

# *Programiranje 2*

## *4. predavanje*

Saša Singer

[singer@math.hr](mailto:singer@math.hr)

[web.math.hr/~singer](http://web.math.hr/~singer)

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

# **Informacije**

Konzultacije (zasad):

- srijeda, 10–11 sati, petak, 19–20 sati.

Praktični kolokvij — prvi krug kreće sljedeći tjedan!

- 30. 3. — 4. 4.,
- subota je namjerno uključena.

Popis s terminima je na zidu, desno od moje sobe.

- Zapišite (prijavite) se što prije — ako već niste!

# 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 variable — ponavljanje**

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 variable — ponavljanje (nastavak)**

- Prema **tipu** imamo variable tipa
  - **int, float, char**, itd.
- Prema **dosegu** variable 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** neke 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

- kreiraju** se na **ulasku** u blok u kome su deklarirane
- i uništavaju** se na **izlasku** iz 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 **mogu** se **inicijalizirati** (kao što je to slučaj s varijablom **y**).

# **Automatske varijable — inicijalizacija**

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

- pri **svakom novom ulazu** 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 memorejske klase*

Identifikatori memorejske 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).

Neki (poput `static`) služe i za kontrolu **dosega** varijabli i funkcija.

## *Identifikatori memoriske 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 **variable** deklarirane **unutar nekog** bloka **implicitno** su **automatske** (ako nisu deklarirane kao **static**),
- a sve **variable** deklarirane **izvan svih** blokova **implicitno** su **statičke**.

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

**Primjer.** Ekvivalentno je napisati

---

{ char c; int i, j, k; ... }	{ auto char c; auto int i, j, k; ... }
--	--

---

## **Identifikator register**

Identifikator memorijske klase **register** sugerira prevoditelju

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

Ideja: dobiti na **brzini** izvođenja porograma, 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.

Registri **nemaju** “standardne” adrese.

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

# Statičke 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 za statičke varijable deklarirane **unutar** nekog bloka je

- varijanta funkcije za **Fibonaccijeve** brojeve (na vježbama).

Napomena: nije lako naći pravi **primjer** ovog tipa.

# 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.

# *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.

---

```
#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)
            /* Rekursivni poziv -
               dodavanje sljedeceg pribrojnika */
            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, ako su tamo deklarirane.

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

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.

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 njeno **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 se smije ispustiti */
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.

## *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 DEVCPP postoje projekti.

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

## *Primjer — za vježbu*

Odredite **dosege** objekata u sljedeće dvije datoteke.

**datoteka\_1.c**

```
A()  
{  
    int x;  
}  
int y;  
/* definicija */  
B()  
{  
    static int w;  
}  
C() { ... }
```

**datoteka\_2.c**

```
D() { ... }  
extern int y;  
/* deklaracija */  
E() { ... }  
F()  
{  
    int v;  
}
```

## **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 **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 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:  
 - operacije s vektorima u 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 <math.h>
#include "vekt.h"

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

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)
norma(a) = 2.23607
norma(b) = 1.41421
a * b = 2
cos(a, b) = 0.632456
a + b = (1, 3, 1)
a x b = (2, -1, 1)
```

---

# 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 s

---

```
static double 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]**, 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

$$i * \text{MAX\_j} + j,$$

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

Prva dimenzija polja (**MAX\_i = 2**), tj. broj **redaka** matrice, **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]** bit će smješten na mjesto

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

gdje su **MAX\_i = 2**, **MAX\_j = 3** i **MAX\_k = 4** dimenzijske vrijednosti.

Prva dimenzija polja (**MAX\_i = 2**) nije nužna za indeksiranje, već za rezervaciju memorije.

## 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 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 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**).

## 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 **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 dimenzijske:

---

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

---

Prva dimenzija **MAX\_m**, tj. 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)
```

---

# Dvodimenzionalna polja — ponavljanje (1)

Ponavljanje par osnovnih stvari vezanih uz višedimenzionalna polja u C-u.

Primjer. Krenimo od sljedeće deklaracije

---

```
int M[4][3] = { {10, 5, 3},  
                {9, 18, 0},  
                {32, 20, 10},  
                {1, 0, 8} };
```

---

Ovdje je **M** matrica s **4** retka i **3** stupca, s elementima tipa **int**.

Stvarno, **M** je jednodimenzionalno polje koje sadrži **4** elementa:

- **M[0]**, **M[1]**, **M[2]**, **M[3]**, istog tipa **int[3]**.

## Dvodimenzionalna polja — ponavljanje (2)

Poredak elemenata polja  $M$  u memoriji:

10	5	3
9	18	0
32	20	10
1	0	8



10	0
5	1
3	2
9	3
18	4
:	:
8	11

$$M[1][1] = 18 \implies 1 \cdot 3 + 1 = 4$$

$M[i][j] \rightarrow$  pozicija u memoriji =  $i \cdot$  broj stupaca +  $j$ .

Pozicija u memoriji **ne** ovisi o broju redaka.

# Dvodimenzionalna polja — inicijalizacija

```
int M[4][3] = { {10, 5, 3},  
                {9, 18, 0},  
                {32, 20, 10},  
                {1, 0, 8} };  
  
M[2][1] = 20;
```

M	[0]	[1]	[2]
[0]	10	5	3
[1]	9	18	0
[2]	32	20	10
[3]	1	0	8

# Dvodimenzionalna polja — veličine objekata (1)

---

```
int M[4][3] = { {10, 5, 3},  
                {9, 18, 0},  
                {32, 20, 10},  
                {1, 0, 8} };
```

---

Primjer. Što ispisuju sljedeće naredbe?

---

```
printf("%u\n", sizeof(M));           /* %u, ne %d */  
printf("%u\n", sizeof(M[0]));  
printf("%u\n", sizeof(M[0][0]));
```

---

Napomena: `sizeof(polje)` vraća **duljinu** cijelog polja, tj. za polje od  $n$  elemenata,  $\text{sizeof} = n \times \text{sizeof(element)}$ .

## Dvodimenzionalna polja — veličine objekata (2)

Odgovor: Što su  $M$ ,  $M[0]$ ,  $M[0][0]$ ? Krenimo unatrag.

- $M[0][0]$  je “obični” int. Onda je
  - $\text{sizeof}(M[0][0]) = 4$ , jer int zauzima 4 bytea.
- $M[0]$  je jednodimenzionalno polje s 3 elementa tipa int, pa je
  - $\text{sizeof}(M[0]) = 3 \times \text{sizeof(int)} = 12$ .
- $M$  je jednodimenzionalno polje s 4 elementa tipa  $\text{int}[3]$ , pa je
  - $\text{sizeof}(M) = 4 \times \text{sizeof}(\text{int}[3]) = 4 \times 12 = 48$ .
- Ekvivalentno,  $M$  je dvodimenzionalno polje s  $4 \times 3 = 12$  elemenata tipa int, pa je
  - $\text{sizeof}(M) = 12 \times \text{sizeof(int)} = 12 \times 4 = 48$ .

## 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** poretkom 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?