

# Programiranje 1

Funkcije, primjeri, rekurzivne funkcije, zadaci o funkcijama

Matej Mihelčić

Prirodoslovno-matematički fakultet  
Matematički odsjek

09. prosinca 2024.



**Funkcija** je programska cjelina koja:

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

Funkcija u C-u je slična matematičkoj funkciji, ima **domenu**, **kodomenu** i **pravilo**.

**Definicija** funkcije ima oblik:

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

## Definicija funkcije

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 obliha** zagrada, **iza** imena funkcije, nalazi se **deklaracija formalnih argumenata** funkcije (ukoliko postoje). Prvi argument `arg_1` je **lokalna** varijabla **tipa** `tip_1`, ...,  $n$ -ti argument `arg_n` je **lokalna** varijabla **tipa** `tip_n`.

**Formalni argumenti** opisuju domenu funkcije.

- Deklaracije pojedinih argumenata međusobno se **odvajaju zarezom** (**ne radi se o zarez operatoru**). **Prvi dio definicije funkcije (ispred tijela)** se još zove i **zaglavlje funkcije**:

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



**Oble** zagrade ( ) **moraju** se napisati čak i kad **nema** argumenata jer signaliziraju da je riječ o **funkciji**.

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

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

Svaki **blok** ili **složena naredba** u programu se sastoji od **deklaracija objekata** (varijabli, tipova) i **izvršnih naredbi**, koje se izvršavaju **ulaskom** u blok.

**Tijelo funkcije** (koje je ujedno i blok) se počinje izvršavati **pozivom** funkcije.

Za proizvoljan blok 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**, ali **prije prvog korištenja objekta**.

**Formalni** argumenti  $arg\_1, \dots, arg\_n$  deklarirani u zaglavlju

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

deklaracijom postaju **lokalne varijable** u toj funkciji. Lokalne varijable se smiju normalno koristiti, ali lokalno (unutar tijela funkcije).

Za razliku od ostalih lokalnih varijabli, **formalni** argumenti dobivaju **vrijednost** prilikom **poziva** funkcije iz **stvarnih** argumenata navedenih u pozivu.

Slično zadavanju **točke** u kojoj računamo **vrijednost** funkcije.

**Primjer:** pišemo **algoritam** za računanje **vrijednosti**  $\sin(x)$  u proizvoljnoj zadanoj točki  $x$ . 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 funkcije `sin`, moramo zadati 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.

Funkcija kao povratni tip (`tip_podatka`) može vratiti: **aritmetički tip**, **strukturu**, **uniju** ili **pokazivač**, ali **ne može vratiti drugu funkciju** ili **polje**. Funkcija može vratiti **pokazivač na funkciju** ili na **polje** (prvi element polja).

Ukoliko `tip_podatka` nije naveden (dozvoljeno) pretpostavlja se da funkcija vraća podatak tipa `int`. Ova funkcionalnost postoji zbog **kompatibilnosti s prastarim C programima** (prije ANSI/ISO standarda).

## Naredba return

Funkcija **vraća rezultat** svog izvršavanja naredbom **return**. Opći oblik te naredbe je `return izraz`; **Izraz se može, ali ne mora staviti u oble zgrade.**

Ako je **tip vrijednosti** izraza u naredbi **return** **različit** od **tipa podataka** koji funkcija vraća, **vrijednost** izraza bit će **pretvorena** u `tip_podatka`. Naredba `return` **završava** izvršavanje funkcije.

Izvršavanje programa **nastavlja** se na poziciji **poziva** funkcije a **vraćena vrijednost** (ako postoji) se **uvrštava** **umjesto poziva** funkcije.

```
1 double y, phi, r;
2 ...
3 r = 4.23;
4 phi = 2.2; //trigonometrijske funkcije se
5 y = r*sin(phi); //nalaze u <math.h>
```

Ako funkcija **vraća** neku **vrijednost**, povratna **vrijednost** se **ne mora** iskoristiti na mjestu **poziva** već se može **odbaciti**.

**Primjer:** standardne funkcije `scanf` i `printf` također **vraćaju** neku **vrijednost** iako se njihova povratna vrijednost uobičajeno **odbacuje**. Npr. `scanf("%d", &n); printf(" n = %d\n", n);`

Ako nam trebaju **vraćene** vrijednosti, možemo koristiti:

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

**Primjer:** navedena funkcija **pretvara mala slova engleske abecede u velika**. Ostale znakove **ne mijenja**.

- Postoji **jedan formalni argument** (*c*) **tipa char**.
- **Vraćena vrijednost je tipa char**.
- **Ime funkcije je malo\_u\_veliko**.

```
1 char malo_u_veliko(char c)
2 {
3     char znak;
4     znak = ('a' <= c && c <= 'z') ?
5           ('A' + c - 'a') : c;
6     return znak;
7 }
```

**Algoritam:** najjednostavniji algoritam bi imao uvjet i pretvorbu za svaki znak (znak-po-znak). Međutim, postoji bolji način za implementaciju navedenog algoritma.

Za sve standardne kodove znakova (npr. ASCII) u **tipu** char vrijedi:

- **Mala** slova engleske abecede su raspoređena u **nizu** (**jedno za drugim**): 'a', 'b', ..., 'z'.
- Pripadni kodovi **rastu** za vrijednost **jedan** počev od 'a'.
- **Isto** vrijedi i za **velika** slova: 'A', 'B', ..., 'Z'.

Koristimo činjenicu da je **tip** char **cjelobrojni** tip pa postoji **uspoređivanje** i **aritmetika** znakova. Zato provjeru *je li c malo slovo* radimo **usporedbom** znakova: 'a'  $\leq$  c && c  $\leq$  'z'.

Odgovarajuće **malo** slovo ( $c$ ) i **veliko** slovo (znak) moraju biti **jednako pomaknuti** u odnosu na odgovarajuće **početno** slovo, 'a', odnosno 'A'.

Pomake realiziramo koristeći **aritmetiku** znakova.

- Pomak malog slova  $c$  od slova 'a' =  $c - 'a'$ .
- Pomak velikog slova znak od slova 'A' =  $\text{znak} - 'A'$ .
- Izjednačavanjem slijedi:  $\text{znak} - 'A' = c - 'a'$ , odnosno  $\text{znak} = 'A' + c - 'a'$ .

**Prednosti** ovakvog pristupa:

- **Nije bitno** jesu li **velika** slova **ispred malih** ili **obratno**.
- **Ne moramo** znati pripadne **kodove** znakova.

## Poziv funkcije

Funkcija se **poziva** navođenjem: a) **imena** funkcije, b) **liste** (popisa) **stvarnih argumenata** u zagradama.

**Primjer:** **poziv** funkcije `malo_u_veliko` se može realizirati kao:  
`veliko = malo_u_veliko(slovo);` Varijabla **slovo** je jedini **stvarni argument** u **pozivu** funkcije.

**Trenutna vrijednost** te varijable se **prenosi** u funkciju kao **početna vrijednost formalnog argumenta** *c*.

```
1 int main(void) // glavni program
2   { // s pozivom funkcije malo_u_veliko
3     char malo, veliko;
4     printf("Unesite malo slovo: ");
5     scanf("%c", &malo);
6     veliko = malo_u_veliko(malo);
7     printf("\n Veliko slovo = %c\n", veliko);
8     return 0; } // za ulaz: d, dobijemo izlaz: D.
```

**Stvarni** argument funkcije je općenito **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`:

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

**Rezultati:** D i G.

U **drugom** pozivu **nema** pretvaranja u veliko slovo.

## Varijante zapisa funkcije

Funkciju `malo_u_veliko` možemo napisati na **razne** načine. Jedan način je da cijeli **uvjetni** izraz zapišemo u sklopu `return` naredbe. Varijabla **znak** nam tada **ne treba**.

```
1 char malo_u_veliko(char c)
2 {
3     return ('a' <= c && c <= 'z') ?
4         ('A' + c - 'a'): c;
5 }
```

Mogu se ispustiti i **oble** zagrade (niski prioritet uvjetnog operatora). Funkcija `malo_u_veliko` radi **isto** što i standardna funkcija `toupper` iz `<ctype.h>`. U toj datoteci **zaglavlja** postoji niz funkcija za testiranje **znakova**.

## 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.

```
1 char malo_u_veliko(char c)
2 {
3     if ('a' <= c && c <= 'z')
4         return ('A' + c - 'a');
5     else return c;
6 }
```

## Funkcija bez rezultata - tip void

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

Ispis maksimalnog od dva cijela broja.

```
1 void ispisi_max(int x, int y)
2 {
3     int max;
4     max = (x>=y) ? x : y;
5     printf("Maksimalna vrijednost = %d\n", max);
6     return;
7 }
```

Naredba **return** **nema izraz**. Ako je na **kraju** funkcije, može se **izostaviti**. Može se **zadržati** radi **preglednosti**.

Kod **poziva** funkcija bez rezultata **povratna vrijednost** se **ne smije koristiti** na mjestu **poziva** (trebalo bi doći i do **greške** pri prevođenju).

## Funkcija bez argumenata

Definicija funkcije bez parametara.

```
1 tip_podataka ime_funkcije(void)
2 {
3     tijelo funkcije
4 }
```

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

**Napomena:** funkcije bez argumenata **nisu** besmislene. Osim što se glavna funkcija `main` može pozvati bez argumenata, standardna funkcija `getchar` za čitanje jednog znaka (sa standardnog ulaza) **nema** argumenata.

Poziv funkcije bez argumenata.

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

Zagrade `()` su **obavezne** - omogućavaju detekciju funkcije.

## Deklaracija funkcije

Do sada smo **odvojeno** pisali **funkciju** i **glavni program (main)** u kojem se **poziva** funkcija. U nastavku definiramo **redosljed pisanja funkcija** u programu.

Svaka bi funkcija **prije** prvog **poziva** u programu **trebala** biti **deklarirana** navođenjem **prototipa**. Mogućnost da se deklaracija ne navede je ostavljena zbog **kompatibilnosti** sa **starim C** programima. Mi ćemo **obavezno** deklarirati funkciju **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,
- **tipu vrijednosti** kojeg funkcija **vraća**

Ako je funkcija **definirana u istoj datoteci** u kojoj se **poziva**, **prije** prvog **poziva** te funkcije, onda **definicija** služi i kao **deklaracija** - posebna deklaracija **nije potrebna**.

U **ostalim** slučajevima funkcija se **mora** posebno **deklarirati**. Slučaj u kojem je funkcija definirana u drugoj datoteci ostavljamo za kolegij Prog2.

Kao **primjer** funkcije koristimo varijantu funkcije `ispisi_max` koja ispisuje **maksimalni** od **dva realna** broja (tipa `double`).

## Primjer funkcije bez obvezne deklaracije

ispisi\_max je **definirana prije** prvog poziva u funkciji main.

```
1 #include <stdio.h>
2 void ispisi_max(double x, double y){
3     double max;
4     max = (x >= y) ? x : y;
5     printf("Maksimalna vrijednost = %g\n", max);
6     return; }
7
8 int main(void){
9     double x, y;
10    printf("Unesite dva realna broja: ");
11    scanf("%lg %lg", &x, &y);
12    ispisi_max(x, y); //prevoditelj zna da je ovo
13 //funkcija koja prima dva argumenta tipa double
14 //i ne vraca nista
15    return 0; }
```

## Deklaracija ili prototip funkcije

Ako **definiciju** funkcije smjestimo **nakon poziva** funkcije, onda tu funkciju moramo **deklarirati prije** prvog **poziva**.

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

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

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

**Imena** argumenata  $arg_1, \dots, arg_n$  se mogu **izostaviti** jer se **tip** argumenta vidi i bez njih.

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

**Deklaracija** objekata **istog** tipa **vrijednosti** mogu se **spojiti** kao i za obične varijable.

**Primjer:**

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

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

**Deklaracija** se obično piše na **početku** datoteke ili **u funkciji** u kojoj je poziv.

## Primjer funkcije s obveznom deklaracijom

**Primjer:** funkcija je **definirana** iza prvog poziva unutar funkcije main a **deklaracija** je **unutar** funkcije main.

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     double x, y;
6     void ispisi_max(double, double); //deklaracija
7     printf("Unesite dva realna broja: ");
8     scanf("%lg %lg", &x, &y);
9     ispisi_max(x,y);
10    return 0;
11 }
```

## Primjer funkcije s obveznom deklaracijom

```
1 void ispisi_max(double x, double y)
2 {
3     double max;
4     max = (x>=y) ? x : y;
5     printf("Maksimalna vrijednost = %g\n", max);
6     return;
7 }
```

Deklaraciju `void ispisi_max(double, double)` smo mogli zapisati i kao `void ispisi_max(double x, double y);`

**Primjer:** Deklaracija funkcije može biti i **izvan** funkcije gdje je poziv (npr. na **početku** datoteke).

**Prednost** globalne deklaracije na **početku** datoteke, **izvan** svih funkcija je ta što se `ispisi_max` može pozvati **u svim** funkcijama **iza** deklaracije. Često se koristi za **sve** funkcije u programu (osim `main`), jer **ne ovisi** o **poretku** pisanja funkcija **iza** toga.

## Primjer funkcije s obveznom deklaracijom

```
1 #include <stdio.h>
2
3 void ispisi_max(double, double); //deklaracija
4
5 int main(void){
6     double x, y;
7     printf("Unesite dva realna broja: ");
8     scanf("%lg %lg", &x, &y);
9     ispisi_max(x,y);
10    return 0; }
11
12 void ispisi_max(double x, double y){
13     double max;
14     max = (x>=y) ? x : y;
15     printf("Maksimalna vrijednost = %g\n", max);
16     return; }
```

**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.

Postoje **dva načina prijenosa** (predavanja) argumenata prilikom **poziva** funkcije:

- prijenos **vrijednosti** argumenta (eng. *call by value*)
- prijenos **adresa** argumenta (eng. *call by reference*)

Kod prijenosa **vrijednosti** argumenata, funkcija prima **kopije** vrijednosti **stvarnih** argumenata i **ne može izmijeniti stvarne** argumente.

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

- **prvo** se izračuna **vrijednost** izraza,
- **zatim** se **vrijednost** izraza **prenosi** u funkciju,
- **zatim** se **prenesena vrijednost kopira** u odgovarajući **formalni** argument.

Kod prijena **adresa** argumenata, funkcija prima **adrese stvarnih** argumenata, dakle funkcija **može izmijeniti stvarne** argumente (**sadržaje** na tim **adresama**).

**Stvarni** argumenti **ne mogu** biti **izrazi** već samo **varijable**, odnosno **objekti** koji **imaju adresu**.

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

- Svaki **formalni** argument je i **lokalna** varijabla u toj funkciji.
- **Stvarni** argumenti u **pozivu** funkcije su **izrazi**.

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 možemo **promijeniti sadržaj** na toj **adresi**, koristeći operator dereferenciranja (\*).

## Primjer prijenosa po vrijednosti

```
1 #include <stdio.h>
2
3 void f(int x){
4     x+=1;
5     printf("Unutar funkcije: x=%d\n", x);
6     return; }
7
8 int main(void){
9     int x = 5;
10    printf("Prije poziva: x=%d\n", x); // x = 5
11    f(x); // x = 6
12    printf("Nakon poziva: x=%d\n", x); // x = 5
13    return 0; }
```

Funkcija  $f$  povećava vrijednost argumenta za 1. Međutim argument je **lokalna** varijabla pa promjena nije vidljiva izvan funkcije.

## Primjer prijenosa po adresi

```
1 void f(int *x){
2     *x += 1;
3     printf("Unutar funkcije: x=%d\n", *x);
4     return; }
5
6 int main(void){
7     int x = 5;
8     printf("Prije poziva: x=%d\n", x); //x =5
9     f(&x); /* Stvarni argument je pokazivac. x = 6 */
10    printf("Nakon poziva: x=%d\n", x); //x = 6
11    return 0; }
```

Povećavamo **sadržaj** na **adresi**  $x$  za 1, stoga je promjena **vidljiva** i **izvan** funkcije  $f$ .

**Pokazivač** (adresa) je **lokalna** varijabla  $x$ , pa njena promjena (promjena adrese) i dalje **nije** vidljiva izvan funkcije  $f$ .

Kod prijenosa **po vrijednosti**, poziv funkcije  $f$  u glavnom programu može sadržavati i izraz kao formalni argument:  $f(x+2)$ ;

Ukoliko se funkciji prenosi **adresa**, poziv **ne smije** biti:  $f(\&(x+2))$  jer izraz nema adresu. Zbog aritmetike pokazivača, poziv smije biti  $f(\&x+2)$ ;

## 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** argumenata **pretvaraju** se u **tip formalnih** argumenata, isto kao i kod pridruživanja.
- **Redoslijed** izračunavanja **stvarnih** argumenata **nije definiran** i ovisi o implementaciji.

**Primjer:** Funkcija `sqrt` iz zaglavlja `<math.h>` ima prototip `double sqrt(double);` **Nakon** `#include <math.h>`, poziv funkcije `sqrt` može biti:

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

Vrijednost izraza  $2*x-3$  je **tipa** `int`. Kod poziva funkcije se: a) prvo ta vrijednost izraza pretvara u `double`, b) **prenosi** se u funkciju.

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

U programu se **moгу** koristiti i funkcije koje **nisu** prethodno **deklarirane** (prijavi se **upozorenje**). Tada:

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

Nikako se ne preporuča korištenje funkcija bez prototipa.

## Primjer funkcije s prototipom

```
1 #include <stdio.h>
2 int f(double); //deklaracija funkcije
3 int main(void){
4     float x = 2.0; /* double const --> float */
5     printf("%d\n", f(2)); /* int --> double */
6     printf("%d\n", f(x)); /* float --> double */
7     return 0;
8 }
9
10 int f(double x) {
11     return (int) x*x; /* int * double --> int */
12 }
```

Isprobajte gornji program za  $x = 2.0$  i  $x = 2.5$ .

Uočite da  $(int)x*x$  **nije isto** što i  $(int)(x*x)$ . Kod  $(int)x*x$  imamo  $int*double$  (prioritet operatora eksplicitne pretvorbe) pa je konačni tip  $double$  (pretvara se u  $int$ ).

## Primjer funkcije bez prototipa

```
1  ... /* Nema prototip za f. */
2  int main(void)
3  {
4      float x = 2.0; /* double const --> float */
5      printf("%d\n", f(2)); /* GRESKA (2 je int) */
6      printf("%d\n", f(x)); /* OK, x --> double */
7      return 0;
8  }
9
10 int f(double x) /* Tip odgovara pretpostavci. */
11 { return (int) x*x; /* int * double --> int */
12 }
```

U gornjem primjeru, funkcija  $f$  stvarno vraća tip `int`. Poziv  $f(2)$  šalje samo 4 bajta u funkciju  $f$  za  $x$  što dovodi do **krivog** rezultata.

## Primjer funkcije bez prototipa

```
1  ... /* Nema prototip za f. */
2  int main(void)
3  {
4      float x = 2.0; /* double const --> float */
5      printf("%d\n", f(x)); /* OK, x --> double */
6      return 0;
7  }
8  double f(double x)
9  /* Upozorenje kod prevodjenja: redefinicija tipa
10 povratne vrijednosti iz int u double. */
11 { return x*x;
12 }
```

U gornjem primjeru, funkcija  $f$  stvarno vraća tip `double` a ne `int`.

## Broj znamenki broja

**Primjer:** treba pronaći **broj znamenki** nenegativnog cijelog broja  $n$  u zadanoj bazi  $b$ .

```
1 unsigned int broj_znamenki(unsigned int n,  
2                             unsigned int b)  
3 {  
4     unsigned int broj_znam = 0;  
5     while (n > 0) {  
6         ++broj_znam;  
7         n /= b;  
8     }  
9     return broj_znam;  
10 }
```

Kod realizacije funkcijom, **ponišćavanje** vrijednosti broja  $n$  **nije** problem pošto se radi o lokalnoj kopiji.

## Provjera znamenki broja

**Primjer:** zadan je nenegativan cijeli broj  $n$ . Treba pronaći odgovor na pitanje: postoji li znamenka tog broja koja je jednaka 5 u zadanoj bazi  $b = 10$ .

```
1 int odgovor(unsigned int n, unsigned int b,  
2             unsigned int trazena)  
3 {  
4     while (n > 0) {  
5         if (n % b == trazena)  
6             return 1;  
7         n /= b;  
8     }  
9     return 0;  
10 }
```

Za prekid petlje umjesto `break`, koristimo `return` s odgovorom. Realizirajte funkcijom i preostale algoritme na cijelim brojevima (spomenute na prethodnom predavanju).

Programski jezik *C* **dozvoljava rekurzivne funkcije** (da funkcija **poziva samu sebe**).

U pravilu su **rekurzivni algoritmi kraći** ali **izvođenje traje dulje** (ponekad i **znatno dulje** ukoliko se isti potproblem rješava više puta).

**Napomena:** svaki **rekurzivni** algoritam **mora** imati **nerekurzivni** dio koji omogućava **prekidanje** rekurzije. Najčešće se realizira naredbom `if` u **inicijalizaciji** rekurzije.

## Primjer rekurzivne funkcije za računanje faktorijela

**Primjer:** radimo rekurzivnu funkciju za računanje faktorijela:

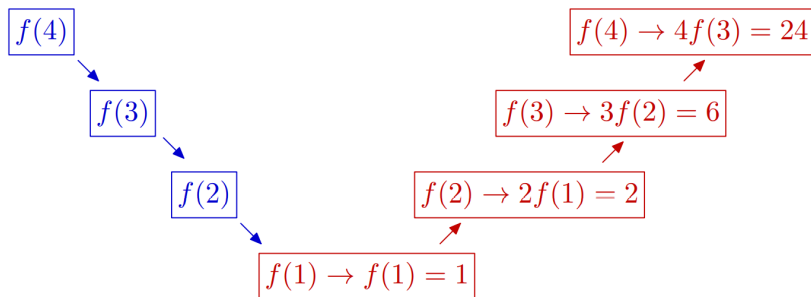
$$n! = 1 \cdot 2 \cdot \dots \cdot (n - 1) \cdot n = n \cdot (n - 1)!$$

```
1 long int fakt(int n)
2 {
3     if (n <= 1)
4         return 1L;
5     else
6         return n * fakt(n - 1);
7 }
```

**Nedostaci** ovakvoga rješenja: a) pozivi su **linearni**, b) **ne računaju ništa kompleksno**. Potrebno je  **$n$  poziva** funkcije da se izračuna  $n!$ .

## Primjer rekurzivne funkcije za računanje faktorijela

Za  $n = 4$  slika rekurzivnih poziva izgleda kao:



## Primjer nerekurzivne funkcije za računanje faktoriijela

**Nerekurzivna funkcija** za računanje faktoriijela radi puno **brže** od rekurzivne (samo jedan poziv funkcije u odnosu na  $n$  poziva kod rekurzije).

```
1 long int fakt(int n)
2 {
3     long int f = 1L;
4     for (; n > 1; --n) f *= n;
5     return f;
6 }
```

Varijanta sa **silaznom** petljom po  $n$ .

**Nerekurzivna funkcija** za računanje faktoriijela sa **uzlaznom** petljom po  $n$ .

```
1 long int fakt(int n)
2 {
3     long int f = 1L;
4     int i;
5     for (i = 2; i <= n; ++i) f *= i;
6     return f;
7 }
```

Drugi **klasični** primjer kad **rekurziju** ne treba koristiti su **Fibonaccijevi** brojevi. Ti brojevi su definirani formulom:  
 $F_0 = 0, F_1 = 1, F_i = F_{i-1} + F_{i-2}$ , za  $i \geq 2$ .

**Primjer:** treba **ispisati prikaz** prirodnog broja  $n$  u zadanoj bazi  $b$  od **vodeće** znamenke do **najniže**.

Dvije bitne činjenice:

- Znamenke broja se puno lakše računaju **straga** (od najniže znamenke prema vodećoj)
- Ispis znamenki mora ići s **lijeva na desno** (obrnuto od smjera lakog računanja znamenaka).

**Rješenje:**

- prvo dođemo do **vodeće** znamenke
- ispisujemo **unatrag**

Za **rješavanje** tog problema koristimo **linearnu rekurziju**.

Svaki poziv funkcije izračuna i **zapamti** svoju znamenku i **piše** tu znamenku tek **nakon rekurzivnog** poziva.

**Algoritam:**

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

Algoritam izvršavamo samo ako je  $n > 0$ , odnosno ima znamenki.  $n == 0$  je **terminalni uvjet**, **uvjet prekida** rekurzije.

Radi jednostavnosti uzimamo da je  $b \leq 10$  (imamo regularne numeričke znamenke).

## Primjer rekurzivne funkcije za prikaz broja u bazi

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

```
1 #include <stdio.h>
2 void ispis_u_bazi(unsigned int n, unsigned int b)
3 {
4     if (n > 0) {
5         ispis_u_bazi(n / b, b);
6         printf("%u", n % b);
7     }
8     return;
9 }
```

Korištenjem tzv. **globalnih varijabli** (koje ćemo obraditi kasnije) možemo izbaciti argument  $b$  koji se ne mijenja.

## Primjer rekurzivne funkcije za prikaz broja u bazi

Glavni program koji poziva rekurzivnu funkciju.

```
1 int main(void) {
2     unsigned int b, n;
3     printf("Upisi nenegativni broj n i bazu b: ");
4     scanf("%u%u", &n, &b);
5     printf("\nPrikaz broja %u u bazi %u:", n, b);
6     ispis_u_bazi(n, b);
7     printf("\n");
8     return 0;
9 }
```

## Primjer rekurzivne funkcije za prikaz broja u bazi

Za **ulaz**: 12 2, **rezultat** je:

Prikaz broja 12 u bazi 2: 1100

Za **ulaz**: 123456 10, **rezultat** je:

Prikaz broja 123456 u bazi 10: 123456

Pogledajmo **pozive** funkcije za  $n = 12$ ,  $b = 2$ .

- (1) Prvi (vanjski) poziv: `ispis_u_bazi(12, 2)`  
(4) lokalno:  $n = 12$ ,  $n / b = 6$ , znamenka  $n \% b = 0$ .
- (2) Drugi poziv: `ispis_u_bazi(6, 2)`  
(3) lokalno:  $n = 6$ ,  $n / b = 3$ , znamenka  $n \% b = 0$ .
- (3) Treći poziv: `ispis_u_bazi(3, 2)`  
(2) lokalno:  $n = 3$ ,  $n / b = 1$ , znamenka  $n \% b = 1$ .
- (4) Četvrti poziv: `ispis_u_bazi(1, 2)`  
(1) lokalno:  $n = 1$ ,  $n / b = 0$ , znamenka  $n \% b = 1$ .
- (5) Peti poziv: `ispis_u_bazi(0, 2)` — odmah se vrati!  
Ispis  $n \% b$  ide unatrag, nakon povratka iz prethodnog poziva.

**Zadatak:** Ako je  $n = 0$ , onda gore definirana rekurzivna funkcija ne ispisuje ništa. Modificirajte rekurzivnu funkciju tako da napiše znamenku 0 za  $n = 0$  (oprez, ne želimo dodati vodeću nulu brojevima  $n > 0$ ).

**Zadatak:** Napravite proširenje na **veće** baze tako da:

- znamenke mogu biti i **slova**
- znamenke u bazi  $b$  se pišu kao **dekadski** brojevi, **odvajamo** ih prazninom (razmakom).

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

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

**Dodatni primjeri** algoritama:

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

Neke funkcije kao npr. `scanf` i `printf` imaju **varijabilni broj** argumenata.

Datoteka zaglavlja `<stdarg.h>` sadrži **niz definicija** i **makro naredbi** koje **omogućavaju** stvaranje funkcija s varijabilnim brojem argumenata.