

# *Programiranje 2*

## *5. predavanje*

Saša Singer

`singer@math.hr`

`web.math.pmf.unizg.hr/~singer`

PMF – Matematički odsjek, Zagreb

# Sadržaj predavanja

- Znakovni nizovi — stringovi:
  - Polje znakova i string.
  - Pokazivači i stringovi.
  - Primjeri operacija sa stringovima (polja i pokazivači).
  - Funkcije za rad sa stringovima iz `<string.h>`.
  - Funkcije za rad sa znakovima iz `<ctype.h>`.
  - Primjeri obrade znakova i stringova.
- Dodatak:
  - String konstante (ponavljanje).
  - Inicijalizacija polja znakova i stringovi (ponavljanje).
  - Primjeri implementacije nekih funkcija iz `<string.h>`  
— preko polja i pokazivača.

## *Informacije — predavanja*

U petak, 5. 4., **nema predavanja** u redovitom terminu.

Cijeli dan održava se **Dan i noć na PMF-u** pa **nema nastave**.

# Uvod u stringove

# Uvod u stringove — polje (niz podataka)

Konačan niz podataka ili objekata nekog tipa odgovara pojmu vektora u matematici:

- vektor = uređeni niz “skalara” tog tipa.

U C-u takav niz prikazujemo

- jednodimenzionalnim poljem objekata tog tipa.

Već smo vidjeli da tip objekata (elemenata) u polju može biti

- jednostavni tip — standardni tipovi u C-u, ali i

- složeni tip, poput polja (za matrice i višedimenzionalna polja).

Može biti i neki drugi složeni tip — na primjer, struktura (v. kasnije).

# Duljina polja — rezervacija memorije

Zbog rezervacije memorije za polje — imamo fundamentalno ograničenje na rad s poljima:

- u definiciji polja moramo zadati duljinu polja = broj elemenata u polju (tzv. “maksimalni” broj elemenata).

Primjer:

---

```
double x[100];           // Polje od 100 doubleova.  
char poruka[128];       // Polje od 128 znakova.
```

---

- `x` je polje (niz) od 100 objekata tipa `double`,
- `poruka` je polje (niz) od 128 objekata tipa `char`.

Polje `x` zauzima 800 byteova, a `poruka` zauzima 128 byteova.

# Duljina polja — rezervacija memorije (nastavak)

Isto vrijedi i kod **dinamičke** alokacije (rezerviranja) **memorije**.

```
double *x;  
char *poruka;  
...  
x = (double *) malloc(100 * sizeof(double));  
poruka = (char *) calloc(128, sizeof(char));
```

Dakle, **duljina** polja je

- 📍 unaprijed **zadana** — **poznata** (tamo gdje se rezervira memorija za polje), kao konstanta ili vrijednost varijable, i **ne ovisi** o **sadržaju** polja.

Osim toga, uglavnom je **fiksna** — osim kad koristimo funkciju **realloc** za **promjenu** duljine **dinamičkog** polja.

# Obrada polja — stvarni broj elemenata

Kod **obrade** polja imamo **dva praktična** problema.

Polje u svakom trenu ima neku **duljinu**, tj. neki **fiksni** broj elemenata koji “**stane**” u njega. U praksi, naravno,

- **smijemo** raditi i s **manjim** brojem elemenata.

Zato se ovaj **fiksni** broj elemenata, koji “**stane**” u prostor za polje, obično zove “**maksimalni**” **broj elemenata** u polju.

**Praktični** problem:

- moramo stalno **paziti** na **stvarni broj** elemenata u polju s kojima radimo.

Striktno govoreći, ne samo na **broj** elemenata, već

- treba **paziti** i na njihove **indekse**, iako obično radimo s “**početkom**” polja (“radni” indeksi idu od **0**).



## Obrada polja — oprez u C-u

Iako se uvijek **točno** zna **koliko memorije** zauzima neko **polje**,

- C, u principu, **ne provjerava granice** indeksa, tj. koristimo li elemente u **rezerviranim** granicama memorije.

Posebno, to vrijedi za **polja** u **funkcijama**, kad je polje **formalni** argument. Razlog:

- u funkciju stiže samo **pokazivač** na “**prvi**” element polja.
- Taj “**prvi**” element **ne mora** biti zaista **prvi**, već može biti **bilo koji**. Obično je to “**prvi radni**” element u polju.

Funkcija tad “**nema pojma**” o količini memorije koja je (negdje drugdje) rezervirana za polje, tj. o **duljini** polja.

- Informacije o “**duljini**” radnog polja moraju, također, **stići** kao **argumenti**.

## Obrada polja — primjer “problema”

Primjer. Funkcija koja računa **Euklidsku** normu vektora **x**.

---

```
double norma(double x[], int n) {
    int i;
    double suma = 0.0;
    for (i = 0; i < n; ++i)
        suma += x[i] * x[i];
    return sqrt(suma);
}
```

---

Funkcija dobiva **pokazivač** na “**prvi radni**” element **x[0]** vektora **x**, a **n** je informacija o “**radnoj duljini**” tog vektora.

🔴 “**Radni** elementi” unutar funkcije su: **x[0]**, ..., **x[n-1]**.

# Obrada teksta i riječi — još veći “problem”

U praksi se vrlo često radi tzv. “**obrada teksta**”, gdje je

- **tekst** sastavljen od **riječi**, a **riječi** su neki nizovi **znakova**.

Pojedinu **riječ** (ili veću cjelinu teksta) možemo prikazati kao

- **polje znakova**.

Za **obradu riječi** ili komada teksta opet koristimo **funkcije**.

Ako te funkcije napišemo po “**uobičajenom**” predlošku za **polja**, imamo ozbiljan problem:

- razne **riječi** imaju vrlo **različit broj znakova** (tj. duljinu),

- pa bi trebalo posebno **pamtiti duljinu** za **svaku riječ**.

A to je podosta **nepraktično** — treba nam još i **polje** za **duljine**.

# Što je “problem”?

Kod obrade “običnih” polja (vektori, matrice), čak i kad ih je nekoliko,

- duljine (dimenzije) polja dolaze iz nekog cjelokupnog problema,
- i ne variraju pretjerano od polja do polja,

pa ih ima smisla pamtiti.

Za razliku od toga, u obradi teksta, čim imamo nekoliko nizova znakova (riječi),

- njihove “stvarne” duljine prirodno variraju od niza do niza.

Pogledajte [ovu](#) stranicu teksta!

# Potreba za stringovima

Ako za spremanje takvih nizova koristimo **polja znakova** (fiksne “maksimalne” duljine), onda je vrlo zgodno

- imati **poseban znak** koji označava “**kraj**” stvarnog ili radnog sadržaja niza.

Tada **ne treba pamtiti** duljinu, već se

- **kraj** niza može “**pročitati**” u samom nizu.

Realizacija takvih nizova znakova u C-u su **stringovi**.

---

"Ovo je jedan string."

---

**Napomena.** Slična ideja **oznake** za **kraj** uređenog niza podataka (nepoznate duljine) pojavit će se kod **vezane liste**.

# Stringovi

# Polja znakova i stringovi

Polje znakova ili “niz znakova” je bilo koje polje znakova.

---

```
char poruka[128];    // Polje od 128 znakova.
```

---

U ovoj definiciji:

- `poruka` je polje (niz) od 128 znakova — objekata tipa `char`.

String ili “znakovni niz” je polje (niz) znakova

- koje sadrži bar jedan nul-znak `'\0'`.

Prvi nul-znak `'\0'` u polju (onaj s najmanjim indeksom)

- ima ulogu oznake za kraj niza, tj. “radni sadržaj” niza nalazi se ispred tog znaka u nizu.

## Polje znakova i string — razlika

Svaki **string** je **polje znakova**. **Razlika** je samo u **interpretaciji** sadržaja. Ta razlika se “**vidi**” tek

- u načinu “**obrade**” ili “**upotrebe**” tog polja.

Na primjer, ako **polje**

- obrađujemo posebnim **funkcijama** za **stringove**, poput onih iz datoteke zaglavlja `<string.h>`, ili
- pišemo kao **string** (`%s`),

sadržaj se **interpretira** kao **string**.

**Oprez:** Polje tada

- mora** sadržavati nul-znak `'\0'` kao oznaku za **kraj**.

U protivnom, “**gazimo** po memoriji”.



## Duljina stringa — funkcija `strlen`

Definicija. Duljina stringa je broj znakova ispred nul-znaka, tj. duljina “radnog sadržaja” stringa.

Osnovna funkcija za rad sa stringovima je funkcija `strlen` koja vraća duljinu zadanog stringa.

Deklarirana je u datoteci zaglavlja `<string.h>`.

Prototip (zaglavlje) funkcije `strlen` je

---

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

---

Vraća duljinu stringa `s` — bez završnog `'\0'` znaka.

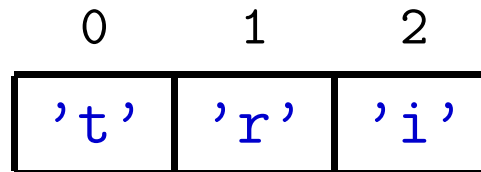
String je zadan pokazivačem na (konstantni) prvi znak, tj. funkcija ne smije promijeniti sadržaj stringa.

# Inicijalizacija polja — niz znakova i string

Primjer. Sljedeća definicija

```
char niz[3] = {'t', 'r', 'i'};
```

kreira polje `niz` od 3 znaka (`sizeof(niz) = 3`) i inicijalizira ga,



Taj niz znakova **nije string**, jer **ne sadrži nul-znak!**

Ponašanje funkcija za **stringove** na ovom nizu **nije definirano**.

**Napomena.** Da smo u definiciji stavili duljinu barem 4, dobili bismo **string**, zbog **dopune** nulama (nul-znakovima).

# Inicijalizacija polja — niz znakova i string

Primjer. Sljedeće dvije definicije su ekvivalentne

```
char s[4] = {'t', 'r', 'i', '\0'};  
char s[4] = "tri";
```

Kreiraju polje `s` od 4 znaka (`sizeof(s) = 4`) i inicijaliziraju ga,

0	1	2	3
't'	'r'	'i'	'\0'

Kod inicijalizacije stringom, dobiveni niz znakova je uvijek string, jer sadrži nul-znak!

Duljina niza `s` kao stringa je 3 (`strlen(s) = 3`).

# Pridruživanje stringa

U inicijalizaciji polja znakova konstantnim stringom, poput

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

piše pridruživanje “cijelog” stringa.

**Pitanje:** Kad smijemo napisati takvo pridruživanje cijelog konstantnog stringa?

**Odgovor:** za polje znakova,

- to je dozvoljeno samo u definiciji varijable — kao inicijalizacija, tj. ne smije pisati u naredbi pridruživanja.

Za razliku od toga, pokazivač na znak (`char`)

- smijemo inicijalizirati stringom u definiciji,
- i smijemo mu pridružiti string u naredbi pridruživanja.

# Pokazivači i stringovi

Uočiti bitnu **razliku** između **definicija**:

```
char amessage[] = "Poruka";    /* Polje. */  
char *pmessage = "Poruka";    /* Pokazivac. */
```

- **amessage** je **polje** od 7 znakova i smijemo **promijeniti sadržaj** polja, ali **ne** i **vrijednost amessage**, jer
  - **amessage** je **konstantni** pokazivač.
- **pmessage** je **pokazivač**, **inicijaliziran** tako da **pokazuje** na string konstantu.
  - Tom **pokazivaču** kasnije smijemo **promijeniti vrijednost** tako da **pokazuje** na nešto drugo.
  - Međutim, **ne smijemo** mijenjati **sadržaj** ovog stringa (kroz **\*pmessage**) — rezultat je nedefiniran!

## Pokazivači i stringovi (nastavak)

Na lijevoj strani naredbe pridruživanja smije biti bilo koji objekt koji nije konstantan (tzv. "lvalue" izraz).

To znači da je dozvoljeno promijeniti vrijednost tog objekta.

Isto vrijedi i za pokazivač na char. Ako nije konstantan,

• smijemo mu pridružiti i konstantni string.

Primjer:

---

```
char *pmessage;  
...  
pmessage = "Poruka";
```

---

Ovo je zaista operacija s pokazivačima. String konstanta je spremljena negdje u memoriji i ima svoju adresu.

## Pokazivači i stringovi (nastavak)

Za razliku od toga, kod polja znakova — nije dozvoljeno pisati:

```
char amessage[10], s[5];  
...  
amessage = "Poruka";    /* Pogresno! */  
s = "tri";              /* Koristiti strcpy! */
```

jer lijeva strana pridruživanja ne smije biti konstantni objekt (a ime polja je konstantni pokazivač).

Za pridruživanje stringa možemo koristiti funkciju strcpy.

```
strcpy(amessage, "Poruka");  
strcpy(s, "tri");
```

# Pokazivači i stringovi — primjer

Primjer (na istu temu):

```
char polje[10], *ptr;

polje = "hello";      /* Greska! */
ptr = "hello";        /* Dozvoljeno. */

printf("%s\n", ptr);  /* Ispis stringa. */
printf("%p\n", ptr);  /* Ispis adrese (%p). */
printf("%u\n", ptr);  /* Ispis adrese (%u)! */
```

Izlaz:

hello

013B318C

20656524 — %u nije za pokazivače (pretvoriti tip!).



# Pokazivači i stringovi — zadavanje i usporedba

**Bitno:** bilo koji **string** je “**zadan**”

- 📍 **pokazivačem** na **prvi** element (**znak**) — kao i svako polje!

To može dovesti do **neočekivanih** rezultata (pazite na **tipove**):

---

```
char s1[] = "Dobar dan", s2[] = "Dobar dan";  
...  
if (s1 == s2) printf("jednaki");  
else printf("razliciti");
```

---

**Rezultat** je **razliciti**, jer se uspoređuju **adrese** stringova, tj.

- 📍 **pokazivači** na **prve** elemente — a oni, naravno, **nisu** isti!

Usporedbu stringova “**znak po znak**” radi funkcija **strcmp**.

# Čitanje i pisanje stringova — ponavljanje Prog1

## Čitanje:

- funkcija `scanf`:
  - konverzija `%s` — string omeđen bjelinama,
  - konverzija `%[...]` i `%[^...]` — string koji **sadrži** ili **ne sadrži** navedene znakove.
- funkcija `gets` (ili `gets_s`) — **cijeli red**, uz `'\n'`  $\mapsto$  `'\0'`.

**Napomene.** U `scanf` **uvijek** treba zadati **maksimalnu širinu** (duljinu) polja. Također, koristite `fgets` (`gets_s`), a **ne gets!**

## Pisanje:

- funkcija `printf`:
  - konverzija `%s` — string **bez** završnog `'\0'`,
- funkcija `puts` — **cijeli red**, uz `'\0'`  $\mapsto$  `'\n'`.

# Čitanje i pisanje stringova — primjer

Primjer. Razlika između čitanja stringa s `%s` i do kraja reda:

---

```
#include <stdio.h>
#include <string.h>
int main(void) {
    char string[80];

    scanf("%79s", string);
    /* scanf("%79[^\n]", string); */

    puts(string);
    printf(" Duljina str. = %u\n", strlen(string));
    return 0; }
```

---

Ulaz: Dobar dan\n

# Čitanje i pisanje stringova — primjer (nastavak)

Ulaz:

Dobar dan\n

Izlaz — za `scanf("%79s", string)`:

Dobar

Duljina str. = 5

Izlaz — za `scanf("%79[^\n]", string)`:

Dobar dan

Duljina str. = 9

Opres: Kod čitanja s `gets` ili `gets_s`, učitava se i kraj linije.

# Obrada znakova i stringova

U C-u postoji mnogo funkcija za

- obradu stringova — u datoteci zaglavlja `<string.h>`,
- obradu znakova — u datoteci zaglavlja `<ctype.h>`.

Opis i moguća implementacija nekih funkcija ide u nastavku.

Kod dinamičke alokacije (rezervacije) memorije vidjeli smo da možemo “miješati” polja i pokazivače, tj. da isti niz možemo obrađivati

- i kao polje — preko indeksa,
- i preko pokazivača — koristeći aritmetiku pokazivača.

To se vrlo često koristi kod obrade stringova.

# Invertiranje stringa

**Primjer.** Treba napisati funkciju `invertiraj` koja **invertira** ulazni string.

- `String` je zadan **pokazivačem** na prvi element.
- **Invertira** = okreće niz **naopako** (v. `Prog1`).

Napomena: **Razlika** između **polja znakova** i **stringa**:

- za **polje** “stiže” i **radna duljina** = broj elemenata u polju,
- za **string** **ne** “stiže” ništa — sami **tražimo kraj** ili koristimo **strlen**.

Napravit ćemo **tri** varijante ove funkcije:

- preko **polja** — koristeći **indekse** (na dva načina), i
- preko **pokazivača** — koristeći **aritmetiku pokazivača**.

# Invertiranje stringa — indeksi

Primjer. Funkcija `invertiraj` — varijanta s `indeksima`.

---

```
void invertiraj(char s[])
{
    int p, k;    /* p = pocetak, k = kraj. */
    char temp;

    for (p = 0, k = strlen(s)-1; p < k; ++p, --k) {
        temp = s[p];
        s[p] = s[k];
        s[k] = temp;
    }
    return;
}
```

---

## Invertiranje stringa — indeksi (skraćeno)

“Kraća” varijanta (v. vježbe) — miče samo **jedan indeks**.

---

```
void invertiraj(char s[])
{
    int i, n = strlen(s);
    char temp;
    /* Ne koristiti strlen u petlji. */
    for (i = 0; i < n/2; ++i) {
        temp = s[i];
        s[i] = s[n - 1 - i];
        s[n - 1 - i] = temp;
    }
    return;
}
```

---



## Invertiranje stringa — pokazivači

Primjer. Funkcija `invertiraj` — varijanta s pokazivačima.

---

```
void invertiraj(char *s)
{
    char temp, *p, *k; /* p = pocetak, k = kraj. */

    p = s; k = p + (strlen(s) - 1);
    while (p < k) {
        temp = *p;
        *p++ = *k; /* <-- ! */
        *k-- = temp; /* <-- ! */
    }
    return;
}
```

---

## Invertiranje stringa — napomene

Napomena uz naredbe iz prošlog primjera (ponavljanje):

```
*p++ = *k;      /* <-- ! */
*k-- = temp;    /* <-- ! */
```

Unarni operatori  $*$ ,  $++$ ,  $--$  imaju asocijativnost  $D \rightarrow L$ .

U izrazu  $*p++$  prvo djeluje  $++$  na  $p$  (a ne na  $*p$ ), pa onda  $*$ . Dakle, inkrementira se pokazivač  $p$ , nakon što se obavi posao s ostatkom izraza, a to je  $*p$ .

To znači da je  $*p++ = *k;$  ekvivalentno s

```
*p = *k;  p++;
```

Analogno,  $*k-- = temp;$  dekrementira  $k$  nakon pridruživanja.

# Invertiranje stringa — glavni program

Primjer (nastavak). Glavni program (neću gets ili gets\_s).

---

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

void invertiraj(char *s);    /* ili (char s[]) */

int main(void) {
    char str[80];    /* Nije inicijaliziran! */

    /* Citanje i provjera za prazan string. */
    if (scanf("%79[^\n]", str) < 1) str[0] = '\0';

    printf(" String (duljine %u):\n", strlen(str));
    puts(str);
}
```

## Invertiranje stringa — glavni i test-primjer

```
invertiraj(string);  
printf(" Invertirani string:\n");  
puts(string);  
  
return 0;  
}
```

### Test-primjer:

```
String (duljine 33):  
Ja sam mala Ruza, mamina sam kci.  
Invertirani string:  
.ick mas animam ,azuR alam mas aJ
```

**Zadatak.** Probati “kraću” varijantu s pokazivačima.

## Datoteka zaglavlja <string.h>

Za obradu **stringova** standardno se koriste **funkcije** deklarirane u datoteci zaglavlja **<string.h>**.

Osnovnu funkciju za duljinu stringa već znamo!

Funkcija **strlen**:

---

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

---

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

String je zadan **pokazivačem s** na **konstantni prvi znak (\*s)**, tj. funkcija **ne smije** promijeniti **\*s = sadržaj** stringa.

## Funkcije iz <string.h> (nastavak)

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`.

**Prvi** znak iz `t` kopira se na mjesto **završnog** nul-znaka `'\0'` u stringu `s`, i tako redom, do kraja stringa `t` (uključujući `'\0'`).

# Funkcije iz <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`.

Ponekad se `strcmp` implementira tako da je izlazna vrijednost = razlika znakova na prvom mjestu na kojem se stringovi razlikuju (onaj iz `s` minus onaj iz `t`), ako takvo mjesto postoji. Na primjer,

• `strcmp("BAB", "BCB") = -2`, jer je `'A' - 'C' = -2`.

## Funkcije iz <string.h> (nastavak)

Funkcije `strchr`, `strstr`:

---

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

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

---

vraćaju pokazivač na znak koji je

- (početak) prvog pojavljivanja znaka `c` (za `strchr`), odnosno, stringa `t` (za `strstr`), u stringu `s`,

ako takvo mjesto postoji.

U protivnom, ako takvo mjesto ne postoji — vraćaju `NULL`.

Ovo nisu sve funkcije iz <string.h>, ima ih još. Pogledati na [webu](#) ili u [KR2](#).



## Primjer za funkcije iz <string.h>

Primjer. Program **zasebno** čita **ime** i **prezime** — svako u **svom** redu, uz preskakanje bjelina na početku reda. Zatim ih spaja u **jedan** string, s **prazninom** između njih, i ispisuje taj string.

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

int main(void) {
    char ime[128], prezime[128];
    char ime_i_prezime[128];    /* Bolje je 256. */

    printf("Unesite ime:");
    scanf(" %127[^\n]", ime);    /* Neprazan! */
    printf("Unesite prezime:");
    scanf(" %127[^\n]", prezime); /* Neprazan! */
```

## Primjer za funkcije iz <string.h> (nastavak)

```
        /* 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;
}
```

---

Ulaz: Sasa\n  
Singer\n

Izlaz:

---

Ime i prezime: Sasa Singer\n

---

# Funkcije za obradu znakova

## Datoteka zaglavlja <ctype.h>

Funkcije za **obradu znakova** deklarirane su u <ctype.h>.

Sve funkcije imaju **jedan** argument tipa **int**, koji **smije** biti:

- ili “**znak**” **EOF** (standardno, **EOF** = **-1**, i **zato** je tip **int**),
- ili **znak** prikaziv kao **unsigned char** (standardni znak).

**Izlazna** vrijednost je tipa **int**.

Funkcije iz <ctype.h> možemo podijeliti u **dvije** grupe:

- funkcije za **provjeru znakova** — vraćaju **int** **različit** od **nule** (**istina**), ako ulazni znak **pripada** određenoj **grupi znakova**. U **protivnom**, vraćaju **nulu** (**laž**);
- funkcije za **pretvaranje znakova** — vraćaju **pretvoreni** ulazni **znak**.

# Funkcije iz <ctype.h>

Funkcije za **provjeru znakova**:

---

```
int isalpha(int c);    /* Slovo, malo ili veliko */
int isdigit(int c);   /* Numer. = dec. znamenka */
int isalnum(int c);   /* Alfnumericki */
int isxdigit(int c);  /* Heksadecimalna znam. */
int islower(int c);   /* Malo slovo */
int isupper(int c);   /* Veliko slovo */
int iscntrl(int c);   /* Kontrolni znak */
int isgraph(int c);   /* Ispisiv, bez blanka */
int isprint(int c);   /* Ispisiv, uklj. blank */
int ispunct(int c);   /* Ispisiv, bez blanka,
                       slova i dec. znamenki */
int isspace(int c);   /* Bl, \n, \t, \v, \f, \r */
```

---

## Funkcije iz <ctype.h> (nastavak)

U 7-bitnom ASCII kôdu (0 do 0x7F, ili 0 do 127):

- **ispisivi** znakovi su: 0x20 ( ' ', tj. blank) do 0x7E ( '~' ),
- **kontrolni** znakovi su: 0 (NUL) do 0x1F (US) i 0x7F (DEL).

Primjer:

---

```
isdigit('0') = 1;    isdigit('C') = 0;
isalpha('0') = 0;    isalpha('C') = 1;
isxdigit('0') = 1;   isxdigit('C') = 1;
```

---

Funkcije za **pretvaranje** — mijenjaju samo **slova**:

---

```
int tolower(int c);    /* Velika u mala */
int toupper(int c);    /* Mala u velika */
```

---

## Primjer — implementacije nekih funkcija

Primjer. Moguće **implementacije** nekih funkcija iz `<ctype.h>`.

Funkcija `isdigit`:

---

```
int isdigit(int c) {  
    return ('0' <= c && c <= '9');  
}
```

---

Funkcija `isalpha`:

---

```
int isalpha(int c) {  
    return ('a' <= c && c <= 'z' ||  
           'A' <= c && c <= 'Z');  
}
```

---

## Primjer — implementacije toupper

Funkcija `toupper` (v. `Prog1`, funkcija `malo_u_veliko`):

---

```
int toupper(int c) {
    if ('a' <= c && c <= 'z')
        return ('A' + c - 'a');
    else
        return c;
}
```

---

ili, uvjetnim operatorom:

---

```
int toupper(int c) {
    return ('a' <= c && c <= 'z') ?
        ('A' + c - 'a') : c;
}
```

---



## Primjena — strtoupr za stringove

**Primjer.** Treba napisati funkciju `strtoupr` koja **sva mala slova** u zadanom **stringu** pretvara u **velika**.

---

```
void strtoupr(char *s)
{
    int i;

    for (i = 0; s[i] != '\0'; ++i)
        if (islower(s[i]))
            s[i] = toupper(s[i]);
    return;
}
```

---

# Primjena — strtoupr za stringove (nastavak)

Primjer (nastavak). Glavni program.

---

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

void strtoupr(char *s);

int main(void) {
    char kolegij[] = "Programiranje 2";

    printf(" Pocetni string:\n");
    puts(kolegij);
}
```

## Primjena — strtoupr za stringove (nastavak)

```
    strtoupr(kolegij);  
    printf(" Obradjeni string:\n");  
    puts(kolegij);  
  
    return 0;  
}
```

Izlaz:

```
Pocetni string:  
Programiranje 2  
Obradjeni string:  
PROGRAMIRANJE 2
```

# Obrada znakova i stringova

## Provjera znakova — zadaci

**Zadaci.** Napišite funkciju koja ima jedan **znak** (tipa **int**) kao argument i **provjerava** je li taj znak:

- **samoglasnik** (može malo ili veliko slovo) (v. vježbe),
- **suglasnik** (može malo ili veliko slovo),
- znak za **kraj rečenice** (**.**, **?**, **!**),
- **arimetrički operator** u C-u (**+**, **-**, **\***, **/**, **%**),
- **zagrada** (otvorena ili zatvorena; okrugla, uglata ili vitičasta).

Po ugledu na funkcije za provjeru znakova iz **<ctype.h>**, ako znak **zadovoljava** navedeni uvjet, izlazna vrijednost je **1**. U **protivnom**, izlazna vrijednost je **0**.

# Provjera znakova — samoglasnik

Primjer. Funkcija `samoglasnik` provjerava je li zadani znak samoglasnik (v. vježbe).

---

```
#include <ctype.h>

int samoglasnik(int c)
{
    /* Pretvori c u malo slovo
       za ubrzanje provjere. */
    c = tolower(c);

    return (c == 'a' || c == 'e' || c == 'i' ||
           c == 'o' || c == 'u');
}
```

---

## Provjera znakova — suglasnik

**Primjer.** Funkcija `suglasnik` provjerava je li zadani **znak suglasnik**. Koristimo činjenicu da je znak **suglasnik**

ako (i samo ako) je **slovo** i **nije samoglasnik**.

To je lakše od `gomile` provjera pojedinačnih znakova.

```
int suglasnik(int c)
{
    /* Koristi funkciju samoglasnik
       za ubrzanje provjere. */

    return (isalpha(c) && !samoglasnik(c));
}
```

Za cijeli program pogledajte `sugl.c`.

## Obrada znakova u stringu — zadaci

**Zadaci.** Pretpostavimo da je unaprijed **zadana** određena **vrsta znakova** — recimo, **samoglasnici**. Napišite funkciju koja ima **string** (tj. pokazivač na **char**) kao argument i **radi** sljedeće:

1. vraća **broj** takvih znakova u stringu,
2. to isto, a kroz **varijabilni** argument vraća **prvi** takav znak u stringu, ako ga ima (u protivnom, ne mijenja taj argument),
3. vraća **pokazivač** na **prvi** takav znak u stringu, ako ga ima (u protivnom, vraća **NULL**),
4. to isto, a kroz **varijabilni** argument vraća **broj** takvih znakova u stringu,
5. vraća **pokazivač** na **zadnji** takav znak u stringu, ako ga ima (u protivnom, vraća **NULL**).



## Samoglasnici u stringu — samogl\_2

Primjer. Funkcija `samogl_2`

- vraća broj samoglasnika u stringu,
- a kroz varijabilni argument vraća prvi samoglasnik u stringu, ako ga ima (u protivnom, ne mijenja taj argument).

Koristimo funkciju `samoglasnik` za provjeru znakova.

## Samoglasnici u stringu — samogl\_2 (nastavak)

```
int samogl_2(char *s, char *p_prvi)
{
    int broj = 0, i;

    for (i = 0; s[i] != '\0'; ++i)
        if (samoglasnik(s[i]))
            if (++broj == 1) *p_prvi = s[i];

    return broj;
}
```

# Samoglasnici u stringu — samogl\_4

Primjer. Funkcija `samogl_4`

- vraća **pokazivač** na **prvi samoglasnik** u stringu, ako ga ima (u protivnom, vraća **NULL**),
- a kroz **varijabilni** argument vraća **broj samoglasnika** u stringu.

Koristimo funkciju `samoglasnik` za provjeru znakova.

## Samoglasnici u stringu — samogl\_4 (nastavak)

```
char *samogl_4(char *s, int *p_broj)
{
    int broj = 0, i;
    char *p_prvi = NULL;

    for (i = 0; s[i] != '\0'; ++i)
        if (samoglasnik(s[i]))
            if (++broj == 1) p_prvi = s + i;

    *p_broj = broj;
    return p_prvi;
}
```

# Samoglasnici — glavni program

Primjer (nastavak). Glavni program (v. `samogl.c`).

---

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

int samoglasnik(int c);
int samogl_2(char *s, char *p_prvi);
char *samogl_4(char *s, int *p_broj);

int main(void) {
    char kolegij[] = "Programiranje (C)";
    char prvi = ' ', *p_prvi = &prvi;
    int broj = -1, *p_broj = &broj;
```

## Samoglasnici — glavni program (nastavak)

```
printf(" Pocetni string:\n");
puts(kolegij);

broj = samogl_2(kolegij, p_prvi);
printf(" samogl_2: broj = %d, prvi = %c:\n",
       broj, *p_prvi);

p_prvi = samogl_4(kolegij, p_broj);
printf(" samogl_4: broj = %d, prvi = %c:\n",
       *p_broj, *p_prvi);

return 0;
}
```

---

# Samoglasnici — izlaz

Izlaz:

---

Pocetni string:

Programiranje (C)

samogl\_2: broj = 5, prvi = o:

samogl\_4: broj = 5, prvi = o:

---

# Stringovi — ponavljanje iz Prog1



# Konstantni stringovi

U C-u postoje objekti koji uvijek imaju “oblik” **stringa** (polja znakova koje završava **nul-znakom**). To su

- konstantni stringovi.

Konstantni string piše se kao:

- niz znakova — različitih od **dvostrukog** navodnika " ,
- unutar (između) para **dvostrukih** navodnika " .

**Dvostruki** navodnici **nisu** dio **sadržaja** niza, već samo

- označavaju **početak** i **kraj** konstantnog stringa.

**Sadržaj** niza (polja) je navedeni **niz znakova** (tim redom), i

- nul-znak `'\0'`, kao **zadnji znak** u nizu (polju).

# Konstantni stringovi — primjer

Primjer. Konstantni string

"Izlaz:\n"

je polje od 8 znakova (7 navednih i nul-znak na kraju):

0	1	2	3	4	5	6	7
'I'	'z'	'l'	'a'	'z'	':'	'\n'	'\0'

Razlika između “duljine” (veličine) polja i duljine stringa:

```
printf("%u\n", sizeof("Izlaz:\n"));    /* 8 */
printf("%u\n", strlen("Izlaz:\n"));    /* 7 */
```

# Konstantni stringovi — primjer (nastavak)

Konstantne stringove smo već koristili

- za “**humani**” tekstualni **izlaz** (poruke).

Osim toga, **funkcije scanf** i **printf** za **formatirani** ulaz i izlaz

- imaju **konstantni** “format” **string** kao **prvi** argument.

Primjeri:

---

```
"Zagreb"
```

```
"01/07/2001"
```

```
"Linija 1\nLinija 2\nLinija3"
```

```
"String %s ima %u znakova\n"
```

---

**Zapamtiti:** **Zadnji** znak u **polju** (nizu) je nul-znak ‘\0’ — iza svih navedenih znakova.

## Konstantni string i znak — razlika

Napomena: 'a' nije isto što i "a".

- 'a' je tipa `char` i sadrži 1 znak: `a`,
- "a" je polje od 2 znaka: `a` i `\0`.

# Konstantni stringovi — napomene

Ako želimo **dvostruki** navodnik " kao **znak u nizu**,

• pišemo ga kao `\`".

Primjer:

---

```
"Program \"gazi\" po memoriji.\n"
```

---

Napomene:

- **Početni** i **završni dvostruki** navodnik **moraju** biti u **istom** redu teksta programa. Izuzetak (v. sljedeća stranica):
  - znak `\` na **kraju** reda označava da se taj red **nastavlja** u sljedećem redu.
- **Dva** konstantna stringa napisana **jedan za drugim spajaju** se u **jedan** string (konkatenacija — v. funkcija `strcat`).

## Dugački konstantni stringovi

Kako pisati **dugačke** stringove koji **ne stanu** uredno u jedan red programa? Imamo **dvije** mogućnosti:

- koristimo znak `\` na kraju linije, kao oznaku da će se string **nastaviti** u sljedećem redu (od početka reda), ili
- **rastavimo** string u **nekoliko** manjih stringova — jedan za drugim, a oni će se **nadovezati** (spojiti) u jedan.

**Primjer.** Stringovi **s0**, **s1** i **s2** su **jednaki**.

---

```
char s0[] = "Vrlo dugacak niz znakova";
char s1[] = "Vrlo dugacak \
niz znakova";
char s2[] = "Vrlo dugacak "
           "niz znakova";
```

---

# Inicijalizacija polja znakova i stringovi

Polje znakova može se u definiciji inicijalizirati (element po element) na dva načina:

- popisom znakova — kao i svako drugo polje,
- konstantnim stringom.

Inicijalizacija popisom znakova (ponavljanje):

- popis znakova piše se unutar vitičastih zagrada,
- u tom popisu, pojedini znakovi odvojeni su zarezom (koji nije operator).

**Oprez:** Dobiveni niz znakova ne mora biti string, tj. ne mora sadržavati nul-znak.

# Inicijalizacija polja — pravila (ponavljanje)

Zato je korisno ponoviti pravila o inicijalizaciji polja u definiciji polja — posebno za polja znakova.

Ako je broj inicijalizacijskih vrijednosti

- veći od dimenzije polja — javlja se greška,
- manji od dimenzije polja, onda će preostale vrijednosti biti inicijalizirane nulom (inicijalizira se cijelo polje).

Za polja znakova:

- “dodane” nule su ekvivalentne nul-znakovima, tj. takav niz znakova je sigurno string.

Prilikom inicijalizacije, dimenzija polja ne mora biti zadana.

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



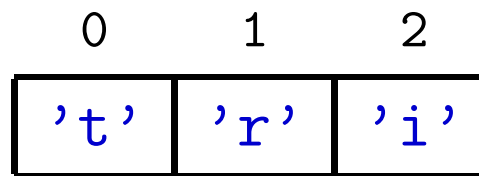
# Inicijalizacija polja znakova popisom

Primjer. Sljedeće dvije definicije su ekvivalentne:

```
char niz[3] = {'t', 'r', 'i'};  
char niz[] = {'t', 'r', 'i'};
```

Obje definicije

- kreiraju polje `niz` od 3 znaka (`sizeof(niz) = 3`),
- i inicijaliziraju ga,



Taj niz znakova nije string, jer ne sadrži nul-znak!

Ponašanje funkcija za stringove na ovom nizu nije definirano.

# Inicijalizacija polja znakova popisom (nastavak)

Primjer. Sljedeća definicija

```
char niz[5] = {'t', 'r', 'i'};
```

radi slično:

- kreira polje `niz` od 5 znakova (`sizeof(niz) = 5`),
- i inicijalizira ga — uz dopunu nulama,

0	1	2	3	4
't'	'r'	'i'	'\0'	'\0'

Taj niz znakova je string, jer sadrži nul-znak!

Duljina tog niza kao stringa je 3 (`strlen(niz) = 3`).

# Inicijalizacija polja znakova stringom

Polje znakova može se inicijalizirati i konstantnim stringom, a ne samo popisom znakova.

Primjer. Definicija

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

- kreira polje `s` od 4 znaka (`sizeof(s) = 4`),
- i inicijalizira ga, znak po znak, navedenim stringom,

0	1	2	3
't'	'r'	'i'	'\0'

Kod inicijalizacije stringom, dobiveni niz znakova je uvijek string, jer sadrži nul-znak!

Duljina niza `s` kao stringa je 3 (`strlen(s) = 3`).

# Inicijalizacija polja znakova stringom (nastavak)

Primjer. Dozvoljeno je i ovo:

```
char tekst[10] = "Init";
```

- To **kreira** polje **tekst** od **10** znakova,
- i **inicijalizira** ga, znak po znak, navedenim **stringom**, a ostatak dopunjava **nulama**,

0	1	2	3	4	5	6	7	8	9
'I'	'n'	'i'	't'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'

Ovdje je:

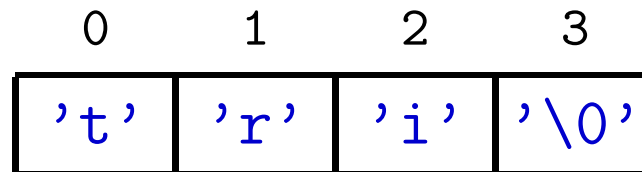
- `sizeof(tekst) = 10,`
- `strlen(tekst) = 4.`

# Inicijalizacija polja znakova — komentar

Svejedno je da li neko polje znakova inicijaliziramo popisom znakova ili konstantnim stringom, ako dobivamo isto polje (element po element).

Primjer. Sljedeće četiri definicije su ekvivalentne:

```
char s[] = "tri";  
char s[4] = "tri";  
char s[] = {'t', 'r', 'i', '\0'};  
char s[4] = {'t', 'r', 'i', '\0'};
```



# Inicijalizacija polja i pridruživanje

Napomena. Inicijalizacija polja znakova popisom znakova, poput

```
char s[] = {'t', 'r', 'i', '\0'};
```

ekvivalentna je (bar za automatsko polje)

👉 pridruživanju “element po element”:

```
char s[4];  
s[0] = 't';  
s[1] = 'r';  
s[2] = 'i';  
s[3] = '\0';
```

# Implementacija nekih funkcija za stringove

## Primjer — implementacije strlen

Primjer. Nekoliko mogućih **implementacija** funkcije **strlen** iz `<string.h>`.

Varijanta s **indeksima** — koristimo **brojač** pomaka do prvog nul-znaka (kraj stringa).

---

```
int strlen(const char *s)
{
    int n;

    for (n = 0; s[n] != '\0'; ++n);
    /* for je prazan! */
    return n;
}
```

---



## Primjer — implementacije strlen (nastavak)

Varijanta s pokazivačima — opet koristimo brojač pomaka.

```
int strlen(const char *s)
{
    int n = 0;

    while (*s++ != '\0')    /* (*s++) */
        ++n;
    return n;    /* Na sto pokazuje s na kraju? */
}
```

Uočimo da je `*s++ != '\0'` uspoređivanje izraza s nulom, tj. različito od nule  $\iff$  istina, pa možemo pisati samo `*s++`.

Ovo više nije jako čitljivo — bolje izbjegavati (iako se često koristi).

## Primjer — implementacije strlen (nastavak)

Varijanta s pokazivačima — koristimo razliku pokazivača.

---

```
int strlen(const char *s)
{
    char *p = s;

    while (*p != '\0')    /* (*p) */
        ++p;
    return p - s;
}
```

---

Umjesto `while`, može i `for (p = s; *p; ++p);`.

Pitanje: Rade li naredbe

🚫 `while (*p++ != '\0');` ili `while (*p++);`? (Ne, +1).

## Primjer — implementacije strcpy

Primjer. Nekoliko mogućih **implementacija** funkcije **strcpy**.

Varijanta s **indeksima**:

- Funkcija **kopira** polje znakova na koje pokazuje **t**, u polje na koje pokazuje **s**.
- Kopiranje se **zaustavlja** kad se **iskopira** nul-znak **'\0'**.

---

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

---

**Napomena.** “Prava” funkcija **strcpy** još vrati **s**.

## Primjer — implementacije strcpy (nastavak)

Varijanta s pokazivačima:

- Inkrementira pokazivače umjesto indeksa.
- Operator dereferenciranja \* ima viši prioritet od operatora pridruživanja = (ne trebaju zagrade).

---

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

---

## Primjer — implementacije strcpy (nastavak)

Kraća varijanta:

- Unarni operatori imaju asocijativnost  $D \rightarrow L$ .
- Koristimo `*s++` i `*t++`, tako da se pokazivači `s` i `t` povećavaju **nakon** pridruživanja.

---

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

---

Pitanje: Na što pokazuju `s` i `t` **nakon while** petlje?

## Primjer — implementacije strcpy (nastavak)

Još kraća varijanta:

- Uočimo da je `(*s++ = *t++) != '\0'` uspoređivanje izraza (pridruživanja) s nulom,
- tj. **različito** od **nule**  $\iff$  **istina**, pa možemo pisati samo `*s++ = *t++`.

---

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

---

Ovo više **nije** jako **čitljivo** — bolje izbjegavati (iako se često koristi).

## Primjer — implementacije strcat

Primjer. Moguća **implementacija** funkcije **strcat** preko **pokazivača** — vrlo “nečitljivo” napisana.

---

```
char *strcat(char *dest, const char *src)
{
    char *p = dest;

    while (*p++);    /* Pomak do IZA '\0' u dest. */
    --p;            /* Natrag na zadnji '\0'. */
    while (*p++ = *src++); /* Kopiraj src u dest. */

    return dest;
}
```

---

**Zadatak.** Napišite “čitljivije” (v. sljedeća stranica)!

## Primjer — implementacije strcat (nastavak)

Čitljivija varijanta s poljima i indeksima:

- Prvo pomaknemo indeks *i* na nul-znak u polju *s*.
- Od tog mjesta u *s*, iskopiramo cijeli *t* (slično `strcpy`).

---

```
char *strcat(char s[], const char t[])
{
    int i = 0, j = 0;

    while(s[i] != '\0') ++i;
    while((s[i++] = t[j++]) != '\0');

    return s;
}
```

---