

Programiranje (C)

5. predavanje

Saša Singer

singer@math.hr
web.math.hr/~singer

PMF – Matematički odjel, Zagreb

Sadržaj predavanja

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

Višedimenzionalna polja

Operacije s matricama i vektorima

Sadržaj

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

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, “programske” — treba “zavrtiti” **dvije** petlje.

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. */
    int i, j;

    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 $m \times l$,
- B — tipa $l \times n$,
- 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 matrične 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, “programske” — treba “zavrtiti” tri petlje.

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` dimenziye (brojevi stupaca) iz definicije ovih matrica — tamo gdje je rezervirana memorija za njih,
- stvarne dimenziye 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 (**blok** transfer u “**cache**”).
- U unutarnjoj petlji (po **k**) — **redak** matrice **A** množi se **stupcem** matrice **B** (skalarni produkt). **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**.
- 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.)

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:

- dimenzijs polja u deklaraciji argumenata funkcije moraju biti konstantni izrazi.

Standard C99 uvodi polja varijabilne duljine, ali ih mnogi prevoditelji još u potpunosti ne implementiraju.

- Polje varijabilne duljine je automatsko polje čije dimenzijs mogu biti zadane varijablama.

Primjer:

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

Polja varijabilne duljine (nastavak)

Osnovna upotreba polja varijabilne duljine je

- pisanje funkcija koje kao argument imaju dimenzijske polja.

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 (nastavak)

```
double E_norm(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);
}
```

Ovaj poredak petlji je **dobar** — elementima matrice A pristupamo **redom** kako su spremljeni u memoriji.

Pokazivači

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) čija **veličina** ovisi o **tipu varijable**.
- Svakoj **memorijskoj lokaciji** pridružena je **jedinstvena adresa**.
- Radi jednostavnosti, možemo zamišljati da je
 - **adresa** čitavog **bloka lokacija** = **adresa** **prve lokacije** u **tom bloku**.Tako svaka **varijabla** ima svoju **jedinstvenu adresu**.
- **Varijabli** se može pristupiti korištenjem
 - **imena** **variable** — prevoditelj “zna” adresu,
 - **adrese** **variable** — **pokazivačem** na tu **varijablu**.

Deklaracija pokazivača

Pokazivač na tip je varijabla koja sadrži adresu variable tog tipa tip.

Deklaracija pokazivača (engl. “pointer”):

```
mem_klasa tip *p_var;
```

Dakle, svi pokazivači imaju tip — onog na što pokazuju, osim tzv. generičkog pokazivača void *p — pokazivača na bilo što.

Primjer:

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

Adresni operator & i operator dereferenciranja *

Adresni operator **&**:

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

Operator dereferenciranja *****:

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

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

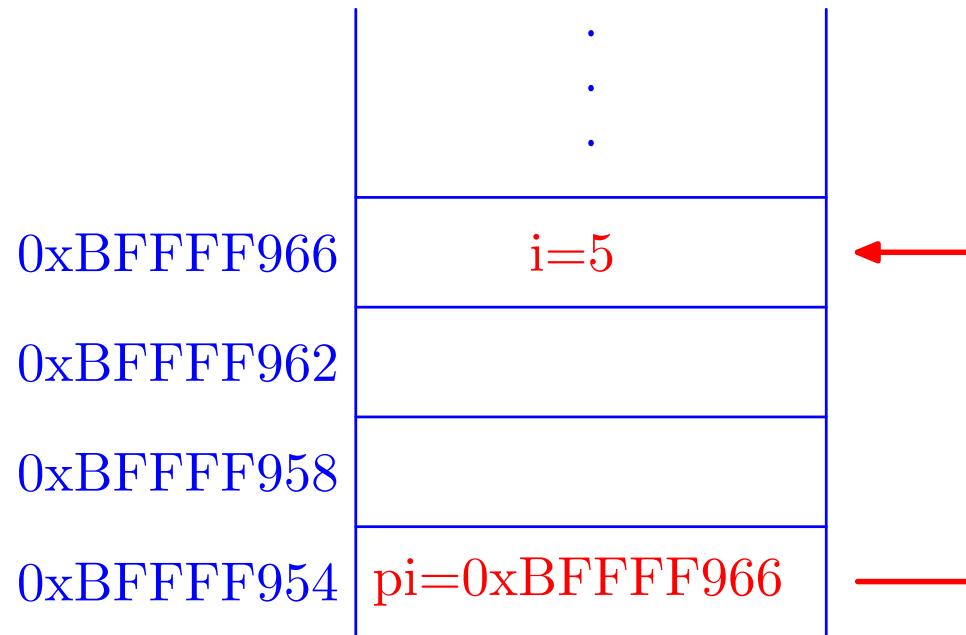
Napomena. Simboli **&** i ***** mogu biti i **binarni** operatori (bit-po-bit i, množenje) — **ovisno** o tome kako su **napisani**:

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

*Adresni operator & i operator dereferenciranja **

Uočite da je:

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



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 primjeni **adresni** operator (mora imati adresu).

Primjer:

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

Izlaz: **i = 22, adresa od i= 0012FF64.**

Pokazivači i funkcije

Pokazivači mogu biti argumenti funkcije.

- U tom slučaju funkcija može promijeniti vrijednost varijable na koju pokazivač pokazuje.

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 (u glavnom programu) treba glasiti:

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

Pokazivači i funkcije (nastavak)

Ponovimo da sljedeći kôd za funkciju zamjena ne valja:

```
void zamjena(int x, int y) {  
    int temp = x;  
    x = y;  
    y = temp; }
```

Razlog je prijenos argumenata po vrijednosti. Pri pozivu

```
zamjena(a, b);
```

funkcija dobiva kopije vrijednosti stvarnih argumenata a i b, a onda međusobno zamjenjuje te kopije.

To nema nikakvog utjecaja na originalne vrijednosti stvarnih argumenata a i b — one ostaju nepromijenjene.

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.

Operacije nad pokazivačima

Sve aritmetičke operacije nad pokazivačima ekvivalentne su

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

Svakom pokazivaču možemo dodati i oduzeti cijeli broj.

Primjer. Ako je `px` pokazivač i `n` varijabla tipa `int`, dozvoljene su operacije:

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

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`

$+ n * \text{sizeof}(\text{tip objekta na koji pokazuje } px)$.

Aritmetika pokazivača

Ove operacije skraćeno zovemo aritmetika pokazivača.

Osnovna veza imedju pokazivača i polja:

- Ime polja je konstantni pokazivač na prvi element polja, tj. adresa elementa tog polja s indeksom [0]!

Primjer. Aritmetika pokazivača (v. UuR):

```
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. UuR):

```
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]=0012FF44, x[1]=0012FF48

Važnost prioriteta i asocijativnosti

Primjer. Unarni operatori `&`, `*`, `++` i `--` imaju viši prioritet od aritmetičkih operatora i operatara 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      /* povećava *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;

    px=&x[0];
    printf("%g, %g\n", *px+1, *px+2); /* 11, 12 */
    px=&x[0];
    printf("%g, %g\n", *px++, *px+2); /* 10, 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 0012FF5C */

    printf("%d, %p\n", *px+1, px); /* 11, 0012FF5C */
    printf("%d, %p\n", ++*px, px); /* 11, 0012FF5C */
    printf("%d, %p\n", *px++, px); /* 11, 0012FF60 */

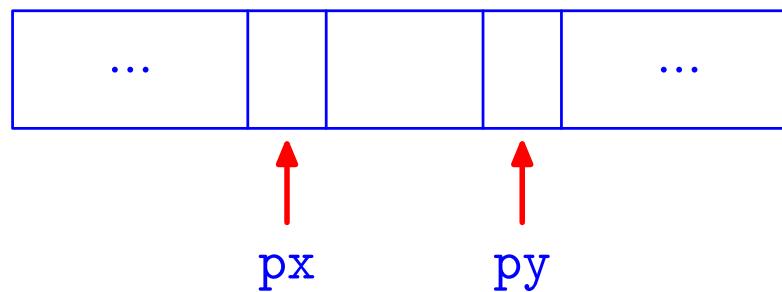
    return 0;
}
```

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
 - $py - px + 1 = \text{broj elemenata između } px \text{ i } py,$ uključujući i krajeve.

Ovaj rezultat odgovara aritmetici pokazivača!



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

Razlika pokazivača (nastavak)

Primjer. Broj elemenata **ispred** nule u polju.

```
int d(int a[]) {  
    int *pa = a;  
    while (*pa != 0) ++pa;  
    return pa - a; /* razlika pokazivaca */  
}
```

Ako je

```
int x[5] = {3, 2, 0, 0, 4};  
...  
printf("%d\n", d(x));
```

izlaz je **2**. Završna vrijednost pokazivača **pa** je **&x[2]**.

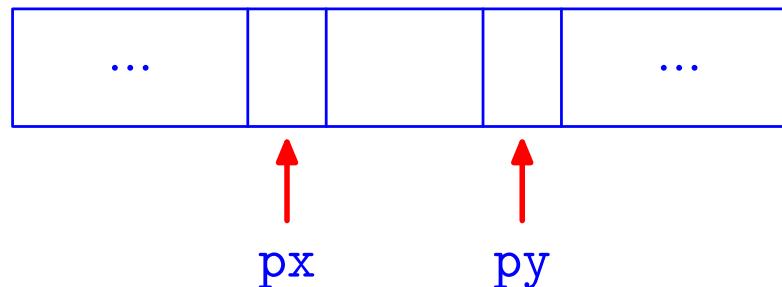
Usporedba pokazivača

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

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

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

px < py px > py px == py px != py



Usporedba pokazivača (nastavak)

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.

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 */

Generički pokazivač

Pokazivači na različite tipove podataka općenito se ne mogu pridruživati.

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

Primjer:

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

Generički pokazivač (nastavak)

Generički pokazivač deklarira se kao pokazivač na **void**.

```
void *p;
```

Pokazivač na bilo koji tip može se **konvertirati** u pokazivač na **void** i **obratno**, bez promjene **vrijednosti** pokazivača.

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.

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:
 - Ime polja i pokazivač na prvi element.
 - Pokazivači i jednodimenzionalna polja.
 - Polje kao argument funkcije.
 - Indeksi i aritmetika pokazivača.
 - Jednodimenzionalno polje.
 - Višedimenzionalno polje.

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!

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 — da li smo 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';
                      ekviv. s   x[2] = 'b'; */
*(px + 130) = 'z';   /* NIJE GRESKA,
                      ali gazimo po memoriji! */
```

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

Funkciju **f** koja uzima jednodimenzionalno polje **v** tipa **tip** kao argument, možemo deklarirati na **dva** načina:

f(tip v[]) ili **f(tip *v)**

U prvom načinu **ne treba** navesti dimenziju.

Drugi način direktno kaže da je **v** pokazivač na objekt tipa **tip** i podrazumijeva se da je to **adresa** “prvog radnog” elementa polja u funkciji, tj. **v = &v[0]**.

Ako **ne želimo** da funkcija mijenja elemente polja **unutar** funkcije, onda pišemo:

f(const tip v[]) ili **f(const tip *v)**

Polje kao argument funkcije (nastavak)

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.

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) $x[1]$.

Višedimenzionalna polja kao argumenti

Pri deklaraciji višedimenzionalnog polja kao 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

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

U definiciji polja (tamo gdje se rezervira memorija), mora biti prvi oblik ili inicijalizacija.

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**”).

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 potrebnii tip pokazivača (**cast operatorom**).

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(-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. UuR).

```
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 ≠ 0** signalizira **grešku**.

Dealokacija memorije — funkcija free

Alociranu memoriju, nakon upotrebe, možemo oslobođiti 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 i dalje **pokazuje** na (sad) oslobođeni dio memorije i **ne smije** se 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.

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

int main(void)
{
    int *a;      /* pokazivac na dinamicko 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(-1);
}
for (i = 0; i < n; ++i) {
    printf("Upisi element polja: ");
    scanf("%d", &a[i]);
}
zbroj = 0;
for (i = 0; i < n; ++i)
    zbroj = zbroj + a[i];
printf("%d\n", zbroj);

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

Alokacija 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”).

- Sadržaj objekta (***p**) ostaje **isti** do **minimuma** stare i nove veličine (kopira se, po potrebi).
- 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**).