Funkcije

Kompjuterski programi sadrže nizove instrukcija koje za cilj imaju izvršavanje određenog zadatka, što u opštem slučaju podrazumeva preuzimanje ulaznih veličina i njihovu obradu u cilju dobijanja određenih rezultata. Imajući ovo u vidu, kompjuterske programe možemo uporediti sa preduzećima koja nabavljaju ulazne sirovine i njihovom obradom dolaze do konačnih proizvoda.

U slučaju malih zanatskih radionica nabavku materijala, obradu i prodaju proizvoda može obavljati samo jedan čovek. Međutim, u velikim preduzećima je neophodno napraviti organizacione jedinice specijalizovane za obavljanje samo određenih zadataka. Na taj način vrši se podela zaduženja i odgovornosti za izvršavanje određenih zadataka, čime se znatno olakšava organizacija posla u preduzeću. Pored toga, zaposleni su specijalizovani za određene zadatke, što znatno povećava efikasnost. Takođe, u slučaju potrebe za promenama u određenim segmentima proizvodnje, ne narušava se funkcionalnost ostalih celina, već se izmene vrše samo u sektoru koji je odgovoran za izvršenje određenog zadatka.

Prilikom rešavanja manjih problema korišćenjem računara, svi zadaci se mogu izvršiti u okviru glavnog programa, baš kao što jedan radnik može obaviti sve zadatke u zanatskoj radionici. Međutim, sa povećanjem složenosti kompjuterskih programa, neophodno je uvesti organizacione koncepte slične onima koji se koriste u velikim preduzećima. Iz tog razloga, potrebno je program podeliti na odgovarajuće logičke i funkcionalne celine, takozvane **potprograme**. Potprogrami omogućavaju organizovanje određenog broja instrukcija u celine koje obavljaju određene zadatak i na koje se možemo pozivati više puta u toku izvršavanja programa.

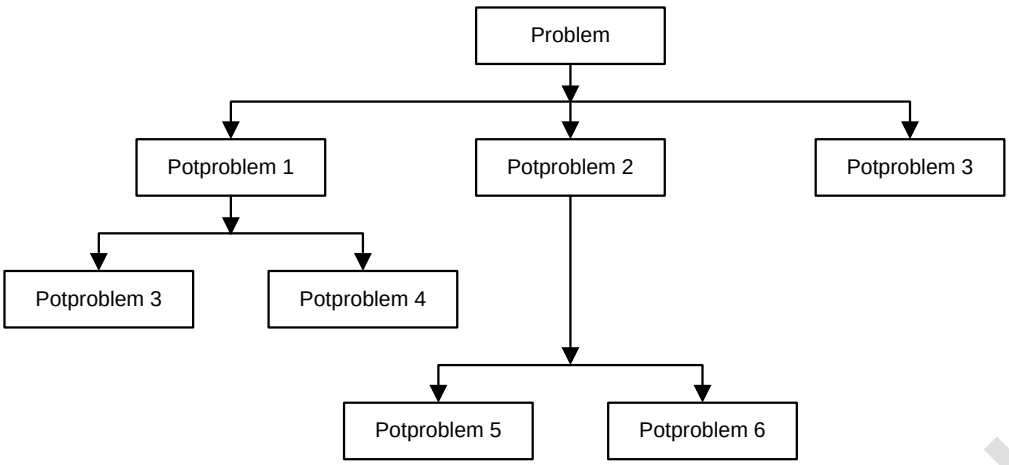
Korišćenje potprograma donosi niz prednosti od kojih možemo izdvojiti sledeće:

- Omogućava koncentrisanje na izvršavanje samo određenog zadatka
- Različiti članovi razvojnog tima mogu razvijati različite potprograme koji se kasnije sklapaju u jednu celinu
- Omogućava izvršavanje istog zadatka na više mesta u programu jednostavnim pozivanjem odgovarajućeg potprograma

Realizacija potprograma u C-u

U programskom jeziku C potprograme je moguće realizovati korišćenjem **funkcija**. **Funkcija** je samostalan deo programa koji obavlja određeni zadatak i preko svog naziva i liste parametara, delu programa iz koga je pozvana može da vraća odgovarajuće rezultate. Svaka funkcija ima jedinstveni naziv preko koga se može pozvati proizvoljan broj puta iz bilo kog dela programa u cilju izvršenja tog zadatka.

Pored toga što se korišćenjem funkcija izbegava nepotrebno ponavljanje delova koda, pa samim tim povećava i čitljivost programa, funkcije omogućavaju realizaciju programiranja "odozgo na dole". Ovaj metod podrazumeva razbijanje problema na manje "potprobleme", a zatim i novodobijenih "potproblema" na još manje celine, sve do trenutka dok se glavni problem ne svede na rešavanje elementarnih problema. Prateći ovaj princip većina programa se može razbiti na određeni broj potproblema kao što je prikazano na slici:



Slika ### Šematski prikaz metoda "odozgo na dole"

Sintaksa

```
<tip_funkcije> <naziv>([lista parametara])
<blok_naredbi>
```

Svaka funkcija mora imati zaglavlje, koje sadrži jedinstveni naziv naveden odmah iza rezervisane reči **tipa funkcije**. Iza naziva funkcije sledi lista parametara unutar oblih zagrada. Za svaki parametar funkcije mora biti naveden i njegov tip, na sličan način kao i kod deklaracija promenljivih. Ukoliko funkcija ne zahteva nikakve ulazne podatke, moguće je ne navesti nijedan parametar.

Nakon definisanja zaglavlja funkcije sledi blok naredbi, odnosno telo funkcije. Kao i svaki drugi blok naredbi, telo funkcije se sastoji od niza komandi ograničenih rezervisanim znakovima **{** i **}**.

U slučaju da funkcija treba da vrati neku vrednost, neophodno je na kraju tela funkcije pomoću rezervisane reči **return** naglasiti koju vrednost funkcija vraća.

return <promenljiva>;

Svaka funkcija može biti pozvana iz glavnog programa (glavne funkcije) ili druge funkcije navođenjem njegovog imena i liste parametara unutar oblih zagrada. Prilikom poziva funkcija potrebno je vrednost koju funkcija vraća dodeliti odgovarajućoj promenljivoj korišćenjem operatora dodele, ili je direktno iskoristiti unutar nekog izraza ili naredbe.

Primer 1

Napisati program koji izračunava i štampa rastojanje između dve tačke čije su koordinate date na ulazu.

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

float rast(float x1, float y1, float x2, float y2)
{
   return sqrt(pow(x2-x1,2) + pow(y2-y1,2));
}

void stampaj(float d)
{
   printf("Rastojanje izmedju tacaka je %10.4f\n", d);
}
```

```
main()
{
   float x1, y1, x2, y2, r;

   printf("Unesite koordinate prve tacke:\n");
   scanf("%f%f", &x1, &y1);


   printf("Unesite koordinate druge tacke:\n");
   scanf("%f%f", &x2, &y2);

   r = rast(x1, y1, x2, y2);

   stampaj(r);
}
```

U prethodnom primeru definisane su dve funkcije. Prva funkcija na osnovu koordinata dve tačke izračunava rastojanje između njih. Druga funkcija vrši štampanje rezultata uz odgovarajući komentar. Druga funkcija nema povratnu vrednost, tako da je tip funkcije **void**.

Nakon unosa koordinata tačaka glavni program poziva funkciju *rast* i šalje joj koordinate tačaka. Kada funkcija završi izračunavanje, ona glavnom programu vraća rastojanje između navedenih tačaka. Glavni program, zatim, dobijenu vrednost dodeljuje promenljivoj *r*. Program dalje poziva funkciju *stampaj* i prosleđuje joj rastojanje *r*, nakon čega funkcija vrši štampanje ove vrednosti uz odgovarajući komentar.

Primer 2

Napisati program koji za *n* vrednosti sa ulaza vrši izračunavanje polinoma $x^5+3x^4-6x^2+2$, bez korišćenja biblioteke sa matematičkim funkcijama.

```
#include <stdio.h>

float stepen(float x, int eksp)
{
   int i;
   float s = 1.0;

   for(i = 0; i < eksp; i++)
      s = s*x;

   return s;
}

float poli(float x)
{
   return stepen(x,5)+3.0*stepen(x,4)-6.0*stepen(x,2)+2;
}

main()
{
   float x;
   int n,i;

   printf("Unesite koliko brojeva zelite:\n");
   scanf("%d", &n);
```

```
   for(i = 0; i < n; i++)
   {
      printf("Unesite %d. broj\n", i + 1);
      scanf("%f", &x);

      printf("Vrednost polinoma za %d. broj je %10.4f\n", i, poli(x));
   }
}
```

U prethodnom primeru su definisane dve funkcije. Funkcija *stepen* izračunava vrednost x^{eksp}, a funkcija *poli* izračunava vrednost polinoma $x^5+3x^4-6x^2+2$. Primetimo da funkcija *poli* vrši izračunavanje polinoma korišćenjem prethodno definisane funkcije *stepen*.

U glavnom programu korisnik unosi *n* realnih brojeva i za svaki od njih se izračunava vrednost polinoma pozivom funkcije *poli*, a dobijeni rezultat štampa na ekranu.

Deklaracija funkcije

Da bi određena funkcija mogla da se pozove iz neke druge funkcije, neophodno je da funkcija koja se poziva bude definisana pre funkcije koja je poziva. Ukoliko raspored funkcija nije takav da je ovaj zahtev obezbeđen, moguće je deklarisati funkciju kako bi se naznačilo da će definicija funkcije koja se poziva uslediti kasnije u kodu. To se postiže tako što se ispred funkcije koja poziva navede samo zaglavlje funkcije koja se poziva, a sama funkcija koja se poziva se navodi kasnije u kodu. Na primer:

```
int f2(...);

void f1(...)
{
   ...
   f2(...);
   ...
}

int f2(...)
{
   ...
}
```

Opseg važenja i životni vek promenljivih

Opseg važenja neke promenljive je deo programa u kome ta promenljiva može biti korišćena. U tom delu programa data promenljiva ima značenje i za nju kažemo da je "vidljiva" u tom opsegu. Na primer, promenljiva deklarisana unutar funkcije ima značenje i vidljiva je samo unutar te funkcije. Takvu promenljivu nazivamo **lokalna promenljiva**. Sa druge strane, **globalna promenljiva** se deklariše van svih funkcija i vidljiva je iz bilo kog dela programa.

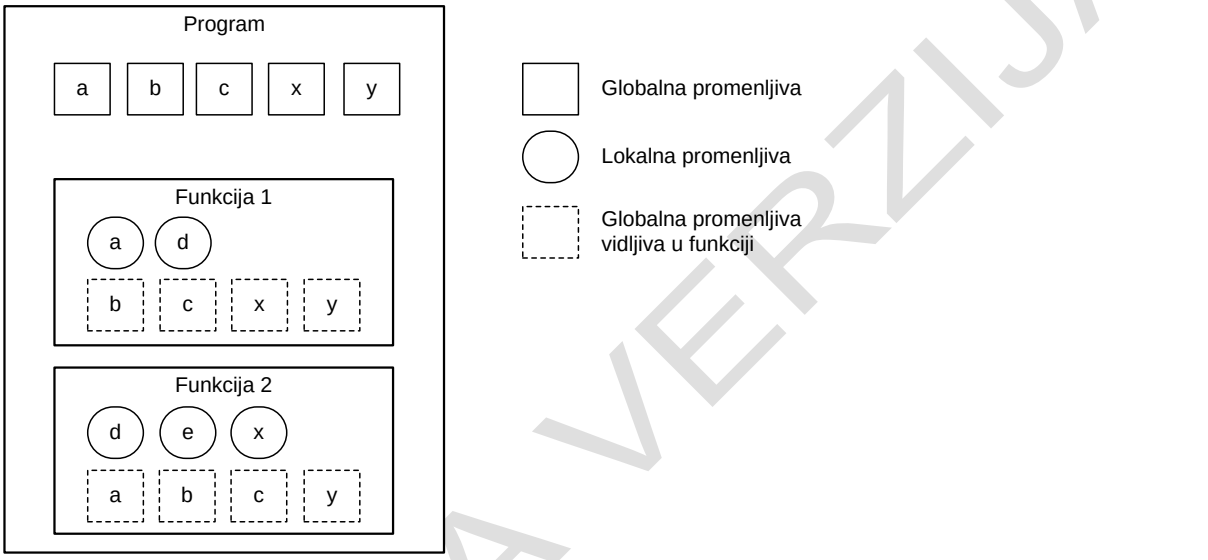
Na Slici ### su šematski prikazani opsezi važenja pojedinih promenljivih. Van svih funkcija su deklarisane globalne promenljive *a*, *b*, *c*, *x* i *y*, koje su vidljive u svim delovima programa pa samim tim i u Funkciji 1 i Funkciji 2.

U Funkciji 1 su deklarisane lokalne promenljive *a* i *d*, koje su vidljive samo u ovoj funkciji i ne mogu se koristiti van nje. Pored ove dve promenljive u Funkciji 1 su vidljive i globalne promenljive *b*, *c*, *x* i *y*. Svaka promena vrednosti ovih promenljivih u bilo kom delu programa je vidljiva i u Funkciji 1. Takođe, promena vrednosti ovih promenljivih u Funkciji 1 izazvala bi promenu njihovih vrednosti i u svim ostalim delovima programa, iz razloga što su to ustvari jedne te iste promenljive. Međutim, globalna promenljiva *a* nije vidljiva u Funkciji 1 iz razloga što istoimena lokalna promenljiva ima prioritet. Iako globalna i lokalna promenljiva *a* imaju

isti naziv, to su zapravo dve različite promenljive. Ukoliko bi u Funkciji 1 došlo do promene vrednosti lokalne promenljive *a*, to ne bi imalo nikakvog uticaja na globalnu promenljivu *a*. Takođe, promena vrednosti globalne promenljive *a* nema uticaja ne vrednost lokalne promenljive *a* u Funkciji 1.

U Funkciji 2 su deklarisane lokalne promenljive *d*, *e* i *x*, koje su vidljive samo u ovoj funkciji i ne mogu se koristiti van njega. Pored ovih promenljivih, u Funkciji 2 su vidljive i globalne promenljive *a*, *b*, *c* i *y*, čije smo osobine opisali na primeru Funkciji 1.

Primetimo da obe funkcije poseduju lokalnu promenljivu *d*, ali zahvaljujući tome što su ove dve promenljive deklarisane u različitim funkcijama, one imaju potpuno odvojen opseg važenja. To praktično znači da promena promenljive *d* u jednoj funkciji nema nikakvog uticaja na istoimenu promenljivu u drugoj funkciji. Iako imaju isti naziv, to su zapravo dve različite promenljive.



Slika ### Šematski prikaz opsega važenja globalnih i lokalnih promenljivih

S obzirom da promenljiva nema značenje van svog opsega, besmisleno je njeno korišćenje van dela programa u kome je deklarisana. U kompajlerskim jezicima se u trenutku prevođenja programa vrši analiza korišćenja promenljivih unutar programa. U slučaju da je pokušano korišćenje neke promenljive van njenog opsega, kompajler prijavljuje grešku.

Životni vek promenljive opisuje u kom delu programa određena promenljiva ima vrednost, odnosno u kojim trenucima izvršenja programa promenljiva započinje i završava svoje postojanje. Na početku životnog veka promenljive, program obezbeđuje deo memorije u kome će biti smeštena njena vrednost. Životni vek promenljive prestaje oslobađanjem prethodno zauzetog dela memorije.

Opseg važenja direktno utiče na životni vek promenljive. Životni vek promenljive obično počinje ulaskom u njen opseg važenja, a prestaje izlaskom iz njega, kako bi se izbeglo nepotrebno trošenje memorije na promenljive koje nisu vidljive u trenutnom opsegu. Ovakve promenljive se često nazivaju i automatske. Pored automatskih promenljivih, u nekim jezicima postoje i takozvane statičke promenljive, čiji životni vek traje i nakon izlaska iz opsega, tako da se ove promenljive mogu ponovo koristiti ukoliko se program vrati u pomenuti opseg.

Dobra programerska praksa podrazumeva pravljenje što manjih opsega važenja pojedinih promenljivih, kako ne bi došlo do slučajnog mešanja promenljivih dva različita dela programa. Iako su globalne promenljive vidljive u svim funkcijama, treba izbegavati njihovo menjanje u funkcijama. Ukoliko funkcije menjaju globalne promenljive, prilikom čitanja

glavnog dela programa veoma je teško uočiti šta se zapravo dešava sa pojedinim promenljivama, što može izazvati neželjene efekte. U slučaju da je potrebno da određena funkcija promeni neku od globalnih promenljivih, najbolje je tu promenljivu proslediti funkciji kao parametar. Na taj način i u glavnom programu postaje jasno da su određene globalne promenljive "poverene" funkciji i da ih ona može promeniti. O načinima prenošenja parametara iz funkcije koja poziva u funkciju koja se poziva biće više reči u narednoj sekciji.

Primer

```
#include <stdio.h>

int a, b, c;
float x, y;

void Funkcija1()
{
   int a, d;

   a = 11;
   d = c + 12;
   b = 13;
   x = 3.14;
}

int Funkcija2()
{
   int d, e, x;

   x = 5;
   d = a+x;
   a = d+1;

   return -1;
}

main()
{
   a = 1;
   b = 2;
   c = 3;
   x = 4.0;
   y = 5.0;

   Funkcija1(); {  Nakon izvrsenja ove linije vrednosti promenljivih su:
                   a=1, b=13, c=3, d nije vidljivo, e nije vidljivo,
                   x=3.14, y=5.0
                }
   Funkcija2(); {  Nakon izvrsenja ove linije vrednosti promenljivih su:
                   a=7, b=13, c=3, d nije vidljivo, e nije vidljivo,
                   x=3.14, y=5.0
                }
}
```

Prenos parametara

Kao što je ranije navedeno, izvršavanje svakog programa u programskom jeziku C započinje izvršavanjem funkcije **main**. Iz tog razloga funkciju **main** često nazivamo **glavni program**, kao što je uobičajeno u nekim drugim programskim jezicima. Radi jednostavnosti pisanja, u

tekstu koji sledi ćemo koristiti termin **glavni program**, ali se navedeni principi mogu primeniti na poziv bilo koje funkcije iz bilo koje druge funkcije.

Do sada smo videli da glavni program ili neka funkcija mogu pozvati određenu funkciju u cilju izvršenja određenog zadatka. Da bi zadatak mogao biti izvršen najčešće je potrebno funkciji proslediti određene podatke koje nazivamo parametrima ili argumentima funkcije.

Takođe, videli smo i da funkcije preko svog imena mogu glavnom programu vratiti neku vrednost. Međutim, u realnim problemima je često slučaj da je potrebno da funkcija glavnom programu vrati više od jedne vrednosti. Vraćanje više od jedne vrednosti moguće je izvršiti preko liste parametara, o čemu će u nastavku biti reči.

Za pravilno korišćenje funkcija veoma je važno potpuno razumevanje funkcionisanja prenosa parametara između glavnog programa i funkcije. Iz tog razloga će u nastavku biti detaljno obrađen ovaj proces.

Svaka funkcija u svom zaglavlju imaju definisanu listu parametara koje zahteva od programa ili funkcije koja ih poziva. Ove parametre nazivamo **formalni parametri**. U većini programskih jezika postoje dve vrste formalnih parametara:

- vrednosni
- promenljivi (varijabilni).

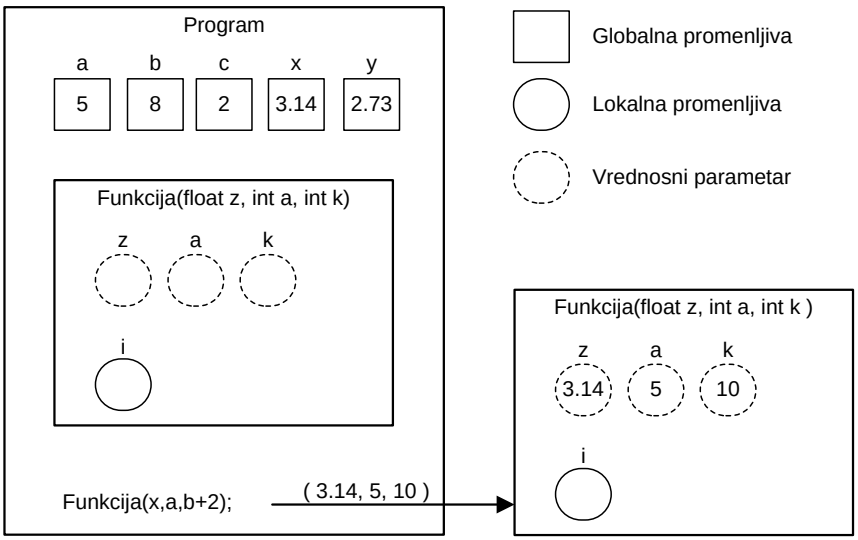
U programskom jeziku C se promenljivi parametri realizuju korišćenjem pokazivačkih promenljivih, o čemu će biti reči u višim kursevima o programskom jeziku C. U okviru ovog kursa se vrednosni i promenljivi parametri razmatraju na nivou koji je uobičajen za većinu programskih jezika, bez ulaženja u mehanizme koji stoje iza toga.

Kada glavni program ili neka funkcija pozove neku funkciju, on joj prosleđuje potrebne vrednosti koje nazivamo **stvarni parametri**. Lista stvarnih parametara mora da odgovara listi formalnih parametara po broju, tipu i redosledu. To znači da glavni program mora funkciji da prosledi tačno onoliko parametara koliko funkcija zahteva, pri čemu parametri moraju da budu istog tipa i navedeni u potpuno istom redosledu kao što su navedeni u zaglavlju funkcije.

Prenos vrednosnih parametara

Razmotrimo šta se dešava prilikom pozivanja funkcije koja zahteva samo vrednosne parametre. U trenutku pozivanja takve funkcije, vrednosti koje su u pozivu navedene kao stvarni parametri se kopiraju u formalne parametre funkcije. Vrednosni formalni parametri funkcije se ponašaju potpuno isto kao i lokalne promenljive te funkcije. Oni su vidljivi samo unutar te funkcije i ne može im se pristupiti iz glavnog programa ili neke druge funkcije. Zahvaljujući tome, bilo kakva promena vrednosti tih promenljivih unutar funkcije, nema nikakvog uticaja na ostatak programa, pa samim tim ni na stvarne parametre koji su prosleđeni funkciji.

Na Slici ### je prikazan postupak prosleđivanja vrednosnih parametara funkciji.



Slika ### Šematski prikaz prenosa vrednosnih parametara

Na Slici ### šematski je prikazan program koji sadrži celobrojne promenljive *a*, *b* i *c* i realne promenljive *x* i *y* sa vrednostima navedenim u kvadratima. U okviru programa je definisana i Funkcija koja ima tri vrednosna parametra i to realni parametar *z* i celobrojne parametre *a* i *k*. Funkcija, takođe, koristi i lokalnu promenljivu *i*.

Glavni program poziva Funkciju korišćenjem linije

```
Funkcija(x,a,b+2);
```

U trenutku poziva vrednosti navedenih stvarnih parametara (*x*, *a*, *b*+2) se prosleđuju Funkciji, i upisuju u formalne parametre (*z*, *a*, *k*). Primetimo da stvarni vrednosni parametri mogu biti promenljive, izrazi i konstante iz razloga što se Funkciji proseđuju samo vrednosti ovih parametara.

Unutar Funkcije promenljive *z*, *a* i *k* se ponašaju kao lokalne promenljive i njihova promena unutar Funkcije neće imati nikakvog uticaja na ostatak programa. Takođe, Glavni program ne može pristupiti ovim promenljivama, a njihove vrednosti može zadati samo prilikom poziva Funkcije. S obzirom da se fiktivni parametri ponašaju kao lokalne promenljive funkcije, njihovi nazivi moraju biti jedinstveni samo unutar Funkcije i nisu ni u kakvoj vezi sa nazivima stvarnih parametara. Tako, vrednost stvarnog parametra *x* se kopira u formalni parametar *z*, vrednost stvarnog parametra *a* se kopira u formalni parametar *a*, a vrednost izraza *b*+2 se upisuje u formalni parametar *k*. Iako stvarni parametar *a* i formalni parametar *a* nose isti naziv, oni međusobno nisu povezani. To su zapravo dve različite promenljive, od kojih je jedna deklarisana unutar Glavnog programa kao globalna promenljiva, a druga kao formalni parametar Funkcije, što znači da su njen opseg važenja i životni vek samo unutar ove funkcije.

Primer

```
#include <stdio.h>

int a, b, c;
float x, y;

void Funkcija(float z, int a, int k)
{
   int i;
   a = 11;
```

```
   for(i = 1; i <= k; i++)
      z = z+1;

   printf("%d, %d\n", a, z);
}

main()
{
   a = 5;
   b = 8;
   c = 2;
   x = 3.14;
   y = 2.73;

   Funkcija(x, a, b+2); { Nakon izvrsenja ove linije vrednosti promenljivih su:
                          a=5, b=8, c=2, x=3.14, y=2.73 }

   printf("%d %d %d %.2f %.2f\n", a, b, c, x, y);
}
```

Prenos promenljivih parametara

U dosadašnjem razmatranju smo imali prilike da vidimo da jedino funkcija može vratiti glavnom programu neku vrednost preko svog naziva. Međutim, u realnim problemima je često slučaj da je potrebno da funkcija glavnom programu vrati više od jedne vrednosti.

U prethodnom delu smo mogli videti kako se vrši prenos vrednosnih parametara iz glavnog programa u funkciju. S obzirom da se vrednosni parametri ponašaju kao lokalne promenljive, ni jedna njihova promena neće biti vidljiva u glavnom programu. Samim tim nije moguće glavnom programu vratiti neku vrednost putem vrednosnih parametara.

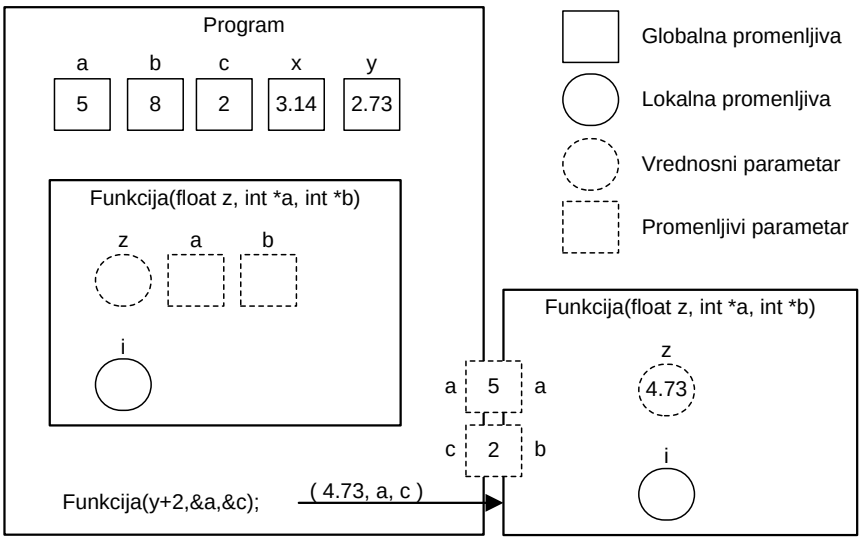
Iz tog razloga postoji i druga vrsta formalnih parametara koje nazivamo promenljivim ili varijabilnim parametrima. Kod ove vrste parametara funkciji se ne prenose vrednosti stvarnih parametara, već njihove adrese. Na taj način i stvarni i formalni parametri ukazuju praktično na istu memorijsku lokaciju, čime se postiže da svaka promena formalnog parametra unutar funkcije direktno menja vrednost odgovarajućeg stvarnog parametra u glavnom programu. Za razliku od *prenosa parametara po vrednosti*, ovaj način prenosa se naziva *prenos parametara po referenci*.

Promenljivi parametri se u programskom jeziku C deklarišu korišćenjem rezervisanog znaka ***** unutar liste parametara. Tako, na primer, funkcija *Test* definisana kao

```
void Test(float z, int *a, int *b);
```

ima jedan realan vrednosni parametar *z* i dva promenljiva parametra *a* i *b*. Promena parametra *z* unutar funkcije neće imati uticaja na promenljive u glavnom programu. Međutim, promena parametara **a* i **b* će direktno značiti i promenu odgovarajućih stvarnih parametara koji su prosleđeni funkciji.

Na Slici ### dat je šematski prikaz prenosa promenljivih parametara:



Slika ### Šematski prikaz prenosa promenljivih parametara

Na Slici ### šematski je prikazan program koji sadrži celobrojne promenljive *a*, *b* i *c* i realne promenljive *x* i *y* sa vrednostima navedenim u kvadratima. U okviru programa je definisana i Funkcija koji ima tri formalna parametra i to realni vrednosni parametar *z* i celobrojne promenljive parametre *a* i b. Funkcija, takođe, koristi i lokalnu promenljivu *i*.

Glavni program poziva Funkciju korišćenjem linije

```
Funkcija(y+2, &a, &c);
```

U trenutku poziva vrednost stvarnog parametara (y+2) se prosleđuje Funkciji, i upisuje u formalni parametar *z*. Pored toga, Funkciji se prosleđuju i stvarni parametri *a* i *c*, koji odogovaraju promenljivim formalnim parametrima *a* i *b*. Za razliku od vrednosnog parametra *z*, u ovom slučaju se ne prosleđuju vrednosti promenljivih *a* i *c*, već njihove adrese u memoriji, korišćenjem rezervisanog znaka **&**. Umesto konkretnih vrednosti, formalnim parametrima *a* i *b* se dodeljuje da ukazuju na istu memorijsku lokaciju prosleđenih promenljivih *a* i *c*. Od tog trenutka promenljiva **a* unutar Funkcije deli istu memorijsku lokaciju sa globalnom promenljivom *a*. Slično i promenljiva **b* deli istu memorisjku lokaciju sa globalnom promenljivom *c*. To praktično znači da će svaka promena promenljive **a* unutar Funkcije istovremeno izazvati promenu i globalne promenljive *a*. Takođe, svaka promena promenljive **b* unutar Funkcije izaziva promenu globalne promenljive *c*. Primetimo da, iako dele isti memorijski prostor, stvarni i formalni parametri funkcije mogu imati iste ili različite nazive, iz razloga što imaju različite opsege važenja.

Korišćenjem opisanog mehanizma moguće je u određenoj funkciji dodeliti vrednosti pojedinim promenljivim parametrima i na taj način direktno promeniti vrednosti više promenljivih u glavnom programu. To praktično omogućava da funkcija glavnom programu vrati više od jedne vrednosti.

Primer

```
#include <stdio.h>

void Funkcija(float z, int *a, int *b)
{
   int i;
   *a = 11;
   *b = *a / 2;
```

```
   for(i = 1; i <= *b; i++)
      z = z+1;

   printf("%d, %d, %f\n", *a, *b, z);
}

main()
{
   int a, b, c;
   float x, y;

   a = 5;
   b = 8;
   c = 2;
   x = 3.14;
   y = 2.73;

   Funkcija(y+2, &a, &c); { Nakon izvrsenja ove linije vrednosti promenljivih su:
a=11, b=8, c=5, x=3.14, y=2.73 }

   printf("%d %d %d %.2f %.2f\n", a, b, c, x, y);
}
```

Primer 2

Napisati program koji za zadati prirodan broj *n* pozivanjem funkcije izračunava sumu i proizvod svih prirodnih brojeva manjih od *n*.

```
#include <stdio.h>

void SumaProizvod(int k, int *suma, int *proizvod)
{
   int i;
   *suma = 0;
   *proizvod = 1;

   for(i = 1; i < k; i++)
   {
      *suma = *suma + i;
      *proizvod = *proizvod * i;
   }
}

main()
{
   int n, s, p;

   printf("Unesite prirodan broj n:\n");
   scanf("%d", &n);

   SumaProizvod(n, &s, &p);

   printf("Suma prirodnih brojeva manjih od %d je %d\n", n, s);
   printf("Proizvod prirodnih brojeva manjih od %d je %d\n", n, p);
}
```

Rekurzivne funkcije

Do sada smo imali prilike da vidimo kako se problem može razbiti na više manjih potproblema, koji se mogu dalje poveriti funkcijama na rešavanje. Štaviše, svaki od potproblema se može dalje deliti na još manje potprobleme koji se rešavaju novim funkcijama. Na taj način veliki problemi se dekomponuju na manje, logički povezane celine, kako bi se pojednostavilo njihovo rešavanje. Funkcije su specijalizovane za rešavanje samo određenih problema, čime se poboljšava čitljivost programa, omogućava njegovo lakše menjanje i olakšava otkrivanje eventualnih grešaka.

U prethodnoj sekciji smo mogli videti da glavni program može pozivati funkcije, ali i da ti funkcije mogu dalje pozivati druge funkcije, kako bi im poverile rešavanje dela problema. Međutim, neki problemi su takve prirode da se mogu rešiti rešavanjem jednog ili više problema sličnih početnom problemu. Posmatrajmo, na primer, izračunavanje faktorijela nekog prirodnog broja. Faktorijel prirodnog broja *n* je proizvod svih prirodnih brojeva od 1 do *n*, što se može napisati kao

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

Ako pažljivo pogledamo prethodni izraz, primetićemo da on može da se napiše i kao

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

Na ovaj način smo izračunavanje faktorijela broja *n* sveli na izračunavanje faktorijela broja *n-1*, koji zatim množimo brojem *n*.

Recimo da u C-u želimo da napravimo funkciju za izračunavanje faktorijela koja bi imala jedan celobrojni parametar *n* i koja bi vraćala celobrojnu vrednost:

```
int fakt(int n)
{
   ...
}
```

Pored klasičnog načina izračunavanja faktorijela množenjem brojeva od 1 do *n* u petlji, ovaj problem se može rešiti i na način prikazan u prethodnim izrazima. Dakle, funkciju za računanje faktorijela broja *n* ćemo realizovati pozivanjem iste te funkcije za broj *n-1*, a zatim ćemo dobijeni rezultat pomnožiti sa *n*:

```
int fakt(int n) /* Ova funkcija nije dobra */
{
   return n*fakt(n-1);
}
```

Ovakva situacija kada funkcija, poziva samu sebe se naziva **rekurzija**. Da bi se u potpunosti ovladalo korišćenjem rekurzivnih funkcija, neophodno je razumeti mehanizam rekurzivnih poziva.

Podsetimo se da svaka funkcija ima svoje lokalne promenljive, kao i formalne parametre koji se ponašaju potpuno isto kao i lokalne promenljive. Nakon pozivanja funkcije kreiraju se njene lokalne promenljive i formalni parametri, čime počinje njihov životni vek. Na kraju izvršavanja funkcije sve ove promenljive se uništavaju, što predstavlja kraj njihovog životnog veka. Dakle, kada pozovemo neku funkciju, praktično se stvara jedan njegov primerak (instanca), koji u sebi sadrži sve lokalne promenljive i formalne parametre. Svaki poziv funkcije, stvara njen novi primerak sa svim potrebnim promenljivama. Svaki primerak funkcije je potpuno nezavistan od svih drugih primeraka iste funkcije ili drugih funkcija. Zahvaljujući ovom mehanizmu moguće je da jedan primerak neke funkcije pozove drugi primerak te iste funkcije, što nazivamo rekurzijom. Imajući u vidu da su ova dva primerka potpuno nezavisna i da svaki od njih ima svoje promenljive, neće doći do njihovog mešanja, već će se poziv odvijati potpuno isto kao da jedna funkcija poziva neku drugu funkciju.

Posmatrajmo rekurzivni poziv funkcije *fakt* u prethodnom primeru. Recimo da glavni program poziva funkciju *fakt* sa parametrom *n=5*. Prilikom poziva formira se prvi primerak funkcije *fakt* čiji je parametar 5. Da bi izračunao faktorijel broja 5, ovaj primerak funkcije poziva novi primerak te iste funkcije i prosleđuje joj parametar *n-1*, odnosno 4. Kada bude dobio rezultat drugog primerka funkcije *fakt*, prvi primerak će ga pomnožiti sa *n* i tako dobiti faktorijel broja 5.

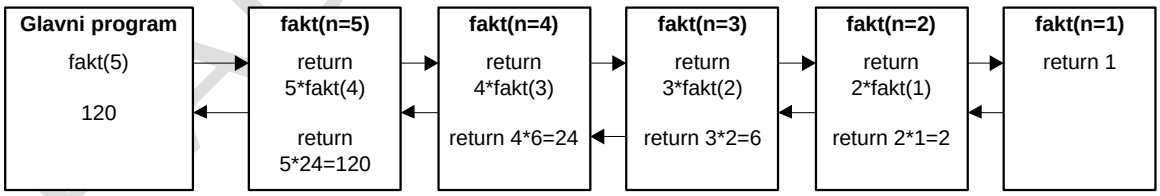
Međutim, da pogledamo kako će drugi primerak funkcije *fakt* izračunati faktorijel broja 4. Na sličan način pozvaće treći primerak te iste funkcije i proslediti mu parametar 3, a zatim rezultat pomnožiti sa 4. Primećujemo da svaki primerak funkcije računa faktorijel tako što pozove novi primerak te iste funkcije i prosledi joj parametar za jedan manji od broja za koji računa faktorijel. Ako bi se tako nastavilo, pozivi novih primeraka funkcije *fakt* bi išli u nedogled i nijedan primerak ne bi obavio svoj posao, već bi svaki od njih prenosio deo problema na novi primerak funkcije. Iz tog razloga ovako napisana funkcija *fakt* nije korektna i nikada neće izračunati faktorijel nekog broja.

Da pozivi novih primeraka ne bi išli u nedogled, neophodno je da postoji **uslov za izlazak iz rekurzije**, odnosno uslov pri kome funkcija više neće pozivati samu sebe, već će se sama pobrinuti za rešenje problema. U slučaju funkcije *fakt* taj uslov bi bio da je *n<2*, zato što je u tom slučaju faktorijel jednak 1 i funkcija nema potrebe da problem dalje razbija i prosleđuje ga novim primercima. Dakle, ispravan oblik funkcije *fakt* bi izgledao ovako:

```
int fakt(int n)
{
   if (n<2)
      return 1;
   else
      return n*fakt(n-1);
}
```

Na ovaj način svaki primerak funkcije će pozivati novi primerak i prosleđivaće mu broj za jedan manji od njenog parametra, sve dok se ne stigne do primerka koji kao parametar dobije broj manji od 2. Kada neki primerak dobije parametar manji od 2, on će prethodnom primerku vratiti rezultat 1. Prethodni primerak će rezultat pomnožiti sa 2 i to rešenje vratiti primerku koji je njega pozvao. Ovi rekurzivni pozivi će se tako razmotavati unazad, sve dok se ne stigne do prvog primerka, koji će konačno vratiti rezultat glavnom programu.

Na Slici ### je šematski prikazano rekurzivno pozivanje funkcije *fakt* i zatim vraćanje rezultata unazad.



Slika ### Rekurzivni pozivi funkcije za računanje faktorijela *fakt*

Rekurzija omogućava veoma elegantno rešavanje nekih problema uz vrlo malo programskog koda. Međutim, prilikom korišćenja rekurzije treba biti oprezan iz razloga što svaki novi primerak funkcije zauzima određeni deo memorije koja je ograničenog kapaciteta, tako da za veliki broj rekurzivnih poziva vrlo lako može doći do prekoračenja i greške u izvršavanju programa. Na sreću, svaki rekurzivni algoritam se može transformisati u iterativni, čime se izbegavaju navedeni problemi. Jednostavne probleme, koji ne zahtevaju korišćenje rekurzije, treba u svakom slučaju rešavati iterativno. Tipičan primer je računanje faktorijela koje se vrlo jednostavno moglo izvesti iterativno, korišćenjem samo jedne *for* petlje. Rekurzivno rešenje će se moći izvršiti samo za manje brojeve, koji ne zahtevaju

preveliki broj primeraka funkcije. U slučaju većih brojeva, vrlo brzo će doći do zagušenja memorije usled velikog broja rekurzivnih poziva.

Primer

Napisati program koji pomoću rekurzije izračunava sumu geometrijskog niza dužine *n* čiji je prvi član *a,* a koeficijent *q*. Da se podsetimo, geometrijski niz je niz brojeva takvih da je količnik svakog broja i njegovog prethodnika konstantan i jednak koeficijentu *q*. Drugim rečima, svaki član niza može se dobiti množenjem prethodnog člana koeficijentom *q*, pri čemu je prvi član niza zadat. Sada geometrijski niz dužine *n* možemo napisati kao

$$a, aq, aq^2, aq^3, aq^4, ..., aq^{n-1}$$

a njegovu sumu u obliku

$$S = a + aq + aq^2 + aq^3 + aq^4 + ,..., + aq^{n-1} = a \cdot \left(1 + q + q^2 + q^3 + q^4 + ,..., + q^{n-1}\right)$$

Ako pažljivo pogledamo izraz u zagradi, uočićemo da se radi o geometrijskom nizu dužine *n* čiji je prvi član 1, a koeficijent *q*, pa zaključujemo da se računanje sume bilo kog geometrijskog niza može svesti na računanje sume geometrijskog niza čiji je prvi član 1 i koju ćemo obeležiti sa S_n.

Međutim, izraz u zagradi možemo napisati i na sledeći način

$$S_n = 1 + q + q^2 + q^3 + q^4 + ,..., + q^{n-1} = 1 + q \cdot \left(1 + q + q^2 + q^3 + ,..., + q^{n-2}\right) = 1 + q \cdot S_{n-1}$$

što jasno pokazuje da se suma S_n može izraziti preko sume geometrijskog niza koji ima jedan element manje, kao

$$S_n = 1 + q \cdot S_{n-1}$$

i što nas upućuje na korišćenje rekurzivne funkcije. Pored rekurzivnog izraza koji je dat, neophodno je definisati i uslov za izlazak iz rekurzije, odnosno uslov pri kome nije moguće dalje razbijati problem da sličan potproblem. To je svakako slučaj kada se traži izračunavanje sume geometrijskog niza dužine 1, za koju sigurno znamo da je jednaka jedinici ($S_1 = 1$).

```
#include <stdio.h>

float sumageo(int n, float q)
{
   if(n == 1)
     return 1;
   else
     return 1.0+q*sumageo(n-1, q);
}

main()
{
   int n;
   float a, q, s;

   printf("Unesite duzinu niza:\n");
   scanf("%d", &n);

   printf("Unesite prvi clan i koeficijent:\n");
   scanf("%f%f", &a, &q);
```

```
   s = a*sumageo(n, q);

   printf("Suma geometrijskog niza je: %.2f\n", s);
}
```