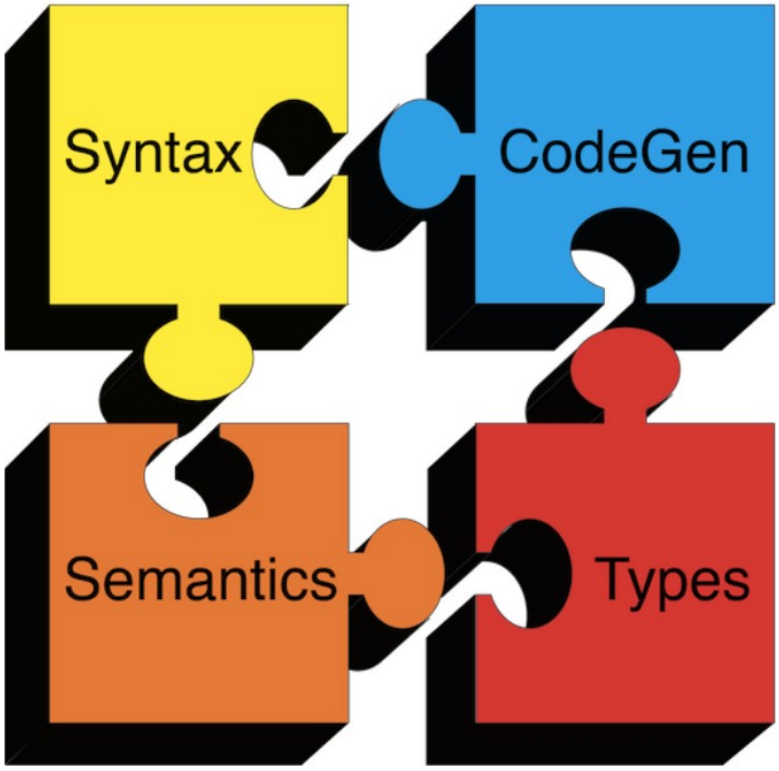
Upravljanje memorijom

preuzeto sa:
https://www.coursera.org/course/compilers

by Alex Aiken



pripremila:
Zorica Suvajdžin Rakić

Upravljanje memorijom

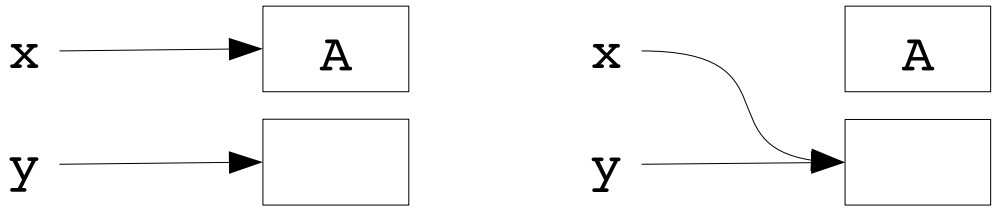
- *memory management*
- *storage management*
- veliki problem modernog programiranja
  - C i C++ programi imaju puno grešaka (*bugs*) u toku smeštanja podataka
    - zaboravljeno oslobađanje nekorišćene memorije
    - dereferenciranje *dangling* pointera
    - slučajno, nenamerno prepisivanje preko delova strukture podataka
    - ...
- Ovakve greške je teško pronaći
  - greška može proizvesti vidljiv efekat daleko od izvora greške
- Postoje dobro poznate tehnike za potpuno automatsko upravljanje memorijom
- Postalo je popularno sa Javom

Upravljanje memorijom

- Kada se kreira objekat, automatski se alocira nekorišćen prostor
  - npr: **`new X`**
- Posle nekog vremena, više neće biti nekorišćenog prostora
- Jedan deo prostora je zauzet objektima koji nikada više neće biti korišćeni
  - ovaj prostor se može osloboditi da bi se kasnije upotrebio
- Kako znamo da se objekat više neće koristiti?
  - napomena: program može koristiti samo one objekte koje može pronaći

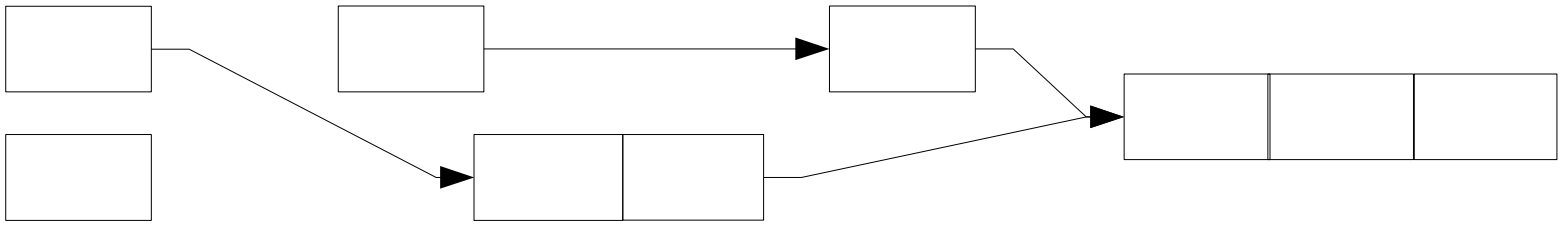
```
x = new A;
x = y;
```

A nedostupan



Upravljanje memorijom

- Objekat x je dostupan (*reachable*) ako i samo ako:
  - ako neki registar sadrži pokazivač na x, ili
    - (pokazivač vrha steka, LV, ...)
  - drugi dostupan objekat y sadrži pokazivač na x
- Svi dostupni objekti se mogu pronaći tako što se
  - pretraga započne od registara, a zatim se
  - prate svi pokazivači



- nedostupan objekat *(unreachable)* se nikada ne može koristiti
  - takvi objekti su smeće (*garbage*)

Upravljanje memorijom

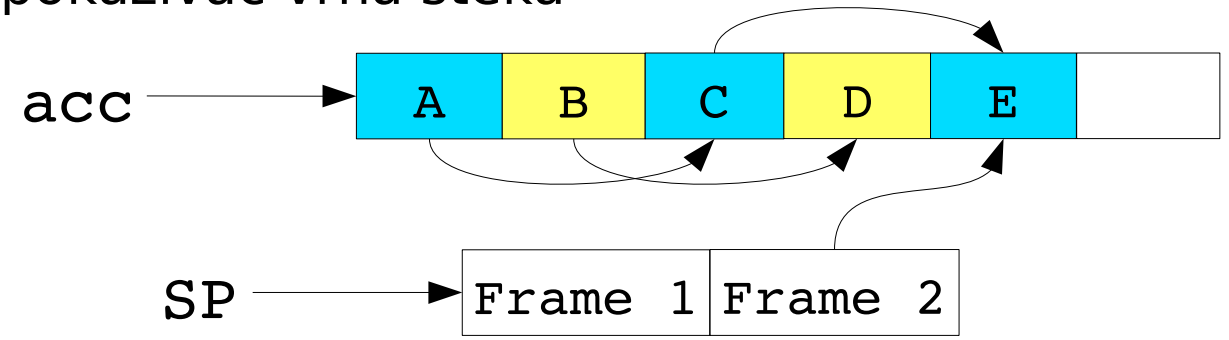
- Razmotrimo program

```
x = new A;
y = new B;
x = y;
GC ----------
x = new A;
```

- posle `x = y;` (pod pretpostavkom da `y` tu postaje mrtva)
  - objekat `A` je nedostupan
  - objekat `B` je dostupan (preko `x`)
  - `B` nije smeće i nije sakupljen
  - ali objekat `B` nikada neće biti korišćen
- dostupnost je aproksimacija
  - to što je objekat dostupan ne znači da će biti ponovo korišćen

Upravljanje memorijom

- Primer:
  - koreni pretrage (*root registers*)
    - akumulator
      - pokazuje na objekat
      - a ovaj objekat može pokazivati na drugi objekat, itd.
    - pokazivač vrha steka



- **A**, **C** i **E** objekti su dostupni
- **B** i **D** su nedostupni iz **acc** i **SP**
  - iako postoji pokazivač na **D**, to ne znači da je on dostupan
  - njihov prostor možemo ponovo iskoristiti

Upravljanje memorijom

- Svaka šema sakupljanja smeća (*garbage collection*, *GC*) ima sledeće korake
  1. alociraj neophodan prostor za nove objekte
  2. kada nestane prostora:
     a) izračunaj koji se objekti mogu ponovo koristiti
        - praćenjem objekata koji su dostupni iz korenskih registara
     b) oslobodi prostor koji je zauzet objektima koji nisu pronađeni u a)
- Sakupljanje smeća se sprovodi
  - pre nego što prostora zaista nestane
  - kada nestane prostora

GC: Mark and sweep

- Kada nestane memorije, GC izvršava 2 faze:
  - *mark*: fazu markiranja
    - pronalazi dostupne objekte
  - *sweep*: fazu čišćenja
    - pokupi objekte koji predstavljaju smeće
- Svaki objekat ima jedan dodatni bit: *mark* bit
  - rezervisan za upravljanje memorijom
  - inicijalno je postavljen na 0
  - postavlja se na 1 u dostupnim objektima u fazi markiranja

GC: Mark and sweep

```
// mark phase

let todo = { all roots }
while todo ≠ ∅ do
  pick v ∈ todo
  todo ← todo - { v }
  if mark(v) = 0 then   // v is unmarked yet
    mark(v) ← 1
    let v1,...,vn be the pointers contained in v
    todo ← todo ∪ {v1,...,vn}
  fi
od
```

GC: Mark and sweep

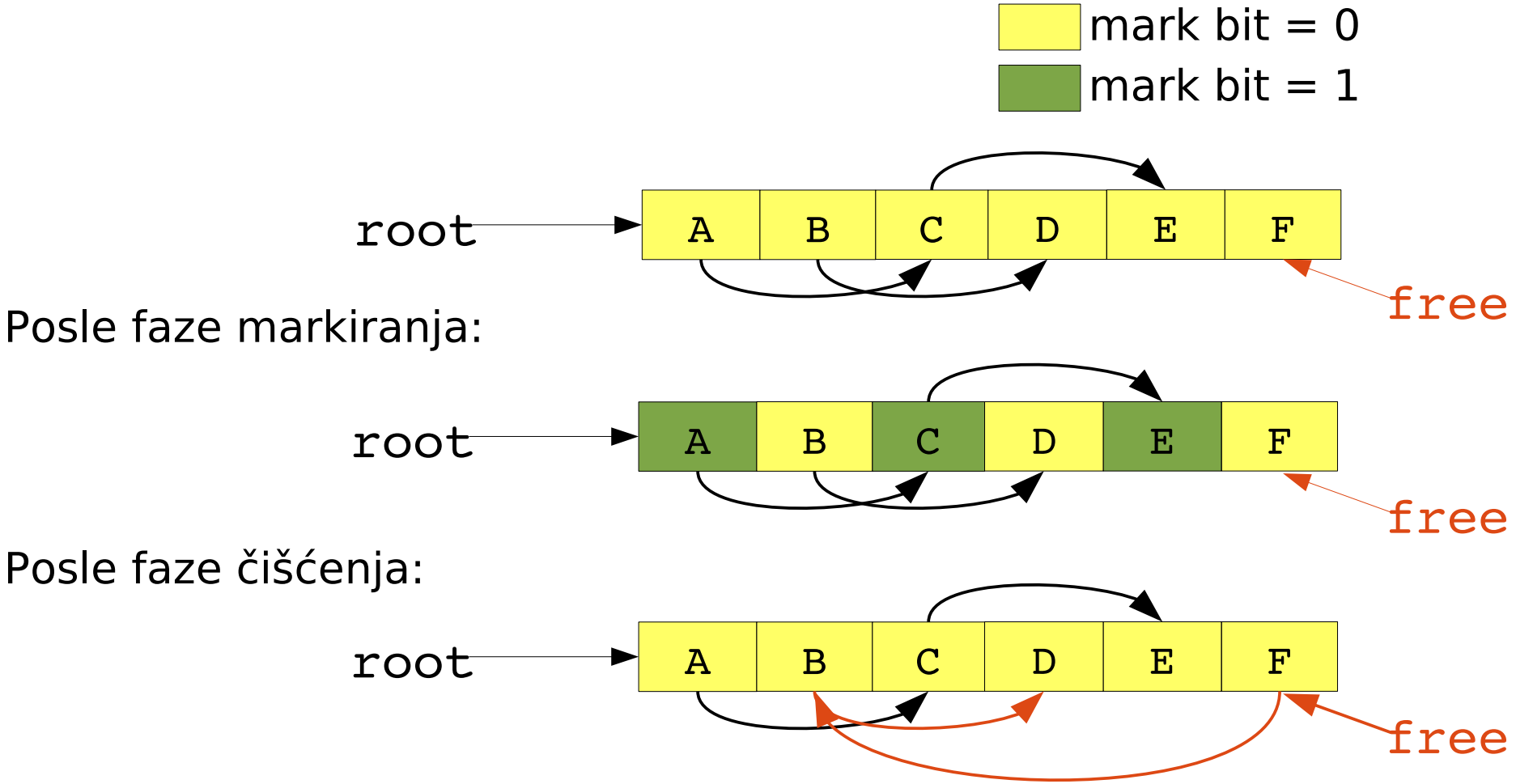
- *Sweep* faza skenira *heap* tražeći objekte sa *mark* bitom = 0
  - ovi objekti nisu bili posećeni u fazi markiranja
  - oni predstavljaju smeće
  - svi takvi objekti se dodaju u *free* listu
- Objekti koji su imali mark bit = 1 resetuju ga na 0

GC: Mark and sweep

```
// sweep phase
// sizeof(p) is the size of block starting at p

p ← bottom of heap
while p < top of heap do
  if mark(p) = 1 then
    mark(p) ← 0
  else
    add block p to freelist
  fi
  p ← p + sizeof(p)
od
```

GC: Mark and sweep



GC: Mark and sweep

- Jednostavan algoritam - tipičan GC algoritam
- Ozbiljan problem u fazi markiranja:
  - poziva se tek kada je nestalo memorijskog prostora
  - ali potreban mu je prostor za pravljenje *todo* liste
  - veličina *todo* liste je neograničena, pa se taj prostor ne može rezervisati ranije
- Rešenje:
  - *todo* lista se koristi kao pomoćna struktura podataka za analizu dostupnosti
  - postoji trik kojim se pomoćna struktura podataka smešta u samim objektima (*pointer reversal*)
  - slično, *free* lista se može smestiti u samim *free* objektima

GC: Mark and sweep

- Prostor za novi objekat se alocira iz *new* liste
  - odabere se dovoljno velik blok
  - alocira se neophodna veličina memorije iz tog bloka
  - ostatak se vrati u *free* listu
- *Mark and sweep* može fragmentirati memoriju
- Prednost: objekti se ne pomeraju tokom GC
  - nema potrebe da se ažuriraju pokazivači na objekte
  - radi za jezike kao što su C i C++

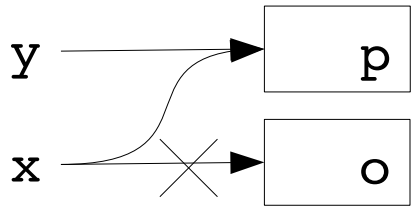
GC: Reference Counting

- Strategija:
  - sakupi objekat čim nema više pokazivača koji pokazuju na njega
  - ne čeka se nestanak memorije
- Kako?
  - brojati pokazivače na objekat
  - u svakom objektu smestiti broj pokazivača koji pokazuju na njega
    - to je broj referenci (*reference count*)
  - svaka operacija dodele mora ažurirati broj referenci

GC: Reference Counting

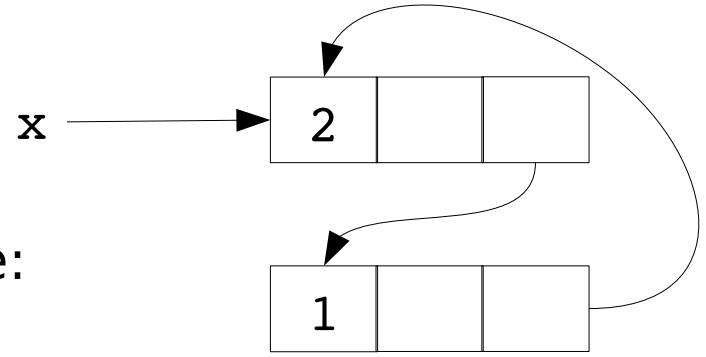
- **new** vraća objekat sa brojem referenci = 1
  - neka je **rc(x)** broj referenci na **x**
  - pp: **x**, **y** pokazuju na objekte **o**, **p**
  - svaka dodela **x ← y** postaje:

```
rc(p) ← rc(p) + 1
rc(o) ← rc(o) - 1
if(rc(o) == 0) then free o
x ← y
```



GC: Reference Counting

- Prednosti:
  - lako se implementira
    - u kompajleru se menja samo iskaz dodele
  - sakuplja smeće inkrementalno bez velikih pauza u izvršavanju
    - interaktivne aplikacije
    - *real-time* aplikacije



- Mane:
  - ne može sakupiti cirkularne strukture:
    - **`x ← null`**
    - povremeno se pusti *mark and sweep* GC koji može da detektuje cirkularnost
  - ažuriranje broja referenci pri svakoj operaciji dodele je vrlo sporo
    - značajan uticaj na program
    - optimizujući kompajleri mogu to dosta dobro da reše

Upravljanje memorijom: pregled

- Automatsko upravljanje memorijom sprečava ozbiljne greške
  - programer ne mora da brine o puno strvari
  - produktivniji način programiranja
  - ako program odgovara - koristiti
- Ali smanjuje kontrolu programera
  - npr: gde su podaci u memoriji (*layout*)
  - npr: kada se dealocira memorija
  - ne koristi se
    - kada program ima puno podataka i
    - kada je potrebna efikasna upotreba memorije
    - npr: naučna računanja
- Pauze su problematične u *real-time* aplikacijama
- *Memory leaks* su mogući (čak verovatni)

Upravljanje memorijom: pregled

- GC je veoma bitna stavka implementacije jezika
- Postoje napredniji GC algoritmi (od pokazanih):
  - konkurentni
    - program se izvršava dok se dešava sakupljanje
  - *real time*
    - ograničava trajanje pauze
  - paralelni
    - nekoliko kolektora radi istovremeno