

# *Programiranje 2*

## *3. predavanje*

Saša Singer

`singer@math.hr`

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

PMF – Matematički odsjek, Zagreb

# Sadržaj predavanja

- **Struktura programa** (kraj):
  - Blokovska struktura jezika (prošli puta).
  - Doseg varijable — lokalne i globalne varijable.
  - Vijek trajanja varijable, memorijske klase.
  - Program smješten u više datoteka. Vanjski simboli.
  - Primjer — operacije s vektorima u  $\mathbb{R}^3$ .
- **Višedimenzionalna polja** (prvi dio):
  - Deklaracija višedimenzionalnog polja.
  - Spremanje višedimenzionalnog polja u memoriji.
  - Inicijalizacija višedimenzionalnog polja.
  - Višedimenzionalno polje kao argument funkcije.

# Informacije — predavanja

Trenutno nema konkretnih informacija.

- Pratite web stranice fakulteta!
- Za “nastavu na daljinu” pratite web stranice kolegija.

# Struktura programa (nastavak)

# Sadržaj

- **Struktura programa** (kraj):
  - Blokovska struktura jezika (prošli puta).
  - Doseg varijable — lokalne i globalne varijable.
  - Vijek trajanja varijable, memorijske klase.
  - Program smješten u više datoteka. Vanjski simboli.

# Atributi varijable — ponavljanje

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

Sve varijable 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, postoje 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 varijable — ponavljanje (nastavak)

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

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

“Upravljanje” **vijekom trajanja** (a, ponekad, i **dosegom**) vrši se tzv. **identifikatorima memorijske klase** (ključne riječi **auto**, **extern**, **register** i **static**) i to kod **deklaracije** objekta.

# Automatske varijable

Automatska varijabla je

- svaka **varijabla** kreirana **unutar** nekog bloka (dakle, **unutar** neke funkcije),
- koja **nema** ključnu riječ **static** u deklaraciji.

Automatske varijable

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

**Memorija** koju su **zauzimale oslobađa se** tada za druge varijable (vrijednosti su izgubljene).

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



# Automatske varijable (nastavak)

Primjer:

---

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

---

Automatske varijable su: **x** (formalni argument) i **y**.

Varijabla **z** nije automatska, nego **statička** — jer ima **static**.

Automatske varijable **mog**u se **inicijalizirati** (kao što je to slučaj s varijablom **y**).

# Automatske varijable — inicijalizacija

Inicijalizacija se vrši (ako je **ima**)

- prilikom **svakog novog ulaza** u blok u kojem je varijabla definirana (“rezerviraj memoriju i inicijaliziraj”).

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 (v. prošli primjer), ali i
- izrazom koji **nije konstantan**, kao u sljedećem primjeru.

# Automatske varijable — inicijalizacija (nast.)

Primjer:

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

U trenutku inicijalizacije

• varijable **x** i **n** već **imaju** neku vrijednost.

# Identifikatori memorijske klase

Identifikatori memorijske klase (engl. “storage class”) su

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

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

Oni (osim `extern`) služe

• `preciziranju` vijeka trajanja varijable,

tj. u kojem dijelu `memorije` se nalazi varijabla.

Identifikator `extern` služi za `deklaraciju` objekata koji se nalaze u `drugim` datotekama (v. kasnije).

Ključna riječ `static` služi još i za kontrolu `dosega` varijabli i funkcija (svrha ovisi o tome gdje se `static` napiše, v. kasnije).

# Identifikatori memorijske klase — nastavak

Pišu 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 ... ;
```

---

Primjer:

---

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

---

# Identifikator `auto`

Identifikator `auto` deklarira **automatsku** varijablu. Međutim,

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

Zato se riječ `auto` obično **ne koristi** u programima (ali smije).

**Primjer.** Ekvivalentno je napisati

---

<pre>{     char c;     int i, j, k;     ... }</pre>	<pre>{     auto char c;     auto int i, j, k;     ... }</pre>
---	---

---

# Identifikator register

Identifikator memorijske klase **register** sugerira prevoditelju

- da **varijablu** smjesti u **registar** procesora, a ne u “običnu” memoriju (ako ide).
- Napomena: prevoditelj to **ne mora** “poslušati”.

Ideja: dobiti na **brzini** izvođenja programa, tako da se **smanji** vrijeme pristupa do te **varijable**.

Oznaka **register** obično se koristi za

- **najčešće** korištene **varijable**, poput **kontrolne varijable** petlje ili **brojača**.

Može se primijeniti **samo** na **automatske** varijable (unutar nekog bloka), a **ne smije** se primijeniti na globalne (ili statičke) varijable.

# Identifikator register (*nastavak*)

Primjer:

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

Zabranjeno je

- primijeniti adresni operator & na register varijablu,
- koristiti pokazivač na takvu varijablu.

Naime, registri **nemaju** “standardne” adrese.

**Savjet:** Pustiti stvar **optimizaciji** prevoditelja — ona bira što će staviti u registre i pazi na pripadne vrijednosti u memoriji.



# Statičke varijable

Statička 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**. Ovdje, **nula** znači “svi bitovi su **nula**”.

## 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 — primjer

**Primjer.** Statička varijabla deklarirana **unutar** funkcije.

---

```
void foo() {
    int a = 10;    static int sa = 10;

    a += 5;    sa += 5;
    printf("a = %d, sa = %d\n", a, sa);
    return;
}
```

---

Varijabla **a** je **automatska** i inicijalizira se prilikom **svakog** ulaska u funkciju.

Varijabla **sa** je **statička** i inicijalizira se samo prilikom **prvog** ulaska u funkciju. **Zadržava** vrijednost kod izlaska iz funkcije.

## Statičke varijable — primjer (nastavak)

Glavni program 3 puta poziva funkciju `foo` (v. `static.c`).

---

```
int main(void) {
    int i;
    for (i = 0; i < 3; ++i)
        foo();
    return 0;
}
```

---

Izlaz je:

---

```
a = 15, sa = 15
a = 15, sa = 20
a = 15, sa = 25
```

---

## Statičke varijable (nastavak)

Primjer za statičke varijable deklarirane **unutar** nekog bloka je

- varijanta funkcije za **Fibonacci**jeve brojeve sa **statičkim** poljem (vježbe, Primjer 2.4.2., dno str. 33).

Ovo je **loš** primjer, jer polje **nije** potrebno!

Puno **bolji** primjer, istog tipa, je

- računanje broja **particija** **Eulerovom** formulom, gdje **statičko** polje **ima** smisla, kao zamjena za rekurziju (v. **euler\_ps.c**).

Jedina svrha “lokalizacije” (globalno  $\mapsto$  lokalno **static**) je

- **onemogućiti** promjene **čuvanih** vrijednosti **izvan** funkcije.

# 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** takvu globalnu varijablu i **promijeniti** njezinu vrijednost.

Na taj način, **više funkcija** može **komunicirati bez upotrebe** formalnih argumenta. Na primjer, brojač poziva u **fib\_r**.

## Broj particija — globalni brojač

Primjer. Brojanje **particija** smo prije realizirali

- funkcijom koja vraća **int** = broj particija.

Istu stvar možemo napraviti

- **void** funkcijom koja koristi **globalni brojač** particija.

Svaki put kad **nađemo** novu particiju, **povećamo** brojač za **1** (v. **parts\_2.c**).

---

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

```
int broj = 0; /* globalni brojac */
```

---

Napomena: **inicijalizacija** na **0** bi se napravila i **bez** da smo to **eksplicitno** napisali (**statička** varijabla).



## Broj particija — funkcija

```
void particije(int suma, int prvi)
{
    int i;

    if (suma == 0)
        ++broj;
    else
        for (i = prvi; i <= suma; ++i)
            /* Sljedeci pribrojnik je i,
               rekurzivni poziv za suma - i. */
            particije(suma - i, i);
    return;
}
```

## Broj particija — glavni program

```
int main(void)
{
    int n;

    printf(" Upisi prirodni broj n: ");
    scanf("%d", &n);

    particije(n, 1);
    printf("\n Broj particija p(%d) = %d\n",
           n, broj);

    return 0;
}
```

## 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;                /* static */
void f(int);
int main(void) {
    int c, d;         /* auto */
    ...
}
int b;                /* static */
void f(int i) {
    int x, y;         /* auto */
    ...
}
```

---

# Program smješten u više datoteka

C program može biti smješten u više datoteka.

- Na 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, uz uvjet da su korektno deklarirane u toj drugoj datoteci.

- Za deklaraciju objekta koji je definiran u nekoj drugoj datoteci koristimo ključnu riječ `extern` (“vanjski”).

Riječ `extern` ispred deklaracije objekta (varijable ili funkcije) informira prevoditelj da se radi o objektu koji je definiran u nekoj drugoj datoteci.

# 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**, kao **extern**.

Sadržaj **datoteke 2**:

---

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

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

---

Nakon **prevođenja obje** datoteke i **povezivanja** (linker), izvršavanje daje rezultat:

---

```
i = 5
```

---

# Vanjski simboli

U programu smještenom u više datoteka

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

Stoga kažemo da su imena funkcija i globalnih varijabli vanjski simboli (tzv. “javni” ili “public” simboli).

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

Kada funkciju ili globalnu varijablu deklariramo kao `static`,

- ona prestaje biti vanjski simbol i može se dohvatiti samo iz datoteke u kojoj je definirana (tzv. “private” simbol).

Ovdje `static` služi za ograničavanje dosega.

## Vanjski simboli (nastavak)

**Primjer.** Želimo **onemogućiti** korištenje funkcije **g** **izvan** prve datoteke — tj. hoćemo da je **g** “private”, a ne “public” simbol.

Sadržaj datoteke 1:

---

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

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

int g(int i) {          /* Definicija funkcije g. */
    return 2 * i - 1;
}
```

---



## Vanjski simboli (nastavak)

Funkciju `g` više **ne možemo dohvatiti** iz druge datoteke, pa je sljedeći program **neispravan** — očekuje `g` kao vanjski simbol. Sadržaj **datoteke 2** (prevodi se **bez greške**):

```
#include <stdio.h>
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)

Potrebni 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);
    return 0;
}
```

Izvršavanjem dobivamo izlaz:

```
i = 3
```

# Definicija i deklaracija globalnih varijabli

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

- U **definiciji** varijable, deklarira se njezino **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** varijable je uvijek i njezina **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.

- U **definiciji**, **globalna** varijabla može biti **inicijalizirana konstantnim** izrazom.
- **Globalna** varijabla koja **nije eksplicitno** inicijalizirana, bit će **inicijalizirana nulom** (= svi bitovi su **nula**).

Dodatna pravila:

- U **deklaraciji** **globalne** varijable, **mora** se koristiti ključna riječ **extern**, a **inicijalizacija nije moguća**.
- U **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 varijable, s istim djelovanjem kao i za funkcije.

- `static` sužava doseg (područje vidljivosti) varijable na datoteku u kojoj je definirana. Ime takve varijable više nije dohvatljivo kao vanjski simbol (postaje “private”).

**Upozorenje:** Oznaka memorijske klase `static` ispred globalne i lokalne varijable 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 NE treba pisati! */
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));
    return 0;
}
```

---

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

# Datoteke zaglavlja (nastavak)

U datotekama zaglavlja \*.h

- **implicitno** se “dodaje” **extern** za sve **deklaracije**, tj. **podrazumijeva** se da su svi objekti **definirani** negdje drugdje. Zato **extern** tada ne treba pisati.

U Code::Blocks postoje **projekti**.

- Sve **potrebne** datoteke treba **dodati** u projekt.

## Primjer — vektori u prostoru

Primjer. Treba napisati “biblioteku” funkcija za operacije s vektorima u prostoru  $\mathbb{R}^3$ .

Vektor  $\vec{a} \in \mathbb{R}^3$ , oblika

$$\vec{a} = a_x \vec{i} + a_y \vec{j} + a_z \vec{k},$$

implementiramo poljem  $\mathbf{a}$  s 3 elementa, koje (redom) sadrži prostorne koordinate  $a_x, a_y, a_z$ .

Našu “biblioteku” funkcija realiziramo tako da se lako koristi u raznim programima:

- deklaracije svih funkcija stavljamo u datoteku zaglavlja `vekt.h`, a
- definicije svih funkcija (zajedno s tijelima) pišemo u “programskoj” datoteci `vekt.c`.

# Primjer — vektori u prostoru (nastavak)

Funkcije i operacije u biblioteci:

● norma vektora:  $\|\vec{a}\| = \sqrt{a_x^2 + a_y^2 + a_z^2}$ ,

● skalarni produkt dva vektora:  $\vec{a} \cdot \vec{b}$ ,

● kosinus kuta između dva vektora:

$$\cos \angle(\vec{a}, \vec{b}) = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\| \cdot \|\vec{b}\|}.$$

● vektorski produkt dva vektora:  $\vec{c} = \vec{a} \times \vec{b}$ ,

● zbroj dva vektora:  $\vec{c} = \vec{a} + \vec{b}$ ,

● množenje vektora skalarom:  $\vec{b} = \lambda \vec{a}$ ,

● ispis vektora u obliku:  $(a_x, a_y, a_z)$ .

## Primjer — vektori u prostoru (nastavak)

Datoteka `zaglavlja vekt.h` sadrži deklaracije svih funkcija:

---

```
/* Datoteka zaglavlja vekt.h:  
   - deklaracije za operacije s vektorima u  $\mathbb{R}^3$ .  
   Implementacija je u datoteci vekt.c.  
*/
```

```
double norma(double a[]);  
double skal_prod(double a[], double b[]);  
double kosinus_kuta(double a[], double b[]);  
void vekt_prod(double a[], double b[], double c[]);  
void zbroj(double a[], double b[], double c[]);  
void skal(double a[], double lambda, double b[]);  
void vekt_print(double a[]);
```

---

## Primjer — vektori u prostoru (nastavak)

Datoteka `vekt.c` sadrži definicije svih funkcija:

---

```
#include <stdio.h>
#include <math.h>
#include "vekt.h"
```

```
/* Implementacija operacija s vektorima u  $\mathbb{R}^3$ . */
```

## *Primjer — vektori u prostoru (nastavak)*

```
double norma(double a[])
{
    int i;
    double suma = 0.0;

    for (i = 0; i < 3; ++i)
        suma = suma + a[i] * a[i];

    return sqrt(suma);
}
```

## *Primjer — vektori u prostoru (nastavak)*

```
double skal_prod(double a[], double b[])
{
    int i;
    double suma = 0.0;

    for (i = 0; i < 3; ++i)
        suma = suma + a[i] * b[i];

    return suma;
}
```



## Primjer — vektori u prostoru (nastavak)

```
double kosinus_kuta(double a[], double b[])  
{  
    return skal_prod(a, b) / (norma(a) * norma(b));  
}
```

```
void vekt_prod(double a[], double b[], double c[])  
{  
    c[0] = a[1] * b[2] - a[2] * b[1];  
    c[1] = a[2] * b[0] - a[0] * b[2];  
    c[2] = a[0] * b[1] - a[1] * b[0];  
  
    return;  
}
```

## *Primjer — vektori u prostoru (nastavak)*

```
void zbroj(double a[], double b[], double c[])
{
    int i;

    for (i = 0; i < 3; ++i)
        c[i] = a[i] + b[i];

    return;
}
```

## *Primjer — vektori u prostoru (nastavak)*

```
void skal(double a[], double lambda, double b[])
{
    int i;

    for (i = 0; i < 3; ++i)
        b[i] = lambda * a[i];

    return;
}
```

## *Primjer — vektori u prostoru (nastavak)*

```
void vekt_print(double a[])
{
    printf("(%g, %g, %g)\n", a[0], a[1], a[2]);

    return;
}
```

---

# Primjer — vektori u prostoru (nastavak)

Primjer. Glavni program (v. vektori.c)

---

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

/* Primjer za operacije s vektorima u R^3. */
int main(void)
{
    double a[3] = { 1.0, 2.0, 0.0 };
    double b[3] = { 0.0, 1.0, 1.0 };
    double c[3];

    printf(" a = "); vekt_print(a);
    printf(" b = "); vekt_print(b);
}
```

## Primjer — vektori u prostoru (nastavak)

```
printf("  norma(a) = %g\n", norma(a));  
printf("  norma(b) = %g\n", norma(b));  
  
printf("      a * b = %g\n", skal_prod(a, b));  
printf("  cos(a, b) = %g\n", kosinus_kuta(a, b));  
  
zbroj(a, b, c);  
printf(" a + b = "); vekt_print(c);  
  
vekt_prod(a, b, c);  
printf(" a x b = "); vekt_print(c);  
  
return 0;  
}
```

## Primjer — vektori u prostoru (nastavak)

Izlaz programa (v. vektori.out):

---

$$a = (1, 2, 0)$$

$$b = (0, 1, 1)$$

$$\text{norma}(a) = 2.23607$$

$$\text{norma}(b) = 1.41421$$

$$a * b = 2$$

$$\cos(a, b) = 0.632456$$

$$a + b = (1, 3, 1)$$

$$a \times b = (2, -1, 1)$$

---

**Zadatak.** Dodajte ovoj biblioteci funkciju za

• **mješoviti** produkt tri vektora:  $\vec{a} \cdot (\vec{b} \times \vec{c})$ .

# Višedimenzionalna polja



# Sadržaj

- Višedimenzionalna polja (prvi dio):
  - Deklaracija višedimenzionalnog polja.
  - Spremanje višedimenzionalnog polja u memoriji.
  - Inicijalizacija višedimenzionalnog polja.
  - Višedimenzionalno polje kao argument funkcije.

# 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  $L \rightarrow D$ .

# Jednodimenzionalna i višedimenzionalna polja

Niz podataka nekog tipa odgovara

- jednodimenzionalnom polju podataka tog tipa u C-u.

Do sada smo koristili samo nizove sastavljene iz podataka jednostavnog tipa (standardni tipovi u C-u).

Odgovarajući matematički pojam je

- vektor = uređeni niz skalara (određenog tipa).

U C-u možemo koristiti i višedimenzionalna polja. Na primjer, matematičkom pojmu matrice

- odgovara dvodimenzionalno polje u C-u.

Kako se višedimenzionalna polja realiziraju i interpretiraju u C-u?

# Višedimenzionalna polja

Dvodimenzionalno polje je jednodimenzionalno polje, čiji elementi su

- jednodimenzionalna polja (istog tipa), a ne skalari.

To znači da **matricu** (iz **matematike**) ovdje gledamo kao **vektor**, čiji elementi su **vektori**.

Analogno, **trodimenzionalno** polje je **jednodimenzionalno** polje, čiji elementi su

- **dvodimenzionalna** polja (istog tipa).

Ovu strukturu možemo **matematički** gledati kao **vektor**, čiji elementi su **matrice**.

I tako redom, za više dimenzije.

# Višedimenzionalna polja — deklaracija

Deklaracija **višedimenzionalnog polja** ima oblik:

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

**Primjer.** Polje **m**, deklarirano na sljedeći način

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

predstavlja **matricu** s **dva retka** i **tri stupca**.

Njezine elemente možemo “prostorno” zamisliti u obliku:

```
m[0][0]   m[0][1]   m[0][2]
m[1][0]   m[1][1]   m[1][2]
```

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** (najdesniji) 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]**, istog tipa **double[3]**.

## Višedimenzionalna polja (nastavak)

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

Kad “raspakiramo” ta **dva** polja u elemente tipa **double**, 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

$$i * MAX\_j + j,$$

gdje je  $MAX\_j = 3$  broj **stupaca** matrice iz deklaracije polja  $m$ .

**Prva** dimenzija polja ( $MAX\_i = 2$ ), tj. broj **redaka** matrice  $m$ , **ne treba** za indeksiranje, već samo za **rezervaciju** memorije.



# Višedimenzionalna polja (nastavak)

Primjer. Trodimenzionalno polje

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

je **jednodimenzionalno** polje s 2 elementa `MM[0]` i `MM[1]`.

● Ti elementi su **dvodimenzionalna** polja tipa `float[3][4]`.

Element `MM[i][j][k]` **smješten** je na mjesto

$$(i * \text{MAX}_j + j) * \text{MAX}_k + k,$$

gdje su `MAX_i = 2`, `MAX_j = 3` i `MAX_k = 4` **dimenzije** polja.

**Prva** dimenzija polja (`MAX_i = 2`) nije nužna za **indeksiranje**, već za **rezervaciju** memorije.

# Pristup elementima polja

Pojedinim **elementima** polja možemo pristupiti na **dva** načina:

- navođenjem pripadnog **indeksa** za svaku **dimenziju** polja,
- preko **pokazivača**, koristeći ekvivalenciju između **polja** i **pokazivača** (o tome — malo kasnije).

**Pristup** elementima polja preko **indeksa**:

- Svaki indeks mora biti u **svojim** uglatim zagradama [ ].

**Nije** dozvoljeno **odvajanje** indeksa **zarezom**, kao u nekim drugim jezicima (Fortran, Pascal)!

**Primjer.** U prethodnim primjerima, elemente smo naveli kao

- `m[i][j]`, odnosno, `MM[i][j][k]`.

**Nije** dozvoljeno napisati `m[i, j]` ili `MM[i, j, k]`.

## Inicijalizacija polja

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

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

Inicijalne vrijednosti pridružuju se elementima matrice **onim redom** kojim su elementi **smješteni** u memoriji (tj. po recima):

```
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.
```

Ovakav način inicijalizacije postaje **nepregledan** za veća polja. Zamislite matricu `A[100][100]` i popis od **10000** vrijednosti.

# Inicijalizacija polja (nastavak)

Zato se **inicijalne** vrijednosti mogu

- **vitičastim** zagradama **grupirati** u **grupe**, koje se **pridružuju** pojedinim **recima**.

Ekvivalentna inicijalizacija:

---

```
static double 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** (i to **bez obzira** na **static**).

- Čim **postoji** inicijalizacija, inicijalizira se **cijelo** polje.

## 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'} }  
};
```

Kao rezultat dobivamo dvije matrice znakova  $A[0]$  i  $A[1]$ , tipa  $\text{char}[2][2]$ :

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

# 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 `MAX_m` redaka i `MAX_n` stupaca može biti deklarirana ovako:

---

```
int mat[MAX_m][MAX_n];  
...  
void readmat(int mat[MAX_m][MAX_n],  
             int m, int n)
```

---

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

# Višedimenzionalno polje kao argument funkcije

Drugi način deklaracije, bez prve dimenzije:

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

Prva dimenzija `MAX_m` = broj redaka matrice — nije bitan za adresiranje elemenata matrice u funkciji.

Princip je isti kao za jednodimenzionalna polja.

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

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

Ovdje je `mat` = pokazivač na redak, tj. na polje od `MAX_n` elemenata tipa `int`, odnosno, `mat` = polje redaka matrice.

## Razlika — matrica i niz nizova

Napomena. Takozvani “četvrti” način (v. sljedeće poglavlje)

```
void readmat(int **mat, int m, int n)
```

nije korektan. Naime, deklaracija `int **mat` kaže da je

• `mat` = pokazivač na pokazivač na `int`,

pa `mat` možemo interpretirati kao

• polje pokazivača na `int`, odnosno, kao niz nizova.

Međutim, to nije matrica, jer ovi drugi nizovi

• nemaju fiksnu duljinu (za razliku od redaka), već njihova duljina smije varirati (v. “argumenti komandne linije”),

• i još ne moraju biti spremljeni “u bloku”, a matrica mora biti u “u bloku” — redak za retkom.



## Primjer funkcije — Euklidska norma matrice

**Primjer.** Napišimo funkcija koja računa **Euklidsku** (ili **Frobeniusovu**) normu matrice.

Neka je  $A$  matrica tipa  $m \times n$  s elementima

$$a_{i,j} \text{ (ili, skraćeno) } a_{ij}, \quad i = 1, \dots, m, \quad j = 1, \dots, n.$$

**Euklidska** ili **Frobeniusova** norma matrice  $A$  definira se ovako

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

To je isto kao da **sve** elemente matrice  $A$  posložimo u jedan **vektor** i onda izračunamo **Euklidsku** normu tog vektora.

## Funkcija E\_norma

Primjer. Prva varijanta s “normalnim” poretkom petlji:

```
double E_norma(double a[10][10], int m, int n) {
    int i, j;
    double suma = 0.0;
    for (i = 0; i < m; ++i)
        for (j = 0; j < n; ++j)
            suma += a[i][j] * a[i][j];
    return sqrt(suma);
}
```

Poziv:

```
printf("%g\n", E_norma(a, 3, 5));
```

## Funkcija E\_norma — obratne petlje

Primjer. Ista funkcija s **obratnim** poretком petlji:

---

```
double E_norma(double a[10][10], int m, int n) {
    int i, j;
    double suma = 0.0;
    for (j = 0; j < n; ++j)
        for (i = 0; i < m; ++i)
            suma += a[i][j] * a[i][j];
    return sqrt(suma);
}
```

---

Pitanje: Koja od ove dvije funkcije je **bolja** (**brža**) — bar za **velike** matrice?

Uputa: Brži je **sekvencijalni** prolaz po memoriji.