

Programiranje (C)

9. predavanje

Saša Singer

`singer@math.hr`

`web.math.hr/~singer`

`www.math.hr/~singer`

PMF – Matematički odjel, Zagreb

Sadržaj predavanja

- Struktura programa:
 - Blokovska struktura jezika.
 - Doseg varijable — lokalne i globalne varijable.
 - Vijek trajanja varijable, memorijske klase.
 - Program smješten u više datoteka. Vanjski simboli.
- Polja (prvi dio):
 - Definicija i inicijalizacija polja.
 - Polja znakova, stringovi.
 - Polja kao argumenti funkcije.
 - Funkcije za rad sa stringovima.
 - Višedimenzionalna polja.

Struktura programa

Struktura C programa

C program je, zapravo

- skup definicija objekata — varijabli i funkcija:
 - varijable “modeliraju” ili zauzimaju memoriju, a
 - funkcije “modeliraju” ili sadrže instrukcije.

Komunikacija između funkcija ide preko:

- argumenata i vrijednosti koje vraćaju funkcije,
- vanjskih ili globalnih varijabli — to su one definirane izvan bilo koje funkcije.

Funkcije mogu biti

- u bilo kojem poretku u izvornom programu,
a program može biti smješten (rastavljen) u više datoteka
(sve dok ne “cijepamo” funkcije).

Struktura C programa (nastavak)

Objekti u C programu mogu biti:

- globalni ili vanjski (engl. external) — definirani izvan bilo koje funkcije, ili
- lokalni ili unutarnji (engl. internal) — što znači da su definirani “lokalno” unutar neke funkcije.

Bitno:

- Funkcije u C-u su uvek globalne ili vanjske, jer C
 - ne dozvoljava da se funkcije definiraju unutar neke druge funkcije (za razliku od nekih drugih jezika, poput Pascal-a).

Struktura C programa (nastavak)

Vanjski objekti imaju svojstvo da se

- svaka referenca na takav objekt s istim imenom zaista i odnosi na istu stvar, tj.
- isto ime ne može značiti dvije različite stvari!

Takva imena su, u načelu, tzv. vanjski simboli, a

● pripadni objekti su univerzalno dohvataljivi (dostupni), čak i kad su funkcije koje ih dohvaćaju u raznim datotekama (i prevode se odvojeno).

Ova univerzalna dohvataljivost može se ograničiti (ključnom riječi static za vanjske objekte, v. malo kasnije).

Struktura C programa (nastavak)

Za **unutarnje** objekte to ne vrijedi — oni **nisu** univerzalno dohvatljivi.

Funkcije ne mogu biti **unutarnji** objekti, tj.

- jedini unutarnji objekti su **variable**, ili preciznije,
- argumenti i **variable** definirani **unutar** funkcija (argumenti su, ionako, **lokalne** **variable**).

Krenimo od **lokalnih varijabli** — njih je **najlakše** objasniti,

- jer se definiraju **unutar** blokova, kao što smo dosad, uglavnom, i radili.

Blokovska struktura jezika

Blok naredbi je svaki **niz naredbi** koji se nalazi **unutar** vitičastih zagrada (recimo, tijelo funkcije).

- C **dozvoljava** da se u **svakom bloku** naredbi **deklariraju varijable**. Takve varijable zovu se **lokalne varijable**.
- Deklaracija varijabli **unutar** bloka **mora prethoditi prvoj** izvršnoj naredbi u bloku (standard **C90**).

Primjer:

```
if (n > 0) {  
    int i;      /* deklaracija varijable */  
    for (i = 0; i < n; ++i)  
        ...  
}
```

Blokovska struktura jezika (nastavak)

Pravila **dosega** (“vidljivosti” ili dostupnosti):

- **Varijabla** definirana **unutar** nekog bloka **vidljiva** je samo **unutar tog bloka** (samo tamo postoji).
- **Izvan** bloka toj varijabli se **ne može pristupiti** (ona ne postoji), tj. njeni **ime** izvan bloka **nije definirano**.
- **Izvan** bloka može biti deklarirana **varijabla istog imena**, ali ona je **nedostupna unutar** bloka, jer je “**prekrivena**” varijablom **istog imena**.
- **Varijabla** definirana **izvan** bloka **vidljiva** je u **tom** bloku ako **nije** “**pokrivena**” lokalnom varijablom **istog imena**, tj. ako u bloku **nije** definirana varijabla **istog imena**.

Ukratko, **dostupna** je samo “**najlokalnija**” varijabla **istog imena**.

Blokovska struktura jezika (nastavak)

Primjer:

```
int main(void) {
    int x, y;
    ...
    if (x > 0)
    {
        double y; /* int y NIJE vidljiv
                     u bloku */
        /* int x JE vidljiv u bloku */
        ...
    }
}
```

Blokovska struktura jezika (nastavak)

Formalni argument funkcije vidljiv je unutar funkcije i nije dohvatljiv (definiran) izvan nje.

- Doseg (vidljivost) formalnog argumenta je isti kao i doseg lokalne variable definirane na početku funkcije.

Primjer:

```
int x, y;  
...  
void f(double x) {  
    double y; /* int x i int y NISU */  
    ... /* vidljivi unutar funkcije */  
}  
int main(void) { ... }
```

Atributi variable

Varijabla je **ime** (sinonim) za neku **lokaciju** ili neki **blok lokacija** u **memoriji** (preciznije, za **sadržaj** tog bloka).

Sve **variable** imaju tri atributa: **tip**, **doseg** (engl. scope) i **vijek trajanja** (engl. lifetime).

- **Tip** = kako se **interpretira** sadržaj tog bloka (piše i čita, u bitovima), što uključuje i veličinu bloka.
- **Vijek trajanja** = u kojem **dijelu memorije** programa se rezervira taj blok.
 - Stvarno postoji **tri** bloka: **statički**, programski **stog** (“run–time stack”) i programska **hrpa** (“heap”).
- **Doseg** = u kojem **dijelu programa** je taj dio memorije “**dohvatljiv**” ili “**vidljiv**”, u smislu da se može koristiti — čitati i mijenjati.

Atributi variabile (nastavak)

- Prema **tipu** imamo variabile tipa
 - **int, float, char**, itd.
- Prema **dosegu** varijable se dijele na:
 - **lokalne** (unutarnje) i **globalne** (vanske).
- Prema **vijeku trajanja** mogu biti:
 - **automatske** i **statičke**.

Doseg i vijek trajanja određeni su, u principu, **mjestom** deklaracije, odnosno, definicije objekta (variable) — unutar ili izvan funkcije.

“Upravljanje” vijekom trajanja (a ponekad i dosegom) vrši se tzv. **identifikatorima memorejske klase** i to kod deklaracije objekta.

Automatske varijable

Automatska varijabla je

- svaka **varijabla** kreirana **unutar** nekog bloka (dakle, **unutar** neke funkcije),
- koja **nije** deklarirana s ključnom riječi **static**.

Automatske varijable se

- **kreiraju** na **ulasku** u blok u kome su deklarirane
- i **uništavaju** na **izlasku** iz bloka.

Memorija koju je automatska varijabla **zauzimala oslobađa se** za druge varijable.

Sve ovo se događa na tzv. “run–time stacku” (programske ili izvršni **stog**).

Automatske varijable (nastavak)

Primjer:

```
...
void f(double x) {
    double y = 2.71;
    static double z;
    ...
}
```

Automatske varijable mogu se inicijalizirati (kao što je to slučaj s varijablom y).

Inicijalizacija se vrši (ako je ima)

- pri svakom novom ulazu u blok u kojem je varijabla definirana (“rezerviraj memoriju i inicijaliziraj”).

Automatske varijable (nastavak)

Automatska varijabla koja **nije inicijalizirana** na neki način, na **ulasku** u blok u kojem je definirana

- dobiva **nepredvidljivu vrijednost** (“rezerviraj memoriju”, bez promjene sadržaja).

Inicijalizaciju automatske varijable moguće je izvršiti:

- **konstantnim** izrazom, ali i
- izrazom koji **nije konstantan**, kao u ovom primjeru:

```
void f(double x, int n) {  
    double y = n * x;  
    ...  
}
```

Identifikatori memorejske klase

Identifikatori memorejske klase su

- `auto`, `extern`, `static` i `register`.

(Stvarno još i `typedef`, s drugom svrhom, v. kasnije).

Oni služe **preciziranju vijeka trajanja** varijable. Neki (poput `static`) služe i za kontrolu **dosega** varijabli i funkcija.

Postavljaju se u **deklaraciji** varijable (ili funkcije) **ispred** identifikatora **tipa** varijable (ili funkcije).

Opći oblik deklaracije:

```
identif_mem_klase tip_var ime_var;  
identif_mem_klase tip_fun ime_fun ... ;
```

Identifikator auto

Primjer:

```
extern double l;  
static char polje[10];  
auto int *pi;  
register int z;
```

Identifikator **auto** deklarira **automatsku** varijablu.

Rijetko se upotrebljava u programima, jer su:

- sve **varijable** deklarirane **unutar nekog** bloka **automatske** (ako nisu deklarirane sa **static**),
- a sve **varijable** **izvan svih** blokova **statičke**.

Posljedica: riječ **auto** se **nikad ne mora** koristiti (ali može).

Identifikator register

Identifikator memorijske klase **register** može se primijeniti samo na **automatske** varijable.

- Ključna riječ **register** sugerira prevoditelju da će varijabla biti **često** korištena i da je treba smjestiti u **registar** procesora, a ne u “običnu” memoriju (ako ide), tako da se smanji vrijeme pristupa. Prevoditelj to **ne mora** “poslušati”.

Primjer:

```
int f (register int m, register long n) {  
    register int i;  
    ...  
}
```

Identifikator register (nastavak)

Zabranjeno je primijeniti adresni operator & na register varijablu (registri nemaju "standardne" adrese).

Savjet: pustiti stvar optimizaciji prevoditelja, da sam bira što će staviti u registre.

Staticke varijable

Staticka varijabla je

- varijabla definirana izvan svih funkcija, ili
- varijabla deklarirana u nekom bloku (na primjer, funkciji) identifikatorom memorijske klase **static**.

Statičke varijable “žive” svo vrijeme izvršavanja programa:

- kreiraju se na početku izvršavanja programa
- i uništavaju tek na završetku programa.

Moguće ih je eksplicitno inicijalizirati,

- ali samo konstantnim izrazima.

Ako nisu eksplicitno inicijalizirane, prevoditelj će ih sam inicijalizirati na nulu.

Statičke varijable (*nastavak*)

Primjer: sljedeći kôd **nije** ispravan jer nije inicijaliziran **konstantnim** izrazom.

```
int f(int j)
{
    static int i = j; /* greska */
    ...
}
```

Statička varijabla deklarirana **unutar** nekog bloka:

- inicijalizira se **samo jednom** i to pri **prvom** ulazu u blok,
- **zadržava** svoju vrijednost pri **izlasku** iz bloka (iako više **nije** dohvatljiva).

Statičke varijable (nastavak)

Primjer. Program koji ispisuje prvih 20 Fibonaccijevih brojeva:

$$F_1 = F_2 = 1, \quad F_i = F_{i-1} + F_{i-2}, \quad i \geq 3.$$

```
#include <stdio.h>
long fibonacci(int);
int main(void) {
    int i;
    for (i = 1; i <= 20; i++)
        printf(" i= %d, F= %ld\n",
               i, fibonacci(i));
    return 0;
}
```

Statičke varijable (nastavak)

```
long fibonacci(int i)
{
    static long f1 = 1, f2 = 1;
    long f;
    f = (i < 3) ? 1 : f1 + f2;
    f2 = f1;
    f1 = f;
    return f;
}
```

Statičke varijable **f1** i **f2** bit će inicijalizirane samo pri prvom pozivu funkcije **fibonacci**. Između svaka dva poziva, one **zadržavaju** svoju vrijednost, tako da je na početku *i*-tog poziva (za $i \geq 3$) stanje: $f1 = F_{i-1}$, $f2 = F_{i-2}$.

Doseg varijable

Doseg varijable je područje programa u kojem je varijabla dostupna (“vidljiva”).

Prema dosegu, varijable se dijele na:

- lokalne (imaju doseg bloka) i
- globalne (imaju doseg datoteke).

Svaka varijabla definirana unutar nekog bloka je

- lokalna varijabla za taj blok.

Ona nije definirana izvan tog bloka čak i kad je statička.

Statička lokalna varijabla postoji za cijelo vrijeme izvršavanja programa, ali

- može se dohvatiti samo iz bloka u kojem je deklarirana.

Globalne varijable

Globalna varijabla je

- varijabla definirana **izvan** svih **funkcija**.

Globalna varijabla (deklarirana izvan svih blokova)

- vidljiva je od mjesta **deklaracije** do **kraja datoteke**, ako **nije** “prekrivena” varijablom **istog** imena **unutar** nekog bloka.

Običaj: globalne varijable deklariraju se na **početku** datoteke, prije svih **funkcija**.

- Svaka **funkcija** može **doseći** globalnu varijablu u svom dosegu i **promijeniti** njenu vrijednost.

Na taj način više funkcija može **komunicirati** bez upotrebe formalnih argumenta.

Globalne varijable (nastavak)

Primjer. Tri funkcije rade na **istom** globalnom polju znakova.

```
#include <stdio.h>
#include <ctype.h>
char string[64]; /* globalna varijabla */
void ucitaj(void);
void malo_u_veliko(void);
void ispisi(void);

int main(void) {
    ucitaj();
    malo_u_veliko();
    ispisi();
    return 0;
}
```

Globalne varijable (nastavak)

```
void ucitaj(void) {
    fgets(string, sizeof(string), stdin);
}

void malo_u_veliko(void) {
    int i;
    for (i = 0; string[i] != '\0'; i++)
        string[i] = toupper(string[i]);
}

void ispisi(void) {
    printf("%s\n", string);
}
```

Globalne varijable (nastavak)

Primjer. Varijabla **a** vidljiva je i u funkciji **main** i u funkciji **f**, dok je varijabla **b** vidljiva u funkciji **f**, ali **ne** i u funkciji **main**.

```
int a;  
void f(int);  
int main(void) {  
    ...  
}  
  
int b;  
void f(int i) {  
    ...  
}
```

Program smješten u više datoteka

C program može biti smješten u više datoteka.

- Primjer: svaka funkcija definirana u programu može biti smještena u zasebnu *.c datoteku.

Globalne varijable i funkcije definirane u jednoj datoteci mogu se koristiti i u bilo kojoj drugoj datoteci, ako su tamo deklarirane.

- Za deklaraciju objekta koji je definiran u drugoj datoteci koristimo ključnu riječ extern.

Program smješten u više datoteka (nastavak)

Primjer. U datoteci 2 koristi se funkcija **f** iz datoteke 1.

Sadržaj datoteke 1:

```
#include <stdio.h>
int g(int);

void f(int i) {
    printf("i=%d\n", g(i));
}
int g(int i) {
    return 2 * i - 1;
}
```

Program smješten u više datoteka (nastavak)

U drugoj datoteci navodi se prototip funkcije f.

Sadržaj datoteke 2:

```
extern void f(int); /* extern i prototip */

int main(void) {
    f(3);
    return 0;
}
```

Vanjski simboli

U programu smještenom u više datoteka

- sve funkcije i globalne variabile mogu se koristiti i u drugim datotekama, ako su tamo deklarirane.

Stoga kažemo da su imena funkcija i globalnih varijabli vanjski simboli.

- Povezivanje deklaracija vanjskih simbola s njihovim definicijama radi linker ("povezivač").

Kada funkciju ili globalnu varijablu deklariramo identifikatorom memorijske klase static,

- ona prestaje biti vanjski simbol i može se dohvatiti samo iz datoteke u kojoj je definirana.

Ovdje static služi za ograničavanje dosega.

Vanjski simboli (nastavak)

Primjer. Želimo onemogućiti korištenje funkcije `g` izvan prve datoteke.

Sadržaj datoteke 1:

```
#include <stdio.h>
static int g(int); /* static ogranicava doseg */

void f(int i) {
    printf("i=%d\n", g(i));
}

static int g(int i) { /* static definicija */
    return 2 * i - 1;
}
```

Vanjski simboli (nastavak)

Sada funkciju **g** više ne možemo dohvatiti iz druge datoteke, pa je sljedeći program **neispravan**.

Sadržaj **datoteke 2** (prevodi se **bez greške**):

```
extern void f(int); /* extern i prototip */
extern int g(int); /* nije vanjski simbol! */

int main(void) {
    f(3);                  /* ispravno */
    printf("g(2)=%d\n", g(2)); /* neispravno */
    return 0;
}
```

Grešku javlja **linker**, jer ne može pronaći funkciju **g**.

Vanjski simboli (nastavak)

Primjer. Globalne varijable i funkcije definirane su u jednoj datoteci, a korištene u drugoj — gdje su deklarirane kao vanjski simboli.

Sadržaj datoteke 1:

```
#include <stdio.h>

int z = 3;          /* definicija varijable z */

void f(int i)    /* definicija funkcije f */
{
    printf("i=%d\n", i);
}
```

Vanjski simboli (nastavak)

Vanjski simboli moraju biti deklarirani (kao vanjski) prije upotrebe.

Sadržaj datoteke 2:

```
extern void f(int); /* deklaracija funkcije f */
extern int z;        /* deklaracija varijable z */

int main(void) {
    f(z);
}
```

Definicija i deklaracija globalnih varijabli

Kod globalnih varijabli treba razlikovati **definiciju** variable i **deklaraciju** variable (slično kao kod funkcija).

- Pri **definiciji** variable deklarira se njen **ime** i **tip**, i
 ● rezervira se memorijska lokacija za varijablu.
- Kod **deklaracije** samo se deklarira **ime** i **tip**,
 ● bez rezervacije memorije.

Podrazumijeva se da je varijabla **definirana negdje drugdje**, i da joj je **tamo** pridružena memorijska lokacija.

- **Definicija** variable je uvijek i njena **deklaracija**.
- **Globalna varijabla** može imati **više deklaracija**, ali **samo jednu definiciju**.

Pravila kod definicije i deklaracije

Globalne (ili vanjske) varijable dobivaju prostor u **statičkom** dijelu memorije programa, kao i sve **statičke** varijable. I pravila **inicijalizacije** su ista.

- Pri **definiciji** globalna varijabla može biti **inicijalizirana konstantnim** izrazom.
- **Globalna** varijabla koja **nije eksplicitno** inicijalizirana bit će **inicijalizirana nulom** (ili **nulama**).

Dodatna pravila:

- Pri **deklaraciji** globalne varijable **mora** se koristiti ključna riječ **extern**, a inicijalizacija **nije moguća**.
- Pri **definiciji** globalnog polja **mora** biti definirana njegova **dimenzija** (zbog rezervacije memorije). Kod **deklaracije** dimenzija **ne mora** biti prisutna.

Sužavanje dosega globalnih varijabli — static

Oznaka memorijske klase **static** može se primijeniti i na globalne variable s **istim** djelovanjem kao i za funkcije.

- **static** sužava **doseg** (područje djelovanja) variable na datoteku u kojoj je **definirana**. Ime takve variable više **nije vanjski simbol**.

Upozorenje: Oznaka memorijske klase **static** ispred globalne i lokalne variable ima **različito** značenje!

Primjer:

```
static int z = 3; /* z nevidljiv izvan datoteke */
void f(int i) {
    static double x; /* x je staticka varijabla */
    ...
}
```

Datoteke zaglavlja

Kad se program sastoji od više datoteka, onda se

- grupe deklaracija vanjskih simbola (varijabli i funkcija) smještaju u posebnu datoteku zaglavlja (*.h),
- koja se uključuje s #include "* .h" u svaku *.c datoteku kojoj su te deklaracije potrebne.

Na taj se način osigurava konzistentnost svih deklaracija.

Primjer. Deklaracije vanjskih simbola grupiramo u datoteku zaglavlja dekl.h. Sadržaj datoteke dekl.h:

```
extern void f(int);
extern int g(int);
extern int z;
```

Datoteke zaglavlja (*nastavak*)

Sadržaj datoteke 1:

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

int z = 3;      /* definicija varijable z */
void f(int i)  /* definicija funkcije f */
{
    printf("i=%d\n", g(i));
}
int g(int i)  /* definicija funkcije g */
{
    return 2 * i - 1;
}
```

Datoteke zaglavlja (*nastavak*)

Sadržaj datoteke 2:

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

int main(void)
{
    f(z);
    printf("g(2)=%d\n", g(2));
}
```

Uočite razliku između uključivanja sistemskih (<...>) i korisničkih ("...") datoteka zaglavlja.

Polja

Polje

Polje je niz varijabli istog tipa (sa zajedničkim imenom) numeriranih cjelobrojnim indeksom.

- Indeks uvijek počinje od nule.
- Radi efikasnosti pristupa, elementi polja smještaju se u uzastopne memorijske lokacije (redom po indeksu).

Primjer:

```
double x[3]; /* polje x tipa double */  
/* s 3 clana ili elementa */  
x[0] = 0.2;  
x[1] = 0.7;  
x[2] = 5.5;  
/* x[3] = 4.4; - greska, nije definirano */
```

Definicija polja

Jednodimenzionalno polje definira se na sljedeći način:

```
mem_klasa tip ime[izraz] ;
```

gdje je:

- **mem_klasa** memorijska klasa cijelog polja,
- **tip** tip podatka svakog elementa polja,
- **ime** ime polja (zajednički dio imena svih elemenata),
- a **izraz** konstantan, cjelobrojni, pozitivan izraz koji zadaje **broj** elemenata.

Ovaj **izraz** je najčešće pozitivna konstanta ili simbolička konstanta.

Definicija polja (nastavak)

Elementi jednodimenzionalnog polja su:

`ime[0], ..., ime[izraz - 1].`

Svaki element je varijabla tipa tip.

Deklaracija memorijske klase nije obavezna.

Polje deklarirano bez memorijske klase:

- unutar funkcije je automatska varijabla (rezervacija memorije na “run-time stacku”, ulaskom u funkciju),
- a izvan svih funkcija je staticka varijabla.

Unutar funkcije polje se može učiniti statickim pomoću identifikatora memorijske klase static.

Inicijalizacija polja

Polja se mogu **inicijalizirati** (element po element),

- navođenjem popisa **vrijednosti** elemenata unutar **vitičastih** zagrada.
- U tom popisu, pojedine vrijednosti **odvojene** su **zarezom** (koji **nije** operator).

Sintaksa:

```
mem_klasa tip ime[izraz] = {v_1, ..., v_n};
```

što daje

```
ime[0] = v_1, ..., ime[n - 1] = v_n.
```

Inicijalizacija polja (nastavak)

Primjer:

```
float v[3] = {1.17, 2.43, 6.11};
```

je ekvivalentno s

```
float v[3];
v[0] = 1.17;
v[1] = 2.43;
v[2] = 6.11;
```

Inicijalizacija polja (nastavak)

Ako je **broj** inicijalizacijskih vrijednosti **n**

- **veći** od **dimenzije** polja — javlja se **greška**,
- **manji** od **dimenzije** polja, onda će preostale vrijednosti biti inicijalizirane **nulom**.

Prilikom **inicijalizacije** dimenzija polja **ne mora** biti zadana.

- Tada se **dimenzija** polja računa **automatski**, iz **broja** inicijalizacijskih vrijednosti.

Primjer: možemo pisati

```
float v[] = {1.17, 2.43, 6.11};
```

što **kreira** polje **v** dimenzije **3** i inicijalizira ga.

Inicijalizacija polja (nastavak)

Polja znakova mogu se **inicijalizirati** znakovnim nizovima.

Primjer: naredbom

```
char c[] = "tri";
```

definirano je polje od **4** znaka:

c[0] = 't', c[1] = 'r', c[2] = 'i', c[3] = '\0'.

Takav način pridruživanja dozvoljen je **samo** u **definiciji variable** (kao inicijalizacija). **Nije dozvoljeno** pisati:

```
c = "tri"; /* Pogresno! Koristiti strcpy! */
```

jer lijeva strana pridruživanja **ne smije** biti **polje** (ime polja je konstantni pointer — adresa prvog elementa).

Polje kao argument funkcije

Zapamtiti: Ime polja je sinonim za

- konstantni pokazivač koji sadrži adresu prvog elementa polja (više u sljedećem poglavlju).

Polje može biti formalni (i stvarni) argument funkcije. U tom slučaju:

- ne prenosi se cijelo polje po vrijednosti (kopija polja!),
- već funkcija dobiva (po vrijednosti) pokazivač na prvi element polja.

Unutar funkcije elementi polja mogu se

- dohvati i promijeniti, korištenjem indeksa polja.

Razlog: aritmetika pokazivača (v. sljedeće poglavlje).

Polje kao argument funkcije (nastavak)

Funkciju **f** koja uzima **polje v** tipa **tip** kao argument, možemo deklarirati na **dva** načina:

f(tip v[]) ili **f(tip *v)**

U prvom načinu **ne treba** navesti dimenziju. Drugi način direktno kaže da je ime polja **v** pokazivač na objekt tipa **tip** i podrazumijeva se da je to **adresa prvog elementa polja**.

Ako **ne želimo** da funkcija **mijenja** elemente polja **unutar** funkcije, onda **dodajemo** ključnu riječ **const** na početku deklaracije argumenta:

f(const tip v[]) ili **f(const tip *v)**

Polje kao argument funkcije (nastavak)

Primjer. Funkciju koja uzima **polje** realnih brojeva (tipa **double**) i računa **srednju vrijednost** svih elemenata polja možemo napisati ovako:

```
double srednja_vrijednost(int n, double v[]) {  
    int i;  
    double suma = 0.0;  
  
    for (i = 0; i < n; i++) suma += v[i];  
    return suma/n;  
}
```

Uočite da je **broj** elemenata **n**, također, argument funkcije. Inače funkcija **ne zna** broj elemenata (osim iz neke globalne variabile).

Polje kao argument funkcije (nastavak)

Pri **pozivu** funkcije koja ima polje kao **formalni** argument, **stvarni** argument je

- ime polja ili pokazivač na “prvi” element u polju.

```
int main(void) {
    int n;
    double v[] = {1.0, 2.0, 3.0}, sv;

    n = 3;
    sv = srednja_vrijednost(n, v);
    return 0;
}
```

Poziv **srednja_vrijednost(2, &v[1])** je korektan!

Datoteka zaglavlja <string.h>

Za dodatne primjere funkcija i programa koji nešto rade s jednodimenzionalnim poljima brojeva — pogledajte vježbe i UuR. Posebno za pretraživanje i sortiranje polja.

Znakovni nizovi (stringovi) standardno se obrađuju funkcijama deklariranim u datoteci zaglavlja <string.h>.

Primjer. Program zasebno učitava ime i prezime (svako u jednom retku) i ispisuje ih zajedno u jednom retku.

```
#include <stdio.h>
#include <string.h>
char ime [128];
char prezime [128];
char ime_i_presime[128];
```

Datoteka zaglavlja <string.h> (*nastavak*)

```
int main(void)
{
    printf("Unesite ime:");
    gets(ime);
    printf("Unesite prezime:");
    gets(prezime);
    /* Spoji ime i prezime u jedan string */
    strcpy(ime_i_prezime, ime);
    strcat(ime_i_prezime, " ");
    strcat(ime_i_prezime, prezime);

    printf("Ime i prezime: %s\n", ime_i_prezime);
    return 0;
}
```

Datoteka zaglavlja <string.h> (*nastavak*)

Datoteka zaglavlja `<string.h>` deklarira **niz funkcija** za rad sa **stringovima**.

Funkcija `strcpy`:

```
char *strcpy(char *s, const char *t)
```

kopira string **t** u string **s** (uključujući i završni '`\0`') i vraća "string **s**", tj. pokazivač na prvi znak iz **s**.

Funkcija `strcat`:

```
char *strcat(char *s, const char *t)
```

nadovezuje (konkatenira) string **t** na **kraj** stringa **s** i vraća **s**.

Datoteka zaglavlja <string.h> (nastavak)

Funkcija `strcmp`:

```
int strcmp(const char *s, const char *t)
```

leksikografski uspoređuje stringove **s** i **t**. Vraća broj

- < 0 , ako je **s** $<$ **t**,
- $= 0$, ako je **s** $=$ **t**,
- > 0 , ako je **s** $>$ **t**.

Prava izlazna vrijednost je razlika znakova na prvom mjestu na kojem se stringovi razlikuju (onaj iz **s** minus onaj iz **t**), ako takvo mjesto postoji.

Tako je: `strcmp("A", "C") = -2.`

Datoteka zaglavlja <string.h> (*nastavak*)

Funkcija **strlen**:

```
size_t strlen(const char *s)
```

vraća **duljinu** stringa **s** (bez završnog '\0' znaka).

Funkcije **strchr**, **strstr**:

```
char *strchr(const char *s, const int c)
char *strstr(const char *s, const char *t)
```

vraćaju **pointer** na **znak** koji je (**početak**) prvog pojavljivanja **znaka c**, odnosno, **stringa t**, u **stringu s**, ako takvo mjesto postoji.

U protivnom, vraćaju **NULL** pointer (v. sljedeće poglavlje).