video funkcija rand() sluzi za generisanje random broja. Ona je vrlo prosta za koriscenje jer ne uzima ni jedan parametar a definisana je u header fajlu stdlib.h. Medjutim ako pokrenete program videcete da BROJ stalno dobija istu vrednost bez obzira koliko vi puta pokrenuli program. To se moze resiti ovako:

```
/*Program RAND2*/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void main(void)
{
      int broj;
      srand( time(NULL));
      broj = rand();
      printf("Dobijeni broj je %d\n", broj);
}
```

Dve su razlike: dodali smo „ `#include<time.h>` “ i „ `srand( time(NULL));` “. Prvu izmenu smo nacinili jer je u time.h definisana funkcija time() koju smo koristili u programu(napomena: funkcije iz time.h necemo objasnjavati).
Funkcija srand() definisana je u stdlib.h i sluzi, slikovito receno, da malo „provrti“ brojeve koje ce koristiti rand(). Nije to bas tako ali zaista nije potrebno da znate kako ovo radi, pa vas necemo sa tim opterecivati. Napomenimo ovde znacenje reci NULL- to je sinonim(tacnije-makro) za pokazivac na tip void(dakle: void *). Sa ovim se (opet) ne morate opterecivati, recicemo samo da ste umesto njega mogli da stavite i nulu jer je vrednost NULL u stvaru nula.
Epilog ove izmene je da cemo dobijati stalno drugacije brojeve, sto nam je i bio cilj. Ipak, opseg u kojem ce dobijeni random brojevi biti nije ogranicen. Sta ako nam treba random broj koji je nula ili jedan? Primer:

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

void main(void)
{
      int broj;
      srand( time(NULL));
      broj = rand() % 2;
      printf("Dobijeni broj je %d\n", broj);
}
```

Ovde niceg nepoznatog nema a zeljeni opseg random broja smo dobili tako sto smo, promenljivoj BROJ dodelili ostatak pri deljenju broja koji vraca rand() sa brojem dva. Logicno ostatak deljenja sa dva je uvek 0 ili 1, zar ne? Kao vezbu, napisite program koji ce generisati 20 random brojeva(pomocu petlje) u opsegu od 0 do 1, i koji ce izbrojati koliko se puta pojavljuje 0 a koliko puta 1.

9. Strukture

Svi do sada upotrebljavani tipovi podataka u ovom tutrialu bili su osnovni(prosti) tipovi. To su na primer INT, FLOAT; CHAR... Nasuprot njima, postoje i izvedeni(slozeni) tipovi podataka a jedan vid tih podataka su i strukture.Deklaracija strukture izgleda ovako:

```
struct ime_strukture {
      int clan1;
      char clan2;
      float clan3[10];
      int clanx = 4579;     /* itd... */
};
```

Rec struct govori da je u pitanju struktura. Sledi ime strukture i zatim blok naredbi(telo strukture) a kao sto znate, blok naredbi je oivicen sa { i }. Unutar tela strukture definisani su clanovi te strukture, njihov broj je neogranicen a mozete napraviti strukturu koja nema ni jedan clan(mada je ta struktura vise nego beskorisna). U ovom primeru, clanovi strukture su prosti tipovi podataka.
Kada i zasto koristiti strukture? Recimo da hocemo napraviti program koji uzima neke osnovne podatke o korisnicima- kao sto su ime, prezime i godiste. To mozemo uraditi i bez struktura ali bi kod bio dosta losije organizovan. Naime, svaki korisnik koji bude upisivao svoje podatke ce morati da upise ime ,prezime i godiste. To nam govori da bi bilo zgodno da napravimo novi tip podatka(strukturu) koji ce se zvati npr „Osoba“ i koji ce u sebi sadrzati podatke o imenu, prezimenu i godistu korisnika. Napisimo tu strukturu:

```
struct Osoba_st {
      int god;
      char ime[25];
      char prezime[25];
};
```

Ovim smo napravili ovi tip podatka koji se zove Osoba_st. To znaci da cemo moci da napravimo(deklarisemo) promenljive ovog tipa, isto kao sto deklarisemo promenljive tipa INT, na primer. Evo kako bi to izgledalo:

```
struct Osoba_st korisnik;
```

Na ovaj nacin smo deklarisali promenljivu koja se zove „korisnik“ koja je tipa Osoba_st. Da nema onog „struct“ ispred Osoba_st, deklaracija bi bila potpuno ista kao i za tip podatka npr. INT ili FLOAT. Medjuim to „struct“ moramo pisati da bi kompajler(prevodilac) znao da se radi o izvedenom tipu podatka. Napisimo programa koji koristi ovu strukturu:

```
/* program STRUKTURE1  */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

//napravi(definisi) novi tip podatka( strukturu)
struct Osoba_st {
      int god;
      char ime[25];
      char prezime[25];
};


//polazna tacka programa
void main(void)
{
      //deklarisi promenljivu tipa Osoba_st
      struct Osoba_st korisnik;
```

```
	//deklarisi "obicnu" promenljivu tipa INT
	int pom;

	//uzmi podatke i smesti ih u promenljivu "korisnik"
	printf("Unesi godiste: ");
	scanf("%d", &pom);
	korisnik.god = pom;
}
```

Definicija strukture Osoba_st treba da vam je jasna. U funkciji main()smo napravili promenljivu „korisnik“ koja je tipa Osoba_st. Zatim deklarisemo pomocnu promenljivu(mada smo mogli kod napisati tako da nam ona i netreba) „pom“. Posle toga od korisnika trazimo da unese svoje godiste(pozivajuci scanf()) i taj podataka smestamo u „pom“. Sledeca linija koda je interesantna: Kada smo definisali tip podatka „Osoba_st“rekli smo da on sadrzi promenljive pod imenom god, ime i prezime, a u ovoj liniji koda pristupamo clanu „god“ i dodeljujemo mu vrednost koju ima promenljiva „pom“. To pristupanje je izvedeno pomocu operatora tacka(.) Dakle struktura je „skup“ vise promenljivih razlicitih tipova, a mi preko operatora „tacka“ mozemo da pristupamo tim promenljivim i menjamo im vrednosti. Kao vezbu zavrsite program tako da u „korisnik“ takodje upise ime i prezime. Ne zaboravite da se za kopiranje stringova koristi funkcija strcpy(), koja je objasnjena u prethodnom poglavlju.

Takodje mozete praviti i nizove izvedenih tipova podataka a njihova upotreba je potpuno ista kao i kod prostih tipova:

```
struct Osoba_st bzv[100];
```

Pristupanje clanovima strukture je takodje prosto:

```
bzv[99].god = 1989;
```

Inicijalizacija clanova strukture se moze izvesti na sledeci nacin:

```
struct Osoba_st korisnik = { 10, "Nemanja" , "Todic" };
```

Dakle otvorena viticasta zagrada i zatim upisujete vrednosti kojim zelite da inicijalizujete promenljive. Vrednosti su razdvojene zarezom i moraju se navoditi istim redom kojim su deklarisane u definiciji strukture. Tj.ako ste u strukturi prvo napravili clan „god“ onda ce prva vrednost koju napisete izmedju viticastih zagrada biti dodeljena upravo clanu „god“ i sve tako redom.

O strukturama ima jos mnogo,mnogo toga da se kaze medjutim posto se ovaj tutrial bavi samo stvarima koje se rade u skoli, zadrzacemo se na do sad recenom. Bitno je za sada da usvojite ovo osnovno shvatanje struktura i da ih primenjujete kad se god za to ukaze potreba.

10. Rad sa fajlovima

U ovom poglavlju cete nauciti da smestate podatke u fajl kao i da iz njega iscitavate podatke. Skoro svi programi barataju sa fajlovima u vecoj ili manjoj meri, pa je s'toga veoma bitno da usvojite sve sto bude receno.

10.1 Standardni tokovi i pojam Bafera

U dosadasnjem delu tutriala koristili smo dva od tri standardna toka- jedan za ispisivanje podataka na konzolu(ekran) i jedan za uzimanje podataka sa konzole. Tok za uzimanje podataka se naziva „ stdin “ a tok za ispis podataka se na konzolu se naziva „ stdout “. Tok „ stdin “ je povezan sa konzolom i to ponasanje ne mozemo promeniti, isto vazi i za tok „ stdout “. Posledica ovoga je da ce „ stdout “ uvek prikazivati tekst na konzolu a nece ga na primer sacuvati u neki fajl, slicno tome „ stdin “ ce kao unesene podatke uzeti samo ono sto je korisnik uneto u konzolu. Da bi pojasnili sta je to tok uzecemo u razmatranje funkciju printf(). Kada joj predamo neki text, na primer „Ovo je poruka!“ ona ce taj text „proslediti“ na tok „ stdout “ pa ce samim tim text biti ispisan na konzoli.
Jos jedna je bitna stvar- prikazivanje text na konzolu je veoma spor proces pa se, da bi se prikazivanje ubrzalo, text ne prikazuje cim bude „prosledjen“ vec se baferuje u redove tj. tekst se cuva u baferu(bafer je naziv za niz promenljivih tipa char, dakle za string) sve dok se ne dodje do znaka za kraj red- '\n' ili za kraj stringa- '\0'. Tad se text prikazuje a bafer se „prazni“ da bi oslobodio prostor za sledece „skladistenje“ prosledjenog texta.

Do sada smo naveli nekoliko funkcija za uzimanje podataka od korisnika ali funkciju getchar() smo namerno preskocili jer niste bili upuceni u to kako radi bafer. Ova funkcija sluzi za uzimanje jednog znaka od korisnika, tacnije iz bafer ali pojasnicemo na sledecem primeru:

```
/* program Bafer1  */
#include <stdio.h>

void main(void)
{
      char zn;
      getchar();
      zn = getchar();
      printf("->%c",zn);
}
```

U funkciji main() smo pozvali getchar() da bi zatrazili od korisnika da unese jedan znak i pretpostavimo da je korisnik uneo bas samo jedan znak. Zatim smo pomocu printf() na ekran istampali „->“ a takodje i vrednost toju sadrzi promenljiva ZN. Mozda ocekujete da ce program od vas traziti da dva puta unesete neki znak ali to se nece dogoditi. Naime, kada smo prvi put pozvali getchar(), ta funkcija je pokusala da uzme znak iz bafera a ne od korisnika. Ali posto je bafer u tom trenutku bio prazan, program je morao da od korisnika zatrazi podatke koje ce smestiti u bafer a zatim ce, dakle kada korisnik unese podatke, funkcija getchar() pokusati ponovo da iscita prvi znak iz bafera(a to je logicno prvi znak koji je korisnik uneo) – ovoga puta to izcitavanje ce biti uspesno. Znak koji funkcija getchar() bude nasla u baferu ce biti vracen kao rezultat.
U main() funkciji smo zatim jos jednom pozvali getchar() i evo sta se sada dogadja: getchar() opet pokusava da uzme znak iz bafera, ovoga puta naravno nece pokusati da uzme prvi znak koji se nalazi u baferu vec drugi. Tacnije, postoji pokazivac na elemente bafera i taj se pokazivac inkrementira kod svakog poziva ove funkcije, pa ce on uvek pokazivati na sledeci element u baferu- getchar() samo proverava dali je znak na koji ovaj pokazivac pokazuje ispravan tj. da li smo dosli do kraja texta koji je smesten u bafer(kao sto vec treba da znate,

znak koji predstavlja kraj texta je '\0') pa ako nismo dosli do kraja texta getchar() kao rezultat vraca element na koji pokazuje pokazivac a u suprotnom od korisnika se trazi da ponovo unese podatke.
I sta mislite dali ce korisnik morati ponovo da unesete podatke(ako pretpostavimo da je prvi put uneo samo jedan znak)? Nece morati! Razlog je jednostavan- kao sto smo rekli baferovanje se vrsi u redove(i za ispis i za citanje) to znaci da mi u bafer nismo upisali samo slovo koje smo otkucali vec i smo zatim morali da pritisnemo ENTER a enter pretstavlja znak za novi red, pa je i taj znak unesen u bafer. Dakle promenljiva zn ce cuvati znak '\n'.
Da bi ste se uverili da ste ovo zaista shvatili razmotrite kako i zasto radi sledeci primer:

```
/* program Bafer1  */
#include <stdio.h>

void main(void)
{
      char zn;

      while( zn = getchar() )//unosite tekst proizvoljne duzine...
            printf("%c", zn); //mogli smo napisati i " putchar(zn); "
}
```

Postoji jos jedan standardni tok povezan sa konzolom- standardni tok za greske „ stderr“ ali njega necemo uzeti u razmatranje.

10.2 Struktura FILE i funkcije za otvaranje novih tokova

Kao sto ste mogli da zakljucite ne postoji ni jedan standardni tok koji je povezan sa fajlovima na Hard Disku(ili nekoj drugoj jedinici za smestanje podataka) medjutim mozemo ih „otvoriti“ u svojim programima ako za to imamo potrebu(a vecina programa ima potrebu za ovim vidom skladistenja podataka). Sturo receno fajl je niz bajtova u memoriji kojim je dato ime, on ima svoju adresu u memoriji i znak za svoj kraj(slicno kao sto i string ima znal '\0'). Na primer ovaj tutrial je fajl, pesme su fajlovi i tako dalje. Da bi „komunicirali“(citali ili upisivali podatke u fajl) potrebna nam je pokazivac na tip FILE. FILE je struktura(dakle izvedeni tip podatka) koja je definisana u header fajlu „ stdio.h “ i ona sadrzi sva potrebna svojstva da bi mogli preko nje da „komuniciramo“ sa nekim fajlom. Samu konstrukciju ovog izvedenog tipa podatka i kako on radi nije neophodno da znate(vec samo da ga koristite) pa vas sa tim necemo opterecivati.
Kada smo napravili pokazivac na FILE mi treba da ga „povezemo“ sa nekim fajlom na HDu, tj. treba da „namestimo“ pokazivac da pokazuje na zeljeni fajl a to se zove „otvaranje toka ka fajlu“. Kada smo to uradili imacemo potpunu kontrolu nad odredjenim fajlom tako da mozemo upisivati i citati podatke iz njega. Otvaranje toka se vrsi pomocu funkcije fopen() koja vraca adresu na kojoj se nalazi fajl sa kojim treba da „komuniciramo“. Demonstracija gore recenog:

```
/* program Fajl1  */
#include <stdio.h>

void main(void)
{
      FILE *pf;
      int broj = 13041989;

      //povezi se fajlom
      pf = fopen("MojFajl.txt", "w");
```

```
	//upisi neke podatke u fajl
	fprintf( pf, "Broj je %d...", broj);

	//zatvori tok
	fclose(pf);
}
```

Prvo stvaramo pokazivac na FILE, zatim pomocu fopen() otvaramo fajl „MojFajl.txt“. Drugi argument pretstavlja „mod“ u kome zelimo da otvorimo fajl(za citanje, pisanje, azuriranje idr.), u ovom slucaju smo predali je “w“ sto znaci da fajl otvaramo za upisivanje(sledi tabela svih znakova koje mozete koristiti kao i njihovo znacenje).
Zatim smo koristili funkciju fprintf() koja ima slicnu sintaksu kao i printf() s’ tom razlikom sto ne mora da ispisuje podatke u stdout vec ih moze upisivati i u tok ka nekom fajlu(a u koji tok se podaci upisuju se navodi prvim argumentom, u ovom slucaju to je PF). Kasnije cemo objasniti neke od osnovnih funkcija za baratanje sa fajlovima.
Na kraju zatvaramo otvoreni tok pozivajuci funkciju fclose() koja kao argument uzima tok koji treba da zatvori.

MOD	OPIS
r	Otvara textualni falj za citanje
w	Zamenjivanje ili kreiranje textualnig fajla za upis
a	Otvaranje ili kreiranje textualnog fajla za dopisivanje
rb	Otvaranje binarnog fajla za citanje
wb	Zamenjivanje ili kreiranje binarnog fajla za upis
ab	Otvaranje ili kreiranje binarnog fajla za dopisivanje

10.3 Funkcije za baratanje sa fajlovima

Objasnicemo 6 najcesce koriscenih funkcija.

int fprintf(FILE *, „upravljacki niz“,parametar1,parametar2,...);
–Nacin koriscenja ove funkcije je veoma slican sa funkcijim printf() sa tom razlikom sto se kao prvi parametar unosi pokazivac na FILE. Podaci ce biti upisani u fajl na koji taj pokazivac pokazuje a ne na „ stdout“.

int fscanf(FILE *, „upravljacki niz“, parametar1,parametar2,...);
–Slicnosti i razlike izmedju scanf() i fscanf() su kao i kod printf() i fprintf(). Dakle kao prvi argument navodite ime fajla iz kojega se podaci iscitavaju a sve ostalo ostaje isto.

int fputc(int, FILE *)
–Sluzi za upisivanje jednog znaka(prvi argument) u fajl koji je naveden drugim argumentom. Primeticete da je prvi argument tipa INT tako da mozete proslediti i promenljivu tipa CHAR a ne samo INT.

int fgetc(FILE *);
–Iscitavanje jednog znaka iz fajla. Iscitani znak funkcija vraca kao povratnu vrednost.

char * fgets(char *, int, FILE *);
–Poput gets(), funkcija fgets() sluzi za iscitavanje celog reda texta ali ne sa konzole vec iz fajla. Kao prvi argument treba predati string u koji ce biti smesten iscitani text, dok nam je znacenje treceg argumenta poznato. Drugi argument izgleda kao visak medjutim on pretstavlja velicinu najvece ocekivane duzine reda texta. Ako na primer unesete broj 10 a neki red u fajlu ima 20 znakova imacete problem jer nema dovoljno mesta da se svi ti znakovi sacuvaju. Uobicajena vrednost koju treba predati ovoj funkciji je duzina stringa koji ste predali kao prvi argument. Dakle ovako:

```
                    char ret[30];
                      FILE *pf;
          pf = fopen(„c:\\toxi.txt", „r");
                 fgets(ret, 30, pf);
```

Sistem po kom ovo funkcijonise bi trebalo da vam je jasan.

int fputs(const char *, FILE *);
–Ponasa se potpuno isto kao i fges() ali upisuje podatke u fajl. Primeticete da nema argumenta duzine znakovno niza posto je on funkciji poznat.

Nesto naprednije I/O u fajl je binarni I/O medjutim njega necemo uzeti u razmatranje.

Kraj...

Mozda cu ovaj tutrial prosiriti novim poglavljima ali sigurn ne uskoro... Ako van nesto nije jasno ili imate kakve primedbe(ili se ste se nasli zateceni mojom gramatickom nepismenoscu) moj e-mail je toxi@sezampro.com.

May the Source be with you!