FESB Split
Zavod za elektroenergetiku

mr.sc. Ranko Goić, dipl. ing.

Programski jezik C

- bilješke s predavanja, privremeni i nepotpuni materijali -

Split, ožujak 2002

1. UVOD

Slijedeći program je najmanji moguću program u C jeziku:

```
main()
{
}
```

Ne radi ništa.

Slijedeći program ipak nešto radi:

```
#include <stdio.h>

main()
{
      printf("FESB\n");
}
```

Ispis programa
FESB
_

Za pisanje C-programa koriste se mala i velika slova, ali sve naredbe u C-u moraju biti napisane malim slovima. Početak C programa označava se sa:

```
main()
```

Zagrade koje se nalaze nakon ključne riječi main označavaju da program ne prima nikakve argumente (što će biti kasnije objašnjeno).

Dvije zagrade, { i }, označavaju početak i kraj dijelova programa. Svrha izraza

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

je da omogući uporabu **printf** funkcije za ispis. Tekst koji će ispisati **printf()** mora biti u navodnicima. Program ima samo jednu naredbu

```
printf("FESB\n");
```

printf() je zapravo funkcija u C-u koja se koristi za formatirani ispis varijabli i teksta. Tekst unutar navodnika " ", ispisuje se onako kaoko je zadano.
Ipak postoje neke iznimke - označavaju se znakovima: \ i **%** . Ovi znakovi su **modifikatori**. U našem primjeru \ koji prethodi znaku **n** predstavlja oznaku prijelaza u novi red. Tako program ispisuje

FESB

i kursor se postavlja na početak nove linije. Kao što ćemo kasnije vidjeti, znak koji slijedi iza znaka \ određuje što se ispisuje, tj. 8 praznih mjesta (tab), prazan ekran, prazna linija itd.
Sve naredbe u C-u završavaju točka-zarezom **;**

PISANJE, KOMPAJLIRANJE I IZVRŠAVANJE PROGRAMA

- **Pisanje koda**

Kod – niz naredbi koje čine program

Kod se piše u **EDITOR**-u koji generira tzv. ACSII file

Napisanom kodu se mora dati neko ime sa ekstenzijom .C .

- **Kompajliranje koda**

Prevođenje napisanih naredbi u tzv. objektni kod.
Kompajliranje se vrši pomoću posebnog programa – kompajlera (prevoditelja) koji generira objektni kod (datoteka s nastavkom .OBJ)

- **Linkanje koda u izvršni program**

Linkanje znači konačno formiranje izvršnog programa.
Linkanje se vrši pomoću posebnog programa – linkera (povezivač) koji generira izvršni kod (datoteka s nastavkom .EXE).

- **Ispravljanje grešaka**

Napisani program u svom izvornom kodu može imati grešaka koje prijavljuje kompajler ili linker.
Ako kompajler ili linker jave neku grešku, treba se vratiti korak ili dva nazad, ispraviti greške i ponoviti postupak.
Ako kompajler i linker ne jave greške, program se može izvršiti (što ne znači da je i dobar).

- **Izvršavanje programa**

Konačni program koji se dobije nakon što linker izvrši svoj dio posla (ako ne javi neku grešku) izvršava se ukucavanjem njegova naziva.

- **Traženje logičkih grešaka**

Ako program radi ali ne ispravno, treba potražiti logičke greške u programu. To se obično radi s tzv. **debuger** programom koji između ostalog omogućava izvršavanje jedne po jedne naredbe i uvid u vrijednost svih varijabli.

KOMENTARI

Dodavanje komentara programu je poželjno. Komentari se C programu mogu dodati unutar znakova /* */:

```
/* bla bla bla bla bla bla */
```

Primijetite da /* otvara polje komentara, a da ga */ zatvara. Komentari mogu obuhvaćati više linija. Međutim, ne mogu se gnijezditi jedan unutar drugog.

```
/* ovo je komentar. /* ovaj komentar je unutra */ pogrešno */
```

U gornjem primjeru, prva pojava znaka */ zatvara komentar za čitavu liniju, što znači da riječ se pogrešno interpretira kao C izraz ili varijabla, i u ovom primjeru, generira grešku.

Zašto se komentari koriste:
- dokumentiranje varijabli i njihovo korištenje
- objašnjavanje složenih dijelova programa
- opis programa, autor, datum, promjene, revizije itd.

OSNOVNA STRUKTURA C PROGRAMA

C programi u osnovi imaju slijedeću strukturu:

```
/* ZAGLAVLJE                    */
/* Sadrži ime programa, ime autora, broj revizije...*/
```

```
/* INCLUDE dio                  */
/* sadrži  #include izraze      */


/* KONSTANTE I TIPOVI        */
/* sadrži tipove i  #define izraze        */


/* GLOBALNE  VARIJABLE          */
/* ako postoje globalne varijable deklariraju se ovdje    */


/* FUNKCIJE                */
/* funkcije koje definira korisnik          */


/* main()                  */

int main()
{


}
```

Pridržavanje ove dobro definirane strukture učinit će vaše programe lakim za čitanje i modificiranje .

2. VARIJABLE, TIPOVI PODATAKA, KONSTANTE

VARIJABLE

Programski jezik C ima ČETIRI osnovna tipa podataka. Varijable koje definira korisnik moraju biti deklarirane prije nego će se koristiti u programu.

C razlikuje mala i velika slova, tako da iako dvije varijable dolje navedene imaju isto ime, u C-u se smatraju različitim varijablama.

sum , Sum

Deklaracija varijabli vrši se nakon otvorene zagrade main(),

```
#include <stdio.h>

main()
{
      int sum;
      sum = 500 + 15;
      printf("Suma 500 i 15 iznosi %d\n", sum);
}
```

Ispis programa:

Suma 500 i 15 iznosi 515

Osnovni format deklariranja varijabli je

tip podatka var1, var2, ... ;

gdje je **tip podataka** jedan od četiri osnovna tipa podataka: **integer, character, float ili double .**

Program deklarira varijablu sum kao tip INTEGER (int). Varijabli sum se onda pridružuje vrijednost 500 + 15 koristeći **operator pridruživanja**, znak = .

```
sum = 500 + 15;
```

Imena varijabli moraju započinjati slovom ili znakom _, a nakon toga bilo kakvom kombinacijom slova, znaka _, ili znamenki 0 - 9. Slijede primjeri dozvoljenih imena varijabli:

zbroj izlazna_zastavica i Jerry7 Broj_poteza _važeća_zastavica

Osigurajte da koristite imena sa značenjem za vaše varijable. Razlozi za to su:

- imena sa značenjem na prvi pogled pokazuju što varijabla radi
- lakša su za razumijevanje
- nema povezanosti s korištenjem prostora u izvršnom kodu (.EXE file)
- čine program lakšim za čitanje

ISPIS

Pogledajmo sada pobliže **printf()** funkciju. Ima dva argumenta, odvojena zarezom. Pogledajmo prvi argument,

```
"Suma 500 i 15 iznosi %d\n"
```

Znak **%** je posebni znak u C-u. Koristi se za ispis vrijednosti varijable. Kad se program izvršava, C ispisuje tekst dok ne naiđe na znak **%** . Kada naiđe, traži slijedeći argument (u ovom slučaju **sum**), ispisuje njegovu vrijednost, te nastavlja. Znak **d** koji prati **%** označava da se očekuje integer tip varijable. Tako, kada se naiđe na znak **%d printf()** funkcija traži slijedeći argument (u ovom slučaju varijablu sum, koja iznosi 515), i ispisuje ga. Znak **\n** se nakon toga izvršava što znači da se kursor prebacuje u novi red.

Tako ispis programa izgleda:

```
Suma 500 i 15 iznosi  515
```

Slijedeći program ispisuje dvije cjelobrojne vrijednosti razdvojene tabulatorom:

To se postiže korištenjem specijalnog znaka **\t**

```
#include <stdio.h>

main()
{
      int sum, value;
      sum = 10;
      value = 15;
      printf("%d\t%d\n", sum, value);
}
```

Ispis programa

```
10      15
```

Što ispisuje slijedeći program?

```
#include <stdio.h>

main()
{
      int   value1, value2, sum;
      value1 = 35;
      value2 = 18;
```

```
        sum = value1 + value2;
        printf("Suma %d i %d iznosi %d\n", value1, value2, sum);
    }
```

Znakovi koji se nalaze iza \ znaka u **printf()** naredbi, imaju slijedeće značenje:

Modifikator	Značenje
\b	pomak za jedno mjesto u nazad (backspace)
\f	nova stranica (form feed)
\n	novi red (new line)
\r	povratak na početak linije (carriage return)
\t	horizontalni tabulator (horizontal tab)
\v	vertikalni tabulator (vertical tab)
\\	obrnuta kosa crta (backslash)
\"	navodnici (double quote)
\'	literal (single quote)
\<enter>	slijedeća linija (line continuation)
\nnn	nnn = oktalna vrijednost znaka
\0xnn	nn = heksadecimalna vrijednost (samo neki kompajleri)

printf("\007Pažnja, to je bio zvučni signal!\n");

Specifikatori formata za ispis varijabli:

%d	decimalni cjelobrojni (integer)
%c	znak (character)
%s	string ili niz znakova (character array)
%f	realni brojevi s pomičnim zarezom (float)
%e	realni brojevi s pomičnim zarezom dvostruke točnosti (double)

TIPOVI PODATAKA I KONSTANTE

Tip podatka označava način kodiranja i operacije koje je dozvoljeno primijeniti na programske varijable, čija je vrijednost smještena u memoriji računala. Položaj varijable u memoriji naziva se adresa ili memorijska referenca varijable. Četiri osnovna tipa podataka su:

Cjelobrojni decimalni (INTEGER)
U ovaj tip podataka spadaju pozitivni i negativni cijeli brojevi. Također imamo i **unsigned int** (samo pozitivni cijeli brojevi). Postoje i **short int** i **long int** cijeli brojevi.

Ključna riječ za definiranje cijelog broja je:
int

Primjer deklaracije i dodjeljivanja vrijednosti integer varijable sum je,

```
int sum;
sum = 20;
```

Realni brojevi s pomičnim zarezom (FLOATING POINT)
Obuhvaćeni su i pozitivni i negativni realni brojevi. Ključna riječ za definiranje realnog broja je,
float ime_varijable;

Primjer deklaracije i dodjele vrijednosti float varijable **money**:

```
float money;
money = 0.12;
```

Realni brojevi s eksponentom (DOUBLE)

To su brojevi zapisani u obliku realnog dijela x 10 na eksponent, pozitivni i negativni. Ključna riječ koja se koristi za definiranje double varijabli je,

double

Primjer:

```
double big;
big = 31.2E+7;   /* znači big=312000000  */
```

Znak (CHARACTER)

Ključna riječ za deklariranje znaka je

char

Primjer:

```
char letter;
letter = 'A';
```

Primijetite da se pridruživanje znaka **A** varijabli **letter** vrši stavljanjem znaka u jednostruke navodnike. Varijabli se može pridružiti samo jedan znak!

Primjer programa koji pokazuje uporabu svih tipova podataka

```
#include < stdio.h >

main()
{
      int   sum;
      float money;
      char  letter;
      double pi;

      sum = 10;             /* pridruživanje integer vrijednosti */
      money = 2.21;          /* pridruživanje  float vrijednosti */
      letter = 'A';         /* pridruživanje character vrijednosti */
      pi = 2.01E6;          /* pridruživanje double vrijednosti */

      printf("vrijednost sum = %d\n", sum );
      printf("vrijednost money = %f\n", money );
      printf("vrijednost letter = %c\n", letter );
      printf("vrijednost pi = %e\n", pi );
}
```

Ispis programa

```
vrijednost sum = 10
vrijednost money = 2.210000
vrijednost letter = A
vrijednost pi = 2.010000e+06
```

Ispisivanje ASCII vrijednosti znakova

Ako se znak ispisuje pomoću **%d** specifikatora, to će značiti kompajleru da ispiše ASCII vrijednost tog znaka.

```
printf("Znak A ima vrijednost %d\n", letter);
```

Program ispisuje integer vrijednost 65.

Što će biti rezultat slijedećih operacija?

```
int c;
c = 'a' + 1;
printf("%c\n", c);
```

INICIJALIZIRANJE VARIJABLI PRI DEKLARIRANJU

C varijable se mogu inicijalizirati na neku vrijednost kada se deklariraju. Pogledajmo slijedeću deklaraciju koja deklarira integer varijablu **count** koja je inicijalizirana na vrijednost 10.

```
int  count = 10;
```

VRIJEDNOST VARIJABLI PRI DEKLARACIJI

Ispitajmo koja se vrijednost automatski pridjeljuje varijabli kada je deklariramo (default). Da bismo to učinili, razmotrimo slijedeći program, koji deklarira dvije varijable, **count** koja je cjelobrojna, i **letter** koja je znakovna. Nijedna varijabla nije prije inicijalizirana. Vrijednost svake od varijabli se ispisuje korištenjem **printf()** funkcije.

```
#include <stdio.h>

main()
{
      int   count;
      char  letter;
      printf("Count = %d\n", count);
      printf("Letter = %c\n", letter);
}
```

Ispis programa

```
Count = 26494
Letter = f
```

Iz ispisa programa može se vidjeti da su vrijednosti koje se pridružuju varijablama pri deklaraciji različite od nule. U C-u, to je uobičajeno, i programeri moraju osigurati da su varijablama pridružene vrijednosti prije no što ih koriste.
Ako bi se program ponovno startao, ispis bi mogao imati i različite vrijednosti za svaku od varijabli. Nikad ne možemo pretpostaviti da će varijable deklarirane na ovaj način poprimiti neku određenu vrijednost.

MIJENJANJE BROJEVNE BAZE

Brojčani podaci mogu se izraziti u bilo kojoj bazi jednostavnim mijenjanjem modifikatora, npr. decimalno, oktalno, ili heksadecimalno. To se postiže slovom koje prati znak **%** u **printf** funkciji.

```
#include <stdio.h>

main() /* Ispisuje istu vrijednost u decimalnoj, heksadecimalnoj i oktalnoj notaciji */

{
      int   number = 100;

      printf("Kao decimalan broj je zapisan sa %d\n", number);
      printf("Kao heksadecimalan broj je zapisan sa %x\n", number);
      printf("Kao oktalan broj je zapisan sa %o\n", number);
      /* što je s  %X\n kao argumentom?  */
}
```

Ispis programa

Kao decimalan broj je zapisan sa 100
Kao heksadecimalan broj je zapisan sa 64
Kao oktalan broj je zapisan sa 144

DEFINIRANJE VARIJABLI U OKTALNOJ I HEKSADECIMALNOJ NOTACIJI

Često kada piše programe, programeru je potrebno zapisati brojeve u nekoj drugoj notaciji osim decimalne. Cjelobrojne konstante mogu se definirati u oktalnoj ili heksadecimalnoj notaciji korištenjem određenog prefiksa, npr., za definiranje cijelog broja u oktalnoj notaciji
koristi se **%o**

```
int    sum = %o567;
```

Za definiranje cijelog broja u heksadecimalnoj notaciji koristi se **%0x**

```
int    sum  = %0x7ab4;
int    flag = %0x7AB4;     /* mogu se koristiti i velika i mala slova */
```

VIŠE O FLOAT I DOUBLE VARIJABLAMA

C ispisuje i **float** i **double** varijable na šest decimalnih mjesta. To se NE odnosi na preciznost (točnost) na koju je broj zapravo pohranjen, već samo na to koliko decimalnih mjesta **printf()** koristi za prikaz tog tipa varijabli. Slijedeći program ilustrira kako se različiti tipovi podataka deklariraju i ispisuju,

```
#include <stdio.h>

main()
{
      int     sum = 100;
      char    letter = 'Z';
      float   set1 = 23.567;
      double  num2 = 11e+23;

      printf("Integer varijabla je %d\n", sum);
      printf("Znak je %c\n", letter);
      printf("Float varijabla je %f\n", set1);
      printf("Double varijabla je %e\n", num2);
}
```

Ispis programa

Integer varijabla je 100
Znak je Z
Float varijabla je 23.567000
Double varijabla je 11.000000e23

Da bismo promijenili broj decimalnih mjesta na koja se ispisuju **float** ili **double** varijable, mijenjamo **%f** ili **%e** tako da uvrstimo i preciznost, npr,

```
printf("Float varijabla je %.2f\n", set1 );
```

U ovom slučaju, uporaba **%.2f** ograničava ispis na dva decimalna mjesta, i sad on izgleda ovako

Ispis programa

Integer varijabla je 100
Znak je Z
Float varijabla je 23.56
Double varijabla je 11.000000e23

PRIVREMENA KONVERZIJA TIPOVA PODATAKA

Pogledajmo slijedeći program:

```
#include <stdio.h>

main()
{
      int  value1 = 12, value2 = 5;
      float answer = 0;

      answer = value1 / value2;
      printf("Vrijednost %d podijeljena s %d iznosi %f\n",value1,value2,answer );

}
```

Ispis programa

Vrijednost 12 podijeljena s 5 iznosi 2.000000

Iako izgleda da je deklaracija u primjeru točna, nije uvijek pouzdana. **answer** ne sadrži točan decimalni dio (tj. sadrži sve nule), budući da se toj varijabli pridružuje rezultat dijeljenja dvije cjelobrojne varijable, a što je opet cjelobrojna varijabla, tj, u primjeru broj 2, koji pridružen varijabli **answer** postaje 2.0.

Da osiguramo pojavljivanje točnog rezultata, tip podataka **value1** i **value2** trebao bi se konvertirati na **float** tip prije pridruživanja **float** varijabli **answer**:

```
answer = (float)value1 / (float)value2;
```

(float) ispred neke varijable je privremeno "pretvara" u varijablu tipa float. Slično vrijedi i za ostale (privremene) konverzije jednog tipa u drugi.

RAZLIČITI TIPOVI INTEGERA

Normalni integer je ograničen vrijednostima +-32767 (u 16-bitnoj verziji kompajlera gdje **integer** varijabla zauzima 2 bajta, ali u novije vrijeme je češći slučaj da **integer** varijabla zauzima 4 bajta). Dakle, ova vrijednost se razlikuje od računala do računala. U C-u je moguće specificirati da se **integer** pohranjuje u više bajtova umjesto na uobičajeni broj (2 ili 4)e. Ovo povećava raspon brojeva i omogućava pohranu vrlo velikih brojeva. To se radi na slijedeći način,

```
long int big_number = 245032L;
```

Za ispisati **long integer**, koristi se **%l**, npr,

```
printf("Veći broj je  %l\n", big_number );
```

Također imamo i **short integer**, npr,

```
short int   small_value = 114h;
printf("Vrijednost je %h\n", small_value);
```

Unsigned integer (samo pozitivne vrijednosti) također mogu biti definirani.

Primjeri:

```
long            x;  /* int sepretpostavlja*/
unsigned char     ch;
signed int       i;  /* signed je default */
unsigned long int l;  /* int se može staviti, ali nije potrebno */
```

3. NAREDBE PREPROCESORU

Ključna riječ **define** se koristi za više namjena. Prva je definiranje konstanti.

ZAMJENA SIMBOLIČKIH KONSTANTI KORIŠTENJEM #define

Pogledajmo slijedeće primjere:

```
#define TRUE    1    /* Nema točka-zareza!!! */
#define FALSE   0
#define NULL    0
#define AND     &
#define OR      |
#define EQUALS  ==

game_over = TRUE;
while( list_pointer != NULL )
      ..............
```

Naredbe preprocesoru započinju simbolom # , i ne završavaju točka-zarezom. Uobičajeno je naredbe preprocesoru navoditi na početku koda.
Preprocesorske naredbe izvršava kompajler (ili tzv. preprocesor) prije no što je program stvarno preveden. Svi # izrazi se izvrše tako da se simboli (poput **TRUE**) koji se pojavljuju u C programu zamjenjuju njihovim vrijednostima (poput **1**). Kad preprocesor izvrši ovu zamjenu, program se prevodi. Obično se preprocesorske konstante pišu VELIKIM SLOVIMA.

Pogledajmo sada nekoliko primjera korištenja simbolićkih konstanti u našim programima. Slijedeći program definira konstantu imena **POREZNA_STOPA**.

```
#include <stdio.h>

#define POREZNA_STOPA  0.10

main()
{
      float glavnica;
      float porez;

      glavnica = 72.10;
      porez = glavnica * POREZNA_STOPA;
      printf("Porez na %.2f iznosi %.2f\n", glavnica, porez );
}
```

Preprocesor prvo zamjenjuje sve simbolieke konstante prije no što je program preveden, tako nakon preprocesora (i prije prevođenja), program izgleda ovako:

```
#include <stdio.h>

#define POREZNA_STOPA  0.10
```

```
main()
{
      float glavnica;
      float porez;

      glavnica = 72.10;
      porez = glavnica * 0.10;
      printf("Porez na %.2f iznosi %.2f\n", glavnica, porez );
}
```

Uzimajući gornji program kao primjer, pogledajte promjene koje smo sada napravili. Dodali smo izraz koji nastoji promijeniti **POREZNU_STOPU** na novu vrijednost.

```
#include <stdio.h>

#define POREZNA_STOPA  0.10

main()
{
      float glavnica;
      float porez ;

      glavnica = 72.10;
      POREZNA_STOPA = 0.15;
      porez = glavnica * POREZNA_STOPA;
      printf("Porez na %.2f iznosi %.2f\n", glavnica, porez  );
}
```

Ovo je nedozvoljeno. Simboličkoj konstanti se ne može pridružiti nova vrijednost.

Kao što je pokazano, preprocesor izvodi pisanu zamjenu simboličkih konstanti. Izmijenimo malo naš program i unesimo grešku da pojasnimo problem koji se može pojaviti:

```
#include <stdio.h>

#define POREZNA_STOPA  0.10;

main()
{
      float glavnica;
      float porez;

      glavnica = 72.10;
      porez = (glavnica * POREZNA_STOPA)+ 10.02;
      printf("Porez na %.2f iznosi %.2f\n", glavnica, porez );
}
```

U ovom slučaju, greška je u tomu što smo **#define** završili točka-zarezom. Preprocesor izvodi supstituciju i pogreška se javlja u liniji (označava je prevodioc)

```
      porez = (glavnica * 0.10; )+ 10.02;
```

Međutim, vi ne vidite izlaz preprocesora. Ako koristite TURBO C, samo vidite

```
      porez = (glavnica * POREZNA_STOPA) + 10.02;
```

označeno kao grešku, a vama to izgleda u redu (ali nije! nakon zamjene).

Svrha korištenja #define u programima je da se učine lakšima za čitanje i mijenjanje. Koristeći gornje programe kao primjere, koje promjene bi trebalo napraviti ako se POREZNA_STOPA promijenila na 20%?
Očito, odgovor je da treba promijeniti **#define** naredbu koja deklarira simboličku konstantu. Pisali bismo

```
#define POREZNA_STOPA = 0.20
```

Bez korištenja simboličkih konstanti, morali bismo unijeti vrijednost 0.20 u program, a ona se može pojavljivati nekoliko puta (ili nekoliko desetaka puta u većim programima). To bi mijenjanje programa učinilo teškim, zato što bi morali tražiti i zamijeniti svaku pojavu te varijable u programu.

Zaključak o **#define**:

- omogućuje upotrebu simboličkih konstanti u programu
- uobičajeno se simboli pišu velikim slovima
- ne završava se točka-zarezom
- općenito se piše na početku koda
- svaki put kada se simbol pojavi, zamjenjuje se vrijednošću
- čini program lakim za čitanje i mijenjanje

PRIDRUŽIVANJE DATOTEKA PROGRAMU (HEADER FILES)

Header files su datoteke koje sadrže definicije funkcija i varijabli koje se mogu uključiti u bilo koji C program korištenjem preprocesorske direktive **#include**. Standardne datoteke dolaze sa svakim prevodiocem i pokrivaju široko područje, operacije sa stringovima, matematičke operacije, konverziju podataka, ispis i učitavanje varijabli. Da bismo koristili neku od standardnih funkcija, moramo uključiti odgovarajući header file. To se čini na početku koda. Naprimjer, da bismo koristili funkciju **printf()** u programu, linija

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

bi trebala biti na početku programa, jer se definicija **printf()** može naći u datoteci **stdio.h**. Sve datoteke imaju ekstenziju **.h** i obično se nalaze u **/include** subdirektoriju.

```
#include <stdio.h>
#include "mydecls.h"
```

Korištenje zagrada <> informira prevodioca da traži u include direktoriju određenu datoteku. Korištenje nazivnika "" oko imena datoteke informira prevodioca da traži u trenutnom direktoriju određenu datoteku.
Naredbe pretprocesoru koriste se i za druge namjene.

4. ARITMETIČKI I RELACIJSKI OPERATORI

Simboli aritmetičkih operatora su:

Operacija	**Operator**	**Opis**
Množenje	*	sum = sum * 2;
Dijeljenje	/	sum = sum / 2;
Zbrajanje	+	sum = sum + 2;
Oduzimanje	-	sum = sum -2;
Inkrement(uvećanje za 1)	++	++sum;
Dekrement(umanjenje za 1)	--	--sum;
Modul (ostatak) (samo za **int**)	%	sum = sum % 3;

Slijedeći dio programa zbraja varijable **loop** i **count** , ostavljajući rezultat u varijabli **sum**

```
sum = loop + count;
```

PREFIX/POSTFIX INKREMENT/DEKREMENT OPERATORI

PREFIX znači da se prvo obavi operacija inkrementa/dekrementa, a zatim nekakva operacija pridruživanja.
POSTFIX znači da se operacija vrši nakon pridruživanja. Pogledajmo slijedeće izraze

```
++count;      /* PREFIX Inkrement, znači povećaj count za jedan*/
count++;      /* POSTFIX Inkrement, znači povećaj count za jedan */
```

U gornjem primjeru, zato što se vrijednost varijable **count** ne pridružuje nijednoj varijabli, učinak PREFIX/POSTFIX operacija je isti: uvećava vrijednost varijable za 1.

Ispitajmo što se događa kada koristimo operator zajedno s operacijom pridruživanja. Pogledajmo slijedeći program:

```
#include <stdio.h>

main()
{
      int count = 0, loop;
      loop = ++count;  /* isto kao count = count + 1; loop = count;  */
      printf("loop = %d, count = %d\n", loop, count);

      loop = count++;  /* isto kao loop = count;  count = count + 1;  */
      printf("loop = %d, count = %d\n", loop, count);

}
```

Ispis programa

```
loop = 1, count = 1
loop = 1; count = 2
```

Ako operator prethodi (ako je s lijeve strane) varijabli, inkrementiranje se izvodi prvo, tako izraz

```
loop = ++count;
```

ustvari znači prvo uvećaj vrijednost varijable **count** za 1, a onda novu vrijednost varijable count pridruži varijabli **loop**.

Ako operator slijedi (ako je s desne strane) varijabli, operacija pridruživanja se izvodi prva, tako izraz

```
loop = ++count;
```

ustvari znači prvo pridruži varijabli loop vrijednost varijable **count** , a onda uvećaj vrijednost varijable **count** za 1.

Kako ćete pisati?
Tamo gdje se inkrement/dekrement operacija koristi za promjenu vrijednosti varijable, i nije uključena ni u kakvu operaciju pridruživanja, svejedno je:

```
++loop_count;
```

ili

```
loop_count++;
```

U složenijim izrazima treba znati koja se operacija treba izvesti prva.

RELACIJSKI OPERATORI

Omogućavaju usporedbu dviju ili više varijabli.

Operator	Značenje
==	jednako
!=	različito

<	manje od
<=	manje ili jednako
>	veće od
>=	veće ili jednako

5. UNOS PODATAKA S TIPKOVNICE

U C-u postoji funkcija koja omogućava prihvat podataka (unos) s tipkovnice. Slijedeći program ilustrira upotrebu ove funkcije:

```
#include <stdio.h>

main()    /* program s primjerom unosa s tipkovnice */
{
      int  number;
      printf("Unesite broj \n");
      scanf("%d", &number);
      printf("Broj koji ste unijeli je %d\n", number);

}
```

Ispis programa

Unesite broj
23 *(ovo je ukucano)*
Broj koji ste unijeli je 23

Definira se integer varijabla **number** . Zatim se ispisuje obavijest o unosu broja funkcijom:

```
printf("Unesite broj \n:");
```

scanf funkcija, koja prihvaća odgovor, ima dva argumenta. Prvi ("**%d**") specifira koji tip podatka se očekuje (npr. char, int, ili float).
Drugi argument (**&number**) specifira varijablu u koju će se smjestiti odgovor s tipkovnice. U ovom slučaju unesena vrijednost će biti smještena namemorijsku lokaciju pridruženu varijabli **number**.
Ovo objašnjava znak **&**, koji znači adresa od varijable.

Primjer programa koji pokazuje upotrebu **scanf()** za učitavanje integer, character i float varijabli

```
#include < stdio.h >

main()
{
      int sum;
      char letter;
      float money;

      printf("Molimo unesite integer ");
      scanf("%d", &sum );

      printf("Molimo unesite character ");
      /* prazno mjesto ispred znaka %c zanemaruje razmake u unosu */
      scanf("  %c", &letter );

      printf("Molimo unesite float varijablu ");
      scanf("%f", &money );

      printf("\nVarijable koje ste unijeli su\n");
      printf("vrijednost sum = %d\n", sum );
```

```
        printf("vrijednost letter = %c\n", letter );
        printf("vrijednost money = %f\n", money );

}
```

Ispis programa:

```
Molimo unesite integer
34
Molimo unesite character
W
Molimo unesite float varijablu
32.3
Varijable koje ste unijeli su
vrijednost sum = 34
vrijednost  letter = W
vrijednost money = 32.300000
```

Program pokazuje nekoliko važnih točaka:
C jezik ne pruža nikakav oblik provjere unosa korisnika. Od korisnika se očekuje da unese točan tip podatka. Na programeru je da ocijeni podatke za točnost tipa i njihov raspon vrijednosti.

ZNAKOVI FORMATIRANJA scanf() FUNKCIJE

Znakovi formatiranja, iza znaka **%** imaju slijedeće značenje.

Modifikator	Značenje
d	učitaj decimalni integer
o	učitaj oktalnu vrijednost
x	učitaj heksadecimalnu vrijednost
h	učitaj short integer
l	učitaj long integer
f	učitaj float vrijednost
e	učitaj double vrijednost
c	učitaj jedan char (znak)
s	učitaj niz znakova, prestani čitati kada se pritisne tabulator ili space tipka
[...]	Učitava se niz karaktera. Karakteri unutar zagrada pokazuju koji su karakteri dozvoljeni-mogući unutar niza. Ako se unese bilo koji drugi karakter, niz se prekida. Ako je prvi znak a ^, preostali znakovi unutar zagrada pokazuju da njihovim unošenjem prekidamo niz.
*	ovo se koristi da bi se preskočilo polja unosa

Primjer **scanf()** modifikatora

```
int number;
char text1[30], text2[30];
scanf("%s %d %*f %s", text1, &number, text2);
```

Ako je korisnikov odgovor,

```
Hello 14 736.55 uncle sam
```

tada je

```
text1 = hello, number = 14, text2 = uncle
```

a slijedeći poziv **scanf** funkcije će nastaviti gdje se posljedni zaustavio, pa ako je

```
scanf("%s ", text2);
```

bio slijedeći poziv, tada je

text2 = sam

6. IF NAREDBA

If naredba omogućava grananje (donošenje odluka) ovisno o vrijednosti ili stanju varijabli. To omogućava da se naredbe izvrše ili preskoče, ovisno o odluci. Osnovni format je:

```
if( izraz )
    programske naredbe;
```

Primjer;

```
if( studenti < 65 )
    ++student_count;
```

U gornjem primjeru, varijabla **student_count** se inkrementira samo ako je vrijednost varijable **studenti** manja od 65.
Slijedeći program koristi **if** naredbu da ocijeni je li korisnikov upisani broj između 1 i 10 (treba ga proučiti nakon obrade WHILE petlje).

```
#include <stdio.h>

main()
{
    int number;
    int valid = 0;

    while( valid == 0 ) {
        printf("Unesite broj između 1 i 10 -->");
        scanf("%d", &number);
        /* pretpostavka da je broj valjan */
        valid = 1;
        if( number < 1 ) {
            printf("Broj je manji od 1. Unesite ponovno\n");
            valid = 0;
        }
        if( number > 10 ) {
            printf("Broj je veći od  10. Unesite ponovno\n");
            valid = 0;
        }
    }

    printf("Broj je %d\n", number );

}
```

Ispis programa

```
Unesite broj između 1 i 10 --> -78
Broj je manji od 1. Unesite ponovno
Unesite broj između 1 i 10 --> 4
Broj je 4
```

Pogledajmo slijedeći program koji utvrđuje je li znak unesen s tipkovnice u rasponu od A do Z.

```
#include <stdio.h>

main()
```

```
{
      char letter;
      printf("Unesite znak-->");
      scanf(" %c", &letter );

      if( letter >= 'A' )
            if( letter <= 'Z' )
                  printf("Znak je između A i Z\n");
}
```

Ispis programa

```
Unesite znak --> C
Znak je između  A i Z
```

Program ne ispisuje ništa ako znak nije između A i Z.

Primjetite korištenje preznog mjesta u izrazu (ispred **%c**)

```
scanf(" %c", &letter );
```

Ovo omogućava preskakanje TABULATORA, razmaka i ENTER tipke. Ako se ne koristi vodeće prazno mjesto, koristio bi se prvi unešeni znak, i **scanf** ne bi ignorirala navedene znakove.

IF ELSE

Opći format ove naredbe je,

```
if( uvjet 1 )
  izraz1;
else if( uvjet 2 )
  izraz2;
else if( uvjet 3 )
  izraz3;
else
  izraz4;
```

else omogućava da se izvrši naredba kada je uvjet neistinit.

Slijedeći program koristi **if else** naredbu da ocijeni je li upis korisnika između 1 i 10 (također ga treba proučiti nakon obrade WHILE petlje).

```
#include <stdio.h>

main()
{
      int number;
      int valid = 0;

      while( valid == 0 ) {
            printf("Unesite broj između 1 i 10 -->");
            scanf("%d", &number);
            if( number < 1 ) {
                  printf("Broj je manji od 1. Unesite ponovno\n");
                  valid = 0;
            }
            else if( number > 10 ) {
                  printf("Broj je veći od  10. Unesite ponovno\n");
                  valid = 0;
            }
```

```
            else
                valid = 1;
        }
        printf("Broj je %d\n", number );
}
```

Ispis programa
Unesite broj između 1 i 10 --> 12
Broj je veći od 10. Unesite ponovno
Unesite broj između 1 i 10 --> 5
Broj je 5

Ovaj program se malo razlikuje od prethodnog primjera u tome da se **else** koristi za postavljanje varijable **valid** na 1. U ovom programu lakše je pratiti logiku.

Slijedeći program simulira kalkulator. Unose se dva broja i operacija koju treba napraviti. Uvosno o unsesenoj operaciji, računa se rezultat u varijabli **rezultat**.

```
/* Ilustrira ugniježđene if else i više argumenata za scanf funkciju.  */

#include <stdio.h>

main()
{
    int    invalid_operator = 0;
    char   operator;
    float  number1, number2, result;

    printf("Unesite dva broja i operator\n");
    printf(" number1 operator number2\n");
    scanf("%f %c %f", &number1, &operator, &number2);

    if(operator == '*')
        result = number1 * number2;
    else if(operator == '/')
        result = number1 / number2;
    else if(operator == '+')
        result = number1 + number2;
    else if(operator == '-')
        result = number1 - number2;
    else
        invalid_operator = 1;

    if( invalid_operator != 1 )
        printf("%f %c %f iznosi %f\n", number1, operator, number2, result );
    else
        printf("Nedozvoljeni operator.\n");

}
```

Ispis programa

Unesite dva broja i operator
number1 operator number2

23.2 + 12
23.2 + 12 iznosi 35.2

UVJETNI OPERATOR

Uvjetni izraz sastoji se iz TRI dijela: **uvjet**, **izraz1** i **izraz2**. Dva simbola koja se koriste da bi se označio ovaj operator su upitnik (**?**) i dvotočka (**:**). Prvi dio se postavlja prije **?**, drugi između **?** i **:**, i treći nakon **:**. Opći format je:

```
uvjet ? izraz1 : izraz2;
```

Ako je rezultat uvjeta TRUE (različit od nule), **izraz1** se računa i rezultat računanja postaje rezultat operacije. Ako je uvjet FALSE (nula), onda se **izraz2** računa i njegov rezultat postaje rezultatom operacije. Pojasnit ćemo primjerom:

```
s = ( x < 0 ) ? -1 : x * x;
```

Ako je x manji od nule onda je s = -1
Ako je x veći od nule onda je s = x * x

Primjer programa koji ilustrira uporabu uvjetnog izraza :

```
#include <stdio.h>

main()
{
   int input;

   printf("Reći ću vam je li broj pozitivan, negativan ili nula!"\n");
   printf("molim sad unesite vaš broj--->");
   scanf("%d", &input );
   (input < 0) ? printf("negativan\n") : ((input > 0) ? printf("pozitivan\n"): :printf("nula\n"));
}
```

Ispis programa

Reći ću vam je li broj pozitivan, negativan ili nula!
molim sad unesite vaš broj---> 32
pozitivan

7. FOR I WHILE PETLJA

FOR

Osnovni format **for** petlje je:

```
for( početni uvjet; uvjet nastavka; ponovna procjena )
      naredbe kontrolirane petljom;
```

Primjer za ilustraciju:

```
/* primjer programa s  for petljom */
#include <stdio.h>

main()   /* Program uvodi for petlju, broji do deset */
{
      int  count;

      for( count = 1; count <= 10; count = count + 1 )
            printf("%d ", count );

      printf("\n");
}
```

Ispis programa

1 2 3 4 5 6 7 8 9 10

Program deklarira integer varijablu **count**. Prvi dio izraza

for(count = 1;

inicijalizira vrijednost **count** na 1. **For** petlja se nastavlja dok je zadovoljen uvjet

count <= 10;

tj. dok je on istinit. Kako je varijabla **count** upravo inicijalizirana na 1, uvjet je ispunjen pa se i naredba

printf("%d ", count);

vrši, ispisuje se vrijednost **count** na ekran, praćena razmakom.

Zatim se vrši slijedeća naredba u for petlji

count = count + 1);

što dodaje jedan trenutnoj vrijednosti **count**. Vraćamo se opet na uvjet,

count <= 10;

koji je opet istinit pa se vrši naredba

printf("%d ", count);

Count se opet inkrementira, uvjet provjerava itd. sve dok se ne dođe do vrijednosti 11.

Kada se to dogodi, uvjet

count <= 10;

postaje netočan, **for** petlja se završava i program prelazi na naredbu

printf("\n");

koja ispisuje novu liniju, i program završava jer više nema naredbi za izvršavanje.

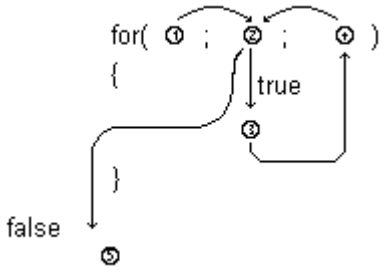
Drugi primjer:

```
/* primjer programa s  for petljom */
#include <stdio.h>

main()
{
      int  n, t_number;

      t_number = 0;
      for( n = 1; n <= 200; n = n + 1 )
            t_number = t_number + n;

      printf("Suma brojeva od 1 do 200 iznosi %d\n", t_number);
}
```

Ispis programa

Suma brojeva od 1 do 200 iznosi 20100

Gornji program koristi for petlju za računanje sume brojeva od 1 do 200 .
Slijedeći dijagram pokazuje kako se izvršavaju dijelovi for petlje.



Primjer korištenja for petlje za ispis znakova :

```
#include <stdio.h>

main()
{
      char letter;
      for( letter = 'A'; letter <= 'E'; letter = letter + 1 )
            printf("%c ", letter);
}
```

Ispis programa
A B C D E

Primjer korištenja for petlje za sumu brojeva, korištenjem dvije inicijalizacije :

```
#include <stdio.h>

main()
{
      int total, loop;
      for( total = 0, loop = 1; loop <= 10; loop = loop + 1 )
            total = total + loop;
      printf("Total = %d\n", total );
}
```

Ispis programa

Total = 55

U gornjem primjeru, varijabla **total** se inicijalizira na 0 u prvom dijelu **fo**r petlje. Dva izraza,

```
for( total = 0, loop = 1;
```

su dio inicijalizacije. Ovo pokazuje da je dozvoljeno više izraza, ali moraju biti odvojeni zarezom.

WHILE PETLJA

Petlja **while** omogućava ponavljanje C naredbi dok je neki uvjet istinit. Njen format je,

```
while( uvjet )
    programske naredbe;
```

Negdje u tijelu **while** petlje mora postojati izraz koji mijenja vrijednost uvjeta tako da petlja može završiti.

```
/* Primjer programa s petljom while  */

#include <stdio.h>

main()
{
    int  loop = 0;

    while( loop <= 10 )
       {
          printf("%d\n", loop);
          ++loop;
       }
}
```

Ispis programa

```
0
1
...

10
```

Gornji program koristi while petlju za ponavljanje ***DVIJE*** naredbe :

```
printf("%d\n", loop);
++loop;
```

dok je vrijednost varijable **loop** manja ili jednaka 10.

Ove dvije naredbe napisane su unutar vitičastih zagrada, koje ih u stvari grupiraju u jednu cjelinu. Općenito se vitičaste zagrade koriste za grupiranje većeg broja naredbi koje se izvršavaju kao jedna cjelina, najčešće u petljama ili u naredbama odlučivanja (IF, SWITCH).

Iako nije bitno na kojem mjestu će se otvoriti i zatvoriti vitičasta zagrada (bitno je da to bude ispred i iza naredbi koje se grupiraju), obično se ona otvara neposredno nakon prethodne naredbe (petlje ili naredbe odlučivanja), a zatvara u redu ispod zadnje grupirane naredbe, tj. u primjeru:

```
while( loop <= 10 ) {
    printf("%d\n", loop);
    ++loop;
  }
```

Zapamtite kako je varijabla o kojoj je petlja **while** ovisna inicijalizirana prije same **while** petlje (u ovom slučaju u prethodnoj liniji), i da se vrijednost varijable mijenja unutar petlje, tako da će jednom uvjet postati neistinit i **while** petlja će završiti.
Ovaj program je ustvari ekvivalentan prethodnom **for** programu koji je brojao do deset.

DO WHILE PETLJA

do { } while petlja omogućava da se naredbe izvršavaju dok je uvjet istinit. Petlja se izvršava barem jednom.

```
/* Demonstracija petlje DO...WHILE    */

#include <stdio.h>

main()
{
      int  value, r_digit;

      printf("Unesite broj koji će se ispisati od kraja.\n");
      scanf("%d", &value);
      do {
            r_digit = value % 10;
            printf("%d", r_digit);
            value = value / 10;
      } while( value != 0 );

      printf("\n");

}
```

Gornji program broj koji unese korisnik ispisuje od kraja. To čini koristeći ostatak dijeljenja (**%** operator) da bi krajnje desnu znamenku upisao u varijablu **r_digit**. Početni broj se zatim dijeli sa 10, i operacija se ponavlja dok broj ne postane jednak 0.

Pogledajmo slijedeći dio programa:

```
      do {
            r_digit = value % 10;
            printf("%d", r_digit);
            value = value / 10;
      } while( value != 0 );
```

NEMA nikakvog izbora hoće li se petlja izvršavati ili ne na samom početku. Ulazak u petlju je automatski, jer je jedini izbor nastavak.
DO WHILE petlja uvijek se bar jednom izvrši, tj. nije osigurana početna kontrola kao u WHILE petlji. To znači da je moguće ući u **do { } while** petlju s nevaljanim podacima.

Gornji program napisan upotrebom **while** petlje:

```
/* ponovljeni program bez do-while */

#include <stdio.h>

main()
{
      int  value, r_digit;

      value = 0;
      while( value <= 0 ) {
            printf("Unesite broj koji će se ispisati od kraja.\n");
            scanf("%d", &value);
            if( value <= 0 )
                  printf("Broj mora biti pozitivan\n");
      }

      while( value != 0 )
      {
            r_digit = value % 10;
            printf("%d", r_digit);
            value = value / 10;
```

```
        }
        printf("\n");

}
```

Ispis programa

```
Unesite broj koji će se ispisati od kraja.
-43
Broj mora biti pozitivan
Unesite broj koji će se ispisati od kraja.
423
324
```

LOGIČKI OPERATORI U SLOŽENIM RELACIJAMA (AND, NOT, OR, EOR)

Logički operatori omogućavaju uspoređivanje više od jednog uvjeta. Ti uvjeti su dio nekog izraza kojeg testiramo. Simboli ovih operatora su:

LOGIČKO I (AND) &&

Logičko I zahtijeva da svi uvjeti budu istiniti .

LOGIČKO ILI (OR) ||

Logičko ILI će se izvršiti ako je bar jedan od uvjeta istinit.

LOGIČKA NEGACIJA (NOT) !

Logička negacija negira uvjet (mijenja ga iz istinitog u neistinitog i obrnuto).

LOGIČKO EKSKLUZIVNO ILI (EOR) ^

Logičko ekskluzivno ILI će se izvršiti ako je jedan od uvjeta istinit, ali NE ako su svi istiniti.

Slijedeći program koristi **if** naredbu s logičkim I - AND da ocijeni je li korisnikov upis u rasponu između 1 i 10.

```
#include <stdio.h>

main()
{
        int number;
        int valid = 0;

        while( valid == 0 ) {
                printf("Unesite broj između 1 i 10 -->");
                scanf("%d", &number);
                if( (number < 1 ) || (number > 10) ){
                        printf("Broj nije između 1 i 10. Unesite ponovno\n");
                        valid = 0;
                }
                else
                        valid = 1;
        }
        printf("Broj je %d\n", number );
}
```

Ispis programa

```
Unesite broj između 1 i 10 --> 56
Broj nije između 1 i 10. Unesite ponovno
Unesite broj između 1 i 10 --> 6
Broj je 6
```

Program se malo razlikuje od prethodnog primjera u tome što LOGIČKO I eliminira jedan od **else** izraza.

Drugi primjer - provjera raspona unijetog broja:

```
#include <stdio.h>

main()
{
    int number;
    int valid = 0;

    while( valid == 0 ) {
        printf("Unesite broj između 1 i 100");
        scanf("%d", &number );
        if( (number < 1) || (number > 100) )
            printf("Broj je izvan zadanog raspona\n");
        else
            valid = 1;
    }
    printf("Broj je %d\n", number );
}
```

Ispis programa

```
Unesite broj između 1 i 100

203
Broj je izvan zadanog raspona
Unesite broj između 1 i 100
-2
Broj je izvan zadanog raspona
Unesite broj između 1 i 100
37
Broj je 37
```

Program koristi varijablu **valid** kao indikaciju koja označava je li uneseni podatak unutar dozvoljenog raspona vrijednosti. **While** petlja se vrši sve dok je **valid** jednak 0.

Izraz

```
if( (number < 1) || (number > 100) )
```

provjerava je li korisnikov upis unutar granica zadanog raspona, a ako jest, postavlja valid na 1, omogućavajući izlaz iz petlje.

Sada pogledajmo program koji provjerava je li znak kojeg upisuje korisnik između A-Z, drugim riječima alfabetski:

```
#include <stdio.h>

main()
{
    char ch;
    int valid = 0;

    while( valid == 0 ) {
```

```
            printf("Unesite znak A-Z");
            scanf(" %c", &ch );
            if( (ch >= 'A') && (ch <= 'Z') )
                valid = 1;
            else
                printf("Znak je izvan dozvoljenog raspona\n");
        }
        printf("Znak je %c\n", ch );
}
```

```
Ispis programa
Unesite znak  A-Z
a
Znak je izvan dozvoljenog raspona
Unesite znak A-Z
0
Znak je izvan dozvoljenog raspona
Unesite znak A-Z
R
Znak je  R
```

U ovom slučaju, logičko I koristimo jer želimo ocijeniti je li znak unutar raspona, tj. sve vrijednosti između donje i gornje granice. U prethodnom slučaju, koristili smo logičko ILI jer smo željeli provjeriti je li unos iznad gornje granice ili ispod donje granice raspona.

8. SWITCH() NAREDBA

Switch naredba je bolji način pisanja programa kada se pojavljuje više naredbi **if else** koje ispituju vrijednost nekog izraza s očekivanim vrijednostima. Opći format je:

```
switch ( izraz ) {
    case  vrijednost 1:
        programske naredbe;
        ......
        break;
    case  vrijednost 2:
        programske naredbe;
        ......
        break;
    case  vrijednost n:
        programske naredbe;
        .......
        break;
    default:
        .......
        .......
        break;
}
```

Switch naredba uspoređuje vrijernost izraza u zagradama **(izraz)**, sa vrijednostima navedenim iz **case**. Izvršava se samo ona skupina naredbi za koju je ovo zadovoljeno.
Default označava alternativu koja će biti izabrana ako nijedna od prethodnih ne odgovara vrijednosti izraza. Default može biti i ispušten. Desna zagrada na kraju označava kraj izbora alternativa.

Pravila za switch naredbu

- vrijednosti **'case'** moraju biti integer ili character konstante

- poredak **'case'** izraza nije važan
- **default** se može pojaviti kao prva alternativa (uobičajeno je posljednja)
- ne mogu se koristiti rasponi ili izrazi kao vrijednosti **'case'**

```
#include <stdio.h>

main()
{
      int menu, numb1, numb2, total;

      printf("unesite dva broja -->");
      scanf("%d %d", &numb1, &numb2 );
      printf("unesite izbor\n");
      printf("1=zbrajanje\n");
      printf("2=oduzimanje\n");
      scanf("%d", &menu );
      switch( menu ) {
            case 1: total = numb1 + numb2; break;
            case 2: total = numb1 - numb2; break;
            default: printf("Nedozvoljen izbor\n");
      }
      if( menu == 1 )
            printf("%d plus %d je %d\n", numb1, numb2, total );
      else if( menu == 2 )
            printf("%d minus %d je%d\n", numb1, numb2, total );
}
```

Ispis programa
unesite dva broja --> 37 23
unesite izbor
1=zbrajanje
2=oduzimanje
2
37 minus 23 je 14

Gornji program koristi **switch** da izabere između korisnikovog upisa simulirajući jednostavan izbor.

9. NIZOVI

Nizovi su složeni tip podataka koji se sastoji od više podataka istog tipa. Razmotrimo situaciju kada programer mora voditi evidenciju određenog broja ljudi unutar neke organizacije. Sve do sada, naš početni pokušaj bi bio stvoriti posebnu varijablu za svakog korisnika. To bi moglo ovako izgledati:

```
int name1 = 101;
int name2 = 232;
int name3 = 231;
```

Naravno da postaje sve teže voditi evidenciju kako raste broj varijabli. Nizovi nude rješenje ovog problema.
Niz je poput kutije s više elemenata, slično arhivu, koji koristi princip indeksiranja da bi pronašao svaku varijablu koja je u njemu pohranjena. U C-u, indeksiranje počinje od nule.
Nizovi, poput drugih varijabli u C-u, moraju biti deklarirani prije nego što se mogu koristiti.
Zamjena gornjeg primjera uz pomoć nizova izgleda ovako:

```
int names[3];

names[0] = 101;
names[1] = 232;
names[2] = 231;
```

Stvorili smo niz imena **names**, koji može sadržavati tri **integer** varijable.
Nizovi koriste uglate zagrade da bi pristupili svakoj pohranjenoj vrijednosti (zovemo je element).

x[i]

Tako se x[5] odnosi na šesti element u nizu imena x. U C-u, indeksi niza počinju od 0. Pridjeljivanje vrijednosti elementima niza se radi kao i za sve druge varijable

x[10] = g;

U slijedećem primjeru, deklariramo niz znakova **word** , i svakom elementu pridružujemo znak. Zadnji element niza je nula, da bi označili kraj stringa (u C-u, string nije tip podatka, tako se za rad sa stringovima koriste nizovi znakova). **Printf** funkcija se koristi za ispis elemenata niza.

```
/*  Predstavljamo nizove, 2  */

#include <stdio.h>

main()
{
      char word[20];

      word[0] = 'H';
      word[1] = 'e';
      word[2] = 'l';
      word[3] = 'l';
      word[4] = 'o';
      word[5] = \0;

      printf("Sadržaj word[] je -->%s\n", word );

}
```

Ispis programa

Sadržaj word[] je Hello

DEKLARIRANJE NIZOVA

Nizovi mogu sadržavati bilo koji dozvoljeni tip podataka. Nizovi se deklariraju zajedno sa svim drugim varijablama u dijelu programa za deklaraciju:

```
/*  Predstavljamo nizove  */

#include <stdio.h>

main()
{
      int   numbers[100];
      float  averages[20];

      numbers[2] = 10;
      --numbers[2];
      printf("Treći element niza je %d\n", numbers[2]);
}
```

Ispis programa

Treći element niza je 9

Gornji program deklarira dva niza, pridružuje 10 vrijednosti trećeg elementa niza **numbers**, dekrementira tu vrijednost **(--numbers[2]**), i naposljetku je ispisuje. Broj elemenata koji svaki od nizova može imati je upisana između uglatih zagrada.

DODJELJIVANJE POČETNIH VRIJEDNOSTI NIZOVIMA

Prije deklaracije pišemo riječ **static**. Početne vrijednosti se pišu u vitičastim zagradama:

```
#include <stdio.h>

main()
{
      int x;
      static int  values[] = { 1,2,3,4,5,6,7,8,9 };
      static char  word[] = { 'H','e','l','l','o' };
      for( x = 0; x < 9; ++x )
            printf("Values [%d] je %d\n", x, values[x]);
}
```

Ispis programa

```
Values[0] je 1
Values[1] je 2
....
Values[8] je 9
```

Prethodni program deklarira dva niza, **values** i **word**. Primijetite da između uglatih zagrada nema varijable koja označava koliko velik će niz biti.
U ovom slučaju, C inicijalizira niz na broj elemenata koji se pojavljuje unutar vitičastih zagrada. Tako se **values** sastoji od 9 elemenata (indeksiranih od 0 do 8), a niz znakova **word** ima 5 elemenata.

Slijedeći program pokazuje kako inicijalizirati sve elemente cjelobrojnog niza na vrijednost koja odgovara indeksu, koristeći **for** petlju da bi pristupili svakom elementu.

```
#include <stdio.h>

main()
{
      int count;
      int  values[100];
      for( count = 0; count < 100; count++ )
            values[count] = count;
}
```

VIŠEDIMENZIONALNI NIZOVI

Višedimenzionalni nizovi imaju dvije ili više indeksnih vrijednosti koje određuju elemente niza.

```
multi[i][j]
```

U gornjem primjeru, prva indeksna vrijednost **i** određuje indeks retka, dok **j** određuje indeks stupca.

Deklaracija i računanje

```
int       m1[10][10];
static int m2[2][2][2] = { {0,1}, {2,3} };

sum = m1[i][j] + m2[k][l];
```

Način na koji se pridjeljuju početne vrijednosti dvodimenzionalnom nizu **m2**: unutar zagrada nalaze se,

{ 0, 1 },
{ 2, 3 }
što odgovara "retcima" niza.

Nizovi se obično dijele na retke i stupce.. Promatrajući početne vrijednosti pridružene **m2**, vidimo da su one:

m2[0][0] = 0
m2[0][1] = 1
m2[1][0] = 2
m2[1][1] = 3

NIZOVI ZNAKOVA [STRINGOVI]

Pogledajmo slijedeći program:

```
#include <stdio.h>

main()
{
      static char name1[] = {'H','e','l','l','o'};
      static char name2[] = "Hello";
      printf("%s\n", name1);
      printf("%s\n", name2);
}
```

Ispis programa

```
Helloxghifghjkloqw30-=kl``
Hello
```

Razlika između ova dva niza je u tome što **name2** ima nulu postavljenu na kraj niza, tj. u **name2[5]**, dok **name1** nema. Nula na kraju znakovnog niza označava kraj niza. Ovo često može rezultirati u znakovima viška koji se ispisuju na kraju. Da postavimo nulu na kraj niza **name1** , inicijalizaciju možemo promijeniti na:

```
static char name1[] = {'H','e','l','l','o','\0'};
```

Pogledajmo slijedeći program koji inicijalizira niz znakova **word** tijekom programa, koristeći funkciju **strcpy**, koja zahtijeva korištenje include datoteke **string.h** , a koja u zadani niz znakova (prvi argument) kopira zadanu vrijednost (drugi argument).

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

main()
{
      char word[20];
      strcpy( word, "hi there." );
      printf("%s\n", word );
}
```

Ispis programa

```
hi there.
```

Nizovi znakova se često u C-u nazivaju **stringovi**. C ne podržava tip **string**, tako da se nizovi znakova koriste umjesto **stringa**. Modifikator **%s** u funkcijama **printf()** i **scanf()** se koristi za ispis i unos niza znakova. Ovo pretpostavlja da je nula pohranjena kao zadnji element niza. Promotrimo slijedeće izraze koji učitavaju string znakova (uključujući razmake) s tipkovnice.

```
char string[18];
scanf("%s", string);
```

Znak **&** ne piše se ispred imena varijable kada se koristi modifikator **%s** !
Ako bi korisnikov upis bio

Hello<enter>

tada je

```
string[0] = 'H'
string[1] = 'e'
....
string[4] = 'o'
string[5] = '\0'
```

Tipku enter **scanf()** ne uzima kao dio stringa.
String završen nulom ('\0') automatski nakon posljednjeg znaka pohranjenog u nizu.

TRAJANA KONVERZIJA TIPOVA PODATAKA

Slijedeće funkcije konvertiraju tipove podataka.

atof() konvertira ASCII niz karaktera u float

atoi() konvertira ASCII niz karaktera u integer

itoa() konvertira integer u niz karaktera

Primjer

```
/* konvertiraj string u integer */
#include <stdio.h>
#include <stdlib.h>
char string[] = "1234";

main()
{
      int sum;
      sum = atoi( string );
      printf("Sum = %d\n", sum );
}
```

```
/* konvertiraj integer u string */

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

main()
{
      int sum;
      char buff[20];
      printf("Unesi integer ");
      scanf(" %d", &sum );
```

```
        printf( "Kao string to je %s\n", itoa( sum, buff, 10 ) );
}
```

Primijetite da **itoa()** prima tri parametra,

- integer koji se konvertira
- char buffer u koji se pohranjuje string
- vrijednost baze (10=decimal,16=hexadecimal)

Kao što je pokazano **itoa()** vraća pokazivač na rezultirajući string.

10. FUNKCIJE

Funkciju u C-u može programer napisati sam, a može se koristiti biblioteke već postojećih funkcija.
Sam C program je jedna funkcija, označena ključnom riječju ***main***, i zatvorena između vitičastih zagrada. Poziva je operacijski sustav kada se program učitava, a kada je završena, vraća se operacijskom sustavu. Također, već smo koristili dvije funkcije: **printf** i **scanf** za ispis i unos podataka. Osnovna struktura svake funkcije (kad se definira nova funkcija) je :

```
tip_povratnog_podatka  ime_funkcije ( argumenti, argumenti )
   deklaracija_tipa_podataka_argumenata;

   {
      tijelo_funkcije
   }
```

Vrijedno je napomenuti da se tip_povratnog_podatka pretpostavlja da je **int** ako nije ništa navedeno, tako da programi koje smo imali do sada impliciraju da **main()** vraća **integer** operacijskom sustavu.

ANSI C se malo razlikuje u tomu kako se funkcije deklariraju. Njihov format je

```
tip_povratnog_podatka  ime_funkcije (tip_podatka ime_varijable, tip_podatka ime_varijable, .. )

{
      tijelo_funkcije
}
```

Ovo omogućava provjeru tipa korištenjem funkcijskih prototipova da bi se kompajler obavijestio o tipu i broju parametara koje funkcija prima. Kada se funkcija poziva, ove informacije se koriste da se obavi provjera tipa i parametara. Ovaj format ćemo koristiti dalje u primjerima.

ANSI C također zahtijeva da se *tip_povratnog_podatka* za funkciju koja ništa ne vraća, tj. ne daje nikakav izlazni rezultat, označava sa **void**. Default *tip_povratnog_podatka* se pretpostavlja **integer** ako nije naveden, ali se mora podudarati s tipom koji specificira deklaracija funkcije.

Jednostavna funkcija koja nema ni ulaznog ni izlaznog argumenta je:

```
void print_message( void )
{
      printf("Ovo je modul imena print_message.\n");
}
```

Primijetite da je ime funkcije *print_message*. Funkcija ne prima nikakve argumente što se označava ključnom riječju **void** u dijelu za parametre funkcijske deklaracije. *Tip_povratnog_podatka* je također **void**, tako da funkcija ne vraća nikakve podatke.

ANSI C funkcijski prototip za **print_message()** je,

```
void print_message( void );
```

Prototipovi funkcija se navode na početku koda. Često mogu biti smješteni u korisničkim datotekama **.h** (header) .

Uključimo sada ovu funkciju u program.

```
/* Program koji ilustrira jednostavni funkcijski poziv */

#include <stdio.h>

void print_message( void );    /* ANSI C prototip funkcije */

main()
{
      print_message();
}

void print_message( void )     /* kod funkcije */
{
      printf("Ovo je modul imena print_message.\n");
}
```

Rezultat izvršenja programa:

```
Ovo je modul imena print_message.
```

Da bismo pozvali funkciju, dovoljno je napisati njeno ime. Kod povezan s imenom funkcije se tada izvršava. Kada funkcija završi izvršavanje se nastavlja sa izrazom koji je prvi nakon poziva funkcije.
U gornjem programu, izvršavanje počinje sa **main()**. Jedini izraz unutar glavnog tijela programa je poziv kodu funkcije **print_message()**. Ovaj kod se izvršava, i kada je završen vraćamo se u **main()**.
Kako program nema više naredbi, završava vraćanjem u operacijski sustav.

U slijedećem primjeru, funkcija prima jednu varijablu, ali ne vraća nikakvu informaciju. Funkcija računa faktorijel od zadanog argumenta.

```
/* Program za računanje faktorijela  */

#include <stdio.h>

void calc_factorial( int );    /* ANSI prototip */

main()
{
      int  number = 0;

      printf("Unesite broj\n");
      scanf("%d", &number );
      calc_factorial( number );
}

void calc_factorial( int n )
{
      int  i, factorial_number = 1;

      for( i = 1; i <= n; ++i )
            factorial_number *= i;
      printf("Faktorijela od %d je %d\n", n, factorial_number );
}
```

Ispis programa

Unesite broj
3
Faktorijela od 3 je 6

Pogledajmo funkciju **calc_factorial()**. Deklaracija funkcije

void calc_factorial(int n)

označava da nema povratne vrijednosti i da funkcija prima jedan **integer**, unutar tijela funkcije označen imenom **n**. Zatim dolazi deklaracija lokalnih varijabli,

int i, factorial_number = 0;

Program radi prihvaćajući varijablu s tipkovnice i predajući je funkciji. Drugim riječima, varijabla **number** unutar **main** tijela se kopira u varijablu **n** u funkciji, koja tada računa rješenje.

VRAĆANJE REZULTATA FUNKCIJE

To se postiže ključnom riječju **return** koju prati varijabla ili konstantnu vrijednost čiji tip podatka se mora podudarati sa deklariranim *tipom_povratnog_podatka* za funkciju.
U slijedećem primjeru funkcija ima dva ulazna cjelobrojan argumenta, a kao rezultat vraća njihov zbroj:

```
float add_numbers( float n1, float n2 )

{
    return n1 + n2;    /* dozvoljeno*/
}
```

Moguće je da funkcija ima više izraza **return**. Vrijednost koju vraća funkcija (rezultat) mora biti deklarirana u funkciji.

```
int validate_input( char command )
{
    switch( command ) {
        case '+' :
        case '-' : return 1;
        case '*' :
        case '/' : return 2;
        default  : return 0;
    }
}
```

Evo još jednog primjera :

```
#include <stdio.h>

int calc_result( int, int );          /* ANSI prototip */

main()
{
    int  digit1 = 10, digit2 = 30, answer = 0;
    answer = calc_result( digit1, digit2 );
    printf("%d pomnoženo sa %d je %d\n", digit1, digit2, answer );
}

int calc_result( int numb1, int numb2 )
{
```

```
    int result;
    result = numb1 * numb2;
    return result;
}
```

Ispis programa

10 pomnoženo sa 30 je 300

LOKALNE I GLOBALNE VARIJABLE , AUTOMATSKE I STATIČKE VARIJABLE

Lokalne varijable
Ove varijable postoje samo unutar određene funkcije koja ih kreira. Nepoznate su drugim funkcijama i glavnom programu. Lokalne varijable prestaju postojati kada se izvrši funkcija koja ih je kreirala. Ponovno se kreiraju svaki put kada se funkcija poziva ili izvršava.

Globalne varijable
Ovim varijablama može pristupiti svaka funkcija iz programa. Ne kreiraju se ponovno ako se funkcija ponovno poziva.

Definiranje globalnih varijabli na primjeru:

```
/* Demonstriranje globalnih varijabli  */

#include <stdio.h>

int add_numbers( void );              /* ANSI prototip funkcije */

int  value1, value2, value3;    /* Ovo su globalne varijable i može im pristupiti svaka funkcija */

main()
{
    int result;
    value1 = 10;
    value2 = 20;
    value3 = 30;
    result = add_numbers();
    printf("Suma %d + %d + %d iznosi %d\n",
        value1, value2, value3, final_result);
}

int add_numbers( void )
{
    int result;
    result = value1 + value2 + value3;  /* koristi globalne varijable*/
    return result;
}
```

Ispis programa

Suma10 + 20 + 30 iznosi 60

Vidljivost globalnih varijabli se može ograničiti pažljivim odabirom mjesta deklaracije. Globalne varijable su vidljive od deklaracije do kraja koda.

```
#include <stdio.h>

void no_access( void ); /* ANSI prototip funkcije */
```

```
void all_access( void );

static int n2;          /* n2 se vidi od ove točke nadalje */

void no_access( void )
{
      n1 = 10;       /* nedozvoljeno, n1 još nije poznat */
      n2 = 5;        /* dozvoljeno */
}

static int n1;          /* n1 se vidi od ove točke nadalje */

void all_access( void )
{
      n1 = 10;       /* dozvoljeno */
      n2 = 3;        /* dozvoljeno */
}
```

C programi imaju više područja (dijelova) gdje su smješteni podaci. Ti segmenti su tipično,

_DATA Statički podaci

_BSS Neinicijalizirani statički podaci, postavljeni na nulu prije poziva main()

_STACK Automatski podaci, smješteni na programskom stogu, lokalne varijable

_CONST konstantni podaci, korištenjem ANSI C ključne riječi const

Korištenje prikladne ključne riječi omogućava pravilno smještanje varijable u željeni segment podataka.
Statičke varijable se stvaraju i inicijaliziraju jednom, pri prvom pozivu funkcije. Slijedeći pozivi funkcije ne stvaraju ih i ne inicijaliziraju ih ponovno. Kada funkcija završava, varijable još uvijek postoje u _DATA segmentu, ali im ne mogu pristupiti vanjske funkcije.
Automatske varijable su suprotne. Stvaraju se i inicijaliziraju ponovno svaki put kada se funkcija pozove. Nestaju (dealociraju se) kada funkcija završava. Kreiraju se u _STACK segmentu.
Statička varijabla deklarira se ključnom riječi **static**, a automatska **auto**. Ako nije ništa navedeno, podrazumijeva se da je varijabla automatska.

Primjer programa koji pokazuje razliku između statičkih i automatskih varijabli :

```
#include <stdio.h>

void demo( void );          /* ANSI prototip funkcije */

void demo( void )
{
      auto int avar = 0;
      static int svar = 0;

      printf("auto = %d, static = %d\n", avar, svar);
      ++avar;
      ++svar;
}

main()
{
      int i;

      while( i < 3 ) {
            demo();
```

```
            i++;
        }
}
```

Ispis programa

```
auto = 0, static = 0
auto = 0, static = 1
auto = 0, static = 2
```

NIZ KAO ARGUMENT FUNKCIJE

Slijedeći program pokazuje kako funkcija prima niz kao ulazni argument:

```
#include <stdio.h>

int maximum( int [] );        /* ANSI prototip funkcije */

int  maximum( int values[5] )
{
      int  max_value, i;
      max_value = values[0];
      for( i = 0; i < 5; ++i )
            if( values[i] > max_value )
                  max_value = values[i];
      return max_value;
}

main()
{
      int values[5], i, max;
      printf("Unesite 5 brojeva\n");
      for( i = 0; i < 5; ++i )
            scanf("%d", &values[i] );
      max = maximum( values );
      printf("\nMaksimalna vrijednost je %d\n", max );
}
```

Ispis programa

```
Unesite 5 brojeva
7 23 45 9 121
Maksimalna vrijednost je 121
```

Program definira niz od pet elemenata (vrijednosti) i inicijalizira svaki element na vrijednost koju upiše korisnik. Niz **values** se tada predaje funkciji. Deklaracija

```
int  maximum( int values[5] )
```

definira ime funkcije **maximum**, i deklarira da je povratna vrijednost integer (rezultat), te da prima tip podatka **values**, koji je deklariran kao niz pet integera. Niz **array** u glavnom programu je sada poznat kao niz **values** unutar funkcije **maximum**. TO NIJE KOPIJA, VEĆ ORIGINAL. To znači da će svaka promjena izmijeniti originalni niz. Lokalna varijabla **max_value** se postavlja na prvi element niza, te se izvršava **for** petlja koja prelazi preko svakog elementa u nizu i najveću vrijednost pridružuje varijabli **max_value**. Ovaj broj se zatim vraća glavnom programu preko izraza **return**, i pridružuje se varijabli **max** u glavnom programu.

FUNKCIJE I NIZOVI

U prethodnom primjeru funkcija je ograničena jer može primiti niz od točno pet elemenata. Zbog toga je poželjno modificirati funkciju tako da može primiti i broj elemenata niza, pa se može iskoristiti za niz proizvoljne veličine. Modificirana verzija je:

```
#include <stdio.h>

int findmaximum( int [], int );          /* ANSI prototip funkcije */

int  findmaximum( int numbers[], int elements )
{
      int  largest_value, i;
      largest_value = numbers[0];

      for( i = 0; i < elements; ++i )
            if( numbers[i] > largest_value )
                  largest_value = numbers[i];
      return largest_value;
}
main()
{
      static int numb1[] = { 5, 34, 56, -12, 3, 19 };
      static int numb2[] = { 1, -2, 34, 207, 93, -12 };

      printf("maksimum numb1[] je %d\n", findmaximum(numb1, 6));
      printf("maksimum  numb2[] je %d\n", findmaximum(numb2, 6));
}
```

Ispis programa

```
maksimum numb1[] je 56
maksimum numb2[] je 207
```

11. STRUKTURE

C struktura je tip podatka koji je prikladan za grupiranje elemenata istih ili različitih tipova. Npr:

```
struct   datum{
      int  mjesec;
      int  dan;
      int  godina;
};
```

Ovo je deklaracija novog tipa podatka koji se zove **datum**. Ova C struktura se sastoji od tri osnovna člana koji su svi tipa **integer**. Ovo je definicija za kompajler. Ona ne kreira prostor za pohranu podataka i ne može se upotrebljavati kao varijabla. U stvari je to nova ključna riječ za tip podataka, kao što su int i char, i može se upotrebljavati pri deklaraciji varijabli. Varijable definirane tipom strukture ***datum*** mogu se definirati na slijedeći način:

```
struct  datum   danas;
```

definira varijablu koja se zove **danas** koja je novo-definiranog tipa podatka strukture **datum**.
Da bi pridružili konkretne vrijednosti članovima strukture, koristi se operator (**.**):

```
danas.dan = 17;
danas.mjesec = 11;
danas.godina = 1999;
scanf("%d\n",&danas.dan);

/* Program za prikaz C strukture */
#include <stdio.h>
```

```
struct datum {              /* globalna definicija tipa podatka */
      int mjesec;
      int dan;
      int godina;
};

main()
{
      struct datum  danas;          /* definicija strukturne varijable */
      danas.dan = 17;
      danas.mjesec = 11;
      danas.godina = 1999;
      printf("Današnji datum je %d/%d/%d\n",  danas.mjesec, danas.dan, danas.godina );
}
```

Primjer: učitava se današnji datum, a ispisuje sutrašnji:

```
#include <stdio.h>
      struct datum {
            int dan, mjesec, godina;
      };
      int br_dana[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
      struct datum danas,sutra;

      void unosdatuma( void );
      void unosdatuma( void )
      {
            int ok = 0;
            while( ok == 0 ) {
                  printf("Unesite trenutnu godinu (1990-1999)-->");
                  scanf("%d", &danas.godina);
                  if( (danas.godina  < 1990) || (danas.godina > 1999) )
                        printf("Pogrešna godina\n");
                  else
                        ok = 1;
            }
            ok = 0;
            while( ok == 0 ) {
                  printf("Unesite trenutni mjesec (1-12)-->");
                  scanf("%d", & danas.mjesec);
                  if( (danas.mjesec < 1) || (danas.mjesec  > 12) )
                        printf("Pogrešan mjesec\n");
                  else
                        ok = 1;
            }
            ok = 0;
            while( ok == 0 ) {
                  printf("Unesite današnji dan u mjesecu (1-%d)-->", br_dana[danas.mjesec -1]);
                  scanf("%d", &danas.dan);
                  if( (danas.dan < 1) || (danas.dan  > br_dana[danas.mjesec -1]) )
                        printf("Pogrešan dan\n");
                  else
                        ok = 1;
            }
      }

      main()
      {
            unosdatuma();
            sutra=danas;
```

```
        sutra.dan++;
        if( sutra.dan > br_dana[sutra.mjesec-1] ) {
            sutra.dan = 1;
            sutra.mjesec++;
            if( sutra.mjesec > 12 ) {
                sutra.godina++;
                sutra.mjesec = 1;
            }
        }
        printf("Sutrašnji datum je %02d:%02d:%02d\n", sutra.dan,sutra.mjesec,sutra.godina );
    }
```

Inicijalizacija strukture slična je inicijalizaciji niza. Članovi se jednostavno nabrajaju unutar zagrada i međusobno se odvajaju zarezom. Inicijalizaciji strukture prethodi ključna riječ **static**.

```
static struct datum danas = { 4,23,1998 };
```

Domaći rad:
- napraviti isti program za vrijeme (sat,minuta,sekunda)
- napraviti isti program uz uvažavanje prijestupnih godina

NIZOVI STRUKTURA

Razmotrimo slijedeće:

```
struct  datum  {
    int mjesec,dan,godina;
};
```

Kreirajmo niz koji se zove **praznik** i koji je tipa **datum**

```
struct datum praznik[5];
```

Kreiran je niz od 5 elemenata koji su tipa strukture datum.
Definiranje vrijednosti:

```
praznik[1].mjesec = 1;
praznik[1].dan   = 1;
praznik[1].godina  = 2000;
praznik[2].mjesec = 5;
praznik[2].dan = 1;
praznik[2].godina  = 2000;
```

itd.

mjesec	1	5	?	?	?
dan	1	1	?	?	?
godina	2000	2000	?	?	?
	praznik[0]	praznik[1]	praznik[2]	praznik[3]	praznik[4]

Primjer:

```
#include <stdio.h>

struct  datum  {    /*Globalna definicija strukture datum(date)*/
   int dan, mjesec, godina;
};

main()
{
     struct datum niz[5];
```

```
        int i;

        for( i = 0; i < 5; ++i ) {
            printf("Unesite datum (dan:mjesec:godinu)" );
            scanf("%d:%d:%d", &niz[i].dan, &niz[i].mjesec,
                &niz[i].godina );
        }
    }
```

NIZOVI KAO ČLANOVI STRUKTURE

Strukture mogu sadržavati i nizove.

```
    struct mjesec {
        int  broj_dana;
        char ime[4];
    };

    static struct mjesec ovaj = { 30, "Stu" };
```

ili

```
    ovaj.broj_dana = 30;
    strcpy( ovaj.ime, "Stu" );

    printf("Mjesec je  %s\n", ovaj.ime );
```

STRUKTURA UNUTAR STRUKTURE

Struktura može također sadržavati i strukturu. Razmotrimo slučaj u kojem su C struktura **datum** struktura **vrijeme** nalaze unutar iste strukture koja se zove ***datum_vrijeme***, npr,

```
    struct datum {
        int  mjesec, dan, godina;
    };

    struct vrijeme {
        int  sat, minuta, sekunda;
    };

    struct datum_vrijeme {
        struct date sdatum;
        struct time svrijeme;
    };
```

Ovo je deklaracija strukture čiji su članovi dvije već deklarirane strukture. Inicijalizacija se može napraviti na slijedeći način:

```
    static struct datum_vrijeme sada= { { 11, 17, 1999 }, { 9, 20,33 } };
```

Svakom elementu unutar strukture može se pristupiti na slijedeći način:

```
    ++sada.sdatum.dan=17;
    if( sada.svrijeme.sekunda == 60 ) ++ sada.svrijeme.sekunda;
```

Primjer: niz unutar strukture i niz struktura:

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

```
 /* Define a structure to hold entries. */

struct entry {
    char fname[20];
    char lname[20];
    char phone[10];
 };

 /* Deklaracija niza struktura*/

struct entry list[4];

 int i;

 main()
 {

    /* Petlja za unos podataka za četiri osobe. */

   for (i = 0; i < 4; i++)
   {
      printf("\nEnter first name: ");
      scanf("%s", list[i].fname);
      printf("Enter last name: ");
      scanf("%s", list[i].lname);
      printf("Enter phone in 123-4567 format: ");
      scanf("%s", list[i].phone);
   }

   /* Ispis dvije prazne linije */

   printf("\n\n");

   /* Petlja za ispis podataka */

   for (i = 0; i < 4; i++)
   {
       printf("Name: %s %s", list[i].fname, list[i].lname);
       printf("\t\tPhone: %s\n", list[i].phone);
    }

    return 0;
 }
```

```
Enter first name: Bradley
Enter last name: Jones
Enter phone in 123-4567 format: 555-1212
Enter first name: Peter
Enter last name: Aitken
Enter phone in 123-4567 format: 555-3434
Enter first name: Melissa
Enter last name: Jones
Enter phone in 123-4567 format: 555-1212
Enter first name: Deanna
Enter last name: Townsend
Enter phone in 123-4567 format: 555-1234
Name: Bradley Jones           Phone: 555-1212
Name: Peter Aitken            Phone: 555-3434
Name: Melissa Jones           Phone: 555-1212
Name: Deanna Townsend         Phone: 555-1234
```

12. DATOTEKE

Da bi se koristile datoteke u C programima, trebata deklarirati varijablu koja će biti pridružena određenoj datoteci (interni naziv datoteke). Ova varijabla mora biti tipa **FILE**, i mora se deklarirati kao pokazivač.

a) FILE je predifiniran tip. Deklaracija je

```
FILE  *var;
```

b) Internom nazivu datoteke (varijabli preko koje će se pristupati datoteci), pridružuje se eksterni naziv datoteke i definira se način rada s datotekom pomoću funkcije **fopen**

```
var = fopen( "naziv.dat", "r" );
```

U ovom primjeru, datoteka **"naziv.dat"** u trenutnom direktoriju je otvorena za čitanje (**r - read**).

c) Obrada podataka u datoteci; koristite odgovarajuće rutine za obradu podataka: **fprintf, fscanf, getc,putc** itd.

d) Kada završi obrada podataka u datoteci, ona se zatvara koristeći **fclose()** funkciju:

```
fclose( var );
```

Slijedeći primjer prikazuje **fopen** funkciju, te provjerava da li je datoteka uspješno otvorena.

```
#include <stdio.h>
   /* deklariramo pokazivač na  input_file, i na fopen funkciju */

FILE   *input_file, *fopen ();

/* pokazivaču na input file pridružena je vrijednostkoju vraća poziv fopen funkcije */
/* fopen pokušava otvoriti file koji se zove datain samo za čitanje(read) */
/* "w" = write(piši),  "a" = append(dodaj).  */

input_file = fopen("datain", "r");

      if( input_file == NULL ) {
       printf("*** datain se ne može otvoriti.\n");
       printf("povratak u dos.\n");
       exit(1);
}
```

Kraća varijanta:

```
if(( input_file = fopen ("datain", "r" )) == NULL ) {
       printf("*** datain se ne može otvoriti.\n");
       printf("povratak u dos.\n");
       exit(1);
}
```

UNOS/ISPIS JEDNOG ZNAKA

Jedan znak može se čitati/pisati u datotekama uporabom dviju funkcija, **getc()**, odnosno **putc()**.

```
int ch;
ch = getc( var );   /*  pridruži znak varijabli ch  */
```

Funkcija **getc()** također vraća vrijednost i za **EOF** (end of file - kraj datoteke),

```
while( (ch = getc( var )) != EOF )
      ......................

putc('a', var ); /* ispisuje znak a u izlaznoj datoteci */
```

Slijedeći program prikazuje kopiranje jedne datoteke u drugu upotrebom funkcija koje smo upravo objasnili.

```
#include <stdio.h>

main()   /* FCOPY.C    */
{
      char in_name[25], out_name[25];
      FILE *in_file, *out_file, *fopen ();
     int c;

      printf("Datoteka koja se kopira:\n");
      scanf("%24s", in_name);
      printf("Ime datoteke u koju se kopira:\n");
      scanf("%24s", out_name);

      in_file = fopen ( in_name, "r");
      if( in_file == NULL )
            printf("Ne može se otvoriti %s za čitanje.\n", in_name);
      else {
            out_file = fopen (out_name, "w");
            if( out_file == NULL )
                  printf("Ne može se otvoriti %s za pisanje.\n",out_name);
            else {
                  while( (c = getc( in_file)) != EOF )
                        putc (c, out_file);
                  putc (c, out_file);   /* copy EOF */
                  printf("Datoteka je kopirana.\n");
                  fclose (out_file);
            }
            fclose (in_file);
      }
}
```

I funkcijom **feof()** se može ispitati jesmo li na kraju datoteke, a nalazi se u **stdio.h** skupu funkcija. Ona vraća 1 ako se pokazivač nalazi na kraju datoteke:

```
if( feof ( input_file ))
      printf("Neme više podataka.\n");
```

NAREDBE **fprintf** i **fscanf** imaju istu funkciju kao **printf** i **scanf**, ali rade sa datotekama, na način da prvi argument označava datoteku:

```
fprintf(output_file, "Sada je vrijeme za sve...\n");
fscanf(input_file, "%f", &float_value);
```

Modovi otvaranja datoteke:
r Otvara datoteku za čitanje. Ako ona ne postoji, funkcija **fopen()** vraća NULL.
w Otvara datoteku za čitanje. Ako ona ne postoji, kreira se. Ako postoji, njen sadržaj se briše i kreira nova prazna datoteka.
a Otvara datoteku za dodavanje podataka. Ako ona ne postoji, kreira se. Ako postoji, novi podaci se dodaju na kraj postojeće datoteke.
r+ Otvara datoteku za čitanje i pisanje. Ako ona ne postoji, kreira se. Ako postoji, novi podaci se dodaju na početak datoteke, uništavajući postojeće.
w+ Otvara datoteku za čitanje i pisanje. Ako ona ne postoji, kreira se. Ako postoji, briše se cijeli sadržaj postojeće datoteke.
a+ Otvara datoteku za čitanje i dodavanje. Ako ona ne postoji, kreira se. Ako postoji, novi podaci se dodaju na kraj datoteke.

Primjer rada s **fprintf** funkcijom:

```
/* Demonstracija rada fprintf() funkcije. */
#include <stdlib.h>
#include <stdio.h>

void clear_kb(void);

main()
{
   FILE *fp;
  float data[5];
  int count;
  char filename[20];

  puts("Enter 5 floating-point numerical values.");

  for (count = 0; count < 5; count++)
     scanf("%f", &data[count]);

  /* Slijedeća funkcija je korisna za "čišćenje" svih znakova koji su eventualno uneseni s tastature, a ostali
  su upamćeni, što bi moglo poremetiti slijedeći unos  */
  clear_kb();

  /* Pročitaj ime datoteke i otvori je */
  puts("Enter a name for the file.");
  gets(filename);

  if ( (fp = fopen(filename, "w")) == NULL)
  {
     fprintf(stderr, "Error opening file %s.", filename);
     exit(1);
  }

  /* Ispiši numeričke podatke u datoteku i na ekran. */

  for (count = 0; count < 5; count++)
  {
     fprintf(fp, "\ndata[%d] = %f", count, data[count]);
     fprintf(stdout, "\ndata[%d] = %f", count, data[count]);
  }
  fclose(fp);
  printf("\n");
  return(0);
}

void clear_kb(void)
/* Kod funkcije koja čisti prethodni upis s tastature */
{
  char junk[80];
  gets(junk);
}
```

```
Enter 5 floating-point numerical values.
3.14159
9.99
1.50
3.
1000.0001
Enter a name for the file.
numbers.txt
```

```
data[0] = 3.141590
data[1] = 9.990000
data[2] = 1.500000
data[3] = 3.000000
data[4] = 1000.000122
```

Primjer rada sa **fscanf** funkcijom

```
/* Program čita datoteku s fscanf() funkcijom */
#include <stdlib.h>
#include <stdio.h>

main()
{
   float f1, f2, f3, f4, f5;
   FILE *fp;

  if ( (fp = fopen("INPUT.TXT", "r")) == NULL)
  {
     fprintf(stderr, "Error opening file.\n");
     exit(1);
  }

  fscanf(fp, "%f %f %f %f %f", &f1, &f2, &f3, &f4, &f5);
  printf("The values are %f, %f, %f, %f, and %f\n.",
        f1, f2, f3, f4, f5);

  fclose(fp);
  return(0);
}
```

The values are 123.45, 87.0001, 100.02, 0.00456, and 1.0005.

13. POKAZIVAČI (POINTERI)

Pokazivač je varijabla koja sadrži adresu (memorijsku) neke druge varijable. Pokazivači omogućavaju rad sa složenim strukturama podataka, mijenjanje vrijednosti argumenata funkcije, dinamičku alokaciju memorije, efektivniji rad s nizovima itd. Pokazivač osigurava indirektan način pristupa vrijednosti određenog podatka, i to pomoću adresa podatka u memoriji računala.
Pogledajmo kako pokazivači rade na jednostavnom primjeru:

```
int  count = 10, *pointer;
```

Deklarirana je cjelobrojna varijabla **count** koja ima vrijednost 10, i pokazivač nazvan **pointer**. Prefiks * definira da je to pokazivač. Nakon deklaracije ovih varijabli, na određenoj adresi u memoriji nalzi se vrijednost varijable **count** (10), dok vrijednost varijable pointer još nije definirana:

ime varijable	adresa	sadržaj
count	1001	10
pointer	1003	?

Pretpostavljeno je da cijeli broj u memoriji zauzima 2 bajta, a pokazivač također 2 bajta. Također je pretpostavljena proizvoljna početna memorijska lokacija na kojoj je smještena varijabla **count**, tako da je adresa slijedeće varijable za dva bajta veća (inače to obično ide obrnuto, tj. svaka slijedeća varijabla koja se deklarira nalazi se na nižoj adresi).

Napomena: pretpostavka o zauzeću 2 bajta za cjelobrojnu varijablu i pokazivač na većin novih računala (kompajlera) nije točna. Međutim dalje će se koristiti radi manjeg kompliciranja. Također je smještaj varijabli

proizvoljno stavljen od fiktivne memorijske adrese 1001, što je ponovo radi jednostavnosti. I na kraju, u memoriji su varijable obično smještene obrnutim redoslijedom, tj. kasnije deklarirana varijabla nalazi se na manjoj memorijskoj adresi.

Da bi uspostavili vezu između varijabli **pointer** i **count**, koristi se operator **&** koji daje adresu odgovarajuće varijable:

pointer = &count

Ovim je pridružena memorijska adresa varijable **count** pokazivaču **pointer**, tj. sadržaj varijable **pointer** je adresa varijable count:

ime varijable	adresa	sadržaj
count	1001	10
pointer	1003	1001

Da bi pridružili vrijednost varijable **count** nekoj drugoj varijabli korištenjem varijable **pointer**, koristi se operator * za pridruživanje, npr.

x = *pointer;

ima isto značenje kao i:

x=count;

tj. u jednom i drugom slučaju vrijednost varijable **x** je 10:

ime varijable	adresa	sadržaj
x	1005	10

Primjer:

```
#include <stdio.h>
int main()                  /* illustration of pointer use */
{
  int index, *pt1, *pt2;
```

ime varijable	adresa	sadržaj
index	1001	?
pt1	1003	?
pt2	1005	?

```
  index = 39;               /* any numerical value */
  pt1 = &index;             /* the address of index */
```

ime varijable	adresa	sadržaj
index	1001	39
pt1	1003	1001
pt2	1005	?

```
  pt2 = pt1;
```

ime varijable	adresa	sadržaj
index	1001	39
pt1	1003	1001
pt2	1005	1001

```
  printf("The value is %d %d %d\n", index, *pt1, *pt2);
  *pt1 = 13;            /* mijenja vrijednost varijable index */
                        /* isti učinak imale bi naredbe:  *pt2=13; i index=13;  */
```

ime varijable	adresa	sadržaj
index	1001	13
pt1	1003	1001
pt2	1005	1001

```
  printf("The value is %d %d %d\n", index, *pt1, *pt2);
  return 0;
}
```

Rezultat izvršenja programa:

```
The value is 39 39 39
The value is 13 13 13
```

POKAZIVAČKA ARITMETIKA

Neka su deklarirane varijable:

```
float f, *pf;
```

Naredbom

```
pf=&f;
```

dodjeljuje se adresa varijable **f** pokazivaču **pf**.

Naredbom

```
pf++;
```

ili

```
pf=pf+1;
```

vrijednost pokazivača **pf** neće se uvećati za 1, već za iznos koji odgovara broju bajtova koje zauzima float varijabla, tako da će sada pokazivač **pf** sadržavati adresu slijedeće memorijske lokacije (iza adrese varijable f).
Ovo vrijedi za sve tipove pokazivača: uvećanje/ umanjenje vrijednosti pokazivača za N stvarno ga uvećava/umanjuje za **N*sizeof(tip pokazivača)**.

Slijedeći primjer pokazuje slijedeće:
Ukoliko u programu deklariramo neki niz, naziv tog niza je ujedno i pokazivač kojemu se pri deklaraciji pridružuje adresa prvog elementa niza. Ujedno se, počevši od te adrese, rezervira potreban prostor gdje će biti smještene vrijednosti niza.
Zato su izrazi:

ime_niza[i] i ***(ime_niza + i)**

istovjetni, tj. oba označavaju vrijednost i+1-og elementa niza.

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

int main()
{
char strg[40], *there, one, two;
int  *pt, list[100], index;
```

ime varijable	adresa	sadržaj	
strg	1001	1003	naziv niza je pokazivač na adresu gdje će biti smještene vrijednosti niza
	1003	?	ovdje će biti smještene vrijednosti niza **strg**
there	1044	?	pokazivač kojemu još nije dodijeljena vrijednost
one	1046	?	
two	1047	?	
pt	1048	?	pokazivač kojemu još nije dodijeljena vrijednost
list	1050	1052	naziv niza je pokazivač na adresu gdje će biti smještene vrijednosti niza
	1052	?	ovdje će biti smještene vrijednosti niza **list**
index	1253	?	

```
strcpy(strg, "This is a character string.");
one = strg[0];
two = *strg;     /* ove dvije naredbe imaju isti učinak, tj. varijablama se pridružuje vrijednost prvog člana
                    niza strg */
```

ime varijable	adresa	sadržaj	
strg	1001	1003	
	1003	This is a character string.\0	nizau **strg** dodijeljena je vrijednost
there	1044	?	
one	1046	'T'	varijabli **one** dodijeljena je vrijednost 'T'
two	1047	'T'	varijabli **two** dodijeljena je vrijednost 'T'
pt	1048	?	
list	1050	1052	
	1052	?	
index	1253	?	

```
printf("The first output is %c %c\n", one, two);
one = strg[8];
two = *(strg+8);    /* ove dvije naredbe imaju isti učinak, tj. varijablama se pridružuje vrijednost devetog
                       člana niza strg */
printf("The second output is %c %c\n", one, two);
there = strg+10;          /* strg+10 je isto šta kao &strg[10], tj. varijabli there dodijeljena je adresa
                             11. člana niza strg */
printf("The third output is %c\n", strg[10]);
printf("The fourth output is %c\n", *there);
for (index = 0 ; index < 100 ; index++)
  list[index] = index + 100;
pt = list + 27;   /* pokazivaču pt dodijeljena je adresa 28. člana niza list */
```

ime varijable	adresa	sadržaj	
strg	1001	1003	
	1003	This is a character string.\0	
there	1044	1013	varijabli **there** dodijeljena je adresa 11. člana niza **strg**
one	1046	'a'	varijabli **one** dodijeljena je vrijednost 'a' (9. čl. niza **strg**)
two	1047	'a'	varijabli **two** dodijeljena je vrijednost 'a' (9. čl. niza **strg**)
pt	1048	1106	pokazivaču **pt** dodijeljena je adresa 28. člana niza **list**
list	1050	1052	
	1052	101 102 103 ... 199	vrijednosti dodijeljene nizu **list**
index	1253	100	zadnja vrijednost koja je ostala u varijabli **index**

```
printf("The fifth output is %d\n", list[27]);
printf("The sixth output is %d\n", *pt);

return 0;
}
```

Rezultat izvršenja programa:

```
The first output is T T
The second output is a a
The third output is c
The fourth output is c
The fifth output is 127
The sixth output is 127
```

POKAZIVAČI I STRINGOVI

Char pokazivač se može definirati da pokazuje na string znakova.

Primjer:

```
#include <stdio.h>
main()
{
      char *text_pointer = "Good morning!";
      for( ; *text_pointer != '\0'; ++text_pointer)
            printf("%c", *text_pointer);
    /* ili    printf("%s", text_pointer);  */
}
```

ime varijable	adresa	sadržaj
text_pointer	1001	1003
	1003	Good morning!\0
	1017	

pokazivaču se dodjeljuje adresa na kojoj se pri inicijalizaciji smještaju znakovi zadanog niza znakova slijedeći slobodni prostor u memoriji

Slijedeći primjer za niz stringova pokazuje kako se pokazivačima može definirati niz stringova. Budući da je svaki string niz znakova, radi se u stvari o dvodimenzionalnom nizu znakova, ali koji je predstavljen jednodimenzionalnim nizom pokazivača. Konačni rezultat je da svaki string može imati proizvoljnu (i različitu) duljinu, a zauzima točno potreban prostor za smještaj zadanih znakova, a osim toga svaki string mora imati pokazivač koji sadrži njegovu adresu:

```
#include <stdio.h>
main()
{
      static char *days[] = {"Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
                             "Saturday"};
      int i;
      for( i = 0; i < 6; ++i )
            printf( "%s\n", days[i]);
}
```

ime varijable	adresa	sadržaj
days	1001	1003
days[0]	1003	1017
days[1]	1005	1024
days[2]	1007	1031
days[3]	1009	1039
days[4]	1011	1049
days[5]	1013	1058
days[6]	1015	1065
	1017	Sunday\0
	1024	Monday\0
	1031	Tuesday\0
	1039	Wednesday\0
	1049	Thursday\0
	1058	Friday\0
	1065	Saturday\0

pokazivač **days** je ime niza i sadrži adresu prvog el. niza
prvi element niza je pokazivač koji sadrži adresu prvog stringa
drugi element niza je pokazivač koji sadrži adresu drugog stringa
itd.

POKAZIVAČ KAO ARGUMENT FUNKCIJE

Pokazivači se mogu prenositi kao argumenti funkcije (prenosi se adresa, a ne kopija varijable). Na taj način se promjenom argumenata u funkciji mijenjaju i vrijednosti odgovarajuće varijable u glavnom programu, što nije slučaj kad se prenosi obična varijabla kao argument (prenosi se vrijednost varijable).

Primjer:

```
#include <stdio.h>
void funkcija(int, int *);

int main()
{
int A,B;

  A = 100;
  B = 101;
```

ime varijable	adresa	sadržaj
A	1001	100
B	1003	101

```
 printf("Pocetne vrijednosti su %d %d\n", A, B);
                  /* kad pozovemo funkciju    */
  funkcija(A, &B);     /* dajemo vrijednost od A i adresu od B  */
```

ime varijable	adresa	sadržaj	
A	1001	100	varijabli **A** nije se promijenila vrijednost
B	1003	172	varijabli **B** vrijednost se promijenila

```
 printf("Konacne vrijednosti su: %d %d\n", A,B);
}

void funkcija(int AA, int *BB)   /* AA je cjelobrojna vrijednost   */
                                 /* BB je pokazivač na cjelobrojnu vrijednost */
{
  printf("Vrijednosti su: %d %d\n", AA, *BB);
```

ime varijable	adresa	sadržaj	
AA	2001	100	u privremenu varijablu **AA** prenesena je vrijednost varijable **A**
BB	2003	1003	u privremenu varijablu **BB** prenesena je adresa varijable **B**

```
  /* sada mijenjamo vrijednosti  */
  AA = 135;
  *BB = 172;
```

ime varijable	adresa	sadržaj	
AA	2001	135	promjenjena je vrijednost privremene varijable **AA** (ne i vrijednost var. **A**)
BB	2003	1003	vrijednost varijable **BB** (pokazivač) nije mjenjana, već je mjenjan sadržaj na toj adresi (što je u stvari sadržaj varijable **B** iz glavnog programa)

```
  printf("Vrijednosti su: %d %d\n" ,AA, *BB);
}
```

Rezultati izvršenja programa:

Početne vrijednosti su: 100 101
Vrijednosti su: 100 101
Vrijednosti su: 135 172
Konacne vrijednosti su: 100 172

Pogledajmo program koji ilustrira funkciju koja vraća samo jednu vrijednost na uobičajeni način (najveća vrijednost niza)

```
 #include <stdio.h>
 #define MAX 11

int najveci(int [], int);
```

```
main()
{
  int niz[MAX];
  int i,n;
  do{
       printf("Unesi br. el. niza: ");
       scanf("%d", &n);
  }while(n>MAX);
  for (i = 0; i < n; i++)
  {
     printf("Unesi vrijednost: ");
     scanf("%d", &niz[i]);
  }
  printf("\n\nNajveci element je: %d\n", najveci(niz,n));
  }

  int najveci(int x[], int n)
  {
  int i,m;
  m=x[0];
  for ( i = 0; i<n; i++)
  {
        if (x[i] > m)
        m = x[i];
   }
  return m;
 }
```

Ako želimo napisati funkciju koja za rezultat vraća 2 ili više vrijednosti, ne možemo koristiti gornji primjer, budući da s **return** možemo vratiti samo jedan rezultat.
Funkciju koja vraća dvije ili više vrijednosti može se definirati na način da su izlazne vrijednosti pokazivači, a sama funkcija je **void**, tj, ne vraća nikakav rezultat (povrat rezultata u ovakvim funkcijama često se koristi za indikaciju greške).
Npr. funkcija koja vraća najveću i najmanju vrijednost niza:

```
 #include <stdio.h>
 #define MAX 11

void najveci_i_najmanji(int [],int,int *,int *);

main()
{
  int niz[MAX];
  int i,najveci,najmanji,n;
  do{
       printf("Unesi br. el. niza: ");
       scanf("%d", &n);
  }while(n>MAX);
  for (i = 0; i < n; i++)
  {
  printf("Unesi vrijednost: ");
  scanf("%d", &niz[i]);
  }
  najveci_i_najmanji(niz,n,&najveci,&najmanji);
  printf("\n\nNajveci i najmanji elementi su: %d %d\n", najveci,najmanji);
}

void najveci_i_najmanji(int x[],int n,int *najv,int *najm)
{
```

```
  int i;
  *najv=x[0];
  *najm=x[0];
  for ( i = 0; i<n; i++)
  {
     if (x[i] > *najv)
                *najv = x[i];
     if (x[i] < *najm)
                *najm = x[i];
   }
}
```

Pogledajmo što se događalo za vrijeme izvršavanja programa:

Nakon unosa elemenata niza, pod pretpostavkom da je uneseno **n**=11 elemenata, situacija je slijedeća:

ime varijable	adresa	sadržaj	
niz	1001	1003	pokazivač niz je ime niza i sadrži adresu prvog el. niza
niz[0]	1003	5	prvi element niza (pretpostavljena je neka vrijednost)
niz[1]	1005	9	drugi element niza (pretpostavljena je neka vrijednost)
...	…	…	itd.
niz[10]	1023	7	zadnji element niza (pretpostavljena je neka vrijednost)
i	1025	12	pomoćna varijabla
najveci	1027	?	nije pridružena nikakva vrijednost
najmanji	1029	?	nije pridružena nikakva vrijednost
n	1031	11	broj članova niza
	1033		

Pozivom funkcije
najveci_i_najmanji(niz,n,&najveci,&najmanji);
kontrolu izvršavanja programa preuzima kod funkcije **najveci_i_najmanji**, na način da se:
- vrijednost pokazivača **niz** (adresa prvog elementa niza = 1003) se prenosi u privremeni pokazivač **x** u memorijskom prostoru rezerviran za privremene varijable tijekom izvođenja funkcije
- vrijednost varijable **n** = 11 se prenosi u vrijednost privremene varijable **n** unutar funkcije
- adresa varijabli **najveci** i **najmanji** prenosi se u pokazivače **najv** i **najm** unutar funkcije

Dakle, samo se za varijablu **n** prenjela konkretna brojčana vrijednost, dok su ostale prenesene varijable u stvari prenijele adrese varijabli iz glavnog programa. Situacija je slijedeća:

ime varijable	adresa	sadržaj	
niz	1001	1003	
niz[0]	1003	5	
niz[1]	1005	9	
...	…	…	
niz[10]	1023	7	
i	1025	12	
najveci	1027	?	
najmanji	1029	?	
n	1031	11	
			memorijski prostor za privremene varijable
x	2001	1003	prenesena je vrijednost varijable **niz**, a to je adresa početka niza
n	2003	11	prenesena je vrijednost varijable **n** iz glavnog programa
najv	2005	1027	prenesena je adresa varijable **najveci** iz glavnog programa
najm	2007	1029	prenesena je adresa varijable **najmanji** iz glavnog programa
			memorijski prostor za lokalne varijable funkcije
i	1031	?	

Sada će sve promjene u funkciji koje se rade preko pokazivača **najv** i **najm** rezultiraju promjenama vrijednosti varijabli **najveci** i **najmanji** u glavnom programu. Također privremena varijabla **x** sadrži istu adresu koja se nalazi i u pokazivaču **niz** u glavnom programu, tako da će vrijednosti **x[i]** uzimati sa adresa **niz[i]**, tj. radi se s vrijednostima niza iz glavnog programa.
Naredbama

```
*najv = x[i];
*najm = x[i];
```

mijenja se sadržaj na odgovarajućim adresama sadržanim u pokazivačima **najv** i **najm.** Budući da su to zapravo adrese varijabli **najveci** i **najmanji** iz glavnog promjene, mijenjati će se njihove vrijednosti u glavnom programu. Dakle, sve izmjene se preko ovih pokazivača dešavaju u glavnom programu.

FUNKCIJE KOJE VRAĆAJU POKAZIVAČ KAO REZULTAT

Funkcija može vratiti pokazivač kao rezultat, ako je definirana na odgovarajući način. U slijedećem primjeru, funkcija koja računa veći od dva ulazna argumenta je definirana na dva načina:
- **f1** je definirana sa **int f1(... ,** što znači da je rezultat funkcije cijeli broj
- **f2** je definirana sa **int *f1(...** , što znači da je rezultat funkcije pointer na cijeli broj, a ne cijeli broj

```
#include<stdio.h>

int f1(int, int);
int *f2(int *, int *);

main()
{
int a, b, r1, *r2;
printf("Unesi dva cijela broja: ");
scanf("%d %d", &a, &b);
r1 = f1(a, b);
printf("\nVeći je %d.", r1);
r2 = f2(&a, &b);
printf("\nVeći je %d.\n", *r2);
}

int f1(int x, int y)
{
   if (y > x)
      return y;
   return x;
}

int *f2(int *x, int *y)
{
   if (*y > *x)
      return y;
   return x;
}

/* rezultati
Unesi dva cijela broja: 1111 3000
Veći je 3000
Veći je 3000
*/
```

Nakon unosa brojeva imamo slijedeću situaciju:

ime varijable	adresa	sadržaj	
a	1001	1111	prvi uneseni broj
b	1003	3000	drugi uneseni broj
r1	1005	?	
r2	1007	?	

Pozivom funkcije **r1=f1(a, b),**
vrijednosti varijabli **a** i **b** se prenose u funkciju, u privremene varijable **x** i **y**

ime varijable	adresa	sadržaj	
a	1001	1111	
b	1003	3000	
r1	1005	?	
r2	1007	?	
x	2001	1111	prenesena vrijednost iz varijable **a**
y	2003	3000	prenesena vrijednost iz varijable **b**

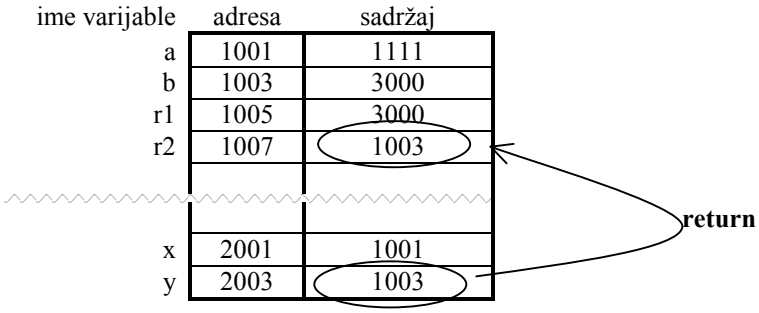
Return naredbom vraća se veća vrijednost kao rezultat funkcije (3000 u primjeru, dobivena **if** naredbom koja je usporedila vrijednosti varijabli **x** i **y**), što se u glavnom programu pridružuje cjelobrojnoj varijabli **r1**.

ime varijable	adresa	sadržaj	
a	1001	1111	
b	1003	3000	
r1	1005	3000	
r2	1007	?	
x	2001	1111	**return**
y	2003	3000	

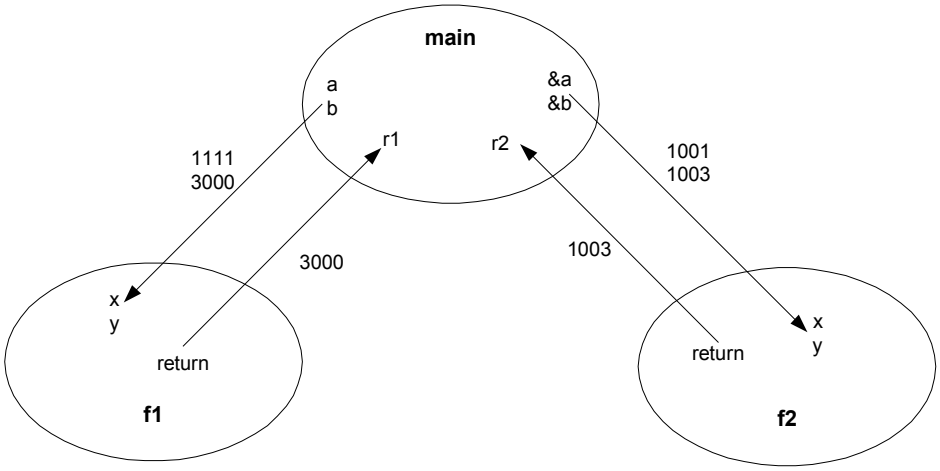
Pozivom funkcije **r2=f2(&a, &b),**
adrese varijabli **a** i **b** se prenose u funkciju, u privremene varijable (sada pokazivače) **x** i **y**

ime varijable	adresa	sadržaj	
a	1001	1111	
b	1003	3000	
r1	1005	?	
r2	1007	?	
x	2001	1001	prenesena adresa varijable **a**
y	2003	1003	prenesena adresa varijable **b**

Return naredbom vraća se adresa varijable čija je vrijednost veća (u primjeru adresa varijable **b** , dobivena **if** naredbom koja je usporedila vrijednosti koje se nalaze na adresama sadržanima u pokazivačima **x** i **y**). Adresa koju vraća funkcija u glavnom programu se pridružuje pokazivaču **r2**. Sada je u njemu adresa varijable čija je vrijednost veća, koja se slijedećom printf naredbom ispisuje.



Razmjena podataka između glavnog programa i funkcija prikazana je na slici:



POKAZIVAČI NA STRUKTURE

Pokazivači na strukturu se često koriste u C-u, pa radi jednostavnosti postoji poseban operator : **->**, koji omogućava da se pokazivač na strukturu (sadrži adresu strukture), umjesto

(*x).y

može zapisati kao:

x->y

Napomena: ***x.y** bi značilo ***(x.y)**, tj. **x.y** bi trebao biti neki pokazivač (radi prioriteta operatora * i .)

Primjer:

```
#include <stdio.h>
main()
{
  struct date {
      int month, day, year;
  };
  struct date today, *date_ptr;
  date_ptr = &today;
  date_ptr->month = 9;
  date_ptr->day = 25;
  date_ptr->year = 1983;
  printf("Današnji datum je %d/%d/%d.\n", date_ptr->month,date_ptr->day, date_ptr->year % 100);
   /* ili
      printf("Današnji datum je %d/%d/%d.\n", today.month, today.day, today.year % 100);
   */
 }
```

ime varijable	adresa	sadržaj	
today.month	1001	9	
today.day	1003	25	
today.year	1005	1983	
date_ptr	1007	1001	sadrži adresu prve komponente strukture **today**

Malo veći primjer:

```
#include <stdio.h>
struct child
{
  char initial;
  int  age;
  int  grade;
};

int main()
{
struct child  kids[12], *point, extra;
int index;

  for (index = 0 ; index < 12 ; index++)
  {
    point = kids + index;
    scanf("%c\n",&point->initial);
    scanf("%d\n",&point->age);
    scanf("%d\n",&point->grade);
  }

  for (index = 0 ; index < 12 ; index++)
  {
    point = kids + index;
    printf("%c  %d  %d\n",
              (*point).initial, kids[index].age, point->grade);
  }
  extra = kids[2];
  extra = *point;
}
```

U programu su deklarirani:
- niz (*kids*)od 12 elemenata tipa **struct child**
- pokazivač (*point*) tipa **struct child**
- obična varijabla (*extra*) tipa **struct child**

Nakon unosa podataka imamo slijedeću situaciju:

ime varijable	adresa	sadržaj	
kids	1001	1003	pokazivač kids je ime niza i sadrži adresu prvog el. niza
kids[0].initial	1003	m	prvi član niza – prva komponenta strukture
kids[0].age		12	prvi član niza – druga komponenta strukture
kids[0].grade		3	prvi član niza – treća komponenta strukture
kids[1].initial	1008	t	drugi član niza – prva komponenta strukture
kids[1].age		14	itd.
kids[1].grade		4	
…			
kids[11].initial	1058	r	
kids[11].age		13	
kids[11].grade		5	zadnji član niza – treća komponenta strukture
point	1063	?	još nedefinirani pokazivač
extra.initial	1065	?	još nedefinirana vrijednost strukture
extra.age		?	
extra.grade		?	
index	1070	13	

Drugom **for** petljom:

```
for (index = 0 ; index < 12 ; index++)
{
  point = kids + index;
  printf("%c  %d  %d\n",
          (*point).initial, kids[index].age, point->grade);
}
```

naredbom
point = kids + index; za početnu vrijednost varijable **index**=0, u pokazivač **point** dolazi adresa sadržana u **kids**, tj. adresa prvog elementa niza struktura:

point	1063	1003

U slijedećem koraku petlje, naredbom
point = kids + index; za vrijednost varijable **index**=1, pokazivaču **point** se ne pribraja vrijednost 1, već potreban broj bajtova: s obzirom da je pokazivač **point** pokazivač na tip **struct child** (koja je veličine 5 bajtova), pribrojiti će se vrijednost od 5 bajtova, pa će adresa sadržana u pokazivaču **point** sada biti 1008, tj. adresa slijedećeg elementa niza **kids** (POKAZIVAČKA ARITMETIKA):

point	1063	1008

Na taj način, pristuo nizu **kids** izveden je preko pokazivača **point.**
Ispis podataka napravljen je za svaku komponentu na drugi način, da se ilustrira tri načina kako se može pristupiti jednoj komponenti strukture (u primjeru onoj koja je na poziciji definiranoj varijablom **indeks**):
(*point).initial, kids[index].age, point->grade : sve se odnosi na isti element niza, tj. istu strukturu!

Pitanje za razmišljanje: šta će biti napravljeno naredbama:

```
extra = kids[2];
extra = *point;
```

14. DINAMIČKO ALOCIRANJE MEMORIJE

Sve varijable koje se deklariraju u programu su tzv. statičke varijable u smislu da se za njih rezervira potreban memorijski prostor pri pokretanju programa, a koji ostaje rezerviran do kraja izvršenja programa. Vrlo je često takav sistem neracionalan, tj. pa postoji način da se prostor za neke varijable (tzv. dinamičke varijable) rezervira tijekom izvršenja programa, a isto tako se može "otkazati" rezervacija, pa se prostor oslobađa.

Funkcija **malloc**

```
#include <stdlib.h>
void *malloc(size_t  size);
```

rezervira prostor u memoriji veličine **size** bajtova, i kao rezultat vraća pokazivač na početak rezerviranog prostora. Ako se traženi iznos ne može naći (rezervirati), kao rezultat se dobije NULL pointer (NULL pokazivač je definirana pokazivačka vrijednost, ali ne označava stvarnu adresu; u stvari njena vrijednost je 0).

primjer:
malloc(100) – treba rezervirati 100 bajtova
ili
malloc(sizeof(int)) - rezervira 2 (4) bajta jer je sizeof(int) 2 ili 4, ovisno o kompajleru

Pokazivač koji je rezultat funkcije je tipa **void**, što znači da nije definirano za koji tip podatka u njemu može biti pohranjena adresa. Međutim, rezultat funkcije se može pridružiti nekom pokazivaču preko *cast* operatora:

```
int *p;
```

ime varijable	adresa	sadržaj	
p	1001	?	deklaracijom se rezervira prostor za pokazivač p s još nedef. sadržajem

```
p=(int *)malloc(sizeof(int))
```

ime varijable	adresa	sadržaj	
p	1001	2001	funkcija malloc rezervirala je prostor negdje u memoriji (u primjeru na adresi 2001 za jedan cijeli broj, a ta adresa je pridružena pokazivaču **p**
	2001	?	rezervirani prostor (dinamička varijabla bez imena)

Rezerviranom prostoru se može pristupiti samo preko pokazivača **p**: naredbom

```
*p=5;
```

na adresi 2001 dolazi vrijednost 5.

Funkcija **free**

```
#include <stdlib.h>
void free(void *ptr);
```

oslobađa prostor u memoriji koji je prethodno rezerviran preko funkcije malloc, a čija se adresa nalazila u pokazivaču **ptr**. U gornjem primjeru, naredba

```
free(p)
```

bi oslobodila prostor u memoriji čija je adresa bila u pokazivaču **p**. Vrijednost koja se nalazi na toj adresi ostaje, ali joj se više ne može pristupiti:

ime varijable	adresa	sadržaj	
p	1001	?	u pokazivaču **p** više nema nikakve adrese
	2001	5	vrijednost je još na istoj adresi

Sličnu funkcionalnost ima i funkcija calloc(), koja ima dodatni parametar:
primjer:
calloc(5,sizeof(int)) - rezervira 5 puta prostor veličine sizeof(int), tj. prostor za 5 cijelih brojeva

Primjer:

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

struct animal
{
  char name[25];
```

```
  char breed[25];
  int  age;
};
int main()
{
  struct animal  *pet1, *pet2, *pet3;

  pet1 = (struct animal *)malloc(sizeof(struct animal));
  /* Ovo bi trebalo nadopuniti provjerom da li je pet1=NULL, tj. da li je uspjela rezervacija */
  gets(pet1->name);
  gets(pet1->breed);
  scanf("%d",&pet1->age);
  pet2 = pet1;   /* pet2 sada sadrži istu adresu kao pet1 */
  pet1 = (struct animal *)malloc(sizeof(struct animal));
  gets(pet1->name);
  gets(pet1->breed);
  scanf("%d",&pet1->age);

  pet3 = (struct animal *)malloc(sizeof(struct animal));
  gets(pet3->name);
  gets(pet3->breed);
  scanf("%d",&pet1->age);
  printf("%s is a %s, and is %d years old.\n", pet1->name, pet1->breed, pet1->age);
  printf("%s is a %s, and is %d years old.\n",  pet2->name, pet2->breed, pet2->age);
  printf("%s is a %s, and is %d years old.\n",  pet3->name, pet3->breed, pet3->age);
  pet1 = pet3;
  free(pet3);      /* oslobađa prostor sa adrese sadržane u pet3   */
  free(pet2);      /* oslobađa prostor sa adrese sadržane u pet2    */
/*  naredba
free(pet1);
bila bi greška! zašto? */
}
```

U programu su deklarirana samo tri pokazivača na strukturu tipa **struct animal**. Prostor za smještaj tri strukture alocira se **malloc** funkcijama. Nakon prvog poziva **malloc** funkcije, situacija je slijedeća:

ime varijable	adresa	sadržaj	
pet1	1001	2001	
pet2	1003	?	
pet3	1005	?	
	2001	?	alocirani prostor čija je adresa u **pet1**
		?	rezerviran je prostor koji odgovara veličini strukture **struct animal**
		?	
	2053	?	slijedeća slobodna memorijska lokacija

Nakon naredbi

```
  gets(pet1->name);
  gets(pet1->breed);
  scanf("%d",&pet1->age);
```

preko kojih se unose podaci za prvu strukturu, na adresi koju sadrži **pet1**, smještene su unesene vrijednosti. Npr.

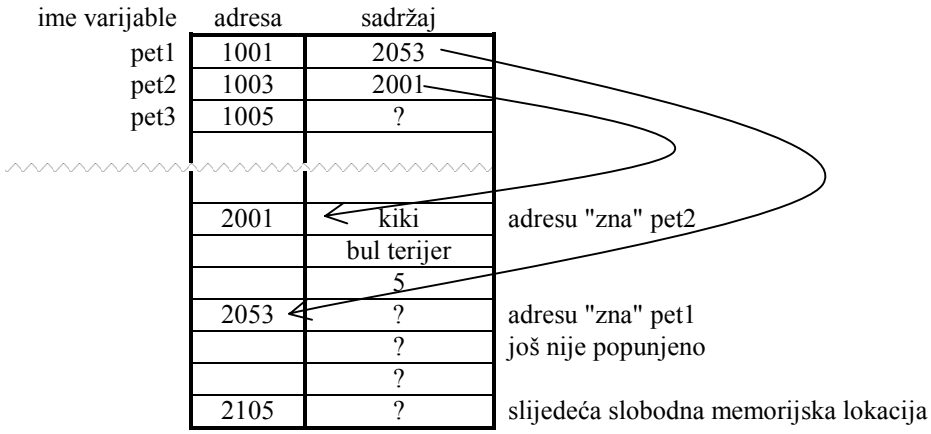
2001	kiki
	bul terijer
	5

Naredbom

```
  pet2 = pet1;
```

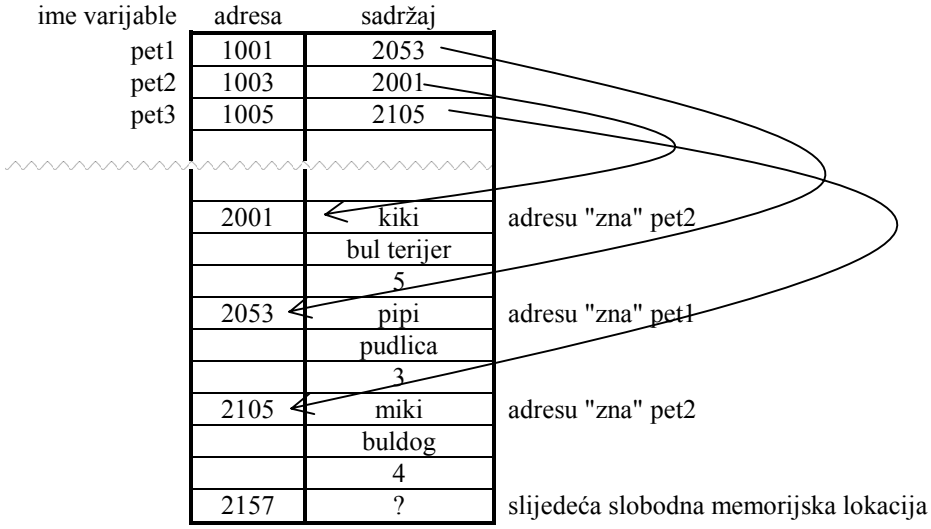
pokazivaču **pet2** se također pridružuje adresa 2001, a naredbom:

pet1 = (struct animal *)malloc(sizeof(struct animal));
u pokazivač **pet2** sada dolazi slijedeća slobodna adresa, prostora koji je alociran funkcijom **malloc:**



Naredbama
gets(pet1->name);
gets(pet1->breed);
scanf("%d",&pet1->age);
unose se podaci u strukturu čuja je adresa u **pet1**, a to je sada 2053.

Naredbama
pet3 = (struct animal *)malloc(sizeof(struct animal));
gets(pet3->name);
gets(pet3->breed);
scanf("%d",&pet1->age);
rezervira se prostor za još jednu strukturu na slijedećoj slobodnoj adresi (2105), koja se sada nalazi u pokazivaču **pet3**. Unose se podaci, pa je nakon ovih naredbi situacija slijedeća:



Printf naredbama ispisuje se sadržaj ovih struktura, kojima se pristupa preko pokazivača.
Naredbom:
pet1 = pet3;
U pokazivač **pet1** dolazi adresa iz **pet3** (2105). Ovime se "izgubio" pristup mjestu u memoriji preko kojeg se prije moglo doći pomoću pokazivača **pet1** (2053), tj. podaci o pasu *pipi* ostaju u memoriji, ali nema načina da im se pristupi. Dakle, *pipi* se sada može nazvati izgubljenim slučajem. Ovo je inače primjer česte greške u programiranju pri radu s pokazivačima (ali koja ostaje sakrivena), kad po memoriji ostane rezerviran prostor čija adresa nije sadržana u nekom pokazivaču, pa se taj prostor ne može ni osloboditi.
Naredbama:
free(pet3);

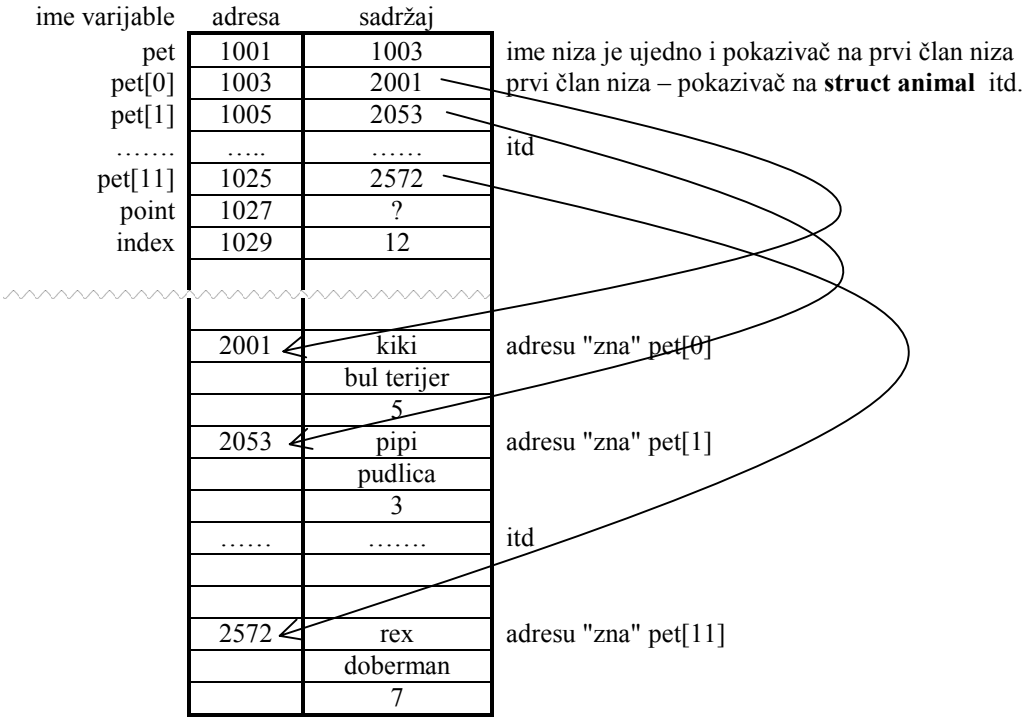
```
  free(pet2);
```
oslobađa se prostor na adresama 2001 i 2105 koji se sada može iskoristiti za nešto drugo.

Pitanje:
šta bi se dogodilo kad bi još napisali naredbu **free(pet1);**?

U slijedećem primjeru prikazana je dinamička alokacija niza struktura preko odgovarajućeg niza pokazivača:

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

struct animal
{
  char name[25];
  char breed[25];
  int  age;
};
int main()
{
  struct animal *pet[12], *point;   /* definirana su 12 + 1 pokazivača    */
  int index;
  for (index = 0 ; index < 12 ; index++)
  {
    pet[index] = (struct animal *)malloc(sizeof(struct animal));
    if (pet[index] == NULL)
    {
      printf("Memory allocation failed\n");
      exit (1);
    }
    gets(pet[index]->name);
    gets(pet[index]->breed);
    scanf("%d",&pet[index]->age );
  }
     /* ispis podataka  */

  for (index = 0 ; index < 12 ; index++)
  {
    point = pet[index];
    printf("%s is a %s, and is %d years old.\n", point->name, point->breed, point->age);
  }
     /* na kraju programa je uvijek dobro osloboditi rezervirani prostor   */
  for (index = 0 ; index < 12 ; index++)
    free(pet[index]);
}
```

Varijabla **pet** je niz od 12 pokazivača na strukturu tipa **struct animal**. U prvoj **for** petlji svakom članu ovog niza dodjeljuje se adresa dobivena alokacijom odgovarajućeg prostora za smještaj jedne strukture tipa **struct animal**. Ujedno se provjerava eventualna greška kod alokacije. Nakon svake alokacije, na preko pokazivača **pet[indeks]** se unose podaci u rezervirani prostor. Nakon završetka prve **for** petlje, situacija je slijedeća:



Drugom **for** petljom ispisuje se sadržaj svih struktura. Pristup svakoj strukturi vrši se pokazivačem **point**, kojemu se pridružuje svaki put slijedeći član niza **pet** (naredba **point=pet[index]**;).
Nakon ispisa, zadnjom **for** petljom se vrši "oslobađanje" alocirane memorije.

15. VEZANE LISTE (ovo još nije sređeno ni iskomentirano, tj. još je neupotrebljivo)

Jednostruko povezane liste:
- sastoje se od struktura u kojima je jedan član pokazivač na strukturu istog tipa:
  primjer:
```
struct naziv
{
  int a;
  struct naziv *slijedeci;
} *prvi, *novi;
```
- memorija za strukture-članove lista alocira se tijekom izvršenja programa, a strukturama se pristupa preko pokazivača
  Inicijalizacija:
```
prvi=NULL;  /* prazna lista */
```
  Ubacivanje prvog elementa u listu:
```
novi=(struct naziv *)malloc(sizeof(struct naziv))  /* rezervacija prostora */
novi->slijedeci=prvi ;   (u stvari =NULL)
prvi=novi;
```
  Ubacivanje drugog elementa u listu:
```
novi=(struct naziv *)malloc(sizeof(struct naziv))
novi->slijedeci=prvi;
prvi=novi;
```
  Itd. (ubacivanje se vrši na početak liste)

Primjer (svaki novi element liste ide na kraj liste):
```
#include <stdio.h>       /* this is needed to define the NULL     */
#include <string.h>
#include <stdlib.h>
#define RECORDS 6
```

```
struct animal
{
  char name[25];            /* The animals name                  */
  char breed[25];           /* The type of animal                */
  int  age;                 /* The animals age                   */
  struct animal *next;       /* pointer to another struct of this type */
} *point, *start, *prior;   /* this defines 3 pointers, no variables */

int index;

int main()
{
    /* the first record is always a special case              */
  start = (struct animal *)malloc(sizeof(struct animal));
  if (start == NULL)
  {
    printf("Memory allocation failed\n");
    exit (1);
  }

  gets(start->name);
  gets(start->breed);
  scanf("%d",&start->age );
  start->next = NULL;
  prior = start;

    /* a loop can be used to fill in the rest once it is started */
  for (index = 0 ; index < RECORDS ; index++)
  {
    point = (struct animal *)malloc(sizeof(struct animal));
    if (point == NULL)
    {
      printf("Memory allocation failed\n");
      exit (1);
    }

    gets(point->name);
    gets(point->breed);
    scanf("%d",&point->age );
    prior->next = point;  /* point last "next" to this record    */
    point->next = NULL;   /* point this "next" to NULL           */
    prior = point;        /* this is now the prior record        */
  }

    /* now print out the data described above                 */
  point = start;
  do
  {
    printf("%s is a %s, and is %d years old.\n",
                    point->name, point->breed, point->age);
    point = point->next;
  } while (point != NULL);

    /* good programming practice dictates that we free up the    */
    /* dynamically allocated space before we quit.               */

  point = start;            /* first block of group              */
  do
  {
    prior = point->next;   /* next block of data                 */
```

```
    free(point);          /* free present block              */
    point = prior;        /* point to next                   */
  } while (prior != NULL);  /* quit when next is NULL          */
}
```

Brisanje: (izbrisati element liste u kojemu je name="Kiki"; pretpostavka je da takav postoji u listi)

```
point = start;          /* first block of group            */
if (point->name=="Kiki")
        {
        start=point->next;
        free(point);
        }
else
        do
          {
            prior=point;
            point=point->next;
            if (point->name=="Kiki")
                  {
                  prior->next=point->next;
                  free(point);
                  break;
                  }
        } while (1);  /* izlaz iz petlje osiguran je break naredbom uz pretpostavku da je bar jedan  "Kiki" u listi */
```

Ubacivanje: (Ubaciti novi element liste, iza elementa u kojemu je name="Kiki"; pretpostavka je da takav postoji u listi) - domaći rad