Uvod u programiranje

- predavanja -

studeni 2025.

13. Funkcije

Nije bilo vodiča…

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

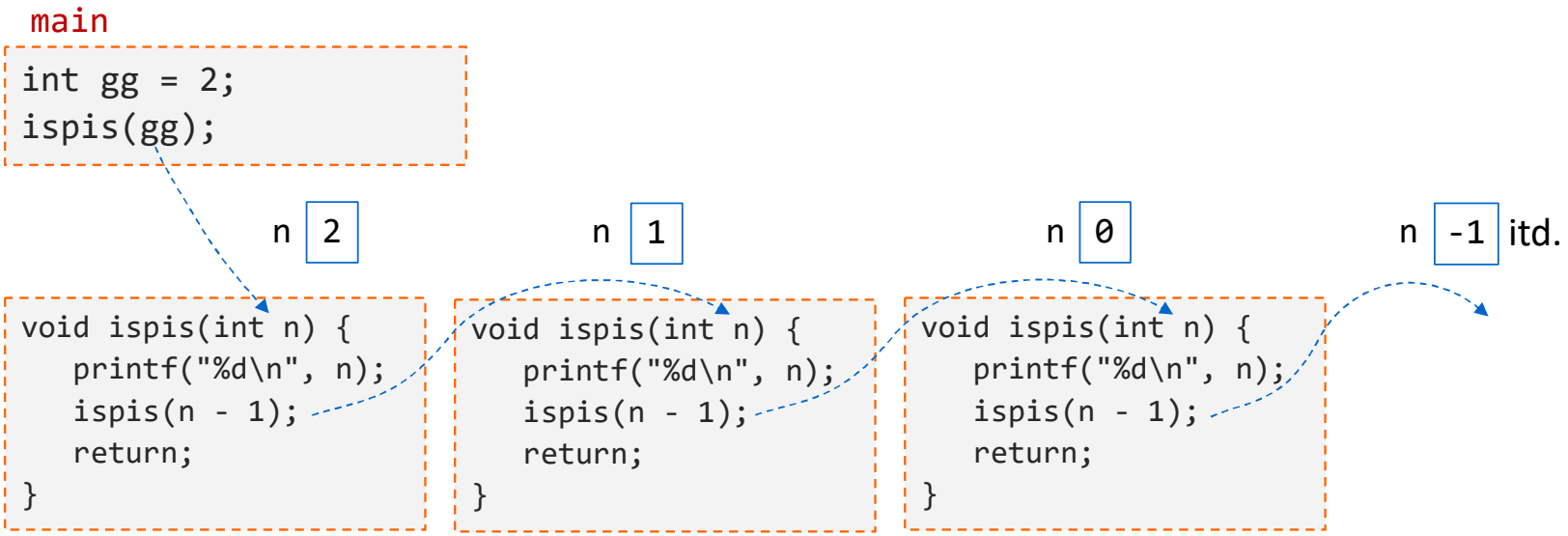
```
void ispis(int n) {
    printf("%d\n", n);
    ispis(n - 1);
    return;
}
... u funkciji main
    int gg = 2;
    ispis(gg);
```

```
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

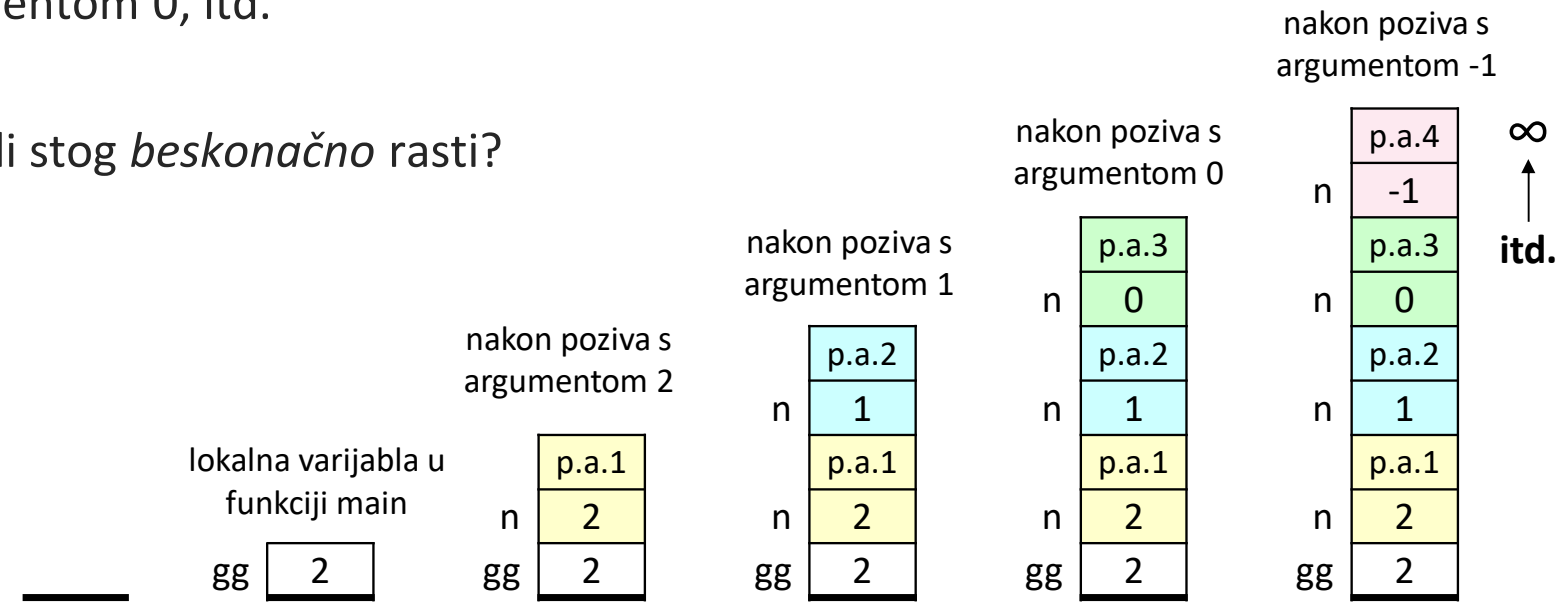
- Zamislimo, radi vizualizacije, da postoji više instanci funkcije ispis
- Prije nego funkcija `ispis` pozvana s argumentom gg=2 završi, poziva funkciju `ispis` s argumentom 1. Prije nego ova završi, poziva funkciju `ispis` s argumentom 0, itd.



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

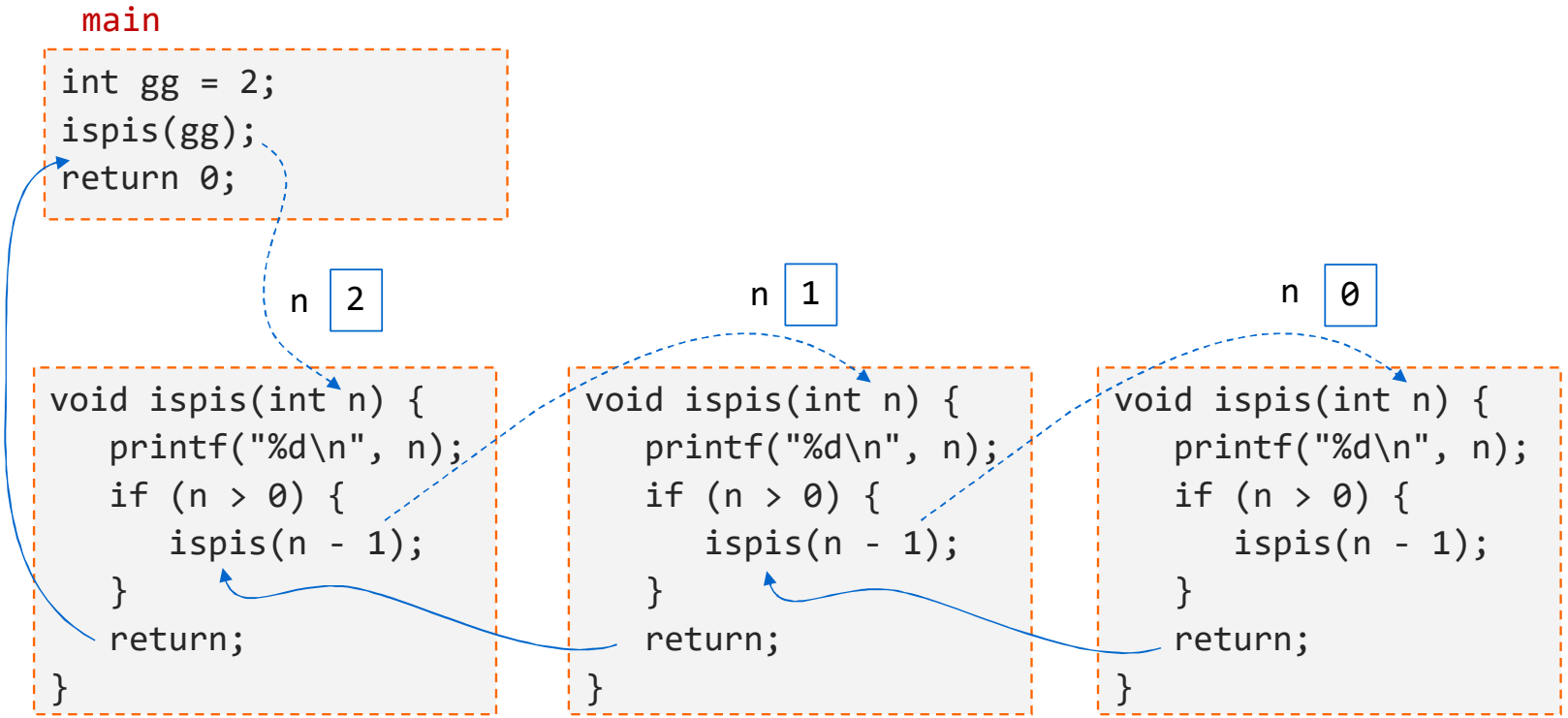
- Rekurzivna funkcija se mora definirati tako da pod nekim uvjetima prestane s daljnjim pozivanjem same sebe

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

```
2↵
1↵
0↵
```

Redoslijed pozivanja i povratka

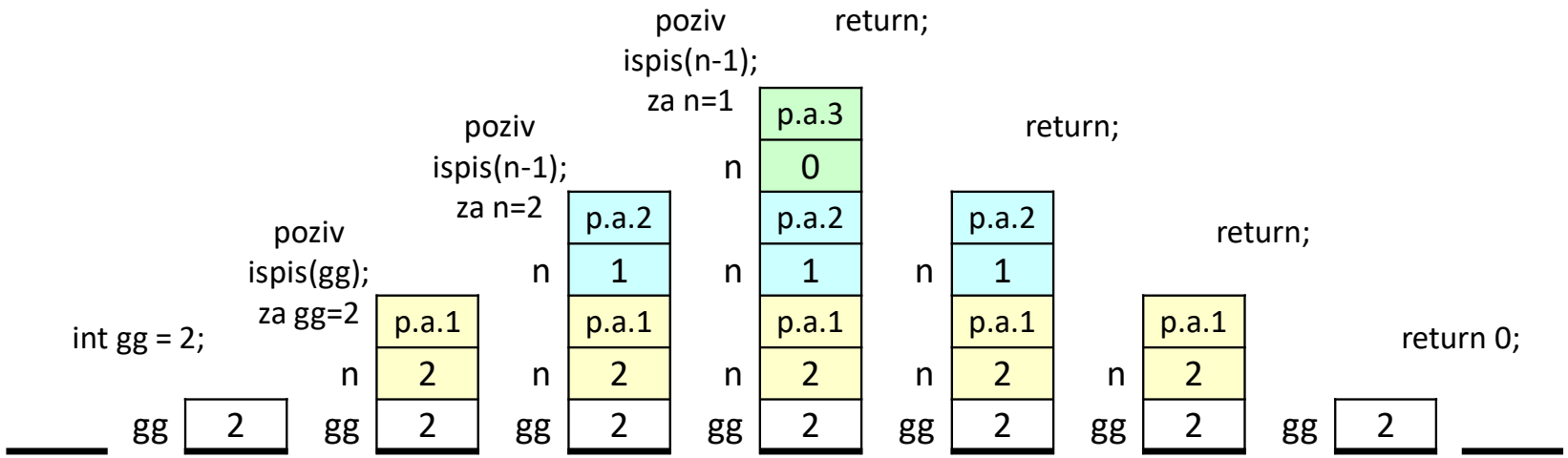
- zamislimo, radi vizualizacije, da postoji više instanci funkcije ispis



Stog u ispravno definiranoj rekurzivnoj funkciji

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

```c
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

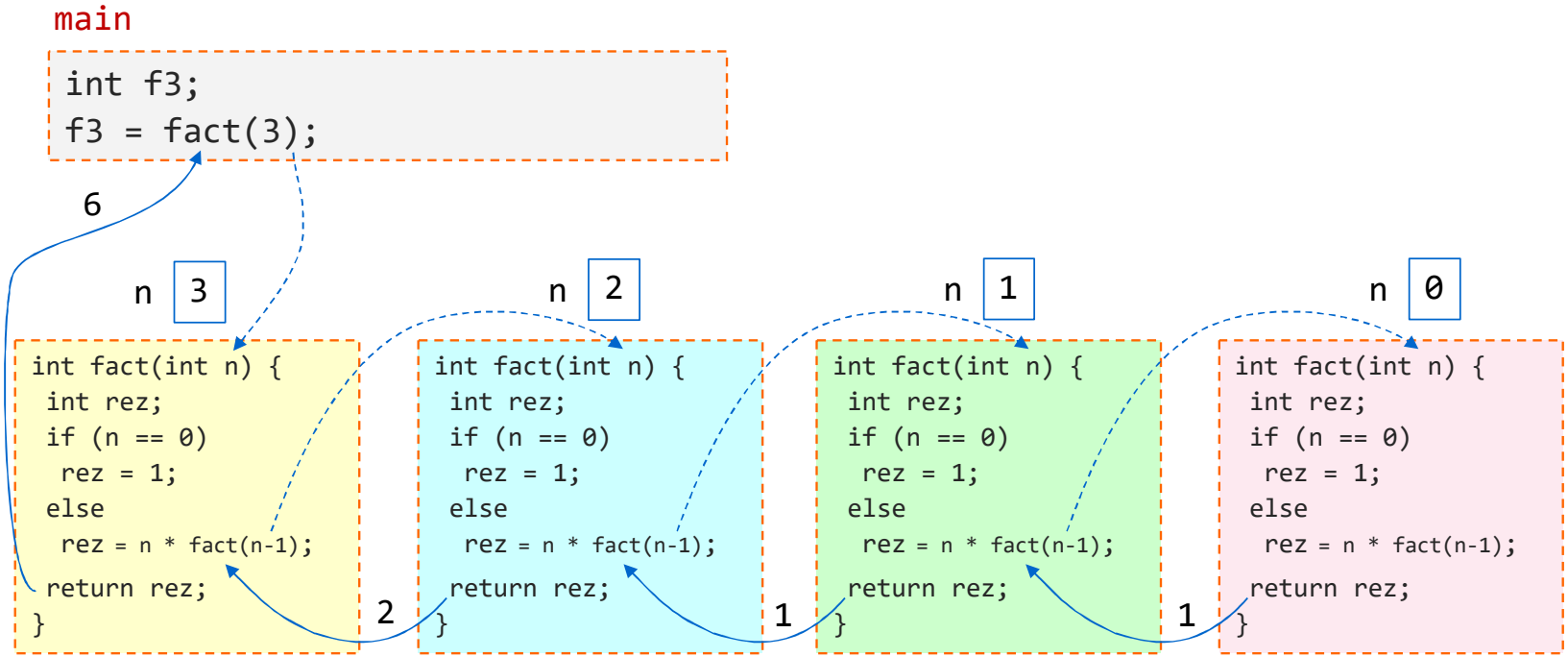
- Matematički rekurzivni izrazi često se bez teškoća pretvaraju u definiciju rekurzivne funkcije u programskom jeziku

```
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*

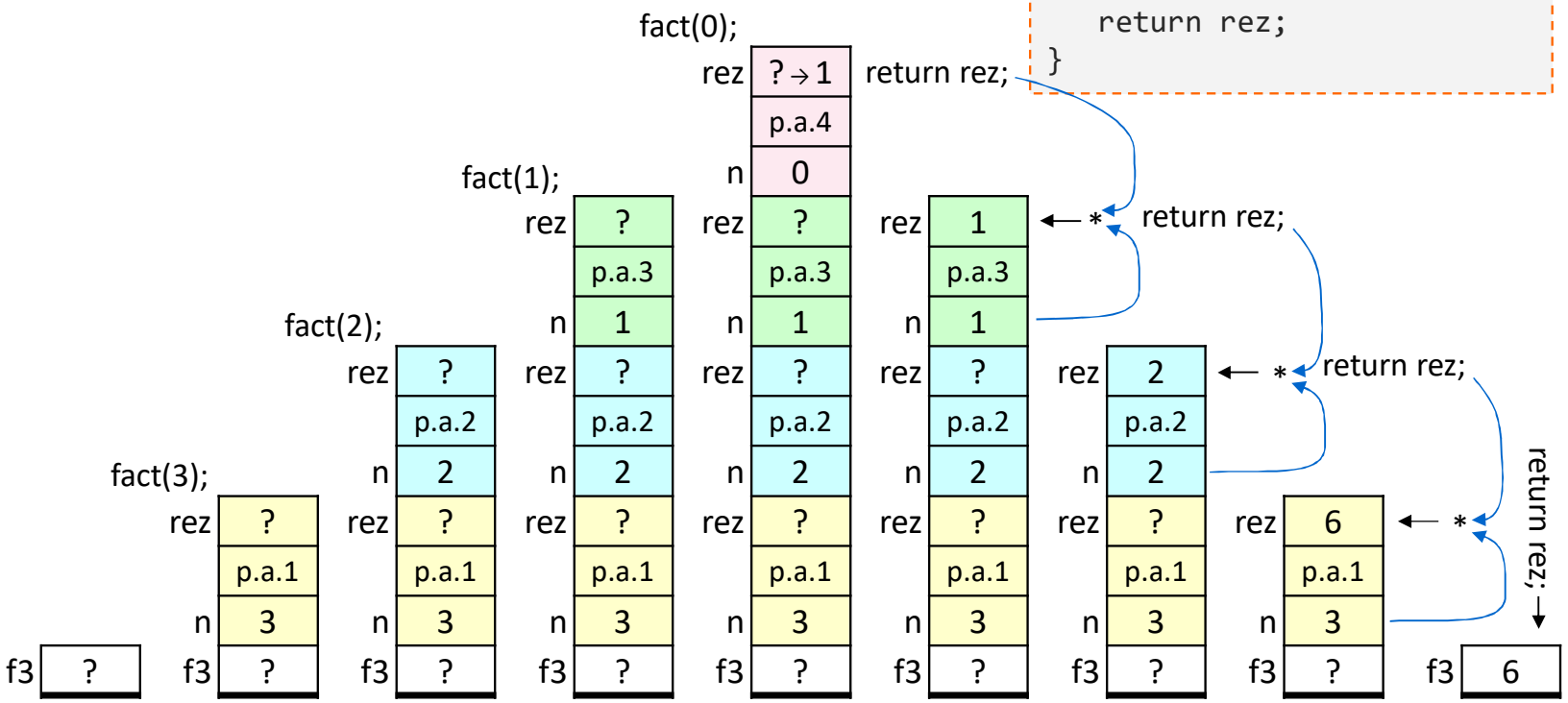


Primjer: stog za poziv `fact(3)`

```
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);
}
```

Zadatak

- Preradimo naš program iz četvrtog predavanja tako da napravimo rekurzivnu funkciju koja ispisuje binarnu reprezentaciju broja (bez korištenja polja):

Primjer

- Programski zadatak
  - Učitati nenegativni cijeli broj. Nije potrebno provjeravati ispravnost unesenog broja. Ispisivati ostatke uzastopnog dijeljenja učitanog broja s 2, a postupak prekinuti kad se dijeljenjem dođe do 0
    - učitani broj može biti 0. Može se dogoditi da se neće ispisati niti jedan ostatak dijeljenja, odnosno da se tijelo petlje neće izvršiti niti jednom
  - Primjeri izvršavanja programa

```
Upisite nenegativan cijeli broj > 11↵
Upisali ste 11↵
Ostatak = 1↵
Ostatak = 1↵
Ostatak = 0↵
Ostatak = 1↵
```

```
Upisite nenegativan cijeli broj > 0↵
Upisali ste 0↵
```

Uvod u programiranje 2024./2025. 31

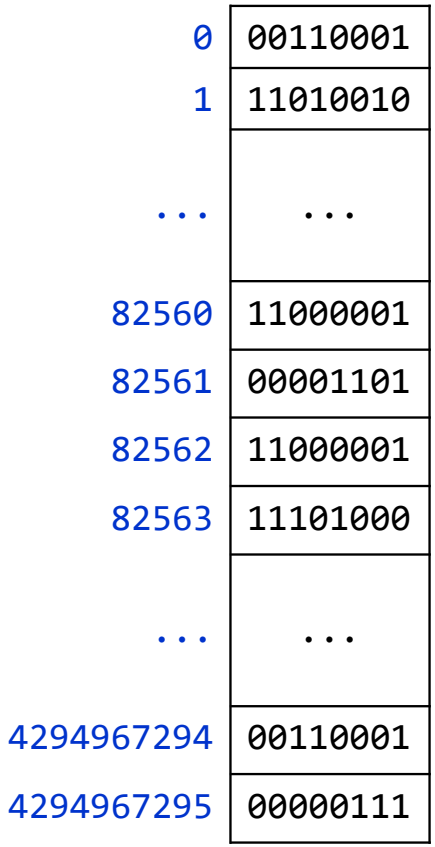
```
Upisite nenegativan cijeli broj : 11

11 = 1011
```

Tip podatka *pokazivač*

Uvod

Radna memorija računala

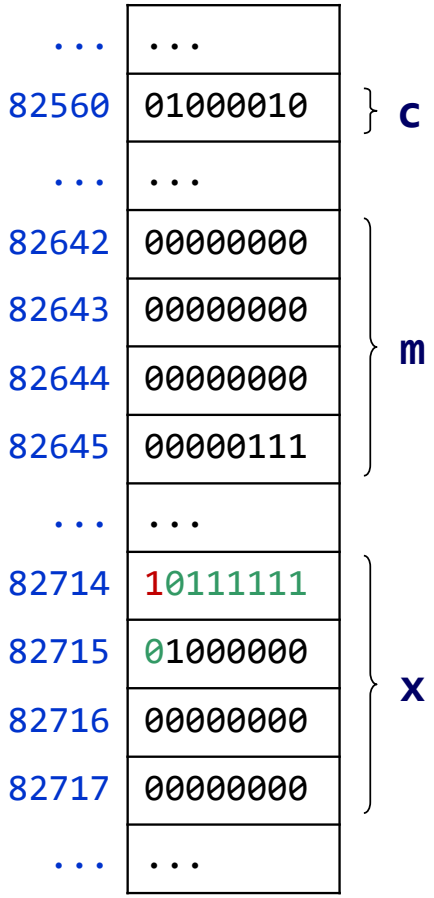
- Radna memorija računala može se promatrati kao kontinuirani niz bajtova, od kojih svaki ima svoj "redni broj", odnosno *adresu*
  - slikom je ilustrirana memorija veličine 4GB



Objekti i vrijednosti u programskom jeziku C

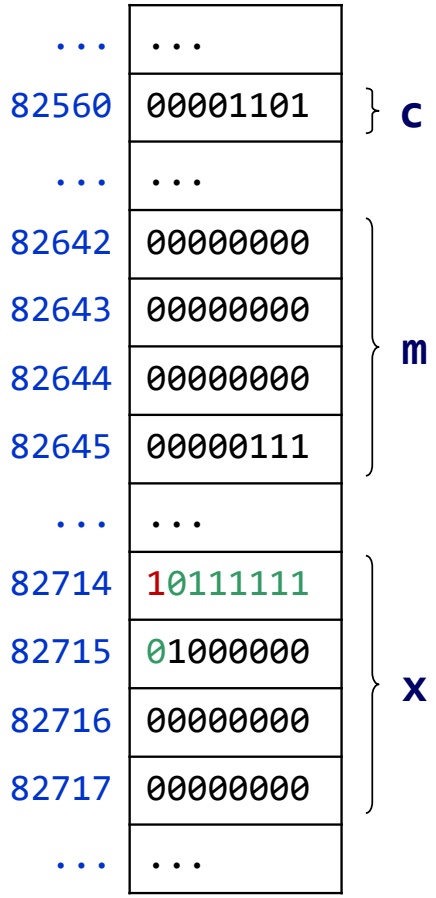
- Objekt (*object*) je područje u memoriji čiji sadržaj reprezentira vrijednost
- Vrijednost (*value*) je interpretacija sadržaja objekta koja se temelji na *tipu* **i** *sadržaju* objekta

```
...
char c = 'B';
...
int m = 7;
...
float x = -0.75f;
...
```



Adresa objekta

- Za objekt kažemo da se nalazi na adresi *A* (ili adresa objekta je *A*) ako je prvi bajt sadržaja objekta pohranjen na adresi *A*
  - Npr. varijabla `m` nalazi se na adresi 82642, odnosno adresa varijable `m` je 82642

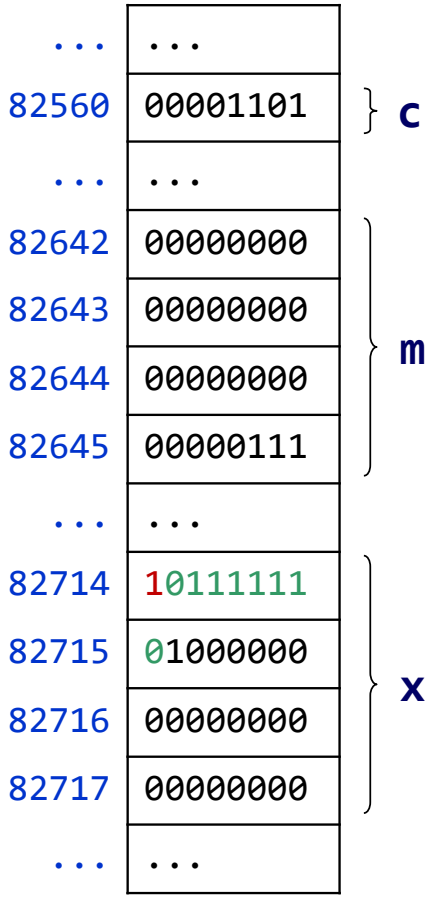


- Radi ilustracije pretpostavljeno je da se varijable nalaze na prikazanim adresama. U stvarnosti, nemoguće je znati o kojim se točno adresama radi prije nego se program pokrene (a nije niti važno znati ih unaprijed).

Kako se u programu dolazi do vrijednosti objekta

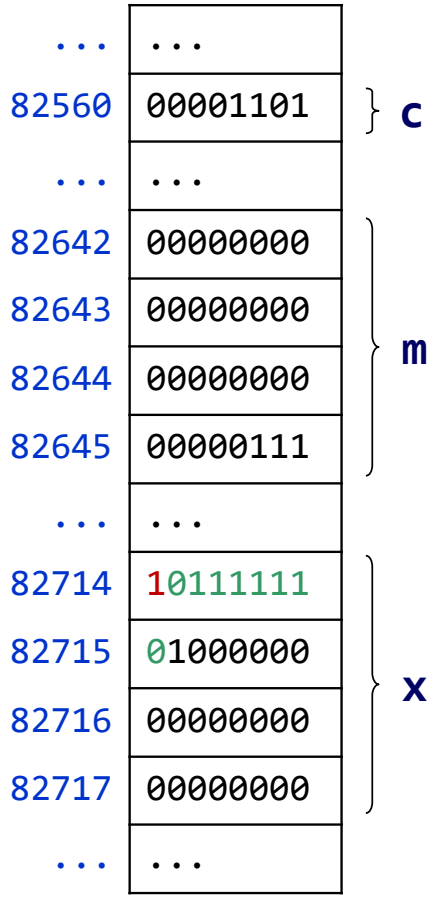
- Pristupanje objektu pomoću *identifikatora*
  - ime varijable (za skalarne tipove), ime varijable i indeks (za polja), ime varijable i ime člana (za strukture), ...
  - navođenjem identifikatora objekta dobiva se *lvalue* koji se može koristiti za čitanje ili postavljanje vrijednosti objekta
    - tip podatka poznat je iz definicije varijable
    - tip je važan: npr. ako tip podatka ne bi bio poznat, ne bi bilo moguće ispravno obavljati operacije

```
double y;
y = m + x;
```



Može li se objektu pristupiti pomoću adrese?

- Može li se do vrijednosti doći pomoću (samo) adrese objekta?
  - npr. ako je poznato da se neki objekt nalazi na adresi 82642?
  - ne, samo adresa nije dovoljna
- Za ispravno pristupanje objektu potrebna je ***i** adresa **i** tip objekta* koji se nalazi na toj adresi
  - adresa i tip objekta predstavljaju jedan oblik *reference* na taj objekt
  - tip objekta kojem se pristupa pomoću *reference* naziva se **referencirani tip** (*referenced type*)



Tip podatka *pokazivač* (*pointer type*)

- Tip podatka koji omogućuje pristupanje objektu pomoću *reference*
  - Ako je referencirani objekt tipa *T*, tada se za pristupanju objektu koristi tip podatka *pokazivač na T.* Npr. podatak tipa *pokazivač na int* omogućuje pristup objektu tipa `int`
  - Za tip podatka *pokazivač* ne postoje zasebne ključne riječi (kao za tipove podataka `int`, `float`, itd.). Tip podatka *pokazivač* opisuje se pomoću naziva referenciranog tipa i znaka `*`

```
int *p1, *p2;
float *p3;
```

  - Varijable `p1` i `p2` su tipa pokazivač na `int`
  - Varijabla `p3` je tipa pokazivač na `float`

Varijable tipa pokazivač

- Za varijablu *tipa pokazivač* vrijedi sve što je do sada navedeno o varijablama ostalih skalarnih tipova, osim:
  - definira se na malo drugačiji način: navođenjem imena *referenciranog tipa* i znaka * ispred imena varijable
  - pohranjuje podatke tipa pokazivač na referencirani tip

```
int m;
```

```
int *p1;
```

referencirani tip

varijabla `p1` nije tipa `int`, nego tipa *pokazivač na* `int`

- dopušteno je u istoj naredbi definirati varijable referenciranog tipa i varijable tipa pokazivača na referencirani tip

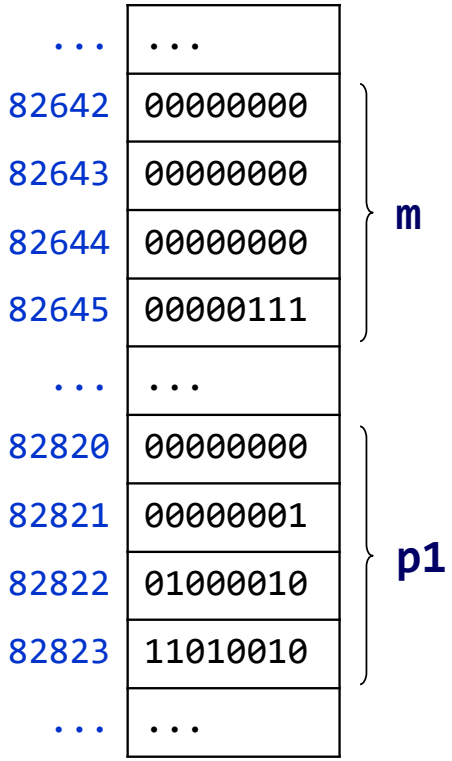
```
int m, *p1, *p2, k;
```

Koju vrijednost upisati u varijablu tipa pokazivač

```
int m = 7, *p1;
p1 = ?
```

- Općenito, ako je x varijabla (ili član polja, ili struktura ili član strukture, ...), tada je &x pokazivač na x
  - & je tzv. *adresni operator*. Rezultat izraza &m je *pokazivač na* `int` jer je `m` objekt tipa `int`
    - adresa odgovara adresi varijable `m` (82642)
    - rezultat je tipa pokazivač na `int`
  - budući da je rezultat izraza &m *pokazivač na int*, smije se pridružiti varijabli `p1` (koja je tipa *pokazivač na int*)

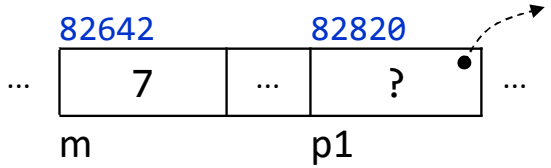
```
p1 = &m;
```



Primjer

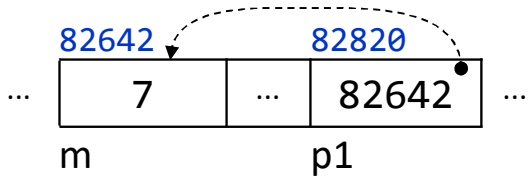
- U nastavku ćemo sadržaj memorije prikazivati na prikladniji način

```
int m = 7, *p1;
```



  - varijabla p1 još uvijek nije inicijalizirana: pokazivač pohranjen u varijabli p1 "pokazuje u nepoznato"

```
p1 = &m;
```



  - u varijablu p1 sada je upisan podatak tipa *pokazivač na int* kojim se može pristupiti objektu tipa `int` na adresi 82642
    - radi pojednostavljenja, koristit će se kolokvijalni izrazi:
      - naredbom `int *p1;` definiran je pokazivač p1
      - naredbom `p1 = &m;` u p1 je upisana adresa varijable m
      - p1 pokazuje na objekt na adresi 82642
      - p1 pokazuje na varijablu m, p1 pokazuje na objekt m

Inicijalizacija varijable tipa pokazivača uz definiciju

- Jednako kao i varijable drugih tipova, varijable tipa pokazivač mogu se inicijalizirati u trenutku definicije

```
int m, *p1 = &m, *p2 = p1;
float x, *p3 = &x, y, *p4 = &y;
```

  - voditi računa o redoslijedu definicije i inicijalizacije. Objekt čija se adresa izračunava adresnim operatorom mora biti definiran

```
int *p1 = &m, m;
```

Neispravno, može se popraviti premještanjem

  - voditi računa o tome da i varijabla tipa pokazivača može sadržavati "smeće" (*garbage value*)

```
int m, *p1;
int *p2 = p1;
p1 = &m;
```

U ovom trenutku p1 još uvijek sadrži "smeće"
Može se popraviti premještanjem naredbe

Paziti na razlike u tipovima pokazivača

- Tipovi pokazivača su međusobno različiti ako se razlikuju njihovi referencirani tipovi
  - u varijable jednog tipa pokazivača nije dopušteno upisivati pokazivače drugog tipa

```
int m;
int *pInt;
float x;
float *pFloat;

pInt = &m;
pFloat = &x;

pFloat = pInt;          Neispravno
pInt = &x;              Neispravno
pFloat = &m;            Neispravno
```

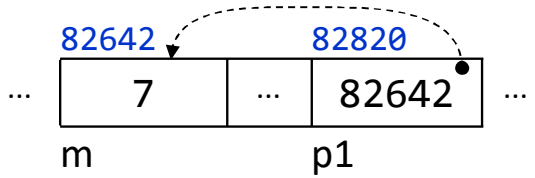
Adresa nije cijeli broj

- Iako *izgleda* kao cijeli broj, adresa u općem slučaju nije `int` (niti `short`, niti `long`, ...). Stoga nema smisla:
  - pokazivač pohranjivati u varijablu tipa `int`
  - cijeli broj pohranjivati u varijablu tipa pokazivača

```
int *p1;
int m;
m = 5;
p1 = &m;

p1 = m;                    Neispravno
m = p1;                    Neispravno
```

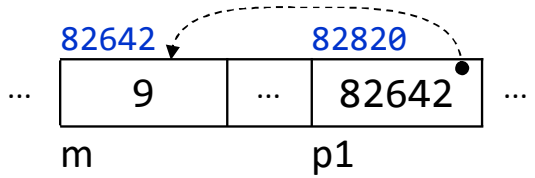
Pristupanje objektu pomoću pokazivača

```
int m = 7, *p1 = &m;
```



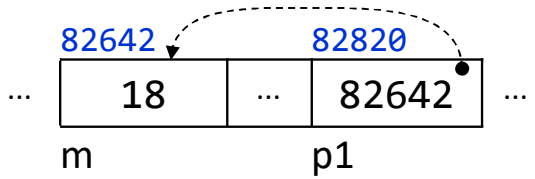
- objektu (7, tip `int`) na adresi 82642 može se pristupiti:
  - (naravno) pomoću imena varijable `m`

```
m = m + 2;
```



  - ali također i primjenom *operatora indirekcije* (unarni operator *) nad pokazivačem pohranjenim u varijabli `p1`

```
*p1 = 2 * *p1;
```



pročitaj cijeli broj s mjesta na kojeg pokazuje p1, dobiveni rezultat (tipa int) pomnoži s 2 i rezultat upiši na mjesto kamo pokazuje p1

```
printf("%d %d", m, *p1);
```

```
18 18
```

Operator indirekcije *

- Operator omogućuje da se objektu pristupi *indirektno* pomoću pokazivača (umjesto *direktno* preko imena varijable)
  - operator je također poznat pod imenom *operator dereferenciranja* jer operator "*dereferencira*" pokazivač (*referencu* na objekt) i tako dolazi do objekta
- Općenito, ako `p` pokazuje na objekt `x`, tada je rezultat operacije `*p` *lvalue* koja predstavlja objekt na kojeg pokazuje `p`
  - to znači: ako je `p` varijabla koja sadrži pokazivač koji pokazuje na objekt u memoriji koji predstavlja varijablu `m`, tada se izraz `*p` može koristiti na svakom mjestu u programu gdje se može koristiti ime varijable `m`
    - za čitanje vrijednosti (npr. u nekom izrazu)
    - za postavljanje vrijednosti (kao lijeva strana izraza pridruživanja), uz uvjet da je sadržaj objekta izmjenljiv

Neke oznake su pomalo zbunjujuće?

```
int m = 7;
int *p1 = &m;
...
p1 = &m;        Ispravno
*p1 = &m;       Neispravno
```

- Kako to da je u naredbi za definiciju varijable p1 ispravno napisati `*p1 = &m`, a naredba `*p1 = &m;` je neispravna?

- u programskom jeziku C isti simboli u različitom kontekstu mogu imati različito značenje

```
int *p1 = &m;
```

ovdje simbol * ne predstavlja operator indirekcije, nego označava da varijabla p1 nije tipa int, nego tipa *pokazivač na int*

p1 definiraj kao varijablu tipa pokazivač na int

varijablu koju si upravo definirao, p1, inicijaliziraj na vrijednost &m

```
*p1 = &m;
```

neispravno jer je rezultat izraza *p1 objekt tipa int, što znači da se u objekt tipa int pokušava upisati vrijednost tipa pokazivač na int

Neke oznake su pomalo zbunjujuće?

```
int m = 7;
int *p1 = &m, *p2 = p1;
...
p2 = p1;        Ispravno
*p2 = p1;       Neispravno
```

- Kako to da je u naredbi za definiciju varijable p2 ispravno napisati `*p2 = p1` , a naredba `*p2 = p1;` je neispravna?

```
int *p1 = &m, *p2 = p1;
```

definirana je varijabla `p1`, inicijalizirana je na `&m`, zatim je definirana varijabla `p2` koja se inicijalizira na vrijednost koja se nalazi u `p1`.

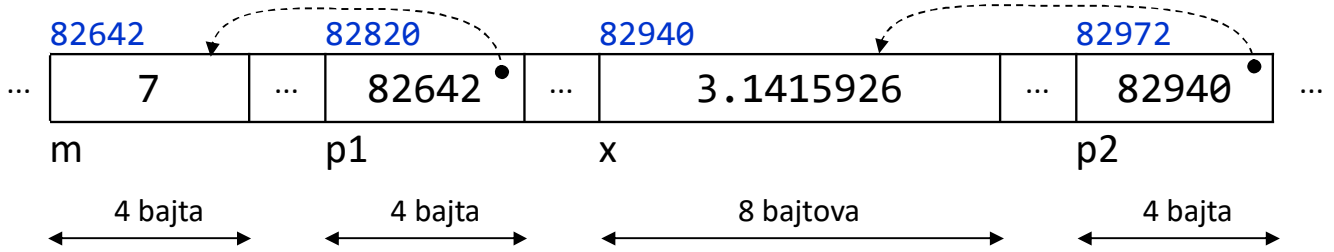
```
p2 = *p1;
```

neispravno jer je rezultat izraza `*p1` objekt tipa `int`, što znači da se vrijednost tipa `int` pokušava upisati u varijablu tipa pokazivač na `int`

Koliko prostora zauzima pokazivač

- Adresa objekta je adresa na kojoj je pohranjen prvi bajt objekta
  - to znači da veličina referenciranog tipa ne bi trebala utjecati na veličinu prostora koju zauzima pokazivač na taj tip

```
int m = 7, *p1 = &m;
double x = 3.1415926, *p2 = &x;
```



  - jednaki prostor (4 bajta) zauzimaju pokazivač `p1` na objekt tipa `int` (koji je veličine 4 bajta) i pokazivač `p2` na objekt tipa `double` (koji je veličine 8 bajtova)

Koliko prostora zauzima pokazivač

- Pokazivači na jednoj platformi (isti operacijski sustav, arhitektura i prevodilac) u principu* zauzimaju jednaku količinu memorije bez obzira na koji tip podatka pokazuju

```
int m = 7, *p1 = &m;
double x = 3.1415926, *p2 = &x;
printf("%u %u %u\n", sizeof(p1), sizeof(m), sizeof(*p1));
printf("%u %u %u", sizeof(p2), sizeof(x), sizeof(*p2));
```

x86_64, Windows, gcc

```
4 4 4
4 8 8
```

x86_64, Linux, gcc

```
8 4 4
8 8 8
```

* U praksi je to uglavnom tako, ali s obzirom da C standard takvo pravilo izrijekom ne propisuje, ne smije se u potpunosti isključiti mogućnost da će se na nekoj platformi veličine pokazivača međusobno razlikovati s obzirom na tip podatka na koji pokazuju.

Generički pokazivač (*pointer to void*)

- Referencirani tip pokazivača mora biti poznat kako bi se na temelju adrese (gdje je objekt) i tipa (kojeg tipa je objekt na toj adresi) sadržaj objekta mogao ispravno interpretirati
- Međutim, postoji specijalni tip pokazivača za kojeg to ne vrijedi
  - *generički pokazivač* (u literaturi također: *pokazivač na void*, *pointer to void*) je pokazivač koji može pokazivati na objekt bilo kojeg tipa
  - budući da referencirani tip generičkog pokazivača nije poznat, neće se moći koristiti za pristup objektu (kažemo: generički pokazivač se ne može *dereferencirati*)
  - ali zato je moguće napraviti eksplicitnu konverziju (*cast*) generičkog pokazivača na tip pokazivača za kojeg će referencirani tip biti T
    - rezultat sljedeće operacije nad generičkim pokazivačem je pokazivač na tip podatka T

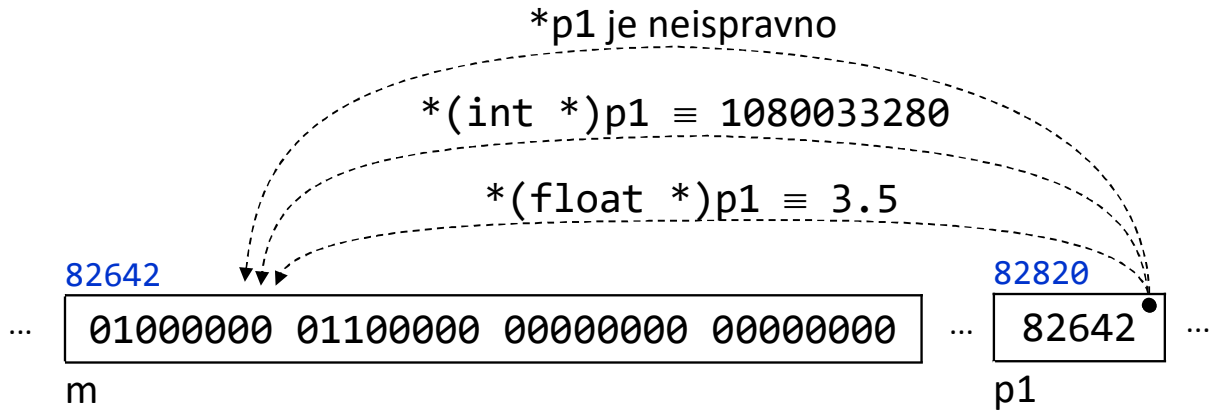
```
(T *) genericki_pokazivac
```

Primjer

```
int m = 1080033280;
void *p1;
p1 = &m;
// printf("%d", *p1);
printf("%d\n", *(int *)p1);
printf("%f\n", *(float *)p1);
```

Neispravno jer p1 nije moguće dereferencirati

```
1080033280
3.500000
```



Zadatak

- Upotrijebimo void* da ispišemo bitove odnosno IEEE754 reprezentaciju float broja (binarnu i hex).
  - Tj. uvjerimo se da su ti bitovi doista u memoriji tako postavljeni

```
Unesite broj: 3.14

P|   K  |          M           |
01000000010010001111010111000011
4048f5c3
```

Konverzijske specifikacije za printf i scanf

- konverzijska specifikacija %p koristi se za ispis i čitanje podatka tipa pokazivač
  - točan oblik ispisa nije propisan standardom (vrijednost će se ispisati kao broj u dekadskom ili heksadekadskom brojevnom sustavu ili u nekom drugom obliku)
    - argument (pokazivač) koji se ispisuje dobro je eksplicitno konvertirati u generički pokazivač, ali u većini slučajeva može se ispustiti

```
int m = 7, *p1 = &m;
printf("m je na adresi %p", (void *)p1);
// printf("m je na adresi %p", p1);
```

Može i bez (void *)

x86_64, Windows, gcc

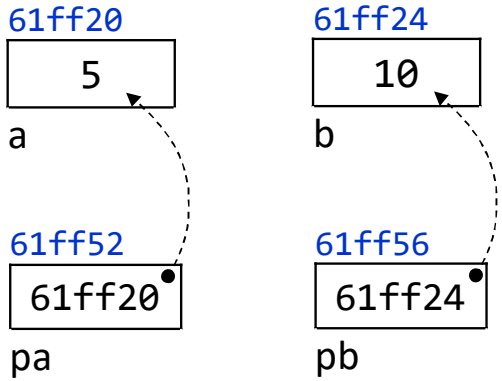
```
m je na adresi 0061ff28
```

x86_64, Linux, gcc

```
m je na adresi 0x7fffd6e8d324
```

Primjer

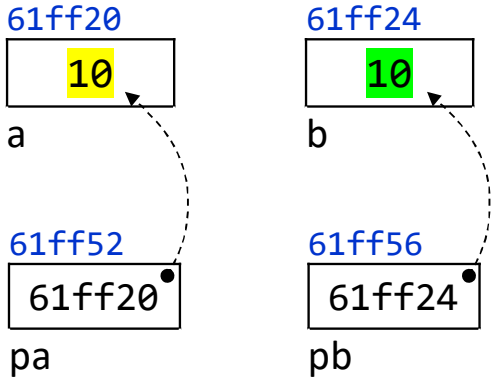
```
int a = 5, b = 10;
int *pa, *pb;
pa = &a;    // pretpostavka pa = 61ff20
pb = &b;    // pretpostavka pb = 61ff24
```



- što će se ispisati sljedećim odsječkom?

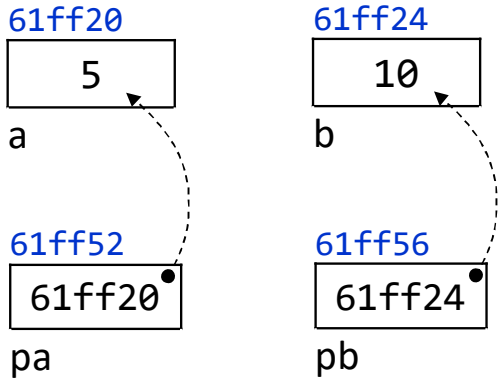
```
*pa = *pb;
printf("%d  %d\n", a, b);
printf("%p  %p\n", (void *)pa, (void *)pb);
printf("%d  %d\n", *pa, *pb);
```

```
10 10
61ff20 61ff24
10 10
```



Primjer

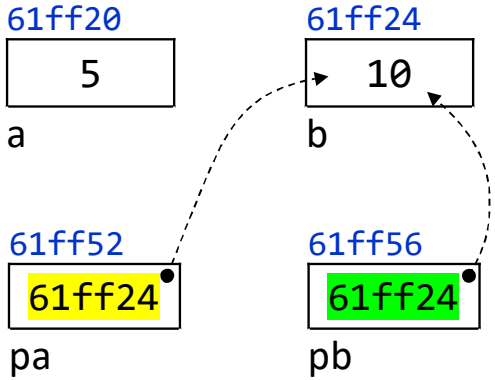
```
int a = 5, b = 10;
int *pa, *pb;
pa = &a;    // pretpostavka pa = 61ff20
pb = &b;    // pretpostavka pb = 61ff24
```



- što će se ispisati sljedećim odsječkom?

```
pa = pb;
printf("%d  %d\n", a, b);
printf("%p  %p\n", (void *)pa, (void *)pb);
printf("%d  %d\n", *pa, *pb);
```

```
5 10
61ff24 61ff24
10 10
```



Primjer

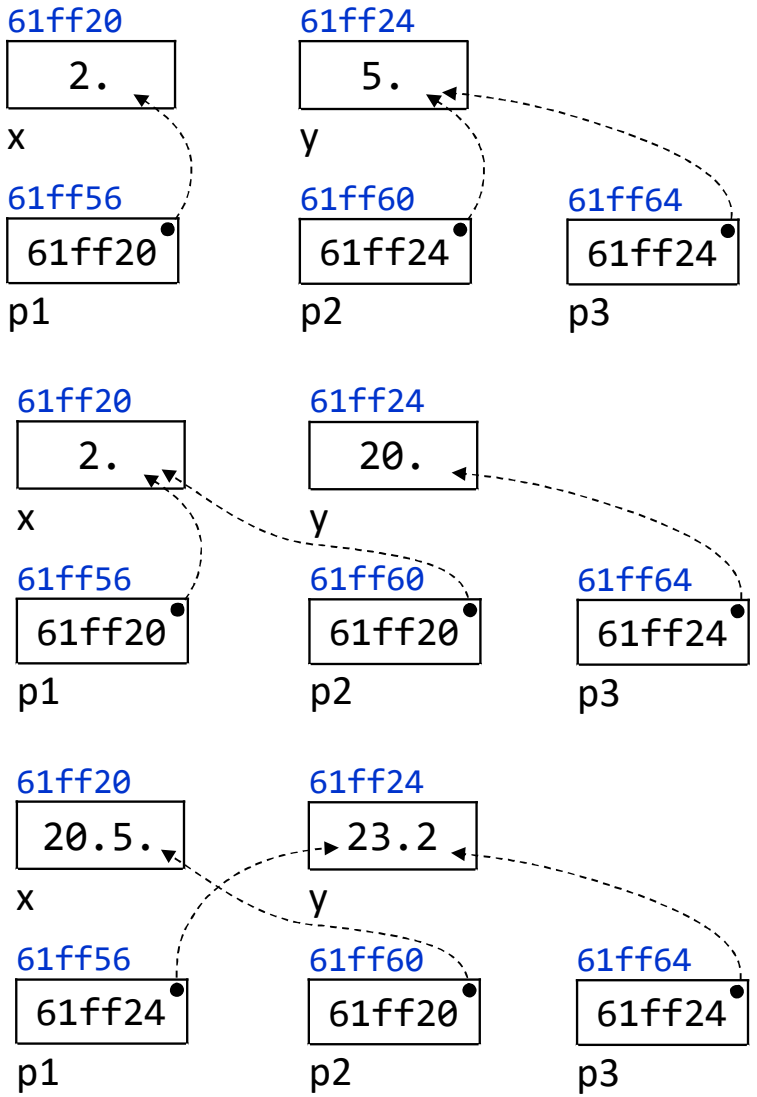
```
float x = 2.f, y = 5.f;
float *p1, *p2, *p3;
p1 = &x;
p2 = p3 = &y;
```

- nacrtati sliku nakon

```
*p3 = *p2 + 3.f * *p2;
p2 = p1;
```

- i nakon

```
*p2 = *p3 + 0.5f;
p1 = p3;
*p1 += 3.2f;
```



Prije sljedećeg predavanja

- Edgar:
  - Tutorial: **nema**
  - **13. vježbe uz predavanja**