

Programiranje 2

4. predavanje

Saša Singer

`singer@math.hr`

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

PMF – Matematički odsjek, Zagreb

Sadržaj predavanja

- Višedimenzionalna polja (kraj):
 - Množenje matrice i vektora. Množenje matrica.
 - Brzina operacija i sekvencijalni pristup podacima.
 - Polja varijabilne duljine (C99).
- Pokazivači (prvi dio):
 - Deklaracija pokazivača.
 - Pokazivači kao argumenti funkcije.
 - Operacije nad pokazivačima. Aritmetika pokazivača.
 - Pokazivači i polja.
 - Indeksi u polju i aritmetika pokazivača.
 - Dinamička alokacija memorije.
- Dodatak: pokazivači – ponavljanje i primjeri.

Informacije — kolokviji

Programiranje 2 je u kolokvijskom razredu C3.

Službeni termini svih kolokvija su:

- Prvi kolokvij: petak, 27. 4. 2018., u 15 sati.
- Drugi kolokvij: četvrtak, 21. 6. 2018., u 15 sati.
- Popravni kolokvij: utorak, 4. 9. 2018., u 15 sati.

Uputa: “izbjegnite” popravni — obavite to ranije!

Informacije

Konzultacije:

🕒 petak, 12–14 sati, ili — po dogovoru.

Ne zaboravite, “žive” su i domaće zadaće na adresi

<http://degorgi.math.hr/prog2/ku/>

Dodatni bodovi “čekaju na vas”.

Višedimenzionalna polja

Operacije s matricama i vektorima

Sadržaj

- Višedimenzionalna polja (kraj):
 - Množenje matrice i vektora.
 - Množenje matrica.
 - Brzina operacija i sekvencijalni pristup podacima.
 - Polja varijabilne duljine (C99).

Množenje matrice i vektora $y = Ax$

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

Formula za elemente vektora y je

$$y_i = \sum_{j=1}^n a_{ij} \cdot x_j,$$

za sve indekse

$$i = 1, \dots, m.$$

Dakle, “programski” — treba “zavrtiti” dvije petlje.

Jedina razlika u C-u — indeksi idu od nule.

Množenje matrice i vektora $y = Ax$ (nastavak)

Primjer. Dio **glavnog** programa.

```
#include <stdio.h>
#define MAX_m 10
#define MAX_n 10
int main(void) {
    int A[MAX_m][MAX_n], x[MAX_n], y[MAX_m];
    int m, n;    /* Stvarne dimenzije matrice A. */

    void umnozak(int, int, int mat1[][MAX_n],
                 int mat2[MAX_n], int mat3[MAX_m]);

    ...
    umnozak(m, n, A, x, y);
    ... }
```

Funkcija umnozak

```
void umnozak(int m, int n, int mat1[][MAX_n],
             int mat2[MAX_n], int mat3[MAX_m])
{
    int i, j;
    /* Mnozenje matrice i vektora */
    for (i = 0; i < m; ++i) {
        mat3[i] = 0;
        for (j = 0; j < n; ++j)
            mat3[i] += mat1[i][j] * mat2[j];
    }
}
```

Zadatak. Prepišite funkciju tako da se varijable zovu **A**, **x** i **y**.

Množenje matrica

Primjer. Zadane su 3 pravokutne matrice:

- A — tipa $m \times \ell$,
- B — tipa $\ell \times n$,
- C — tipa $m \times n$.

Treba izračunati izraz

$$C := C + A * B.$$

Akumulacija (“nazbrajavanje”) produkta $A * B$ u matrici C

- standardni je oblik BLAS-3 rutine xGEMM za množenje matrica,

tj. baš ova operacija se često koristi u praksi.

Množenje matrica — formula

“Matematička” realizacija **matrice** operacije

$$C := C + A * B$$

po **elementima** je trivijalna:

$$c_{ij} := c_{ij} + \sum_{k=1}^{\ell} a_{ik} \cdot b_{kj},$$

za sve indekse

$$i = 1, \dots, m, \quad j = 1, \dots, n.$$

Dakle, “programski” — treba “zavrtiti” **tri** petlje.

Opet, jedina razlika u **C**-u — indeksi idu od **nule**.

Množenje matrica — skica funkcije

Funkcija `matmul` koja računa $C := C + A * B$ mora imati sljedeće argumente:

- matrice A , B i C u obliku

- `double A[][lda]`,

- `double B[][ldb]`,

- `double C[][ldc]`,

gdje su `lda`, `ldb` i `ldc` druge dimenzije (brojevi stupaca = duljine redaka) iz definicije ovih matrica — tamo gdje je rezervirana memorija za njih,

- stvarne dimenzije matrica

- `int m`, `int n` i `int l`

s kojima ćemo raditi.

Množenje matrica — funkcija

```
void matmul( int m, int n, int l,  
             double A[][lda],  
             double B[][ldb],  
             double C[][ldc] )  
{  
    int i, j, k;  
  
    for (i = 0; i < m; ++i)  
        for (j = 0; j < n; ++j)  
            for (k = 0; k < l; ++k)  
                C[i][j] += A[i][k] * B[k][j];  
  
    return;  
}
```

Množenje matrica — brzina

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

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

Množenje matrica — povećanje brzine

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

```
...  
for (i = 0; i < m; ++i)  
    for (k = 0; k < l; ++k)  
        for (j = 0; j < n; ++j)  
            C[i][j] += A[i][k] * B[k][j];  
...
```

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

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

Množenje matrica — kraj

Napomena. Ako trebamo samo **produkt** $C = A * B$, onda

🔴 inicijaliziramo $C = 0$ (na **nul-matricu**).

To možemo učiniti **prije** poziva naše funkcije, ili **u funkciji**, ali **ispred** one **tri** petlje za množenje (da ne mijenjamo **optimalni** poredak petlji).

```
for (i = 0; i < m; ++i)
    for (j = 0; j < n; ++j)
        C[i][j] = 0.0;
for (i = 0; i < m; ++i)
    for (k = 0; k < l; ++k)
        for (j = 0; j < n; ++j)
            C[i][j] += A[i][k] * B[k][j];
```

Polja varijabilne duljine

Problem u **C90** standardu za rad s **matricama**:

- **dimenzije** polja u deklaraciji **argumenata** funkcije moraju biti **konstantni** izrazi, pa **promjena dimenzije** u **definiciji** (višedim.) polja $\Rightarrow$ **promjena deklaracije** svih funkcija.

Standard **C99** uvodi tzv. polja “**varijabilne**” **duljine**, ali ih neki prevoditelji još uvijek **ne** implementiraju u potpunosti.

- Polje **varijabilne** duljine je **automatsko** polje, čije **dimenzije** mogu biti zadane **vrijednostima varijabli**.

Primjer:

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

Polja varijabilne duljine (nastavak)

Osnovna svrha polja varijabilne duljine je pisanje funkcija

koje, kao argument, primaju dimenzije polja.

Ideja: te dimenzije su one iz definicije polja (prva nije bitna).

Primjer. Funkcija koja računa Euklidsku (ili Frobeniusovu) normu matrice. Neka je A matrica tipa $m \times n$ s elementima

$$a_{ij}, \quad i = 1, \dots, m, \quad j = 1, \dots, n.$$

Euklidska ili Frobeniusova norma matrice A definira se ovako

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

Polja varijabilne duljine — pogrešno

Prva verzija (v. `e_norm_1.c`):

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

    for (i = 0; i < m; ++i)
        for (j = 0; j < n; ++j)
            suma += A[i][j] * A[i][j];
    return sqrt(suma);
}
```

Međutim, **NE TAKO!** To **nije** svrha (ideja) i još **NE RADI**.

Polja varijabilne duljine (nastavak)

Argumenti `m` i `n` su stvarne dimenzije radnog dijela matrice `A`.

No, zbog deklaracije matrice `A` u obliku `double A[m][n]`,

- funkcija “vidi” retke matrice `A` kao polja duljine `n`,
- što ne mora odgovarati pravoj duljini redaka iz definicije matrice (negdje izvan funkcije).

Posljedica: funkcija “čita” elemente matrice `A` s pogrešnih adresa (od drugog retka nadalje) $\implies$ rezultat je pogrešan!

Popravlak. Funkcija mora dobiti još jedan argument,

- nazovimo ga `lda` (“last dimension of A”), koji sadrži drugu dimenziju matrice iz njezine definicije (rezervacije memorije) = broj stupaca = duljina svakog retka,
- a matricu `A` treba deklarirati kao `double A[m][lda]`.

Polja varijabilne duljine — ispravno

Dakle, **zaglavlje** funkcije treba biti (v. `e_norm_2.c`):

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

a tijelo funkcije je **isto** kao i prije.

Funkcija sad **korektno** “vidi” **pravu duljinu** svakog **retka** i to je **jedina** svrha argumenta **lda**.

- Argument **lda** mora biti **ispred** `double A[m][lda]`.
- Prva** dimenzija matrice (polja) se **ignorira**, kao i prije, pa možemo pisati i `double A[][lda]` (bez `m`).

Općenito, **prvu** dimenziju polja iz **definicije** **ne treba** slati kao argument (za matrice, to je rezervirani **broj** redaka).

Polja varijabilne duljine — kraj

Pogledajte programe `e_norm_1.c` i `e_norm_2.c`, zajedno s rezultatima (`*.out`) i provjerite da prva verzija radi pogrešno.

Usput, poredak petlji u funkciji je dobar — elementima matrice `A` pristupamo redom kako su spremljeni u memoriji.

Zadatak. Preuredite funkciju `readmat` za čitanje matrice tako da koristi polja varijabilne duljine i vraća učitane “radne” dimenzije matrice `m` i `n` kao varijabilne argumente.

Zadatak. Preuredite funkciju `matmul` za množenje matrica tako da koristi polja varijabilne duljine.

Napomena. Kod prevođenja ovih programa, prevoditelju treba zadati opciju da koristi jezička pravila iz C99 standarda. Inače, javlja greške!

Pokazivači, aritmetika pokazivača

Sadržaj

- Pokazivači (prvi dio):
 - Deklaracija pokazivača.
 - Pokazivači kao argumenti funkcije.
 - Operacije nad pokazivačima. Aritmetika pokazivača.
 - Pokazivači i polja.
 - Indeksi u polju i aritmetika pokazivača.
 - Dinamička alokacija memorije.

Podsjetnik

Svakoj **varijabli** u programu pridružena je **memorijska lokacija** (ili blok lokacija). **Veličina** tog bloka ovisi o **tipu varijable**.

- Svaka **memorijska lokacija** ima **jedinstvenu adresu**.
- Radi jednostavnosti, možemo zamišljati da je
 - **adresa** čitavog **bloka** lokacija = **adresa prve** lokacije u tom **bloku**.

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

Varijabli (njezinom sadržaju) se može pristupiti

- **izravno** — **imenom** varijable (prevoditelj “zna” adresu),
- **indirektno** — preko **adrese** varijable, tj. **pokazivačem** na tu varijablu.

Deklaracija pokazivača

Pokazivač na **tip** je varijabla koja sadrži **adresu** varijable tog tipa **tip**.

Deklaracija **pokazivača** (engl. “pointer”):

```
mem_klasa tip *p_var;
```

Dakle, tip **pokazivača** je vezan uz **tip** “onog **na što** pokazuje”, osim tzv. **generičkog pokazivača** **void *p** — on pokazuje na **bilo što** (ili na “ništa”).

Primjer. Zvezdica ***** uvijek djeluje na **prvi sljedeći** simbol.

```
static int *pi;  
char* pc;  
float* pf, f;
```

```
double *px;  
int a, *b;  
void *p;
```

Adresni operator & i operator dereferenciranja *

Adresni operator &:

- $\&x$ = **adresa** varijable x .

Operator **dereferenciranja** * (“uzimanje sadržaja na adresi”):

- $*p_x$ = **vrijednost** spremljena u memorijsku lokaciju na koju **pokazuje** p_x ,
- tj. **sadržaj** na **adresi** spremljenoj u p_x .

Operatori & i * su **unarni** operatori, asocijativnost $D \rightarrow L$.

Napomena. Simboli & i * mogu biti i **binarni** operatori (bit-po-bit i, množenje) — **ovisno** o tome kako su **napisani**:

- **ispred** operanda $\rightarrow$ **unarni**,
- **između** dva operanda $\rightarrow$ **binarni**.

Inicijalizacija pokazivača

Pokazivačku varijablu pri definiciji možemo inicijalizirati

📍 adresom neke druge varijable.

Varijabla čiju adresu koristimo, mora biti definirana prije no što se na nju primijeni adresni operator (mora imati adresu).

Primjer. Indirektni pristup varijabli preko pokazivača.

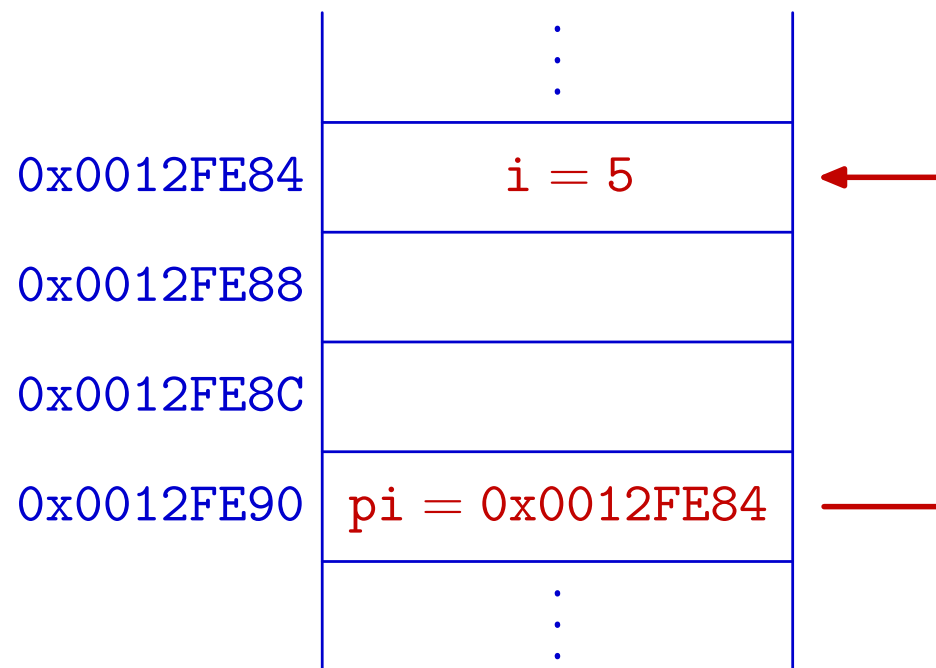
```
int i = 5;
int *pi = &i;  /* inicijalizacija adresom */
...
i = 2 * (*pi + 6);    /* indirektni pristup */
printf("i = %d, adresa od i = %p\n", i, pi);
```

Izlaz: i = 22, adresa od i = 0012FE84.

Adresni operator & i operator dereferenciranja *

Uočite da je:

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



Pokazivači i funkcije

Pokazivači mogu biti argumenti funkcije. U tom slučaju,

- funkcija može promijeniti vrijednost varijable na koju pokazivač pokazuje (tzv. varijabilni argument).

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

```
void zamjena(int *px, int *py) {  
    int temp = *px;  
    *px = *py;  
    *py = temp; }  

```

Poziv funkcije treba glasiti:

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

Operacije nad pokazivačima — uvod

Nad **pokazivačima** smijemo raditi samo **poneke operacije**,

- koje su “**konzistentne**” — tj. imaju **smisla** za memorijske adrese.

Na primjer, **množenje** pokazivača (adresa) **nema** smisla i **nije** dozvoljeno, iako su pokazivači, interno, neka vrsta cijelih brojeva bez predznaka.

Dozvoljene su sljedeće **operacije** nad pokazivačima.

- Svakom **pokazivaču** smijemo **dodati** i **oduzeti cijeli broj**.
- Oduzimanje** pokazivača **istog** tipa — i to je **jedina** dozvoljena **aritmetička** operacija (za **dva** pokazivača).
- Uspoređivanje** pokazivača **istog** tipa — **relacijskim** operatorima.

Aritmetika pokazivača = indeksi u polju

Sve aritmetičke operacije nad pokazivačima ekvivalentne su

- aritmetici indeksa u polju odgovarajućeg tipa, a ne stvarnoj aritmetici adresa.

Krenimo od polja (v. Prog 1):

- Ime polja je konstantni pokazivač na prvi element polja.

Ako je a neko polje, onda je: $a = \&a[0]$ ili $*a = a[0]$.

Elementi polja spremaju se na uzastopnim lokacijama u memoriji. Zato za svaki element polja a vrijedi veza:

- $a + i = \&a[i]$ ili $*(a + i) = a[i]$, za svaki i .

Stvarne adrese ovise o “duljini” elemenata u polju, tj. o tipu

$a + i \iff \text{adresa u } a + i * \text{sizeof}(\text{tip elemenata u polju } a).$

Aritmetika pokazivača (nastavak)

Vrijedi i “**obrat**”: Svaki **pokazivač** na neki objekt možemo interpretirati i kao

📍 **pokazivač** na **prvi element** u **polju** objekata tog tipa.

Onda, svakom **pokazivaču** možemo **dodati i oduzeti cijeli broj**.

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

px + n px - n ++px --px px++ px--

Pokazivač **px + n** pokazuje na **n-ti** objekt **nakon** onog na kojeg pokazuje **px**, tj. u terminima **adresa** vrijedi

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

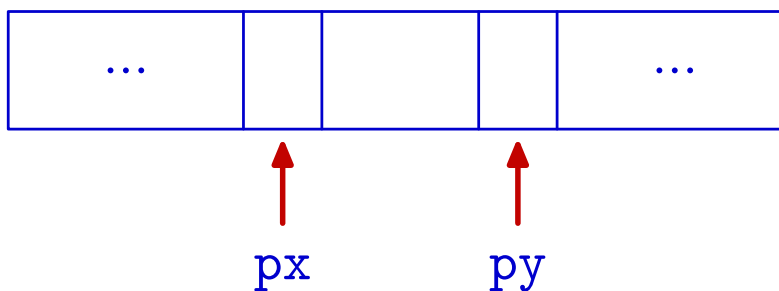
+ n * sizeof(tip objekta na koji pokazuje px).

Razlika pokazivača

Pokazivače **istog** tipa smijemo **oduzimati**.

- ☛ To ima **smisla samo** ako oni pokazuju na **isto polje**.
- ☛ Ako su **px** i **py** dva pokazivača (na isto polje), onda je
 - ☛ $py - px + 1 =$ **broj elemenata** između **px** i **py**,
uključujući i krajeve.

Ovaj rezultat odgovara **aritmetici** pokazivača!



Razlika pokazivača je vrijednost **cjelobrojnog tipa**, preciznije, tipa **ptrdiff_t** definiranog u **<stddef.h>**.

Usporedba pokazivača

Pokazivače **istog** tipa smijemo međusobno **uspoređivati** **relacijskim** operatorima.

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

px <= **py**

px >= **py**

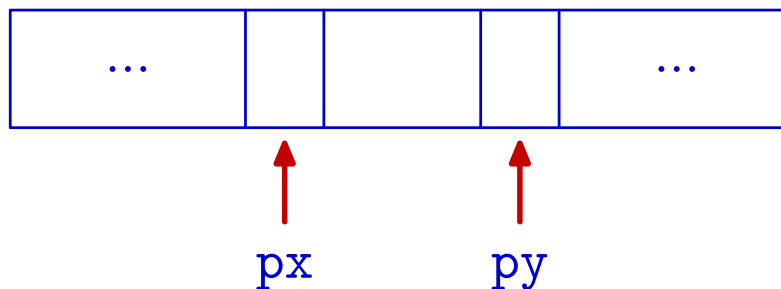
px < **py**

px > **py**

px == **py**

px != **py**

- 📌 Opet, to uspoređivanje ima **smisla samo** ako pokazivači pokazuju na **isto polje**.



Pokazivači i cijeli brojevi

Pokazivaču **nije moguće pridružiti** vrijednost **cjelobrojnog** tipa, **osim nule**.

- **Nula nije** legalna adresa!
- Ona označava da pokazivač **nije inicijaliziran**.

Može se pisati

```
double *p = 0;
```

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

```
double *p = NULL;
```

Pokazivači i cijeli brojevi (nastavak)

Primjer:

```
double *px;
```

```
...
```

```
if (px != 0) ...      /* Korektno!  
                        Je li pokazivac  
                        inicijaliziran? */
```

```
if (px != NULL) ...  /* Jos bolje! */
```

```
if (px == 0x3451) ... /* GRESKA!  
                        Usporedjivanje  
                        s cijelim brojem */
```

Pokazivači različitih tipova

Pokazivači na različite tipove podatka, općenito, se ne mogu međusobno pridruživati.

- 🔴 Ako to želimo, treba koristiti eksplicitno pretvaranje tipa (cast operator).

Jedini izuzetak je tzv. generički pokazivač (v. malo kasnije).

Primjer:

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

Pokazivači različitih tipova (nastavak)

Razlog za zabranu “miješanja tipova” je vrlo jednostavan.

- Osnovne operacije s pokazivačima — dereferenciranje, povećanje i smanjenje, bitno ovise o tipu pokazivača.

Primjer. Uzmimo iste deklaracije kao u prošlom primjeru:

```
char *pc;  
int *pi;  
...  
pi = (int*)pc;  
printf("%u\n", (pi == (int*)pc));  
printf("%u\n", (pi + 1 == (int*)(pc + 1)));
```

Prva naredba ispiše 1, a druga 0. Dakle, pomaci su različiti.
Probajte ispisati vrijednosti sva 4 pokazivača u %p formatu!

Generički pokazivač

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`).

Primjer:

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

Generički pokazivač (nastavak)

Osnovna uloga generičkog pokazivača je omogućiti da

- funkcija prima pokazivač na bilo koji tip podatka.

Primjer:

```
double *pd0;  
void f(void *);  
...  
f(pd0);    /* OK */
```

Korist od toga je donekle ograničena, jer

- generički pokazivač se ne smije dereferencirati, povećati i smanjiti.

Naime, sve navedene operacije bitno ovise o tipu pokazivača.

Generički pokazivač — svrha

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

```
void qsort(void *base, size_t n, size_t size,  
          int (*comp) (const void *, const void *));  
void *bsearch(const void *key, const void *base,  
             size_t n, size_t size,  
             int (*comp) (const void *, const void *));
```

Ovim funkcijama je **svejedno** koji je **tip podataka** u **nizu** i zato su argumenti tipa `void *` (tj. **generički** pokazivači).

Samo funkcija `comp` — za **uspoređivanje** podataka u nizu, mora voditi računa o **tipu**. Ona će **generičke** pokazivače **pretvoriti** u pokazivače na **odgovarajući tip** (v. **KR2**, str. **120**).

Pokazivači i const

Modifikator (ključnu riječ) `const` koristimo za definiciju konstanti.

```
const double g = 9.81;  /* ubrz. 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 konstantni pokazivač

- na nekonstantni tip i na konstantni tip.

Primjer (v. sljedeća stranica):

Pokazivači i const (*nastavak*)

```
double x[] = {0.1, 0.2, 0.3};  
const double y[] = {0.1, 0.2, 0.3};
```

```
const double *p1;          /* ptr na konst. double */  
double * const p2 = x;     /* konst. ptr na double */  
const double * const p3 = y;  
                           /* konst. ptr na konst. double */
```

```
p1 = x;   /* OK, ali x NE mogu mijenjati kroz p1 */  
p1[1] = 4.0;    /* GRESKA */  
p2 = &x[2];     /* GRESKA */  
p3 = &y[2];     /* GRESKA */  
*p3 = 4.0;     /* GRESKA */
```

Pokazivači i polja

Sadržaj

- Pokazivači i polja:
 - Pokazivači i jednodimenzionalna polja.
 - Ime polja = pokazivač na prvi element.
 - Indeksi u polju i aritmetika pokazivača.
 - Jednodimenzionalno polje.
 - Višedimenzionalno polje.
 - Polje/pokazivač kao argument funkcije.
 - Dvodimenzionalna polja — ponavljanje, tipovi, sizeof.

Pokazivači i jednodimenzionalna polja

Neka je **p** bilo koji pokazivač osim generičkog, tj. **p** može biti pokazivač na bilo koji tip (osim **void**).

Onda **p** možemo interpretirati i kao

● pokazivač na prvi element u polju odgovarajućeg tipa, tj. kao **p = &p[0]**. Nadalje, za pokazivač **p** smijemo koristiti ● i aritmetiku pokazivača i indekse (možemo ih “miješati”).

Veza između aritmetike pokazivača i indeksiranja je

$$p + i = \&p[i], \quad *(p + i) = p[i],$$

gdje je **i** cijeli broj (smije biti i negativan).

Za razliku od “običnog” polja, ako **p** nije definiran kao konstantan pokazivač, smijemo mu mijenjati vrijednost.

Pokazivači i jednodimenzionalna polja (nast.)

Oprez! Kod ovih operacija **nema** nikakve **kontrole indeksa** — jesmo li u korektno rezerviranim **granicama memorije**!

Primjer. Aritmetika pokazivača za **jednodimenzionalna** polja.

```
char *px, x[128];

px = &x[0];          /* Isto kao  px = x; */
*(px + 3) = 'd';      /* Isto kao  x[3] = 'd'; */
++x;                 /* GRESKA - konst. pointer */
++px;                 /* Isto kao  px = &x[1]; */
*(px + 1) = 'b';      /* Isto kao  px[1] = 'b';
                       ekv. s      x[2] = 'b'; */
*(px + 130) = 'z';    /* Formalno, NIJE GRESKA,
                       ali gazimo po memoriji! */
```

Aritmetika pokazivača i višedimenzionalna polja

Indeksiranje **jednodimenzionalnog** polja:

```
double x[10];
```

🔴 $x[i] \iff *(x + i).$

Indeksiranje **višedimenzionalnog** polja:

```
double x[10][20];
```

🔴 $x[i][j] \iff *(x[i] + j) \iff *(*x + i) + j).$

$x[i]$ je **pokazivač** na prvi element u polju $x[i]$, a to je $x[i][0]$. Dakle, $x[i] = \&x[i][0]$, kao za svako polje.

Također, $x + 1$ je **pokazivač** na sljedeći redak (polje) $x[1]$.

Aritmetika pokazivača i višedimenzionalna polja

Primjer. Aritmetika pokazivača ovisi o **tipu** pokazivača.

```
#include <stdio.h>
int main(void) {
    int a[2][3] = {{1,2,3}, {4,5,6}}, *pa;

    pa = a[0];      /* ekviv. s pa = (int *) a; */
    pa = pa + 3;     /* idemo 3 int-a dalje u polju */

    printf("%d\n", *pa);           /* 4 */
    printf("%d\n", **(a + 1));     /* 4 */
    printf("%d\n", *(a[1] + 1));   /* 5 */
    return 0;
}
```

Polje kao argument funkcije

Polje smije biti argument funkcije. Funkcija tada

- ne dobiva kopiju čitavog polja, već
- samo kopiju pokazivača na prvi element polja.

U pozivu funkcije, smijemo navesti

- ime polja (bez uglatih zagrada), jer ono predstavlja pokazivač na prvi element, ili
- pokazivač na bilo koji drugi element polja, ili
- pokazivač na bilo koji objekt odgovarajućeg tipa.

Unutar funkcije elementi polja mogu se

- dohvatiti i promijeniti, korištenjem indeksa polja ili aritmetike pokazivača.

Jednodimenzionalno polje kao argument

Kod deklaracije jednodimenzionalnog polja kao formalnog argumenta funkcije, mogu se koristiti ekvivalentne forme:

```
tip_pod ime[izraz_1]
```

ili bez prve dimenzije

```
tip_pod ime[]
```

ili pomoću pokazivača — to je “stvarno stanje stvari”

```
tip_pod *ime  
tip_pod (*ime)
```

Okrugle zagrade ovdje nisu nužne, ali za višedimenzionalna polja jesu.

Višedimenzionalno polje kao argument

Kod deklaracije višedimenzionalnog polja kao formalnog argumenta funkcije, mogu se koristiti ekvivalentne forme:

```
tip_pod ime[izraz_1][izraz_2]...[izraz_n]
```

ili bez prve dimenzije

```
tip_pod ime[][izraz_2]...[izraz_n]
```

ili pomoću pokazivača — okrugle zagrade su nužne

```
tip_pod (*ime)[izraz_2]...[izraz_n]
```

Ako ne želimo da funkcija mijenja elemente polja unutar funkcije, onda ispred tipa dodamo ključnu riječ `const`.

Polje kao argument funkcije — napomene

Napomena. Funkcija dobiva samo **pokazivač** na **jedan** objekt odgovarajućeg tipa, i

- 🕒 “**nema pojma**” je li to (**izvan** funkcije) **zasebni** objekt ili **element** nekog **polja**.

Koristeći

- 🕒 **ekvivalenciju indeksa** u polju i **aritmetike** pokazivača, **unutar** funkcije, taj **pokazivač** možemo interpretirati kao
- 🕒 **pokazivač** na **prvi element** polja odgovarajućeg tipa.

Primjer. Funkcija “**nema pojma**” o stvarnoj duljini polja. Operator **sizeof** vraća **stvarnu** duljinu polja samo tamo gdje je polje **definirano**, ali **ne** i za polje koje je **argument** funkcije (duljina se “ne vidi” — v. sljedeći primjer za **readmat**).

Dvodimenzionalna polja: ponavljanje, tipovi, sizeof

Dvodimenzionalna polja — ponavljanje (1)

Ponavljjanje 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} };
```

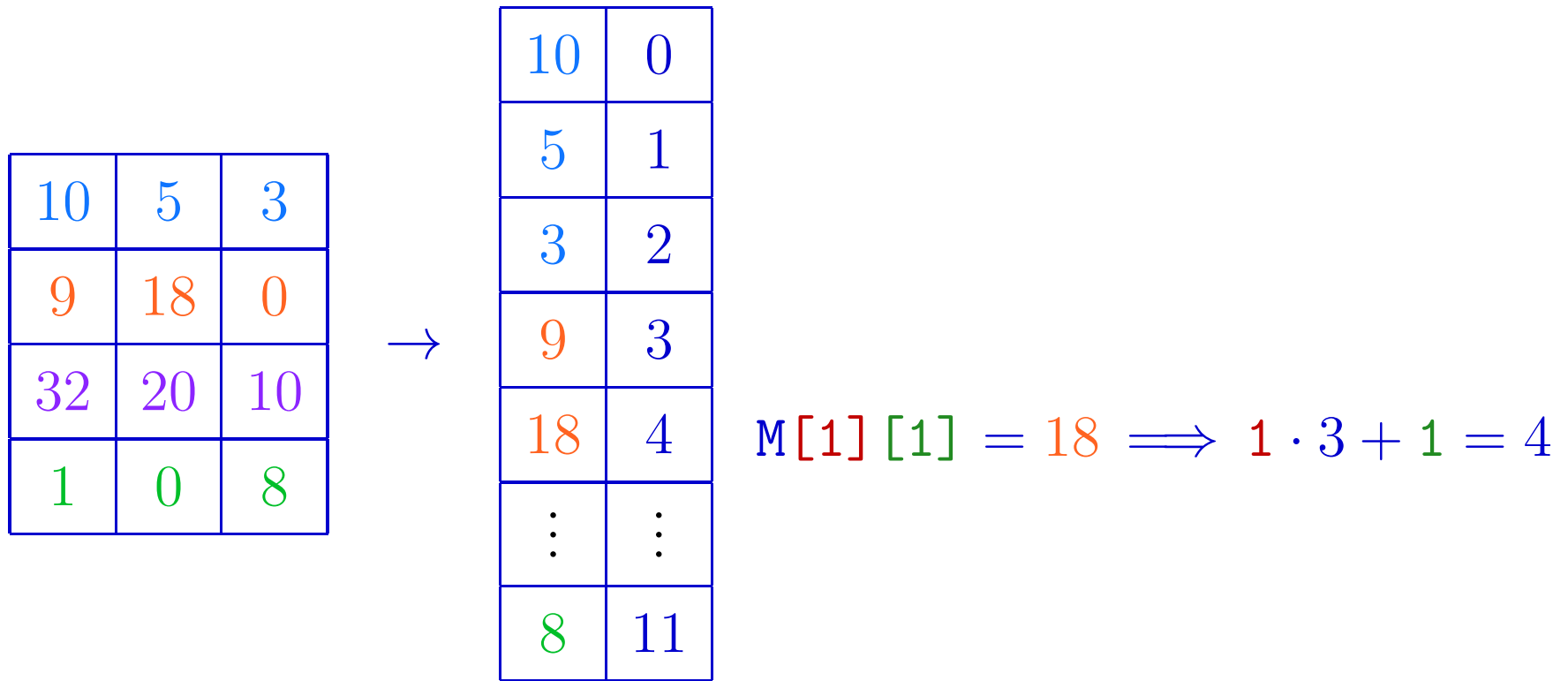
Ovdje je **M** matrica s 4 retka i 3 stupca, s elementima tipa **int**.

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:



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

Pozicija u memoriji **ne** ovisi o **broju redaka**, već samo o **duljini**.

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 (v. `sizes.c`)?

```
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, `sizeof =  $n \times \text{sizeof}(\text{element})$` .
To vrijedi **samo** tamo gdje je **polje definirano**, inače **ne**!

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
 - $\text{sizeof}(M[0]) = 3 \times \text{sizeof}(\text{int}) = 12$.
- M je **jednodimenzionalno** polje s 4 elementa tipa `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
 - $\text{sizeof}(M) = 12 \times \text{sizeof}(\text{int}) = 12 \times 4 = 48$.

Dvodimenzionalna polja i pokazivači (1)

Napomena. Prošli primjer može navesti na **pogrešan** zaključak

☛ što su **stvarni tipovi** objekata **M** i **M[0]**, ..., **M[3]**.

Usput, **M[0][0]** je “obični” **int** i tu nema problema.

Davno smo rekli i to **uvijek** vrijedi:

☛ **ime** polja je **sinonim** za **pokazivač** na **prvi element** polja!

M je **jednodimenzionalno** polje s 4 elementa **tipa int[3]**, pa je

☛ **tip** od **M** = **pokazivač** na **int[3]**, tj. **pokazivač** na **polje** od 3 **int**-a.

☛ **Zapis** tipa je **int (*M)[3]**, ili samo **int (*)[3]**.

Taj pokazivač ima **konstantnu** vrijednost **M = &M[0]**, tj. on je **sinonim** za (ili “sadrži”) adresu prvog elementa polja.

Dvodimenzionalna polja i pokazivači (2)

$M[i]$ je **jednodimenzionalno** polje s 3 elementa **tipa** **int**, pa je
● **tip** od $M[i]$ = **pokazivač** na **int**.

Taj pokazivač ima **konstantnu** vrijednost $M[i] = \&M[i][0]$,
tj. on je **sinonim** za (ili “sadrži”) adresu prvog elementa polja.

Bitno. Tamo gdje je polje **M** **definirano** (rezervirana memorija za njega), ovi pokazivači **M** i $M[i]$, za $i = 0, \dots, 3$,

- **nemaju** adresu — **nisu** spremljeni u memoriji,
- ali se **smiju** koristiti, s tim da prevoditelj **izračuna** njihove **vrijednosti** (adrese prvih elemenata odgovarajućih polja).
- Zato se **vrijednosti** **ne mogu** mijenjati (i zato “sadrži”)!

Usput, zato **sizeof** operator vraća **duljinu** odgovarajućih **polja**, a ne duljinu **pokazivača** (čim zna **duljinu** polja iz tipa)!

Dvodimenzionalna polja i pokazivači (3)

Kad je takvo **polje argument** neke funkcije, onda je **malo drugačije**. Na primjer, **bez obzira** na zaglavlje/prototip

```
void readmat(int mat[4][3], int m, int n)
void readmat(int mat[][3], int m, int n)
void readmat(int (*mat)[3], int m, int n)
```

unutar funkcije **readmat**, u sva **tri** slučaja, za **lokalnu** varijablu **mat** vrijedi

🕒 **tip** od **mat** = **pokazivač** na **int[3]**.

Osim toga, varijabla **mat** **ima adresu** i zato je

🕒 **sizeof(mat)** = duljina **pokazivača** = 4 ili 8.

Prva dimenzija polja, i kad piše, se **ne uzima** u obzir (ignorira)!

Dvodimenzionalna polja i pokazivači (4)

Međutim, za `mat[i]` vrijedi **isto** kao i prije:

- to je ime **polja** = `i`-tog retka u matrici `mat`, pa onda
- nema** adresu, već se vrijednost `mat[i] = &mat[i][0]` **računa** iz `mat`, indeksa `i` i **poznatog** tipa za `mat` (aritmetika pokazivača na retke = objekte tipa `int[3]`).

Zato se vrijednosti `mat[i]` opet **ne mogu** mijenjati, ali je

- duljina** polja `mat[i]` **poznata** iz tipa `int[3]`, tj. to **polje** ima **3** elementa tipa `int`, pa je
- `sizeof(mat[i]) = sizeof(int[3])`
`= 3 × sizeof(int) = 12.`

Probajte `sizeof` **izvan** i **unutar** funkcije!

Dinamičko rezerviranje memorije

Dinamičko rezerviranje memorije — uvod

Dosad smo upoznali (i koristili) samo objekte za koje se

- memorija rezervira odmah prilikom definicije objekta.

Znamo da postoje dvije vrste takvih objekata:

- automatski — rezervacija pri svakom ulasku u blok,
- statički — rezervacija jednom, na početku izvršavanja programa.

Postoji i treća vrsta objekata — tzv. dinamički objekti. Njih kreiramo po potrebi, za vrijeme izvršavanja programa,

- koristeći dinamičko rezerviranje (alokaciju) memorije.

Dinamičke objekte možemo i uništiti,

- oslobađanjem alocirane memorije.

Dinamička alokacija memorije (nastavak)

Svrha? **Dinamička** alokacija memorije služi za kreiranje

- **polja** kod kojih **dimenzija nije unaprijed** poznata, (tu se vidi prava korist **ekvivalencije pokazivača** i **jednodimenzionalnih polja**),
- **dinamičkih** struktura podataka — na pr. **vezane liste**, **stabla**, i sl.

Dinamički objekti “**žive**” u bloku memorije koji se zove “**hrpa**” (engl. “**runtime heap**”).

Funkcije za alokaciju i dealokaciju memorije deklarirane su u datoteci zaglavlja **<stdlib.h>** (standardna biblioteka):

- **alokacija**: funkcije **malloc**, **calloc**, **realloc**,
- **dealokacija**: funkcija **free**.

Alokacija memorije — funkcija `malloc`

Memoriju možemo **dinamički** alocirati funkcijom `malloc`.

```
void *malloc(size_t n);
```

gdje je `size_t` cjelobrojni tip bez predznaka (za spremanje “**veličina**” objekata) definiran u `<stddef.h>`, a

🔴 `n` = **ukupan broj** bajtova koji treba alocirati.

Funkcija `malloc` rezervira **blok memorije** od `n` bajtova. Vraća:

🔴 **pokazivač** na rezervirani blok memorije, ili

🔴 `NULL`, ako zahtjev **nije** mogao biti ispunjen.

Vraćeni pokazivač je **generički**, tipa `void*` — **prije** upotrebe treba ga **eksplicitno konvertirati** u **potrebni tip** pokazivača (`cast` operatorom). Stvarno, **nije** nužno, ali — kao da **je**!

Alokacija memorije — funkcija `calloc`

Druga mogućnost za **dinamičku** alokaciju memorije je funkcija `calloc`.

```
void *calloc(size_t nobj, size_t size);
```

Funkcija `calloc` rezervira **blok memorije** za spremanje

- 🔴 `nobj` objekata, od kojih svaki **pojedini** objekt ima veličinu `size`, tj. **ukupan** broj rezerviranih bajtova je `nobj * size`.
- 🔴 Dodatno, **inicijalizira** cijeli rezervirani prostor na **nule**, preciznije, na **nul-znakove** (`'\0'`).

Kao i `malloc`, vraća **pokazivač** na rezervirani blok, ili `NULL`.

Dinamička alokacija memorije (nastavak)

Primjer. Alokacija memorije za 128 objekata tipa double.

```
double *p;  
...  
p = (double *) malloc(128 * sizeof(double));  
if (p == NULL) {  
    printf("Greska: alokacija nije uspjela!\n");  
    exit(EXIT_FAILURE);    // exit(1);  
}
```

Može i ovako, s inicijalizacijom na nule:

```
p = (double *) calloc(128, sizeof(double));
```

Funkcija `exit` — ponavljanje

Napomena. Kod dinamičke alokacije memorije **uvijek** treba provjeriti je li alokacija **uspjela** ili **ne**: `if (p == NULL) ...`

Ako **nije** uspjela, obično nemamo što dalje raditi, već treba “**uredno**” **prekinuti** izvršavanje programa. Za to koristimo funkciju `exit` deklariranu u `<stdlib.h>` (v. **Prog 1**).

```
void exit(int status);
```

Poziv `exit(status);` **zaustavlja** izvršavanje programa i vrijednost `status` predaje **operacijskom sustavu**, tj.

☛ radi **isto** što i `return status;` u funkciji `int main`, s tim da funkciju `exit` možemo koristiti u bilo kojoj funkciji.

Standardno, `status` $\neq 0$ signalizira **grešku**.

Dealokacija memorije — funkcija free

Alociranu memoriju, nakon upotrebe, možemo osloboditi funkcijom `free`.

```
void free(void *p);
```

Funkcija `free` uzima pokazivač `p` na početak alociranog bloka memorije i oslobađa taj blok memorije.

Ako je `p == NULL`, onda ne radi ništa!

Oprez: funkcija `free` ne mijenja pokazivač `p`. Nakon poziva `free(p)`; taj pokazivač `p` i dalje pokazuje na (sad) oslobođeni dio memorije i taj dio memorije (tj. `*p`) se ne smije koristiti.

Najbolje je odmah iza poziva `free(p)`; staviti i `p = NULL;`.

Dinamičko kreiranje polja

Primjer. Program **dinamički** “kreira” **polje a** cijelih brojeva tipa **int**, s tim da se **broj n** elemenata polja **učitava**. Ispisuje **zbroj** svih elemenata u polju (v. **dinpolje.c**).

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

int main(void)
{
    int *a;    /* pokazivac na dinamičko polje */
    int i, n, zbroj;

    printf("Upisi broj elemenata polja a:");
    scanf("%d", &n);
```

Dinamičko kreiranje polja (nastavak)

```
if ((a = (int*) calloc(n, sizeof(int)))
    == NULL) {
    printf("Alokacija nije uspjela.\n");
    exit(EXIT_FAILURE); } // exit(1);
for (i = 0; i < n; ++i) {
    printf("Upisi element polja a[%d]: ", i);
    scanf("%d", &a[i]); }
zbroj = 0;
for (i = 0; i < n; ++i)
    zbroj = zbroj + a[i];
printf("Zbroj svih elemenata = %d\n", zbroj);

free(a); /* a = NULL; ne treba, gotovi smo. */
return 0; }
```

Realokacija memorije — funkcija `realloc`

Treća mogućnost za **dinamičku** alokaciju memorije je funkcija **`realloc`**. Služi za **promjenu veličine** već alociranog bloka.

```
void *realloc(void *p, size_t size);
```

Funkcija **`realloc`** **mijenja veličinu** objekta na kojeg pokazuje **`p`** na **zadanu** veličinu **`size`** (tj. “realocira memoriju”).

- ➊ Sadržaj objekta (**`*p`**) ostaje **isti** do **minimuma** stare i nove veličine (kopira se, po potrebi).
- ➋ Ako je **nova** veličina objekta **veća** od stare, dodatni prostor se **ne inicijalizira**.

Vraća **pokazivač** na **novorezervirani** prostor, ili **`NULL`**, ako zahtjev **nije** ispunjen (i tada **`*p`** ostaje **nepromijenjen**).

Realokacija memorije — funkcija realloc

Stvarna **svrha** funkcije **realloc** je **produljavanje** dinamičkih objekata (polja/nizova) **nepoznate** duljine.

U prethodnom primjeru za **calloc/malloc**,

- 🕒 **prvo** smo učitali **duljinu** niza **n**, a zatim smo **odjednom** rezervirali memoriju za **cijeli** niz.

Ako broj članova niza **ne znamo unaprijed**, već čitamo

- 🕒 **nepoznati** broj članova, do neke oznake za **kraj niza**,
- 🕒 za svaki **novi** član niza, funkcijom **realloc** **produljujemo** postojeći niz za po **jedan** član, počev od praznog niza.

Primjer za takvo kreiranje niza **brojeva** je na **vježbama**.

Isti princip možemo iskoristiti kod kreiranja **rječnika** (niza stringova) — kad dođemo na **sortiranje rječnika**.

Pokazivači — ponavljanje i primjeri

Pokazivači i polja — ponavljanje

Ime polja je konstantni pokazivač na prvi element polja, tj.

📍 ime polja je adresa elementa tog polja s indeksom [0].

Primjer: Imenu polja ne smije se mijenjati vrijednost.

```
int a[10], b[10];
```

```
...
```

```
a = a + 1; /* GRESKA, a je konst. pokazivac */
```

```
b = a;      /* GRESKA! */
```

Ova veza između jednodimenzionalnih polja i pokazivača
(polje je pokazivač na prvi element polja)

📍 vrijedi i u obratnom smjeru!

Aritmetika pokazivača

Ako je **a** neko polje, onda je: **a = &a[0]** ili ***a = a[0]**.

Dodatno, za **svaki element** polja **a** vrijedi veza:

 **a + i = &a[i]** ili ***(a + i) = a[i]**, za svaki **i**.

Primjer. Aritmetika pokazivača (v. Prog 1):

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

Aritmetika pokazivača (nastavak)

Primjer. Aritmetika pokazivača i indeksi elemenata u jednodimenzionalnom polju (v. Prog 1):

```
int a[10], *pa;  
...  
pa = &a[0];  
*(pa + 3) = 20; /* ekviv. s a[3] = 20; */  
*(a + 1) = 10; /* ekviv. s a[1] = 10; */
```

Malo kasnije ćemo napraviti sličnu vezu pokazivača i indeksa za višedimenzionalna polja.

Operacije nad pokazivačima (nastavak)

Primjer:

```
#include <stdio.h>
int main(void) {
    float x[] = {1.0, 2.0}, *px = x;  /* &x[0] */
    printf("Vrijednosti: x[0] = %g, x[1] = %g\n",
           x[0], x[1]);
    printf("Adrese      : x[0] = %p, x[1] = %p\n",
           px, px + 1);
    return 0; }
```

Izlaz:

```
Vrijednosti: x[0] = 1, x[1] = 2
Adrese      : x[0] = 0012FE80, x[1] = 0012FE84
```

Važnost prioriteta i asocijativnosti

Primjer. Unarni operatori `&`, `*`, `++` i `--` imaju viši prioritet od aritmetičkih operatora i operatora pridruživanja.

```
*px += 1;    /* ili samo izraz *px + 1 */
```

Prvo djeluje `*`. Zato se povećava za jedan

📍 vrijednost na koju `px` pokazuje (`*px`), a ne sam pokazivač.

Zbog asocijativnosti unarnih operatora $D \rightarrow L$, isti izraz možemo napisati kao

```
++*px    /* povecava *px */
```

(prvo dereferenciranje, pa inkrementiranje, pa iskoristi povećani `*px`).

Važnost prioriteta i asocijativnosti (nastavak)

Kod **postfiks** notacije operatora **inkrementiranja**,

- ako želimo **povećati** ili **smanjiti sadržaj**, moramo koristiti **zagrade**.

```
(*px)++    /* povecava *px */
```

Izraz bez zagrada

```
*px++    /* povecava pokazivac px */
```

inkrementira **pokazivač px**, i to **nakon** što iskoristi ***px** (**vrijednost** na koju **px** pokazuje).

Operacije nad pokazivačima — primjer 1

```
#include <stdio.h>

int main(void)
{
    double x[] = {10.0, 20.0, 30.0, 40.0}, *px = x;

    printf("%g, %g\n", *px+1, *px+2); /* 11, 12 */
    printf("%g, %g\n", *px++, *px+2); /* 10, 22 */
    /* gcc (Code::Blocks): 10, 12 */
    printf("%g, %g\n", *px, *px+2); /* 20, 22 */
    return 0;
}
```

Probajte za polje `double x[] = {1.0, 2.0, 3.0, 4.0};`.

Operacije nad pokazivačima — primjer 2

```
#include <stdio.h>

int main(void)
{
    int x[3] = {10, 20, 30}, *px;
    px = x;                               /* adresa 0012FE80 */

    printf("%d, %p\n", *px+1, px);         /* 11, 0012FE80 */
    printf("%d, %p\n", ++*px, px);         /* 11, 0012FE80 */
    printf("%d, %p\n", *px++, px);         /* 11, 0012FE84 */
    printf("%d, %p\n", *px, px);           /* 20, 0012FE84 */
    return 0;
}
```

Usporedba pokazivača — primjer

Sljedeće **dvije** petlje su **ekvivalentne**:

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

```
...
```

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

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

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

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

Polje kao argument funkcije (ponavljanje)

Primjer. Polje kao argument funkcije i **dozvoljeni** pozivi.

```
char z[100];  
void f(char *);  
...  
f(z);          /* isto sto i f(&z[0]); */  
f(&z[50]);
```

U zadnjem pozivu, funkcija **f** dobiva **z[50]** kao “**prvi radni**” element polja — funkcija ga **vidi** kao element s indeksom **[0]**.

Oprez: funkcija “**nema pojma**” o stvarnoj **duljini** polja (nema kontrole **indeksa** ili **broja** elemenata) — moramo sami **paziti** da ne “gazimo” po memoriji.