Uvod u programiranje

- predavanja -

studeni 2019.

7. Funkcije

Funkcije

Uvod

Primjer

- Učitati cijele brojeve m i n, pri čemu vrijedi 0 ≤ n ≤ m. Nije potrebno provjeravati učitane vrijednosti. Izračunati binomni koeficijent *C*(m, n) ili "m povrh n", pri čemu koristiti sljedeći izraz:

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

  - Primjeri izvršavanja programa

```
Upisite m i n > 10 3↵
C(10, 3) = 120
```

```
Upisite m i n > 12 8↵
C(12, 8) = 495
```

Rješenje bez funkcije (1. dio)

prog.c

```
#include <stdio.h>

int main(void) {
   int m, n;
   int i, mFakt, nFakt, mnFakt, binKoef;

   printf ("Upisite m i n > ");
   scanf ("%d %d", &m, &n);

   ...
```

Rješenje bez funkcije (2. dio)

prog.c (nastavak)

```
...
   mFakt = 1;                                          m!
   for (i = 2; i <= m; ++i)
      mFakt = mFakt * i;

   nFakt = 1;                                          n!
   for (i = 2; i <= n; ++i)
      nFakt = nFakt * i;

   mnFakt = 1;                                         (m - n)!
   for (i = 2; i <= m - n; ++i)
      mnFakt = mnFakt * i;

   binKoef = mFakt / (nFakt * mnFakt);

   printf("C(%d, %d) = %d", m, n, binKoef);

   return 0;
}
```

Komentar prethodnog rješenja

- Sličan programski odsječak ponavlja se tri puta

- Nedostaci:
  - broj linija programskog koda raste
  - povećava se mogućnost pogreške (kod pisanja i kod izmjene)

- Preporuka:
  - program razdvojiti u logičke cjeline koje obavljaju određene, jasno definirane poslove

Rješenje s korištenjem funkcije (1. dio)

prog.c

```
#include <stdio.h>

// Funkcija za racunanje n faktorijela
int fakt(int n) {
   int i;
   int umnozak = 1;
   for (i = 2; i <= n; ++i)
      umnozak = umnozak * i;
   return umnozak;
}

...
```

zbog printf u funkciji main

Rješenje s korištenjem funkcije (2. dio)

prog.c (nastavak)

```
...
// kolokvijalno: glavni program
int main(void) {
   int m, n;
   int binKoef;

   printf ("Upisite m i n > ");
   scanf ("%d %d", &m, &n);

   binKoef = fakt(m) / (fakt(n) * fakt(m - n));

   printf("C(%d, %d) = %d", m, n, binKoef);


   return 0;
}
```

Alternativno: s funkcijom za binomni koeficijent

prog.c

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

zbog printf u funkciji main

```
// funkcija za racunanje n faktorijela

int fakt(int n) {
   int i;
   int umnozak = 1;
   for (i = 2; i <= n; ++i)
      umnozak = umnozak * i;
   return umnozak;
}

// funkcija za racunanje binomnog keoficijenta

int binKoef(int m, int n) {
   return fakt(m) / (fakt(n) * fakt(m - n));
}

...

   // poziv funkcije u glavnom programu (funkciji main)
   printf("C(%d, %d) = %d", binKoef(m, n));
```

Varijanta s *unsigned long long* umjesto *int*

prog.c

- domena funkcije fakt u prethodnoj varijanti je skup cijelih brojeva iz intervala [0, 12] jer je 13! = 6 227 020 800

```
unsigned long long fakt(unsigned int n) {
   unsigned int i;
   unsigned long long umnozak = 1ULL;
   for (i = 2U; i <= n; ++i)
      umnozak = umnozak * i;
   return umnozak;
}

unsigned long long binKoef(unsigned int m, unsigned int n) {
   return fakt(m) / (fakt(n) * fakt(m - n));
}
// poziv funkcije u glavnom programu (funkciji main)
...
   printf("C(%u, %u) = %llu", m, n, binKoef(m, n));
```

- ovdje je domena funkcije fakt skup cijelih brojeva iz intervala [0, 20].

```
Upisite m i n > 20 5↵
C(20, 5) = 15504
```

Korištenje boljeg algoritma

- Prethodni algoritam za izračunavanje binomnog koeficijenta je prikladan za objašnjavanje koncepta funkcije, ali primjenom multiplikativne formule dobilo bi se znatno bolje rješenje:
  - izbjegava se veliki broj koraka pri zasebnom izračunavanju m!, n! i (m-n)!
  - izbjegava se preliv koji može nastati zbog strmog rasta funkcije faktorijela

```
int binKoef(int m, int n) {
   int rez = 1;
   int i;

   if (n < m - n)
      for (i = 1; i <= n; ++i)
         rez = rez * (m - n + i) / i;
   else
      for (i = 1; i <= m - n; ++i)
         rez = rez * (n + i) / i;

   return rez;
}
```

$$\binom{m}{n} = \prod_{i=1}^{n} \frac{m-(n-i)}{i} = \prod_{i=1}^{m-n} \frac{n+i}{i}$$

```
Upisite m i n > 20 5↵
C(20, 5) = 15504
```

Varijanta s *unsigned long long* umjesto *int*

```
unsigned long long
binKoef(unsigned int m,
        unsigned int n) {
   unsigned long long rez = 1ULL;
   unsigned int i;

   if (n < m - n)
      for (i = 1U; i <= n; ++i)
         rez = rez * (m - n + i) / i;
   else
      for (i = 1U; i <= m - n; ++i)
         rez = rez * (n + i) / i;

   return rez;
}
// poziv funkcije u glavnom programu (funkciji main)
...
   printf("C(%u, %u) = %llu", m, n, binKoef(m, n));
```

Definicija funkcije

- definicijom funkcije opisuje se
  - tip rezultata funkcije
  - naziv funkcije
  - (opcionalno) lista parametara funkcije, za svaki parametar: tip i naziv
  - tijelo funkcije: definicije i deklaracije, naredbe koje se obavljaju kad se funkcija pozove
  - (opcionalno) naredba za povratak rezultata i programskog slijeda

Opći oblik

```
tip_rezultata naziv_funkcije(lista_parametara) {
   definicije, deklaracije i naredbe;
   return rezultat;
}
```

Definicija funkcije: tip funkcije

- tip rezultata (tip funkcije) mora biti naveden i može biti
  - bilo koji tip podatka **osim polja (*array*)**
    - osnovni tipovi, pokazivači, strukture
  - npr. ako funkcija vraća rezultat tipa int, kažemo: funkcija je tipa int
  - ako funkcija ne vraća rezultat (kažemo: funkcija je tipa void), kao *`tip_rezultata`* mora se navesti ključna riječ *`void`*

Definicija funkcije: parametri

- tip i naziv parametra trebaju biti navedeni za svaki parametar (ako funkcija ima parametre)
  - parametar može biti bilo kojeg tipa podatka **osim polja (*array*)**
    - osnovni tipovi, pokazivači, strukture
  - broj parametara nije ograničen (a može ih i ne biti)
    - ako funkcija nema niti jedan parametar, tada se na mjestu `lista_parametara` mora navesti ključna riječ `void`
  - parametar je *modifiable lvalue*: u tijelu funkcije može se koristiti na isti način kao svaka druga varijabla definirana u tijelu funkcije
    - od varijable definirane u tijelu funkcije parametar se razlikuju samo po tome što mu se na početku izvršavanja funkcije pridružuje vrijednost *argumenta* s kojim je funkcija pozvana

Definicija funkcije: povratak rezultata

- povratak rezultata i programskog slijeda na pozivajuću razinu (na mjesto u izrazu u kojem se nalazi poziv funkcije) obavlja se naredbom *return*
  - Opći oblik naredbe `return` *izraz*$_{\text{opcionalno}}$`;`
  - funkcija može vratiti najviše jednu vrijednost (ili niti jednu)
  - rezultat (*izraz*) može biti bilo kojeg tipa podatka **osim polja (array)**
    - ako se tip izraza razlikuje od tipa funkcije, obavlja se implicitna konverzija vrijednosti rezultata u tip funkcije
- u funkciji koja ne vraća rezultat ova naredba se može
  - izostaviti, u kojem slučaju će se povratak programskog slijeda dogoditi na kraju tijela funkcije
  - ili koristiti bez navođenja izraza koji predstavlja rezultat
- naredba se može navesti više puta unutar iste funkcije
  - iako to nije sasvim u skladu sa strukturiranim programiranjem

Primjer: definicija funkcije

- funkcija x^n, $x \in \mathbb{R}$, $n \in \mathbb{N}^0$

Rezultat funkcije je tipa double.

Parametri funkcije

prog.c

```
double eksp(float x, int n) {
   int i;
   double rez = 1.;
   for (i = 0; i < n; ++i)
      rez *= x;
   return rez;
}
...
int main(void) {
   ...
```

Varijable i, rez su lokalne varijable, vidljive samo u tijelu funkcije

Naredba za povratak (programski slijed i rezultat)

definicija funkcije main

- u programskom jeziku C funkcija se nikad ne definira *unutar* definicije neke druge funkcije

Poziv funkcije: argumenti

- argumenti funkcije su izrazi koji se navode pri pozivu funkcije, u okruglim zagradama iza naziva funkcije, odvojeni zarezima
  - argument može biti bilo koji tip podatka **osim polja (*array*)**
    - osnovni tipovi, pokazivači, strukture
  - broj i redoslijed argumenata treba odgovarati broju i redoslijedu parametara u funkciji koja se poziva
  - tipovi podataka argumenata moraju odgovarati tipovima podataka odgovarajućih parametara ili mora biti moguća konverzija podatka iz tipa podatka argumenta u tip podatka parametra
    - npr. ako je parametar funkcije tipa double, funkcija se može pozvati s argumentom tipa int (vodeći računa o mogućim gubicima informacija koje mogu nastati pri konverziji tipova podataka)
  - ako funkcija nema parametre, tada se pri pozivu funkcije na mjestu liste argumenata ne piše ništa (ne piše se void)

Primjer: poziv funkcije

prog.c

```
#include <stdio.h>

double eksp(float x, int n) {
   ...
   return rez;
}
...
int main(void) {
   ...
   double y = eksp(3.f, 4);
   printf("Tri na cetvrtu je %lf", y);
   printf("Dva na petu je %lf", eksp(2.f, 5));
   eksp(3.f, 2);
   y = eksp(3.5f, 2) + 2 * eksp(2.1f, 3);
   y = eksp(eksp(2 * 3.f, 2), 4);
   ...
}
```

zbog printf u funkciji main

- argumenti mogu biti izrazi
- rezultat funkcije može se koristiti u raznim izrazima

ispravno, ali beskorisno

$3.5^2 + 2 \cdot 2.1^3$

$(6^2)^4$

- za sada, definiciju funkcije *main* uvijek napisati na kraju. Kasnije će biti objašnjeno na koji se način redoslijed pisanja definicija funkcija može promijeniti.

Primjer

- funkcija koja nema parametre, ali vraća rezultat

```
...
int prebroji(void) {
   char c;
   int brojac = 0;
   do {
      scanf("%c", &c);
      ++brojac;
   } while (c != '#');
   return brojac - 1;
}
...
int main(void) {
   printf("%d\n", prebroji());
   printf("%d\n", prebroji());
   ...
```

- broji koliko je znakova upisano preko tipkovnice prije pojave prvog znaka #

```
koliko znakova?#a sada?#↵
15↵
7↵
```

```
##↵
0↵
0↵
```

Primjer

- funkcija koja ima parametre, ali ne vraća rezultat

```
...
void ispisXY(float x, float y) {
   printf("(%.4f, %.4f)", x, y);
   return;      može se izostaviti
}
...
int main(void) {
   float x1 = 3.25f, y1 = -12.f;
   float x2 = 0.1f, y2 = 4.5f;

   printf("Koordinate T1: ");
   ispisXY(x1, y1);
   printf("\nKoordinate T2: ");
   ispisXY(x2, y2);
   ...
```

- ispisuje x, y koordinate unutar okruglih zagrada, s 4 znamenke iza decimalne točke

```
Koordinate T1: (3.2500, -12.0000)
Koordinate T2: (0.1000, 4.5000)
```

Primjer

- funkcija koja nema parametre i ne vraća rezultat

```
...
void preskoci(void) {
   char c;
   do {
      scanf("%c", &c);
   } while (c != '#');
   return;
}
...
int main(void) {
   int prvi, drugi;
   preskoci();
   scanf("%d", &prvi);
   preskoci();
   scanf("%d", &drugi);
   printf("prvi = %d, drugi = %d", prvi, drugi);
   ...
```

može se izostaviti

- preskače znakove upisane preko tipkovnice, do znaka #

- čita po jedan cijeli broj iza prva dva znaka #

```
preskoci sve ovo#567 preskoci i ovo#98765↵
prvi = 567, drugi = 98765
```

Primjer

- funkcija u kojoj se na više mjesta koristi naredba return

```
...
double apsolut(double x) {
   if (x < 0) {
      return -x;
   } else {
      return x;
   }
}
...
int main(void) {
   double x = -3.5;
   printf("abs(%lf) = %lf", x, apsolut(x));
   ...
```

- izračunava apsolutnu vrijednost realnog broja

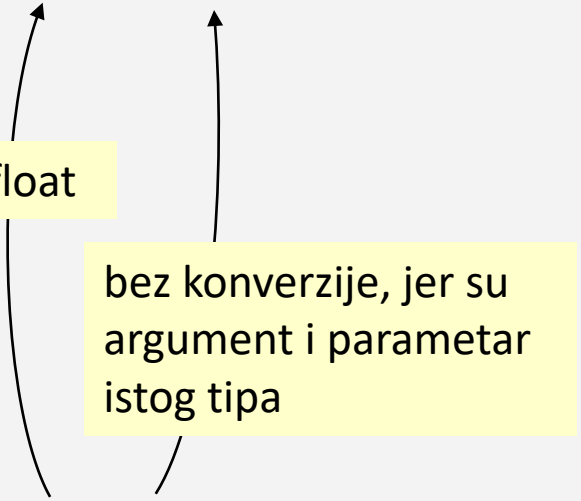
```
abs(-3.500000) = 3.500000
```

Primjer

- implicitna konverzija argumenata

```
...
double eksp(float x, int n) {
   ...
   return rez;
}


...
int main(void) {
   ...
   double y = eksp(3, 4);
   printf("%lf", y);
   ...
```



- slično, npr. funkcija `sqrt` iz `<math.h>`, čiji je parametar tipa `double`, vratit će ispravan rezultat i onda kada se kao argument koristi cijeli broj

Primjer

- implicitna konverzija rezultata

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

...
int main(void) {
   printf("%c", malo_u_veliko('f'));
   printf("%c", malo_u_veliko('B'));
   printf("%c", malo_u_veliko('*'));
```

int → char, jer `c - ('a' - 'A')` je tipa int

bez konverzije, jer `c` već jest tipa char

```
F
B
*
```

Primjer

- funkcija **ne može** izmjenom vrijednosti parametara promijeniti vrijednosti argumenata
  - jer parametar sadrži *kopiju* vrijednosti argumenta

```
#include <stdio.h>
void pokusajPromijenitiArgument(int n) {
   n = 10;
   printf("Funkcija je parametar promijenila u n = %d\n", n);
   return;
}

int main(void) {
   int n = 5;
   printf("Funkcija je pozvana s argumentom n = %d\n", n);
   pokusajPromijenitiArgument(n);
   printf("Ali argument je ostao n = %d", n);
   return 0;
}
```

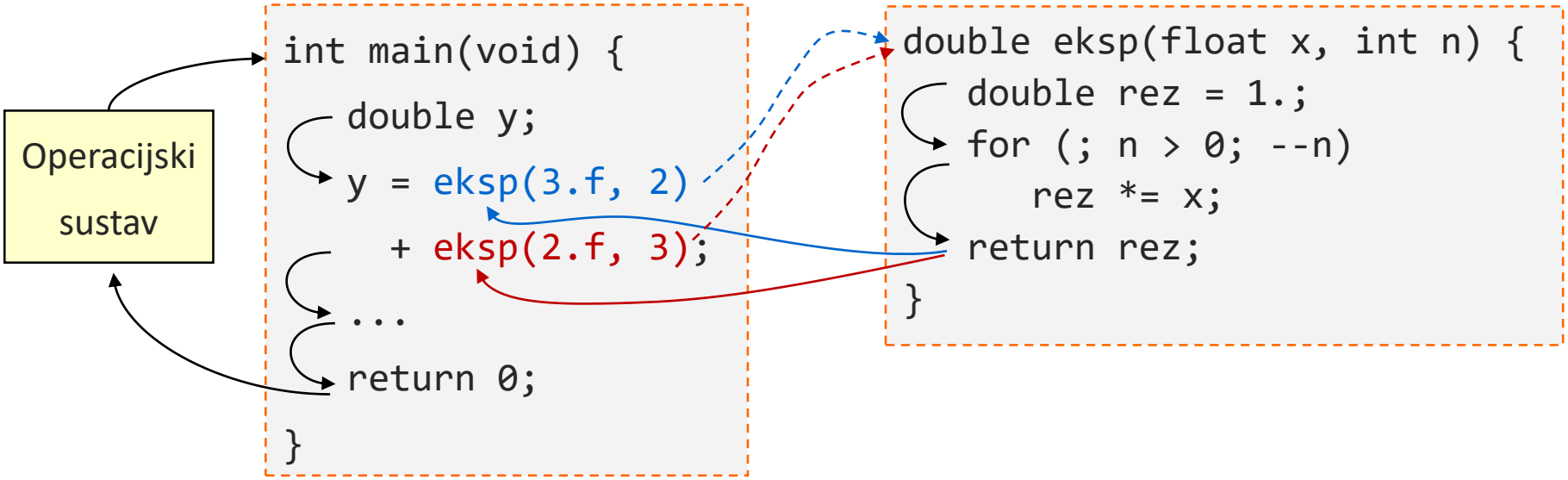
```
Funkcija je pozvana s argumentom n = 5↵
Funkcija je parametar promijenila u n = 10↵
Ali argument je ostao n = 5
```

Funkcije

Mehanizmi
prijenosa argumenata i
povrata rezultata

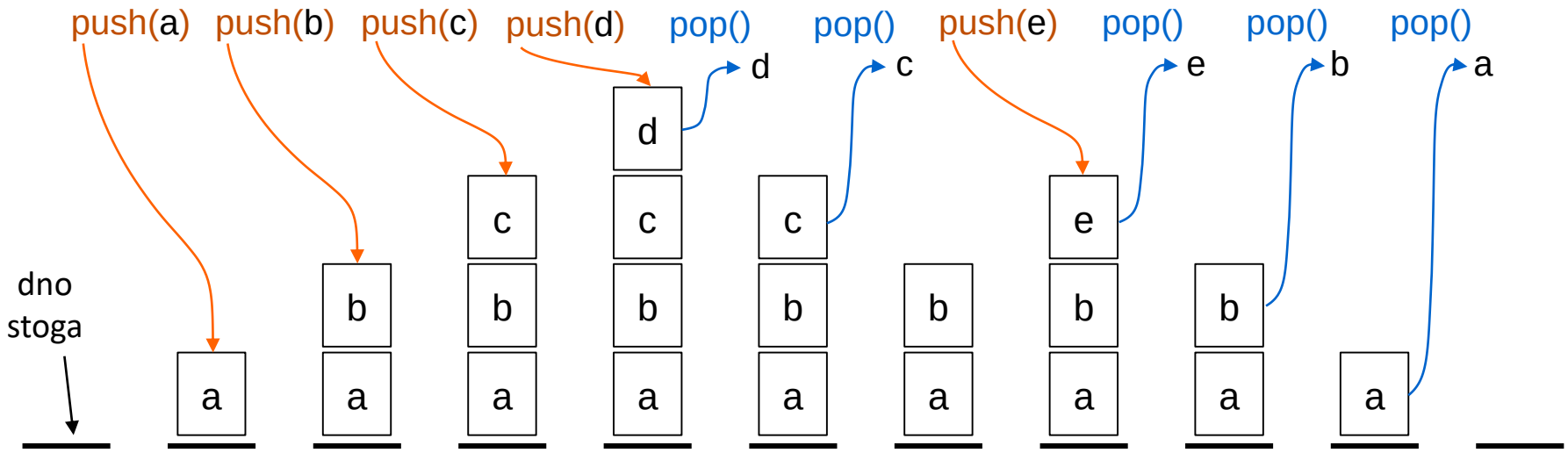
Mehanizam poziva funkcije

- u trenutku poziva funkcije potrebno je rezervirati memoriju za pohranu parametara i adrese instrukcije na koju se treba vratiti (povratne adrese)
- za vrijeme izvršavanja funkcije potrebno je rezervirati memoriju za varijable koje se definiraju u funkciji
- nakon završetka funkcije rezerviranu memoriju treba osloboditi



Stog (*stack*) općenito

- svaka razina poziva funkcije zahtijeva rezervaciju memorije za *svoje* parametre, varijable i povratnu adresu
  - implementira se pomoću stoga. Osnovne operacije nad stogom su:
    - dodavanje podatka na stog (*push*)
    - uzimanje podatka s vrha stoga (*pop*)
  - posljednji na stog postavljeni podatak prvi je na redu za uzimanje
    - *posljednji unutra, prvi van* (*last in, first out – LIFO*)



U programskom jeziku C (pojednostavljeno)

```
int main(void) {
1   int r, i = 2;
    r = puta20(i);
    ...
8   return 0;
}
```

```
int puta20(int p) {
3   int rez;
    rez = putaN(p, 20);

7   return rez;
}
```

```
int putaN(int p,
          int n) {
5   int rez;
    rez = p * n;
6   return rez;
}
```

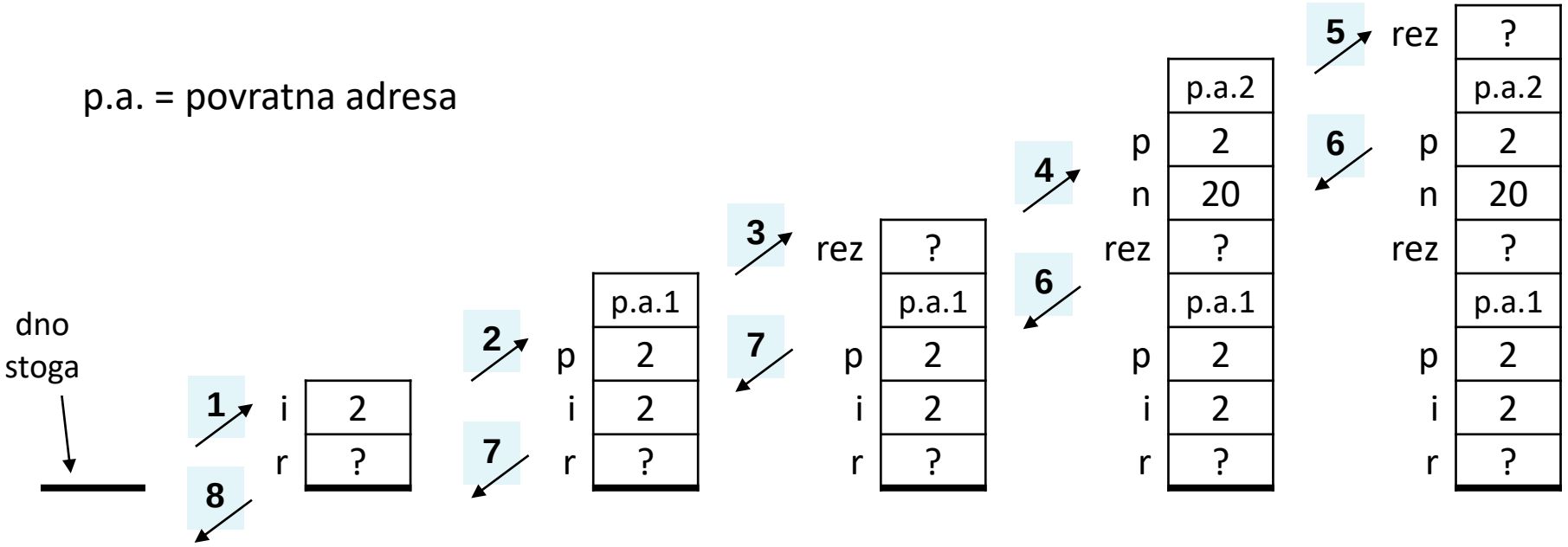
p.a.1

p.a.2

2

4

p.a. = povratna adresa



Objašnjenje

1. na stog: prostor za lokalne varijable `r`, `i`
2. na stog: vrijednost za parametar `p` i povratnu adresu p.a.1
3. na stog: prostor za lokalnu varijablu `rez`
4. na stog: vrijednosti za parametre `p`, `n` i povratnu adresu p.a.2
5. na stog: prostor za lokalnu varijablu `rez`
6. uklanjanje lokalnih varijabli, parametara i povratne adrese sa stoga nakon povratka na p.a.2
7. uklanjanje lokalnih varijabli, parametara i povratne adrese nakon povratka na p.a.1
8. uklanjanje lokalnih varijabli sa stoga pri završetku programa

Funkcije

Rekurzivne funkcije

Rekurzivna funkcija

- Funkcija koja poziva samu sebe naziva se *rekurzivnom funkcijom*
  - izravna rekurzija: npr. funkcija *f* sadrži poziv funkcije *f*
  - neizravna rekurzija: npr. funkcija *f* sadrži poziv funkcije *g* koja sadrži poziv funkcije *f*
- Primjer definicije i poziva rekurzivne funkcije

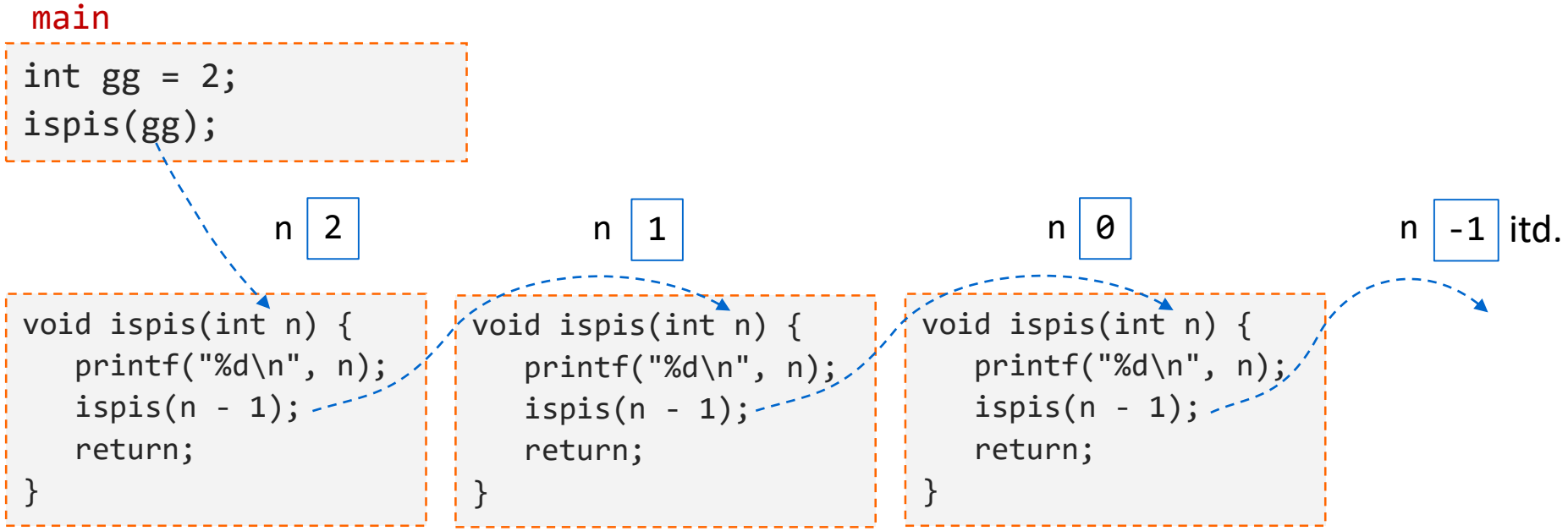
```
void ispis(int n) {
    printf("%d\n", n);
    ispis(n - 1);
    return;
}
... u funkciji main
    int gg = 2;
    ispis(gg);
```

```
2↵
1↵
0↵
...  ?  Kada će se ispisivanje cijelih  brojeva prekinuti?
```

- Što će biti rezultat poziva funkcije? Koju veliku pogrešku sadrži ova definicija funkcije?

Redoslijed pozivanja

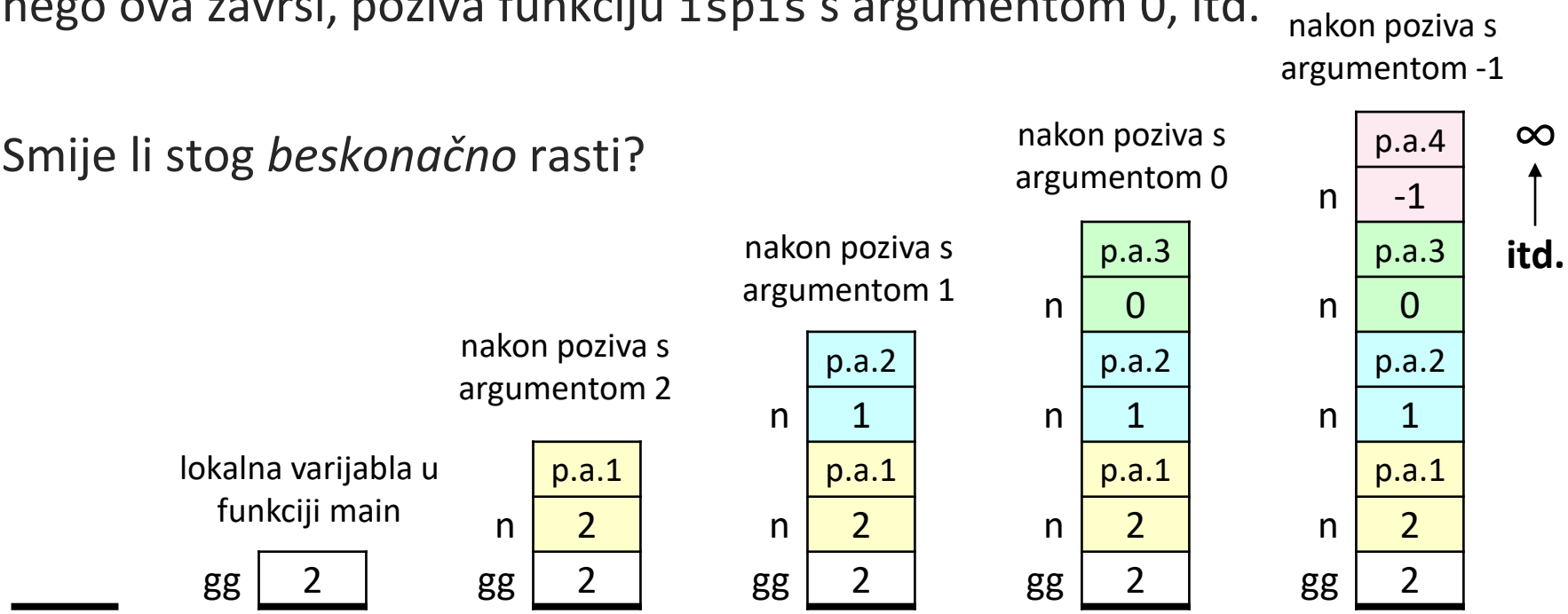
- Zamislimo, radi vizualizacije, da postoji više instanci funkcije ispis
- Prije nego funkcija `ispis` pozvana s argumentom gg=2 završi, poziva funkciju `ispis` s argumentom 1. Prije nego ova završi, poziva funkciju `ispis` s argumentom 0, itd.



Stog u loše definiranoj rekurzivnoj funkciji

- Prije nego funkcija ispis pozvana s argumentom 2 završi (dakle, dok sa stoga još nisu uklonjeni parametar n=2 i povratna adresa p.a.1 za povratak u funkciju main), poziva funkciju `ispis` s argumentom 1. Prije nego ova završi, poziva funkciju `ispis` s argumentom 0, itd.

- Smije li stog *beskonačno* rasti?



  - kada se memorija inicijalno dodijeljena programu za stog potroši, program će se prekinuti zbog pogreške tijekom izvršavanja.

Posljedica loše definirane rekurzivne funkcije

```
void ispis(int n) {
   printf("%d\n", n);
   ispis(n - 1);
   return;
}
```

```
2↵
1↵
0↵
-1↵
...
-392858↵
-392859↵
Segmentation fault (core dumped)
```

izvršavanje u operacijskom sustavu Linux

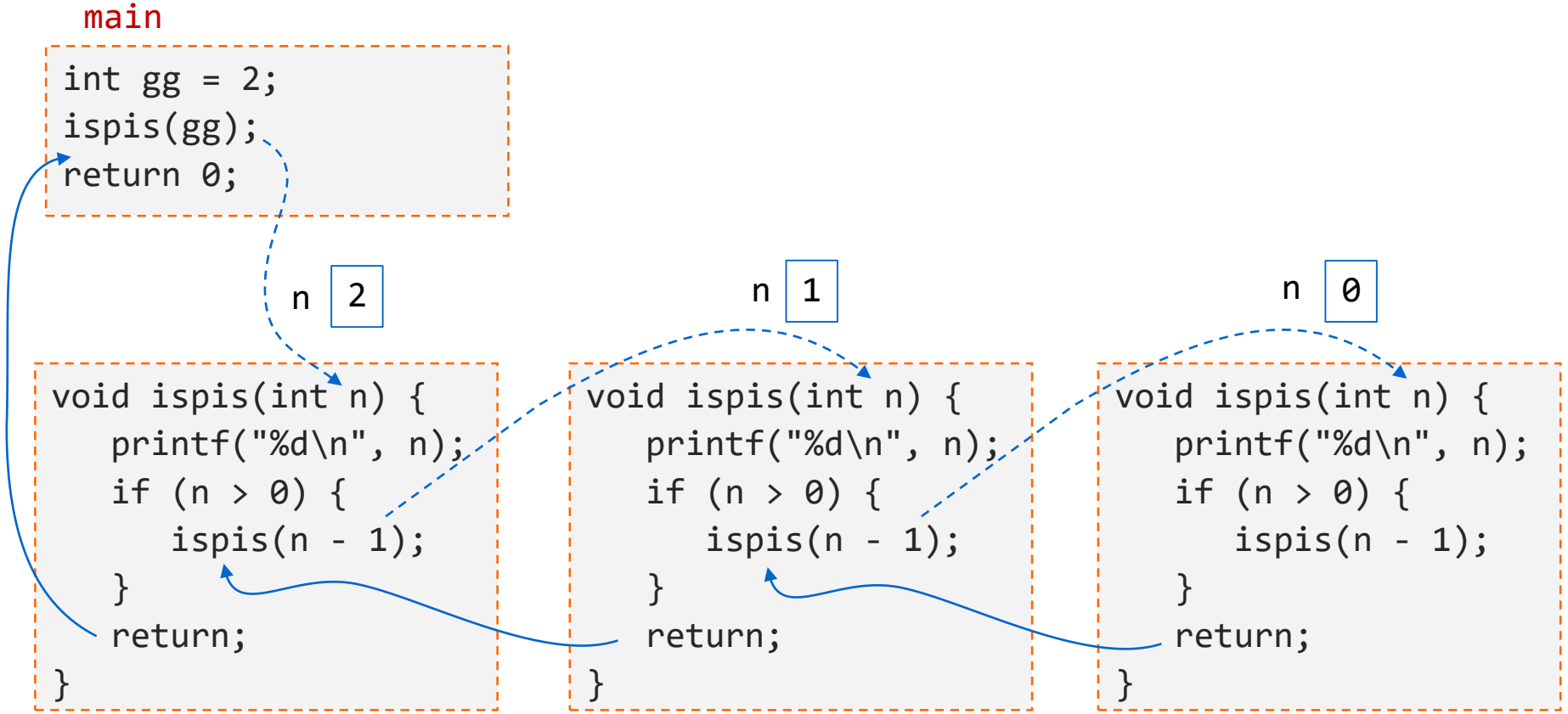
- Rekurzivna funkcija se mora definirati tako da pod nekim uvjetima prestane s daljnjim pozivanjem same sebe

```
void ispis(int n) {
   printf("%d\n", n);
   if (n > 0) {
      ispis(n - 1);
   }
   return;
}
```

```
2↵
1↵
0↵
```

Redoslijed pozivanja i povratka

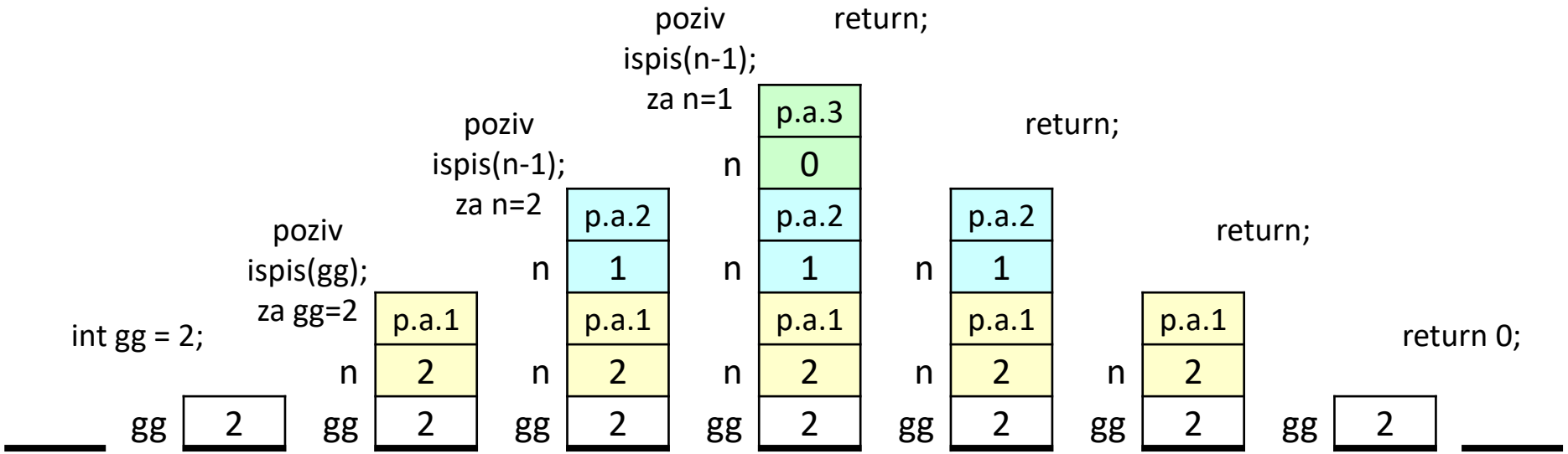
- zamislimo, radi vizualizacije, da postoji više instanci funkcije ispis



Stog u ispravno definiranoj rekurzivnoj funkciji

```
int main(void) {
   int gg = 2;
   ispis(gg);
   return 0;
}
```

```
void ispis(int n) {
   printf("%d\n", n);
   if (n > 0) {
      ispis(n - 1);
   }
   return;
}
```



Primjer: matematička definicija funkcije

- Rekurzivna definicija funkcije *fact(n)*:
- $$fact(n) = \begin{cases} 1 & za\ n = 0 \\ n \cdot fact(n-1) & za\ n > 0 \end{cases}$$

```
fact(4) =
4 · (fact(3)) =
4 · (3 · (fact(2))) =
4 · (3 · (2 · (fact(1)))) =
4 · (3 · (2 · (1· (fact(0))))) =
4 · (3 · (2 · (1· (1))))
```

Primjer: definicija funkcije u C-u

- Matematički rekurzivni izrazi često se bez teškoća pretvaraju u definiciju rekurzivne funkcije u programskom jeziku

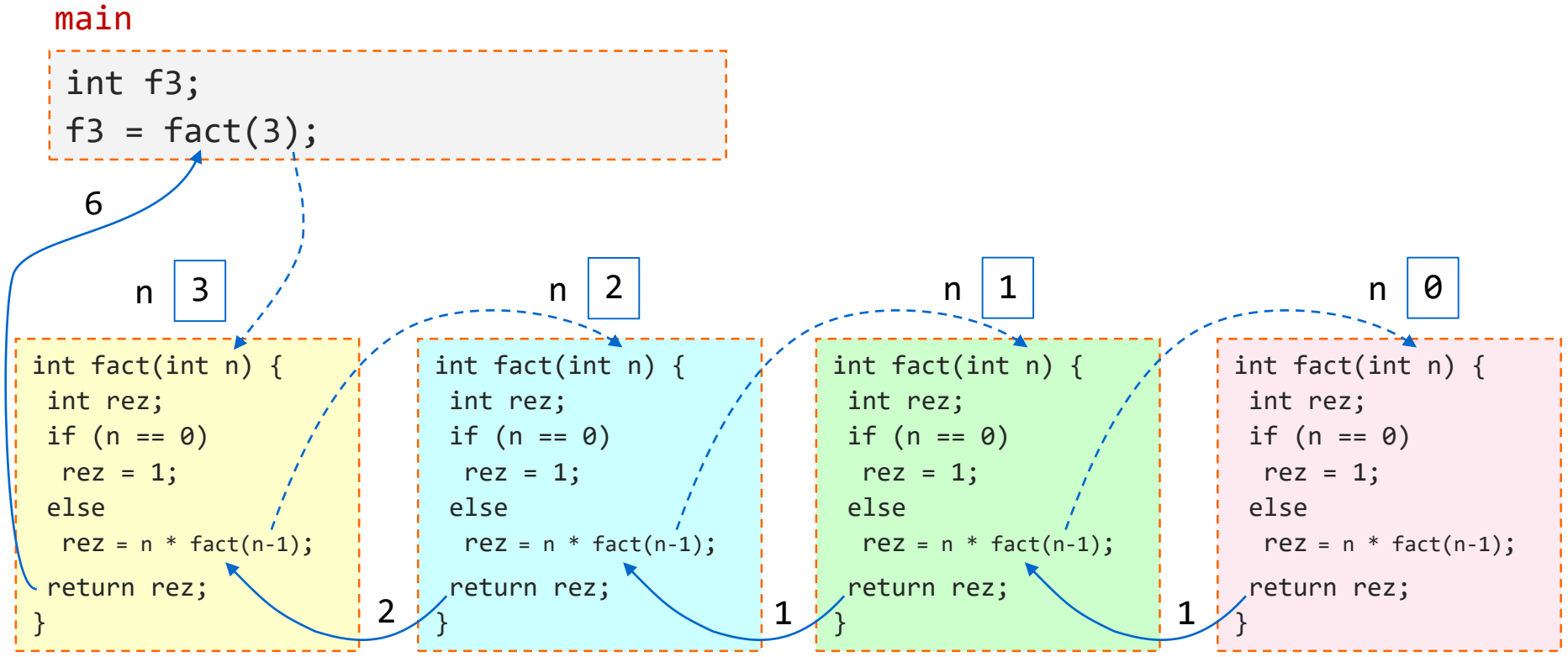
```
int fact(int n) {
   int rez;
   if (n == 0)
      rez = 1;
   else
      rez = n * fact(n - 1);

   return rez;
}
```

```
int fact(int n) {
   if (n == 0)
      return 1;
   else
      return n * fact(n - 1);
}
```

Primjer: redoslijed pozivanja i povratka

- Zamislimo, radi vizualizacije, da postoji više instanci funkcije *fact*



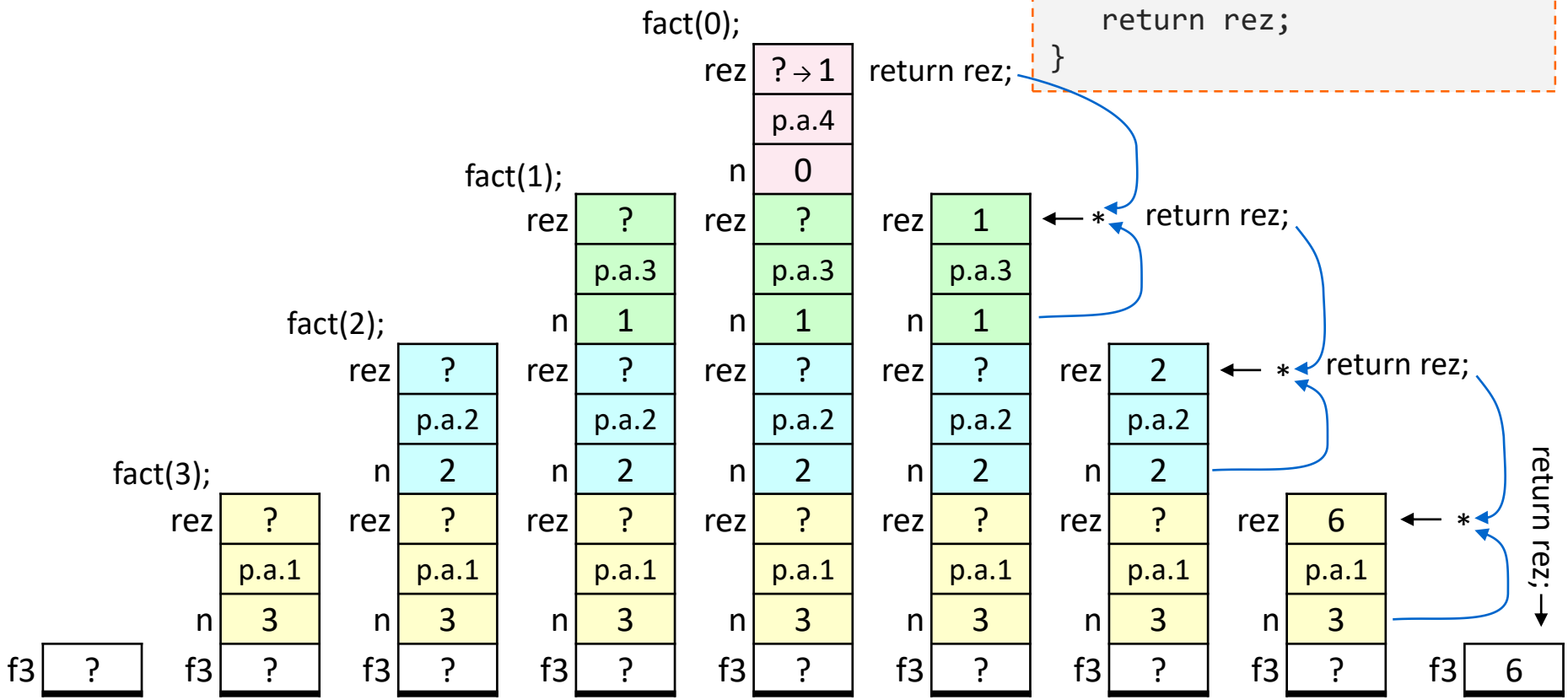
Primjer: stog za poziv `fact(3)`

```
int fact(int n) {
   int rez;
   if (n == 0)
      rez = 1;
   else
      rez = n * fact(n - 1);

   return rez;
}
```

main

```
int f3;
f3 = fact(3);
```



Varijanta s *unsigned long long* umjesto *int*

- Korištenjem ovog tipa podatka, domena funkcije *fact* povećava se na cijele brojeve iz intervala [0, 20]

```
unsigned long long
fact(unsigned int n) {
   unsigned long long rez;
   if (n == 0)
      rez = 1ULL;
   else
      rez = n * fact(n - 1);

   return rez;
}
```

```
unsigned long long
fact(unsigned int n) {
   if (n == 0)
      return 1ULL;
   else
      return n * fact(n - 1);
}
```