

# *Programiranje 1*

## *10. predavanje*

Saša Singer

`singer@math.hr`

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

PMF – Matematički odsjek, Zagreb

# Sadržaj predavanja

- **Funkcije:**
  - Definicija funkcije.
  - Naredba `return`.
  - Funkcija tipa `void`.
  - Funkcija bez argumenata.
  - Deklaracija funkcije.
  - Prijenos argumenata po vrijednosti.
  - Vraćanje vrijednosti preko argumenata.
  - Funkcije bez prototipa (**ne koristiti!**).
  - Primjeri funkcija za algoritme na cijelim brojevima.
  - Rekurzivne funkcije.
- **Dodatak:** Binomni koeficijenti, binarno potenciranje.

# Informacije — Praktični kolokvij

Praktični kolokvij — prvi krug kreće

- 🕒 za dva tjedna = kraj 11. i 12. tjedan nastave,
- 🕒 subota, 8. 12. — subota, 15. 12.,
- 🕒 obje subote su namjerno uključene (prazni praktikumi).

Razlog: da stignemo napraviti funkcije i na vježbama.

Termini će biti poznati najkasnije

- 🕒 za tjedan dana — do kraja 10. tjedna nastave.

Prijava za PK (zauzimanje termina) je onaj tjedan iza,

- 🕒 u 11. tjednu nastave, 3. 12. — 7. 12.,
- 🕒 preko aplikacije za zadaće!

# Praktični kolokvij — upute

Zapamtite: Vrijeme za rješenje je 45 minuta.

- Zadaci su objavljeni na webu.

U praktični kolokvij ulaze svi zadaci!

- Smatramo da znate funkcije, a “nizovi” nisu potrebni.

Korisno je odmah pogledati i početi vježbati, ako već niste.

Bilo bi vrlo lijepo da na praktičnom znate

- nakucati nešto, prevesti to (compile), isprobati (run), ...
- i još dobiti točne rezultate — samo to provjeravamo.

Dakle, vježbajte!

# Praktični kolokvij — upute

Barem dan–dva prije praktičnog kolokvija provjerite:

- da vam account “štima”,
- da se znate “logirati”,
- da znate naći IDE u kojem mislite raditi.

Uz svaki ponuđeni termin piše i praktikum u kojem se radi,

- tj. sami birate termin i praktikum.

Dakle, imate dovoljno vremena za isprobati (na licu mjesta) “je l’ vam sve radi”.

Ako vam account ne radi na praktičnom kolokviju, ili si ne znate password, ili ne znate naći/koristiti compiler, ...

- to je vaš problem. Što se nas tiče — to je pad.

# Informacije

Teme sljedećih predavanja su redom:

- “Ulaz i izlaz podataka”,
- “Polja” i “Operacije s nizovima podataka”,
- “Pretraživanje i sortiranje nizova”.

Na mom [webu](#), pod **dodatnim** materijalima za **Prog1** i **Prog2**, nalazi se tekst

- [in\\_out.pdf](#) (7 stranica, 59 kB),

koji sadrži **detaljan** opis funkcija za **formatirani ulaz** i **izlaz** podataka. Pogledajte za sljedeće predavanje!

# Funkcije

# Sadržaj

- **Funkcije:**
  - Definicija funkcije.
  - Naredba `return`.
  - Funkcija tipa `void`.
  - Funkcija bez argumenata.
  - Deklaracija funkcije.
  - Prijenos argumenata po vrijednosti.
  - Vraćanje vrijednosti preko argumenata.
  - Funkcije bez prototipa (**ne koristiti!**).
  - Primjeri funkcija za algoritme na cijelim brojevima.
  - Rekurzivne funkcije.
  - Funkcije s varijabilnim brojem argumenata.



# Definicija funkcije

Funkcija je programska cjelina koja

- uzima neke ulazne podatke,
- izvršava određeni niz naredbi,
- i vraća rezultat svog izvršavanja na mjesto poziva.

Slično kao u matematici: domena, kodomena, “pravilo”.

Definicija funkcije ima oblik:

---

```
tip_podatka ime_funkcije(tip_1 arg_1,  
                        ..., tip_n arg_n)  
{  
    tijelo funkcije  
}
```

---

# Definicija funkcije (nastavak)

Opis pojedinih dijelova definicije funkcije:

- `tip_podatka` je `tip podatka` koji će funkcija `vratiti` kao `rezultat` svog izvršavanja (opis `kodomene` funkcije).
- `ime_funkcije` je identifikator.
- Unutar `okruglih` zagrada, `iza` imena funkcije, nalazi se `deklaracija formalnih argumenata` funkcije (ako ih ima).
  - Prvi argument `arg_1` je `lokalna` varijabla `tipa tip_1`,
  - drugi argument `arg_2` je `lokalna` varijabla `tipa tip_2`, i tako redom.

`Formalni argumenti` opisuju `domenu` funkcije.

- Deklaracije pojedinih argumenata međusobno se `odvajaju zarezom` (to `nije` zarez operator).

## Definicija funkcije (nastavak)

Prvi dio **definicije** funkcije, **ispred** tijela,

```
tip_podatka ime_funkcije(tip_1 arg_1,  
                        ..., tip_n arg_n)
```

katkad se još zove i **zaglavlje** funkcije.

**Okrugle** zagrade ( ) **moraju** se napisati, čak i kad **nema** argumenata (v. malo kasnije), jer signaliziraju da je riječ o **funkciji**, a ne o nečem drugom (na pr. obična varijabla).

Na **kraju** **definicije**, **iza** zaglavlja, nalazi se **tijelo funkcije**.

- **Tijelo funkcije** piše se **unutar vitičastih** zagrada i ima strukturu **bloka**, odnosno, **složene naredbe**.

# Tijelo funkcije

Svaki blok ili složena naredba u programu sastoji se od

- deklaracija objekata (varijabli, tipova) i izvršnih naredbi, koje se izvršavaju ulaskom u blok.

Isto vrijedi i za tijelo funkcije, a izvršavanje počinje pozivom funkcije.

Za bilo koji blok, pa tako i za tijelo funkcije, vrijede sljedeća pravila o redoslijedu deklaracija i izvršnih naredbi.

- Po standardu C90, deklaracije svih objekata moraju prethoditi prvoj izvršnoj naredbi.
- Standard C99 dozvoljava deklaracije objekata bilo gdje u bloku, samo da su prije prvog korištenja objekta.

(Detaljnije u poglavlju “Struktura programa” na Prog2.)

# Formalni argumenti kao varijable

Formalni argumenti `arg_1`, `...`, `arg_n`, deklarirani u zaglavlju

```
tip_podatka ime_funkcije(tip_1 arg_1,  
                        ..., tip_n arg_n)
```

tom deklaracijom postaju

- lokalne varijable u toj funkciji.

Smijemo ih normalno koristiti, ali samo lokalno,

- unutar tijela te funkcije.

Razlika između ovih varijabli i ostalih varijabli, deklariranih u tijelu funkcije:

- formalni argumenti dobivaju vrijednost prilikom poziva funkcije — iz stvarnih argumenata (navedenih u pozivu).

# Formalni argumenti kao varijable (nastavak)

To je kao kad **zadajemo** “**točku**” u kojoj treba izračunati **vrijednost** funkcije.

Na primjer, pišemo **algoritam** za računanje **vrijednosti**  $\sin(x)$

- u **bilo kojoj** zadanoj točki  $x$  (probajte smisliti algoritam).

Algoritam možemo realizirati kao funkciju s imenom **sin**,

- kojoj će **zadana** vrijednost  $x$  biti **formalni** argument,

- a cijeli postupak pišemo u terminima te **varijable**  $x$ , bez obzira na **stvarnu** vrijednost te varijable.

Kod **poziva** te funkcije, moramo **zadati** “**pravu**” (konkretnu) vrijednost za  $x$  u kojoj želimo izračunati **vrijednost** funkcije.

- Ta vrijednost u **pozivu** je **stvarni** argument.

Na primjer, u pozivu **sin(2.35)**, **stvarni** argument je **2.35**.

# Povratna vrijednost funkcije

Što sve može biti **vrijednost** funkcije, tj.

- što se **smije** navesti kao **tip\_podatka** u zaglavlju?

**Pravilo.** Funkcija **može** vratiti

- aritmetički tip, strukturu, uniju, ili pokazivač,

ali **ne može** vratiti drugu funkciju ili polje. Međutim, **može** vratiti **pokazivač** na funkciju ili na polje (prvi element polja).

Usput, ako **tip\_podatka nije naveden** (što je dozvoljeno),

- pretpostavlja se da funkcija vraća podatak tipa **int**.

**Nemojte** to koristiti — prevoditelj to radi samo zbog **kompatibilnosti** s **prastarim C** programima (pisanim u “prvotnom” C-u prema **KR1**, **prije** ANSI/ISO standarda).

## Naredba return

Funkcija **vraća rezultat** svog izvršavanja naredbom **return**.  
Opći oblik te naredbe je:

---

```
return izraz;
```

---

**Izraz** se **može** staviti u **okrugle** zagrade, ali to **nije nužno**.

---

```
return (izraz);
```

---

Ako je **tip vrijednosti** izraza u naredbi **return** **različit** od **tipa podatka** koji funkcija vraća,

- **vrijednost** izraza će biti **konvertirana** u **tip\_podatka**.

Naredba **return**, ujedno, **završava** izvršavanje funkcije.



# Što se zbiva nakon return?

Izvršavanje programa **nastavlja** se tamo gdje je funkcija bila **pozvana**, a **vraćena vrijednost** (ako je ima)

● “**uvrštava**” se **umjesto poziva** funkcije.

**Primjer.** Poziv funkcije **sin** u sklopu složenog izraza

---

```
double y, phi, r;  
...  
r = 6.81;  
phi = 2.35;  
y = r * sin(phi);
```

---

**Trigonometrijske** funkcije postoje u **standardnoj** biblioteci.  
Pripadna datoteka zaglavlja je **<math.h>**.

# Korištenje povratne vrijednosti

Ako funkcija **vraća** neku **vrijednost**,

- povratna **vrijednost** se **ne mora** iskoristiti na mjestu **poziva**, već se može i “**odbaciti**”.

**Primjer.** Standardne funkcije **scanf** i **printf**, također, **vraćaju** neku **vrijednost** (više na sljedećem predavanju).

Uobičajeni pozivi tih funkcija “**odbacuju**” vraćenu vrijednost!

---

```
scanf("%d", &n);    printf(" n = %d\n", n);
```

---

Ako nam **vraćene** vrijednosti **trebaju**, smijemo napisati

---

```
procitano = scanf("%d", &n);  
napisano = printf(" n = %d\n", n);
```

---

## Primjer funkcije

Primjer. Sljedeća funkcija pretvara mala slova engleske abecede u velika. Ostale znakove ne mijenja.

- Formalni argument je samo jedan (`c`) i tipa je `char`.
- Vraćena vrijednost je tipa `char`.
- Ime funkcije je `malo_u_veliko`.

---

```
char malo_u_veliko(char c)
{
    char znak;
    znak = ('a' <= c && c <= 'z') ?
           ('A' + c - 'a') : c;
    return znak;
}
```

---

# Objašnjenje algoritma

**Algoritam.** Možemo pitati **znak-po-znak**, ali to **nije** potrebno.

Za sve standardne kôdove znakova (na pr. ASCII) u **tipu char** vrijedi sljedeće.

- Mala slova engleske abecede dolaze “**u bloku**” — **jedno za drugim**: **'a'**, **'b'**, ..., **'z'**,
- tj. pripadni kôdovi **rastu** za **po jedan**, počev od **'a'**.
- Potpuno **isto** vrijedi i za **velika** slova: **'A'**, **'B'**, ..., **'Z'**.

A sad iskoristimo da je **tip char**, zapravo, **cjelobrojni** tip, pa postoji **uspoređivanje** i **aritmetika** znakova.

Zato **test** “je li **c** malo slovo” glasi:

- **'a' <= c && c <= 'z'** (**uspoređivanje** znakova).

## Objašnjenje algoritma (nastavak)

Nadalje, odgovarajuće malo slovo ( $c$ ) i veliko slovo ( $znak$ )

- mora biti jednako “pomaknuto” (ili udaljeno) u odnosu na odgovarajuće “početno” slovo — ‘a’, odnosno, ‘A’.

Za ove pomake koristimo aritmetiku znakova.

- Pomak malog slova  $c$  od slova ‘a’  $= c - 'a'$ .
- Pomak velikog slova  $znak$  od slova ‘A’  $= znak - 'A'$ .

Kad ih izjednačimo, slijedi  $znak - 'A' = c - 'a'$ . Onda je

- $znak = 'A' + c - 'a'$  (to tražimo).

Prednosti ovog algoritma:

- Nije bitno jesu li velika slova ispred malih, ili obratno!
- Ne moramo znati pripadne kôdove znakova.

# Poziv funkcije

Funkcija se **poziva** navođenjem

- imena funkcije i
- liste (popisa) **stvarnih argumenata** u **zagradama**.

Primjer. **Poziv** funkcije `malo_u_veliko` može izgledati ovako:

---

```
veliko = malo_u_veliko(slovo);
```

---

Ovdje je varijabla `slovo`

- jedini **stvarni** argument u **pozivu** funkcije.

**Trenutna** vrijednost te varijable se **prenosi** u funkciju,

- kao **početna** vrijednost **formalnog** argumenta `c`.

## Primjer poziva funkcije

Primjer. Glavni program (funkcija `main`) s pozivom funkcije `malo_u_veliko` iz prethodnog primjera (v. `p_01.c`).

---

```
int main(void)
{
    char malo, veliko;

    printf(" Unesite malo slovo: ");
    scanf("%c", &malo);
    veliko = malo_u_veliko(malo);
    printf("\n Veliko slovo = %c\n", veliko);
    return 0;
}
```

---

Za ulaz: `d`, dobijemo izlaz: `D`.

# Poziv funkcije — izraz kao stvarni argument

Stvarni argument funkcije može biti izraz. Sasvim općenito,

● stvarni argument je uvijek izraz.

Prvo se računa vrijednost tog izraza, a zatim se ta vrijednost prenosi u funkciju — dodjeljuje formalnom argumentu.

Primjer. Poziv trigonometrijske funkcije  $\sin(2 * x + y)$ .

Primjer. Pozivi funkcije `malo_u_veliko` mogu biti i ovo:

---

```
veliko = malo_u_veliko('a' + 3);  
veliko = malo_u_veliko(veliko + 3);
```

---

Rezultati su: D i G (v. `p_01a.c`).

Uočite da u drugom pozivu nema pretvaranja u veliko slovo.



## Primjer funkcije — varijante zapisa

Funkciju `malo_u_veliko` možemo napisati na **razne** načine.

Cijeli **uvjetni** izraz možemo **odmah** napisati u **return** naredbi, tako da nam varijabla `znak` uopće **ne treba** (v. `p_02.c`).

---

```
char malo_u_veliko(char c)
{
    return ('a' <= c && c <= 'z') ?
           ('A' + c - 'a') : c;
}
```

---

**Okrugle** zagrade, također, **nisu** potrebne,

● zbog **niskog** prioriteta **uvjetnog** operatora.

Služe samo za **preglednost**.

## Primjer funkcije — varijante zapisa (nastavak)

Usput, ako **obrišemo okrugle** zagrade, dobijemo ovo:

```
char malo_u_veliko(char c)
{
    return 'a' <= c && c <= 'z' ? 'A' + c - 'a' : c;
}
```

Pa vi probajte **pročitati!** Izgleda prilično “**odurno**”, zar ne, iako uredno **radi** (v. **p\_03.c**).

**Napomena.** Naša funkcija **malo\_u\_veliko** radi **isto** što i

• standardna funkcija **toupper** iz **<ctype.h>**.

U toj datoteci **zaglavlja** postoji još hrpa funkcija za testiranje **znakova** (v. drugi semestar).

## Višestruke return naredbe

Ako se programski tok **grana unutar** funkcije, onda smijemo

● imati **više return** naredbi **unutar iste** funkcije.

**Primjer.** Funkcija koja pretvara **mala** u **velika** slova, napisana **if-else** naredbom (v. **p\_04.c**).

---

```
char malo_u_veliko(char c)
{
    if ('a' <= c && c <= 'z')
        return ('A' + c - 'a');
    else
        return c;
}
```

---

## Funkcija bez rezultata — tipa void

Ako funkcija **ne vraća** nikakvu **vrijednost**, onda se za **tip** “**vraćene vrijednosti**” koristi **ključna** riječ **void** (“prazan”).

**Primjer.** Ispis maksimalnog od **dva cijela** broja (v. **p\_maxi.c**).

---

```
void ispisi_max(int x, int y)
{
    int max;
    max = (x >= y) ? x : y;
    printf(" Maksimalna vrijednost = %d\n", max);
    return;
}
```

---

Naredba **return** **nema** izraz. Ako je na **kraju** funkcije, može biti **izostavljena**. No, **bolje** ju je **zadržati**, radi **preglednosti**.

## Funkcija bez rezultata (nastavak)

Kod **poziva** takve funkcije (tipa **void**) treba malo **paziti**.

Zato što funkcija **ne vraća** nikakvu **vrijednost**,

- **povratna vrijednost** se **ne smije** “iskoristiti” na mjestu **poziva**.

**Primjer.** Na **pokušaj korištenja** povratne vrijednosti, poput

---

```
m = ispisi_max(x, y);
```

---

prevoditelj bi se **trebao** “pobuniti” i javiti **grešku**.

Ako **ne javi** grešku i prevede takav program,

- **dodijeljena vrijednost** je sigurno neko “**smeće**”.

Isto vrijedi i za prevoditelj!

# Funkcija bez argumenata

Funkcija koja **nema** nikakve **argumente** **definira** se ovako:

```
tip_podatka ime_funkcije(void)
{
    tijelo funkcije
}
```

Ključna riječ **void** (**unutar** zagrada) označava da funkcija **ne uzima argumente**.

**Napomena.** Ovakve funkcije **nisu** besmislene. Na primjer, standardna funkcija **getchar** za čitanje **jednog znaka** (sa standardnog ulaza) **nema** argumenata (v. sljedeće predavanje).

Osim toga, glavni program — funkcija **main**, također, bar zasad, **nema** argumenata. Može ih **imati** (v. drugi semestar).

# Funkcija bez argumenata — poziv

Poziv takve funkcije ima

- praznu listu stvarnih argumenata u zagradama.

Primjer. Poziv funkcije bez argumenata.

---

```
varijabla = ime_funkcije();
```

---

Zagrade () su obavezne, jer informiraju prevoditelj da je

- identifikator ime\_funkcije ime funkcije, a ne nešto drugo (na pr. obična varijabla).

# Deklaracija funkcije

Do sad smo odvojeno pisali funkciju i glavni program (`main`) u kojem se poziva funkcija.

Pitanje: Kako se funkcije “spajaju” u cijeli program, tj.

- kojim redom se pišu funkcije u programu?

Svaka bi funkcija, prije prvog poziva u programu, trebala biti deklarirana — navođenjem tzv. prototipa.

- Mogućnost da se to ne napravi ostavljena je samo zbog kompatibilnosti s prastarim C programima, i ne treba ju koristiti!

Osim toga, u jeziku C++ više nije dozvoljena.

Zato: “trebala bi” → “mora”!



# Deklaracija funkcije (nastavak)

Dakle, jednostavno smatrajte da svaka funkcija

- mora biti deklarirana prije poziva u programu.

Svrha deklaracije (prototipa) je kontrola ispravnosti svih poziva funkcije — prilikom prevođenja programa.

- Deklaracija informira prevoditelj o:
  - imenu funkcije,
  - broju i tipu argumenata,
  - te tipu vrijednosti kojeg funkcija vraća.

U nastavku, kao primjer funkcije, koristimo varijantu funkcije `ispisi_max`, koja

- ispisuje maksimalni od dva realna broja (tipa `double`).

# Definicija funkcije kao deklaracija

Ako je funkcija

- definirana u istoj datoteci u kojoj se poziva,
- i to prije svog prvog poziva,

onda definicija služi i kao deklaracija,

- pa posebna deklaracija nije potrebna.

U svim ostalim slučajevima,

- funkcija se mora posebno deklarirati.

Kako se to radi kad se funkcija nalazi u drugoj datoteci, bit će riječi u poglavlju “Struktura programa” na Prog2.

Do tada, sve programe pišemo u jednoj datoteci.

## Primjer — deklaracija nije potrebna

Primjer. Funkcija je definirana prije prvog poziva (u `main`), tj. ispred funkcije `main` (v. `p_maxd_1.c`).

---

```
#include <stdio.h>

void ispisi_max(double x, double y)
{
    double max;
    max = (x >= y) ? x : y;
    printf(" Maksimalna vrijednost = %g\n", max);
    return;
}
```

## Primjer — deklaracija nije potrebna (nastavak)

```
int main(void)
{
    double x, y;

    printf(" Unesite dva realna broja: ");
    scanf("%lg %lg", &x, &y);
    ispisi_max(x, y);
    return 0;
}
```

U trenutku **prvog poziva** prevoditelj **zna** da je **ispisi\_max** funkcija koja

- ima **dva** argumenta tipa **double**,
- i ne vraća **ništa**.

# Deklaracija ili prototip funkcije

Ako **definiciju** funkcije smjestimo **nakon poziva** funkcije,

- moramo tu funkciju **deklarirati prije** prvog **poziva**.

**Deklaracija** ili **prototip** funkcije ima oblik:

```
tip_podatka ime_funkcije(tip_1 arg_1,  
                        ..., tip_n arg_n);
```

Dakle, **deklaracija** sadrži samo **zaglavlje** funkcije, **bez bloka** u kojem je **tijelo** funkcije.

**Imena** argumenata **arg\_1**, ..., **arg\_n** mogu biti **izostavljena**, jer se **tip** argumenata vidi i bez toga.

```
tip_podatka ime_funkcije(tip_1, ..., tip_n);
```

# Deklaracija ili prototip funkcije (nastavak)

Deklaracije objekata “istog” tipa **vrijednosti** mogu se spojiti, slično kao za obične varijable.

Primjer.

---

```
int n, f(double), g(int, double);
```

---

U ovoj deklaraciji,

- **n** je **varijabla** tipa **int**, a
- **f** i **g** su **funkcije** koje **vraćaju** vrijednost tipa **int**.

Obično se **deklaracija** piše

- na **početku** datoteke,
- ili **u funkciji** u kojoj je poziv.

## Primjer — deklaracija potrebna

Primjer. Funkcija je definirana **iza** prvog poziva (u `main`), a deklaracija je **unutar** funkcije `main` (v. `p_maxd_2.c`).

---

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

```
int main(void)
```

```
{
```

```
    double x, y;
```

```
    void ispisi_max(double, double); /* Dekl. */
```

```
    printf(" Unesite dva realna broja: ");
```

```
    scanf("%lg %lg", &x, &y);
```

```
    ispisi_max(x, y);
```

```
    return 0;
```

```
}
```

## Primjer — deklaracija potrebna (nastavak)

```
void ispisi_max(double x, double y)
{
    double max;
    max = (x >= y) ? x : y;
    printf(" Maksimalna vrijednost = %g\n", max);
    return;
}
```

U ovom primjeru, deklaracija (prototip) funkcije je

```
void ispisi_max(double, double);
```

Mogli smo napisati i

```
void ispisi_max(double x, double y);
```



## Primjer — deklaracija potrebna (nastavak)

Primjer. Deklaracija funkcije može biti i **izvan** funkcije gdje je poziv — na primjer, na **početku** datoteke (v. **p\_maxd\_3.c**).

---

```
#include <stdio.h>

void ispisi_max(double, double); /* Deklaracija */

int main(void)
{
    double x, y;
    printf(" Unesite dva realna broja: ");
    scanf("%lg %lg", &x, &y);
    ispisi_max(x, y);
    return 0;
}
```

## Primjer — deklaracija potrebna (nastavak)

```
void ispisi_max(double x, double y)
{
    double max;
    max = (x >= y) ? x : y;
    printf(" Maksimalna vrijednost = %g\n", max);
    return;
}
```

**Prednost** ove globalne deklaracije na početku datoteke, **izvan** svih funkcija:

- funkcija `ispisi_max` može se pozvati **u svim** funkcijama **iza** deklaracije.

Ovo se često koristi za **sve** funkcije u programu (osim `main`), jer **ne ovisi** o **poretku** pisanja funkcija iza toga.

# Načini prijenosa argumenata

Formalni i stvarni argumenti (ili parametri):

- Argumenti deklarirani u definiciji funkcije nazivaju se formalni argumenti.
- Izrazi koji se pri pozivu funkcije nalaze na mjestima formalnih argumenata nazivaju se stvarni argumenti.

Veza između formalnih i stvarnih argumenata uspostavlja se

- prijenosom argumenata prilikom poziva funkcije.

Sasvim općenito, postoje dva načina prijenosa (ili predavanja) argumenata prilikom poziva funkcije:

- prijenos vrijednosti argumenata — engl. “call by value”,
- prijenos adresa argumenata — engl. “call by reference”.

# Prijenos argumenata po vrijednosti

Kod prijenosa **vrijednosti** argumenata

- funkcija prima **kopije** vrijednosti **stvarnih** argumenata, što znači da
- funkcija **ne može izmijeniti stvarne** argumente.

**Stvarni** argumenti **mogu** biti **izrazi**. Prilikom poziva funkcije,

- **prvo** se izračuna **vrijednost** tog izraza,
- a **zatim** se ta **vrijednost prenosi** u funkciju,
- i **kopira** u odgovarajući **formalni** argument.

# Prijenos argumenata po adresi

Kod prijenosa **adresa** argumenata

- funkcija prima **adrese stvarnih** argumenata, što znači da

- funkcija **može izmijeniti stvarne** argumente, tj. **sadržaje** na tim **adresama**.

**Stvarni** argumenti, u principu, **ne mogu** biti **izrazi**,

- već **samo varijable**,

- odnosno, **objekti** koji **imaju adresu**.

# Prijenos argumenata u C-u

U C-u postoji **samo** prijenos argumenata **po vrijednosti**.

- Svaki **formalni** argument ujedno je i **lokalna** varijabla u toj funkciji.
- **Stvarni** argumenti u **pozivu** funkcije su **izrazi** (izračunaj vrijednost, kopiraj ju u **formalni** argument).

Ako funkcijom želimo **promijeniti** vrijednost nekog **podatka**,

- pripadni argument **treba** biti **pokazivač na taj podatak**, tj. njegova **adresa**!
- Tada se **adresa** prenosi **po vrijednosti** — **kopira** u funkciju (promjena te kopije **ne** mijenja **stvarnu** adresu),
- ali smijemo **promijeniti sadržaj** na toj **adresi**, koristeći operator dereferenciranja **\***.

## Primjer — prijenos po vrijednosti

Primjer. Prijenos argumenata **po vrijednosti**.

---

```
#include <stdio.h>

void f(int x)
{
    x += 1;
    printf("Unutar funkcije: x = %d\n", x);
    return;
}
```

---

Funkcija **f** povećava vrijednost argumenta za **1**. Međutim, to povećanje **x** za **1** događa se

 u **lokalnoj** varijabli **x**, pa **nema** traga **izvan** funkcije **f**.

## Primjer — prijenos po vrijednosti (nastavak)

```
int main(void)
{
    int x = 5;
    printf("Prije poziva:      x = %d\n", x);
    f(x);
    printf("Nakon poziva:     x = %d\n", x);
    return 0;
}
```

Rezultat izvršavanja programa (`arg1_1.c`) je:

```
Prije poziva:      x = 5
Unutar funkcije:  x = 6
Nakon poziva:     x = 5
```



## Primjer — prijenos “po adresi”

Primjer. Prijenos argumenata “po adresi” preko pokazivača. Jednostavno, u funkciji **f**, svagdje “dodamo” **\*** ispred **x**.

---

```
void f(int *x)
{
    *x += 1;
    printf("Unutar funkcije: x = %d\n", *x);
    return;
}
```

---

Ovdje povećavamo

🔴 sadržaj na adresi **x** za **1**, pa ima traga izvan funkcije **f**.

Pokazivač (adresa) je lokalna varijabla **x**. Promjena te varijable (tj. adrese) i dalje nema traga izvan funkcije **f**.

## Primjer — prijenos “po adresi” (nastavak)

U prvom primjeru — `void f(int x)`

• `x` je lokalna varijabla tipa `int`.

U drugom primjeru — `void f(int *x)`

• `x` je lokalna varijabla tipa `int *`, tj. pokazivač na `int`.

Nije lijepo da se razne stvari isto zovu! Na primjer, ime `px` je puno bolje u drugom primjeru, zato da asocira na pokazivač.

---

```
void f(int *px)
{
    *px += 1;
    printf("Unutar funkcije: x = %d\n", *px);
    return;
}
```

---

## Primjer — prijenos “po adresi” (nastavak)

```
int main(void)
{
    int x = 5;
    printf("Prije poziva:      x = %d\n", x);
    f(&x);    /* Stvarni argument je pokazivac. */
    printf("Nakon poziva:     x = %d\n", x);
    return 0;
}
```

Rezultat izvršavanja programa (`arg1_2.c`, `arg1_3.c`) je:

```
Prije poziva:      x = 5
Unutar funkcije:  x = 6
Nakon poziva:     x = 6
```

## Primjer — završni komentar o pozivima

Kod prijenosa po **vrijednosti**, poziv funkcije **f** u glavnom programu može glasiti i ovako:

---

```
f(x + 2);
```

---

tj. **stvarni** argument **smije** biti **izraz**.

Za razliku od toga, kod prijenosa **adrese**, poziv funkcije **f** u glavnom programu **ne smije** biti:

---

```
f(&(x + 2));
```

---

jer **izraz nema** adresu (v. **arg1\_4.c**)!

Ali, **smije** se napisati **f(&x + 2);** — aritmetika pokazivača!

**Sami** pogledajte sljedeća dva primjera (**arg2\_1.c**, **arg2\_2.c**).

## Primjer 2 (zadaca) — prijenos po vrijednosti

```
#include <stdio.h>
void f(int x, int y) {
    x += y;
    y += x;
    printf("Unutar funkcije: x=%d, y=%d\n", x, y);
    return; }
int main(void) {
    int x = 2, y = 3;
    printf("Prije poziva: x=%d, y=%d\n", x, y);
    f(y, x + y);
    printf("Nakon poziva: x=%d, y=%d\n", x, y);
    return 0; }
```

Ispisane vrijednosti: 2 3 8 13 2 3.

## Primjer 2 (zadaca) — prijenos “po adresi”

```
#include <stdio.h>
void f(int *x, int *y) {
    *x += *y;
    *y += *x;
    printf("Unutar funkcije: x=%d, y=%d\n", *x, *y);
    return; }
int main(void) {
    int x = 2, y = 3, z;
    z = x + y;
    printf("Prije poziva: x=%d, y=%d\n", x, y);
    f(&y, &z);
    printf("Nakon poziva: x=%d, y=%d\n", x, y);
    return 0; }
```

Ispisane vrijednosti: 2 3 8 13 2 8. Koliko je **z** na kraju?

# Pravila o argumentima

Pravila pri prijenosu argumenata:

- Broj stvarnih argumenata pri svakom pozivu funkcije mora biti jednak broju formalnih argumenata.
- Ako je funkcija ispravno deklarirana, tj. prevoditelj pri pozivu zna broj i tip argumenata,
  - stvarni argumenti čiji se tip razlikuje od tipa odgovarajućih formalnih argumenta,
  - pretvaraju se u tip formalnih argumenata, isto kao kod pridruživanja.
- Redosljed izračunavanja stvarnih argumenata nije definiran i ovisi o implementaciji.

Dakle, ne mora biti  $L \rightarrow D$ . Ponekad je obratan!

## Poziv funkcije — pretvaranje tipova

Primjer. Funkcija `sqrt` iz zaglavlja `<math.h>` ima prototip

```
double sqrt(double);
```

Nakon `#include <math.h>`, poziv funkcije `sqrt` može biti:

```
int x; double y;  
...  
y = sqrt(2 * x - 3);
```

Vrijednost izraza `2 * x - 3` je tipa `int`. Kod poziva,

- prvo se ta vrijednost pretvara u `double`,
- a zatim se prenosi u funkciju.

Varijabla `y` korektno poprima `double` vrijednost  $\sqrt{2x - 3}$ .



# Funkcije *bez prototipa* — **NE KORISTITI**

Ako **zaboravite** deklaraciju — da znate što se onda **zbiva!**  
U programu se **moгу** koristiti i funkcije koje **nisu** prethodno **deklarirane**. U tom slučaju vrijedi:

- Prevoditelj pretpostavlja da funkcija **vraća** podatak tipa **int** i ne pravi **nikakve** pretpostavke o **broju** i **tipu** argumenata.
- Na svaki **stvarni** argument **cjelobrojnog** tipa primjenjuje se **integralna** promocija (konverzija argumenata tipa **short** i **char** u **int**), a svaki **stvarni** argument tipa **float** konvertira se u **double**.
- **Broj** i **tip** (konvertiranih) **stvarnih** argumenata **mora se podudarati** s **brojem** i **tipom formalnih** argumenata, da bi poziv bio **korektan**.

## Primjer — funkcija s prototipom

Primjer. Probajte s  $x = 2.0$  (kao što piše) i s  $x = 2.5$ .

```
#include <stdio.h>
int f(double);    /* Ključno - ima prototip! */
int main(void)
{
    float x = 2.0; /* double const --> float */
    printf("%d\n", f(2)); /* int --> double */
    printf("%d\n", f(x)); /* float --> double */
    return 0;
}
int f(double x) {
    return (int) x*x; /* int * double --> int */
}
```

## Primjer — komentar rezultata (prioriteti)

Naredba `return` u funkciji `f` glasi:

```
return (int) x*x; /* int * double --> int */
```

Uočite da `(int) x*x` nije isto što i `(int) (x*x)`.

**Unarni** operator promjene tipa `(int)` ima **viši** prioritet od **množenja**, pa se izraz `(int) x*x` svodi na

• `int * double` i **tip vrijednosti** je `double`.

Tek **na kraju** ide konverzija u navedeni tip rezultata `int`.

Za `x = 2.5` dobivamo

• `(int) x*x = 2 * 2.5 = 5.0`, pa je `f(2.5) = 5`,

• `(int) (x*x) = (int) 6.25 = 6`.

## Primjer — funkcija *bez* prototipa

Primjer. Funkcija *f* stvarno vraća tip *int*.

---

```
... /* Nema prototip za f */
int main(void)
{
    float x = 2.0; /* double const --> float */
    printf("%d\n", f(2)); /* greska u izvodenju */
    printf("%d\n", f(x)); /* O.K. */
    return 0;
}
int f(double x) { /* tip odgovara pretpostavci */
    return (int) x*x; /* int * double --> int */
}
```

---

Poziv *f(2)* šalje samo 4 bajta u *f* za *x* — rezultat je “*smeće*”.

## Primjer — funkcija *bez* prototipa (nastavak)

Primjer. Funkcija *f* stvarno vraća tip *double*, a *ne int*.

---

```
... /* Nema prototip za f */
int main(void)
{
    float x = 2.0; /* double const --> float */
    printf("%d\n", f(x)); /* O.K. */
    return 0;
}
double f(double x) {
    /* greska u prevodjenju:
       redefinicija simbola f iz int u double */
    return x*x;
}
```

---

# Primjeri funkcija za algoritme na cijelim brojevima

# Sadržaj

- Primjeri funkcija za algoritme na cijelim brojevima:
  - Broj znamenki cijelog broja.
  - Provjera znamenki broja.
  - Najveća zajednička mjera — Euklidov algoritam.
  - Potencija broja 2.

# Broj znamenki broja

**Primjer.** Treba naći broj znamenki nenegativnog cijelog broja  $n$  u zadanoj bazi  $b$ .

Algoritam za brojanje znamenki (od prošli puta) je:

- “brisanje” znamenki i to “straga”,
- i brojanje obrisanih znamenki.



# Broj znamenki broja u zadanoj bazi (nastavak)

Bitni odsječak programa je izgledao ovako:

---

```
unsigned int b = 10, n, broj_znam;
...
broj_znam = 0;
while (n > 0) {
    ++broj_znam;
    n /= b;
}

printf(" ima %u znamenki u bazi %u\n",
       broj_znam, b);
```

---

## Broj znamenki broja u zadanoj bazi (nastavak)

Kod realizacije funkcijom, “destrukcija” ulaznog broja **n** nije problem, jer **uništavamo** lokalnu varijablu!

---

```
unsigned int broj_znamenki(unsigned int n,  
                           unsigned int b)  
{  
    unsigned int broj_znam = 0;  
  
    while (n > 0) {  
        ++broj_znam;  
        n /= b;  
    }  
    return broj_znam;  
}
```

---

# Provjera znamenki broja

**Primjer.** Zadan je nenegativni cijeli broj  $n$ . Treba naći odgovor na pitanje

- postoji li znamenka tog broja koja je jednaka 5 (u zadanoj bazi  $b = 10$ ).

Traženu znamenku zovemo **trazena**.

Koristimo “**skraćeni**” algoritam provjere, koji **idealno** odgovara realizaciji funkcijom:

- čim **saznamo** odgovor — odmah se **vratimo!**

## Postoji znamenka ... ? (nastavak)

Bitni odsječak programa je izgledao ovako:

---

```
odgovor = 0;  /* NE, laz */
while (n > 0) {
    znam = n % b;
    if (znam == trazena) {
        odgovor = 1;
        break;
    }
    n /= b;
}
```

---

## Postoji znamenka ... ? (nastavak)

Odgovarajuća funkcija je:

---

```
int odgovor(unsigned int n, unsigned int b,
            unsigned int trazena)
{
    while (n > 0) {
        if (n % b == trazena)
            return 1;
        n /= b;
    }
    return 0;
}
```

---

Za **prekid** petlje, umjesto **break**, odmah pišemo **return** s odgovorom.

# Najveća zajednička mjera

**Primjer.** Treba naći **najveću** zajedničku mjeru  $M(a, b)$ , **cijelih** brojeva  $a$  i  $b$ , uz pretpostavku da je  $b \neq 0$ .

Algoritam se bazira na Euklidovom teoremu o dijeljenju

$$\bullet a = q \cdot b + r, \text{ za neki } q \in \mathbb{Z}, \text{ gdje je } r \text{ ostatak, } 0 \leq r < |b|.$$

Ključni koraci:

- $\bullet$  Ako  $d \mid a$  i  $d \mid b$ , onda  $d \mid r$ , pa je  $M(a, b) = M(b, r)$  (“**smanjujemo**” argumente, po apsolutnoj vrijednosti).
- $\bullet$  Ako je  $r = 0$ , onda je  $a = q \cdot b$ , pa je  $M(a, b) = b$  (**kraj**).

## Najveća zajednička mjera (nastavak)

Dio programa koji računa  $M(a, b)$  (u komentaru je promjena za rezultat  $M(a, b) > 0$ ):

```
int a, b, ostatak, mjera;
...
while (1) {
    ostatak = a % b;
    if (ostatak == 0) {
        mjera = b;          // mjera = abs(b);
        break;
    }
    a = b;
    b = ostatak;
}
```

## Najveća zajednička mjera (nastavak)

Odgovarajuća funkcija koja vraća  $M(a, b)$ :

---

```
int euklid(int a, int b)
{
    int ostatak;

    while (1) {
        ostatak = a % b;
        if (ostatak == 0)
            return b;          // return abs(b);
        a = b;
        b = ostatak;
    }
}
```

---



## Potencija broja 2

**Primjer.** Za zadani nenegativni cijeli broj  $n$  treba naći odgovor na pitanje

• je li broj  $n$  **potencija** broja  $d = 2$ ,

tj. može li se  $n$  prikazati u obliku

•  $n = d^k$ , s tim da je eksponent  $k > 0$ ?

Odgovarajuća **funkcija** mora **vratiti**:

• **odgovor** na pitanje — kao **povratnu** vrijednost,

• i eksponent  $k$  — kroz “**varijabilni**” argument.

Dakle, pripadni argument mora biti **pokazivač** — adresa varijable u koju želimo spremiti  $k$ .

## Potencija broja 2 (nastavak)

Bitni odsječak programa je izgledao ovako:

---

```
unsigned int n, d = 2, k, odgovor;

k = 0;
    /* Sve dok je n djeljiv s d,
       podijeli ga s d. */
while (n % d == 0) {
    ++k;
    n /= d;
}

    /* mora ostati n == 1 */
odgovor = n == 1 && k > 0;
```

---

## Potencija broja 2 (nastavak)

Odgovarajuća funkcija:

---

```
int odgovor(unsigned int n, unsigned int d,
            unsigned int *pk)
{
    unsigned int k = 0;

    while (n % d == 0) {
        ++k;
        n /= d;
    }
    *pk = k;
    return n == 1 && k > 0;
}
```

---

# Rekurzivne funkcije

# Rekurzivne funkcije

Programski jezik C dozvoljava tzv. rekurzivne funkcije, tj.

- da funkcija poziva samu sebe.

U pravilu,

- rekurzivni algoritmi su kraći,
- ali izvođenje, u načelu, traje dulje.

Katkad — puno dulje, ako puno puta računamo istu stvar.  
Zato oprez!

**Napomena.** Svaki rekurzivni algoritam mora imati

- “nerekurzivni” dio, koji omogućava prekidanje rekurzije.

Najčešće je to neki if u inicijalizaciji rekurzije.

# Primjer rekurzivne funkcije — faktorijele

Primjer. Za računanje faktorijela

$$n! = 1 \cdot 2 \cdot 3 \cdots n = n \cdot (n - 1)!$$

možemo napisati **rekurzivnu** funkciju (v. `fakt_r.c`):

---

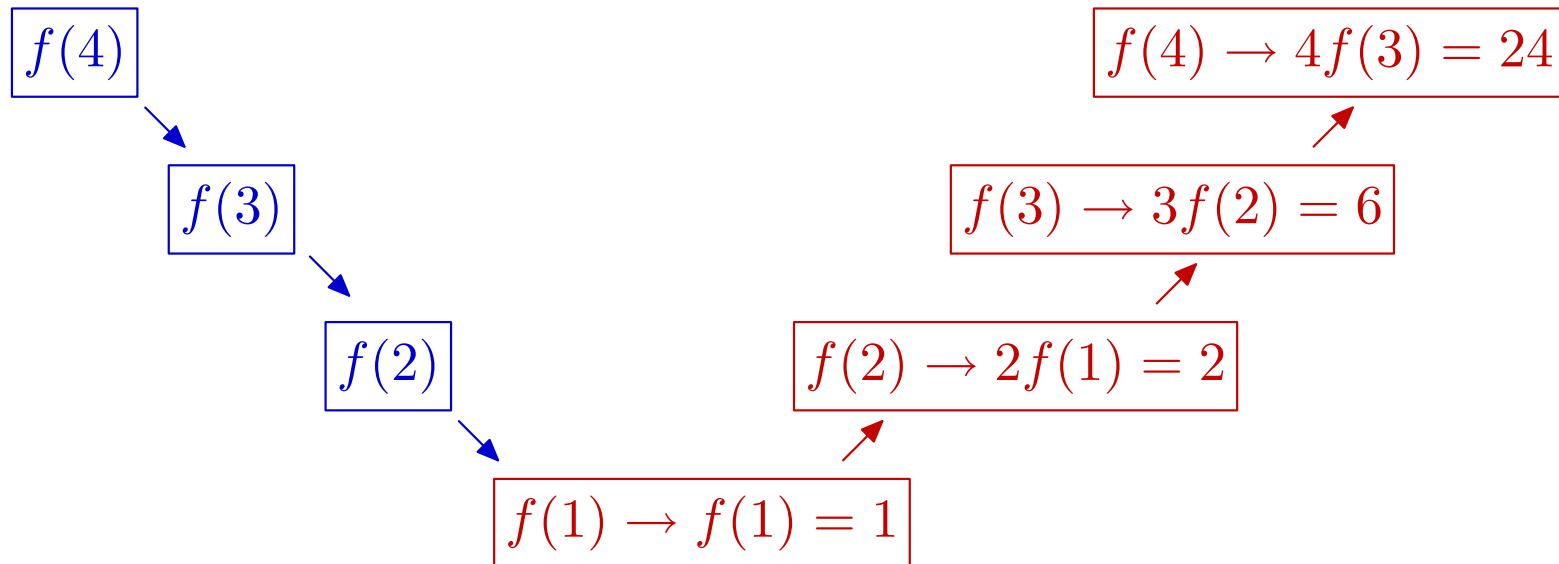
```
long int fakt(int n)
{
    if (n <= 1)
        return 1L;
    else
        return n * fakt(n - 1);
}
```

---

Ali, **nemojte** to raditi. **Zabranjujem!**

# Faktorijele — rekurzivno (nastavak)

Za  $n = 4$ , “slika” rekurzivnih poziva funkcije izgleda ovako:



Osnovni **nedostatak** ovog primjera:

• pozivi su “**linearni**” i ništa “**pametno**” **ne rade**.

Ukupno je potrebno  $n$  **poziva** funkcije da se izračuna  $n!$  (uključivo i polazni **vanjski** poziv).

## Faktorijele — bez rekurzije

To ide **puno brže nerekurzivno**, bez svih tih silnih poziva.

Faktorijele, naravno, **možemo** izračunati u **jednoj petlji**.

Varijanta sa **silaznom** petljom po **n** je (v. **fakt\_p.c**):

---

```
long int fakt(int n)
{
    long int f = 1L;
    for (; n > 1; n--) f *= n;
    return f;
}
```

---

To je bitno **efikasnije**, jer trebamo samo **jedan poziv** funkcije. Sve ostalo (tj. **množenja**) traje **podjednako** kao u rekurziji!



## Faktorijele — bez rekurzije (nastavak)

Varijanta s **uzlaznom** petljom **do n**, ali trebamo **pomoćnu** varijablu za **faktor** u petlji (v. **fakt\_p1.c**):

---

```
long int fakt(int n)
{
    long int f = 1L;
    int i;
    for (i = 2; i <= n; ++i) f *= i;
    return f;
}
```

---

**Drugi** “klasični” primjer kad **rekurziju ne treba** koristiti su **Fibonaccijski** brojevi — definirani ovako:  $F_0 = 0$ ,  $F_1 = 1$ , i  $F_i = F_{i-1} + F_{i-2}$ , za  $i \geq 2$ . Detaljno objašnjenje u **Prog2**.

# Primjer rekurzivne funkcije — prikaz broja u bazi

**Primjer.** Treba **ispisati prikaz** prirodnog broja  $n$  u zadanoj bazi  $b$  — sve njegove znamenke, od **vodeće** do **najniže**.

Što ga to ima s **rekurzijom**? Polako ...

- **Ima “štos”** — baš za rekurziju.

**Ispis** znamenki broja mora ići “slijeva nadesno” ( $\longrightarrow$ ), tj. treba početi **sprijeda** — od **vodeće** znamenke. To znači da

- **prvo** treba doći do **vodeće** znamenke,
- pa **onda** ići **unatrag**.

S druge strane,

- znamenke broja se **puno lakše** “skidaju” **straga**.

Međutim, to je **naopako** od redosljeda za **ispis**.

## Prikaz broja u bazi (nastavak)

Zato koristimo “**linearnu**” **rekurziju**. Svaki poziv funkcije

- “skine” i **zapamti** “**svoju**” znamenku,
- a **piše** tu znamenku tek **nakon** **rekurzivnog** poziva.

Tako jednostavno dobivamo “**naopaki**” **ispis**!

**Algoritam**. Ispis znamenki broja  $n$  u bazi  $b$ :

- **prvo** ispiši sve znamenke od  $n$ , **osim zadnje**, tj.
  - ispiši sve znamenke broja  $n \text{ div } b$  u bazi  $b$  (**rekurzija**),
- zatim, ispiši **zadnju** znamenku  $n \text{ mod } b$ .

Naravno, to radimo samo ako je  $n > 0$ , tj. ima znamenki.

Radi jednostavnosti, uzimamo da je  $b \leq 10$ ,

- zato da znamenke budu “obične” — **numeričke**.

## Prikaz broja u bazi (nastavak)

Argumenti funkcije su “trenutni” broj  $n$  i baza  $b$ .

---

```
#include <stdio.h>

void ispis_u_bazi(unsigned int n, unsigned int b)
{
    if (n > 0) {
        ispis_u_bazi(n / b, b);
        printf("%u", n % b);
    }
    return;
}
```

---

**Napomena.** Bazu  $b$  bi trebalo **izbaciti**, jer se **ne mijenja**, ali još “ne znamo” za **globalne** varijable.

## Prikaz broja u bazi (nastavak)

Glavni program — za bazu  $b = 10$  (v. prikaz\_1.c):

```
int main(void) {
    unsigned int b = 10, n;
    printf(" Upisi nenegativni broj n: ");
    scanf("%u", &n);
    printf("\n Prikaz broja %u u bazi %u: ", n, b);
    ispis_u_bazi(n, b);
    printf("\n");
    return 0;
}
```

Za ulaz: 123456, rezultat je:

Prikaz broja 123456 u bazi 10: 123456

## Prikaz broja u bazi (nastavak)

Glavni program — za bazu  $b = 2$  (v. prikaz\_2.c):

```
int main(void) {
    unsigned int b = 2, n;
    printf(" Upisi nenegativni broj n: ");
    scanf("%u", &n);
    printf("\n Prikaz broja %u u bazi %u: ", n, b);
    ispis_u_bazi(n, b);
    printf("\n");
    return 0;
}
```

Za ulaz: 12, rezultat je:

Prikaz broja 12 u bazi 2: 1100

## Prikaz broja u bazi (nastavak)

Pogledajmo kako **točno** idu pozivi funkcije za  $n = 12$  i  $b = 2$ .

- **Prvi** (vanjski) poziv: `ispis_u_bazi(12, 2)`
  - lokalno:  $n = 12$ ,  $n / b = 6$ , znamenka  $n \% b = 0$ .
- **Drugi** poziv: `ispis_u_bazi(6, 2)`
  - lokalno:  $n = 6$ ,  $n / b = 3$ , znamenka  $n \% b = 0$ .
- **Treći** poziv: `ispis_u_bazi(3, 2)`
  - lokalno:  $n = 3$ ,  $n / b = 1$ , znamenka  $n \% b = 1$ .
- **Četvrti** poziv: `ispis_u_bazi(1, 2)`
  - lokalno:  $n = 1$ ,  $n / b = 0$ , znamenka  $n \% b = 1$ .
- **Peti** poziv: `ispis_u_bazi(0, 2)` — odmah se vrati!

Uočite da **ispis** znamenke  $n \% b$  ide unatraske, **nakon** povratka iz prethodnog poziva — baš u tom je “štos”!

## Prikaz broja u bazi — zadaci

**Zadatak.** Ako je  $n = 0$ , onda naša funkcija **ne piše** ništa (nema znamenki). Modificirajte funkciju tako da napiše znamenku  $0$  za  $n = 0$ . Oprez,

- to hoćemo **samo** kad je **ulazni** broj  $n$  baš jednak  $0$ , tj.
- ne želimo** dodati vodeću **nulu** svim ostalim brojevima  $n > 0$ !

**Zadatak.** Napravite proširenje na **veće** baze, tako da

- znamenke mogu biti i **slova** (to ide do trideset i nešto),
- ili se znamenke u bazi  $b$  **pišu** kao **dekadski** brojevi, ali ih onda **odvajamo** prazninom.



# Rekurzivne funkcije — pravi primjeri

Pravi primjeri **rekurzivnih** algoritama i funkcija su:

- quicksort i mergesort algoritmi za sortiranje,
- Hanojski tornjevi,
- particije broja u pribrojnice.

Sve ove algoritme napraviti ćemo u **drugom** semestru — većinu odmah na **početku**, a **mergesort** kad dođemo na **vezane liste**.

Još nekoliko primjera **rekurzivnih** algoritama:

- obrada **binarnih stabala** i drugih sličnih struktura,
- **sintaktička** analiza programa, po gramatičkim pravilima jezika (tzv. “parser”).

# Funkcije s varijabilnim brojem argumenata

Već ste vidjeli da funkcije `scanf` i `printf`

- imaju **varijabilni broj** argumenata.

Datoteka zaglavlja `<stdarg.h>` sadrži niz **definicija** i **makro naredbi** koje i **nama** omogućavaju

- pisanje funkcija s **varijabilnim** brojem argumenata.

Opširnije u skripti i knjizi **KR2**.

# Dodatak o funkcijama

# Sadržaj

- Dodatak o funkcijama:
  - Zadaci i varijacije.
  - Binomni koeficijenti i Pascalov trokut.
  - Obično i binarno potenciranje realnog broja.

# Zadaci iz funkcija na cijelim brojevima

Zadaci za funkcije s cijelim brojevima:

- Prebacite u funkcije ostale primjere od prošli puta.
- Najmanji djelitelj broja, strogo veći od 1 (ako ga ima).
- Najveći djelitelj strogo manji od broja (ako ga ima).
- Provjera je li broj prost ( $n = 1$  nije prost).
- Najmanji/najveći prosti faktor broja (ako ga ima).
  - Dodatak: vratiti i pripadnu potenciju iz rastava broja na proste faktore, kroz “varijabilni” argument, preko pokazivača.
- Najmanji prosti broj  $p$  veći od zadanog.
- Najveći prosti broj  $p$  manji od zadanog (ako ga ima).

# Zadaci iz funkcija na cijelim brojevima (nast.)

- Broj koji se iz  $n$  dobiva cikličkom rotacijom njegovih binarnih znamenki za  $k$  mjesta udesno, odnosno, ulijevo.
  - Realizacija dijeljenjem/množenjem potencijama od 2,
  - ili operatorima  $\ll i \gg$ .

# Zadaci iz funkcija na realnim brojevima

Zadaci za funkcije s realnim brojevima:

- Najveći cijeli broj manji ili jednak od  $x$ , tj.  $\lfloor x \rfloor$ .
- Najmanji cijeli broj veći ili jednak od  $x$ , tj.  $\lceil x \rceil$ .

U matematičkoj biblioteci `<math.h>` postoje funkcije za to. Zovu se `floor` i `ceil`, ali

- vraćaju vrijednost tipa `double`, a ne tipa `int`!

Razmislite kako biste signalizirali grešku ako je ulazni broj  $x$  tipa `double`, a korektni rezultat nije prikaziv u tipu `int`.

- Razlomljeni dio (nenegativnog) realnog broja  $x$ , tj. broj  $x - \lfloor x \rfloor$ .

# Zadaci iz funkcija na realnim brojevima (nast.)

- Potenciranje cjelobrojnim eksponentom, tj. za zadane  $x$  i  $n$ , treba vratiti  $x^n$  (uz kontrolu greške). Probajte to napraviti bez funkcije `pow` iz zaglavlja `<math.h>`.

- Spora varijanta je “ponovljeno množenje”. Na pr.

$$x^6 = x \cdot x \cdot x \cdot x \cdot x \cdot x.$$

- Puno brža varijanta je tzv. “ponovljeno kvadriranje i množenje”, ili “binarno potenciranje” — iz binarnog zapisa eksponenta  $n$ . Na pr.,  $6 = (110)_2$ , pa je

$$x^6 = ((x^2) \cdot x)^2 = (x^2) \cdot (x^2)^2.$$

Pogledajte vježbe (zadatak 8.5.6.) i nastavak dodatka.



# Binomni koeficijenti i Pascalov trokut

# Binomni koeficijenti i Pascalov trokut

**Primjer.** Treba napisati funkciju koja ima dva cjelobrojna argumenta  $n$  i  $k$  (tipa `int`). Funkcija treba izračunati i vratiti binomni koeficijent

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Binomni koeficijent je korektno definiran za  $n \geq 0$  i  $0 \leq k \leq n$ . Zato provjeravamo ulazne vrijednosti.

🔴 U slučaju greške, vraćamo vrijednost 0.

Glavni program treba ispisati Pascalov trokut za  $n \leq 20$ .

🔴 U liniji s indeksom  $n$  nalaze se brojevi  $\binom{n}{k}$ , za sve vrijednosti  $k = 0, \dots, n$ .

## Binomni koeficijent (nastavak)

Binomni koeficijenti su **doobar** primjer problema u kojem

- treba voditi računa o veličini i **prikazivosti** rezultata u cjelobrojnoj aritmetici računala.

Znamo da **faktorijele** vrlo brzo **rastu**. Zato originalna formula

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

**nije dobra**, jer već **13!** **nije** prikaziv u tipu **int** (na **32** bita).

Puno bolje je **skratiti jedan** od faktora iz nazivnika, **k!** ili **(n - k)!** — sigurno je i u brojniku. Samo **koji?**

- **Veći** od ta dva, naravno!

## Binomni koeficijent (nastavak)

No, umjesto da testiramo i radimo s dvije formule, uočimo da su binomni koeficijenti **simetrični** u  $k$  i  $n - k$ , tj. vrijedi

$$\binom{n}{k} = \binom{n}{n-k}.$$

Zbog toga, po potrebi, možemo **zamijeniti** uloge  $k$  i  $n - k$ .

• Ako je  $k > n - k$ , **zamijenimo** im uloge:  $k = n - k$ .

• Nakon toga je sigurno  $k \leq n - k$ .

Zatim, **skratimo** zadnji (ujedno i **veći**) faktor  $(n - k)!$ , pa je

$$\binom{n}{k} = \frac{n \cdot (n - 1) \cdots (n - k + 1)}{1 \cdot 2 \cdots k}.$$

## Binomni koeficijent (nastavak)

I sad ide **ključna** “sitnica”.

- Kojim **poretkom** izvršavamo aritmetičke **operacije** množenja i dijeljenja u ovoj formuli?

“Očiti” poredak operacija, s **jednim** dijeljenjem na **kraju**,

- pomnoži **brojnik**, pomnoži **nazivnik**, pa onda **podijeli**, **nije** dobar, jer **brojnik** opet **brzo naraste** (v. malo kasnije)!

**Pravi** algoritam je — pomnoži, **podijeli**, pomnoži, **podijeli**, ...

$$\binom{n}{k} = \frac{n}{1} \cdot \frac{n-1}{2} \cdots \frac{n-k+1}{k}.$$

Dodatno, zbog  $k < n - k + 1$ , svi faktori, uključivo i **zadnji**, su **veći** od **1**. Rezultat stalno raste, i to puno **sporije**.

# Binomni koeficijent — funkcija

Odgovarajuća funkcija je (v. `binom_f1.c`):

---

```
/* Funkcija binom(n, k) racuna binomni
   koeficijent n povrh k. */

int binom(int n, int k)
{
    int bin_coef, j;

    /* Provjera granica i signal greske. */
    if (n < 0 || k < 0 || k > n) return 0;

    /* Smanji donji argument. */
    if (k > n - k) k = n - k;
```

## Binomni koeficijent — funkcija (nastavak)

```
if (k == 0) return 1;

bin_coef = n;
for (j = 2; j <= k; ++j)
    bin_coef = bin_coef * (n - j + 1) / j;

return bin_coef;
}
```

---

**Uočite:** Funkcija `binom` vraća **rezultat** tipa `int`.

Ako je tip `long` **veći** od `int`, isplati se staviti

• tip `long` za **vrijednost** funkcije i varijablu `bin_coef`,  
jer binomni koeficijenti mogu biti **veliki**.

## Binomni koeficijent — funkcija (nastavak)

Ključni dio funkcije `binom` možemo realizirati i `for` petljom (v. `binom_f2.c`):

```
bin_coef = n;  
for (j = 2, nmj = n - 1; j <= k; ++j, --nmj)  
    bin_coef = bin_coef * nmj / j;
```

Petlja “paralelno” mijenja dvije varijable:

- `j` za nazivnik — s pomakom `unaprijed`, i
- `nmj` za brojnik — s pomakom `unatrag`.

Operator zarez `,` ovdje služi za izvršenje

- po `dvije` naredbe u `inicijalizaciji` i `pomaku` petlje.

Sekvencijalno izvođenje tih operacija ovdje nije bitno!



## Binomni koeficijent — *Ne tako!*

Na vježbama je napravljen algoritam koji odgovara sljedećem kôdu (v. `binom_fv.c`):

```
bin_coef = 1;
for (j = n; j > n - k; --j)
    bin_coef = bin_coef * j;
for (j = 2; j <= k; ++j)
    bin_coef = bin_coef / j;
```

Uočite da

- 🔴 **prvo** množimo **sve** brojeve u **brojniku** (taj rezultat **brzo raste**),
- 🔴 **zatim** dijelimo sa **svim** brojevima u **nazivniku** (rezultat stalno **pada**).

## Binomni koeficijent — *Ne tako!* (nastavak)

Nažalost, to je samo **malo** bolje od onog ranijeg

- pomnoži **brojnik**, pomnoži **nazivnik**, pa onda **podijeli**, jer **brojnik** odmah **brzo naraste**.

- Dijeljenja ima **puno**, ali su **prekasno** (opet na kraju)!

Ovaj algoritam prvi puta “**umire**” na

$$\binom{18}{9} = 48620.$$

Umjesto toga, vraćeni rezultat je **1276**.

Naša funkcija radi malo dalje. :-)

## Binomni koeficijent — Zadaci

**Zadatak.** Naša funkcija `binom` računa **binomni koeficijent** tako da u brojniku i nazivniku idemo “**unaprijed**” ( $\rightarrow$  po formuli):

---

```
bin_coef = n;  
for (j = 2; j <= k; ++j)  
    bin_coef = bin_coef * (n - j + 1) / j;
```

---

Razmotrite je li **bolje** računati tako da u brojniku idemo “**unazad**”, a u nazivniku “**unaprijed**”:

---

```
bin_coef = n - k + 1;  
for (j = 2; j <= k; ++j)  
    bin_coef = bin_coef * (n - k + j) / j;
```

---

**Pitanje:** Moramo li u nazivniku ići “**unaprijed**”. Zašto?

## Binomni koeficijent — Zadaci (nastavak)

Zadatak. Ispitajte **testiranjem** za koje ulazne brojeve  $n$  i  $k$

• **razne** verzije funkcije **binom** rade **dobro**,  
tj. korektno računaju binomni koeficijent  $\binom{n}{k}$ .

Odgovor (v. **binom\_t.c**). U tipu **int** s **32** bita, obje funkcije

• množenjem “**unaprijed**” i “**unazad**” u brojniku,  
prvi puta **griješe** na istom mjestu

$$\binom{30}{15} = 155117520 \neq -131213633 \quad (\text{vraćena vrijednost}).$$

Usput, prvi **neprikazivi** binomni koeficijenti su  $\binom{34}{16}$  i  $\binom{34}{17}$ .  
Kako biste to **testirali**?

## Binomni koeficijent — Zadaci (nastavak)

**Napomena.** Još “pažljiviji” algoritam na bazi množenja možemo dobiti rastavom svih brojeva na **proste faktore**, tj.

- “praćenjem” **potencija prostih** faktora brojeva u brojniku i nazivniku (svi prosti faktori su  $\leq n$ ).

No, to se **ne isplati** — predugo traje!

**Zadatak.** Probajte sastaviti odgovarajući algoritam.

# Glavni program — Pascalov trokut

Primjer. Pascalov trokut za  $n \leq 10$  izgleda ovako:

```
n = 0          1
n = 1        1  1
n = 2      1  2  1
n = 3    1  3  3  1
n = 4  1  4  6  4  1
n = 5  1  5 10 10  5  1
n = 6  1  6 15 20 15  6  1
n = 7  1  7 21 35 35 21  7  1
n = 8  1  8 28 56 70 56 28  8  1
n = 9  1  9 36 84 126 126 84 36  9  1
n = 10 1 10 45 120 210 252 210 120 45 10  1
```

## Glavni program — Pascalov trokut (nastavak)

Radi jednostavnosti, **Pascalov** trokut ispisujemo poravnato po **lijevoj** strani, s **jednim** razmakom između brojeva:

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
1 10 45 120 210 252 210 120 45 10 1
...
```

# Glavni program — Pascalov trokut (nastavak)

```
#include <stdio.h>
    ...                /* Funkcija binom dodje tu. */
int main(void)
{
    int n, k;

    for (n = 0; n <= 20; ++n) {
        for (k = 0; k <= n; ++k)
            printf("%d ", binom(n, k));
        printf("\n");
    }
    return 0;
}
```



## Pascalov trokut — Zadaci

**Zadatak.** Preuredite **glavni** program tako da ispisuje **Pascalov** trokut **centrirano**, kao u primjeru!

**Zadatak.** Kad napravimo strukturu **polja**, napravite program koji računa **red po red** **Pascalovog** trokuta, koristeći **polje** za jedan red trokuta, i formulu

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

Ova formula vrijedi za  $n \geq 1$  i  $k \in \{1, \dots, n-1\}$ .

**Dokažite** da ovaj algoritam, zato što **zbraja**, **korektno** računa **sve prikazive** binomne koeficijente.

Kako biste **testirali** korektnost? (Pogledajte **binom\_t.c.**)

# Obično i binarno potenciranje realnog broja

# Cjelobrojna potencija realnog broja

**Primjer.** Zadani su **realni** broj  $x$  tipa **double** i **cijeli** broj  $n$  tipa **int**. Treba napisati funkciju koja računa i vraća

•  $n$ -tu **potenciju** broja  $x$ , tj. rezultat je  $x^n$ .

Kad je rezultat  $x^n$  **korektno** definiran, uz  $x \in \mathbb{R}$  i  $n \in \mathbb{Z}$ ?

• Ako je  $x \neq 0$ , onda  $x^n$  postoji za **bilo koji**  $n \in \mathbb{Z}$ .

• Ako je  $x = 0$ , onda je  $x^n = 0$  za  $n > 0$ .

**Dogovor.** U slučaju **greške** u argumentima, kad rezultat **nije** korektno definiran, **vraćamo** rezultat **nula**.

• Dakle, za  $x = 0$  **uvijek** vraćamo rezultat  $x^n = 0$ , što je zgodno olakšanje.

# Cjelobrojna potencija realnog broja (nastavak)

Uočimo da za  $x \neq 0$  vrijedi

$$x^0 = 1, \quad x^{-n} = \frac{1}{x^n},$$

pa nam preostaje izračunati  $x^n$  za  $n > 0$ , odnosno,  $x^{|n|}$ .

Funkcija za cjelobrojnu apsolutnu vrijednost zove se **abs**. Deklarirana je u zaglavlju `<stdlib.h>`, a prototip joj je

---

```
int abs(int)
```

---

Poziv **abs(n)** vraća vrijednost  $|n|$ .

Cijeli algoritam zovemo **int\_pow**, što je skraćeno od

🔴 engl. “integer power” = **cjelobrojna potencija**.

# Cjelobrojna potencija — kostur algoritma

Kostur algoritma `int_pow` onda ima sljedeći oblik:

---

```
    /* Provjera x = 0. */  
    if (x == 0.0) return 0.0;  
  
    /* Zapamti predznak od n. */  
    neg = n < 0;  n = abs(n);  
  
    /* Izracunaj pot = x^n, uz n >= 0. */  
    ...  
  
    if (neg) pot = 1.0 / pot;  
    return pot;
```

---

Računanje  $\text{pot} = x^n$ , za  $n \geq 0$ , realiziramo na dva načina.

## Obično potenciranje — ponovljeno množenje

Spora varijanta rješenja je “ponovljeno množenje” broja  $x$  sa samim sobom, koliko puta treba,

• ovisno o inicijalizaciji za akumulaciju produkta.

Produkt (potenciju) akumuliramo u varijabli `pot`.

Ako želimo da algoritam radi i za  $n = 0$ , onda je zgodno inicijalizirati produkt `pot` na 1 — neutral za množenje.

Ovaj algoritam odgovara računanju potencija  $x^n$  po sljedećoj “rekurzivnoj” relaciji

$$x^n = \begin{cases} 1, & \text{za } n = 0, \\ x \cdot x^{n-1}, & \text{za } n > 0. \end{cases}$$

**Složenost:** treba nam tačno  $n$  množenja.

## Obično potenciranje — funkcija

Funkcija za obično potenciranje (v. `pot_mul.c`):

---

```
double int_pow_mul(double x, int n)
{
    double pot = 1.0;
    int neg, i;

    /* Provjera x = 0. */
    if (x == 0.0) return 0.0;

    /* Zapamti predznak od n. */
    neg = n < 0;
    n = abs(n);
```

## Obično potenciranje — funkcija (nastavak)

```
    /* Potenciranje množenjem. */  
    for (i = 1; i <= n; ++i)  
        pot *= x;  
  
    if (neg) pot = 1.0 / pot;  
    return pot;  
}
```

---



## Obično potenciranje — glavni program

```
int main(void)
{
    double x = 2.0;
    int n;

    n = 5;
    printf(" Potencija %g na %2d = %g\n",
           x, n, int_pow_mul(x, n) );
    n = -5;
    printf(" Potencija %g na %2d = %g\n",
           x, n, int_pow_mul(x, n) );
    return 0;
}
```

# Potenciranje — format ispisa i rezultati

**Ispis.** U oznaci konverzije `%2d`, broj `2` zadaje minimalnu širinu ispisa, tj. minimalni broj znakova koji će se ispisati.

Ako podatak treba:

- manje znakova od zadanog broja, bit će slijeva dopunjen bjelinama do tog broja znakova (osim ako nije zadano drugačije dopunjavanje — tzv. “zastavicama”);
- više znakova od minimalne širine ispisa, bit će korektno ispisan sa svim potrebnim znakovima.

**Rezultati.** Za  $x = 2$  i  $n = \pm 5$ , dobivamo

---

Potencija 2 na 5 = 32

Potencija 2 na -5 = 0.03125

---

# Binarno potenciranje — kvadriranje i množenje

Puno brža varijanta je “ponovljeno kvadriranje i množenje”,  
ili, standardnim imenom, “binarno potenciranje”,  
jer se dobiva iz binarnog zapisa eksponenta  $n$ .

Pretpostavimo da je  $n > 0$  i neka je

$$n = a_k 2^k + a_{k-1} 2^{k-1} + \dots + a_1 2 + a_0 = \sum_{i=0}^k a_i 2^i$$

normalizirani prikaz broja  $n$  u bazi 2. Za znamenke vrijedi

$$a_0, \dots, a_k \in \{0, 1\} \quad \text{i} \quad a_k > 0,$$

a broj znamenki u tom prikazu jednak je

$$k + 1 = \lfloor \log_2 n \rfloor + 1.$$

# Binarno potenciranje (nastavak)

Onda je

$$x^n = x^{\left(\sum_{i=0}^k a_i 2^i\right)} = \prod_{i=0}^k x^{a_i 2^i}.$$

No, **binarne** znamenke  $a_i$  mogu biti **samo** 0 ili 1, pa je

$$x^{a_i 2^i} = \begin{cases} 1, & \text{za } a_i = 0, \\ x^{2^i}, & \text{za } a_i = 1. \end{cases}$$

Dakle, u gornjem produktu ostaju **samo** faktori za znamenke  $a_i = 1$  u binarnom zapisu broja  $n$

$$x^n = \prod_{\substack{i=0 \\ a_i=1}}^k x^{2^i}.$$

## Binarno potenciranje (nastavak)

Faktori u tom produktu dobivaju se **kvadriranjem** prethodnog

$$x^{2^i} = x^{2 \cdot 2^{i-1}} = (x^{2^{i-1}})^2, \quad i > 0,$$

uz početak  $x^{2^0} = x^1 = x$ .

Ako definiramo **novi niz** vrijednosti  $b_i := x^{2^i}$ , za  $i = 0, \dots, k$ , onda članove tog niza računamo po “**rekurzivnoj**” relaciji

$$b_i = \begin{cases} x, & \text{za } i = 0, \\ (b_{i-1})^2, & \text{za } i > 0. \end{cases}$$

Tražena potencija je

$$x^n = \prod_{\substack{i=0 \\ a_i=1}}^k b_i.$$

## Binarno potenciranje (nastavak)

Neka je potencija `pot` inicijalizirana na 1, kao prije. Algoritam **binarnog** potenciranja “**paralelno**” radi sljedeće **tri** operacije

- izdvaja **binarne** znamenke  $a_i$  eksponenta  $n$ ,
- računa članove niza  $b_i$  — **kvadriranjem** u varijabli **kvad**,
- akumulira u **pot** produkt članova  $b_i$  za koje je  $a_i = 1$ .

Na primjer, za  $n = 6 = (110)_2$ , imamo

$$x^6 = (x^2) \cdot (x^2)^2 = b_1 \cdot b_2.$$

**Složenost**: treba nam točno  $k + 1$  množenja za članove  $b_i$  i **najviše** još  $k + 1$  množenja za akumulaciju potencije (kad je  $n = 2^{k+1} - 1$ ).

Dakle, treba nam **najviše**  $2(\lfloor \log_2 n \rfloor + 1)$  množenja!

# Binarno potenciranje — funkcija

Funkcija za binarno potenciranje (v. `pot_bin.c`):

---

```
double int_pow_bin(double x, int n)
{
    double pot = 1.0, kvad = x;
    int neg;

    /* Provjera x = 0. */
    if (x == 0.0) return 0.0;

    /* Zapamti predznak od n. */
    neg = n < 0;
    n = abs(n);
```

## Binarno potenciranje — funkcija (nastavak)

```
    /* Potenciranje kvadriranjem i mnozenjem. */  
while (n > 0) {  
    if (n % 2 == 1) pot *= kvad;  
    kvad *= kvad;  
    n /= 2;  
}  
  
if (neg) pot = 1.0 / pot;  
return pot;  
}
```

---

Za  $x = 2$  i  $n = \pm 5$ , rezultati su, naravno, isti kao i prije.



## Binarno potenciranje — bolji primjer

Ogromna razlika u brzini se baš i ne vidi, sve dok ne probate ovako nešto:  $n = 10^9$  i  $x = 1 + 10^{-9} = 1 + \frac{1}{n}$  (v. `pot_test.c`).

---

```
int main(void)
{
    double x = 1.0000000001;
    int n = 1000000000;

    printf(" Potencija %11.9f na %2d = %11.9f\n",
           x, n, int_pow_mul(x, n) );
    printf(" Potencija %11.9f na %2d = %11.9f\n",
           x, n, int_pow_bin(x, n) );
    return 0;
}
```

---

# Preciznost ispisa realnih brojeva

Pored minimalne širine, moguće je zadati i **preciznost** ispisa. Kod realnih brojeva, **preciznost** je

- (najveći) broj **decimala** (za **%f** i **%e**), odnosno, **vodećih** znamenki (za **%g**), koje će biti ispisane.

Sintaksa:

- **%a.bf** ili **%a.be** ili **%a.bg**, gdje je
  - **a** — minimalna širina ispisa,
  - **b** — preciznost.

Primjer.

- **%11.9f** — znači ispis u **f** formatu s **najmanje 11** znakova, pri čemu je **najviše 9** znamenki iza decimalne točke.

Ispis **bez** specificirane **preciznosti**  $\implies$  **preciznost = 6**.

# Binarno potenciranje — bolji primjer (nastavak)

Rezultati su:

---

Potencija 1.000000001 na 1000000000 = 2.718282052

Potencija 1.000000001 na 1000000000 = 2.718282031

---

Tu se dobro vide razlike u brzini i točnosti.

Na prvi rezultat (`int_pow_mul`) čekam oko 1 sekundu.

- 👉 Vjerovali ili ne, to je jako brzo, jer Intelov compiler još vektorizira petlju u funkciji!

Drugi rezultat (`int_pow_bin`) izađe “trenutno” i nešto je točniji, zbog manje akumulacije grešaka zaokruživanja.

Probajte na svom “kompu”!

## Realna potencija realnog broja — funkcija `pow`

U matematičkoj biblioteci `<math.h>` postoji **opća** funkcija za potenciranje **realnih** brojeva tipa `double`. Prototip je

---

```
double pow(double, double)
```

---

a poziv `pow(x, y)` vraća vrijednost  $x^y$ .

**Zadatak.** Dodajte ispis vrijednosti `pow(x, n)` u glavni program iz prošlog primjera i provjerite **točnost** rezultata.

**Točan rezultat** na 25 decimala za  $x = 1 + 10^{-9}$  i  $n = 10^9$  je

$$x^n = 2.71828 18270 99904 32237 66440.$$