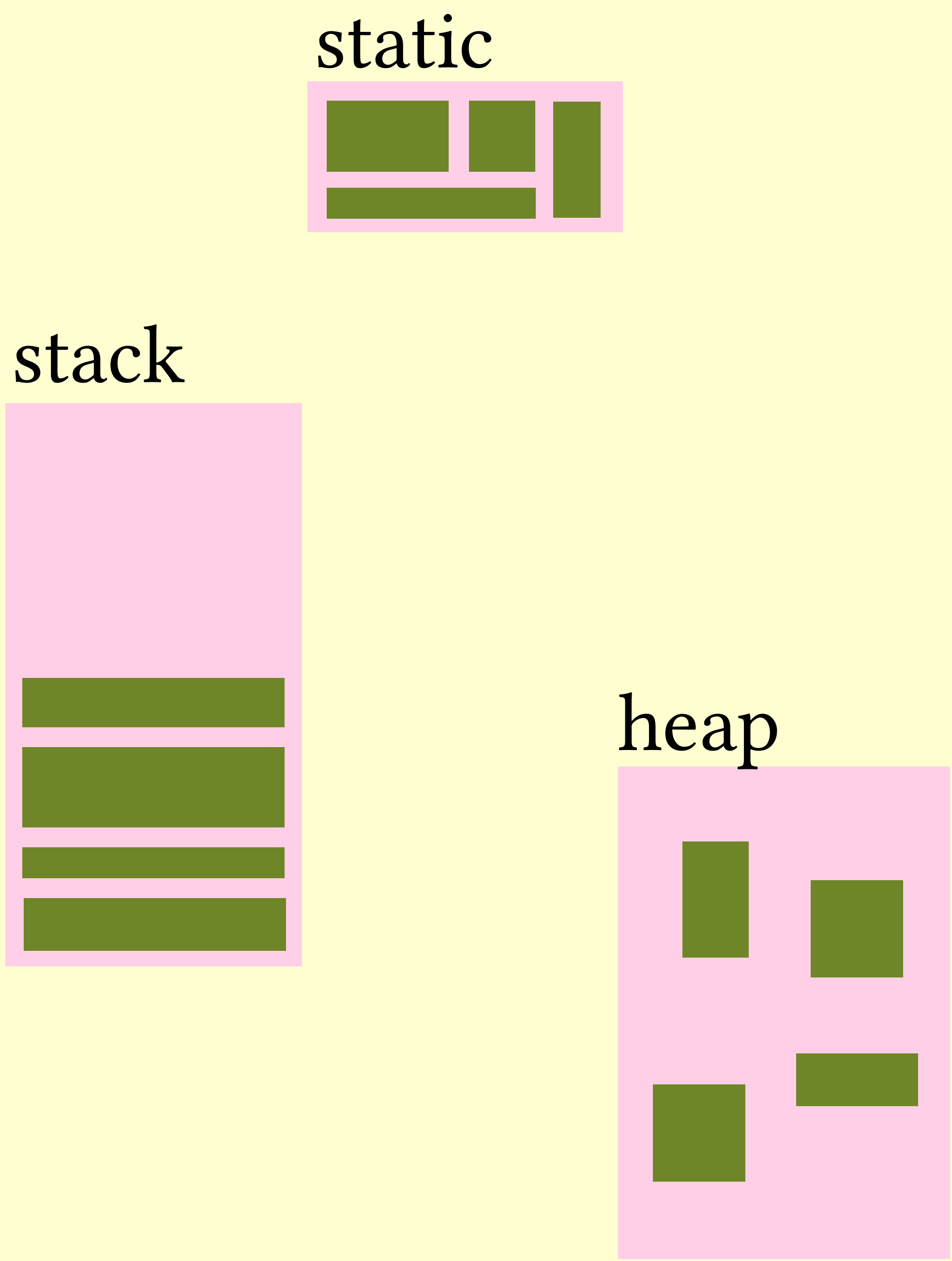
Lecture 13: Memory Management

John Wickerson

Compilers

Memory management

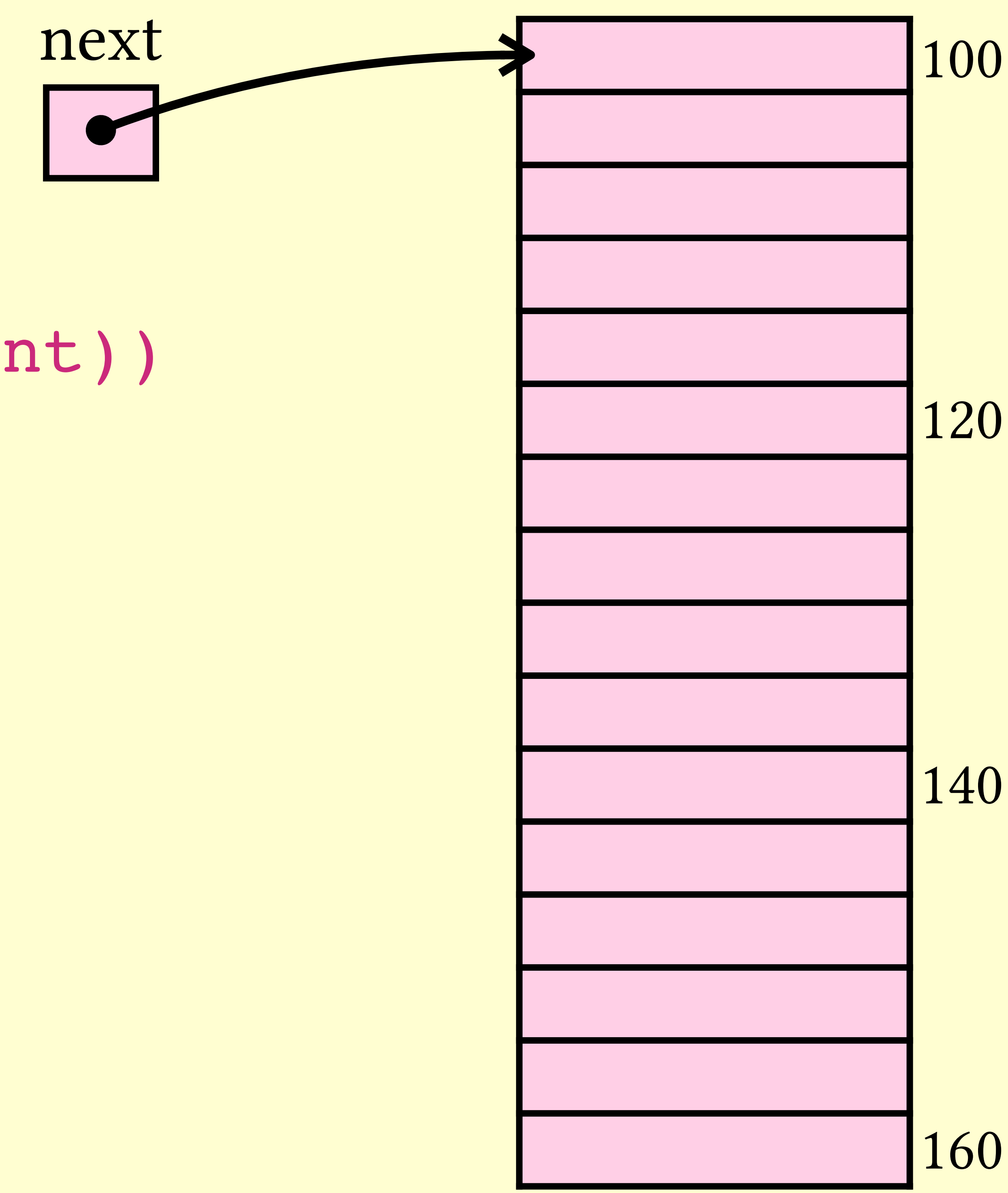


- Static allocation (e.g. global variables).
- Automatic/stack allocation (e.g., local variables).
- Dynamic/heap allocation, managed by programmer (i.e., `malloc`, `free`).
- Dynamic/heap allocation, managed automatically (i.e., garbage collection).

Lecture outline

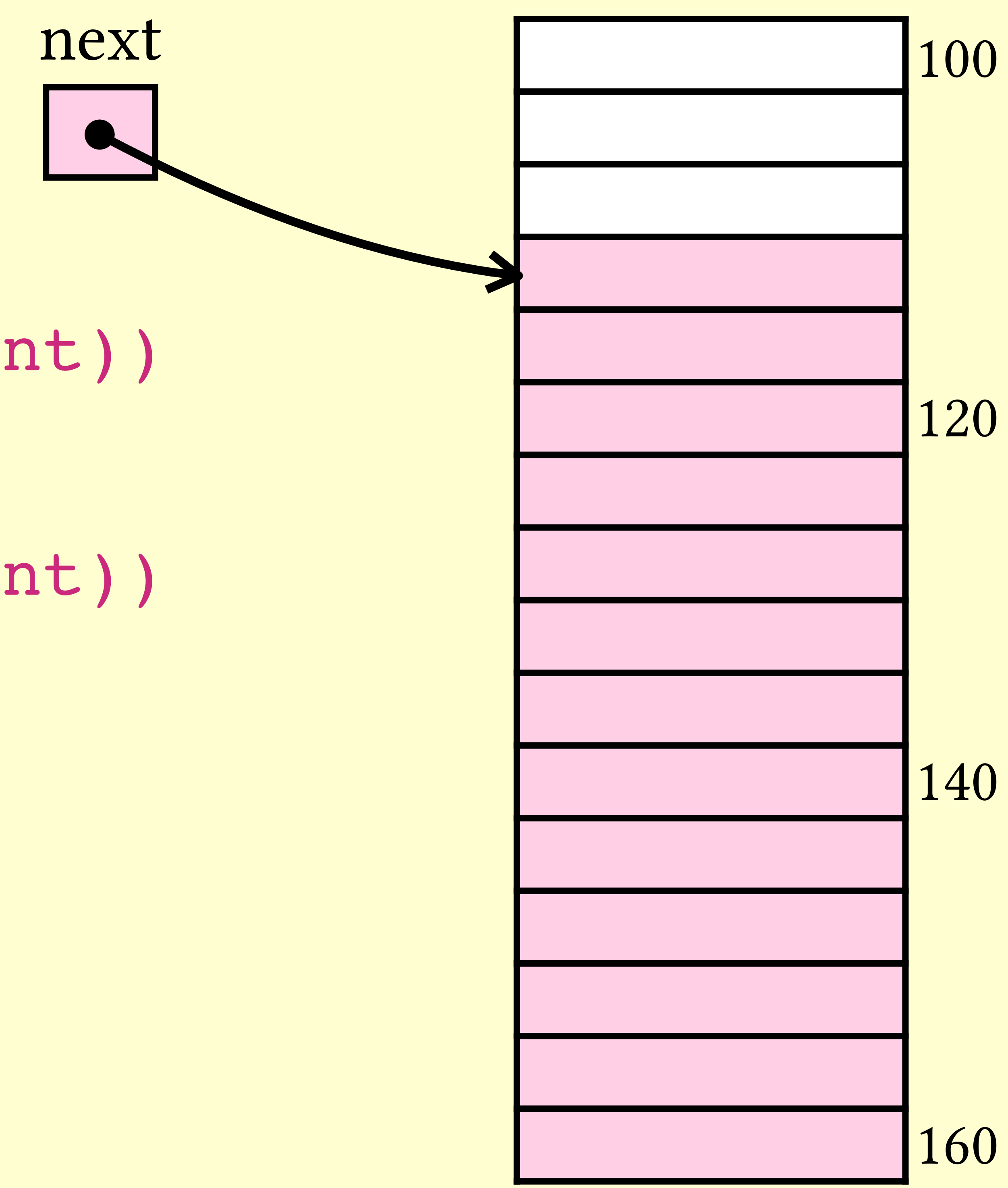
- Dynamic memory allocation (`malloc` and `free`)
- Garbage collection

First malloc



request: `malloc(3*sizeof(int))`

First malloc

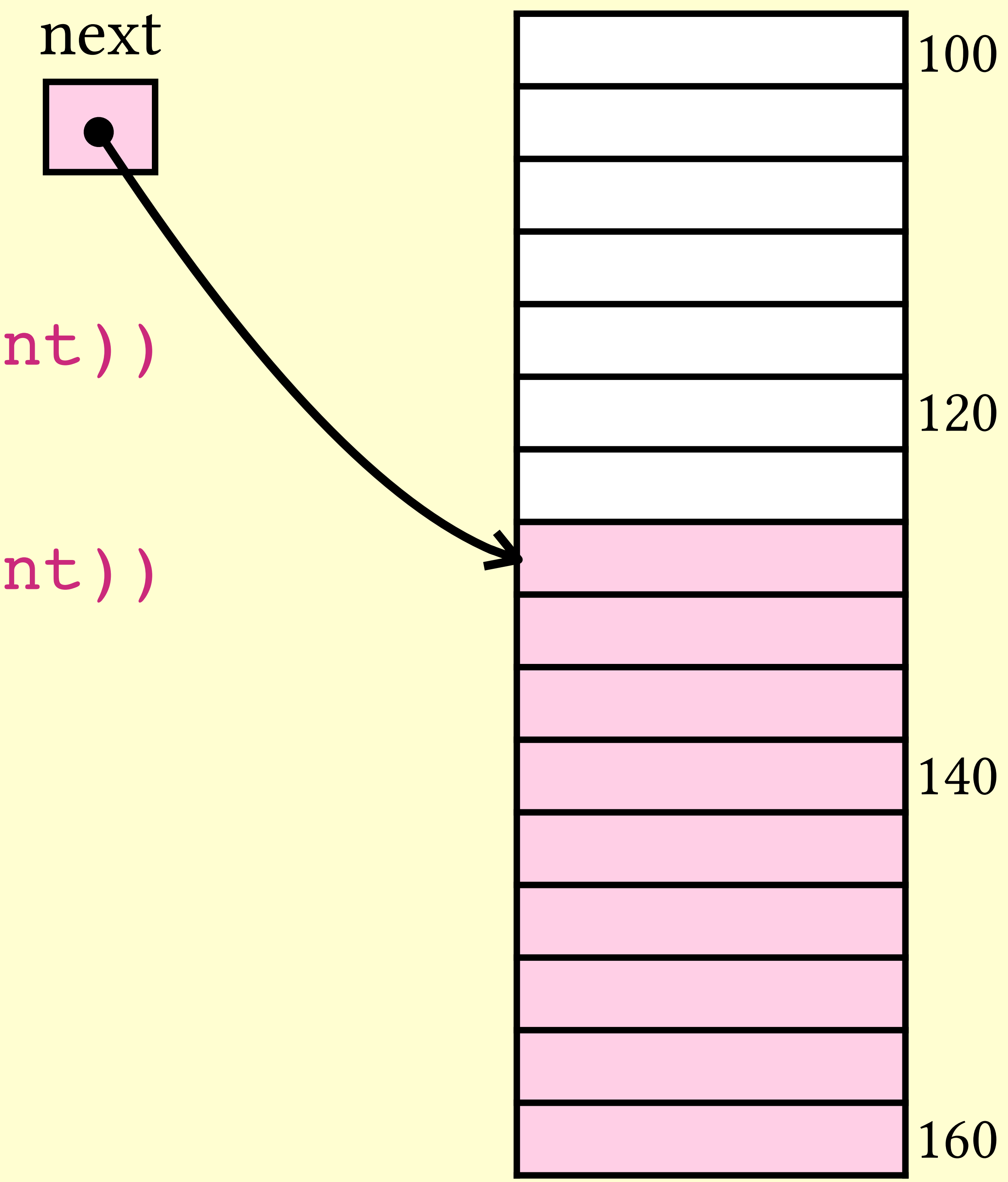


request: `malloc(3*sizeof(int))`

response: `100`

request: `malloc(4*sizeof(int))`

First malloc



request: `malloc(3*sizeof(int))`

response: `100`

request: `malloc(4*sizeof(int))`

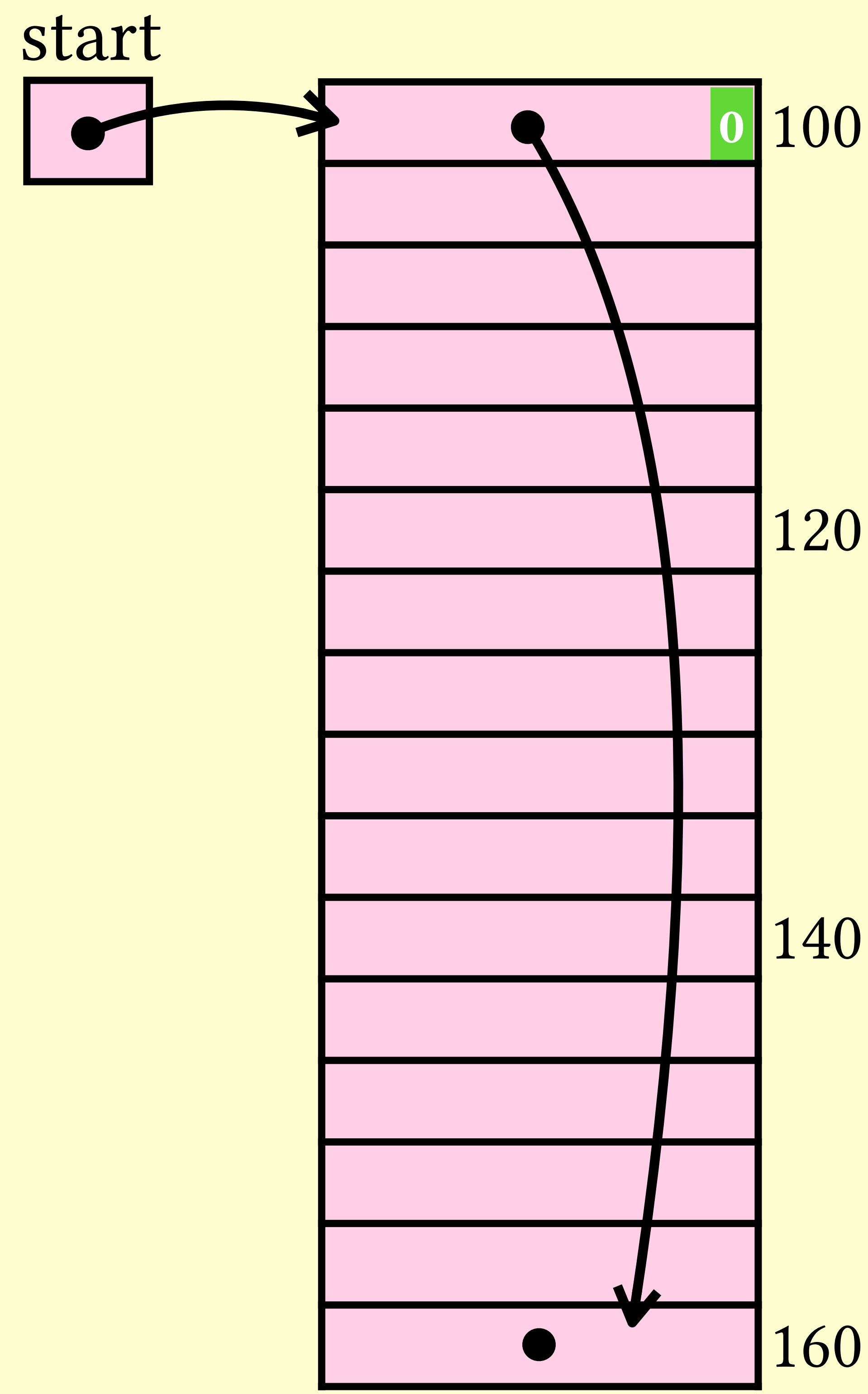
response: `112`

request: `free(100)`

response: `[done]`

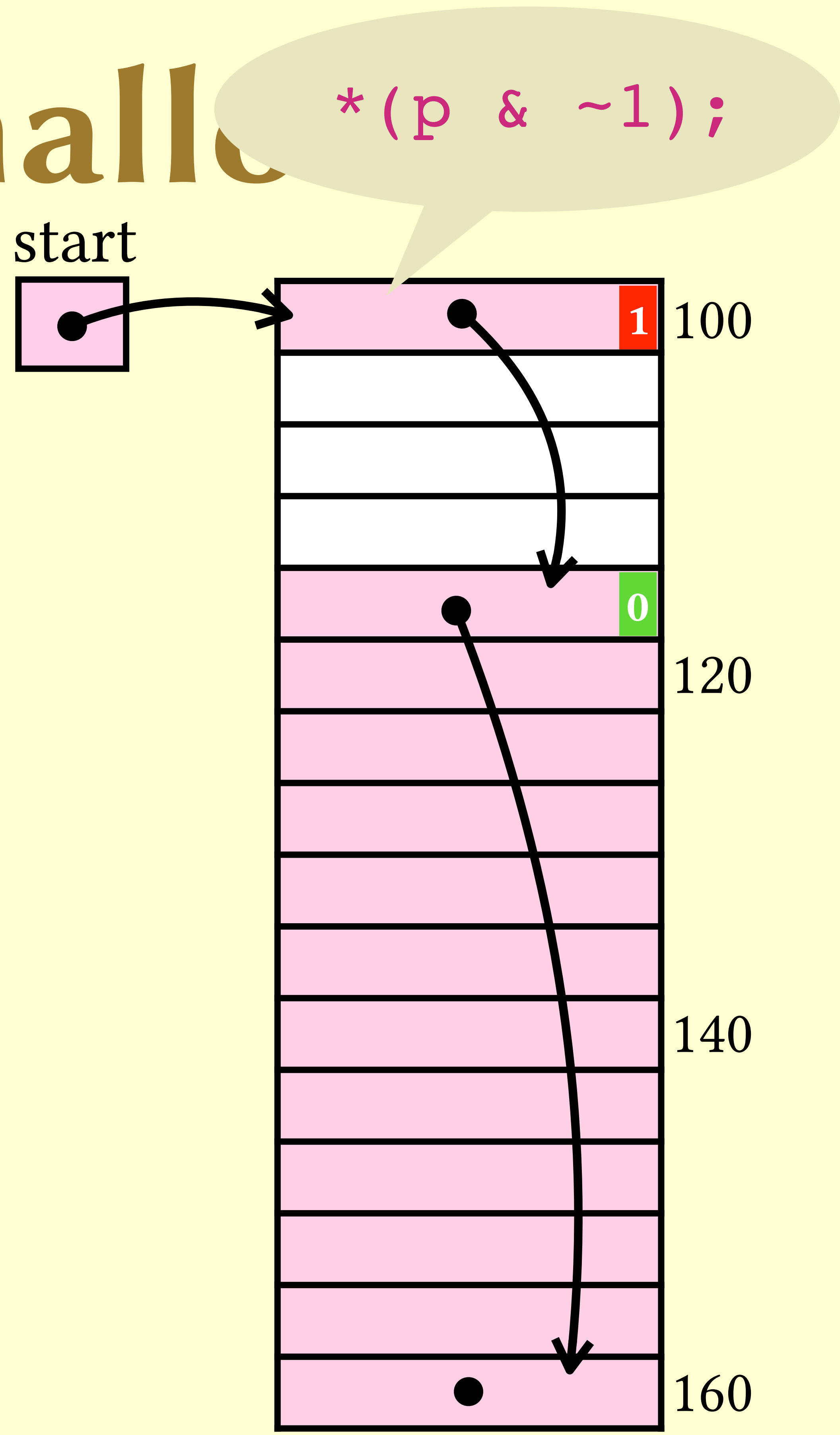
Can we do better?

Second malloc



request: `malloc(3*sizeof(int))`

Second mallo

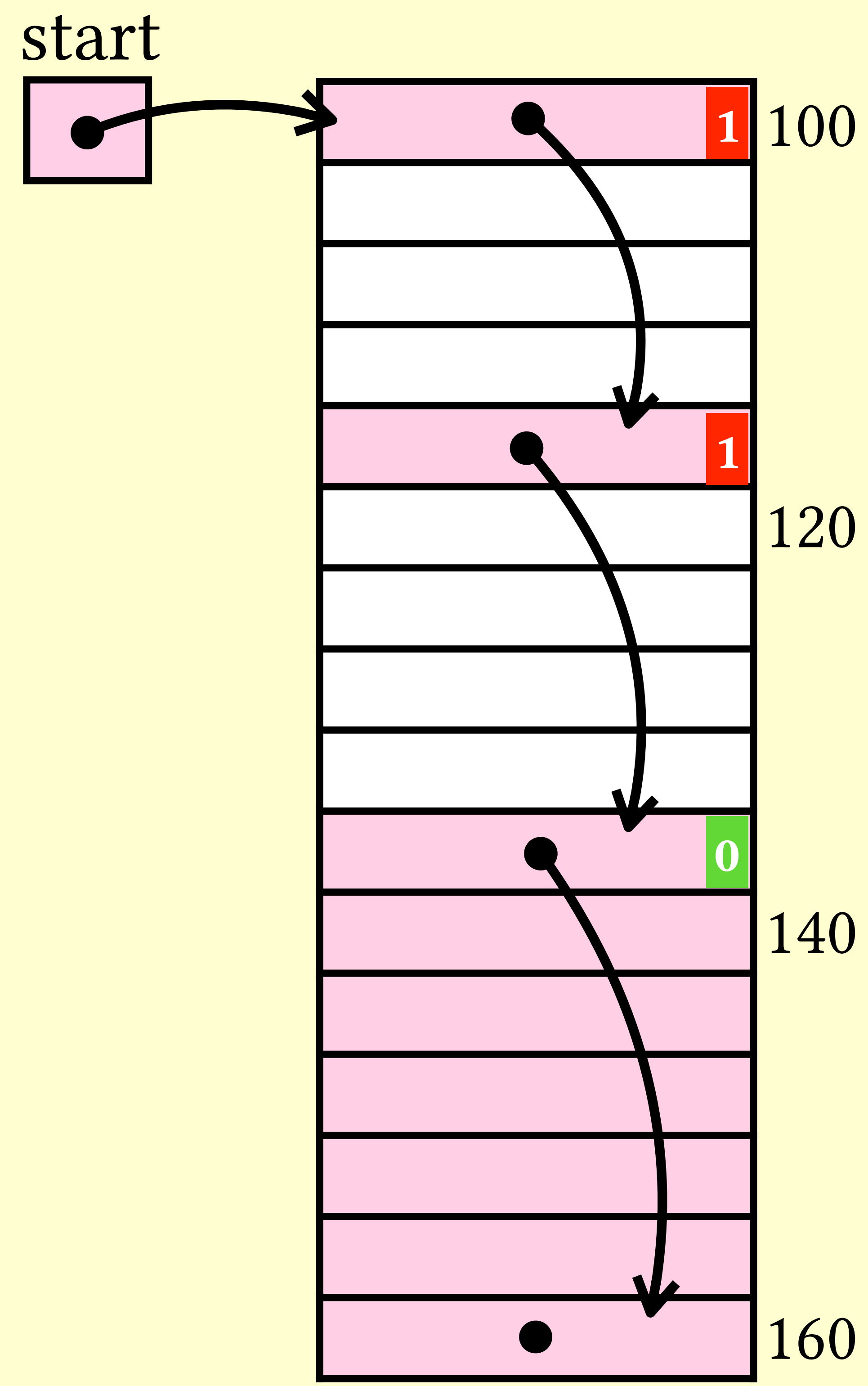


request: `malloc(3*sizeof(int))`

response: `104`

request: `malloc(4*sizeof(int))`

Second malloc



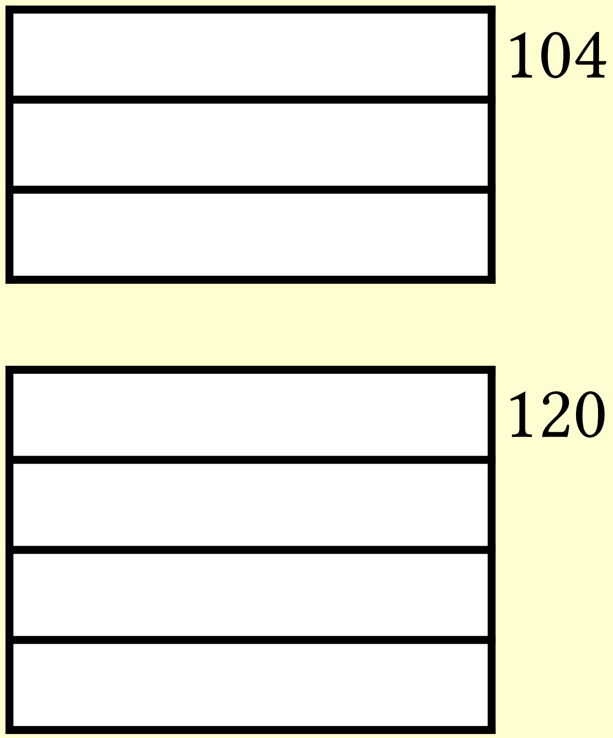
request: `malloc(3*sizeof(int))`

response: `104`

request: `malloc(4*sizeof(int))`

response: `120`

Second malloc



request: `malloc(3*sizeof(int))`

response: `104`

request: `malloc(4*sizeof(int))`

response: `120`

Second malloc

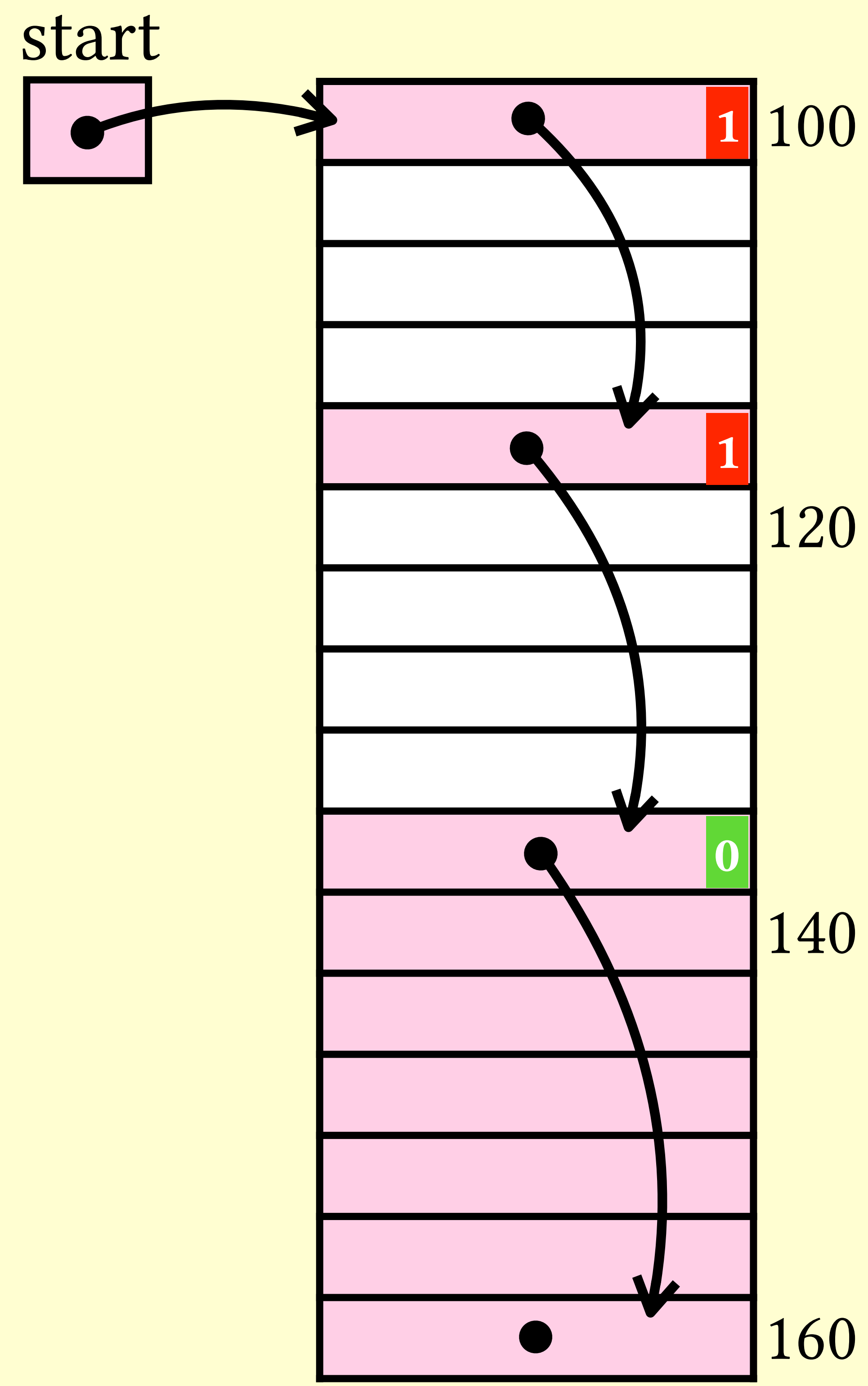
request: `malloc(3*sizeof(int))`

response: `104`

request: `malloc(4*sizeof(int))`

response: `120`

request: `free(104)`



Second malloc

request: `malloc(3*sizeof(int))`

response: `104`

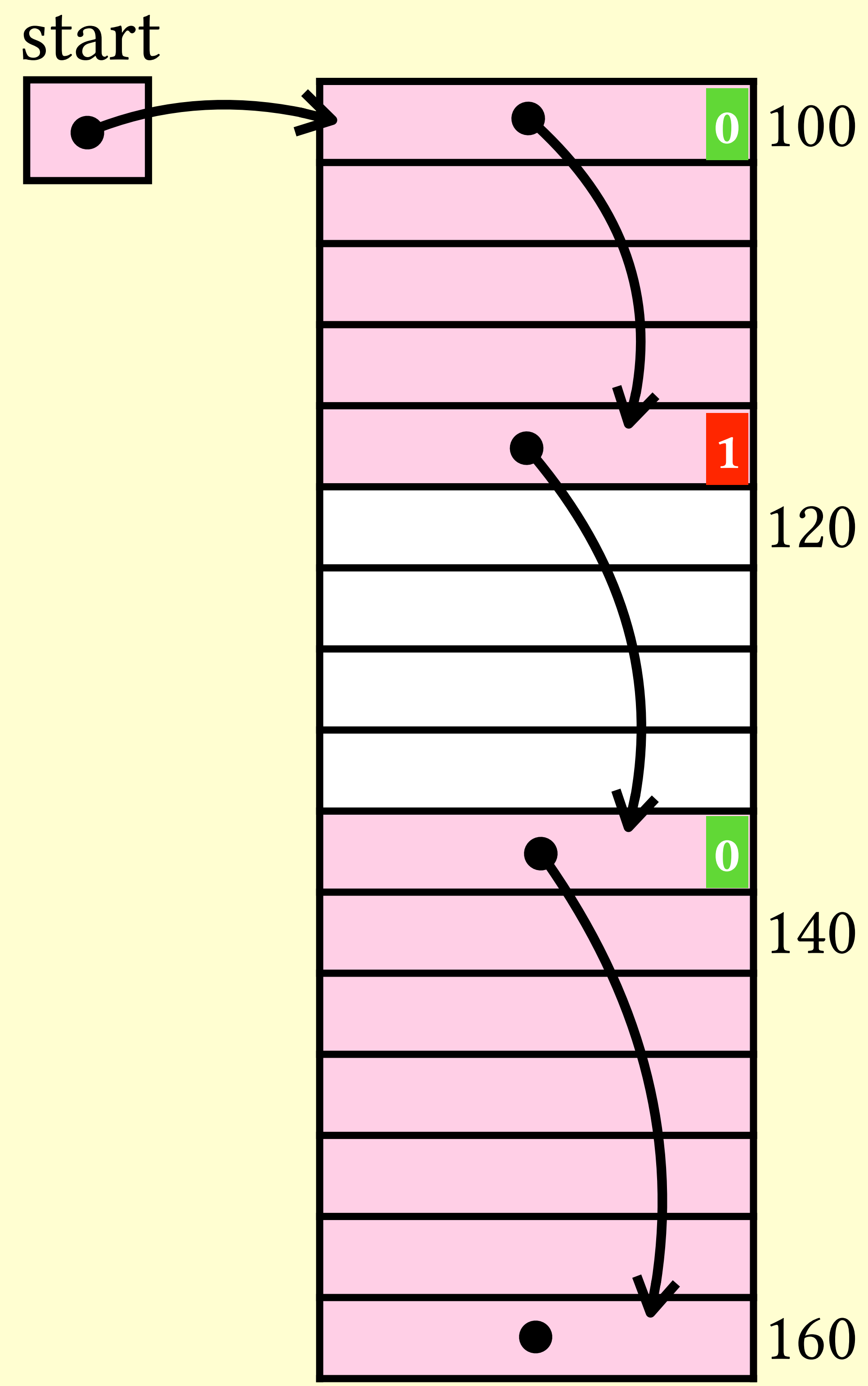
request: `malloc(4*sizeof(int))`

response: `120`

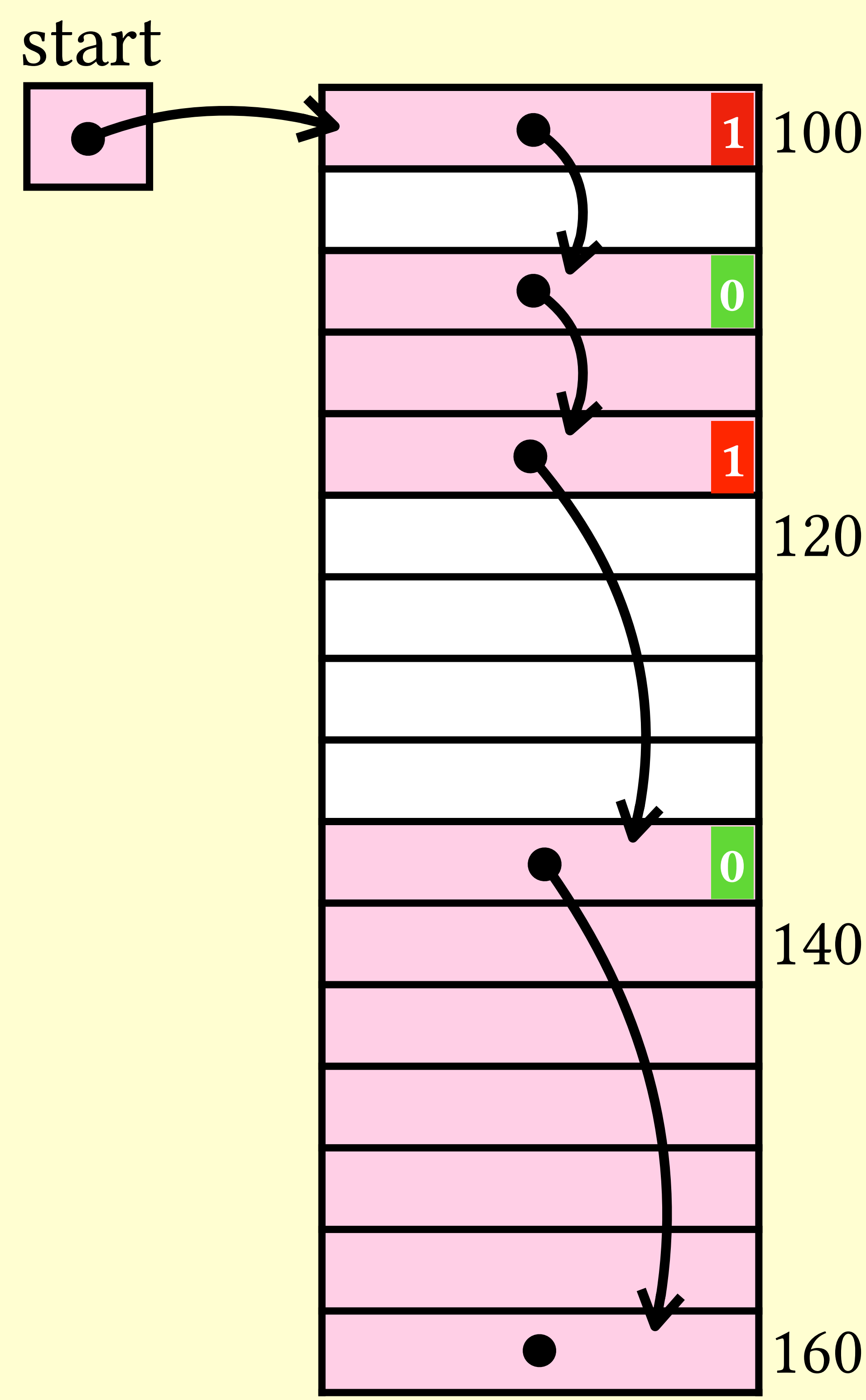
request: `free(104)`

response: `[done]`

request: `malloc(1*sizeof(int))`



Second malloc



request: `malloc(3*sizeof(int))`

response: `104`

request: `malloc(4*sizeof(int))`

response: `120`

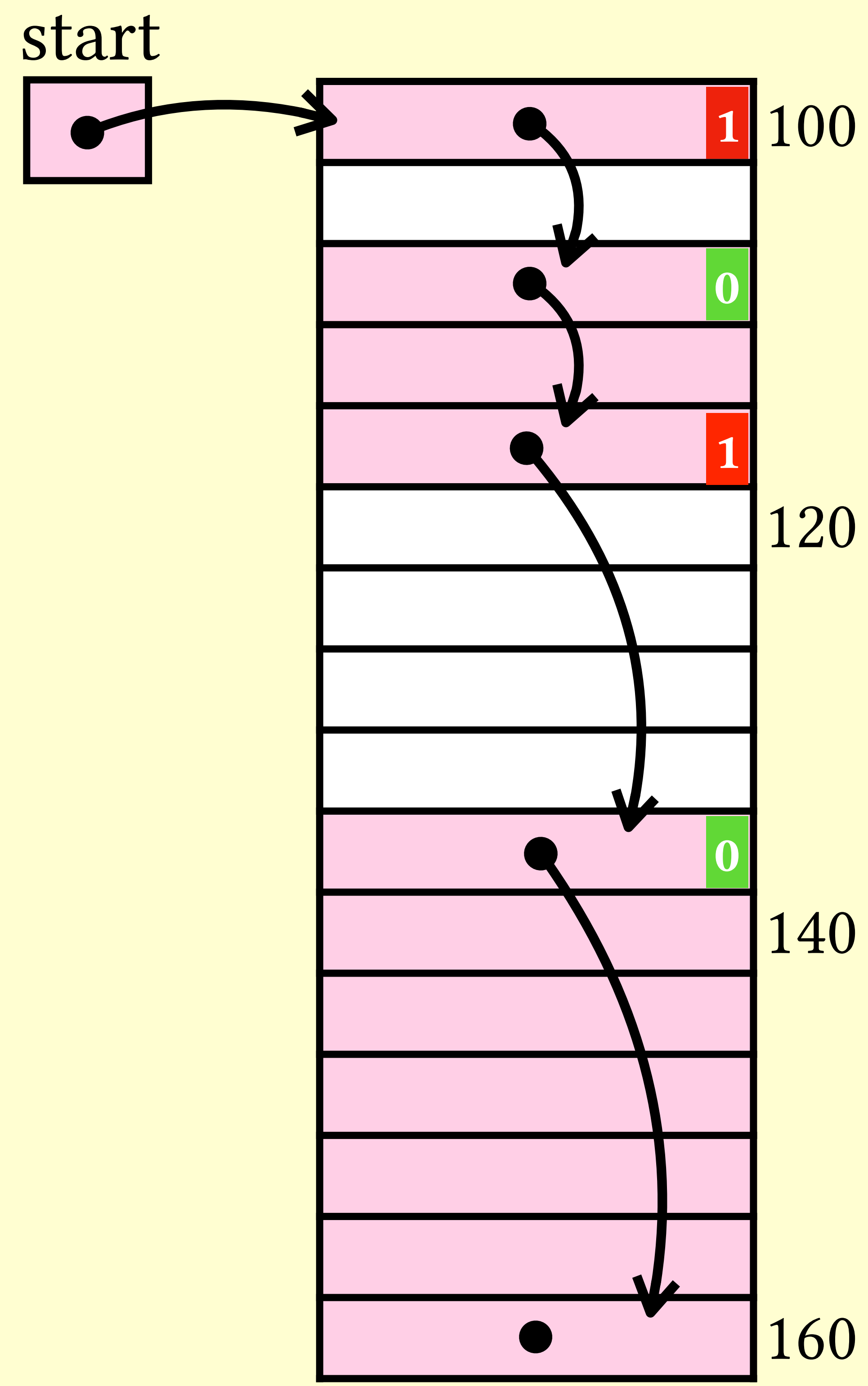
request: `free(104)`

response: `[done]`

request: `malloc(1*sizeof(int))`

response: `104`

Second malloc



request: `malloc(3*sizeof(int))`

response: `104`

request: `malloc(4*sizeof(int))`

response: `120`

request: `free(104)`

response: `[done]`

request: `malloc(1*sizeof(int))`

response: `104`

Second malloc

request: `malloc(3*sizeof(int))`

response: `104`

request: `malloc(4*sizeof(int))`

response: `120`

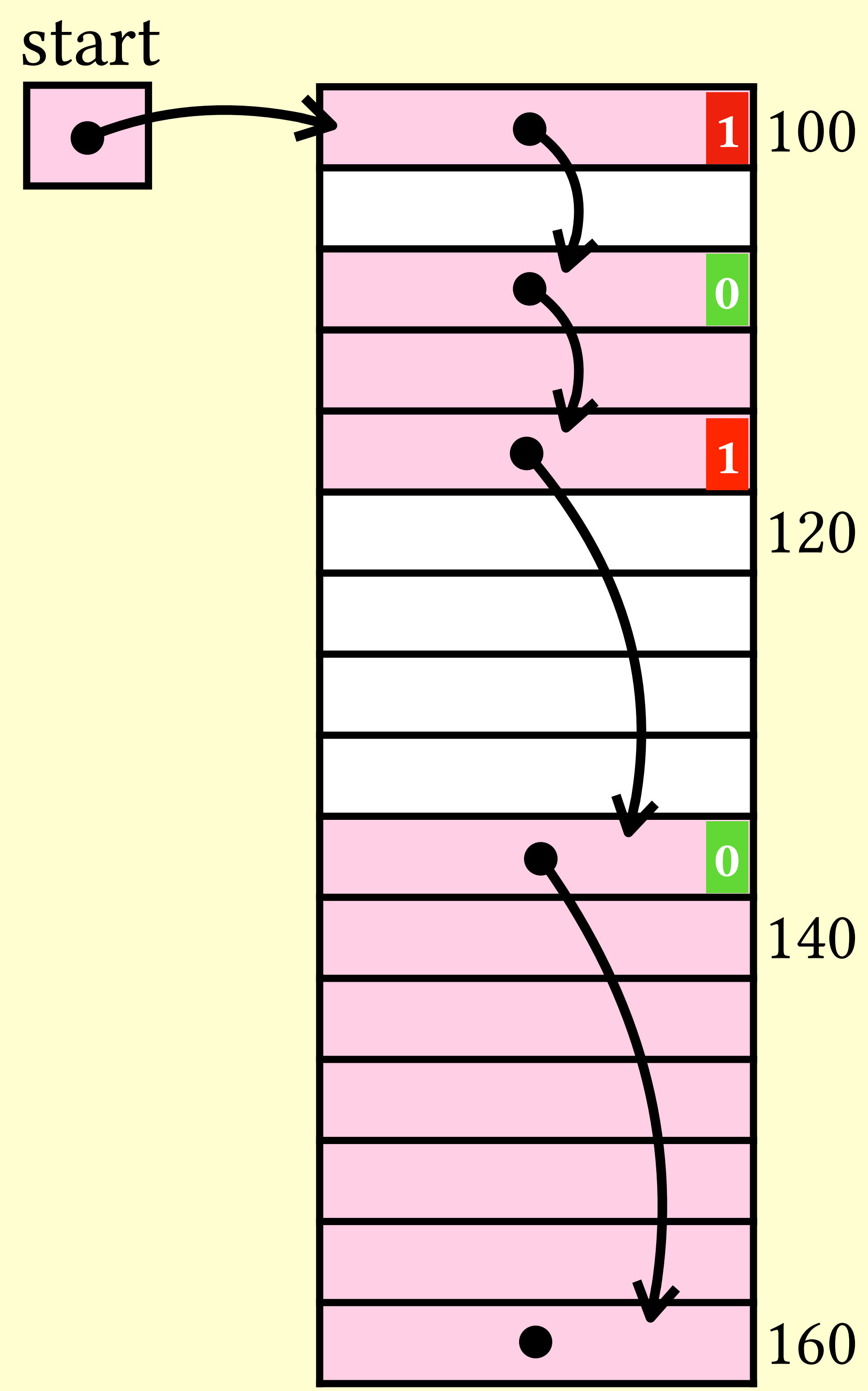
request: `free(104)`

response: `[done]`

request: `malloc(1*sizeof(int))`

response: `104`

request: `malloc(5*sizeof(int))`



Second malloc

request: `malloc(3*sizeof(int))`

response: `104`

request: `malloc(4*sizeof(int))`

response: `120`

request: `free(104)`

response: `[done]`

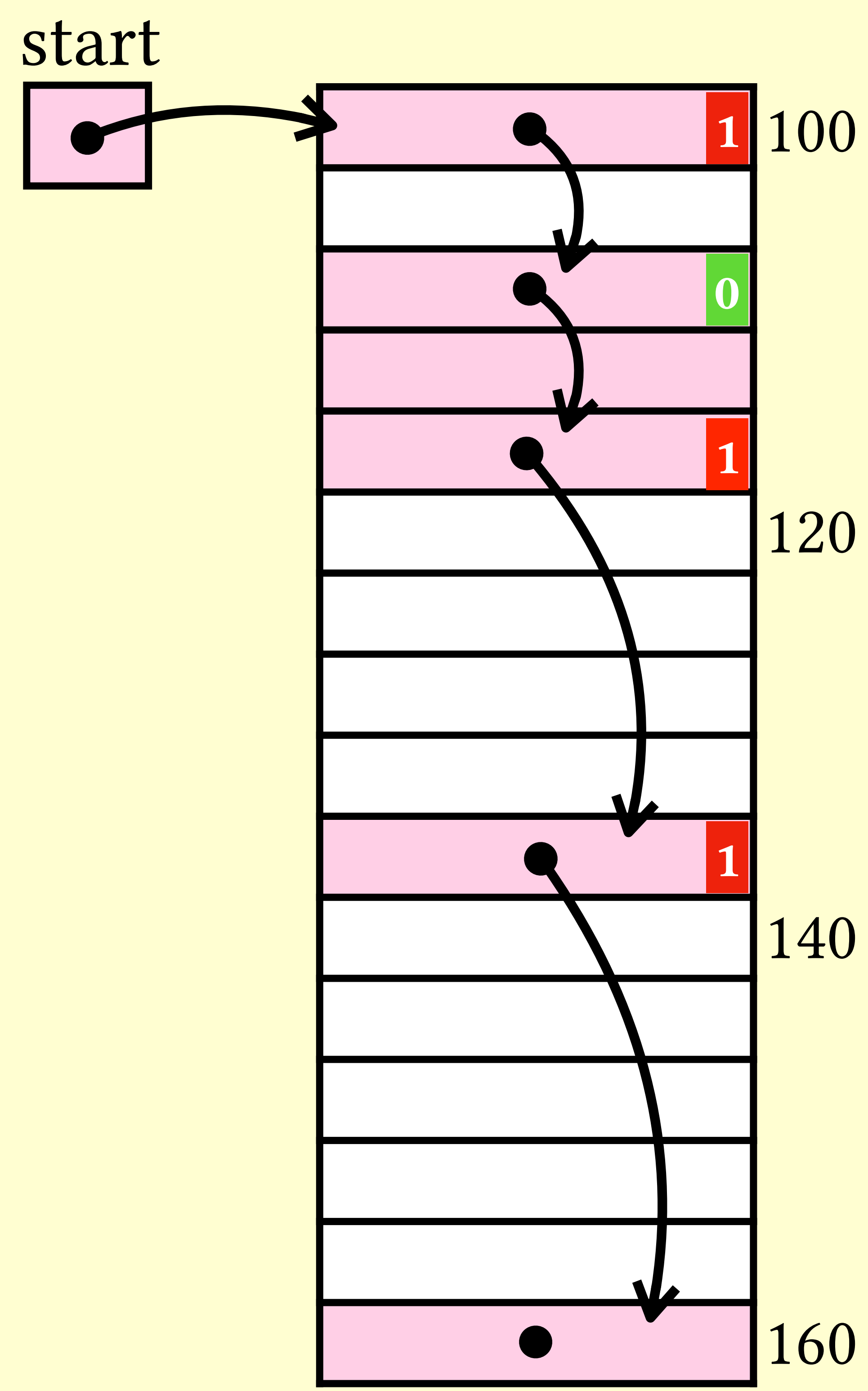
request: `malloc(1*sizeof(int))`

response: `104`

request: `malloc(5*sizeof(int))`

response: `140`

request: `free(120)`



Second malloc

request: `malloc(3*sizeof(int))`

response: `104`

request: `malloc(4*sizeof(int))`

response: `120`

request: `free(104)`

response: `[done]`

request: `malloc(1*sizeof(int))`

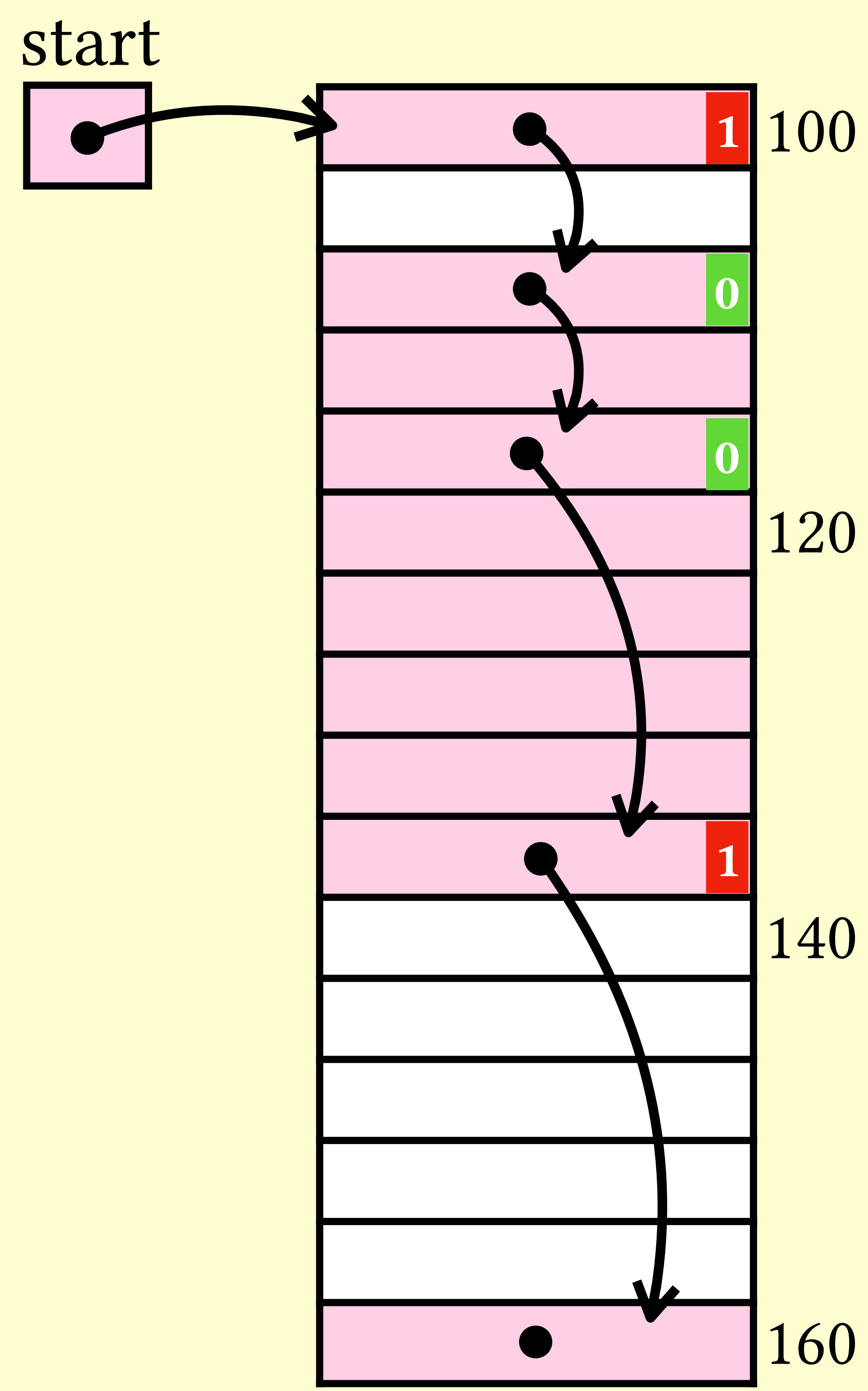
response: `104`

request: `malloc(5*sizeof(int))`

response: `140`

request: `free(120)`

response: `[done]`



Second malloc

request: `malloc(3*sizeof(int))`

response: `104`

request: `malloc(4*sizeof(int))`

response: `120`

request: `free(104)`

response: `[done]`

request: `malloc(1*sizeof(int))`

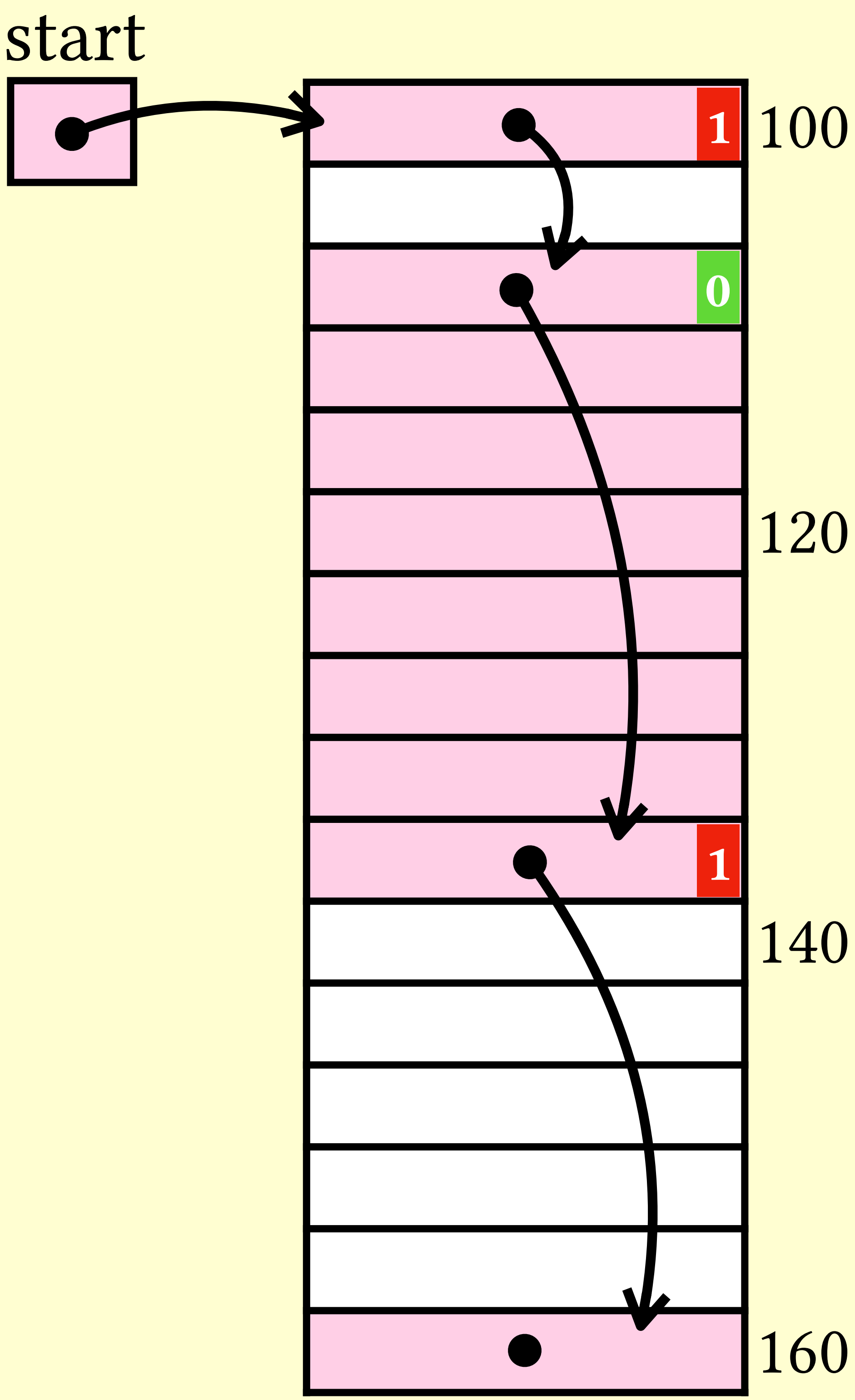
response: `104`

request: `malloc(5*sizeof(int))`

response: `140`

request: `free(120)`

response: `[done]`



Second malloc

Explicit Stabilisation for Modular Rely-Guarantee Reasoning

John Wickerson, Mike Dodds and Matthew Parkinson

University of Cambridge Computer Laboratory

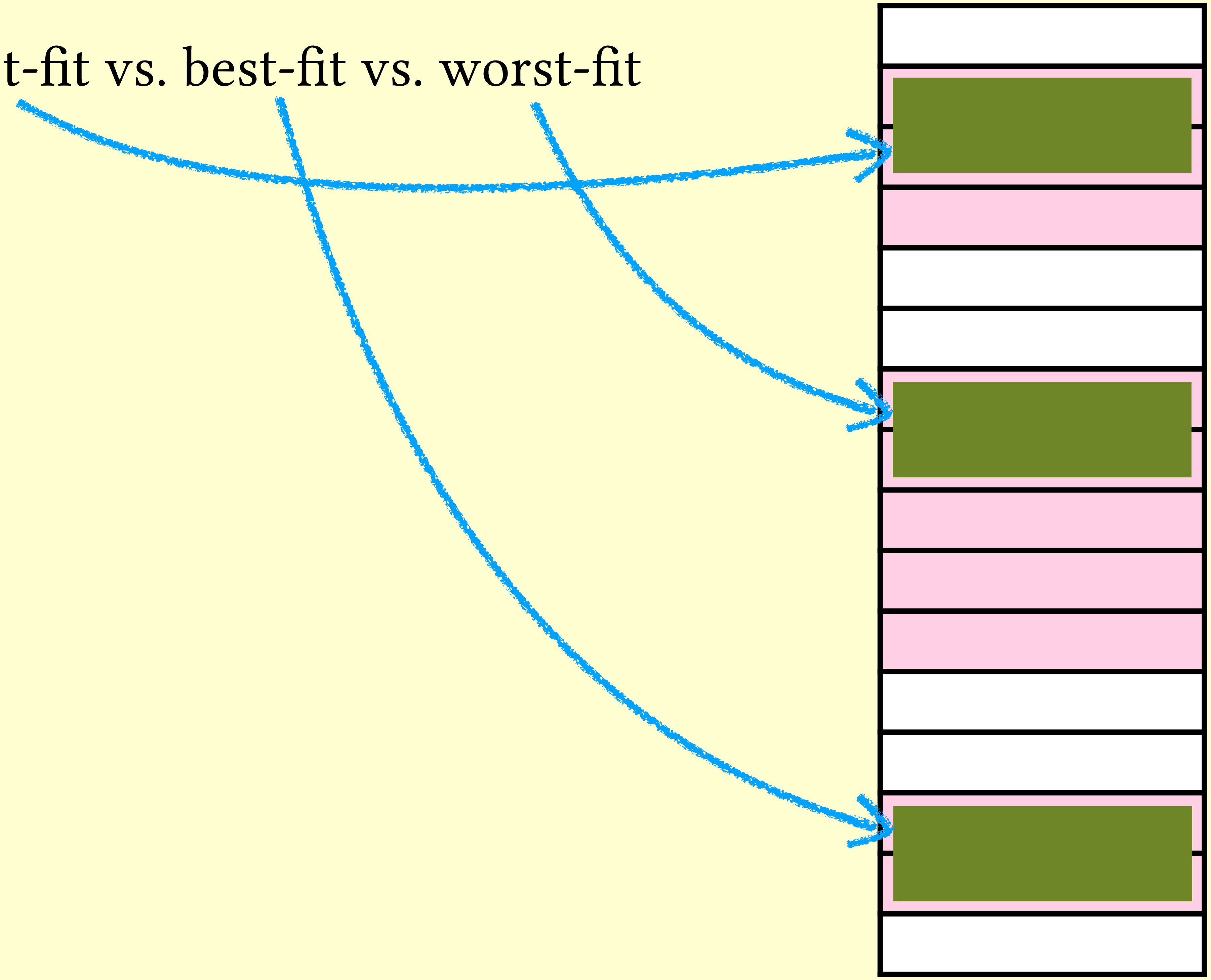
Abstract. We propose a new formalisation of stability for Rely-Guarantee, in which an assertion's stability is encoded into its syntactic form. This allows two advances in modular reasoning. Firstly, it enables Rely-Guarantee, for the first time, to verify concurrent libraries independently of their clients' environments. Secondly, in a sequential setting, it allows a module's internal interference to be hidden while verifying its clients. We demonstrate our approach by verifying, using RGSep, the Version 7 Unix memory manager, uncovering a twenty-year-old bug in the process.

1 Introduction

... programs is hard because commands from different

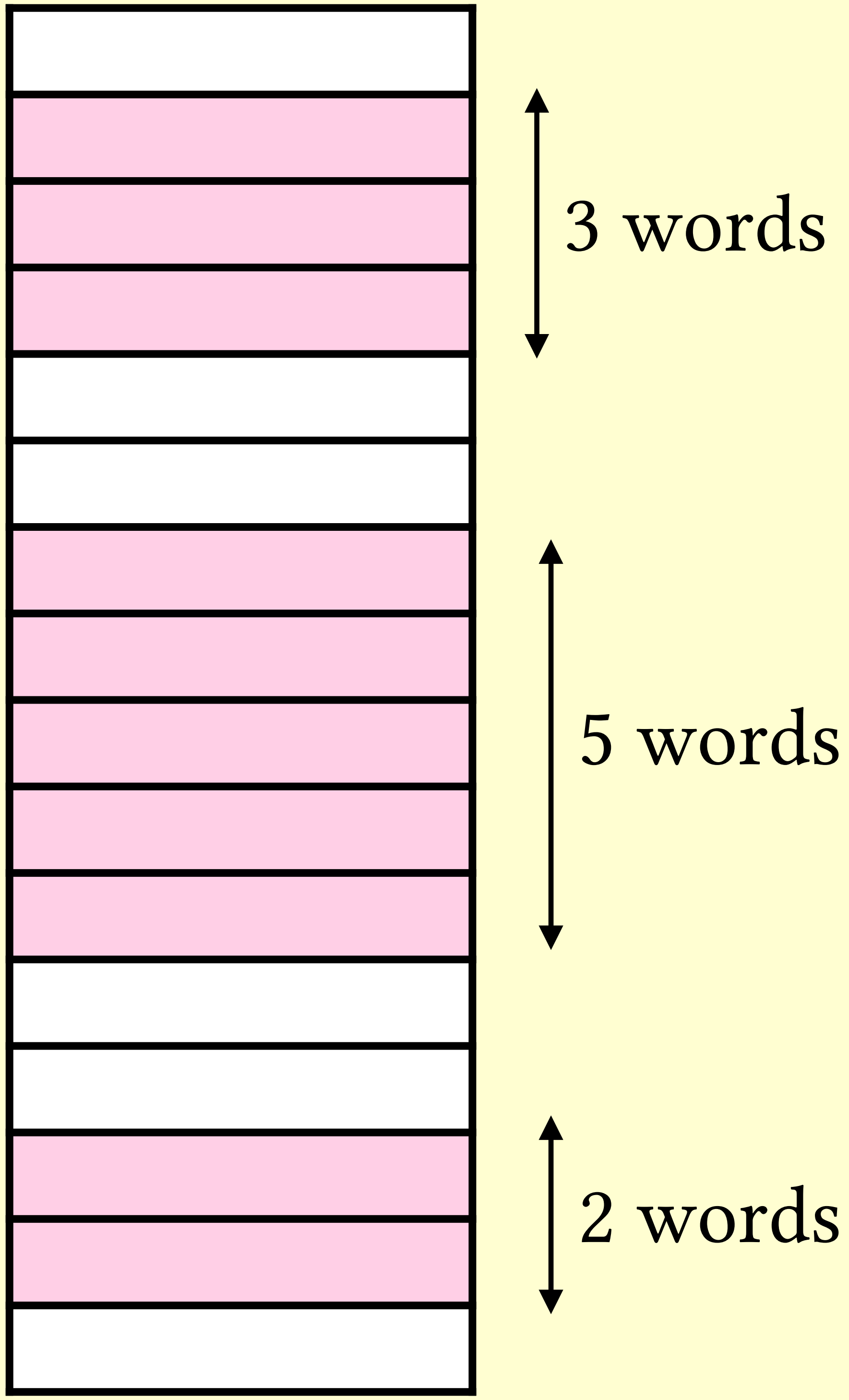
Malloc decisions

- First-fit vs. best-fit vs. worst-fit



Malloc decisions

- First-fit vs. best-fit vs. worst-fit
- **Exercise.** Devise a sequence of `malloc` calls where worst-fit is the best of the three strategies.

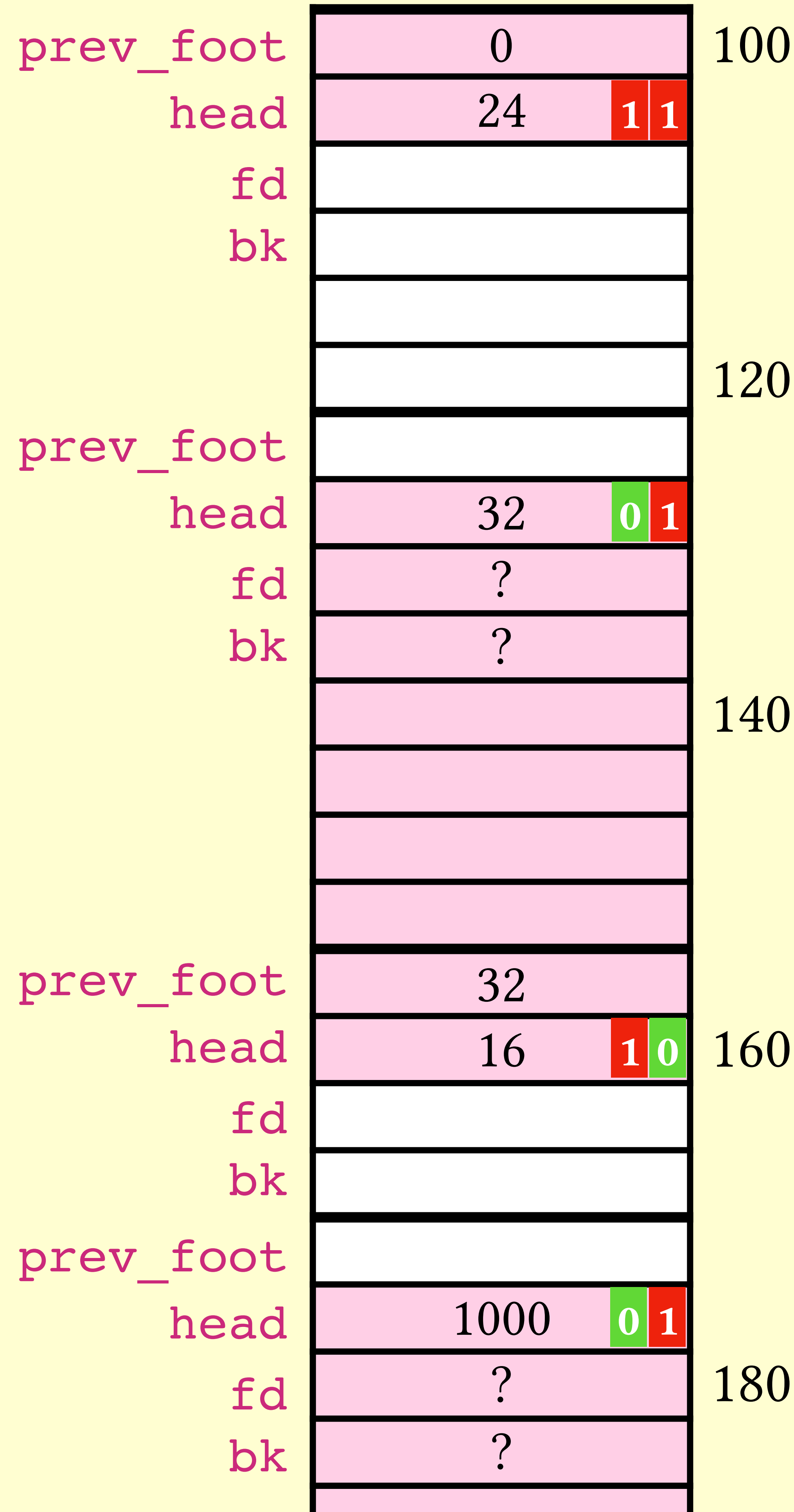


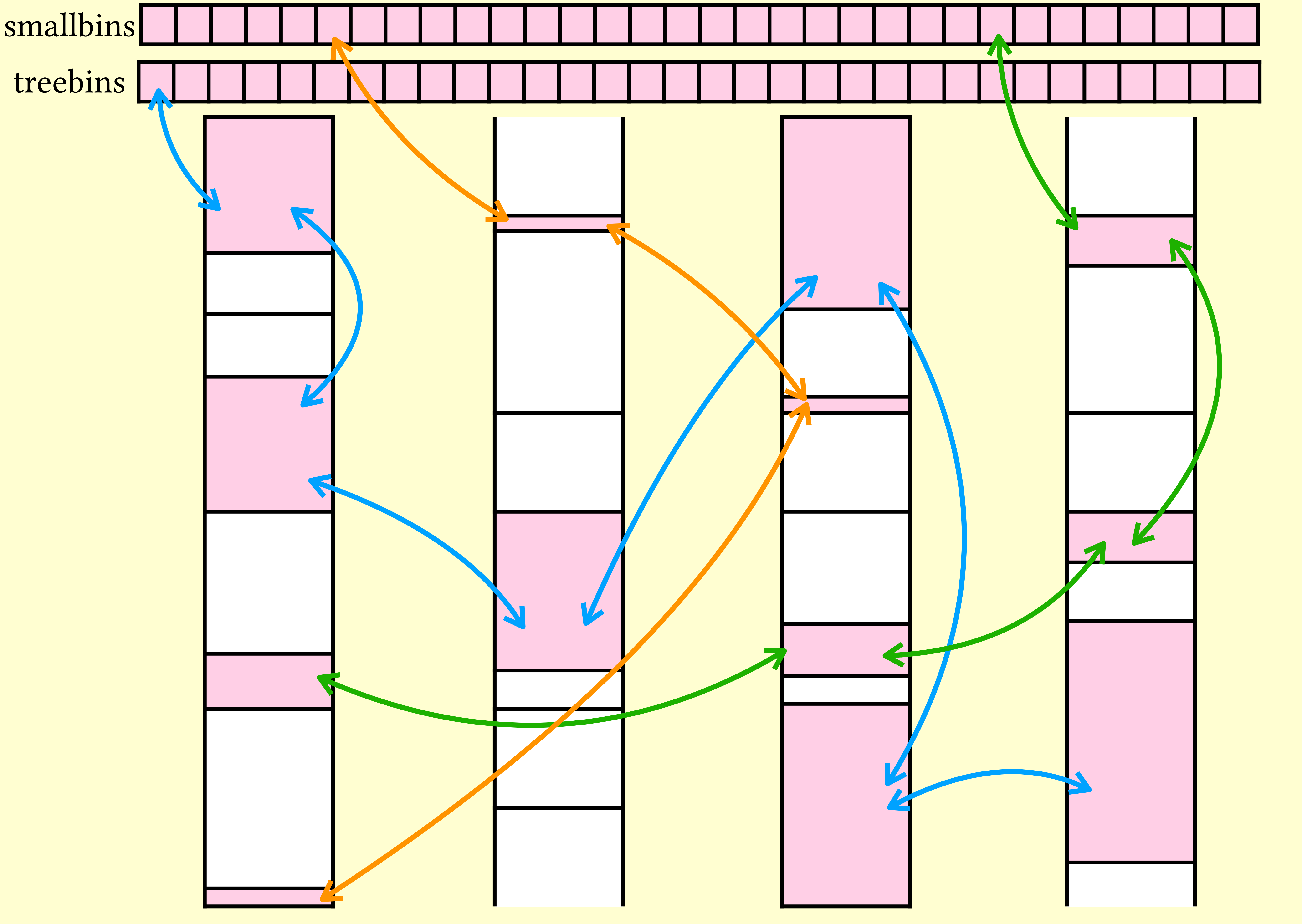
Can we do better?

Third malloc

- Doug Lea's malloc (`dlmalloc`) was invented around 1987. Its derivatives are widely used in GNU/Linux.

Doug Lea


prev_foot
0
100
head
24
1
1
fd
bk
120
prev_foot
head
32
0
1
fd
?
bk
?
140
prev_foot
32
head
16
1
0
160
fd
bk
prev_foot
head
1000
0
1
fd
?
180
bk
?


smallbins
treebins

Can we do better?

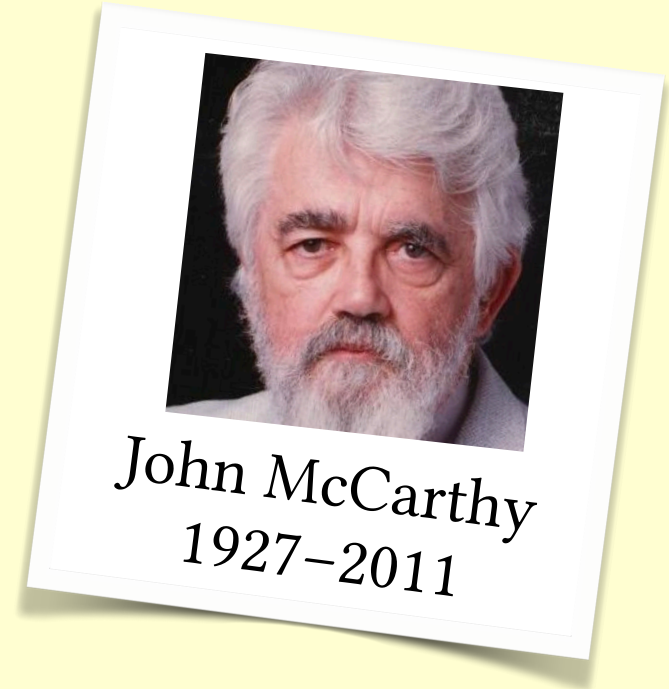
Other mallocs

- `phkmalloc`, `ptmalloc`, `jemalloc`, `snmalloc`, …
- Better performance when multiple threads are concurrently calling malloc and free.

Lecture outline

- ✓ Dynamic memory allocation (`malloc` and `free`)
- Garbage collection

Garbage collection



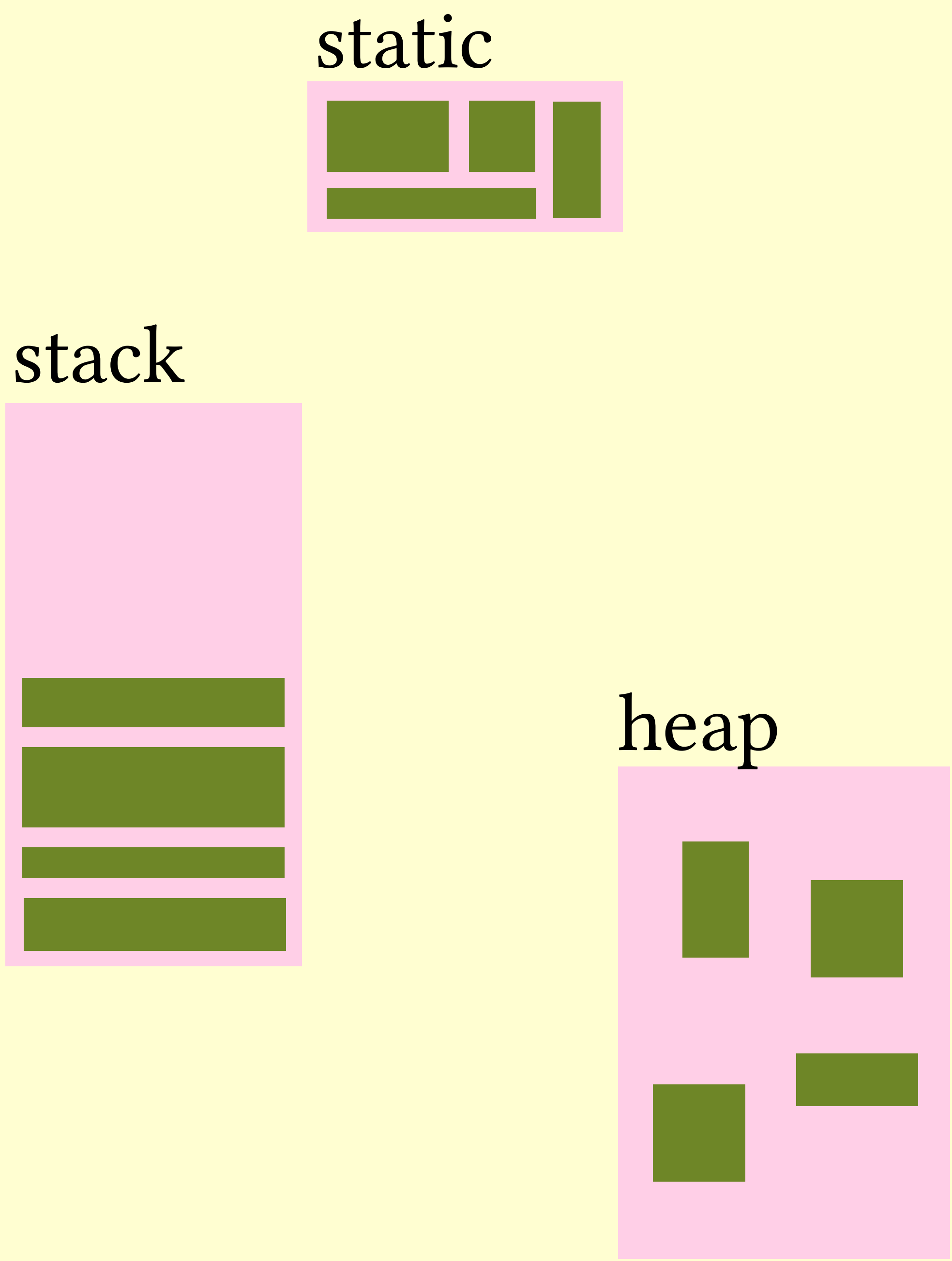
- Invented around 1959 by John McCarthy and implemented in his LISP programming language.

- Alternative to `free`. The garbage collector automatically identifies memory that will never be used again by the program, and makes it available for reallocation.

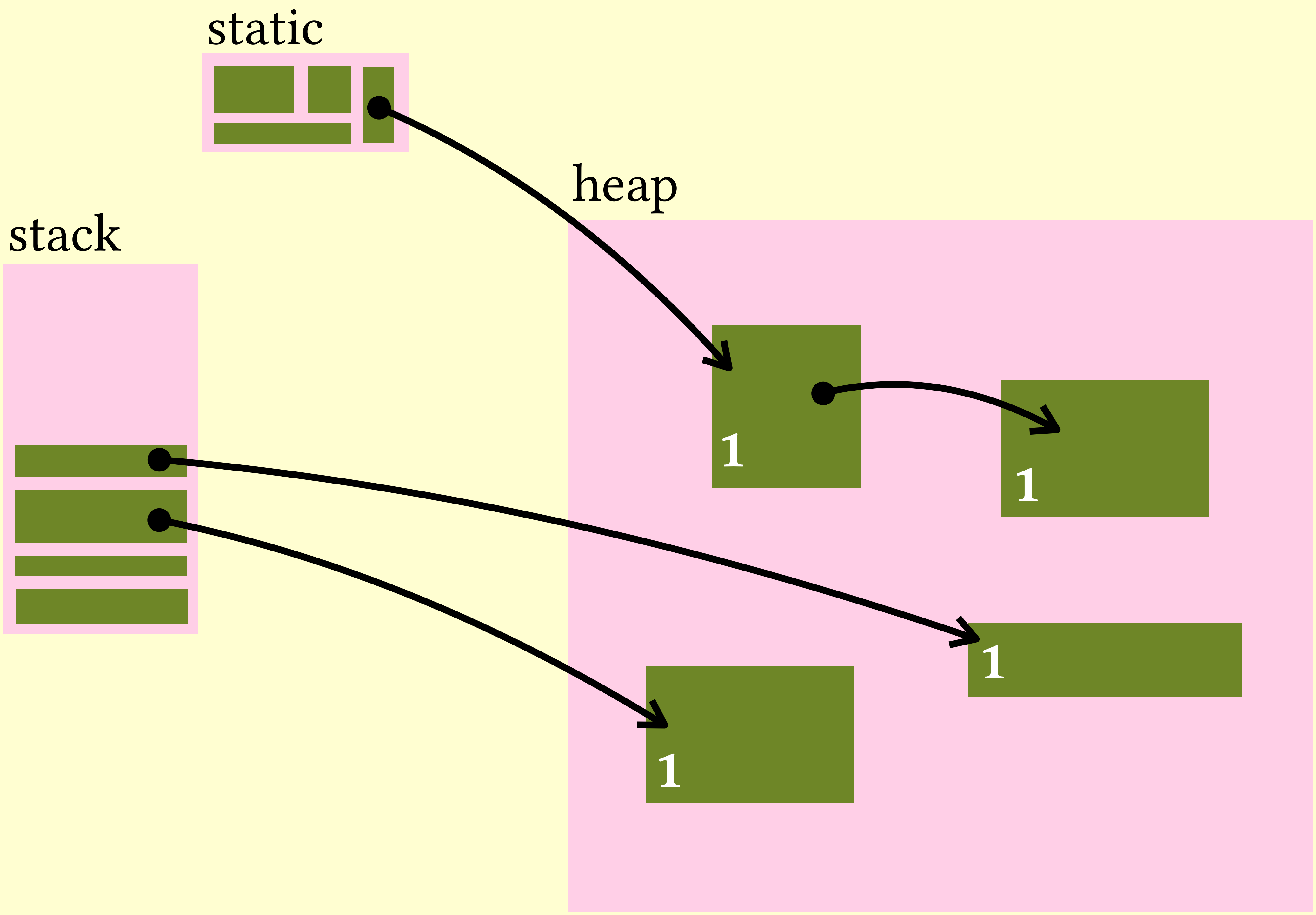
- **Question.** Which languages use garbage collection?

- **Question.** What are the advantages/disadvantages of GC?

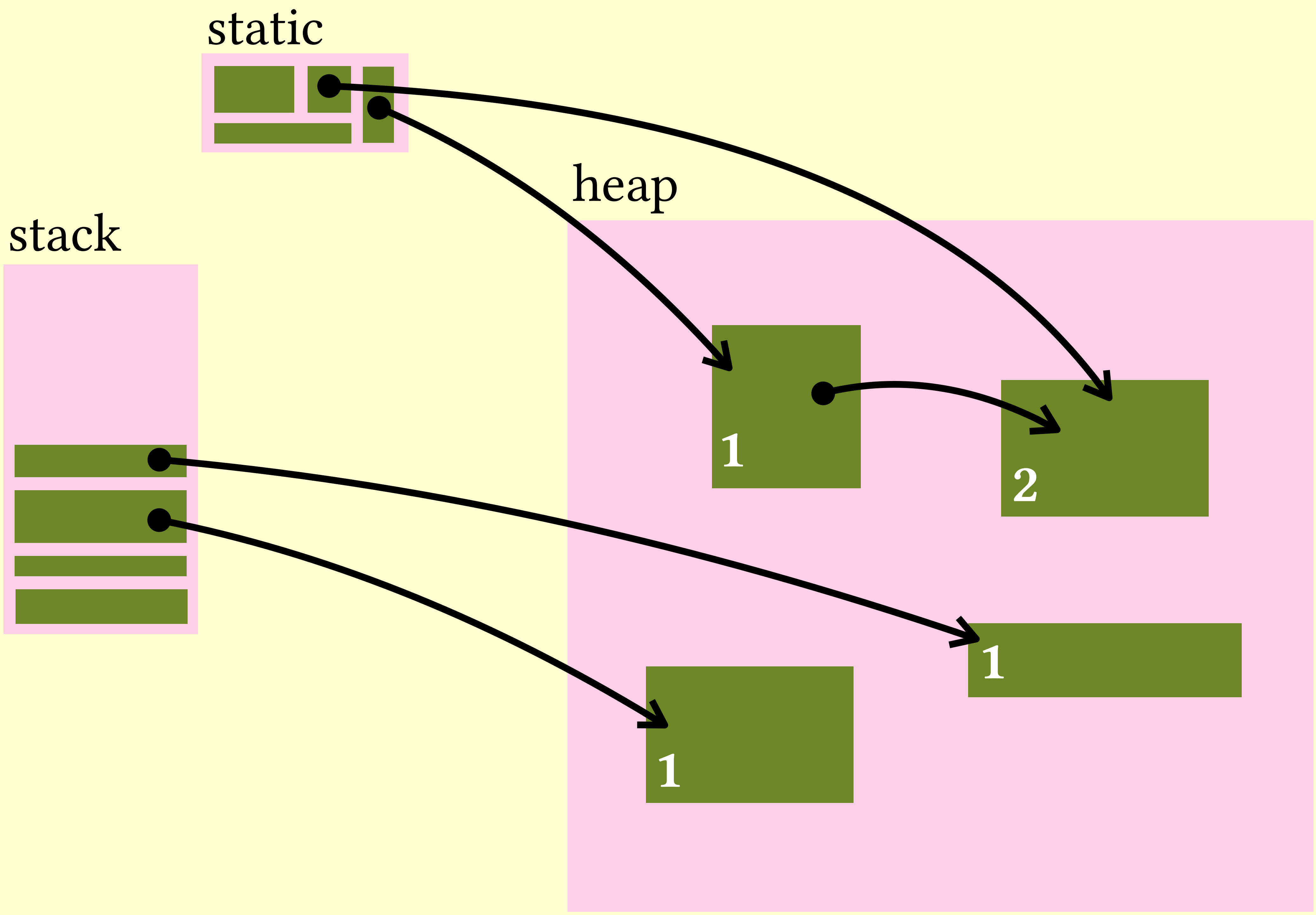
Reference counting



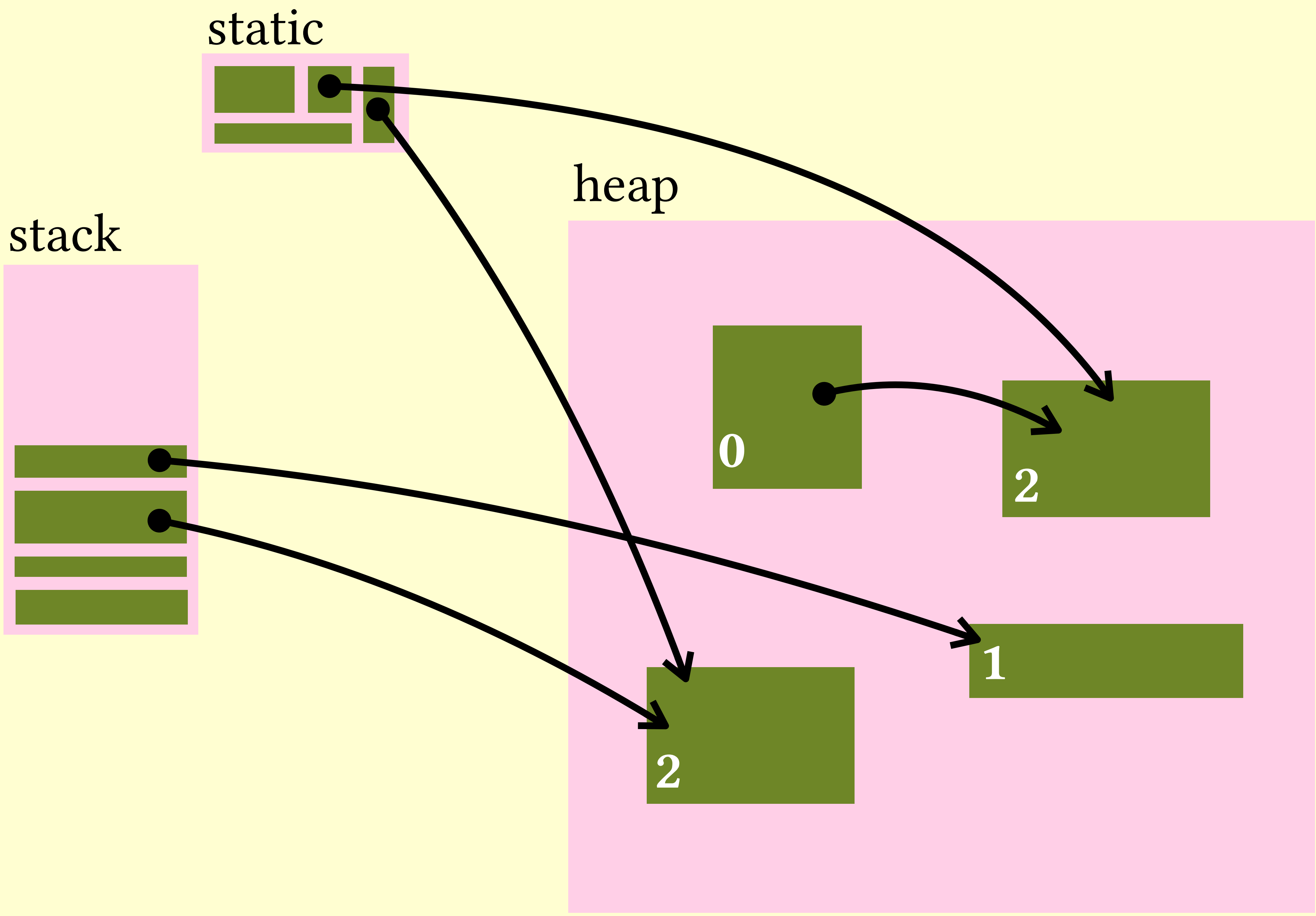
Reference counting



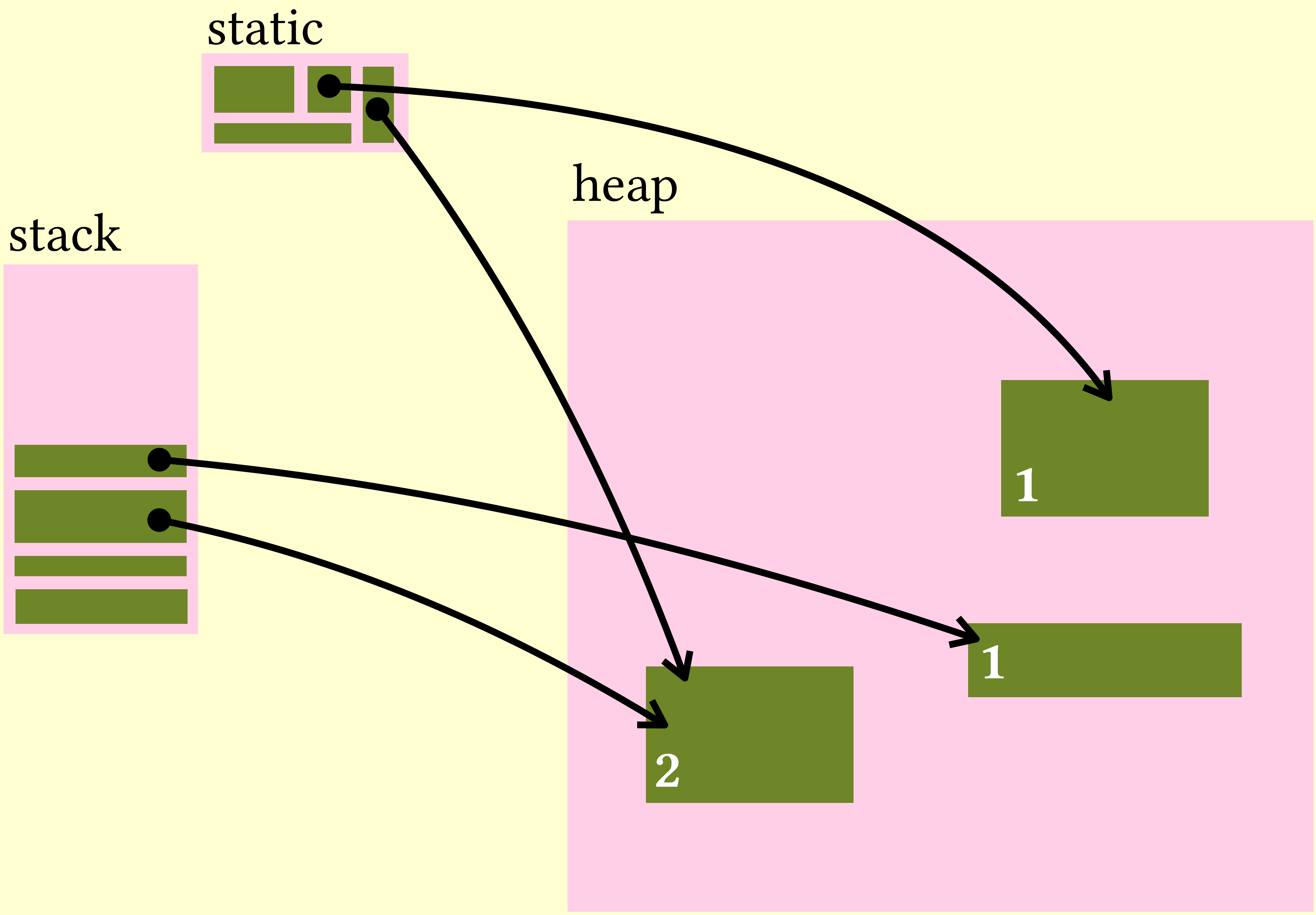
Reference counting



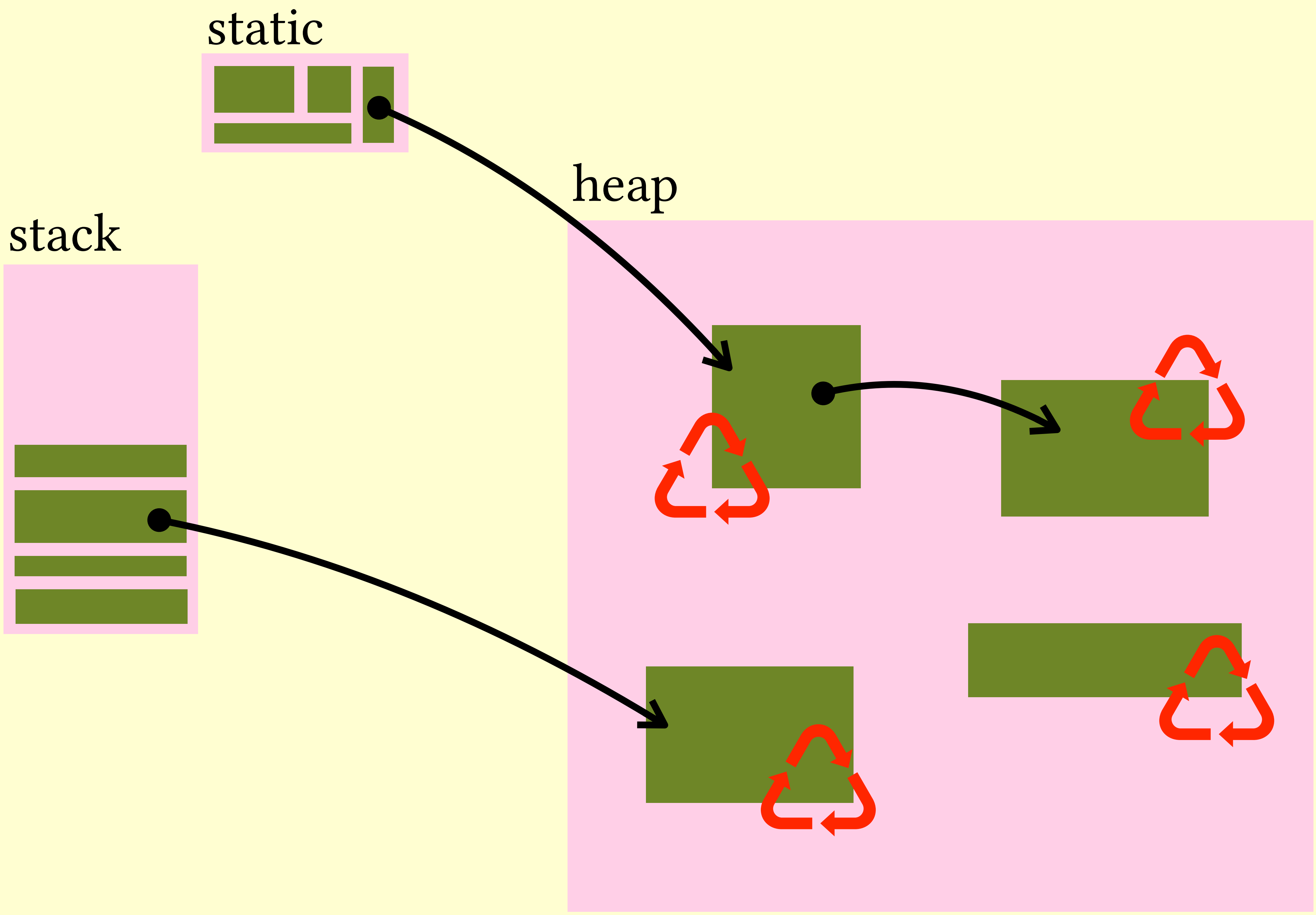
Reference counting



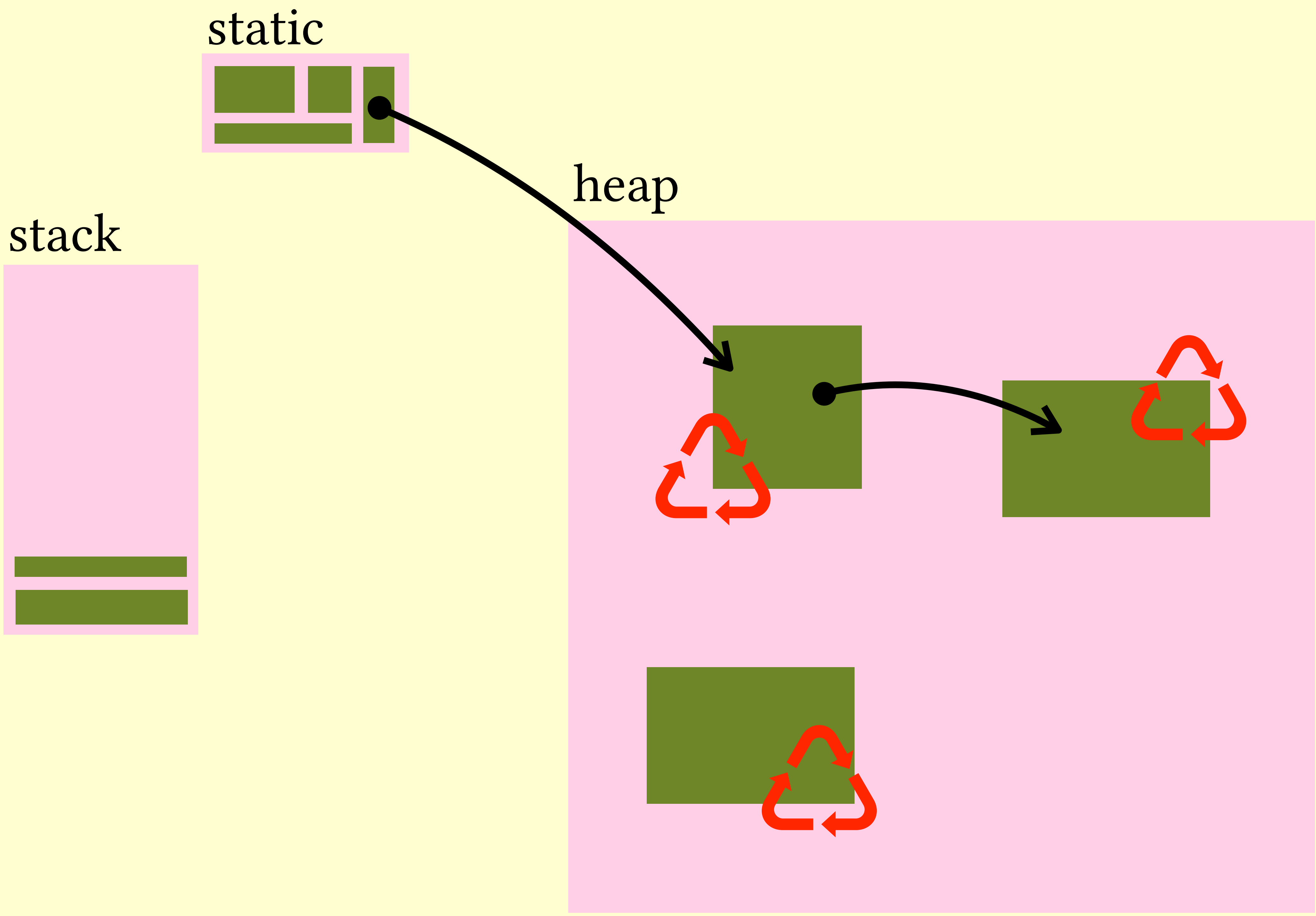
Reference counting



Tracing GC



Tracing GC



Comparison

- **Question.** What are the advantages/disadvantages of Reference counting vs. Tracing GC?

- Common problem: pointer arithmetic.

```
int *p = malloc(sizeof(int));
p += 1000;
```

Do we still have a reference?

- Common problem: imprecision.

```
int *p = malloc(sizeof(int));
*p = 42;
int *q = malloc(sizeof(int));
return;
```

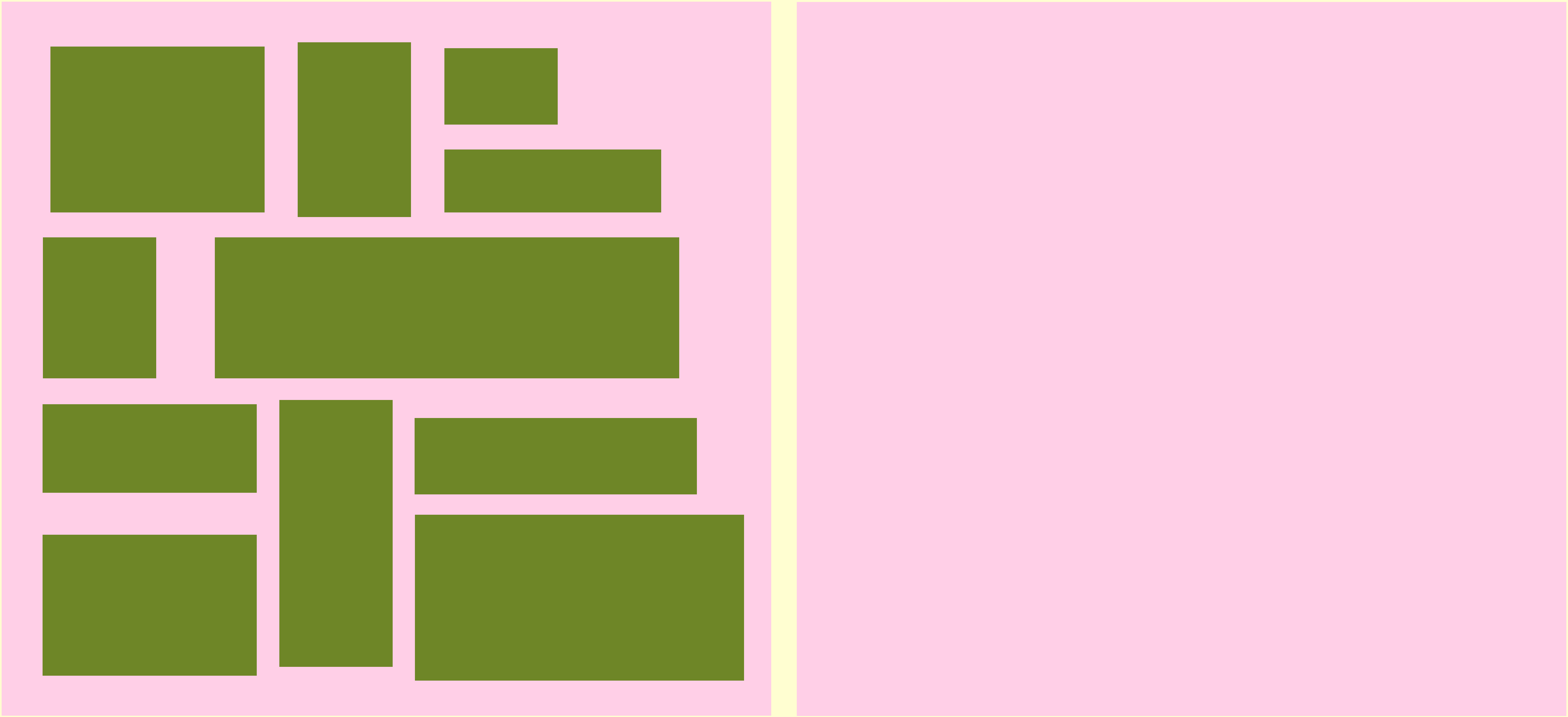
Ref. counting will reclaim `p` here

Tracing GC will reclaim `p` here

But we could reclaim `p` here

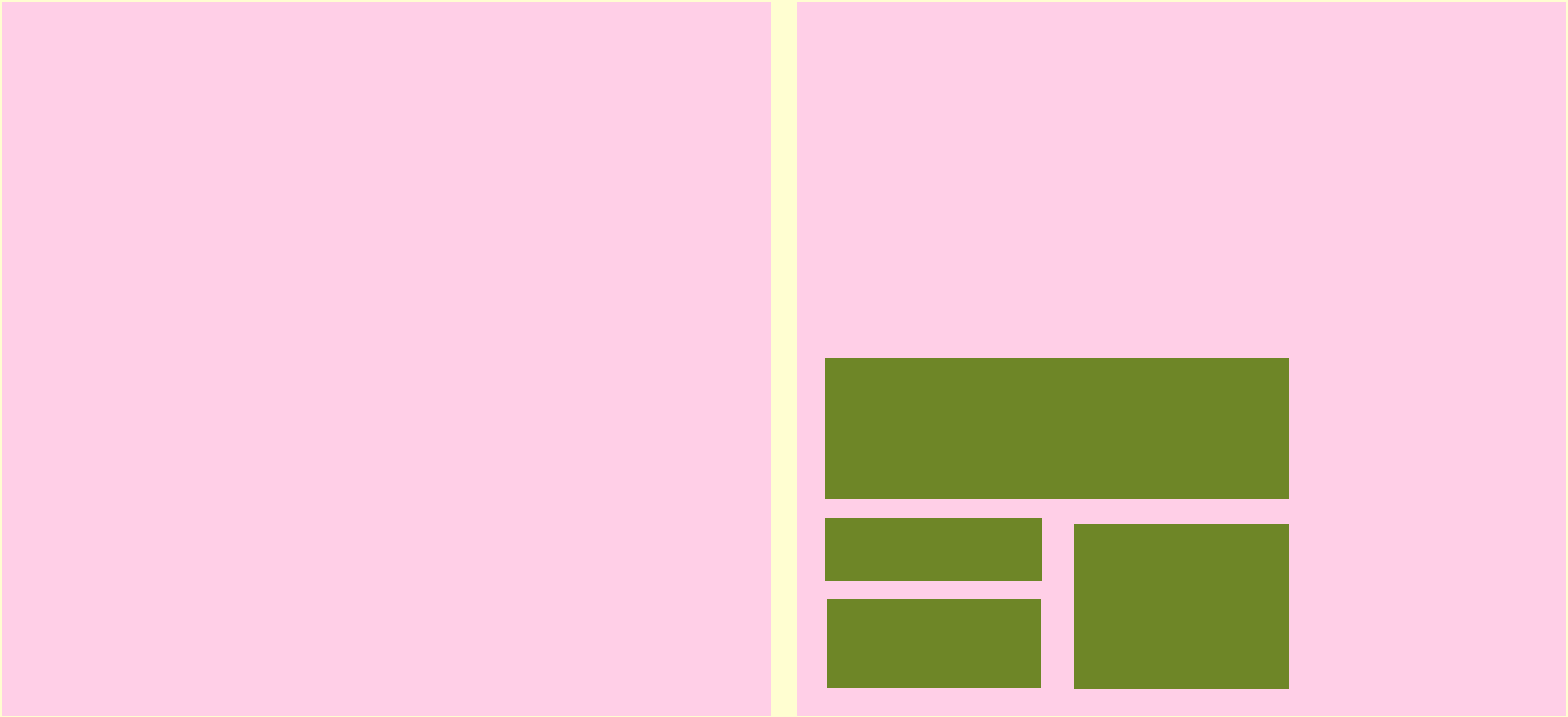
Other GCs

- Copying GC



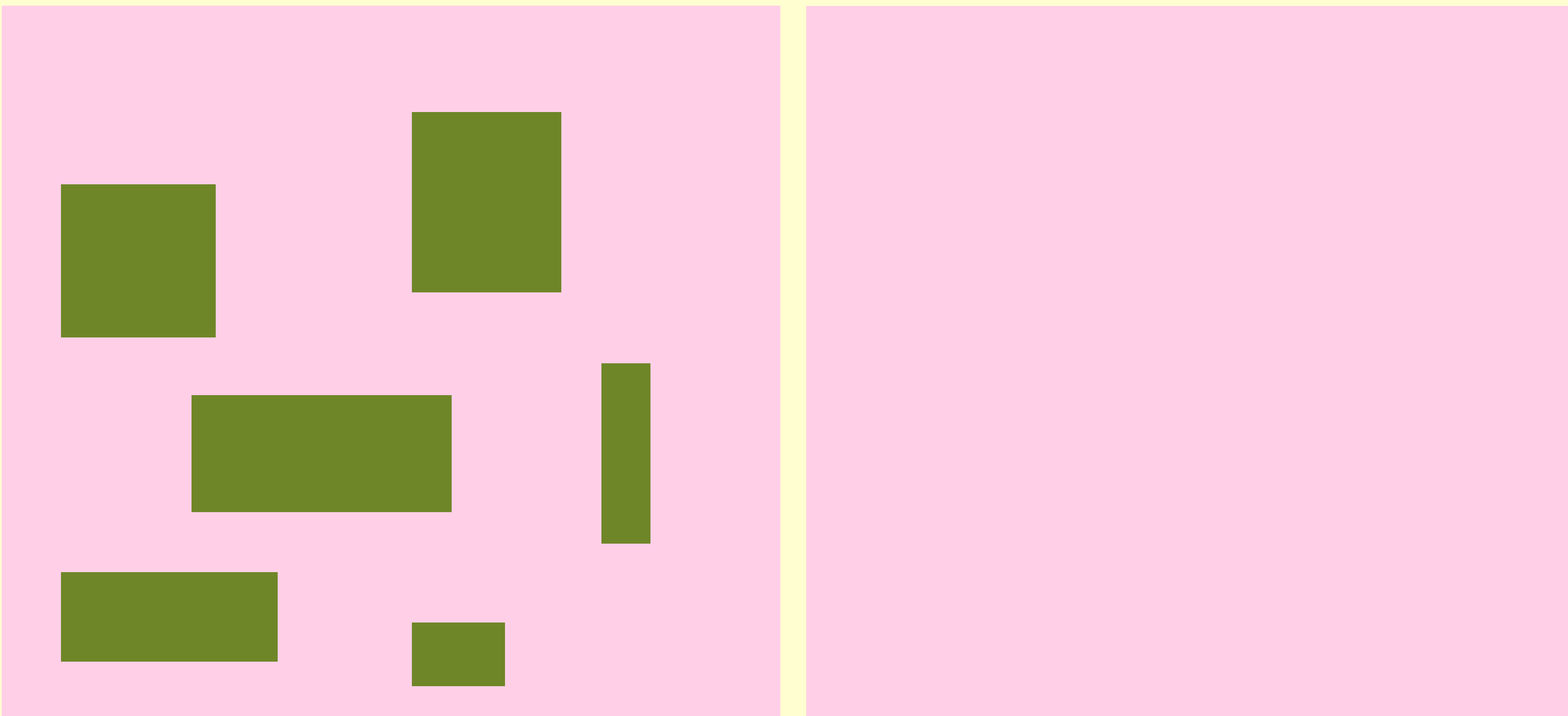
Other GCs

- Copying GC



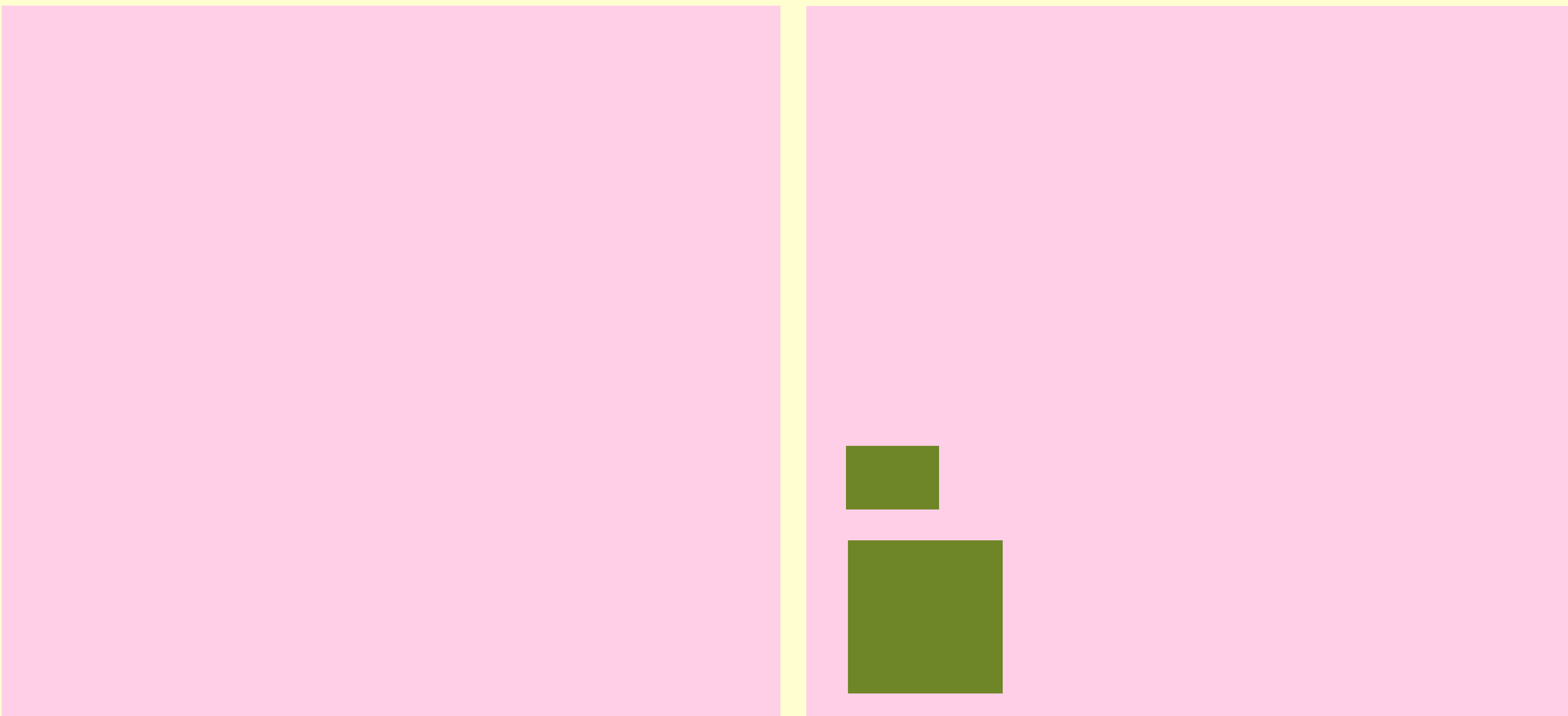
Other GCs

- Generational GC



Other GCs

- Generational GC



Lecture outline

- ✓ Dynamic memory allocation (`malloc` and `free`)
- ✓ Garbage collection

Summary

- Memory regions: static, stack, heap.
- Memory allocation should: be fast, be responsive, avoid fragmentation, be aware of spatial locality, use little memory, and be concurrency-safe.
- Heap deallocation is manual (`free`) or automatic (GC).
- GC should: be fast, be responsive, be aware of spatial locality, use little memory, be concurrency-safe, be sound (don't collect non-garbage), and be precise (collect all garbage).