

# *Programiranje (C)*

## *10. predavanje*

Saša Singer

`singer@math.hr`

`web.math.hr/~singer`

`www.math.hr/~singer`

PMF – Matematički odjel, Zagreb

## Domaće zadaće

- Promjena: Obavezno predati jednu od dvije, a ne dvije od tri. (Razlog: nedostatak resursa na faksu.)
- Druga zadaća ima 4 zadatka.
  - Predaje se asistentima.
  - Asistent bira jedan od tih zadataka za predaju (kraće traje)!
  - Sva ostala pravila ostaju ista kao za prvu zadaću.
- Tko je predao prvu zadaću, može (ako hoće) predati i drugu, ali demonstratorima.

# *Sadržaj predavanja*

- **Polja** (zadnji dio):
  - Višedimenzionalna polja.
- **Pokazivači** (prvi dio):
  - Deklaracija pokazivača.
  - Pokazivači kao argumenti funkcije.
  - Operacije nad pokazivačima. Aritmetika pokazivača.
  - Pokazivači i polja.
  - Dinamička alokacija memorije.
  - Pokazivač na funkciju.
  - Argumenti komandne linije.
  - Složene deklaracije.

# Višedimenzionalna polja

# *Primarni operatori*

Pojedini **element polja** zadajemo tako da **iza imena polja u uglatim zagradama** pišemo pripadni indeks tog elementa:

---

ime[indeks]

---

Uglate zagrade [ ] su ovdje **primarni operator pristupa elementu polja**.

Sasvim analogno, kod poziva funkcije

---

funkcija(...)

---

okrugle zagrade ( ) su **primarni operator poziva funkcije**.

Primarni operatori pripadaju **istoj** grupi s **najvišim** prioritetom, a **asocijativnost** im je uobičajena  $L \rightarrow D$ .

# Višedimenzionalna polja

Deklaracija višedimenzionalnog polja ima oblik:

---

```
mem_klasa tip ime[izraz_1] ... [izraz_n];
```

---

Primjer: polje `m` deklarirano s

---

```
static float m[2][3];
```

---

predstavlja matricu s dva retka i tri stupca.

Njezine elemente možemo “prostorno” zamisliti u obliku:

<code>m[0][0]</code>	<code>m[0][1]</code>	<code>m[0][2]</code>
<code>m[1][0]</code>	<code>m[1][1]</code>	<code>m[1][2]</code>

Ovi elementi pamte se u memoriji računala jedan za drugim, tj. kao jedno jednodimenzionalno polje. Kojim redom?

# Višedimenzionalna polja (nastavak)

Odgovor slijedi iz asocijativnosti  $L \rightarrow D$  operatora `[ ]`.

To znači da je **višedimenzionalno** polje s deklaracijom:

---

```
mem_kl tip ime[izraz_1] [izraz_2] ... [izraz_n];
```

---

potpuno isto što i

- jednodimenzionalno polje “**duljine**” `izraz_1`,  
a elementi tog polja su
    - polja dimenzije manje za jedan, oblika
- 

```
tip[izraz_2] ... [izraz_n]
```

---

I tako redom, za svaku “**dimenziju**”, slijeva nadesno  $L \rightarrow D$ .

## Višedimenzionalna polja (nastavak)

Točno tako se **spremaju** elementi polja u memoriji računala:

- “**najsporije**” se mijenja **prvi** (najljeviji) indeks,
- a “**najbrže**” se mijenja **zadnji** (njadesniji) indeks.

Za **matrice** (dvodimenzionalna polja) to znači da su

- elementi poredani po **recima**, tj. **prvi** redak, **drugi** redak, ... (onako kako “čitamo” matricu),

jer se **brže** mijenja **zadnji** indeks — indeks **stupca**.

Napomena: U Fortranu je **obratno**!

Primjer. Dvodimenzionalno polje **m[2][3]** iz prethodnog primjera sastoji se od

- dva polja **m[0]** i **m[1]** tipa **float[3]**.

## Višedimenzionalna polja (nastavak)

U memoriji se **prvo** sprema “element”  $m[0]$ , a **zatim**  $m[1]$ .

Kad “raspakiramo” ta **dva** polja u elemente tipa **float**, poredak **elemenata** matrice **m** u memoriji je:

$m[0][0]$ ,  $m[0][1]$ ,  $m[0][2]$ ,  $m[1][0]$ ,  $m[1][1]$ ,  $m[1][2]$ .

Stvarno “**linearno**” indeksiranje elemenata, onako kako su **spremljeni** — indeksima od **0** do **5**, radi se na sljedeći način. Element  $m[i][j]$  **spremljen** je u memoriji na mjestu

$$i * \text{MAXY} + j,$$

gdje je **MAXY = 3** broj **stupaca** matrice.

Prva dimenzija polja (**MAXX = 2**), tj. broj **redaka** matrice, **ne treba** za indeksiranje, već samo za **rezervaciju** memorije.

# Višedimenzionalna polja (nastavak)

Primjer. Trodimenzionalno polje

---

```
float M[2][3][4];
```

---

je jednodimenzionalno polje s 2 elementa  $M[0]$  i  $M[1]$ . Ti elementi su dvodimenzionalna polja tipa  $\text{float}[3][4]$ .

Element  $M[i][j][k]$  bit će smješten na mjesto

$$(i * \text{MAXY} + j) * \text{MAXZ} + k,$$

gdje su  $\text{MAXX} = 2$ ,  $\text{MAXY} = 3$  i  $\text{MAXZ} = 4$  dimenzijske vrijednosti polja.

Prva dimenzija polja ( $\text{MAXX} = 2$ ) nije nužna za indeksiranje, već za rezervaciju memorije.

## Inicijalizacija polja

Primjer. Dvodimenzionalno polje `m` iz prethodnih primjera možemo inicijalizirati na sljedeći način:

---

```
static float m[2][3]
= {1.0, 2.0, 3.0, 4.0, 5.0, 6.0};
```

---

Inicijalne vrijednosti bit će pridružene elementima matrice onim redom kojim su elementi smješteni u memoriji (tj. po recima):

---

$$\begin{aligned}m[0][0] &= 1.0, & m[0][1] &= 2.0, & m[0][2] &= 3.0, \\m[1][0] &= 4.0, & m[1][1] &= 5.0, & m[1][2] &= 6.0.\end{aligned}$$

---

## *Inicijalizacija polja (nastavak)*

Prethodni način inicijalizacije je **nepregledan**.

Zato se **inicijalne vrijednosti** mogu

- **vitičastim zagradama grupirati u grupe**,

koje se **pridružuju pojedinim recima**.

Ekvivalentna inicijalizacija:

---

```
static float m[2][3] = { {1.0, 2.0, 3.0},  
                        {4.0, 5.0, 6.0}  
};
```

---

Grupiranje **odgovara dimenzijama**. Ako je neka grupa  
“**kraća**” od odgovarajuće dimenzije, **preostali elementi se**  
**inicijaliziraju nulama**.

## Inicijalizacija polja (nastavak)

Prvu dimenziju polja (ako **nije** navedena) prevoditelj može izračunati iz inicijalizacijske liste:

```
char A [] [2] [2] = { {{'a', 'b'}, {'c', 'd'}},  
                      {{'e', 'f'}, {'g', 'h'}}  
};
```

Rezultat: dobivamo **dvije** matrice znakova **A[0]** i **A[1]**, tipa **char [2] [2]**

$$A[0] = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \quad A[1] = \begin{bmatrix} e & f \\ g & h \end{bmatrix}.$$

# Višedimenzionalno polje kao argument funkcije

Višedimenzionalno polje kao formalni argument funkcije može se deklarirati navođenjem:

- svih dimenzija polja, ili
- svih dimenzija polja, osim prve.

Primjer. Funkcija koja čita matricu s MAXX redaka i MAXY stupaca može biti deklarirana ovako:

---

```
int mat[MAXX][MAXY];  
...  
void readmat(int mat[MAXX][MAXY],  
             int m, int n)
```

---

Argumenti **m** i **n** su stvarni brojevi redaka i stupaca koje treba učitati.

# *Višedimenzionalno polje kao argument funkcije*

Drugi način, bez prve dimenzije:

---

```
void readmat(int mat[] [MAXY] , int m, int n)
```

---

Prva dimenzija **MAXX**, tj. broj redaka matrice nije bitan za adresiranje elemenata matrice u funkciji.

Treći način, preko **pokazivača** (v. sljedeće poglavlje):

---

```
void readmat(int (*mat) [MAXY] , int m, int n)
```

---

# *Višedimenzionalno polje kao argument funkcije*

Primjer. Ilustracija inicijalizacije trodimenzionalnog polja.

```
#include <stdio.h>
char A[] [2] [2] = { {{'a', 'b'}, {'c', 'd'}},
                     {{'e', 'f'}, {'g', 'h'}} };
void f(char a[2] [2]); /* Ispis matrice */

int main(void)
{
    printf("Matrica A[0]:\n");
    f(A[0]);
    printf("Matrica A[1]:\n");
    f(A[1]);
    return 0;
}
```

# *Višedimenzionalno polje kao argument funkcije*

```
void f(char a[2][2]) /* Ispis matrice */
{
    printf("%c %c\n", a[0][0], a[0][1]);
    printf("%c %c\n", a[1][0], a[1][1]);
}
```

---

Ispis programa:

---

Matrica A[0] :

a b

c d

Matrica A[1] :

e f

g h

## **Primjer — množenje matrica**

Primjer. Program **množi** matricu  $A$  dimenzije  $2 \times 2$  s matricom  $B$  dimenzije  $2 \times 3$ . Produkt je matrica  $C = A \cdot B$  dimenzije  $2 \times 3$ .

---

```
#include <stdio.h>
double A[2][2] = { {1.0, 2.0}, {3.0, 4.0} };
double B[2][3] = { {0.0, 1.0, 0.0},
                  {1.0, 0.0, 1.0} };
double C[2][3]; /* Vanjsko polje je
                  staticka varijabla, pa je
                  inicijalizirano na nule! */
int main(void) {
    int i, j, k;
```

## *Primjer — množenje matrica (nastavak)*

```
for (i = 0; i < 2; ++i)
    for (j = 0; j < 3; ++j)
        for (k = 0; k < 2; ++k)
            C[i][j] += A[i][k] * B[k][j];

printf("Matrica C:\n");
for (i = 0; i < 2; ++i) {
    for (j = 0; j < 3; ++j)
        printf("%f ", C[i][j]);
    printf("\n");
}
return 0;
}
```

## *Primjer — množenje matrica (nastavak)*

Množenje matrica vrši se u **trostrukoj petlji**. Poredak petlji je **proizvoljan**, pa imamo  $3! = 6$  verzija algoritma.

- Računala imaju **hijerarhijski** organiziranu memoriju u kojoj se **bliske** memorijske lokacije mogu dohvatiti **brže** od udaljenih (**blok** transfer u “**cache**”).
- U unutarnjoj petlji (po **k**) — **redak** matrice **A** množi se **stupcem** matrice **B** (skalarni produkt). **Brzina?**
- Elementi **retka** od **A** su na **susjednim** lokacijama, pa je dohvat **brz**.
- Elementi **stupca** od **B** nalaze se na memorijskim lokacijama međusobno **udaljenim** za **duljinu retka**.
- Kod **velikih** matrica ta je udaljenost **velika** — posljedica je **sporiji** kôd.

## *Primjer — množenje matrica (nastavak)*

Efikasnija verzija algoritma ima “okrenute” petlje po  $j$  i  $k$ , tako da je petlja po  $j$  unutarnja:

```
...
for (i = 0; i < 2; ++i)
    for (k = 0; k < 2; ++k)
        for (j = 0; j < 3; ++j)
            C[i][j] += A[i][k] * B[k][j];
...
...
```

U unutarnjoj petlji (po  $j$ ) dohvaćaju se reci matrica  $C$  i  $B$ , a nema dohvata stupaca. Element  $A[i][k]$  može se čuvati u cacheu.

Ovo je daleko najbrža od svih 6 varijanti algoritma za velike matrice. (Kog zanima, nek' mi se javi.)

# Polja varijabilne duljine

Problem u C90 standardu za rad s matricama:

- dimenzijs polja u deklaraciji argumenata funkcije moraju biti konstantni izrazi.

Standard C99 uvodi polja varijabilne duljine, ali ih mnogi prevoditelji još u potpunosti ne implementiraju.

- Polje varijabilne duljine je automatsko polje čije dimenzijs mogu biti zadane varijablama.

Primjer:

---

```
int m = 3;
int n = 3;
double a[m][n]; /* PVD: polje var. duljine */
```

---

# Polja varijabilne duljine (nastavak)

Osnovna upotreba polja varijabilne duljine je

- pisanje funkcija koje kao argument imaju dimenzijske polja.

Primjer. Funkcija koja računa Frobeniusovu normu matrice. Ako je  $A = (a_{i,j})_{i=1}^m, j=1^n$  matrica tipa  $m \times n$ , onda je njena Frobeniusova norma

$$\|A\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n a_{i,j}^2}.$$

## *Polja varijabilne duljine (nastavak)*

```
double Fnorm(int m, int n, double A[m] [n])
{
    double norm = 0.0;
    int i, j;

    for (i = 0; i < m; ++i)
        for (j = 0; j < n; ++j)
            norm += A[i] [j] * A[i] [j];

    return sqrt(norm);
}
```

# *Polja varijabilne duljine (nastavak)*

Glavni program:

---

```
#include <stdio.h>
#include <math.h>
double Fnorm(int m, int n, double A[m][n]);
void Hilbert(int m, int n, double A[m][n]);
int main(void) {
    int m = 3, n = 3;
    double A[m][n];
    Hilbert(m, n, A);
    printf("norma matrice = %f\n",
           Fnorm(m, n, A));
    return 0;
}
```

---

## *Polja varijabilne duljine (nastavak)*

Funkcija **Hilbert** inicijalizira matricu  $A$  na tzv. Hilbertovu matricu:

$$a_{i,j} = \frac{1}{i + j - 1}, \quad i = 1, \dots, m, \quad j = 1, \dots, n.$$

---

```
void Hilbert(int m, int n, double A[m][n])
{
    int i, j;

    for (i = 0; i < m; ++i)
        for (j = 0; j < n; ++j)
            A[i][j] = 1.0 / (1.0 + i + j);
}
```

---

# Pokazivači

# Podsjetnik

- Svakoj **varijabli** u programu pridružena je **memorijska lokacija** (ili blok lokacija) čija **veličina** ovisi o **tipu varijable**.
- Svakoj **memorijskoj lokaciji** pridružena je **jedinstvena adresa**.
- Radi jednostavnosti, možemo zamišljati da je
  - **adresa** čitavog **bloka lokacija** = **adresa** **prve lokacije** u tom **bloku**.

Tako svaka **varijabla** ima svoju **jedinstvenu adresu**.

- **Varijabli** se može pristupiti korištenjem
  - **imena** **variable** — prevoditelj “zna” adresu,
  - **adrese** **variable** — **pokazivačem** na tu **varijablu**.

# Deklaracija pokazivača

Pokazivač na tip je varijabla koja sadrži adresu variable tog tipa tip.

Deklaracija:

---

```
mem_klasa tip *p_var;
```

---

Dakle, svi pokazivači imaju tip — onog na što pokazuju, osim tzv. generičkog pokazivača void \*p — pokazivača na bilo što.

Primjer:

---

```
static int *pi;           double *px;
char* pc;                 int a, *b;
float* pf, f;            void *p;
```

---

# **Adresni operator & i operator dereferenciranja \***

- Pri definiciji **pokazivačke** varijable, ona može biti **inicijalizirana** — adresom neke varijable.
- Varijabla čiju **adresu** koristimo, mora biti definirana **prije** no što se na nju primjeni **adresni operator**.

Primjer:

---

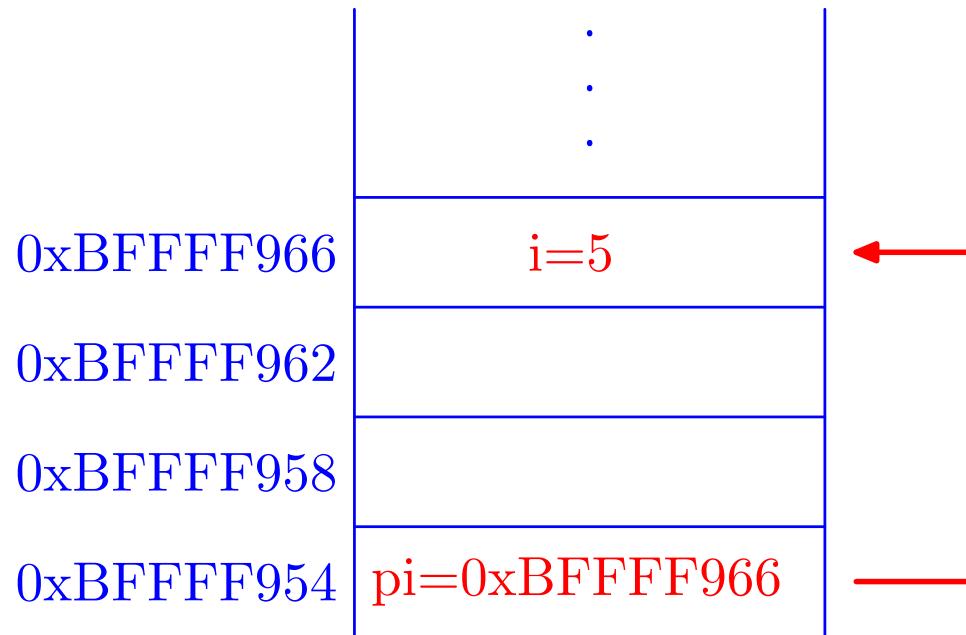
```
int i = 5;
int *pi = &i;
...
i = 2 * (*pi + 6);
printf("i = %d, adresa od i= %p\n", i, pi);
```

---

# *Adresni operator & i operator dereferenciranja \**

Uočite da je:

- $\&i$  = adresa varijable  $i$ ,
- $*pi$  = vrijednost spremljena u memorijsku lokaciju na koju pokazuje  $pi$ .



# *Pokazivači i funkcije*

- Pokazivači mogu biti argumenti funkcije.
- U tom slučaju funkcija može promijeniti vrijednost varijable na koju pokazivač pokazuje.

Primjer. Funkcija **zamjena** zamjenjuje vrijednosti cjelih brojeva **x** i **y**. Argumenti su **pokazivači** na **x** i **y**.

---

```
void zamjena(int *x, int *y) {  
    int temp = *x;  
    *x = *y;  
    *y = temp; }
```

---

Poziv funkcije (u glavnom programu) treba glasiti:

---

```
zamjena(&a, &b);
```

---

# *Pokazivači i funkcije (nastavak)*

Primijetite da sljedeći kôd za funkciju **zamjena** ne valja:

```
void zamjena(int x, int y) {  
    int temp = x;  
    x = y;  
    y = temp; }
```

Razlog je **prijenos argumenata po vrijednosti**. Pri pozivu funkcije

```
zamjena(a, b);
```

funkcija dobiva **kopije** stvarnih argumenata **a** i **b** koje međusobno zamjenjuje. To **nema nikakvog utjecaja** na stvarne argumente.

# *Polje kao argument funkcije*

- Ako funkcija ima **polje** kao argument, onda funkcija **ne dobiva kopiju čitavog polja**, već samo **pokazivač na prvi element polja**.
- Pri pozivu, funkciji se daje **samo ime polja** (bez uglatih zagrada) jer ono predstavlja **pokazivač na prvi element**.

---

```
char z[100];
void f(char *);

...
f(z);
f(&z[0]);
f(&z[50]);
```

---

Zadnji od poziva daje **zadnjih 50 elemenata** polja **z**.

# *Operacije nad pokazivačima*

- Aritmetičke operacije nad pokazivačima su dozvoljene i ekvivalentne su aritmetici indeksa u polju odgovarajućeg tipa, a ne aritmetici adresa.
- Svakom pokazivaču možemo dodati i oduzeti cijeli broj.

Primjer: Ako je `px` pokazivač i `n` varijabla tipa `int`, dozvoljene su operacije:

---

`++px`    `--px`    `px+n`    `px-n`

---

Pokazivač `px + n` pokazuje na `n`-ti objekt nakon onog na koga pokazuje `px`, tj.

`px + n`  $\iff$  adresa u `px`

`+ n * sizeof(tip objekta na koji pokazuje px)`.

## *Operacije nad pokazivačima (nastavak)*

Primjer:

---

```
int a[10], *ptr;  
  
ptr = a;      /* = &a[0] */  
ptr = ptr + 2;  
/* = &a[0] + (2 * sizeof(int)) = &a[2] */  
ptr++;        /* = &a[3] */
```

---

# *Operacije nad pokazivačima (nastavak)*

Primjer:

---

```
#include <stdio.h>

int main(void) {
    float x[] = {1.0, 2.0}, *px;
    px = &x[0];
    printf("Vrijednosti: x[0]=%g, x[1]=%g\n",
           x[0], x[1]);
    printf("Adrese      : x[0]=%x, x[1]=%x\n",
           px, px + 1);
    return 0;
}
```

---

## *Usporedba pokazivača*

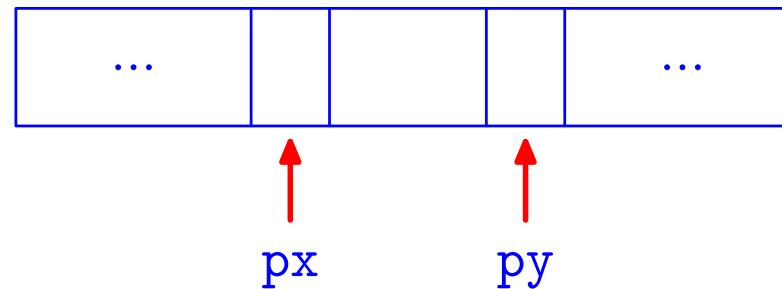
- Pokazivače **istog tipa** možemo međusobno uspoređivati pomoću relacijskih operatora.
- Uspoređivanje ima smisla ako pokazivači pokazuju na **isto polje**.

Ako su **px** i **py** dva pokazivača istog tipa, onda možemo koristiti izraze:

---

**px < py      px > py      px == py      px != py**

---



## *Usporedba pokazivača (nastavak)*

Sljedeće dvije petlje su ekvivalentne:

```
int i, *pi, x[10];
```

```
...
```

```
for (i = 0; i < 10; i++)
    x[i] = 1;
```

```
for (pi = &x[0]; pi <= &x[9]; ++pi, ++i)
    *pi = i;
```

Prva varijanta je, očito, bitno **jednostavnija** i **čitljivija!**

U prvoj verziji folija, **druga** petlja s pointerima imala je točno **tri** greške u tipkanju.

# *Pokazivači i cijeli brojevi*

- Pokazivaču nije moguće pridružiti vrijednost cjelobrojnog tipa, osim nule.
- Nula nije legalna adresa i ona označava da pokazivač nije inicijaliziran.

Može se pisati

---

`double *p = 0;`

---

ali je bolje naglasiti da se radi o pokazivaču i koristiti simboličku konstantu `NULL` definiranu u `<stdio.h>`.

---

`double *p = NULL;`

---

## *Pokazivači i cijeli brojevi (nastavak)*

Primjer:

```
double *px;  
...  
if (px != 0) ... /* Korektno!  
Je li pokazivac  
inicijaliziran? */  
if (px == 0x3451) ... /* GRESKA!  
Usporedjivanje  
s cijelim brojem */
```

## Važnost prioriteta i asocijativnosti

Unarni operatori `&` i `*` imaju viši prioritet od aritmetičkih operatora i operatora pridruživanja.

Primjer:

---

```
*px += 1;
```

---

Dolazi do povećanja za jedan vrijednosti na koju `px` pokazuje, a ne samog pokazivača. Zbog asocijativnosti unarnih operatora  $D \rightarrow L$ , isti izraz možemo napisati kao (prvo dereferenciranje, pa inkrementiranje):

---

```
++*px;
```

---

## **Važnost prioriteta i asocijativnosti (nastavak)**

Kod postfiks notacije operatora inkrementiranja, **moramo koristiti zagrade**:

---

`(*px)++;`

---

Izraz `*px++` inkrementira pokazivač nakon što vrati vrijednost na koju `px` pokazuje.

## Razlika pokazivača

- Jedan pokazivač možemo **oduzeti** od drugoga ako oni pokazuju na **isto polje**.
- Ako su **px** i **py** dva pokazivača (na isto polje) i ako je **py > px**, tada je **py - px + 1** broj elemenata između **px** i **py**, uključujući krajeve.
- Razlika pokazivača je vrijednost **cjelobrojnog tipa**, preciznije tipa **ptrdiff.t** definiranog u **<stddef.h>**.

Primjer: Implementacija funkcije **strlen** iz **string.h**.

---

```
int strlen (char *s) {
    char *p = s;
    while(*p != '\0') p++;
    return p - s; }
```

---

# Primjer

Primjer: Implementacija funkcije `strcpy`.

Verzija s indeksima: Funkcija kopira polje znakova na koje pokazuje `t` u polje na koje pokazuje `s`. Kopiranje se zaustavlja kada se kopira nul–znak `'\0'`.

---

```
void strcpy (char *s, char *t)
{
    int i = 0;
    while ((s[i] = t[i]) != '\0') i++;
}
```

---

## *Primjer (nastavak)*

Verzija s **pokazivačima**: Koristi viši prioritet operatora dereferenciranja od operatora pridruživanja i inkrementira pokazivače umjesto indeksa.

```
void strcpy (char *s, char *t)
{
    while( (*s = *t) != '\0' ) {
        s++; t++;
    }
}
```

## *Primjer (nastavak)*

Kraća verzija: Unarni operatori imaju asocijativnost  $D \rightarrow L$ , pa se kôd može skratiti. Pokazivači **s** i **t** povećavaju se nakon pridruživanja.

---

```
void strcpy (char *s, char *t)
{
    while ((*s++ = *t++) != '\0') ;
}
```

---

## *Primjer (nastavak)*

Još kraća verzija: Kod možemo još malo skratiti ako uočimo da je `... != '\0'` uspoređivanje izraza s nulom. Kod **slabo izražava** namjeru programera, pa ga treba izbjegavati.

---

```
void strcpy (char *s, char *t)
{
    while(*s++ = *t++) ;
}
```

---

# *Generički pokazivač*

Pokazivači na različite tipove podataka općenito se **ne mogu pridruživati**.

Primjer:

---

```
char *pc;  
int *pi;  
  
...  
pi = pc;          /* GRESKA */  
pi = (int *) pc; /* ISPRAVNO */
```

---

# *Generički pokazivač (nastavak)*

- Pokazivač može biti deklariran kao pokazivač na **void** i to je **generički pokazivač**.

---

```
void *p;
```

---

Pokazivač na bilo koji tip može se **konvertirati** u pokazivač na **void** i obratno, bez promjene pokazivača.

Primjer:

---

```
double *pd0, *pd1;  
void *p;  
.  
.  
.  
p = pd0; /* ISPRAVNO */  
pd1 = p; /* ISPRAVNO */
```

---

## **Generički pokazivač (nastavak)**

Osnovna uloga generičkog pokazivača je omogućiti funkciji uzmimanje pokazivača na bilo koji tip podatka.

Primjer:

---

```
double *pd0;  
void f (void *);  
...  
f(pd0); /* O.K. */
```

---

Generički pokazivač se **ne može** dereferencirati, povećati i smanjiti.