Squawk – Status Update

Nik Shaylor
Doug Simon

What's new

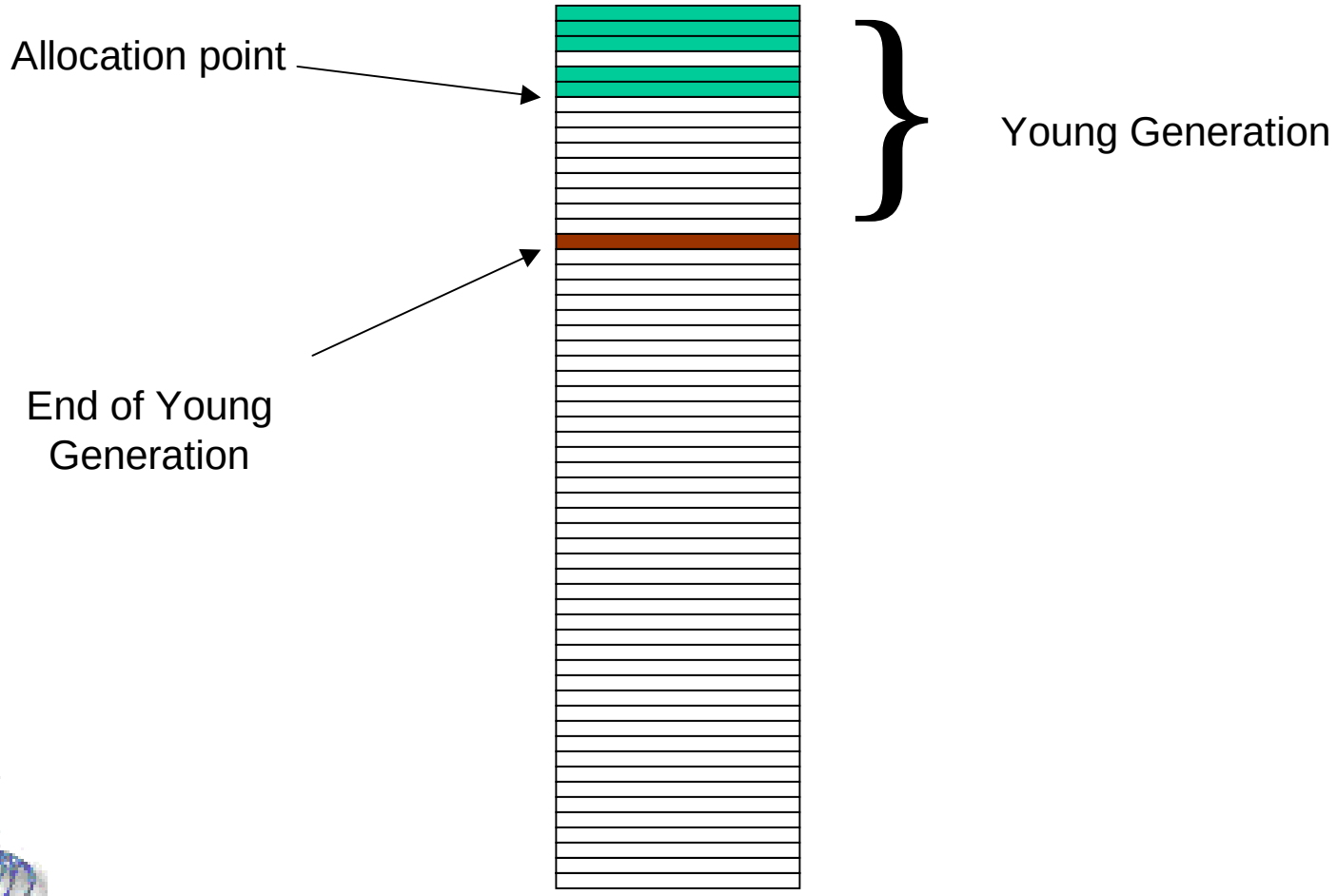
- Prototype RAM usage numbers
- Compacting, generational garbage collector for working storage (RAM)
- Mark-sweep, non-compacting collector for persistent memory (EEPROM)
- Migration/copying objects from RAM → EEPROM
- Loader runs inside Squawk
- Support for finalizers and auto-migration

Loader & RAM numbers

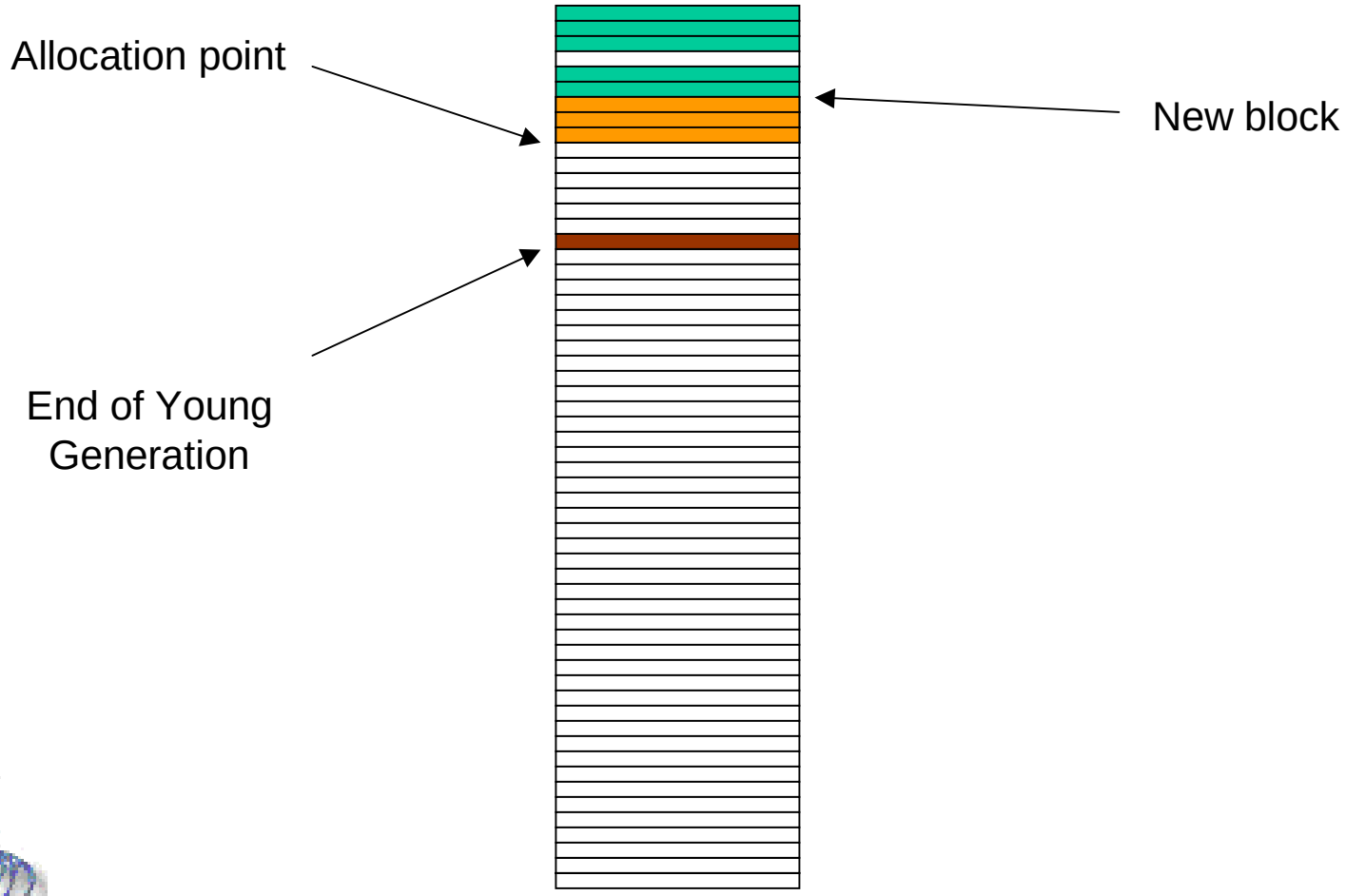
- The suite loader now runs in Squawk
  - Uses RAM → EEPROM migration functionality
- RAM usage by Squawk VM when running 'nop' program:
  - Java Heap: ~844 bytes
  - Native stack & data: ~532 bytes
- ROM required for Java Purse: 4516 bytes

RAM architecture

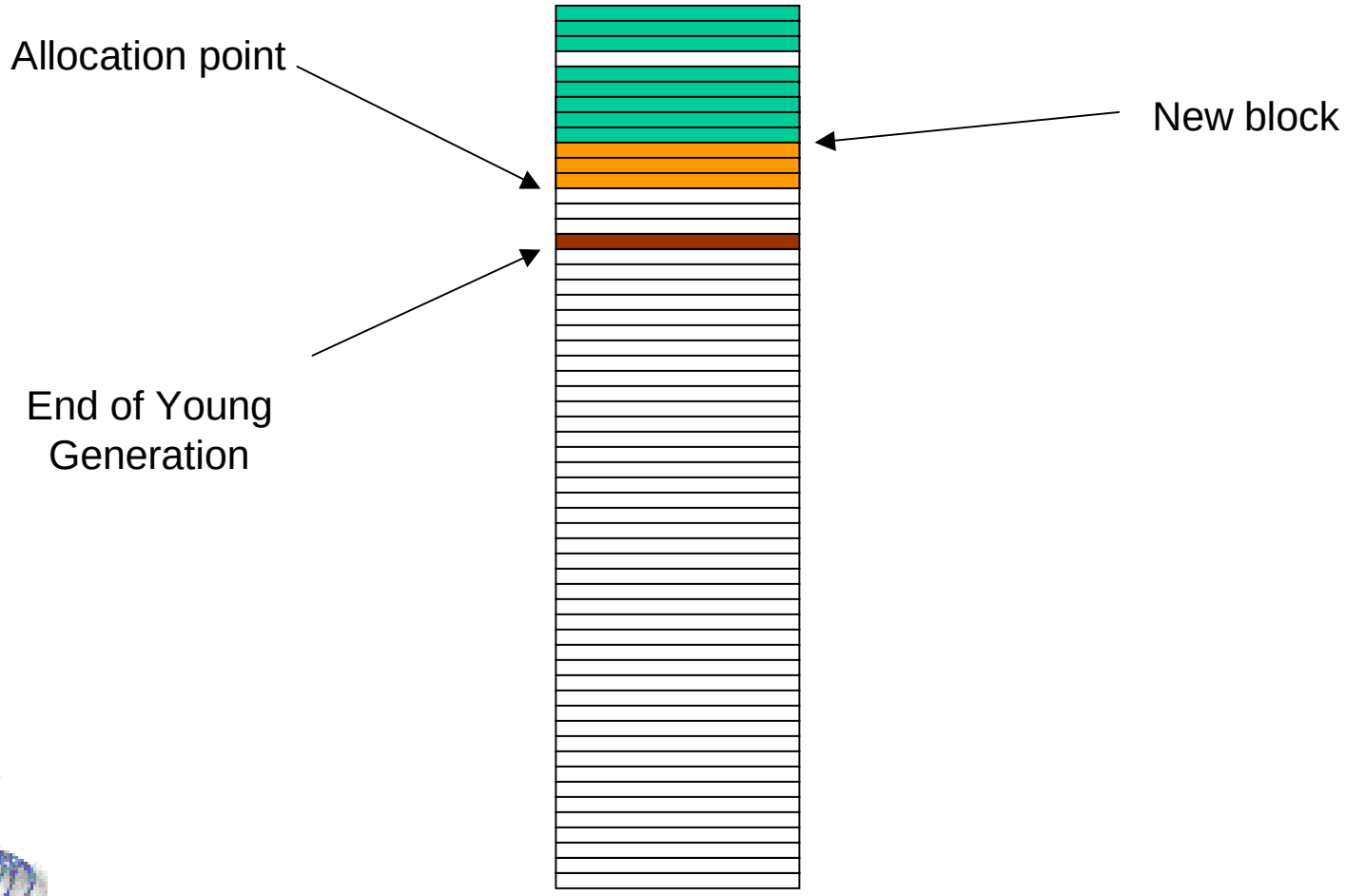
RAM architecture



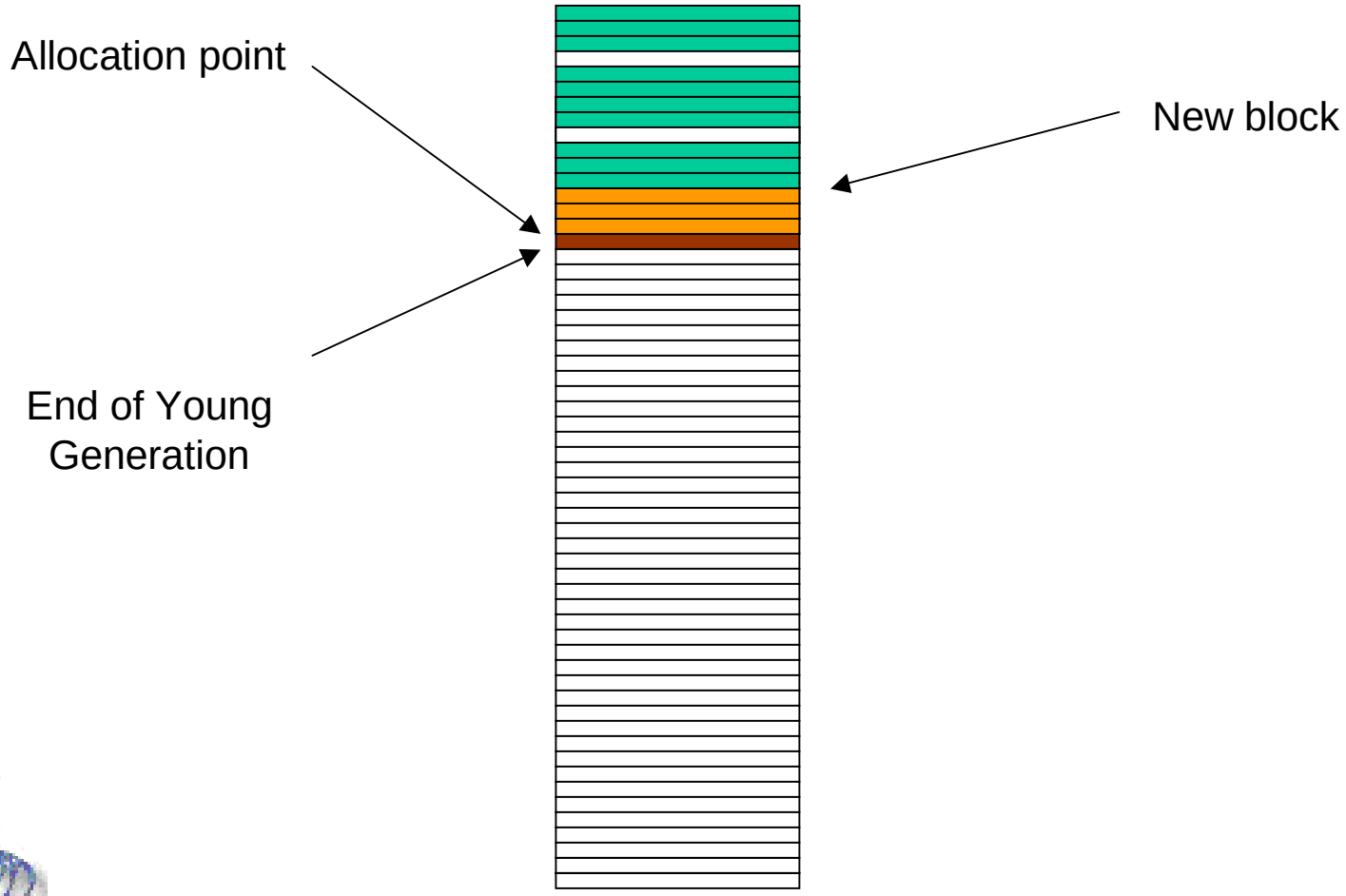
RAM architecture



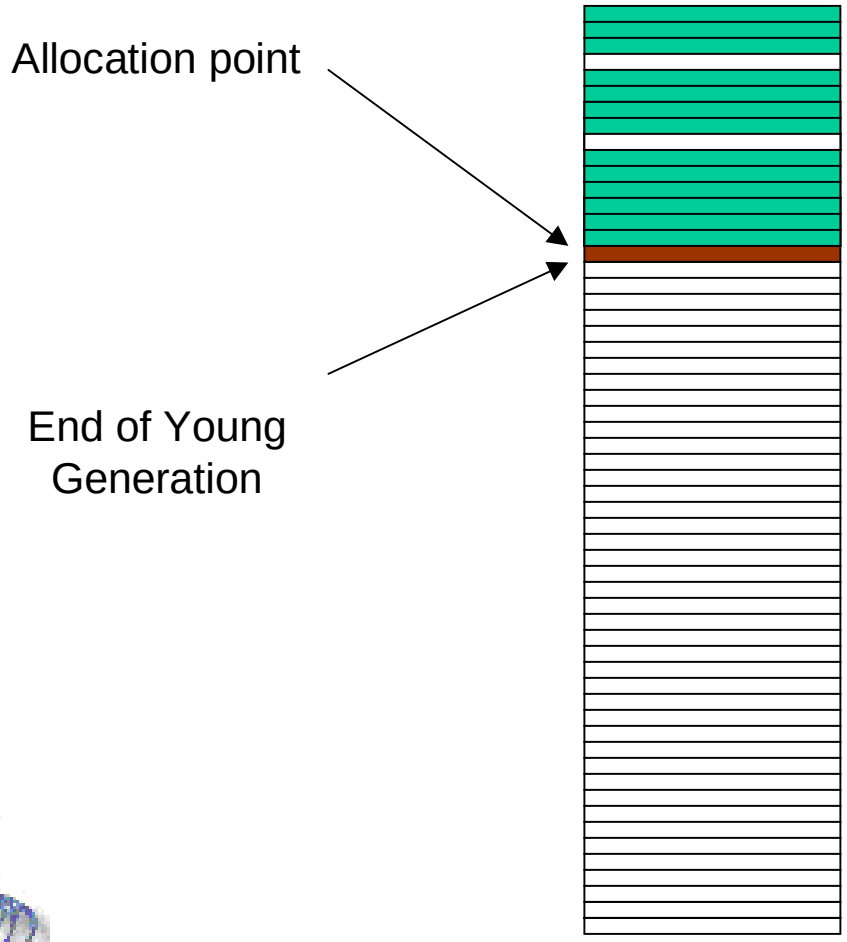
RAM architecture



RAM architecture

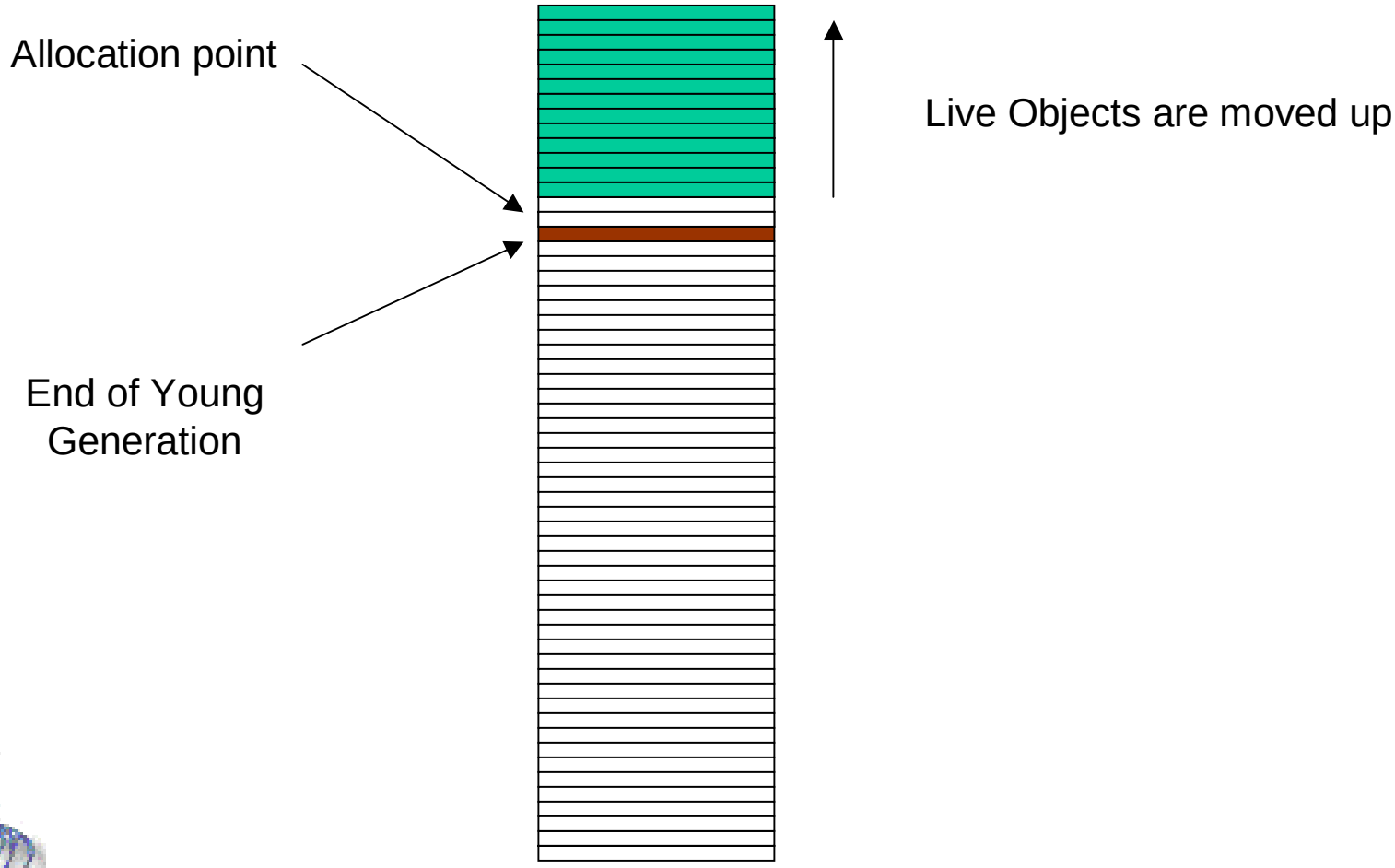


RAM architecture

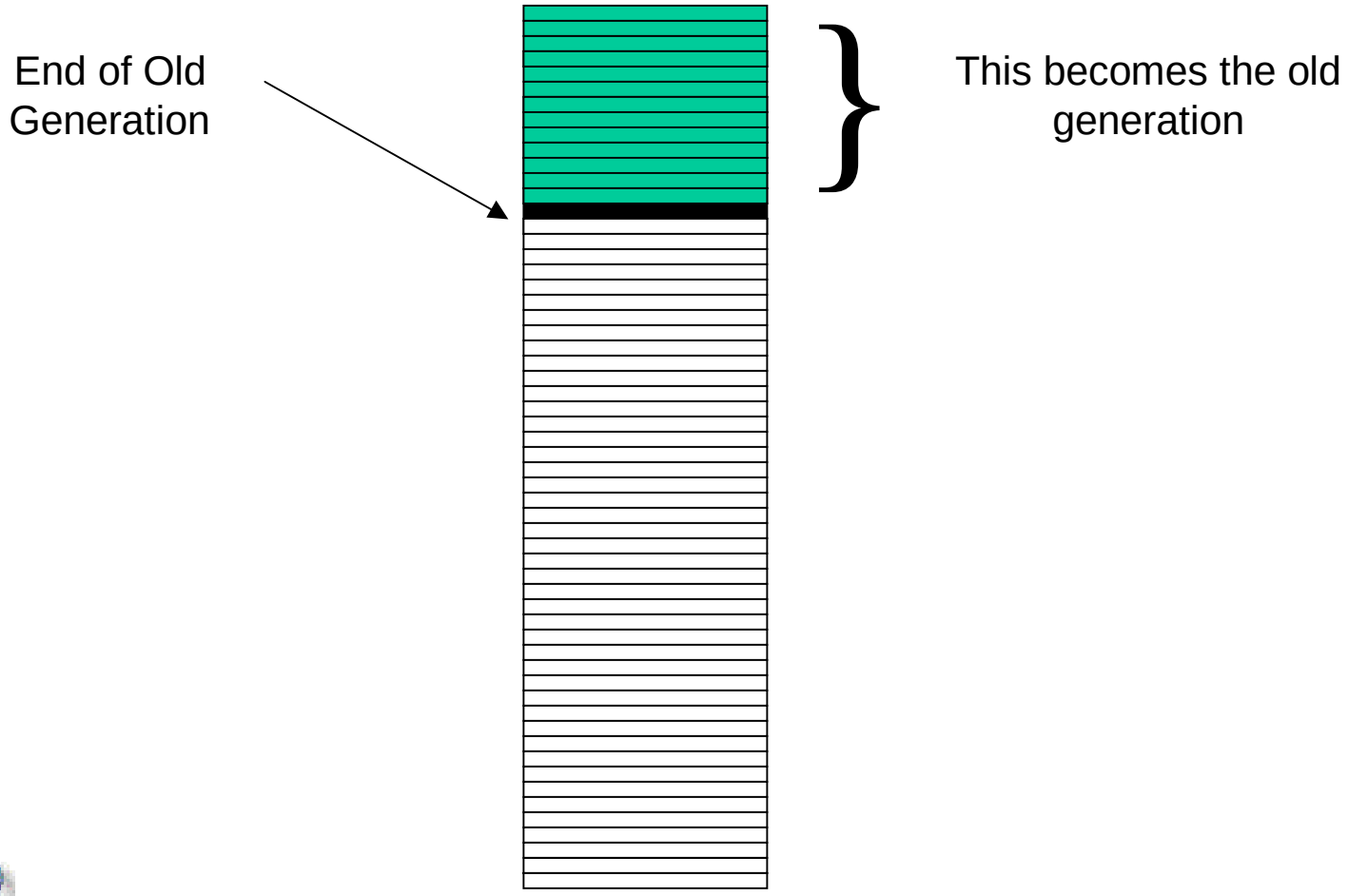


GC Required

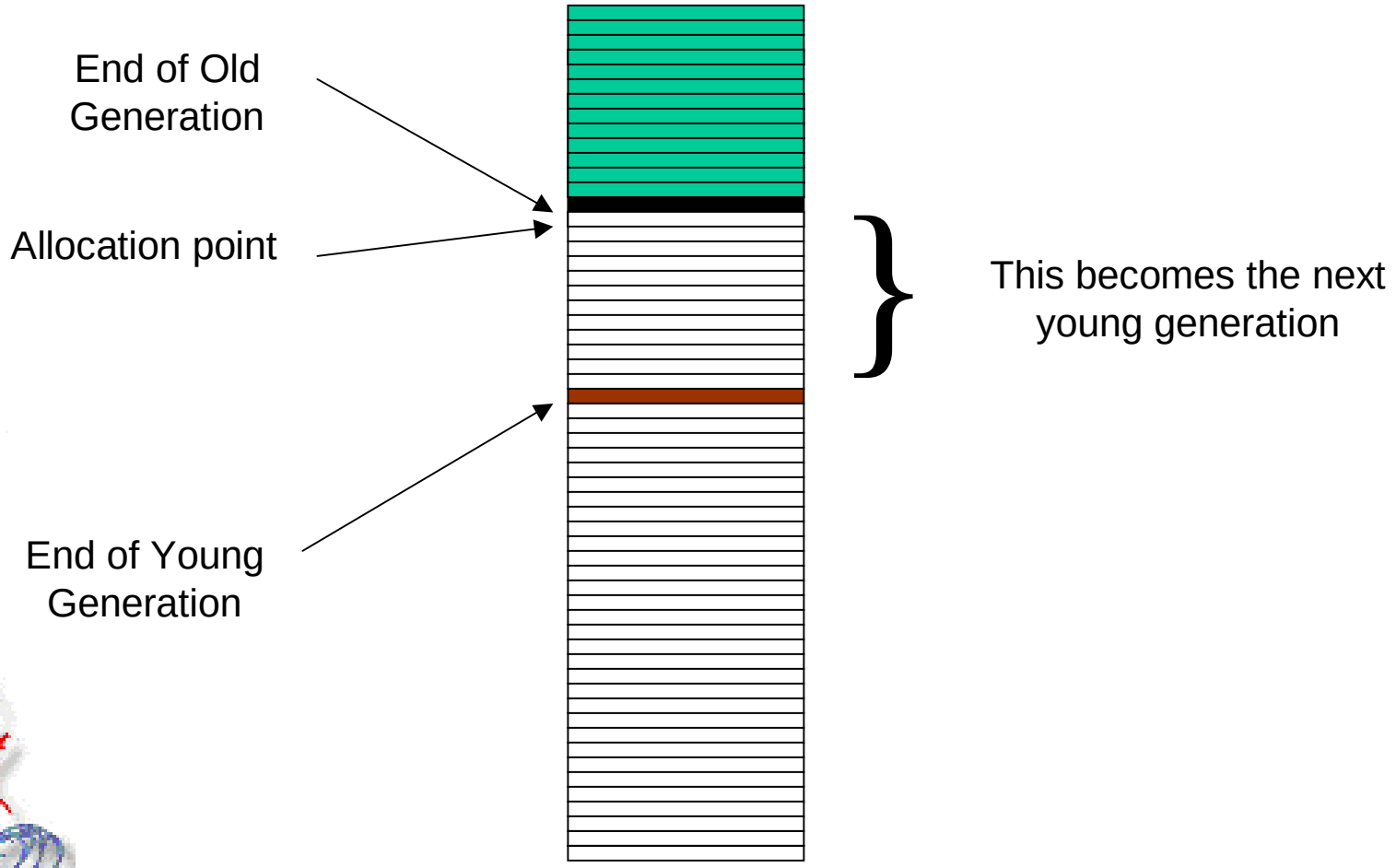
RAM architecture



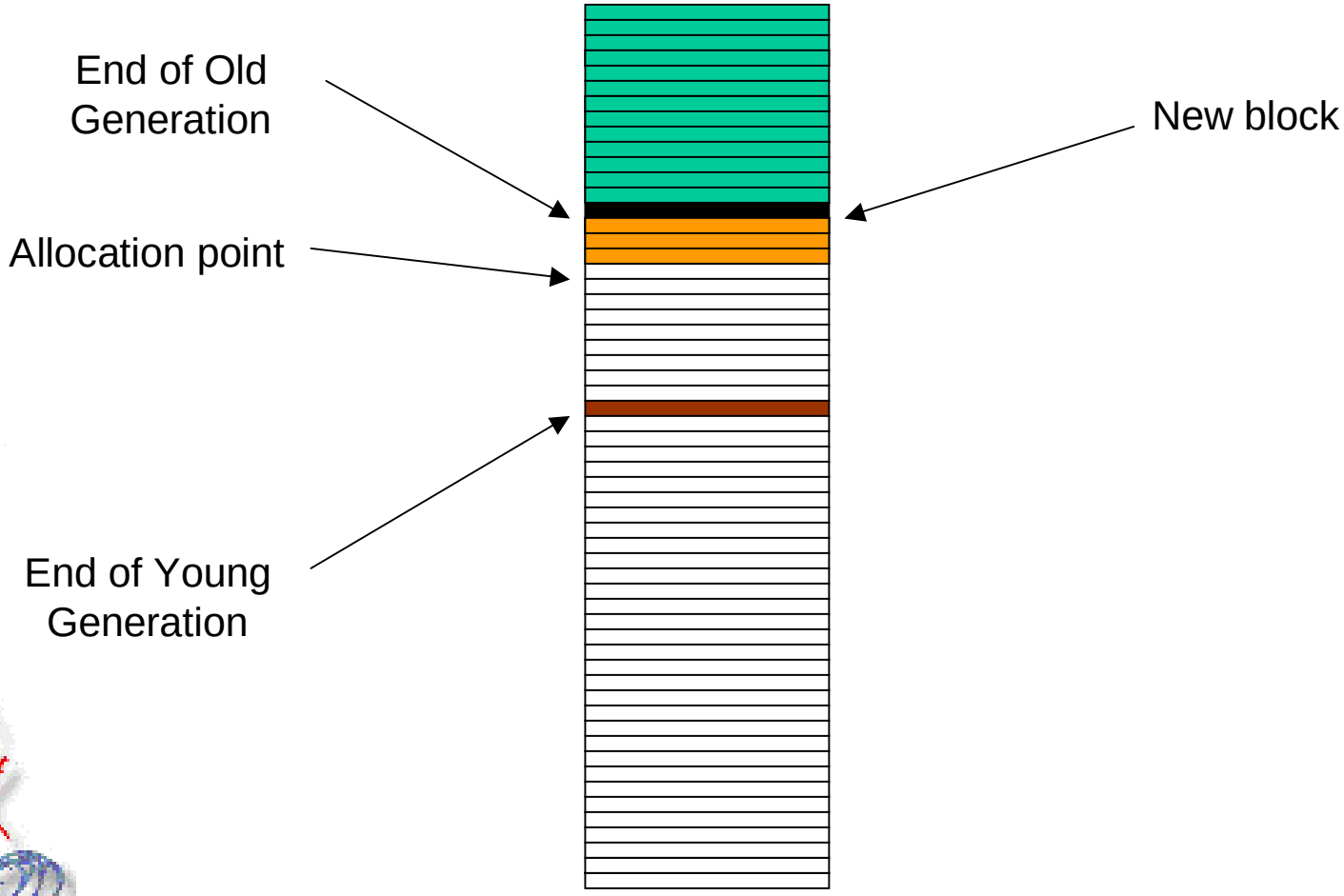
RAM architecture



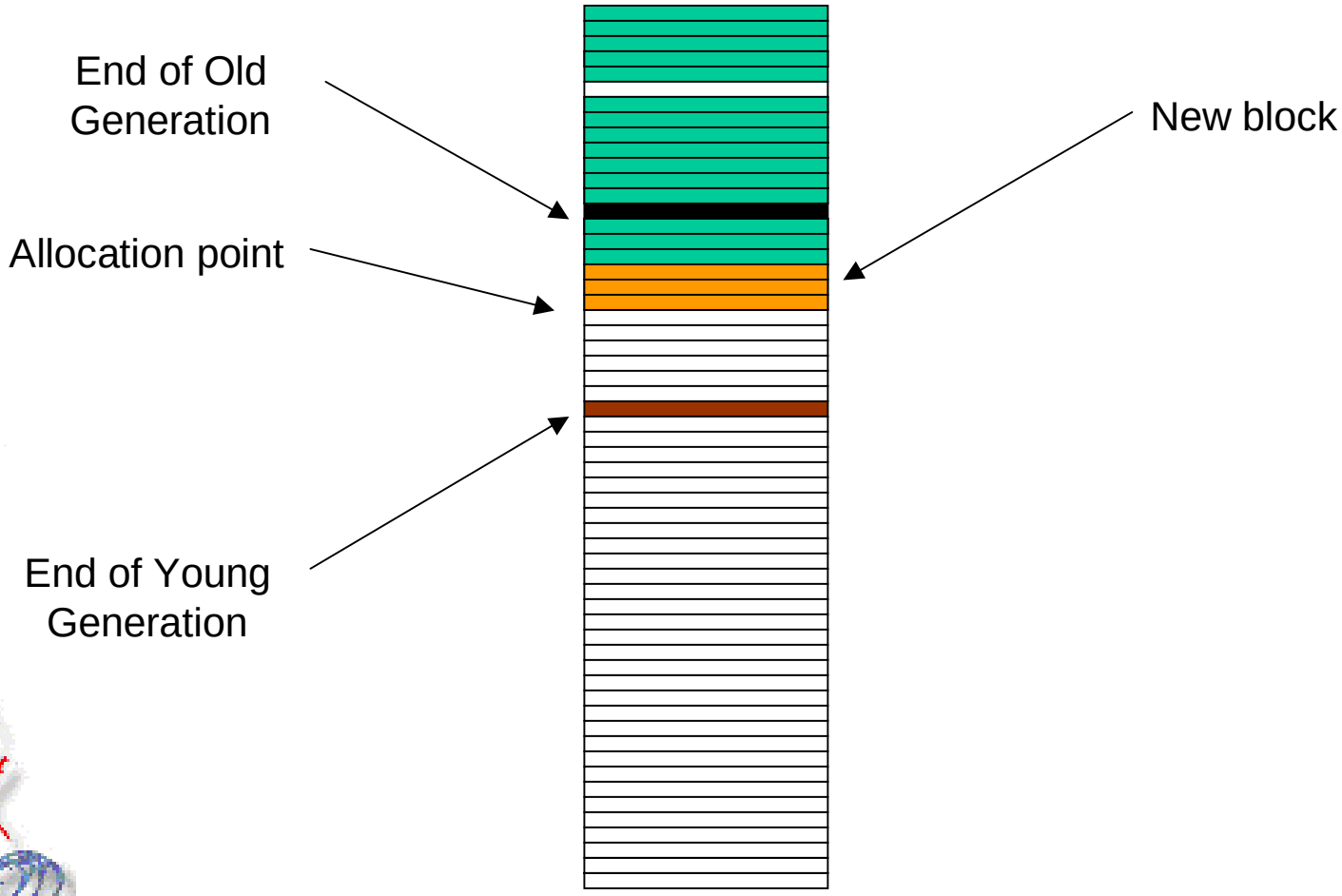
RAM architecture



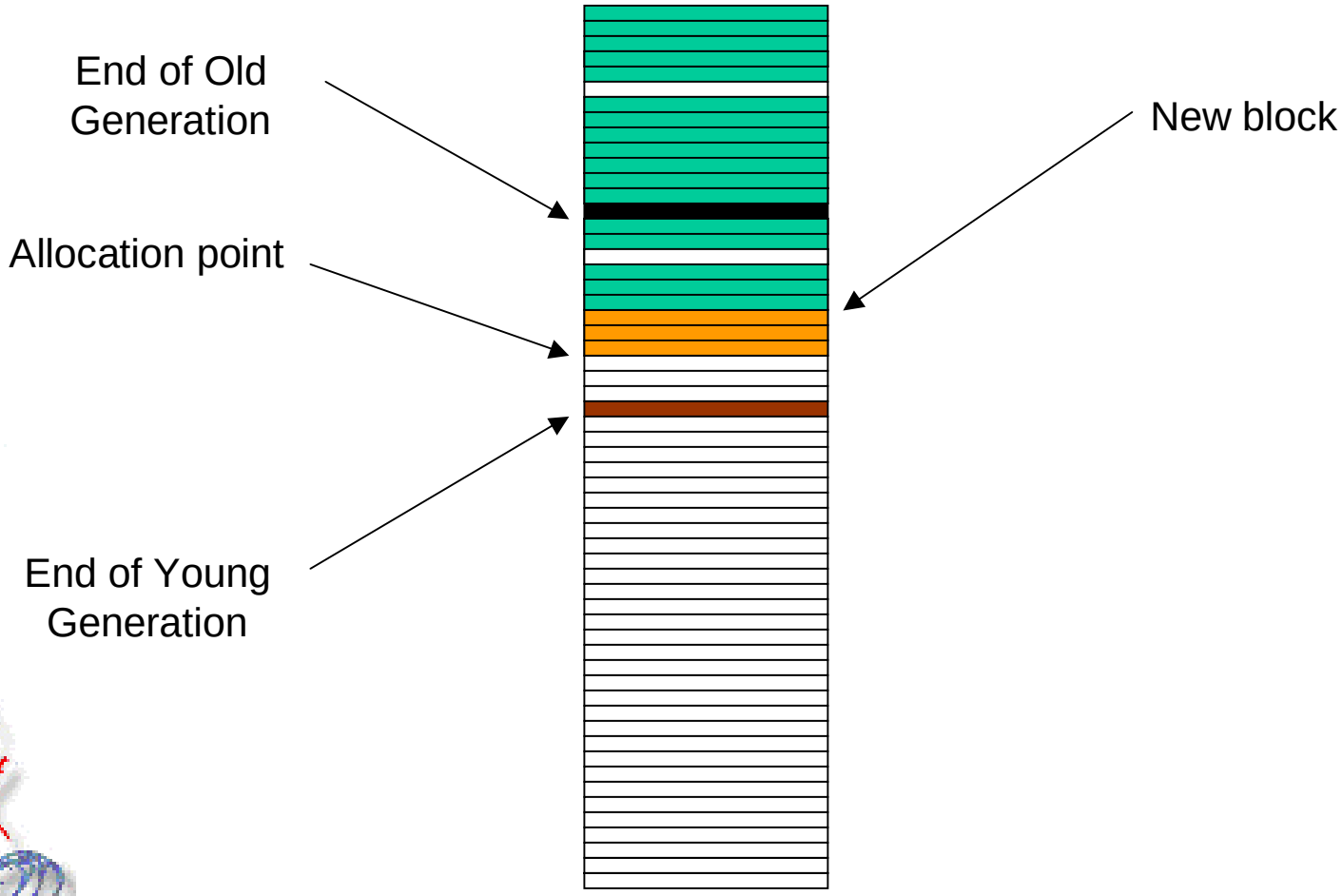
RAM architecture



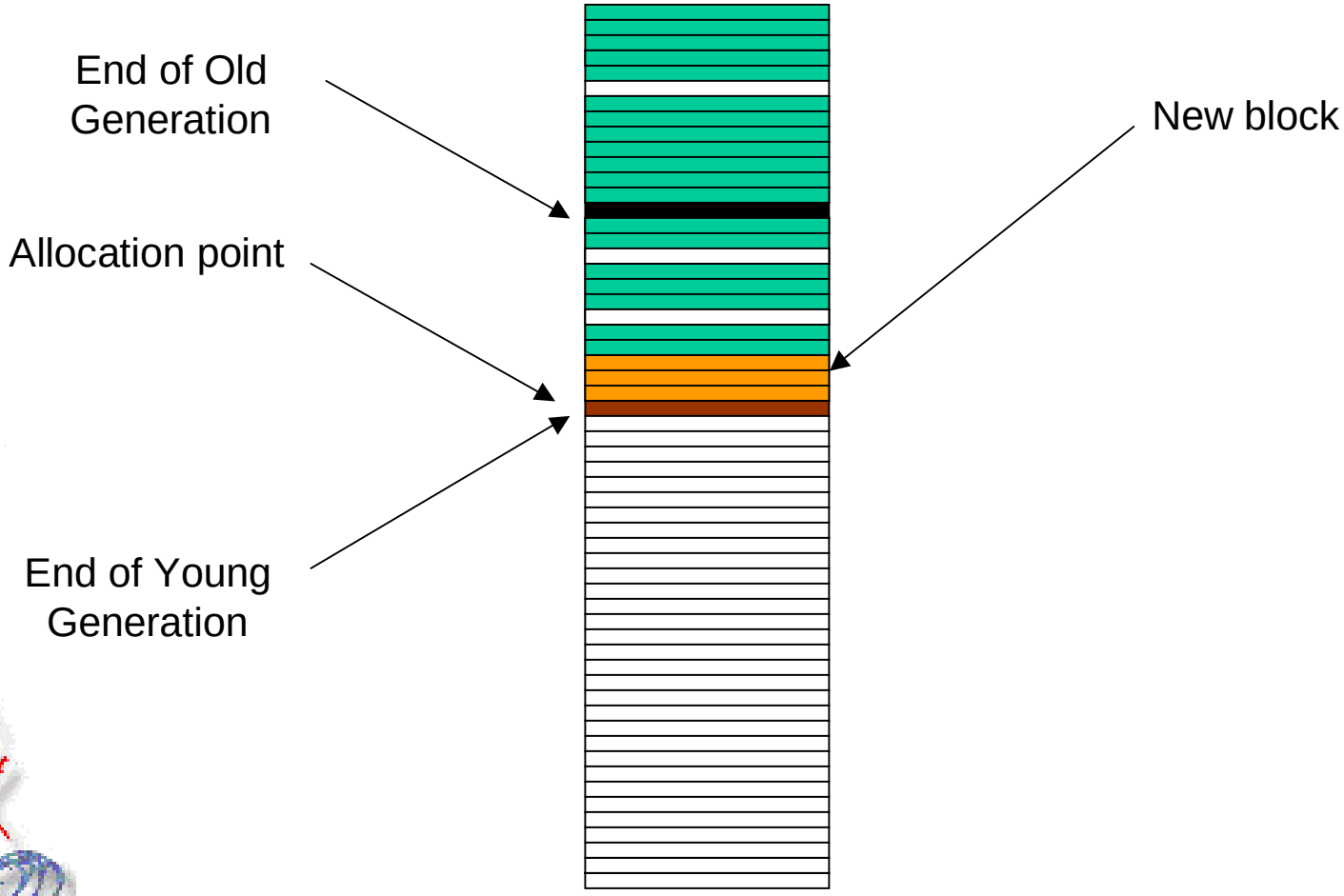
RAM architecture



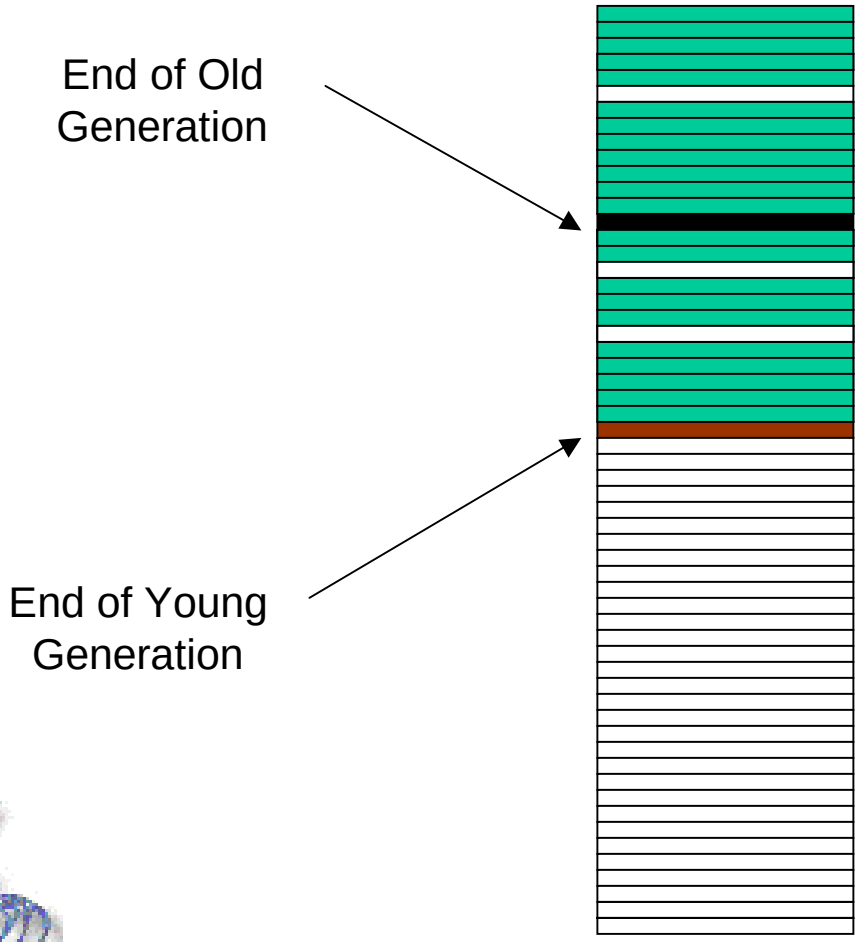
RAM architecture



RAM architecture

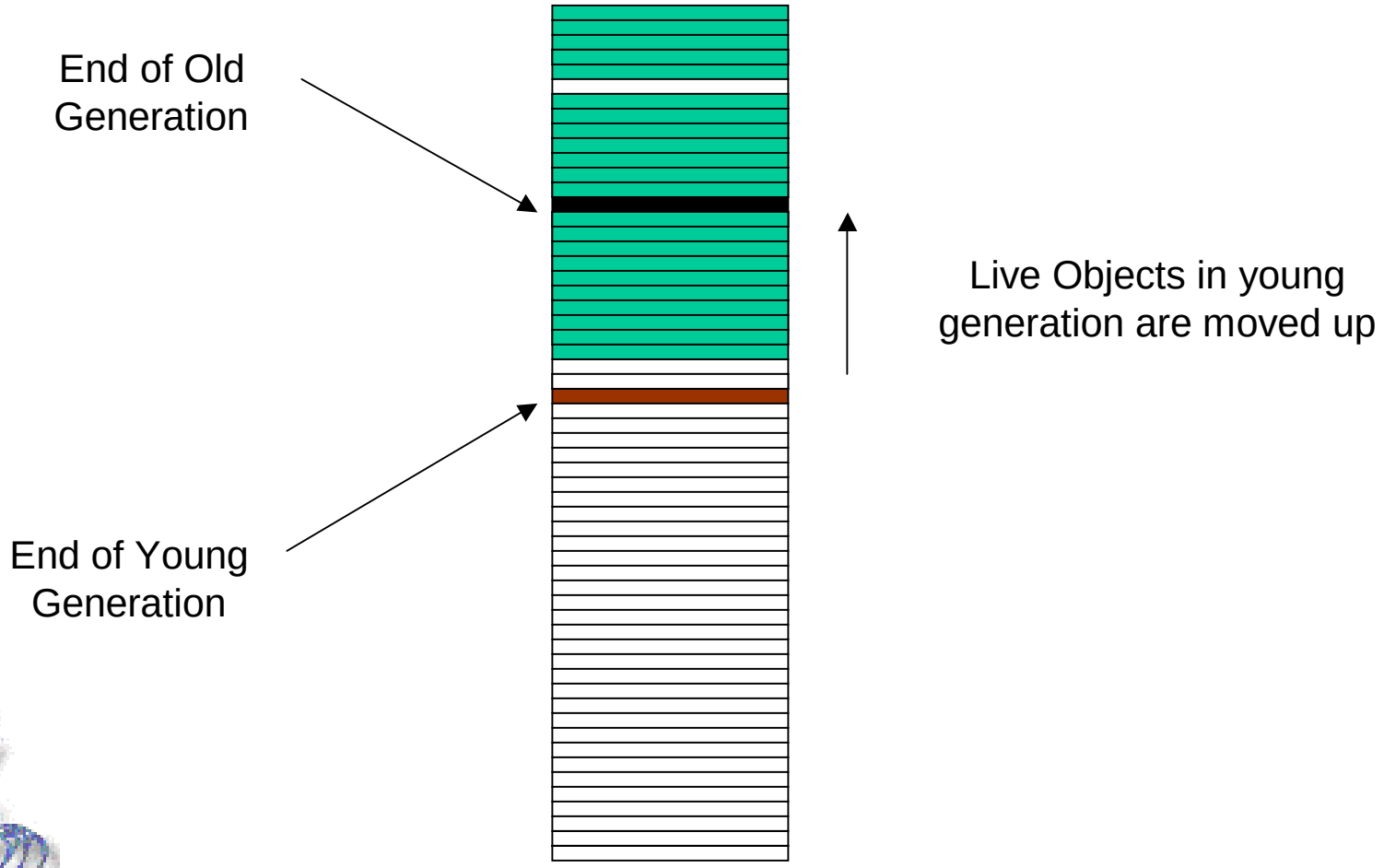


RAM architecture

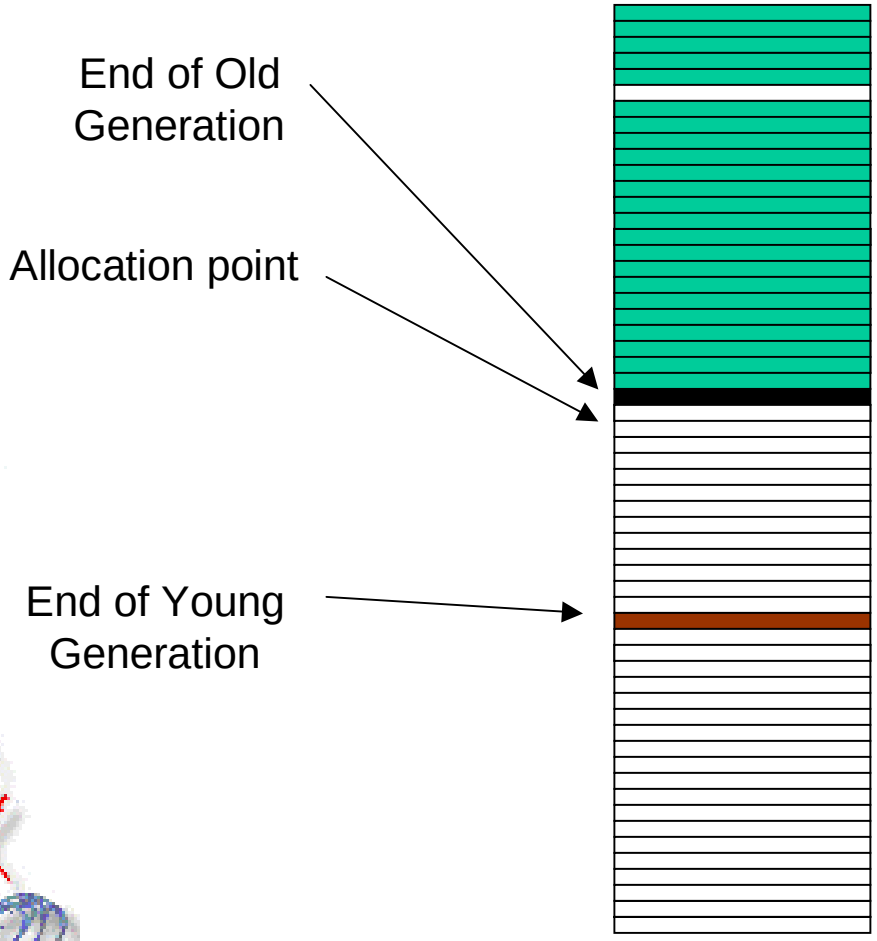


GC required

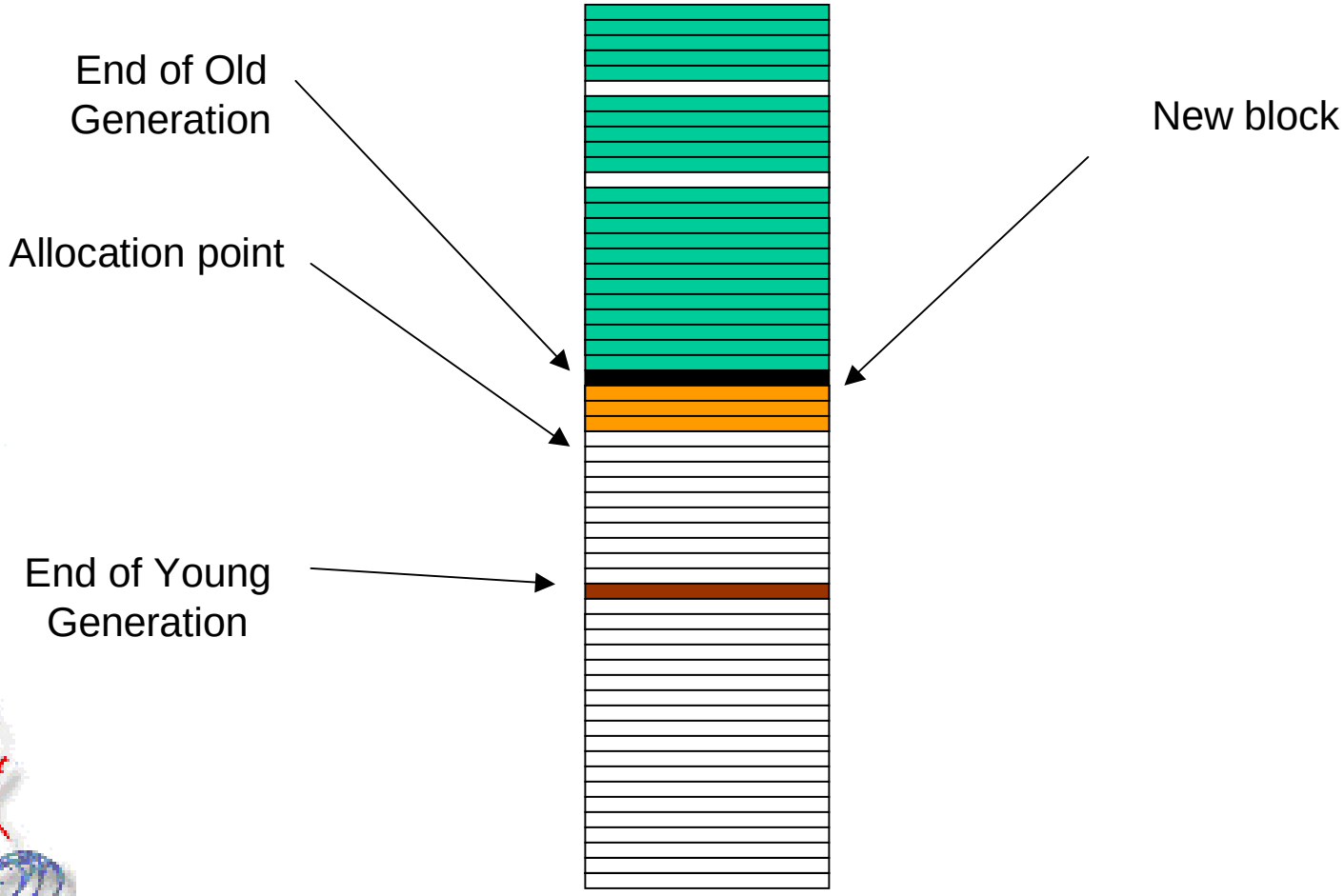
RAM architecture



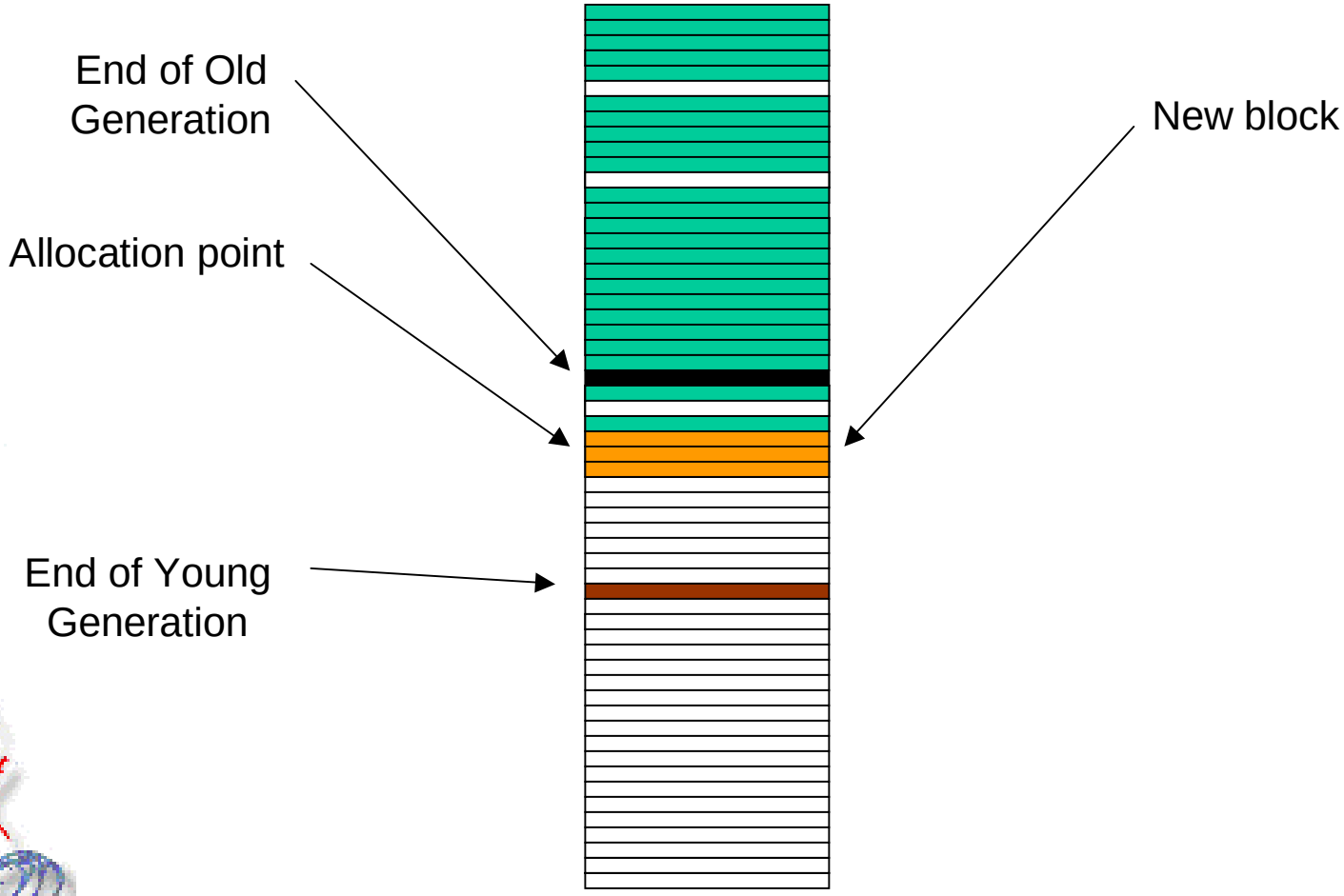
RAM architecture



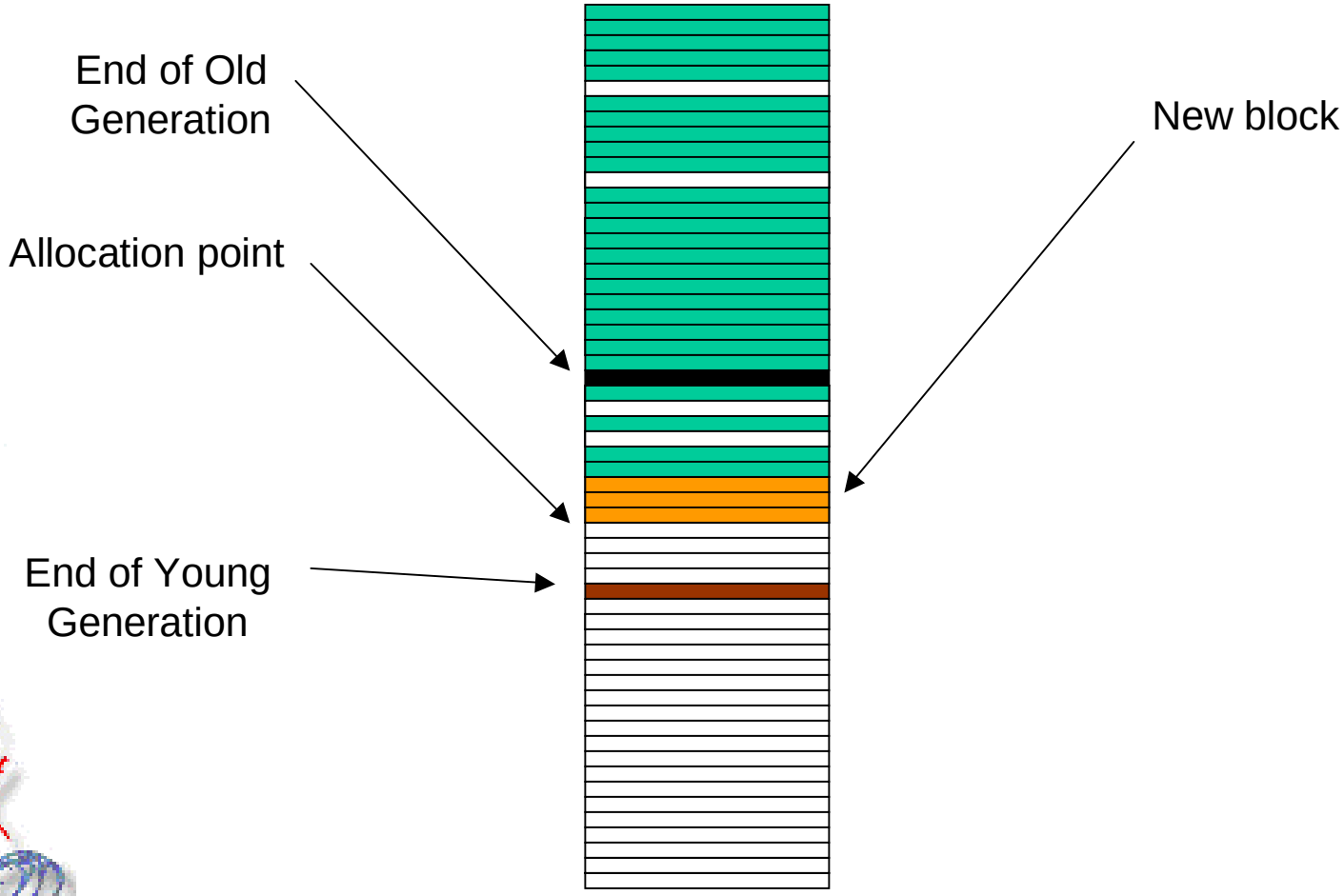
RAM architecture



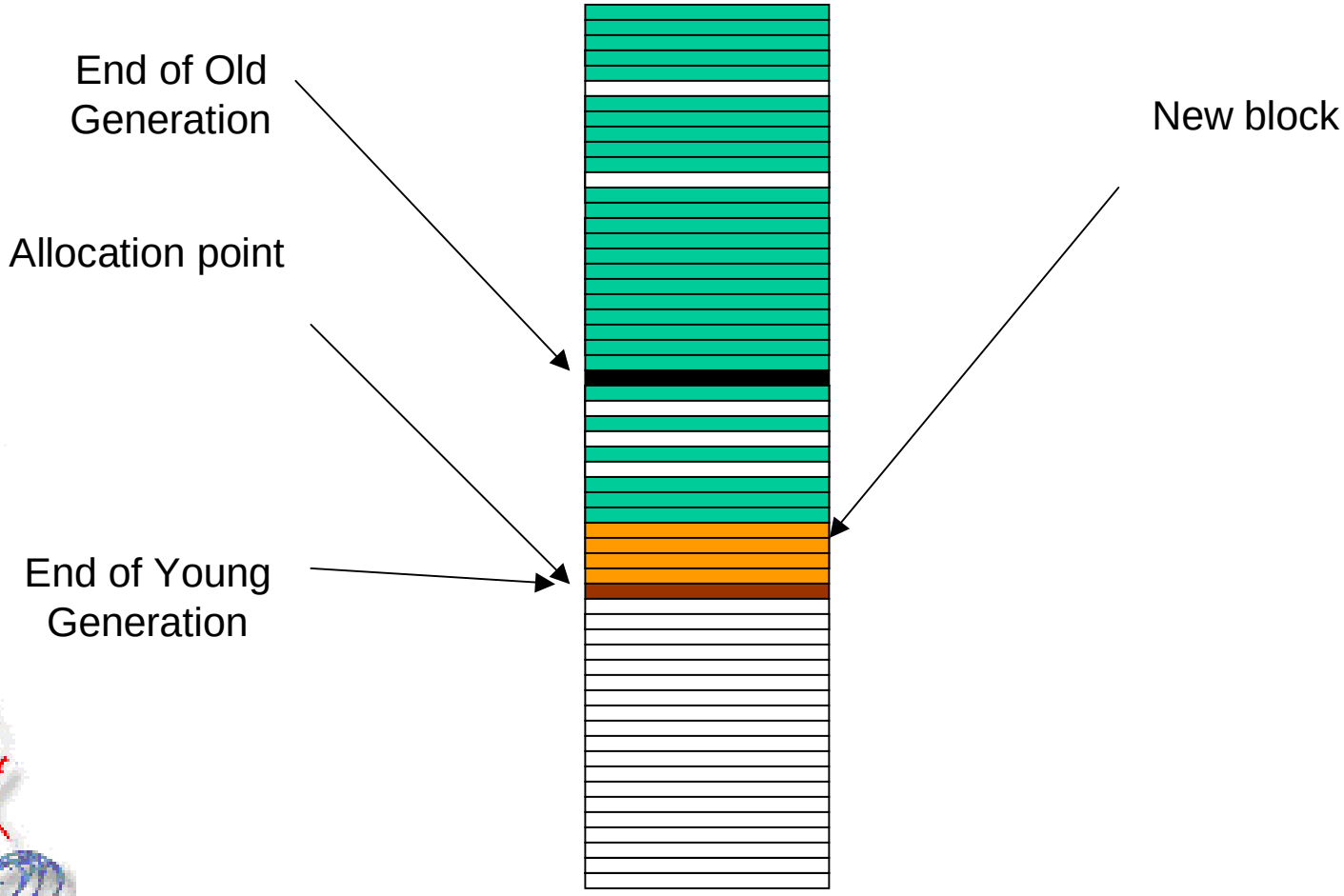
RAM architecture



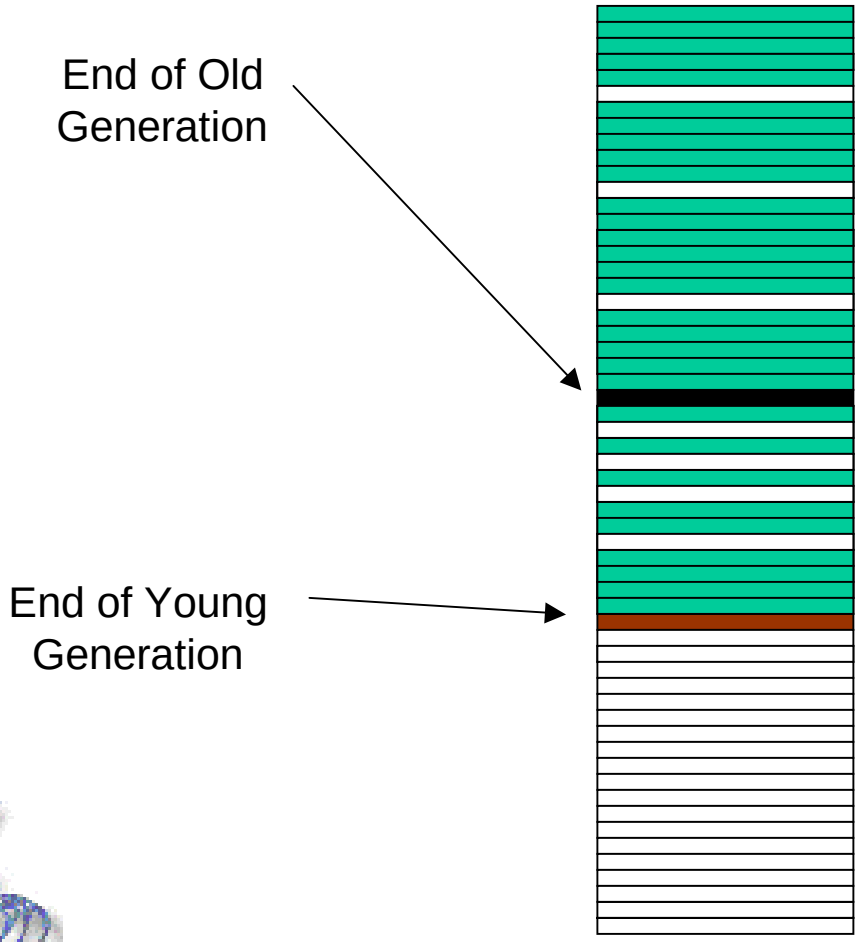
RAM architecture



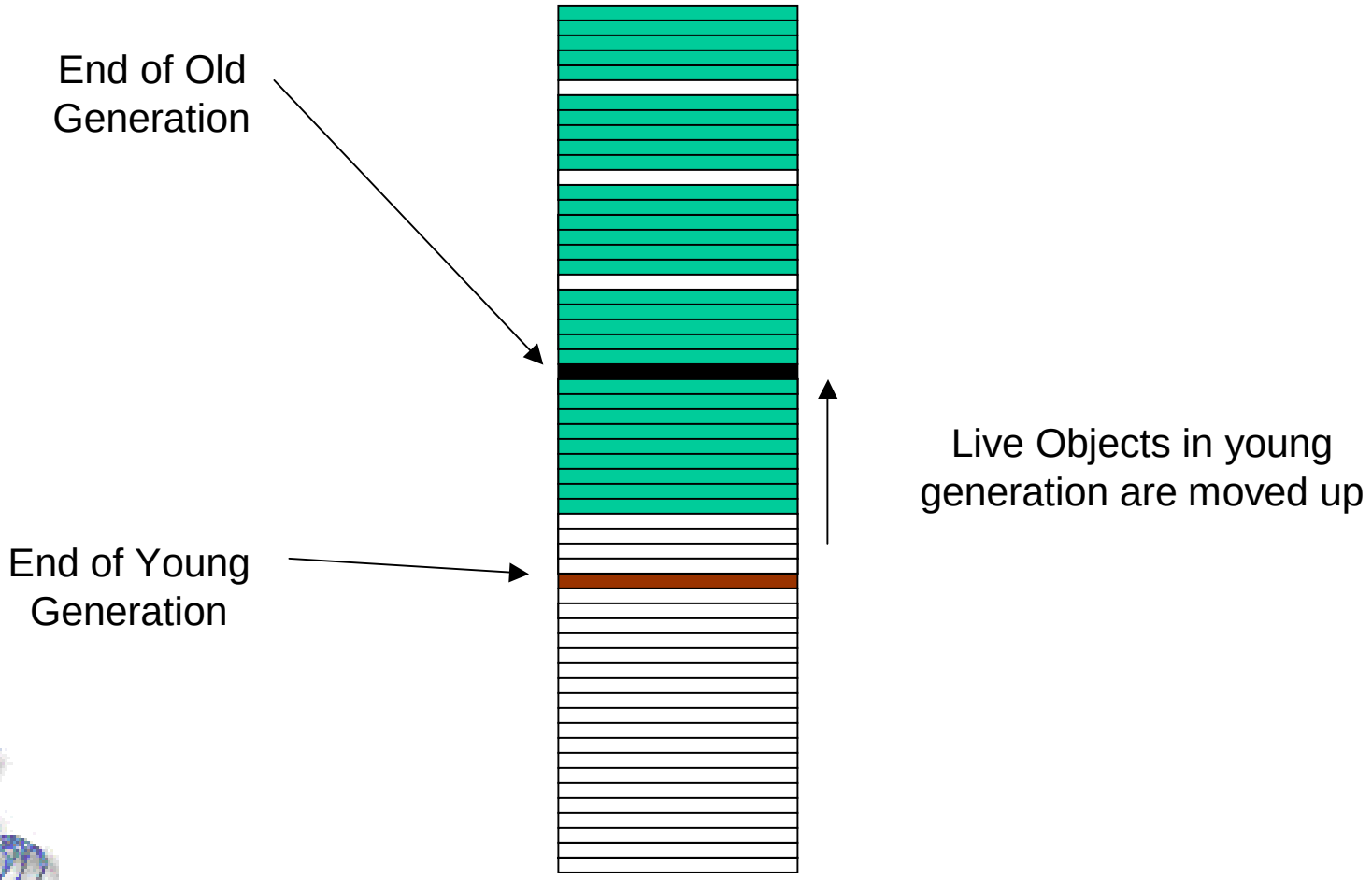
RAM architecture



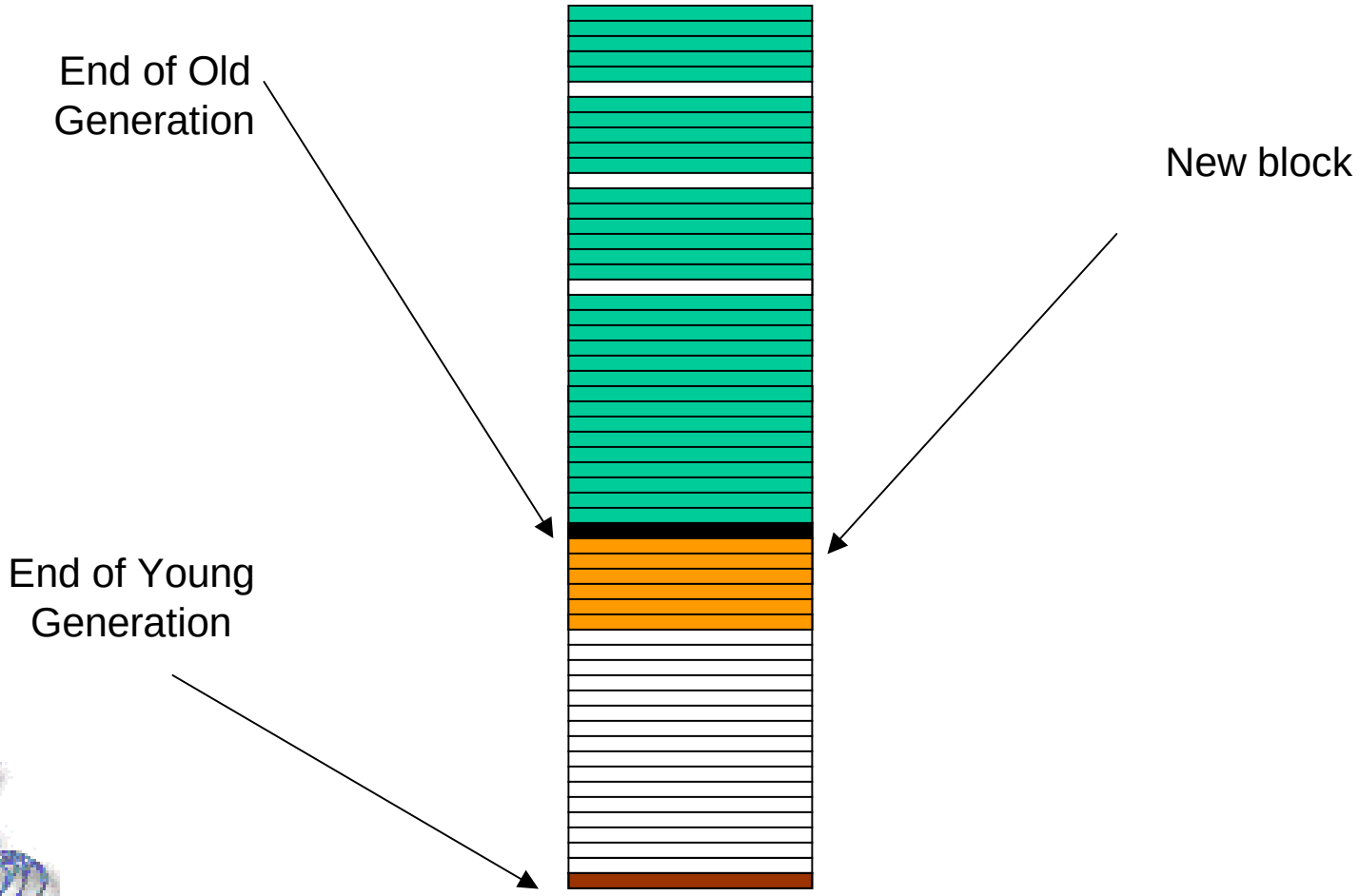
RAM architecture



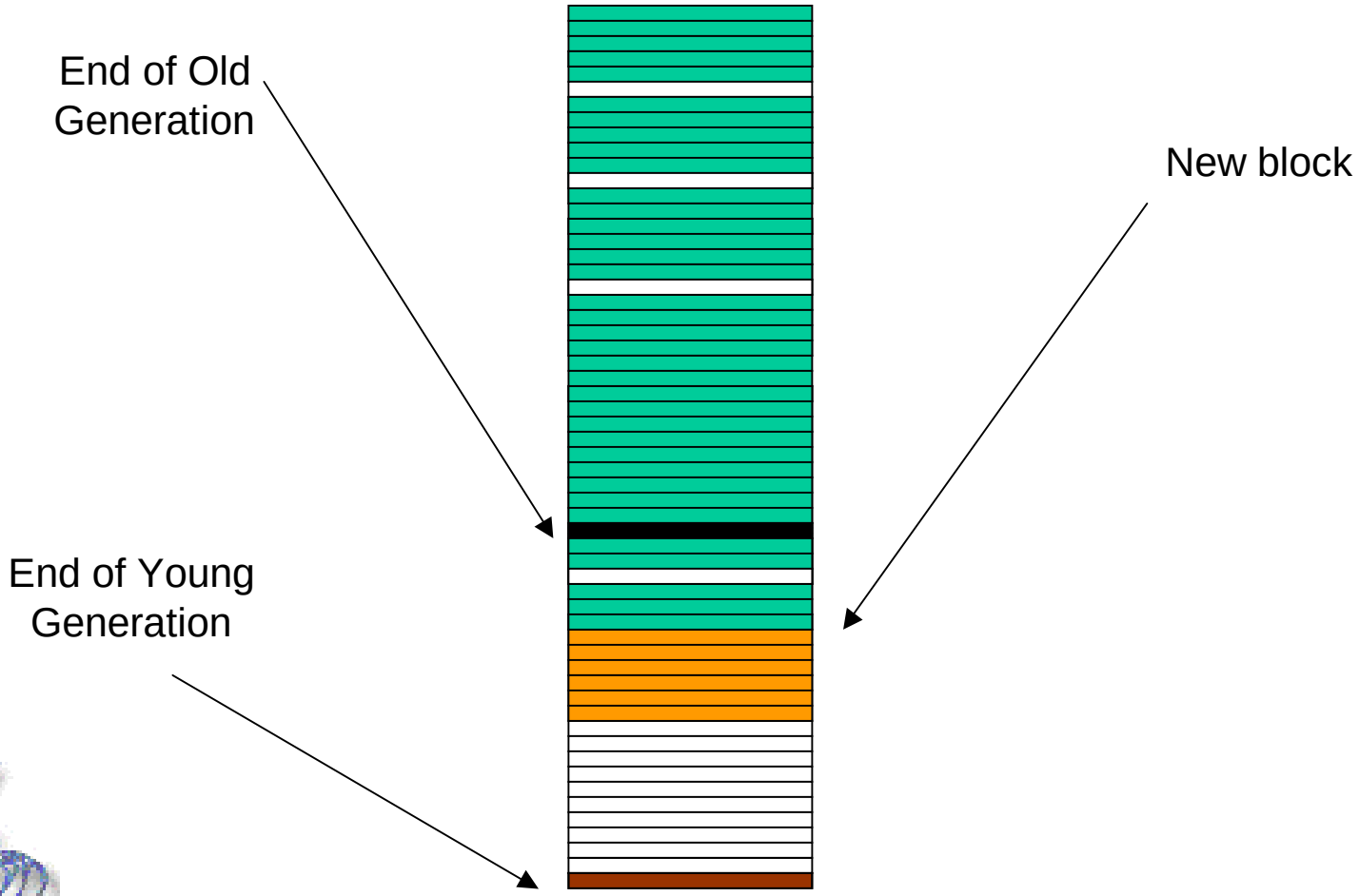
RAM architecture



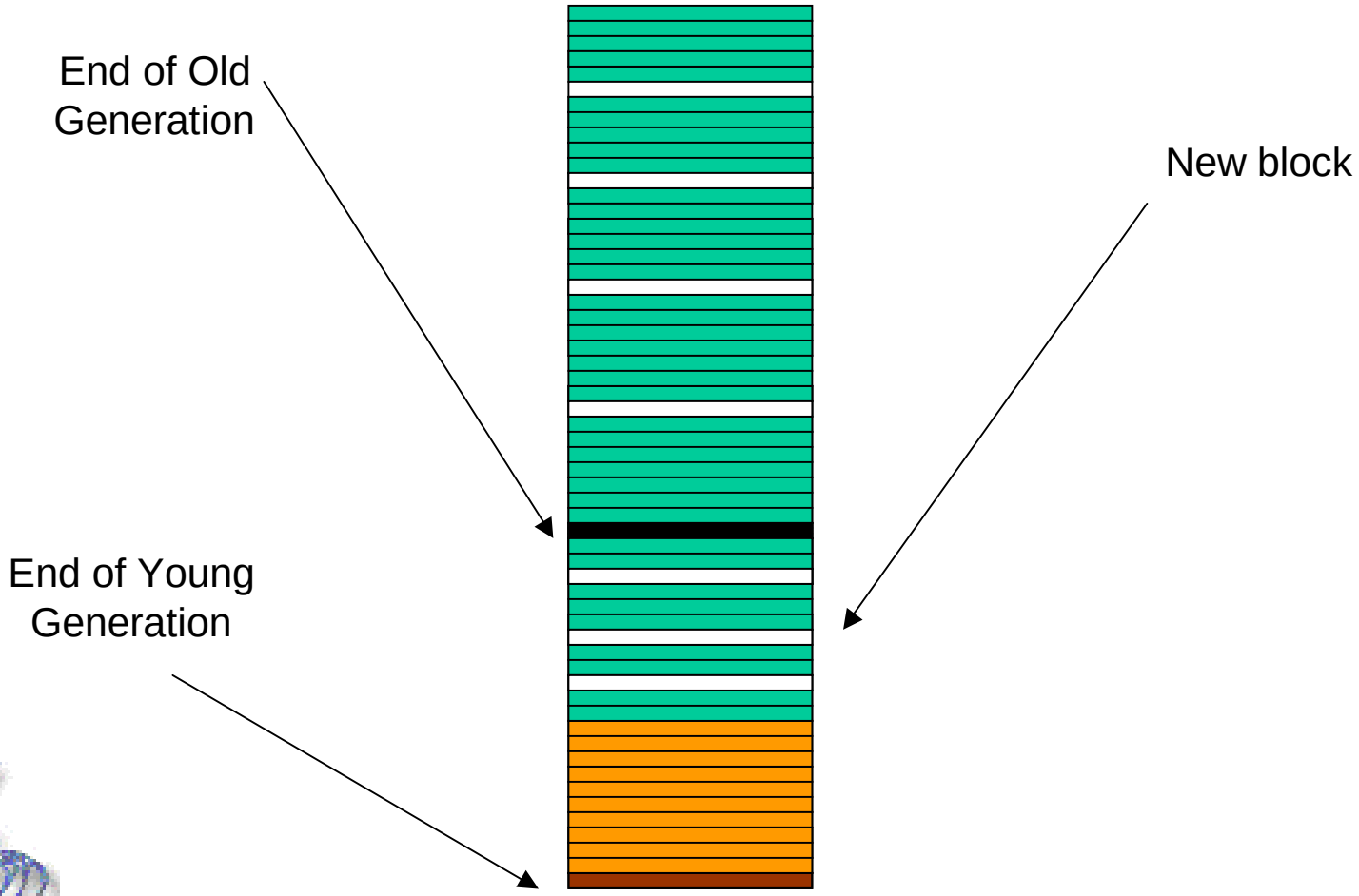
RAM architecture



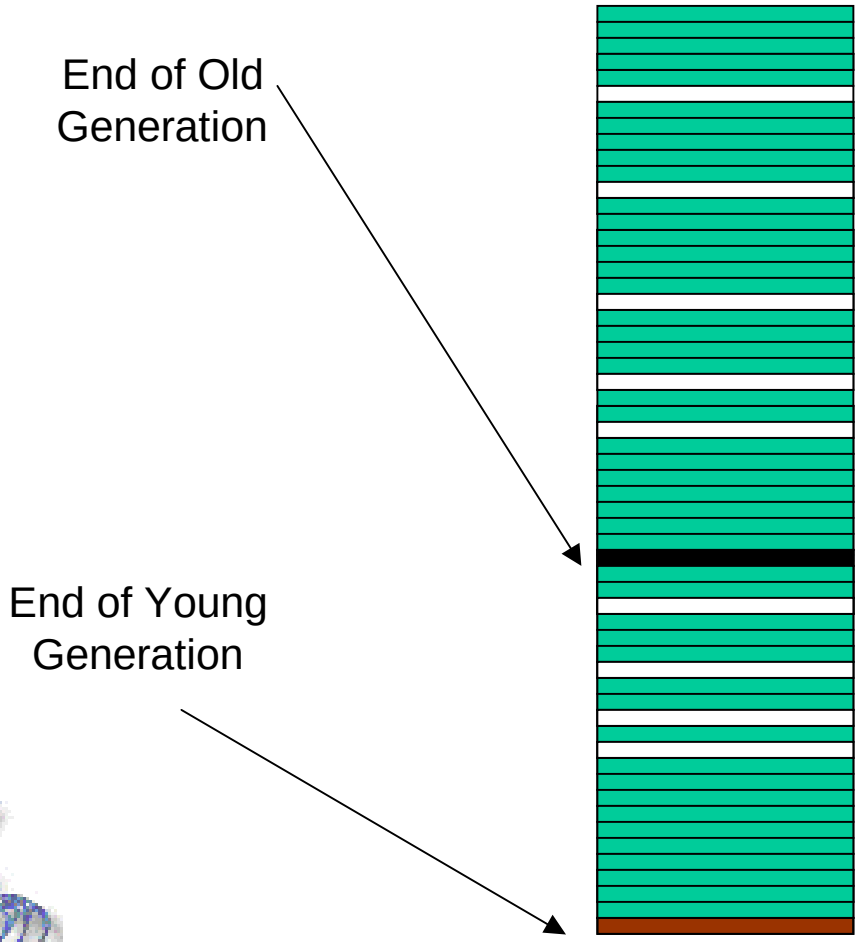
RAM architecture



RAM architecture

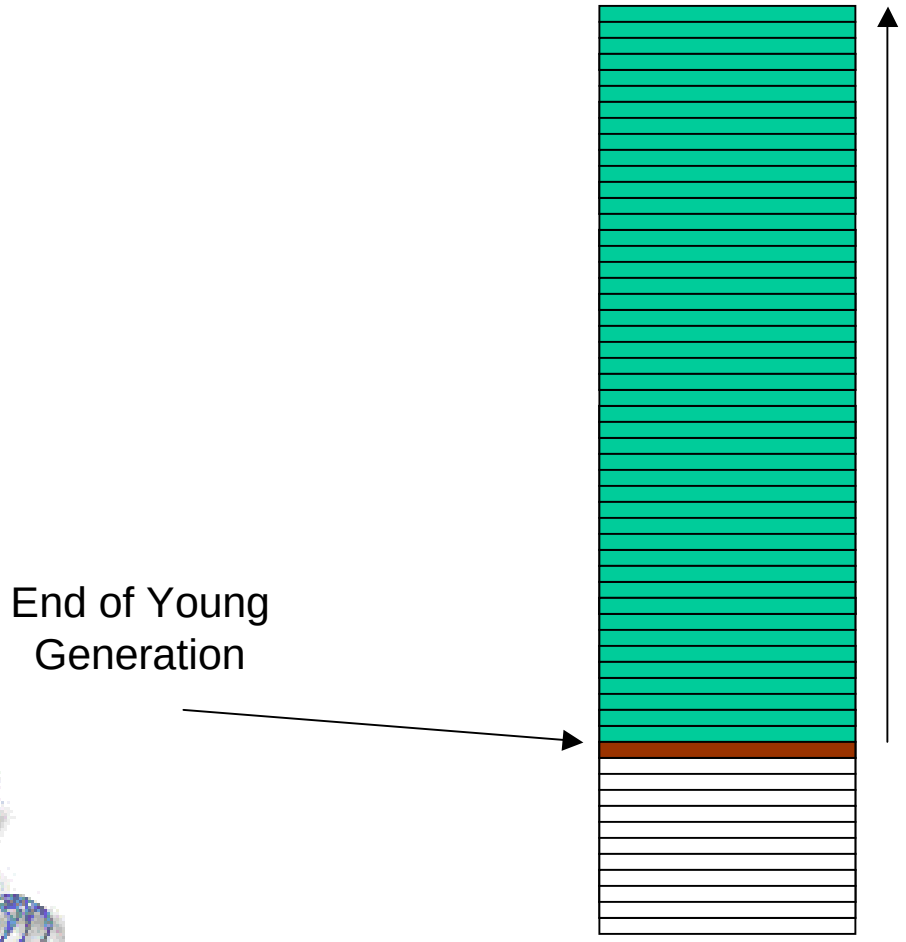


RAM architecture



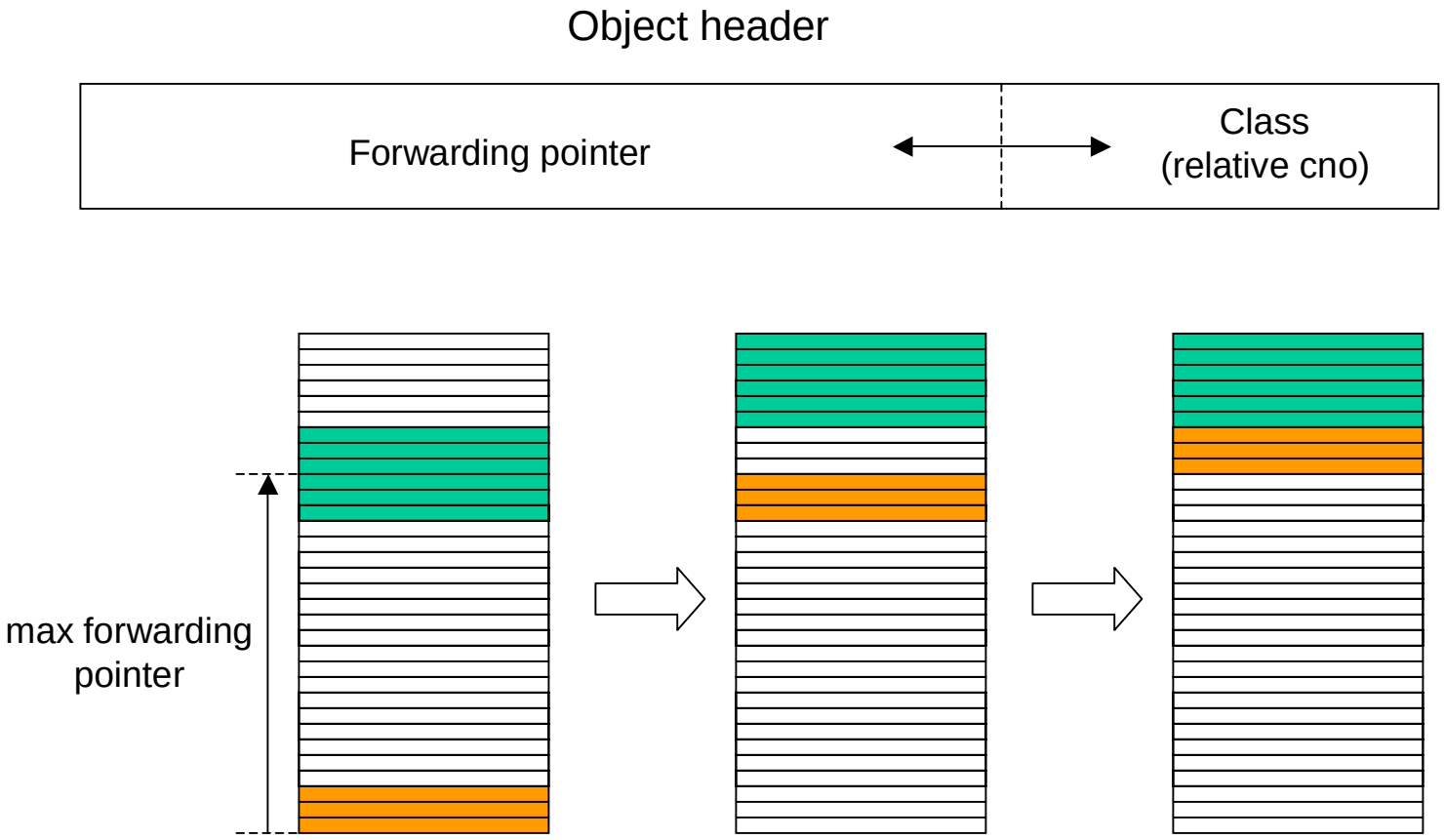
Full GC Required

RAM architecture



Live Objects in heap
generation are moved up

Collecting large heaps

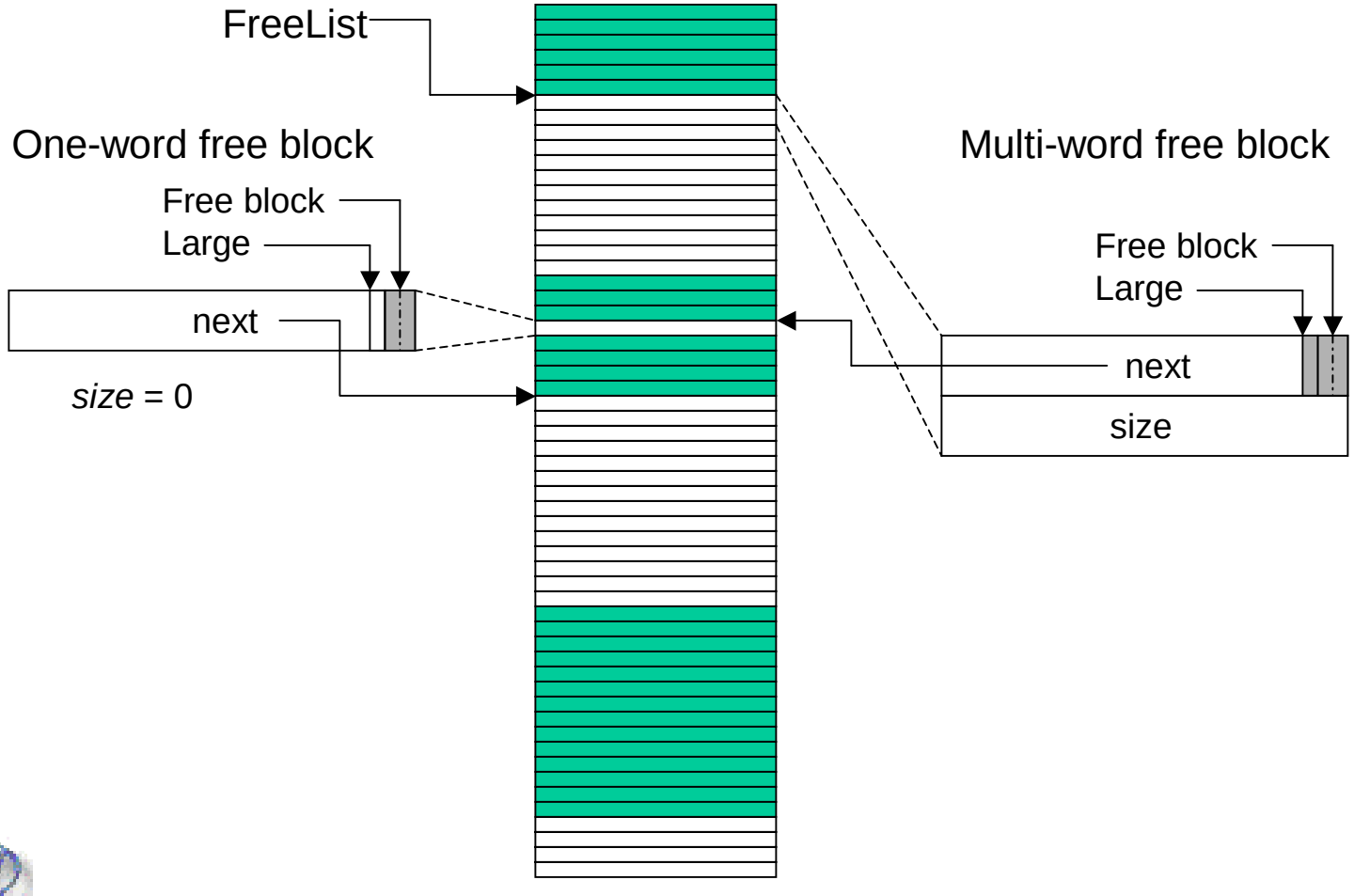


EEPROM

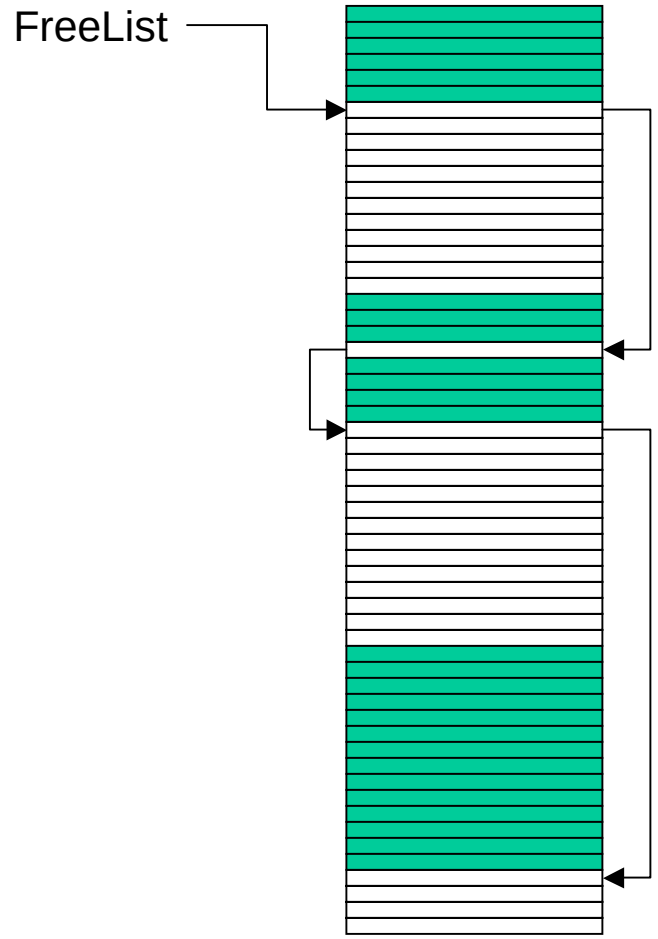
Design goals

- Object memory (i.e. not a file-system)
- Minimize EEPROM writes:
  - Use mark-sweep, *non-compacting* collector
  - Use free list for allocation
  - Doesn't automatically zero allocated blocks

EEPROM allocation

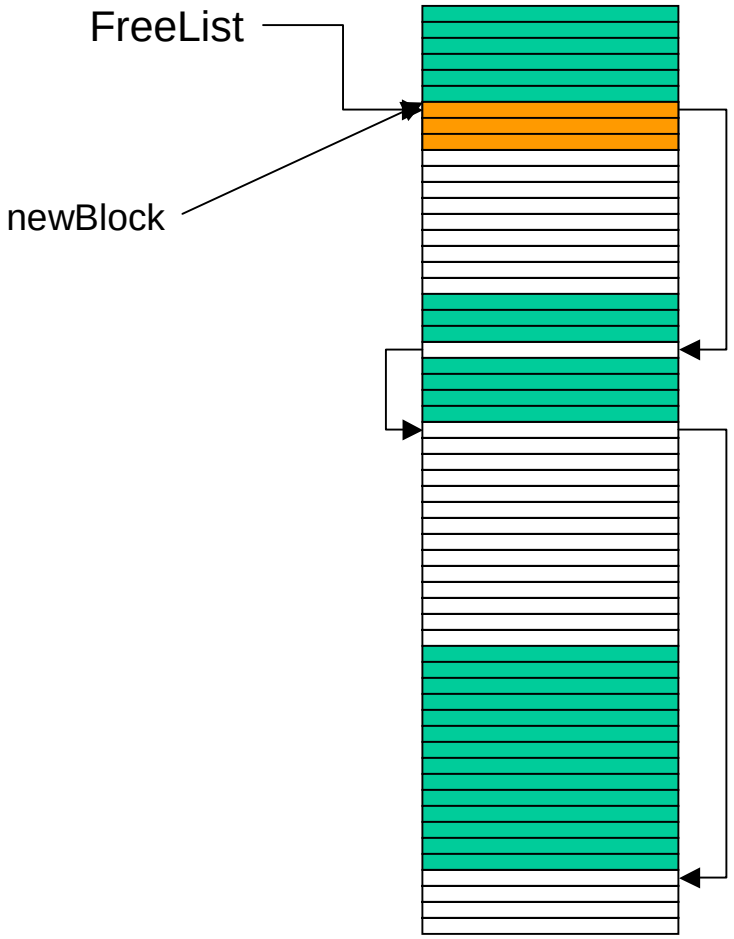


EEPROM allocation



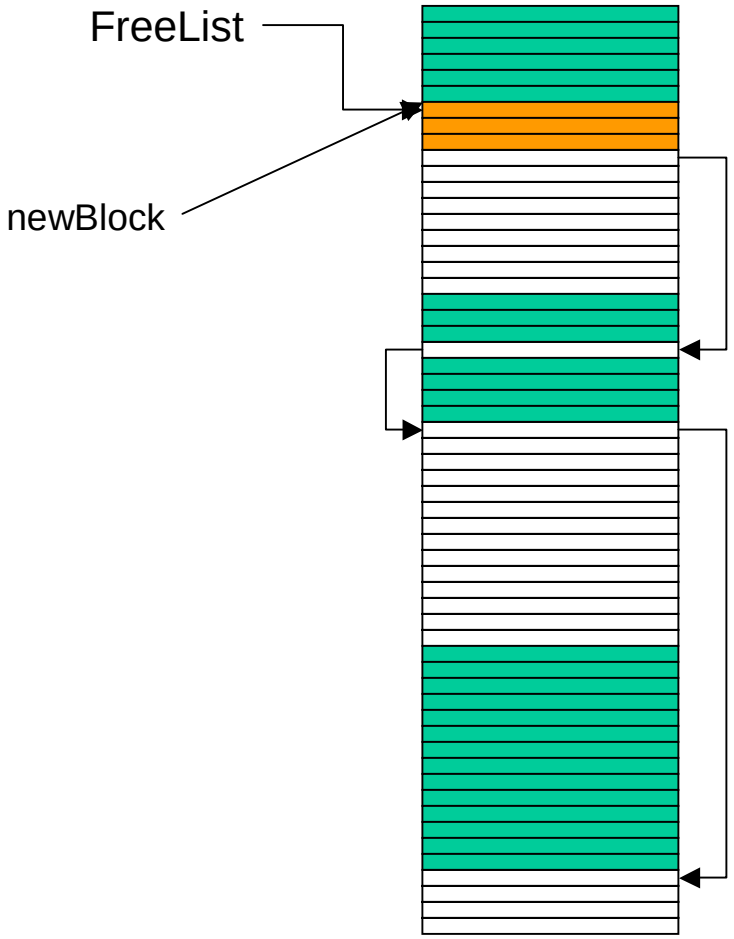
1. Allocate 12 bytes

EEPROM allocation



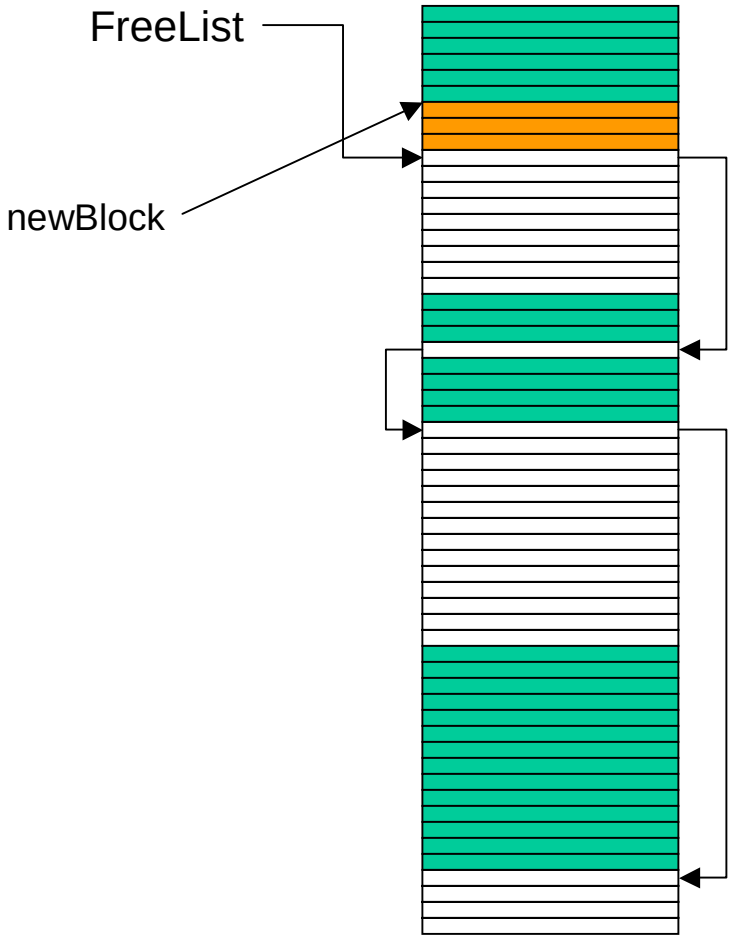
1. Allocate 12 bytes
2. Find chunk >= 12 bytes

EEPROM allocation



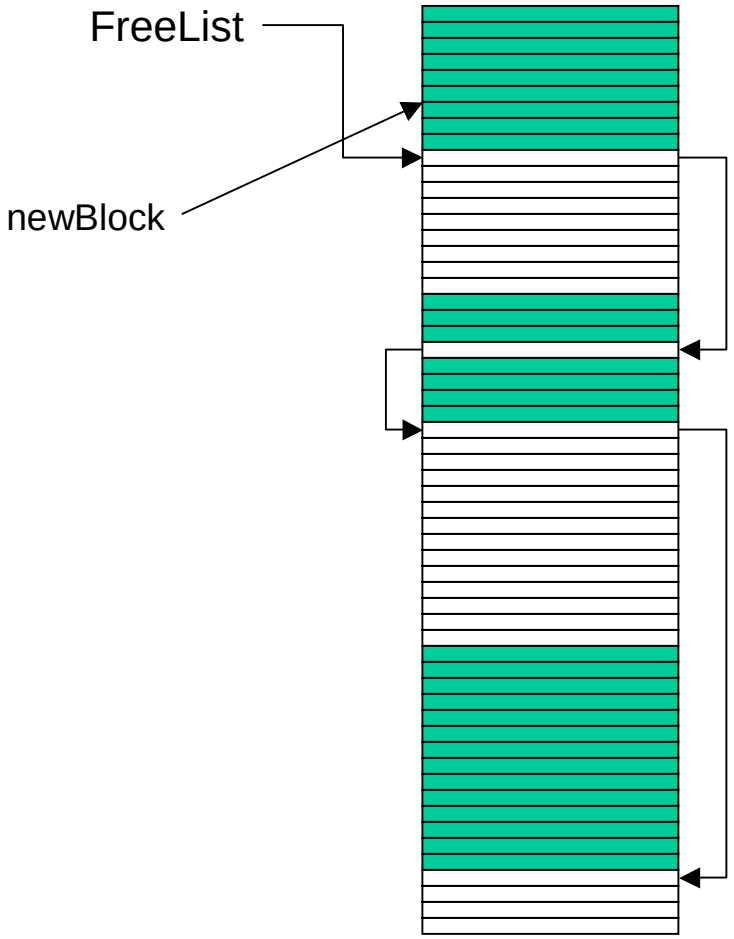
1. Allocate 12 bytes
2. Find chunk >= 12 bytes
3. Split chunk

EEPROM allocation



1. Allocate 12 bytes
2. Find chunk >= 12 bytes
3. Split chunk
4. Fix up free list

EEPROM allocation



1. Allocate 12 bytes
2. Find block >= 12 bytes
3. Split block
4. Fix up free list
5. Return new block

EEPROM collector

- Primary goal is to minimize EEPROM writes.
- Uses 2 RAM data structures:
  1. Bit vector for marking → requires 3% EEPROM length
  2. Small stack to limit recursion (~ 20 bytes)

- Persistent GC is only run at well known points when RAM is always available
- Collector auto restarts if interrupted by power tear (via invalidating free list pointer)

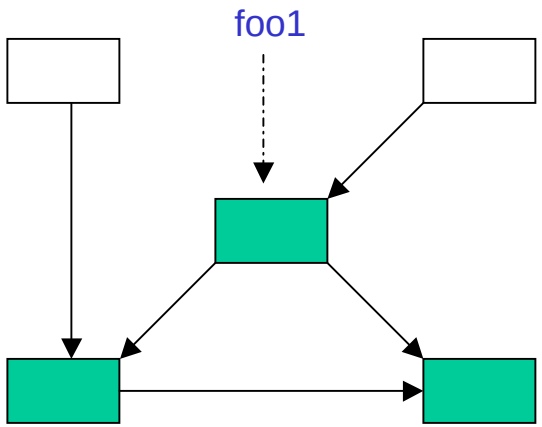
Copying to EEPROM

Copying to EEPROM

```
Foo foo2 = (Foo)PersistentMemory.makePersistentCopy(foo1);
```

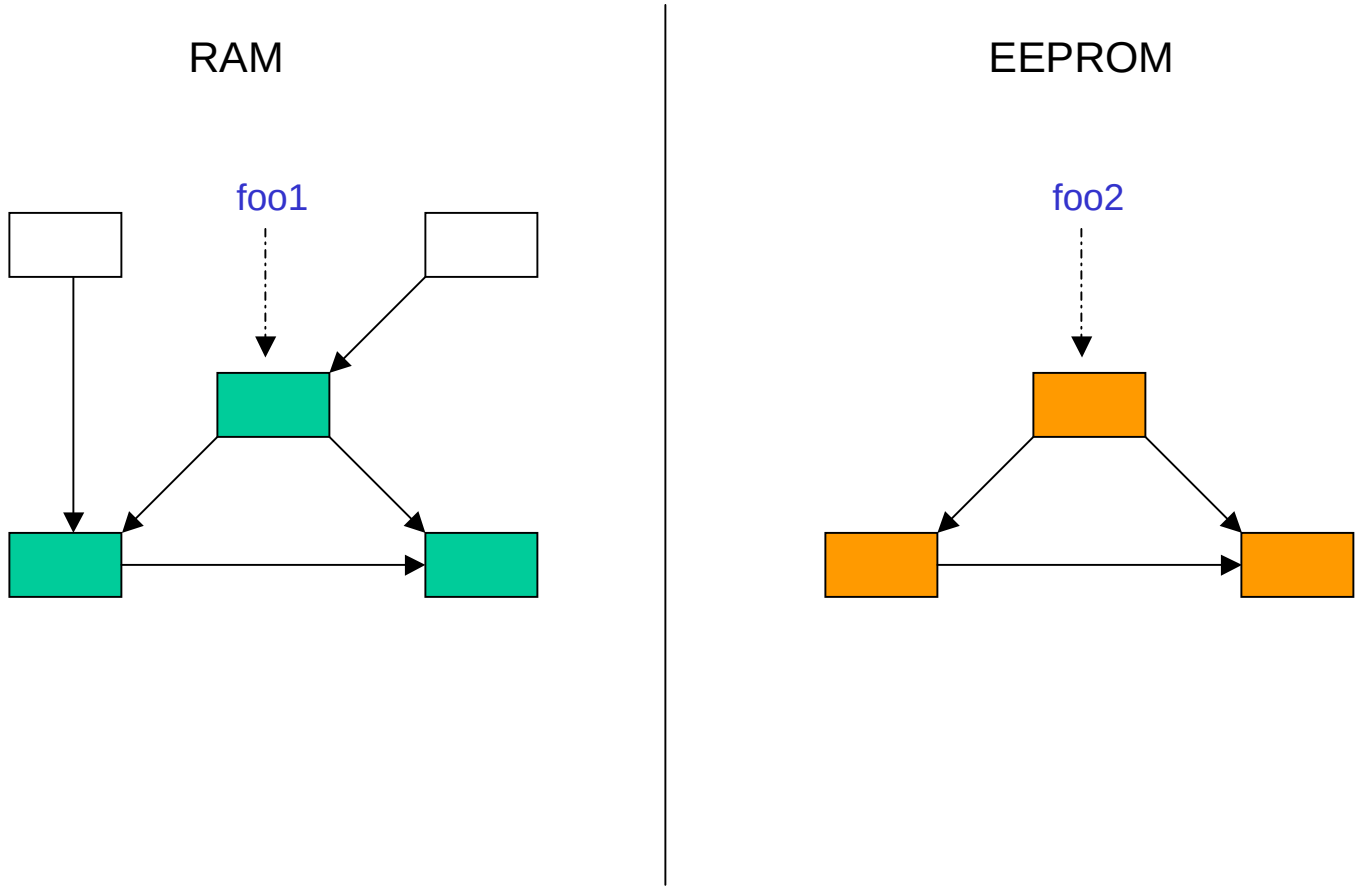
RAM

EEPROM



Copying to EEPROM

```
Foo foo2 = (Foo)PersistentMemory.makePersistentCopy(foo1);
```



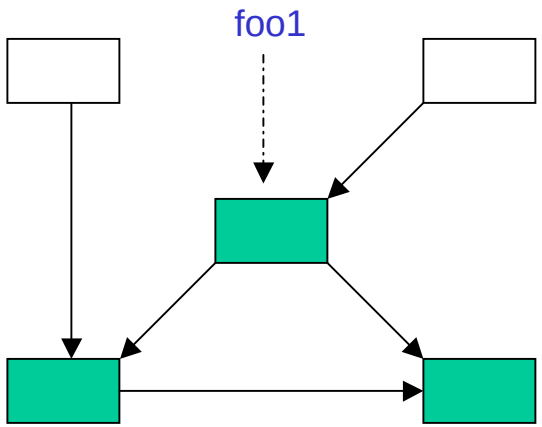
Migrating to EEPROM

Migrating to EEPROM

PersistentMemory.makePersistent(foo1);

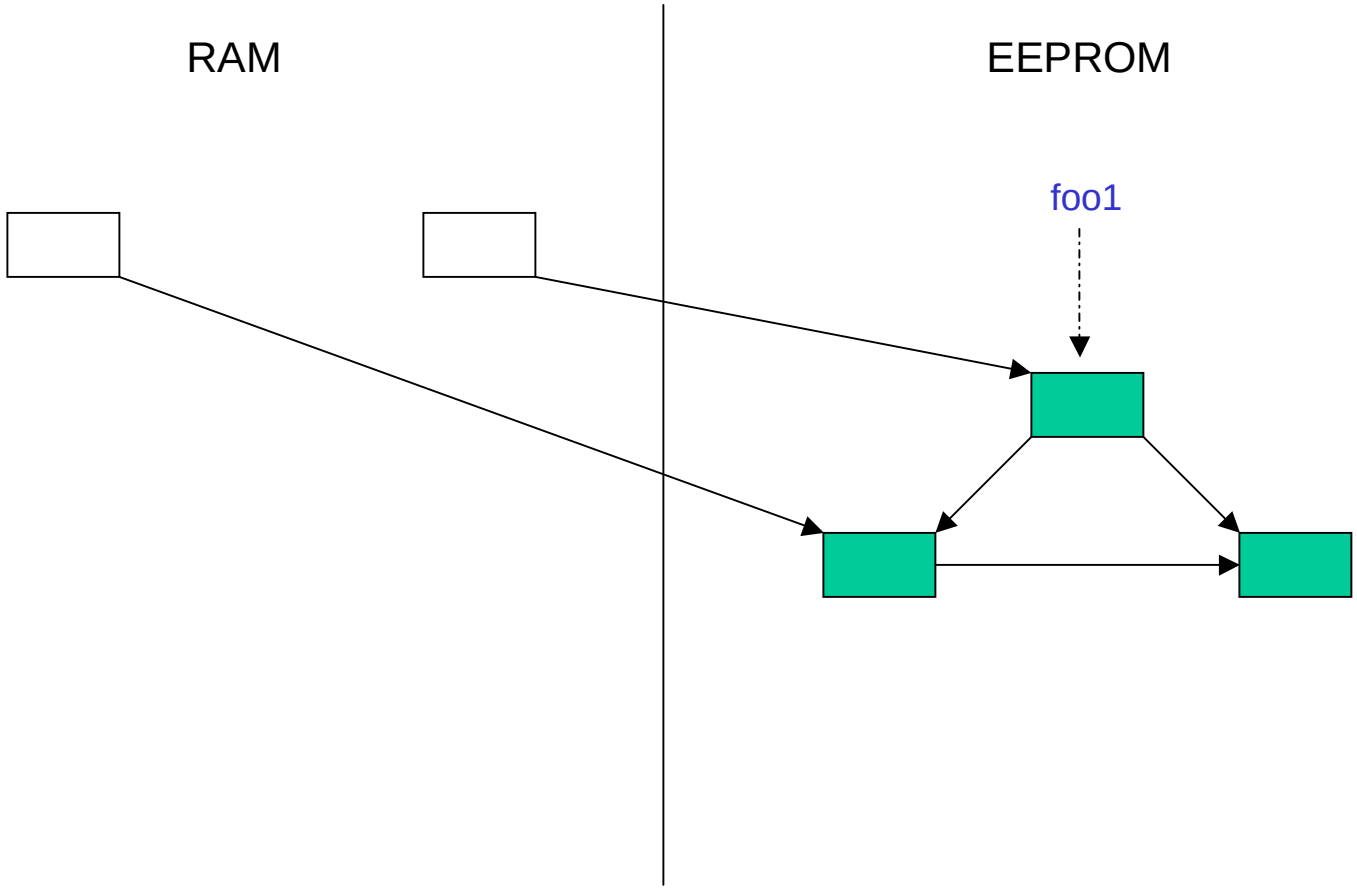
RAM

EEPROM



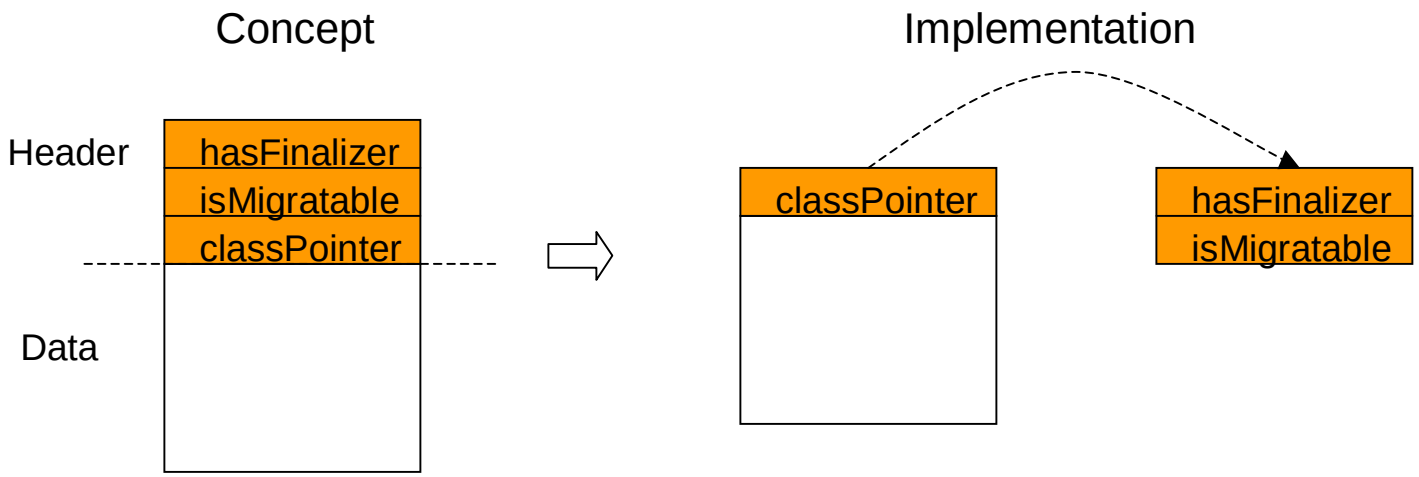
Migrating to EEPROM

PersistentMemory.makePersistent(foo1);



Finalizers and Auto-migration

- Implemented with object associations
- Associations maintained by garbage collector
- Associations created “on-demand” and are rare



Finalizers

- Objects queued for finalization by collector
- Finalizers run at context switch:
  1. `Thread.yield()`
  2. `Thread.sleep()`
  3. Thread dies
- Separate thread for each finalizer – safer!

Auto-migration

```
PersistentMemory.allowMigration(Object obj)
```

- Collector migrates marked objects to EEPROM while insufficient memory reclaimed
- Persistent collector must later be run to reclaim auto-migrated objects