# Programiranje 2

## Predavanje 04 - višedimenzionalna polja, pokazivači - prvi dio

Matej Mihelčić

Prirodoslovno-matematički fakultet
Matematički odsjek

20. ožujka 2024.

# Višedimenzionalna polja

**Niz podataka** nekog tipa se u C-u reprezentira jednodimenzionalnim poljem podataka tog tipa. Do sada smo koristili nizove podataka jednostavnog (standardnog) tipa. Vidjet ćemo da možemo imati polja i **složenog** tipa.

Odgovarajući matematički analogon C polja je vektor (uređeni niz skalara određenog tipa).

U C-u možemo reprezentirati i kompliciranije objekte. Npr. matrici odgovara dvodimenzionalno polje u C-u. **Dvodimenzionalno polje** je jednodimenzionalno polje čiji elementi su jednodimenzionalna polja (istog tipa), ne skalari. Matematički gledano, matricu promatramo kao vektor čiji elementi su vektori. Analogno, trodimenzionalno polje je jednodimenzionalno polje čiji elementi su dvodimenzionalna polja (istog tipa). Matematički gledano, vektor čiji elementi su matrice. Na isti način promatramo i polja viših dimenzija.

# Višedimenzionalna polja

Deklaracija višedimenzionalnog polja ima oblik:

```
1 mem_klasa tip ime[izraz_1]...[izraz_n];
2
3 /*dvodimenzionalno polje - matrica s 2 retka i
4 3 stupca*/
5 static double m[2][3];
```

Prostorno, elemente matrice `m` možemo zamisliti:

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

Navedeni elementi se u memoriji računala pamte **jedan za drugim - kao jednodimenzionalno polje**.

# Višedimenzionalna polja

Višedimenzionalno polje s deklaracijom:

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

je jednodimenzionalno polje duljine `izraz_1`, a elementi tog polja su polja dimenzije $n-1$, oblika:

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

Isto vrijedi za svaku dimenziju slijeva nadesno. Operator [] ima asocijativnost $L \rightarrow D$.

Na isti način se spremaju elementi polja u memoriju računala. Kod spremanja kao jednodimenzionalno polje najsporije se mijenja prvi indeks a najbrže zadnji indeks.

Npr. kod matrica, elementi su poredani po recima, prvo se spremaju elementi prvog retka, zatim drugog itd.

# Višedimenzionalna polja

Dvodimenzionalno polje `m[2][3]` iz prethodnog primjera se sastoji od dva retka `m[0]` i `m[1]` koja su tipa `double[3]`. Matrica `m` se u memoriju sprema kao:

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

prvo elementi retka `m[0]`, zatim elementi retka `m[1]`.

Linearno indeksiranje elementa se vrši računanjem `i*MAX_j+j`, gdje je `MAX_j = 3` (broj stupaca matrice iz deklaracije polja *m*. Dimenzija `MAX_i = 2` - broj redaka matrice *m* nije potreban za indeksiranje već samo za **rezervaciju memorije**.

Linearno indeksiranje se **ne vrši** u svim programskim jezicima na identičan način. Npr. u `Fortranu` se indeksiranje vrši *po stupcima*.

# Višedimenzionalna polja

**Trodimenzionalno polje** `float MM[2][3][4]` je jednodimenzionalno polje s dva elementa `MM[0]` i `MM[1]`. Oba elementa su dvodimenzionalna polje tipa `float[3][4]` - matrice s 3 retka i 4 stupca.

Element `MM[i][j][k]` smješten je na poziciju
`(i*MAX_j+j)*MAX_k+k`
`MAX_i = 2, MAX_j = 3, MAX_k = 4` su dimenzije polja. `MAX_i` je bitan jedino za rezervaciju memorije.

# Pristup elementima polja

Elementima polja možemo pristupiti na dva načina:

- Navođenjem pripadnog indeksa za svaku dimenziju polja.
- Preko pokazivača, koristeći ekvivalenciju između polja i pokazivača

Ukoliko koristimo pristup elementima preko indeksa, svaki indeks mora biti u svojim uglatim zagradama []. Nije dozvoljeno odvajanje indeksa zarezom (kao npr. u `Fortranu` i `Pascalu`).

Primjeri: `m[i][j]`, `MM[i][j][k]`.

Oznake `m[i,j]` ili `MM[i,j,k]` **nisu dozvoljene u C-u**.

# Inicijalizacija polja

Dvodimenzionalno polje *m* iz prethodnog primjera možemo inicijalizirati na sljedeći način:

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

Inicijalne vrijednosti pridružuju se elementima matrice redom kojim su elementi smješteni u memoriji (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. Npr. za matricu `A[1000][1000]` treba zapisati 1000000 vrijednosti.

# Atributi varijable

Varijabla je ime (sinonim) za sadržaj neke lokacije ili bloka lokacija u memoriji. Varijable imaju tri atributa: a) tip, b) doseg (engl. scope) i c) vijek trajanja (eng. lifetime).

- Tip - način interpretacije sadržaja bloka (piše/čita se u bitovima), uključuje i veličinu bloka.
- Vijek trajanja - u kojem dijelu memorije programa se rezervira taj blok. Postoje tri dijela memorije: statički, programski stog (eng. run-time stack) i programska hrpa (eng. heap).
- Doseg - u kojem dijelu programa je dio memorije dohvatljiv ili vidljiv za korištenje (čitanje, mijenjanje vrijednosti).

# Inicijalizacija polja

Inicijalne vrijednosti se mogu grupirati **vitičastim zagradama**. Rezultantne grupe se pridružuju pojedinim recima.

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

Grupiranje odgovara dimenzijama. U primjeru gore, svaka grupa predstavlja jedan redak matrice. Ukoliko je neka grupa kraća od odgovarajuće dimenzije, preostali elementi se inicijaliziraju nulama (bez obzira na `static`). **Ukoliko postoji inicijalizacija, uvijek se inicijalizira cijelo polje**.

# Inicijalizacija polja

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

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

Kao rezultat dobijemo dvije matrice znakova, `A[0]` i `A[1]` tipa `char[2][2]`.

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

# Višedimenzionalna polja kao argumenti funkcije

Višedimenzionalno polje kao formalni argument funkcije može se deklarirati navođenjem:

- **svih dimenzija polja**
- **svih dimenzija polja, osim prve**

Funkciju koja čita matricu s `MAX_m` redaka i `MAX_n` stupaca može biti deklarirana kao:

```
1 int mat[MAX_m][MAX_n];
2 ...
3 void readmat(int mat[MAX_m][MAX_n], int m, int n)
```

Argumenti *m* i *n* su **stvarni** broj redaka i stupaca matrice, koje **učitavamo**.

Deklaracija bez prve dimenzije:

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

# Višedimenzionalna polja kao argumenti funkcije

`MAX_m` (broj redaka matrice) nije bitan za adresiranje elemenata matrice u funkciji. Princip je isti kao za jednodimenzionalna polja.

Možemo koristiti i pokazivače:

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

`mat` je **pokazivač** na **redak**, **polje** od `MAX_n` elemenata tipa `int`. Matrica se reprezentira kao **polje redaka** matrice.

Postoji mogućnost reprezentacije i kao **niz nizova**. Međutim, taj generalniji način **ne prikazuje matricu**. Reci mogu biti različitih duljina. Također, reci nisu spremljeni u bloku već su razasuti po memoriji.

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

`mat` je pokazivač na pokazivač na `int`. Odnosno, možemo proslijediti polje pokazivača na `int`.

# Euklidska norma matrice

Računamo Euklidsku (Frobeniusovu) normu matrice.

Neka je $A$ matrica tipa $m \times n$ s elementima $a_{i,j},\ i = 1, \ldots, m,\ j = 1, \ldots, n$. **Euklidska** ili **Frobeniusova** norma matrice $A$ definira se kao:

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

Ekvivalentno računanju Euklidske norme vektora koji sadrži sve elemente matrice.

# Euklidska norma matrice

Navodimo dvije varijante funkcije za računanje Frobeniusove norme matrice.

```
1  double E_norma(double a[10][10], int m, int n) {
2  int i, j;
3  double suma = 0.0;
4  for (i = 0; i < m; ++i)
5      for (j = 0; j < n; ++j)
6          suma += a[i][j] * a[i][j];
7  return sqrt(suma);
8  }
```

Poziv:

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

# Euklidska norma matrice

Isto možemo izračunati i tako da okrenemo poredak petlji:

```
1  double E_norma(double a[10][10], int m, int n) {
2  int i, j;
3  double suma = 0.0;
4  for (j = 0; j < n; ++j)
5       for (i = 0; i < m; ++i)
6            suma += a[i][j] * a[i][j];
7  return sqrt(suma);
8  }
```

Koja od dvije funkcije je brža?

Brži je **sekvencijalni** prolaz po memoriji!

# Množenje matrice i vektora

Zadana je pravokutna matrica $A$ tipa $m \times n$, i vektor $x$ duljine $n$. Treba izračunati vektor $y = Ax$ (duljine $m$).

Formula za izračun elemenata vektora $y$ je:

$$y_i = \sum_{j=1}^{n} a_{i,j} \cdot x_j, \ \forall i = 1, \ldots, m$$

Za računanje moramo koristiti dvije petlje, jednu za iteriranje po indeksu $i$ i jednu za iteriranje po indeksu $j$. Treba imati na umu da petlje u C-u počinju od nule.

# Množenje matrice i vektora

```
1  void umnozak(int m, int n, int mat1[][MAX_n],
2  int mat2[MAX_n], int mat3[MAX_m])
3  {
4  int i, j;
5  /* Mnozenje matrice i vektora. */
6  for (i = 0; i < m; ++i) {
7          mat3[i] = 0;
8          for (j = 0; j < n; ++j)
9              mat3[i] += mat1[i][j] * mat2[j];
10 }
11 return;
12 }
```

Dio **glavnog** programa:

```
1  #include <stdio.h>
2  #define MAX_m 10
3  #define MAX_n 10
4  int main(void) {
5      int A[MAX_m][MAX_n], x[MAX_n], y[MAX_m];
6      int m, n; /* Stvarne dimenzije matrice A. */
7      //deklaracija matrice umnozak
8      void umnozak(int, int, int mat1[][MAX_n],
9                   int mat2[MAX_n], int mat3[MAX_m]);
10 ...
11 umnozak(m, n, A, x, y);
12 ... }
```

# Množenje matrica

Zadane su 3 pravokutne matrice:
$A$ dimenzija $m \times l$, $B$ dimenzija $l \times n$ i $C$ dimenzija $m \times n$.

Želimo izračunati izraz:

$$C = C + A \cdot B$$

Navedena operacija (*nazbrajavanje*) produkta $A \cdot B$ matrici $C$ je standardni oblik `BLAS-3` funkcije `xGEMM` za množenje matrica. Operacija se često koristi u praksi.

Matematički, operaciju po elementima realiziramo:

$$c_{i,j} = c_{i,j} + \sum_{k=1}^{l} a_{i,k} \cdot b_{k,j}, \ \forall i = 1, \ldots, m, \ j = 1, \ldots n$$

Potrebne su nam **tri** petlje, uz uvažavanje činjenice da u C-u indeksi kreću od nule.

# Množenje matrica

```
1  void matmul(int m, int n, int l, double A[][lda],
2  double B[][ldb], double C[][ldc] ) {
3  int i, j, k;
4
5  for (i = 0; i < m; ++i)
6      for (j = 0; j < n; ++j)
7          for (k = 0; k < l; ++k)
8              C[i][j] += A[i][k] * B[k][j];
9
10 return;
11 }
```

# Množenje matrica

Pošto se množenje matrica vrši u trostrukoj petlji, imamo $3! = 6$ verzija algoritma. Računala imaju hijerarhijski organiziranu memoriju, u kojoj se bliske memorijske lokacije mogu dohvatiti brže od udaljenih (blok-transfer u *cache* memoriju).

U prethodnom kodu, u unutarnjoj petlji (po $k$), računa se skalarni umnožak $i$-tog retka matrice $A$ i $j$-tog stupca matrice $B$. Elementi retka od $A$ su na susjednim lokacijama pa je dohvat brz. Međutim, elementi stupca od $B$ se nalaze na memorijskim lokacijama udaljenim za duljinu retka (`ldb`). Kod velikih matrica ta udaljenost može biti velika što dovodi do sporijeg izvršavanja.

# Množenje matrica

Efikasnija verzija algoritma koristi petlju po $j$ kao unutarnju.

```
1  ...
2  for (i = 0; i < m; ++i)
3       for (k = 0; k < l; ++k)
4            for (j = 0; j < n; ++j)
5                 C[i][j] += A[i][k] * B[k][j];
6  ...
```

U unutarnjoj petlji (po $j$), dohvaćaju se reci matrica $C$ i $B$, a nema dohvata stupaca. Element `A[i][k]` može se čuvati u cache memoriji. Elemente od $A$ isto dohvaćamo po recima.

Ovo je daleko najbrža od svih 6 varijanti algoritma za množenje kod velikih matrica.

# Množenje matrica

Ukoliko želimo računati samo umnožak $C = A \cdot B$, inicijaliziramo $C = 0$.

```
1  for (i = 0; i < m; ++i)
2       for (j = 0; j < n; ++j)
3            C[i][j] = 0.0;
4
5  for (i = 0; i < m; ++i)
6       for (k = 0; k < l; ++k)
7           for (j = 0; j < n; ++j)
8                C[i][j] += A[i][k] * B[k][j];
```

# Polja varijabilne duljine

Problem u C90 standardu je što dimenzije polja u deklaraciji argumenata funkcije moraju biti konstantni izrazi. Promjena dimenzije višedimenzionalnog polja čini nužnom promjenu deklaracije svih funkcija.

C99 uvodi polja *varijabilne duljine* koje neki prevoditelji još uvijek ne implementiraju u potpunosti. Polje varijabilne duljine je **automatsko polje** čije dimenzije mogu biti zadane vrijednostima varijabli.

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

# Polja varijabilne duljine

Korištenjem polja varijabilne duljine možemo napisati generalnije funkcije. **Treba paziti da dimenzije, osim prve predstavljaju dimenziju polja iz definicije**. U suprotnom će se memorija polja, unutar funkcije, interpretirati na krivi način.

```
1  double A[m][dim_n]; //deklaracija - main
2
3  //funkcija
4  double E_norma(int m, int n, double A[m][n]){
5  double suma = 0.0;
6  int i, j;
7  //A je niz redaka duljine n, ukoliko n!=dim_n
8  //pristupamo memoriji na krivi nacin
9  for (i = 0; i < m; ++i)
10       for (j = 0; j < n; ++j)
11            suma += A[i][j] * A[i][j];
12 return sqrt(suma); }
```

# Polja varijabilne duljine

Ukoliko želimo koristiti funkciju nad poljem fiksne duljine, ali koristiti samo dio alocirane memorije za računanje unutar funkcije, dodajemo još jedan parametar (stvarni broj stupaca iz definicije).

```
1 double E_norma(int m, int n, int lda,
2                                 double A[m][lda])
```

Tijelo funkcije je kao u prethodnom primjeru. Funkcija sada interpretira memoriju na pravi način. Argument `lda` se u listi argumenata funkcije mora nalaziti ispred `double A[m][lda]`.

**Zadatak:** preuredite funkciju `matmul` za **množenje matrica** tako da koristi polja **varijabilne** duljine.

# Pokazivači, aritmetika pokazivača

Pokazivač na `tip` je varijabla koja sadrži adresu varijable tipa `tip`. Pokazivač se deklarira kao:

```
1 mem_klasa tip *p_var;
```

Tip pokazivača je vezan uz tip memorijske lokacije na koju pokazuje. Vrijedi za sve pokazivače osim generičke pokazivače `void *p` koji pokazuju na bilo što.

Znak * uvijek djeluje na prvi sljedeći simbol.

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

# Adresni operator i operator dereferenciranja

Adresni operator & se koristi za dohvaćanje adrese varijable. &x (adresa varijable $x$).

Operator dereferenciranja * dohvaća sadržaj na zadanoj adresi. *p (vrijednost spremljena u memorijsku lokaciju na koju pokazuje p).

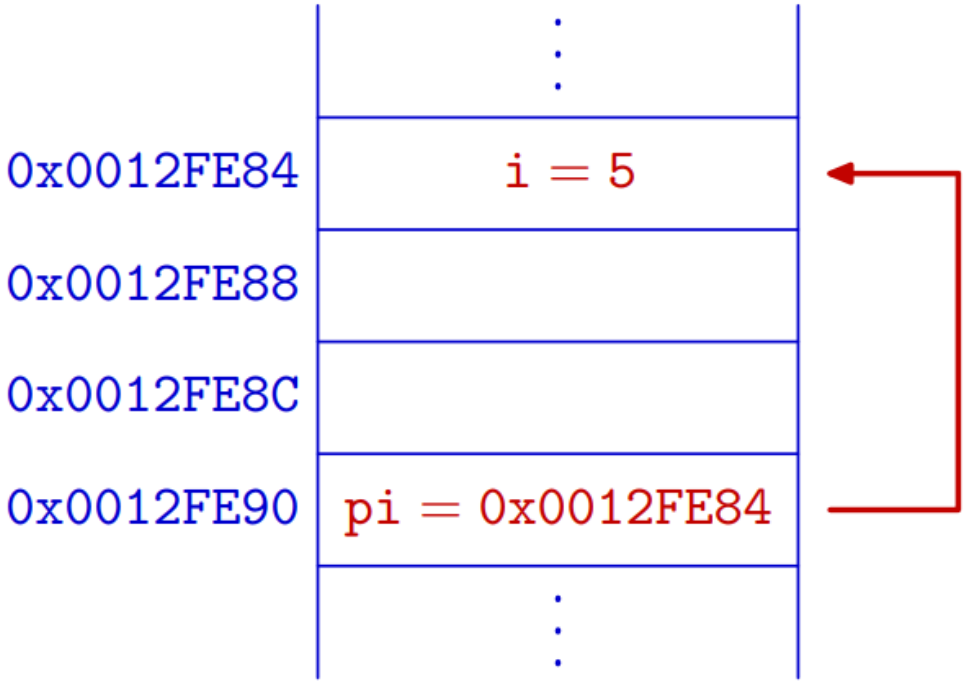
Operatori & i * su unarni operatori, asocijativnosti $D \rightarrow L$ ukoliko su zapisani ispred operanda. Ukoliko su zapisani između dva operanda, radi se o binarnim operatorima bit-po-bit i, te množenju.

Varijablu tipa pokazivač možemo inicijalizirati pri definiciji adresom neke druge varijable. Ta druga varijabla mora biti definirana prije no što se na nju primijeni adresni operator (mora imati adresu).

```
1 int i = 5;
2 int *pi = &i; /* Inicijalizacija adresom */
3 i = 2 * (*pi + 6); /* Indirektni pristup */
4 printf("i = %d, adresa od i = %p\n", i, pi);
```

# Adresni operator i operator dereferenciranja

`&i` - **adresa** varijable i.
`*pi` - **vrijednost** spremljena u memorijsku lokaciju na koju pokazuje pi.



Pokazivači mogu biti argumenti funkcije. U tom slučaju, funkcija može promijeniti vrijednost varijable na koju pokazivač pokazuje (tzv. varijabilni argument).

```
1 void zamjena(int *px, int *py) {
2 int temp = *px;
3 *px = *py;
4 *py = temp; }
```

Poziv funkcije treba biti:

```
1 zamjena(&a, &b); /* Poslati adrese! */
```

# Operacije nad pokazivačima

Nad pokazivačima je definiran ograničen skup operacija (moraju imati smisla za memorijske adrese).

Množenje pokazivača nema smisla i **nije dozvoljeno**, iako su pokazivači vrsta cijelih brojeva bez predznaka.

Dozvoljene operacije nad pokazivačima:

- Dodavanje ili oduzimanje cijelog broja pokazivaču.
- Oduzimanje pokazivača istog tipa (jedina dozvoljena aritmetička operacija za dva pokazivača).
- Uspoređivanje pokazivača istog tipa relacijskim operatorima.

# Aritmetika pokazivača i indeksi u polju

Sve aritmetičke operacije nad pokazivačima **ekvivalentne su** aritmetici indeksa u polju odgovarajućeg tipa (ne stvarnoj aritmetici adresa).

Ime polja je **konstantni pokazivač** na prvi element polja. Za polje *a*, `a = &a[0]` ili `*a = a[0]`.
Također, `a + i = &a[i]`, `*(a + i) = a[i]`, $\forall$ i.

Stvarne adrese ovise o **tipu** (veličini elemenata u polju).
`a+i` $\Leftrightarrow$ `adresa u a+i*sizeof(tip elemenata u polju a)`.

Svaki pokazivač na neki objet možemo interpretirati i kao pokazivač na **prvi element** u polju objekata tog tipa. Svakom pokazivaču možemo **dodati i oduzeti** cijeli broj.

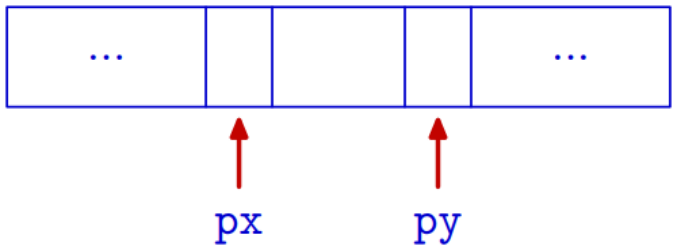
# Aritmetika pokazivača i indeksi u polju

Ako je *p* pokazivač (osim generičkog) i *n* varijabla cjelobrojnog tipa, dozvoljene su operacije:

```
1 p + n   p - n   ++p   --p   p++   p--
```

Pokazivač `p+n` pokazuje na *n*-ti objekt nakon onog na kojeg pokazuje *p*, odnosno vrijedi: `p+n` $\Leftrightarrow$ `adresa u p + n*sizeof(tip objekta na koji pokazuje p)`.

Pokazivače istog tipa smijemo oduzimati. Operacija ima smisla ako oba pokazivača pokazuju na **isto polje**. Za dva pokazivača `px` i `py`, koji pokazuju na isto polje), `py-px+1` je broj elemenata između `px` i `py` (uključujući rubove).



# Pokazivači

Razlika pokazivača je vrijednost cjelobrojnog tipa (tipa `ptrdiff_t`) definiranog u `<stddef.h>`.

Pokazivače istog tipa smijemo međusobno **uspoređivati** relacijskim operatorima. Ako su `px` i `py` dva pokazivača istog tipa, onda možemo koristiti izraze:

```
1 px <= py   px >= py   px < py   px > py
2 px == py   px != py
```

Uspoređivanje ima smisla samo ako oba pokazivača pokazuju na **isto** polje.

Pokazivaču nije moguće pridružiti vrijednost cjelobrojnog tipa, osim nule (koja nije legalna adresa). Nula označava da pokazivač nije inicijaliziran (`double *p = 0`). Bolje je naglasiti da se radi o pokazivaču i koristiti simboličku konstantu `NULL` definiranu u zaglavlju `<stdio.h>` (`double *p = NULL`).

# Pokazivači

```
1 double *px;
2 if (px != 0) ... /* Korektno! */
3 if (px != NULL) ... /* Jos bolje! */
4 if (px == 0x3451) ... /* GRESKA! Usporedjivanje
5 s cijelim brojem. */
```

Pokazivači na različite tipove podataka općenito se ne mogu međusobno pridruživati (osim pridruživanja pokazivača proizvoljnog tipa generičkom pokazivaču). Pridruživanje pokazivača različitih tipova se može izvršiti jedino korištenjem eksplicitnog pretvaranja tipa (cast operator).

```
1 char *pc;
2 int *pi;
3 pi = pc; /* GRESKA. */
4 pi = (int *) pc; /* ISPRAVNO. */
```

# Generički pokazivači

Generički pokazivač deklarira se kao pokazivač na `void`. `void *p;`
Vrijednost pokazivača na bilo koji tip može se dodijeliti pokazivaču na `void` i obratno, bez promjene tipa pokazivača (ne treba cast).

```
1 double *pd0, *pd1;
2 void *p;
3 ...
4 p = pd0; /* ISPRAVNO. */
5 pd1 = p; /* ISPRAVNO. */
```

Osnovna uloga generičkog pokazivača je omogućavanje definiranja funkcija koje mogu primati pokazivač na proizvoljni tip podataka.

```
1 double *pd0;
2 void f(void *);
3 ...
4 f(pd0); /* OK. */
```

# Generički pokazivač

Generički pokazivač se **ne smije** dereferencirati, povećati ili smanjiti (te operacije ovise o tipu pokazivača).

U `<stdlib.h>` postoje funkcije `qsort` i `bsearch` za općenito sortiranje niza podataka i binarno traženje.

```
1 void qsort(void *base, size_t n, size_t size,
2 int (*comp) (const void *, const void *));
3 void *bsearch(const void *key, const void *base,
4 size_t n, size_t size,
5 int (*comp) (const void *, const void *));
```

Funkcijama gore je svejedno koji je tip podataka u nizu i zato su argumenti tipa `void *`. Samo funkcija `comp` mora voditi računa o tipu i pretvoriti generičke pokazivače u pokazivače na odgovarajući tip.

# Pokazivači i const

Modifikator (ključnu riječ) `const` koristimo za definiciju konstanti.
`const double g = 9.81; //ubrzanje gravitacije`
Varijabli $g$ tada ne smijemo promijeniti vrijednost. Modifikator `const` smijemo primijeniti i na pokazivače. Konstantni pokazivač uvijek pokazuje na istu lokaciju. Moguće je definirati obični i konstantni pokazivač na nekonstantni ili konstantni tip.

```
1  double x[] = {0.1, 0.2, 0.3};
2  const double y[] = {0.1, 0.2, 0.3};
3  const double *p1; /* Ptr na konst. double */
4  double * const p2 = x; /* Konst. ptr na double */
5  const double * const p3=y;/*K. p. na k. double*/
6  p1 = x; /* OK, ali x NE mogu mijenjati kroz p1 */
7  p1[1] = 4.0; /* GRESKA. */
8  p2 = &x[2]; /* GRESKA. */
9  p3 = &y[2]; /* GRESKA. */
10 *p3 = 4.0; /* GRESKA. */
```

# Pokazivači i polja

Pokazivač $p$ na proizvoljni tip, koji nije generički, može se interpretirati kao pokazivač na prvi element u polju odgovarajućeg tipa (`p = &p[0]`). Za pokazivač $p$ smijemo koristiti i aritmetiku pokazivača i indekse (mogu se i miješati).

Veza između aritmetike pokazivača i indeksiranja je `p+i = &p[i]`, `*(p+i) = p[i]`, gdje cijeli broj $i$ može biti i negativan. Pokazivaču $p$ koji nije konstantan smijemo mijenjati vrijednost.

```
1  char *px, x[128];
2  px = &x[0]; /* Isto kao px = x; */
3  *(px + 3) = 'z'; /* Isto kao x[3] = 'z'; */
4  ++x; /* GRESKA - konst. pointer */
5  ++px; /* Isto kao px = &x[1]; */
6  *(px + 1) = 'h'; /* Isto kao px[1] = 'h';
7  ekviv. s x[2] = 'h'; */
8  *(px + 130)='d';/*Izvan polja, ne javlja gresku*/
```

# Pokazivači i polja

Indeksiranje jednodimenzionalnog polja:
`double x[10]; x[i]` $\Leftrightarrow$ `*(x+i)`

Indeksiranje višedimenzionalnog polja:
`double x[10][20]; x[i][j]` $\Leftrightarrow$ `*(x[i]+j)` $\Leftrightarrow$ `*(*(x+i)+j)`.

```
1  #include <stdio.h>
2  int main(void) {
3  int a[2][3] = {{1,2,3}, {4,5,6}}, *pa;
4  pa = a[0]; /* Ekviv. s pa = (int *) a; */
5  pa = pa + 3; /* Idemo 3 int-a dalje u polju. */
6  printf("%d\n", *pa); /* 4 */
7  printf("%d\n", **(a + 1)); /* 4 */
8  printf("%d\n", *(a[1] + 1)); /* 5 */
9  return 0;
10 }
```

# Polje kao argument funkcije

Polje smije biti argument funkcije. Polje se ne kopira već samo pokazivač na prvi element polja. Kod poziva smijemo navesti polje (bez zagrada) ili pokazivač na bilo koji element polja (objekt odgovarajućeg tipa). Unutar funkcije elementi se mogu dohvatiti/mijenjati korištenjem indeksa ili aritmetike pokazivača.

Kod deklaracije jednodimenzionalnog polja kao formalnog argumenta funkcije, može se koristiti:

```
1 tip_pod ime[izraz_1]
2 tip_pod ime[]
3 tip_pod *ime
4 tip_pod (*ime)
```

Okrugle zagrade su nužne kod višedimenzionalnih polja.

# Polje kao argument funkcije

Kod deklaracije višedimenzionalnog polja kao formalnog argumenta funkcije, može se koristiti:

```
1 tip_pod ime[izraz_1][izraz_2]...[izraz_n]
2 tip_pod ime[][izraz_2]...[izraz_n]
3 tip_pod (*ime)[izraz_2]...[izraz_n]
```

Ako ne želimo dozvoliti mijenjanje polja unutar funkcije, ispred tipa dodamo ključnu riječ `const`.