Složeni tipovi podataka – strukture, unije, polja bitova (nastavak)

Strukture i pokazivači

Vrlo čest slučaj je da se deklariše pokazivač na strukturu:

```
struct strukProm *pokStrukt;
```

Tada se za pristup elementu koristi zaseban operator, **->**:

```
pokStrukt->elementStrukture1
```

Prethodno je skraćeni zapis za:

```
(*pokStrukt).elementStrukture1
```

Primer:

```
struct racun *mojRacun;
mojRacun->brojRacuna = 10482;
strcpy(mojRacun->imePrezime, "Dusan Gajic");
mojRacun->stanje = -1077.45;
```

Unija

Statička struktura **varijantnog (promenljivog) polja**

Može da sadrži više elemenata (članova), ali u jednom trenutku **samo jedan član može da sadrži vrednost**

Predstavlja **efikasan način** da se **jedna memorijska lokacija** koristi **za čuvanje različitih tipova**

Rezervisana reč **union**

Memorijska lokacija u kojoj se čuva unija zauzima onoliko prostora koliko i najveći tip elementa unije, tako da ako unija sadrži **char**, **int** i **float** unija će zauzimati onoliko bajtova koliko je potrebno za memorisanje **float** tipa (**nije char+int+float**)

Unija – deklarisanje

Unija se deklariše na sledeći način:

```
union [identifikator]
{
    deklaracija elementa1;
    deklaracija elementa2;
    ...
    deklaracija elementaN;
} [jedna ili više promenljivih date strukture];
```

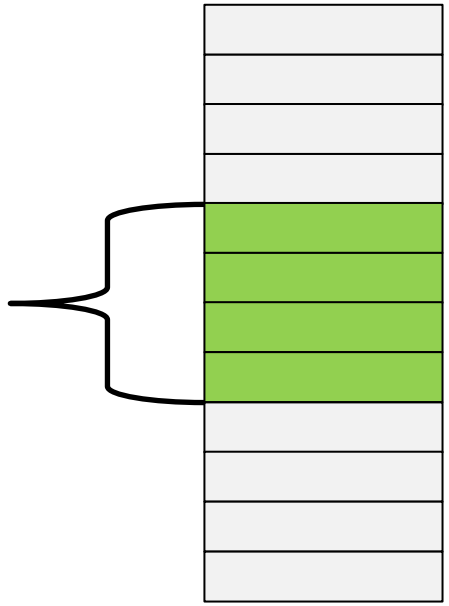
Sve što je važilo za upotrebu identifikatora, deklarisanje promenljivih i deklarisanje tipova strukture, važi i za uniju

Unija – deklarisanje

Ako se pretpostavi da char zauzima 1 bajt, short int zauzima 2 bajta, a float 4 bajta, u tom slučaju će unija zauzimati 4 bajta:

```
union podaci
{
    char znak;
    short int ceoBroj;
    float realniBroj;
} prviPod;
```

union podaci prviPod



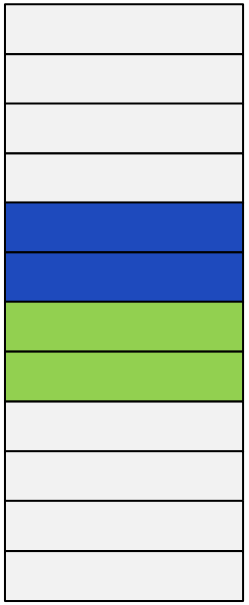
Unija – inicijalizacija

Elementu unije pristupa se na isti način kao i elementu sktrukture preko operatora . (tačka)

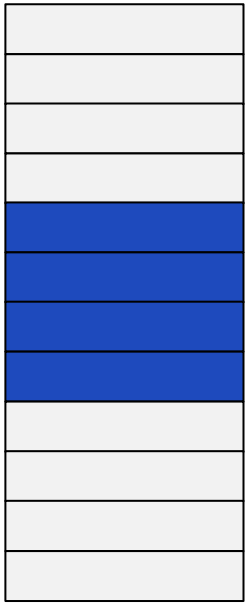
Samo jedan element u jednom trenutku može da bude inicijalizovan

prviPod.znak = 'a';

prviPod.ceoBroj = 10;



prviPod.realniBroj = 9.9f;



Struktura i unija – primer

```
typedef enum {INTEGER, STRING, REAL, POINTER} tTip;
typedef struct {
        tTip tip;
        union {
                int integer;
                char *string;
                float real;
                void *pointer;
        } x;
} vrednost;
....
vrednost vNoviInt(int v) {
        vrednost v;
        v.tip = INTEGER;
        v.x.integer = v;
        return v;
}
```

Polje bitova

Omogućuje da se u strukturi označi koliko koje polje zauzima bitova

Pristupa se **na nivou bitova**

Kao i kod obične strukture navode se elementi i njihov tip

- svaki elemenat se može podeliti na više polja koja zauzimaju određen broj bitova
- broj bitova svakog polja eksplicitno se navodi
- ukupna suma bitova koje zauzimaju polja jednog elementa može biti manja ili jednaka veličini tipa elementa kojem polja pripadaju
- podacima se pristupa preko naziva polja

Predstavlja vezu prema programskim jezicima prve i druge generacije

Treba voditi računa o **prenosivosti koda** (engl. *portability issues*)

Polje bitova – sintaksa

Prvi način za deklarisanje polja bitova:

```
struct [identifikator]
{
    tip polje1 : n1,
         polje2 : n2,
         ...
         poljeN : nN;
} [jedna ili više promenljivih date strukture];
```

Suma mora (**n1+n2+nN**) biti jednaka ili manja od **veličine tipa**

Polje bitova – sintaksa

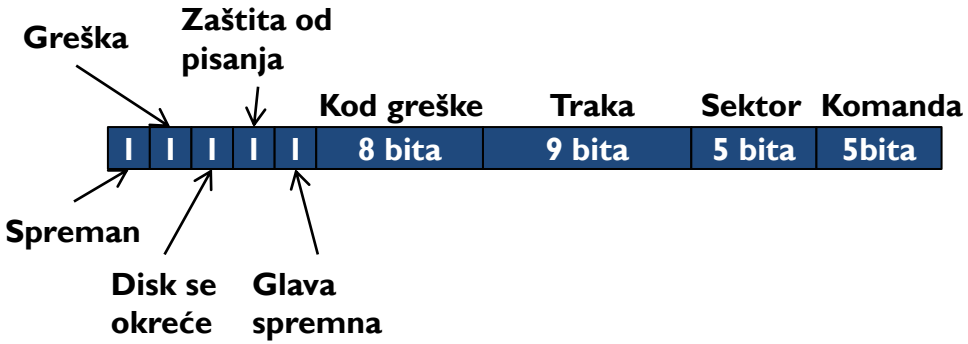
Drugi način za deklarisanje polja bitova:

```
struct [identifikator]
{
   tip   polje1 : n1,
   tip   polje2 : n2,
         ...
   tip   poljeN : nN;
} [jedna ili više promenljivih date strukture];
```

U ovom slučaju ne mora se voditi računa o tome da li je suma (**n1+n2+nN**) jednaka ili manja od veličine **tipa**, ali ako je suma veća od veličine **jednog tipa**, zauzeće se u memoriji dovoljno prostora da stanu **dva tipa** do svoje pune veličine

Polje bitova – primer

Primer registra UI (ulazno-izlaznog) kontrolera hard diska čija ukupna veličina ne prelazi 32 bita




```
typedef struct  {
    unsigned    spreman : 1;
    unsigned    greska : 1;
    unsigned    diskSeOkrece : 1;
    unsigned    zastitaOdPisanja : 1;
    unsigned    glavaSpremna : 1;
    unsigned    kodGreske : 8;
    unsigned    traka : 9;
    unsigned    sektor : 5;
    unsigned    komanda : 5;
} tHDRegistar;
```

Polje bitova – primer

Ako se pretpostavi da postoji konstanta DISK_REGISTER_MEMORY u operativnom sistemu (u kojoj se nalazi adresa registra za upravljanje diskom), onda se za pristup disku može napraviti promenljiva **diskReg**

```
tHDRegistar *diskReg = (tHDRegistar *) DISK_REGISTER_MEMORY;
// definisati sektor i traku iz koje se čitaju podaci, kao i tip operacije – READ
diskReg–>sektor = noviSektor;
diskReg–>traka = novaTraka;
diskReg–>komanda = READ;
// čekanje na kraj operacije, kada disk postaje spreman
while (!diskReg–>spreman);
// provera da li je došlo do greške
if (diskReg–>greska) {
    // otkrivanje vrste greške na osnovu njenog koda
    switch(diskReg–>kodGreske)
    .....
}
```

Dinamička alokacija memorije u jeziku C

Dinamički nizovi u jeziku C

Kada postoji potreba za dinamičkom alokacijom homogenog prostora

Veličina niza se alocira dinamički na zahtev programa u vreme njegovog izvršavanja, a u skladu sa potrebama korisnika

Primer alokacije i rada sa dinamičkim nizom u C-u:

```
int i, vel = 5;
int* niz = (int*) calloc(vel, sizeof(int));
if (niz == NULL)
    printf("Nije uspela alokacija memorije!\n")
else{
    // rad sa dinamičkim nizom
    for (i = 0; i < vel; i++)
        niz[i] = /* nesto */ ...
        ...
}
...
free(niz);
```

Funkcije za dinamičku alokaciju

Funkcije za dinamičku alokaciju iz **stdlib.h**:

malloc funkcija, alocira memoriju tražene veličine i vraća pokazivač na alociranu memoriju (ako je uspelo) ili NULL (ako alociranje nije uspelo)

```
int *nizInt = (int*)malloc(10*sizeof(int));
```

- **sizeof** je operator za određivanje količine memorije koju zauzima neka promenljiva ili tip

calloc funkcija, alocira blok kontinualne memorije za **N** elemenata određene veličine, i za razliku od **malloc** postavlja vrednosti svih lokacija na 0, vraća pokazivač na alociranu memoriju (ako je uspelo) ili NULL (ako alociranje nije uspelo)

```
double *nizDbl = (double*)calloc(10, sizeof(double));
```

Funkcije za dinamičku alokaciju

realloc funkcija, pokušava da proširi alociranu memoriju tako da odgovara traženoj veličini, vraća pokazivač na alociranu memoriju (ako je uspelo) ili NULL (ako alociranje nije uspelo)

```
double *nizDbl = (double*) realloc(nizDbl , 15* sizeof(double));
```

Kada se završi sa radom, alociranu memoriju je potrebno osloboditi, C program to ne radi automatski

free funkcija, koristi se za oslobađanje (dealociranje) dinamički zauzete memorije (**malloc, calloc, realloc**), ne vraća nikakvu vrednost

```
free (nizInt);
free (nizDbl);
```

Pravilno korišćenje realloc()

Nikada ne treba koristiti realloc na sledeci način:

```
niz = realloc(niz, novaVelicina);
```

Ako realloc ne uspe, gubi se originalni pokazivač, a realloc ne oslobađa originalno zauzetu memoriju (ne poziva free()), pa dolazi do curenja memorije (engl. *memory leak*)

Dobra praksa je koristiti funkciju realloc na sledeći način:

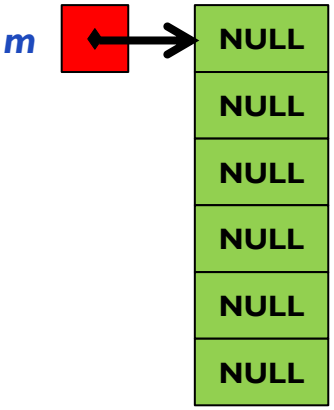
```
tmp = realloc(niz, novaVelicina);
if (tmp == NULL) {
    // realloc nije uspeo, niz je jos uvek validan, ispisati gresku
} else {
        niz = tmp;
}
```

Dinamička matrica

Primer dinamičke matrice:

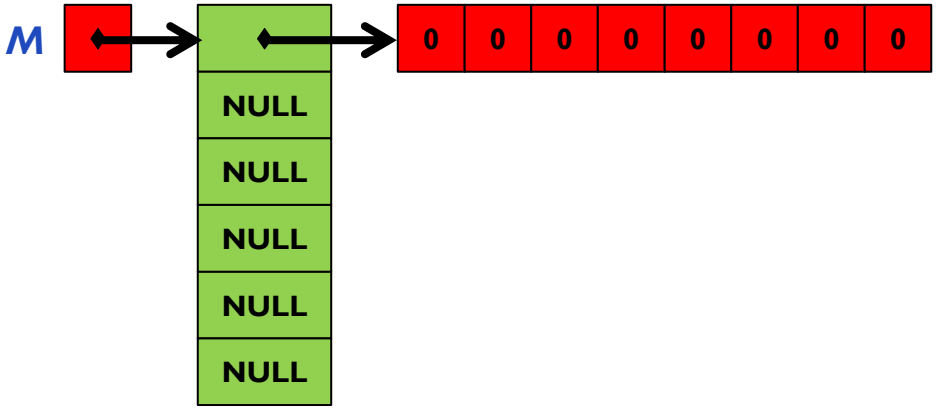
(a) Kreiranje osnove za vel=6

```
m = (int**) calloc(vel, sizeof (int*));
```



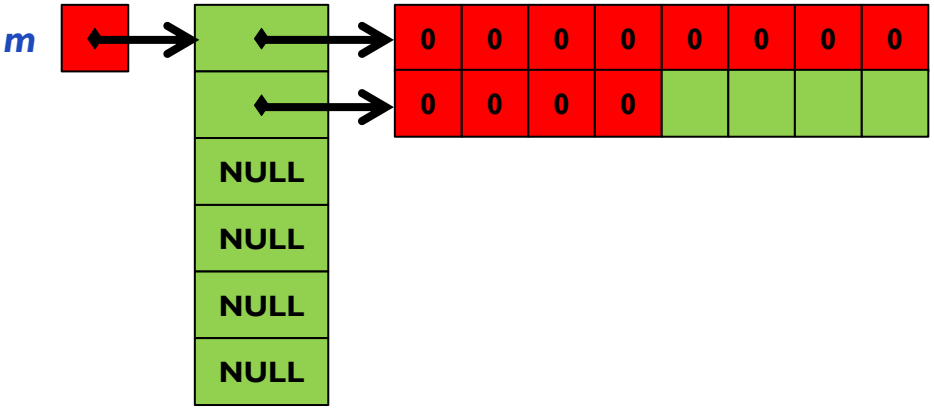
(b) Prva vrsta od k1=8 elemenata

```
m[0] = (int*) calloc(k1, sizeof (int));
```



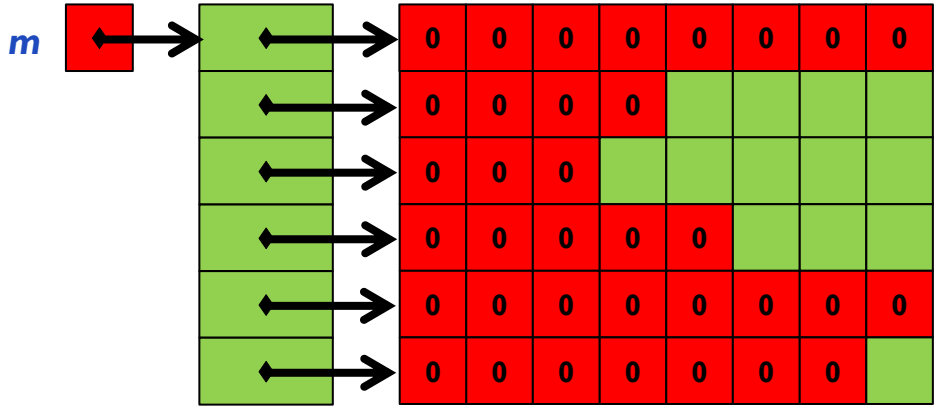
(c) Druga vrsta od k2=4 elementa

```
m[1] = (int*) calloc(k2, sizeof (int));
```



(d) vel-ta vrsta od kvel=7 elemenata

```
m[vel–1] = (int*) calloc(kvel, sizeof (int));
```



Funkcije

Osnovno o potprogramima

Omogućavaju dekompoziciju problema na manje, lakše rešive probleme

Implementiraju **semantički zatvoren zadatak**, tako da se mogu koristiti u drugom rešenju

Obezbeđuju **modularnost** i ponovno **korišćenje koda**

Generalizuju često korišćeni deo koda, čine program značajno **razumljivijim** i **čitljivijim**,

Pružaju dodatni nivo apstrakcije (možemo koristiti funkciju, a da ne znamo detalje njene implementacije) – **razdvajanje interfejsa od implementacije**

Dva tipa – **procedure** (skup naredbi) i **funkcije** (imaju povratnu vrednost)

Osnovno o funkcijama

Implementiraju potprograme u jeziku C

Identifikuju se **nazivom**

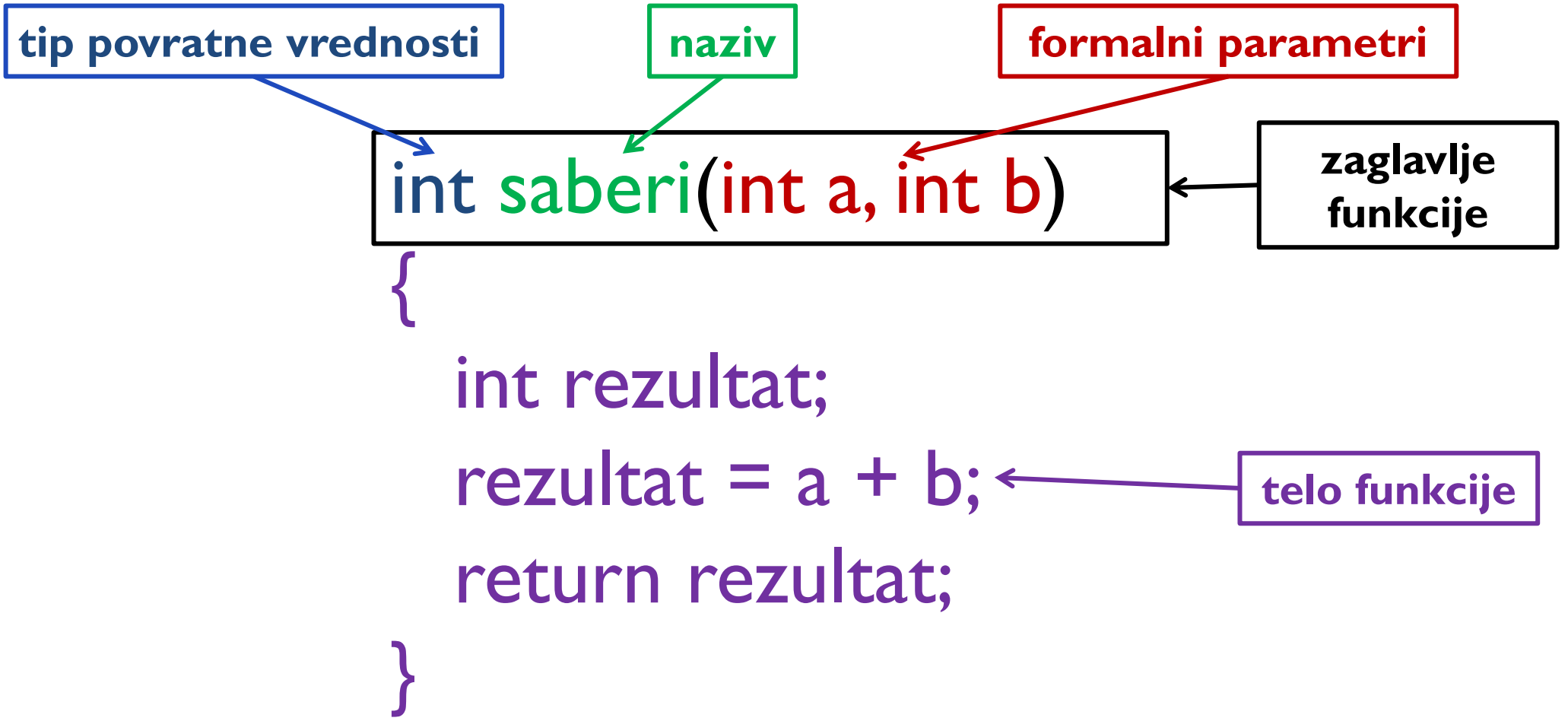
Potrebne podatke za rad dobijaju putem **ulaznih** i/ili **ulazno/izlaznih** parametara

Rezultate rada prosleđuju **nazivom** i/ili **izlaznim** parametrima

Mogu biti **ugrađene** (deo pratećih C zaglavlja) ili implementirane u samom programu

Telo funkcije predstavlja njenu **implementaciju**

Deklarisanje funkcije



Povratna vrednost funkcije

Funkcija svojim nazivom vraća vrednost

Ova vrednost postaje vrednost izraza u kome se nalazi poziv funkcije

Tip povratne vrednosti u zaglavlju (prototipu, deklaraciji) funkcije određuje kog tipa će biti vrednost koju funkcija vraća

Ako funkcija ne vraća nikakvu vrednost, povratni tip je **void**

Povratna vrednost funkcije

Ako funkcija vraća vrednost, u telu funkcije se mora nalaziti **return** iza koje sledi izraz čiji se rezultat vraća kao povratna vrednost funkcije

return vraća vrednost tipa koji odgovara tipu povratne vrednosti koji je naveden u zaglavlju funkcije (moraju se slagati)

Poziv naredbe return završava izvršavanje funkcije. Sav kod posle return se ignoriše!

Može biti više **return** naredbi, ali prva na koju se naiđe završava izvršavanje programa i vraća vrednost

Ako funkcija ne vraća vrednost, ne mora sadržati **return**

Povratna vrednost – primeri

```
void f1( );              // funkcija nema povratnu vrednost

int f2( );               // povratna vrednost tipa int
short f3( );             // povratna vrednost tipa short int
unsigned int f4( );      // povratna vrednost tipa unsigned int
char f5( );              // povratna vrednost tipa char

float f6( );             // povratna vrednost tipa float
double f7( );            // povratna vrednost tipa double

int *f8 ( );             // povratna vrednost tipa int *
                            (pokazivač na int)
float *f9 ( );           // povratna vrednost tipa float *
                            (pokazivač na float)
```

Naziv funkcije

Identifikuje funkciju

Mora biti **jedinstven**

Koristi se za pozivanje funkcije u okviru programa

Trebalo bi da ima **semantičko značenje**, tako da se iz naziva lako identifikuje zadatak funkcije, primeri:

sumirajNiz, pronadjiMax, ucitajMatricu

Parametri funkcije

Lista parametra služi da funkcija dobije podatke potrebne za rad ili da prosledi rezultate svog rada

Vrednost parametara je potrebno definisati (saopštiti funkciji) prilikom svakog poziva

Postoje funkcije koje nemaju parametre

Svaki parametar deklariše **lokalnu promenljivu** koja je vidljiva samo u telu funkcije

Vrednost ove promenljive postavlja se na vrednost koja je saopštena prilikom prilikom poziva funkcije

Prestaje da važi neposredno nakon izlaska iz funkcije

Parametri funkcije

Parametar funkcije nije isto što i promenljiva čija vrednost se prosleđuje funkciji

Prilikom poziva funkcije, **stvarni parametri** moraju se poklapati sa **formalnim parametrima** po:

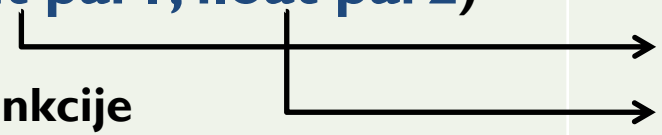
- **broju,**
- **tipu podataka parametra, i**
- **moraju biti u odgovarajućem redosledu**

Nazivi formalnih i stvarnih parametara ne moraju se poklapati!

Parametri funkcije

Deklaracija funkcije	Pozivanje funkcije

int fun1(int par1, float par2)
{
  // telo funkcije
}
	// kod iz kojeg se poziva funkcija
	int arg1 = 21;
	float arg2 = 22.23;
	int rez;
	rez = fun1(arg1, arg2);



- **Formalni parametri** se navode u deklaraciji funkcije i nemaju konkretne vrednosti
- **Stvarni parametri** se navode u pozivu funkcije i prenose funkciji konkretne vrednosti

Prenos parametara funkcije

C funkcija dozvoljava prenos parametara samo po vrednosti

Mehanizam steka i stek frejma

Prilikom poziva funkcije, vrednost promenljive kopira se u parametar funkcije (koliko god ta promenljiva zauzimala memorije), te funkcija nema pristup promenljivoj

Zbog toga, ako funkcija i promeni vrednost jednom od parametara, to ne utiče na promenljivu čija vrednost je prosleđena funkciji

Prenos po vrednosti

Primer kada se vrednost promenljive ne menja:

Pozivanje funkcije

```
// ...

int a = 21;

r = fun1(a);

printf("%d", a);

// ...
```

Funkcija

```
int fun1(int pa)
{
    pa *= 10;
    return pa;
}
```

U memoriji

```
a  == 21
pa == 21
pa == 210
a  == 21
```

Prenos parametara funkcije

Na prvi pogled, u C programima ne postoji mogućnost da se deklarišu izlazne i ulazno/izlazne promenljive ?!?

Šta je sa POKAZIVAČIMA?

Ako izvršavanje funkcije treba da ima **bočni efekat**, kao parametar se prenosi pokazivač na promenljivu čijom se vrednošću želi manipulisati

Bočni efekat se koristi i kada je promenljiva suviše velika (zauzima suviše memorijskog prostora) da bi se prenosila po vrednosti

Pošto se prenosi pokazivač, izmene u funkciji su vidljive u promenljivoj

Kako je naziv niza adresa, on se uvek prenosi kao ulazno/izlazni ili izlazni

Prenos po referenci (adresi)

Primer kada se vrednost promenljive menja zato što je prenos izvršen po referenci (adresi):

Pozivanje funkcije

```
// ...
int a = 21;
r = fun1(&a);
printf("%d", a);
// ...
```

Funkcija

```
int fun1(int *pa)
{
    *pa *= 10;
    return *pa;
}
```

U memoriji

```
 a == 21
*pa == 21
*pa == 210
 a == 210
```