

# *Programiranje 2*

## *7. predavanje*

Saša Singer

`singer@math.hr`

`web.math.hr/~singer`

PMF – Matematički odsjek, Zagreb

# Sadržaj predavanja

- Tipovi i složene deklaracije:
  - Pokazivač i polja (ponavljanje).
  - Pokazivač na funkciju (ponavljanje).
  - Složene deklaracije — primjeri.
  - Deklaracija tipova — typedef.
- Struktura (prvi dio):
  - Deklaracija strukture. Struktura i typedef.
  - Rad sa strukturama. Operator točka.
  - Struktura i funkcije.
  - Struktura i pokazivači. Operator strelica (->).
  - Unije.
  - Polja bitova.

# Informacije

Konzultacije (službeno):

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

Programiranje 2 je u kolokvijskom razredu C3.

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

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

Dodatni bodovi “čekaju na vas”.

# Tipovi i složene deklaracije

# Sadržaj

- Tipovi i složene deklaracije:
  - Pokazivači i polja (ponavljanje).
  - Pokazivač na funkciju (ponavljanje).
  - Složene deklaracije — primjeri.
  - Deklaracije tipova — typedef.

# Polje pokazivača i pokazivač na polje

Polje pokazivača ima deklaraciju:

```
tip_pod *ime[izraz];
```

Napomena: **Primarni** operator `[ ]` ima viši prioritet od **unarnog** operatora `*`.

**Primjer.** Razlikujte **polje** pokazivača (ovdje 10 pokazivača):

```
int *ppi[10];
```

od **pokazivača na polje** (ovdje od 10 elemenata):

```
int (*ppi)[10];
```

# Pokazivač na funkciju

Pokazivač na funkciju deklarira se kao:

```
tip_pod (*ime)(tip_1 arg_1, ..., tip_n arg_n);
```

Ovdje je *ime* varijabla tipa — *pokazivač* na *funkciju* koja

- uzima *n* argumenata tipa *tip\_1* do *tip\_n*

- i vraća vrijednost tipa *tip\_pod*.

Slično kao i u *prototipu* funkcije, *ne treba* pisati *imena* argumenata *arg\_1* do *arg\_n*.

Primjer:

```
int (*pf)(char c, double a);  
int (*pf)(char, double);
```

## Pokazivač na funkciju (nastavak)

U deklaraciji **pokazivača** na **funkciju** — **zgrade** su nužne.

- **Primarni** operator **( )** — “poziva” ili argumenata funkcije, ima viši prioritet od **unarnog** operatora **\***.

**Primjer.** Razlikujte **funkciju** koja vraća **pokazivač** na neki tip (ovdje na **double**):

---

```
double *pf(double, double);  
double *(pf(double, double)); /* Isto */
```

---

od **pokazivača** na **funkciju** koja vraća **vrijednost** nekog tipa (ovdje **double**):

---

```
double (*pf)(double, double);
```

---



## *Pokazivač na funkciju — primjeri*

Primjeri **pokazivača** na **funkciju** *f* iz integracije (prošli put):

---

```
double integracija(double, double,  
                  double (*)(double));
```

```
double integracija(double a, double b,  
                  double (*f)(double)) {  
    return 0.5 * (b - a) * ( (*f)(a) + (*f)(b) );  
}
```

---

ili:

---

```
double integracija(double, double, int,  
                  double (*)(double));
```

...

---

## Složene deklaracije — primjeri

Pri interpretaciji deklaracije uzimaju se u obzir **prioriteti** i **asocijativnost** pojedinih operatora. Ti prioriteti **moгу se promijeniti** upotrebom **zagrada**.

**Primjeri.** Što je **p** u sljedećim deklaracijama?

---

<code>int *p[10];</code>	<code>/* polje od 10 ptr na int */</code>
<code>int *p(void);</code>	<code>/* funkcija koja nema arg i vraca pokazivac na int */</code>
<code>int p(char *a);</code>	<code>/* funkcija koja uzima ptr na char i vraca int */</code>
<code>int *p(char *a);</code>	<code>/* funkcija koja uzima ptr na char i vraca ptr na int */</code>
<code>int (*p)(char *a);</code>	<code>/* ptr na funkciju koja uzima ptr na char i vraca int */</code>

## *Složene deklaracije — primjeri (nastavak)*

```
int (*p(char *a))[10];    /* funk. uzima ptr na char
                           i vraca ptr na polje
                           od 10 elt tipa int */
int p(char (*a)[]);       /* funk. uzima ptr na polje
                           znakova i vraca int */
int (*p)(char (*a)[]);    /* ptr na funk. koja uzima
                           ptr na polje znakova i
                           vraca int */
int *(*p)(char (*a)[]);   /* ptr na funk. koja uzima
                           ptr na polje znakova i
                           vraca ptr na int */
int *(*p[10])(char *a);    /* polje 10 ptr na funk.
                           koja uzima ptr na char
                           i vraca ptr na int */
```

# Deklaracije tipova — typedef

# Ključna riječ typedef

Korištenjem ključne riječi typedef

- ☛ postojećim tipovima podataka dajemo nova imena (ne kreiramo nove objekte ili nove tipove!).

Jednostavni oblik typedef deklaracije je:

---

```
typedef tip_podatka novo_ime_za_tip_podatka;
```

---

To znači da:

- ☛ novo\_ime\_za\_tip\_podatka postaje sinonim za tip\_podatka

i smije se tako koristiti u svim kasnijim deklaracijama — tamo gdje smijemo napisati jedno, smijemo napisati i drugo, i to s istim značenjem.

# Jednostavne typedef deklaracije

Primjer. Deklaracijom

```
typedef double Masa;
```

identifikator **Masa** postaje **sinonim** za **double**.

Nakon toga, varijable tipa **double** možemo deklarirati i kao:

```
Masa m1, m2, *pm1;  
Masa elementi[10];
```

Uočite da je

• **pm1** — **pokazivač** na **double**,

• **elementi** — **polje** od **10** elemenata tipa **double**.

Međutim, nije baš jasno što smo s tim “**dobili**”!

# Svrha deklaracije tipova

Zaista, kod ovako **jednostavnih** deklaracija — svrha se **ne** vidi odmah.

Stvarna **svrha** deklaracije ili **imenovanja tipova** je:

- lakše **razumijevanje** (čitanje) kôda i
- **dokumentiranje** programa.

To postaje **vrlo korisno** kod **složenijih** tipova podataka — kad u programu koristimo

- čitavu **hijerahiju** tipova — koji se grade jedni iz drugih.

Korist će se vidjeti vrlo **skoro**, kad dođemo na

- **strukture** i **samoreferencirajuće strukture** (vezane liste, binarna stabla i sl.).

# Primjer jednostavne deklaracije tipova

Korist od deklaracije **tipova** može se vidjeti i na **jednostavnim** primjerima — ako **dobro** izaberemo **ime** za **tip**.

Primjer.

```
typedef int Metri, Kilometri;  
Metri duljina, sirina;  
Kilometri udaljenost;
```

Ideja (ili svrha): ovdje **ime tipa** sugerira **jedinice** u kojima su izražene određene vrijednosti!

No, stvarna **korist** od **typedef** je tek kod **složenijih** tipova.

🔴 Kako se **pišu** takve deklaracije?



## *Složenije typedef deklaracije*

Sasvim općenito, deklaracija *imena* za složeniji *tip*:

- *počinje* s *typedef*, a
- dalje ima *isti* oblik kao i deklaracija *variable* tog *imena* i tog *tipa*.

Sve je isto, osim što tada

- *ime nije* varijabla tog *tipa* (ne dobiva memorijski prostor i adresu), već
- *ime* postoje *sinonim* za *tip* kojeg “*bi imala*” takva varijabla.

---

```
typedef  deklaracija_za_tip_podatka;
```

---

## Primjer složenije deklaracije tipa — za polja

**Primjer.** Uvedimo imena tipova za vektore i matrice odgovarajućih dimenzija (recimo,  $n = 10$ ).

```
#define n 10
typedef double skalar;
typedef skalar vektor[n];
typedef vektor matrica[n];
```

Zadnje dvije deklaracije daju imena poljima:

- **vektor** je ime tipa za polje od  $n$  (10) skalara (**double**),
- **matrica** je ime tipa za polje od  $n$  (10) vektora, tj.
- **matrica** je dvodimenzionalno polje skalara, ili sinonim za tip `double[n][n] = double[10][10]`.

## typedef *i polja (nastavak)*

Funkciju za računanje **produkta**  $y = Ax$ , kvadratne matrice  $A$  i vektora  $x$ , možemo i ovako napisati:

```
void prod_mat_vek(matrica a, vektor x, vektor y)
{
    int i, j;
    for (i = 0; i < n; ++i) {
        y[i] = 0;
        for (j = 0; j < n; ++j)
            y[i] += a[i][j] * x[j];
    }
}
```

**Napomena.** Ovdje je **n fiksni** — **n = 10**. Popravite funkciju tako da **stvarni red** matrice i vektora bude **argument** funkcije.

## Primjer deklaracije tipa — stringovi

Primjer. Kod obrade **stringova** možemo uvesti deklaraciju

```
typedef char *string;
```

Ovdje je **string** sinonim za **pokazivač** na **char** (tip **char \***), s **očitom** svrhom:

- 📍 taj **pokazivač** interpretiramo kao pokazivač na **prvi** element u **polju** znakova,
- 📍 a to **polje** znakova obrađujemo kao **string** (do nul-znaka)!

Funkcija **strcmp** za **uspoređivanje** **stringova** smije se ovako deklarirati:

```
int strcmp(string, string);
```

## typedef *i pokazivači (nastavak)*

Primjer. Pokazivač na `double` nazvat ćemo `Pdouble`.

```
typedef double *Pdouble;
```

`Pdouble` postaje pokazivač na `double`, pa smijemo pisati:

```
Pdouble px;    /* = double *px */
```

```
void f(Pdouble, Pdouble);  
    /* = void f(double *, double *); */
```

```
px = (Pdouble) malloc(100 * sizeof(Pdouble));
```

## typedef *i deklaracije funkcija*

Općenito, `typedef` koristimo za **kraće** zapisivanje **složenih** deklaracija.

Primjer. Pokazivač na funkciju.

```
typedef int (*PF)(char *, char *);
```

`PF` postaje ime za pokazivač na funkciju koja uzima dva pokazivača na `char` i vraća `int`. Umjesto deklaracije:

```
void f(double x, int (*g)(char *, char *)) { ... }
```

možemo pisati:

```
void f(double x, PF g) { ... }
```

# Strukture

# Sadržaj

- **Strukture** (prvi dio):
  - Deklaracija strukture. Strukture i typedef.
  - Rad sa strukturama. Operator točka.
  - Strukture i funkcije.
  - Strukture i pokazivači. Operator strelica (->).
  - Unije.
  - Polja bitova.



# Što je struktura?

Struktura je složeni tip podataka, kao i polje. Za razliku od polja, koje služi

- grupiranju podataka istog tipa, struktura služi
- grupiranju podataka različitih tipova.

Može i ovako — malo detaljnije.

- Svi elementi polja imaju isti tip i zajedničko ime, a razlikuju se po indeksu. To se vidi i u deklaraciji polja.
- Elementi (ili članovi) strukture mogu, ali ne moraju, biti različitog tipa i svaki element ima svoje posebno ime.

Zato u deklaraciji strukture moramo navesti ime i tip svakog člana. Tip strukture možemo deklarirati na dva načina.

# Deklaracija strukture — bez typedef

Prvi način — bez `typedef`. Tip strukture deklarira se ovako:

```
struct ime {  
    tip_1 ime_1;  
    tip_2 ime_2;  
    ...  
    tip_n ime_n;  
};
```

Ovdje je `struct` rezervirana riječ, a `ime` je ime strukture.

Stvarni tip strukture je

🕒 `struct ime` — dvije riječi (i to je nezgodno!).

Unutar vitičastih zagrada popisani su članovi strukture.

# Definicija varijabli tipa strukture — bez typedef

Napomena. Kao i kod polja, članovi strukture

• smješteni su u memoriji jedan za drugim.

Kod ovakve deklaracije tipa strukture, varijable tog tipa, općenito, definiramo ovako:

---

```
mem_klasa struct ime var_1, var_2, ..., var_n;
```

---

• var\_1, var\_2, ..., var\_n su varijable tipa struct ime.

## Primjer — struktura za točke

**Primjer.** Struktura `točka` definira točku u ravnini. Uzmimo da `točka` ima cjelobrojne koordinate, poput `pixela` na ekranu.

```
struct točka {  
    int x;  
    int y;  
};
```

**Variable** tipa strukture `točka` možemo definirati na (barem) **dva** načina.

Nakon gornje deklaracije **strukture** `točka` (kao tipa), napišemo “običnu” definiciju **varijabli**:

```
struct točka t1, t2;
```

## Primjer — struktura za točke (nastavak)

Deklaraciju **tipa** strukture **ne** moramo napisati **posebno**.

Možemo ju napisati i **u sklopu** definicije **varijabli** tog tipa:

---

```
struct tocka {  
    int x;  
    int y;  
} t1, t2;
```

---

**Prvi** način je **pregledniji**!

Međutim, postoji i **bolji** način deklaracije **tipa strukture**, koji olakšava i definiciju **varijabli** tog tipa — preko **typedef**.

**Prednost**: tako možemo **izbjeći** stalno navođenje riječi **struct** u deklaracijama varijabli.

# Deklaracija strukture — preko typedef

Drugi način — preko `typedef`.

Tip strukture deklarira se ovako:

```
typedef struct ime {  
    tip_1 ime_1;  
    tip_2 ime_2;  
    ...  
    tip_n ime_n;  
} ime_tipa;
```

Ovdje smo još, na **kraju** deklaracije, cijelom **tipu** strukture dali ime **ime\_tipa**. Stvarni **tip** strukture je onda i

🕒 **ime\_tipa** — kao **sinonim** za **struct ime**.

Sve ostalo je isto kao i prije.

## Definicija varijabli tipa strukture — uz typedef

**Napomena.** “Prvo” ime strukture (odmah iza `struct`) smijemo i **ispustiti**, ako ga nigdje nećemo koristiti! (Uvijek bi trebalo pisati `struct` ispred tog imena.)

Ako pišemo “prvo” ime, ono **mora** biti **različito** od svih ostalih imena (identifikatora), pa i od `ime_tipa`.

📌 Običaj: prvo ime = `_ime_tipa` (na primjer, `_osoba`).

Kod ovakve deklaracije tipa strukture, **varijable** tog tipa, općenito, definiramo ovako:

---

```
mem_klasa ime_tipa var_1, var_2, ..., var_n;
```

---

📌 `var_1, var_2, ..., var_n` su varijable tipa `ime_tipa`, što je **sinonim** za `struct ime`.

## Primjer — struktura za točke

**Primjer.** Umjesto ranije definicije strukture za točku u ravnini, možemo uvesti tip **Tocka** za cijelu strukturu.

```
typedef struct {  
    int x;  
    int y;  
} Tocka;  
...  
Tocka t1, t2, *pt1;
```

Identifikator **Tocka** je ime **tipa** za cijelu strukturu, a **t1** i **t2** su varijable **tipa** **Tocka**. Što je **pt1**?

Uočite da ovdje **nismo** napisali **ime** strukture iza **struct**, jer ga nećemo ni koristiti.



# Inicijalizacija strukture

Varijablu tipa **struktura** možemo **inicijalizirati** pri definiciji (kao i svaku drugu varijablu):

---

```
mem_klasa struct ime var = {v_1, ..., v_n};  
mem_klasa ime_tipa    var = {v_1, ..., v_n};
```

---

Konstante **v\_1**, **v\_2**, ... , **v\_n** pridružuju se navedenim **redom**  
🕒 odgovarajućim članovima strukture **var** — **član, po član.**

# Inicijalizacija strukture — primjer

Primjer. Ako je definirana **struktura**

```
struct racun {  
    int broj_racuna;  
    char ime[80];  
    float stanje;  
};
```

onda **varijablu kupac** možemo **inicijalizirati** ovako:

```
struct racun kupac = { 1234, "Pero Bacilova",  
                      -12345.00f };
```

# Inicijalizacija polja struktura

Primjer. Slično se može inicijalizirati i čitavo polje struktura:

```
struct racun kupci[] = {  
    2234, "Goga",    456.00f,  
    1235, "Josip",  -234.00f,  
    436, "Martina",  0.00f };
```

# Operator točka — pristup članu strukture

Članovima strukture može se pojedinačno pristupiti korištenjem primarnog operatora točka (.

- Operator točka (.) separira ime varijable i ime člana te strukture.

Ako je var varijabla tipa strukture koja sadrži član memb, onda je

---

```
var.memb
```

---

član memb u strukturi var (preciznije, vrijednost tog člana).

Napomena. Ime člana je lokalno za svaku strukturu. Zato smijemo koristiti

- isto ime člana u raznim strukturama.

# Prioritet i asocijativnost operatora točka

Operator **točka** (.)

- spada u **najvišu** prioritetnu grupu (**primarni** operatori) i ima asocijativnost  $L \rightarrow D$ .

Zbog **najvišeg** prioriteta vrijedi:

- $++\text{varijabla.clan} \iff ++(\text{varijabla.clan})$
- $\&\text{varijabla.clan} \iff \&(\text{varijabla.clan})$

Član **struktura** (kao i element polja), naravno,

- smije pisati na **lijevoj** strani naredbe **pridruživanja**.

# Rad sa strukturama — pristup članovima

Primjer. Pristup članovima strukture.

```
struct tocka {  
    int x;    /* prvi clan strukture */  
    int y;    /* drugi clan strukture */  
};  
struct tocka ishodiste;
```

Imena objekata i značenje:

- 📍 `ishodiste` je **varijabla** tipa `struct tocka`,
- 📍 `ishodiste.x` je **prvi** član (ili prva komponenta) varijable `ishodiste`,
- 📍 `ishodiste.y` je **drugi** član (ili druga komponenta) varijable `ishodiste`.

## *Pristup članovima strukture (nastavak)*

Primjer. Ako je

```
struct racun {  
    int broj_racuna;  
    char ime[80];  
    float stanje;  
} kupac = { 1234, "Pero Bacilova", -12345.00f };
```

tada je, redom:

```
kupac.broj_racuna = 1234,  
kupac.ime = "Pero Bacilova",  
kupac.stanje = -12345.00f.
```

# Struktura definirana pomoću strukture

Strukture mogu sadržavati druge strukture kao članove.

Primjer. Pravokutnik paralelan koordinatnim osima možemo zadati parom dijagonalno suprotnih vrhova — na pr. donjim lijevim (pt1) i gornjim desnim (pt2). Vrhovi su točke.

```
struct pravokutnik {  
    struct tocka pt1;    /* ili Tocka pt1; */  
    struct tocka pt2;    /* ili Tocka pt2; */  
};
```

Deklaracija strukture tocka mora prethoditi deklaraciji strukture pravokutnik.

U različitim strukturama mogu se koristiti ista imena članova.



## Polje kao član strukture

Kada **struktura** sadrži **polje** kao član strukture, onda se elementima polja (zovimo ga **clan**) pristupa izrazom:

---

```
varijabla.clan[izraz]
```

---

Koristi se asocijativnost  $L \rightarrow D$  za **primarne** operatore

- **točka** ( **.** ) i
- **indeksiranje** polja ( **[]** ).

## *Polje kao član strukture — primjer*

### Primjer.

---

```
typedef struct {
    int broj_racuna;
    char ime[80];
    float stanje;
} Racun;
Racun kupac = { 1234, "Pero Bacilova",
                -12345.00f };

...
if (kupac.ime[0] == 'P') puts(kupac.ime);
```

---

# Polje struktura

Ako imamo **polje struktura**, onda za pojedini **element** polja, **članu** pripadne strukture pristupamo izrazom

---

```
polje[izraz].clan
```

---

Asocijativnost je bitna, jer su svi operatori istog prioriteta.

Primjer.

---

```
struct tocka {  
    int x;  
    int y;  
} vrhovi[1024] ;  
  
...  
if (vrhovi[17].x == vrhovi[17].y) ...
```

---

# Strukture — operacije i funkcije

Dozvoljene operacije nad strukturom kao cjelinom su:

- pridruživanje,
- uzimanje adrese, primjena sizeof operatora.

Napomena. Nije dozvoljeno uspoređivanje struktura.

Strukture i funkcije:

- Struktura može biti argument funkcije. Funkcija tada dobiva kopiju strukture kao argument.
- Funkcija može vratiti strukturu.

## Strukture i funkcije — primjer

**Primjer.** Argumenti funkcije `suma` su dvije strukture tipa `tocka`, a funkcija vraća `sumu` argumenata (tipa `tocka`).  
Suma točaka = zbroj odgovarajućih koordinata (kao vektori).

---

```
struct tocka {
    int x;
    int y;
} t, ishodiste = {0, 0}, t1 = {1, 7};

struct tocka suma(struct tocka p1,
                  struct tocka p2) {
    p1.x += p2.x;
    p1.y += p2.y;
    return p1;
}
```

## Strukture i funkcije — primjer (nastavak)

```
...  
    /* Dodjeljivanje struktura:  
       t i ishodiste moraju biti istog tipa */  
  
t = ishodiste;  
  
printf("%d\n", sizeof(t));  
  
    /* Zbroj tocaka. */  
  
t1 = suma(t1, t1);  
printf("t1 = (%d, %d)\n", t1.x, t1.y);
```

---

# Strukture i funkcije — kompleksni brojevi

Primjer. Biblioteka `funkcija` za osnovne operacije s kompleksnim brojevima.

---

```
typedef struct {  
    double re;    /* ili x */  
    double im;    /* ili y */  
} complex;  
  
/* Napomena: cabs vec postoji u <math.h>! */  
double zabs(complex a) {  
    return sqrt( a.re * a.re + a.im * a.im );  
}
```

---

U C99 standardu postoje tipovi i odgovarajuće funkcije za kompleksne brojeve (zaglavlje `<complex.h>`).

# Strukture i pokazivači

Pokazivač na strukturu definira se isto kao i pokazivač na druge tipove objekata.

```
struct tocka {  
    int x;  
    int y;  
} p1, *pp1;
```

...

```
pp1 = &p1;  
(*pp1).x = 13;  
(*pp1).y = 27;
```

```
*pp1.x = 13;  /* GRESKA */  
              /* *pp1.x je isto sto i *(pp1.x) */
```



# Operator strelica (->)

Primarni operator **strelica** (->) nudi jednostavan način dohvaćanja **člana** strukture preko **pokazivača** na tu strukturu.

☞ **Asocijativnost** operatora -> je  $L \rightarrow D$ .

Ako je **ptvar** **pokazivač** na strukturu, a **clan** jedan **član** te strukture, onda je:

$$ptvar \rightarrow clan \iff (*ptvar).clan$$

**Primjer.**

---

```
struct tocka p1, *pp1 = &p1;  
pp1->x = 13;  
pp1->y = 27;
```

---

## Složeni izrazi

Pristup **koordinatama** vrhova **pravokutnika** **r** — **izravno** i preko **pokazivača** **pr**.

```
struct pravokutnik {  
    struct tocka pt1;  
    struct tocka pt2;  
} r, *pr = &r;
```

Sljedeći su izrazi **ekvivalentni** (**x**-koordinata prvog vrha **pt1**):

<code>r.pt1.x</code>	<code>/* operatori . i -&gt; */</code>
<code>pr-&gt;pt1.x</code>	<code>/* imaju isti prioritet */</code>
<code>(r.pt1).x</code>	<code>/* i asocijativnost */</code>
<code>(pr-&gt;pt1).x</code>	<code>/* L -&gt; D */</code>

# Unije

# Unija

Unija je složeni tip podataka sličan strukturi, jer sadrži

- članove različitog tipa.

Gdje je razlika?

Članovi strukture su

- smješteni u memoriji jedan za drugim.

Za razliku od toga, svi članovi unije

- počinju na istom mjestu u memoriji — na istoj lokaciji, tj. dijele jedan zajednički dio memorije (na početku), ovisno o veličini članova unije.

Ukupna rezervirana memorija za varijablu tipa unije

- dovoljno je velika da u nju stane “najveći” član unije.

# Svrha unije i rad s unijama

**Ideja:** taj **zajednički** dio memorije možemo interpretirati

- na **razne** načine — kao vrijednost **različitih** tipova.

Zato i ime — **unija** tipova!

**Napomena.** Osnovna svrha unije **nije**

- ušteta** memorijskog prostora,

iako se može koristiti i za to.

Osim navedene **razlike** između **unija** i **strukture** u **rezervaciji** memorije, sve ostalo u **C**-u je potpuno **isto**, samo

- umjesto ključne riječi **struct** za **strukture**,
- pišemo ključnu riječ **union** za **unije**.

# Deklaracija unije

Deklaracija **tipa unije** ima **isti** oblik kao i za **tip strukture** — umjesto **struct**, pišemo **union**.

```
union ime {  
    tip_1 ime_1;  
    ...    ...  
    tip_n ime_n;  
};
```

Kao i kod struktura, **bolje** je koristiti **typedef** za deklaraciju tipa unije.

Varijable **x** i **y** tipa ove unije mogu se deklarirati ovako:

```
union ime x, y;
```

# Unija (nastavak)

## Primjer.

```
union pod {  
    int i;  
    float x;  
} u, *pu;
```

Ovdje su:

- `u.i` i `pu->i` — varijable tipa `int`.
- `u.x` i `pu->x` — varijable tipa `float`.

Član `i` (tipa `int`) i član `x` (tipa `float`)

- `počinju` na `istoj` lokaciji u memoriji.

Standardno zauzimaju po 4 bajta, tj. “`dije`le” `isti` prostor!

## Unija (nastavak)

Primjer. Uniju možemo iskoristiti za ispis

- “binarnog” (preciznije, **heksadecimalnog**) oblika prikaza realnog broja tipa **float** u računalu.

---

```
u.x = 0.234375f;  
printf("0.234375 binarno = %x\n", u.i);
```

---

Za pravi “binarni” prikaz možemo iskoristiti algoritam s Prog1

- koji ispisuje binarni prikaz cijelog broja.



## Primjer — binarni prikaz realnog broja

**Primjer.** Napisati program koji učitava **realni** broj tipa **double** i piše **binarni prikaz** tog broja u računalu (v. **p\_double.c**).

Broj tipa **double** standardno zauzima 8 bajtova = 64 bita. Taj prostor “gledamo” kao

- **polje** od 2 cijela broja tipa **int** (= 2 “riječi”).

Još jedna “sitnica” — bitovi u **IEEE** prikazu za **double** imaju sljedeći **raspored** po bajtovima (na **IA-32**):

- 1. bajt = bitovi 7 – 0 (donji bitovi),
- 2. bajt = bitovi 15 – 8,
- ...
- 8. bajt = bitovi 63 – 56 (gornji bitovi).

# Binarni prikaz realnog broja — program

Početak programa s globalnom deklaracijom tipa unije za

- jedan `double` i
- polje od 2 `int`-a.

---

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

```
typedef union {  
    double d;      /* 8 bajtova = 64 bita. */  
    int i[2];      /* 2 riječi od po 4 bajta. */  
} Double_bits;
```

## *Binarni prikaz realnog broja — program (nast.)*

```
void prikaz_int(int broj)
{
    int nbits, bit, i;
    unsigned mask;

    /* Broj bitova u tipu int. */
    nbits = 8 * sizeof(int);

    /* Pocetna maska ima bit 1
       na najznacajnijem mjestu. */
    mask = 0x1 << nbits - 1;
```

## *Binarni prikaz realnog broja — program (nast.)*

```
for (i = 1; i <= nbits; ++i) {  
    /* Maskiranje odgovarajućeg bita. */  
    bit = broj & mask ? 1 : 0;  
    printf("%d", bit);  
    if (i % 4 == 0) printf(" ");  
    /* Pomak maske za 1 bit udesno. */  
    mask >>= 1;  
}  
printf("\n");  
  
return;  
}
```

## *Binarni prikaz realnog broja — program (nast.)*

```
void prikaz_double(double d)
{
    Double_bits u;

    u.d = d;

    printf("    1. rijec: ");
    prikaz_int( u.i[0] );
    printf("    2. rijec: ");
    prikaz_int( u.i[1] );

    return;
}
```

## *Binarni prikaz realnog broja — program (kraj)*

```
int main(void)
{
    double d;

    printf(" Upisi realni broj: ");
    scanf("%lf", &d);
    printf(" Prikaz broja %10.3f u racunalu:\n", d);

    prikaz_double(d);

    return 0;
}
```

---

## Binarni prikaz realnog broja — rezultati

Za **ulaz 1.0**, dobivamo (v. **p\_d\_2.out**):

---

Prikaz broja 1.000 u racunalu:

1. rijec: 0000 0000 0000 0000 0000 0000 0000 0000

2. rijec: 0011 1111 1111 0000 0000 0000 0000 0000

---

Za **ulaz 0.1**, dobivamo (v. **p\_d\_6.out**):

---

Prikaz broja 0.100 u racunalu:

1. rijec: 1001 1001 1001 1001 1001 1001 1001 1010

2. rijec: 0011 1111 1011 1001 1001 1001 1001 1001

---

Obratite pažnju na **zadnja 2** bita u **prvoj** riječi — to je rezultat **zaokruživanja**!

# Binarni prikaz realnog broja — zadaci

**Zadatak.** Napišite varijantu ovog programa za **realni** broj tipa **float** (v. **p\_float.c**).

**Zadatak.** Preuredite oba programa tako da **pregledno** ispisuju sve **bitne** dijelove u prikazu realnog broja:

- bit **predznaka**,
- bitove **karakteristike** (eksponenta),
- bitove **značajnog** dijela (mantise).



# Polja bitova

# Polja bitova

Polja bitova (engl. “bit-fields”) omogućuju rad s pojedinim bitovima unutar jedne riječi u računalu.

- Polje bitova je skup susjednih bitova u sklopu jedne memorijske jedinice (riječi).
- Može se proširiti na više susjednih riječi — spremanje “u bloku”.

## Upotreba:

- Spremanje 1-bitnih zastavica (engl. flag) u jednu riječ. Na primjer, koriste se u aplikacijama kao što je tablica simbola za kompajler,
- Komunikacija s vanjskim uređajima – treba postaviti ili očitati samo dijelove riječi.

# Deklaracija polja bitova

Deklaracija **polja bitova** je slična deklaraciji **strukture**:

---

```
struct ime {  
    clan_1 : broj_bitova_1;  
    ...  
    clan_n : broj_bitova_n;  
};
```

---

Svaki **clan\_k polja bitova** predstavlja

- **jedno** polje bitova unutar riječi u računalu,
- **duljine broj\_bitova\_k**.

# Polja bitova

## Primjer.

---

```
struct primjer {  
    unsigned int a : 1;  
    unsigned int b : 3;  
    unsigned int c : 2;  
    unsigned int d : 1;  
};  
struct primjer v;  
...  
if (v.a == 1) ...  
v.c = STATIC;
```

---

## Polja bitova (nastavak)


- Prva deklaracija definira **strukturu** razbijenu u četiri polja bitova: **a**, **b**, **c** i **d**.
- Ta polja redom imaju duljinu **1**, **3**, **2** i **1** bit. Prema tome zauzimaju **7** bitova.
- Poredak tih bitova unutar jedne riječi u računalu **ovisi o implementaciji**.
- Pojedine članove polja bitova dohvaćamo na isti način kako se dohvaćaju i članovi strukture — **v.a**, **v.b** itd.
- Ako broj bitova deklariran u polju bitova **nadmašuje** duljinu **jedne** riječi u računalu, za pamćenje polja bit će upotrebljeno **više** riječi.

## Polja bitova (nastavak)

**Primjer.** Program koji upotrebljava polje bitova:

```
#include <stdio.h>
int main(void) {
    struct {
        unsigned int a : 5;
        unsigned int b : 5;
        unsigned int c : 5;
        unsigned int d : 5;
    } v = {1, 2, 3, 4};
    printf(" v.a = %d, v.b = %d, v.c = %d,"
           " v.d = %d\n", v.a, v.b, v.c, v.d);
    printf(" sizeof(v) = %u\n", sizeof(v));
    return 0; }
```

## Polja bitova (nastavak)

Raspored polja unutar riječi može se kontrolirati korištenjem  neimenovanih članova pozitivne duljine unutar polja, kao u sljedećem primjeru.

Primjer.

---

```
struct primjer {  
    unsigned int a : 5;  
    unsigned int b : 5;  
    unsigned int   : 5;  
    unsigned int c : 5;  
};  
struct primjer v;
```

---

# Polja bitova (nastavak)

Primjer. Neimenovani član duljine 0 bitova

- “tjera” prevoditelj da sljedeće polje smjesti u sljedeću računalnu riječ.

---

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

```
int main(void) {  
    struct{  
        unsigned int a : 5;  
        unsigned int b : 5;  
        unsigned int   : 0;  
        unsigned int c : 5;  
    } v = {1, 2, 3};  
}
```



## *Polja bitova (nastavak)*

```
printf(" v.a = %d, v.b = %d, v.c = %d\n",  
      v.a, v.b, v.c);  
printf(" sizeof(v) = %u\n", sizeof(v));  
return 0;  
}
```

---