Generational GC

The Generational Hypothesis

Hypothesis: most objects "die young"

- i.e., they are only needed for a short time, and can be collected soon

- A great example of empirical systems work
- Found to hold for programs in practice
  - Regardless of programming style! (functional, imperative, OO, etc.)

The Generational Hypothesis

```
(define (euclidian-norm v)
  (sqrt (foldl + 0 (map square v))))
```

- `square` produces floats which are only used until the addition
- `map` produces a whole list which is only used until we fold over it
- `+` produce a whole lot of intermediate floats which are only used for the next addition

The Generational Hypothesis

So, how can we use this to guide GC design?

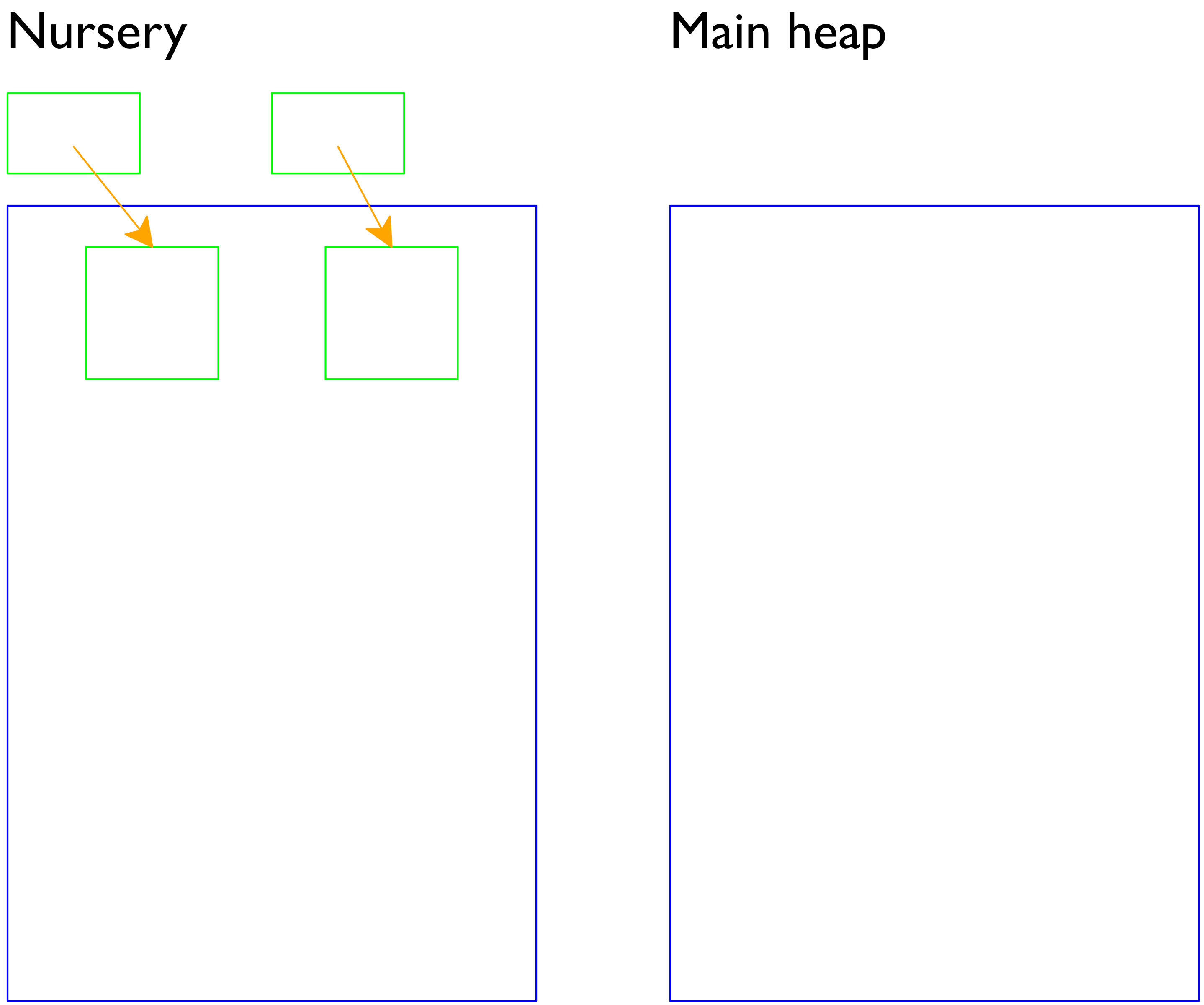
- Young objects have a high chance of being garbage
  - But they're only a small portion of all objects
  - So if we focus our efforts on them, can free a lot of space in not much time!
- Have a separate region for young objects: nursery
  - Allocate new objects there
  - When we run out of space, collect there (only!)
  - When objects get old enough, so less chance of dying, migrate them to the main heap
  - When we run out of space in the main heap, only then do we GC it

Generational Garbage Collection

Ok, so how to GC?

- Clearly want copying GC for nursery
    - Cost proportional to # of live objects
        - Which should be few in nursery
    - Use the main heap as to-space!
        - And don't swap spaces! Always in the same direction
        - Can use the whole heap with a copying GC!
    - Survivors get copied out of the nursery naturally!
        - Surviving a GC = old enough to move out
- What about main heap?
    - Doesn't matter as much (stay tuned)

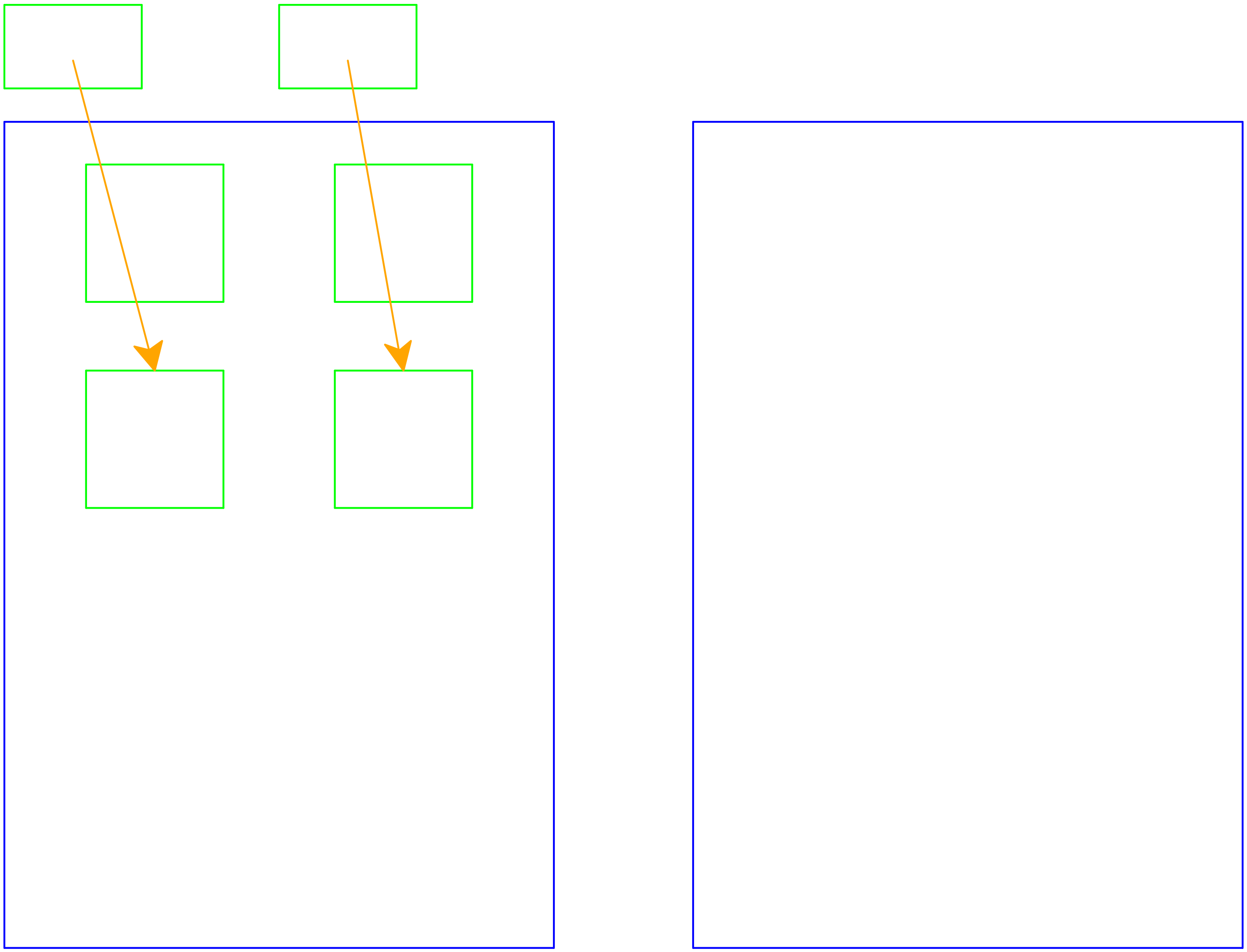
Generational Garbage Collection



Generational Garbage Collection

Nursery

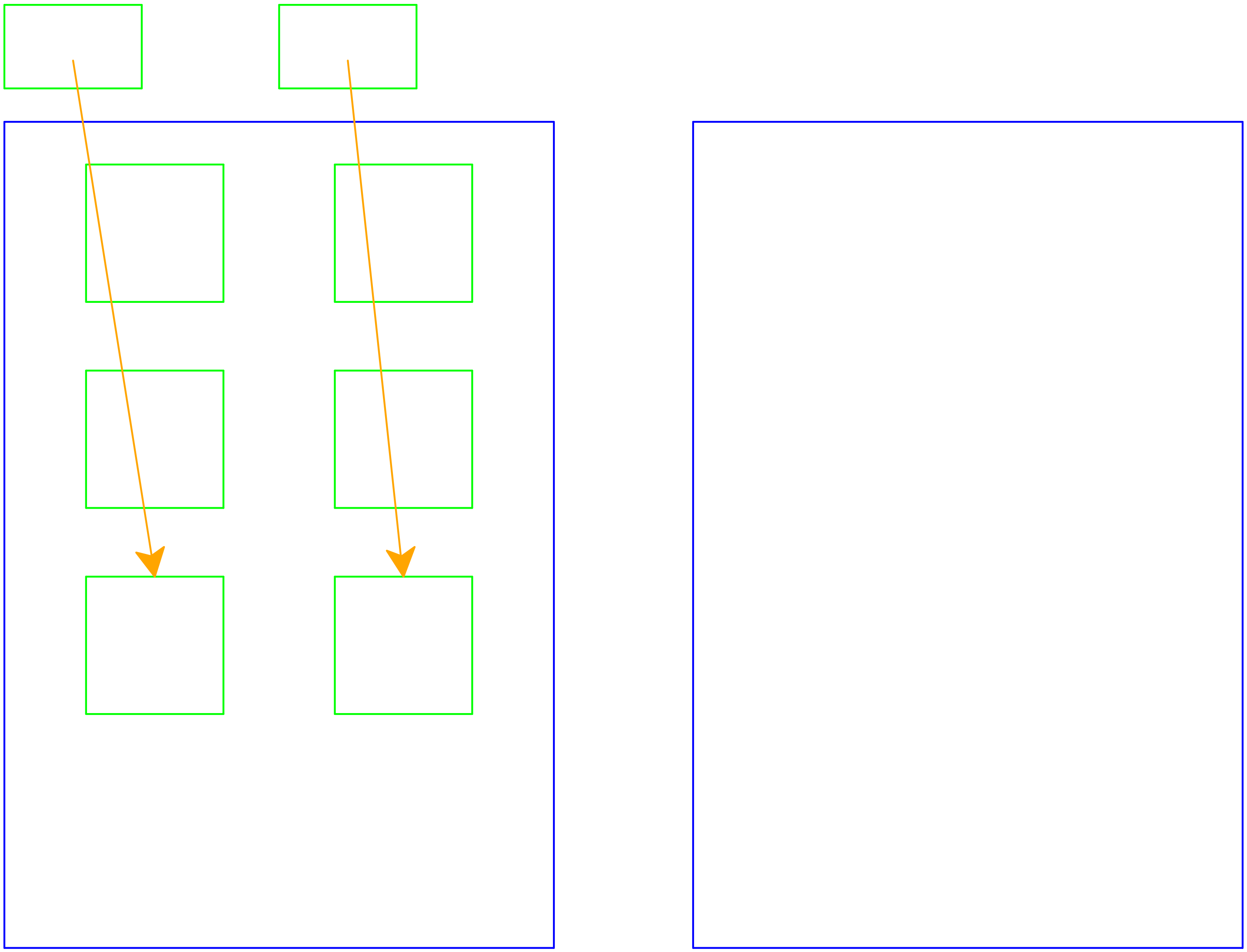
Main heap



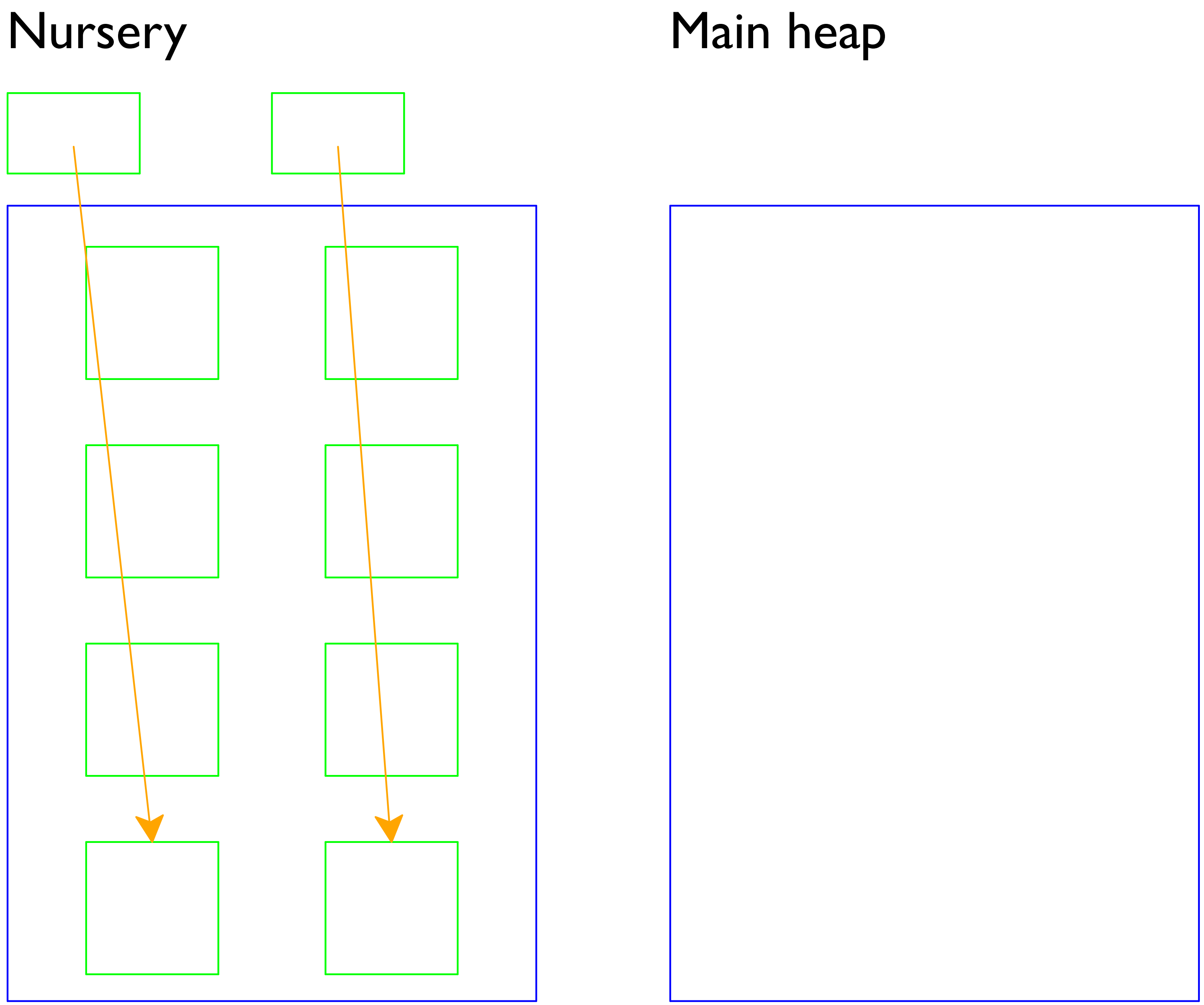
Generational Garbage Collection

Nursery

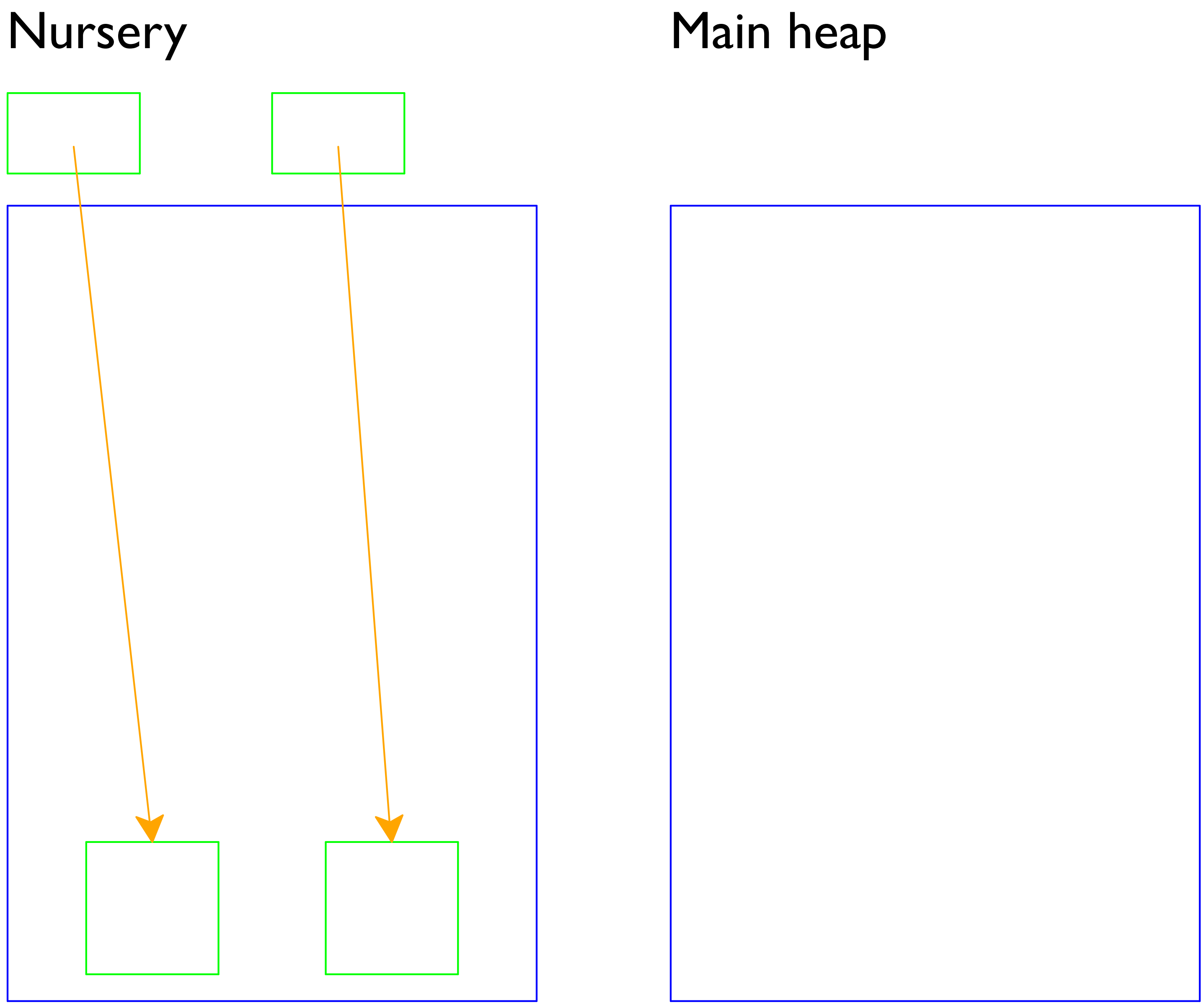
Main heap



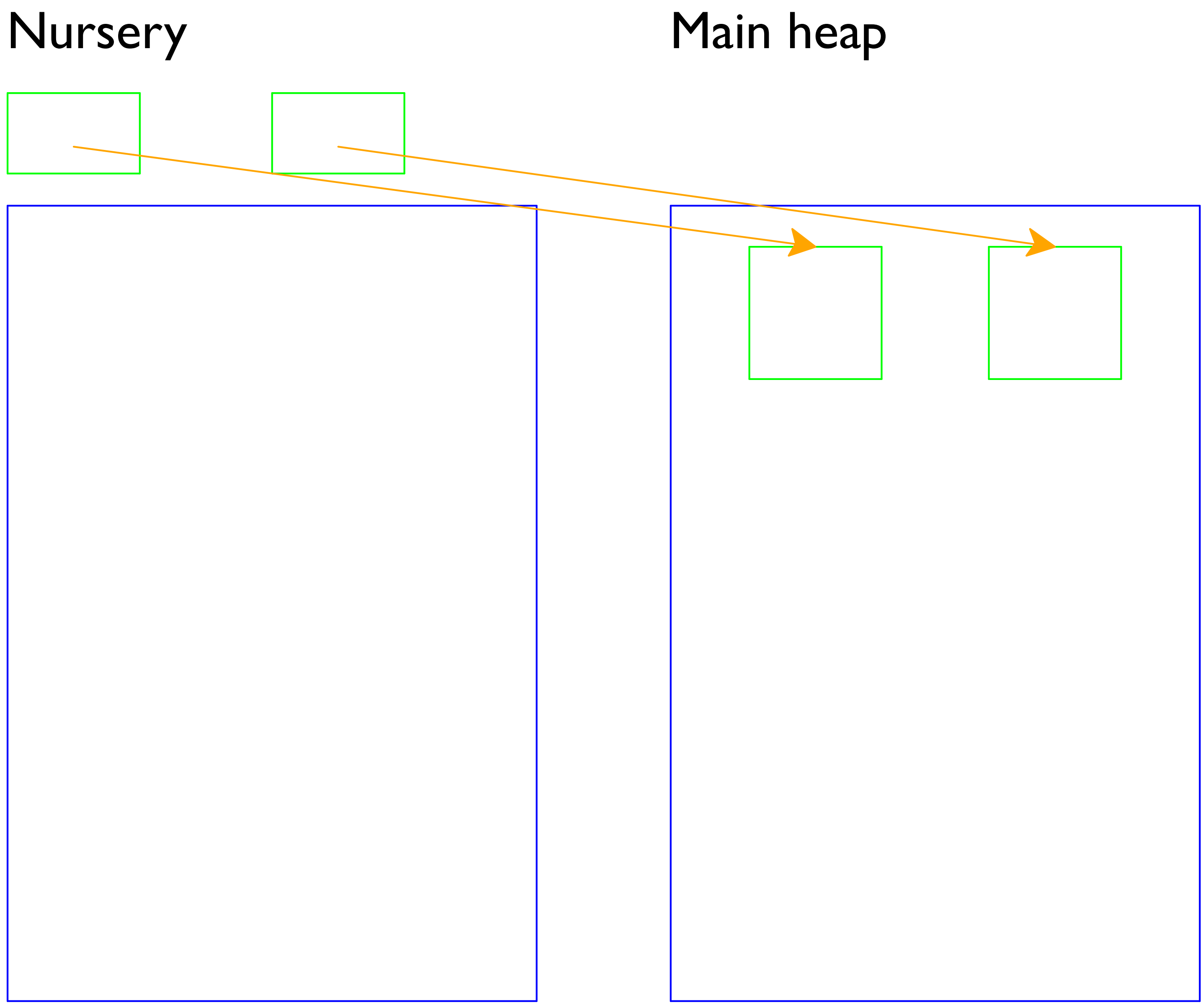
Generational Garbage Collection



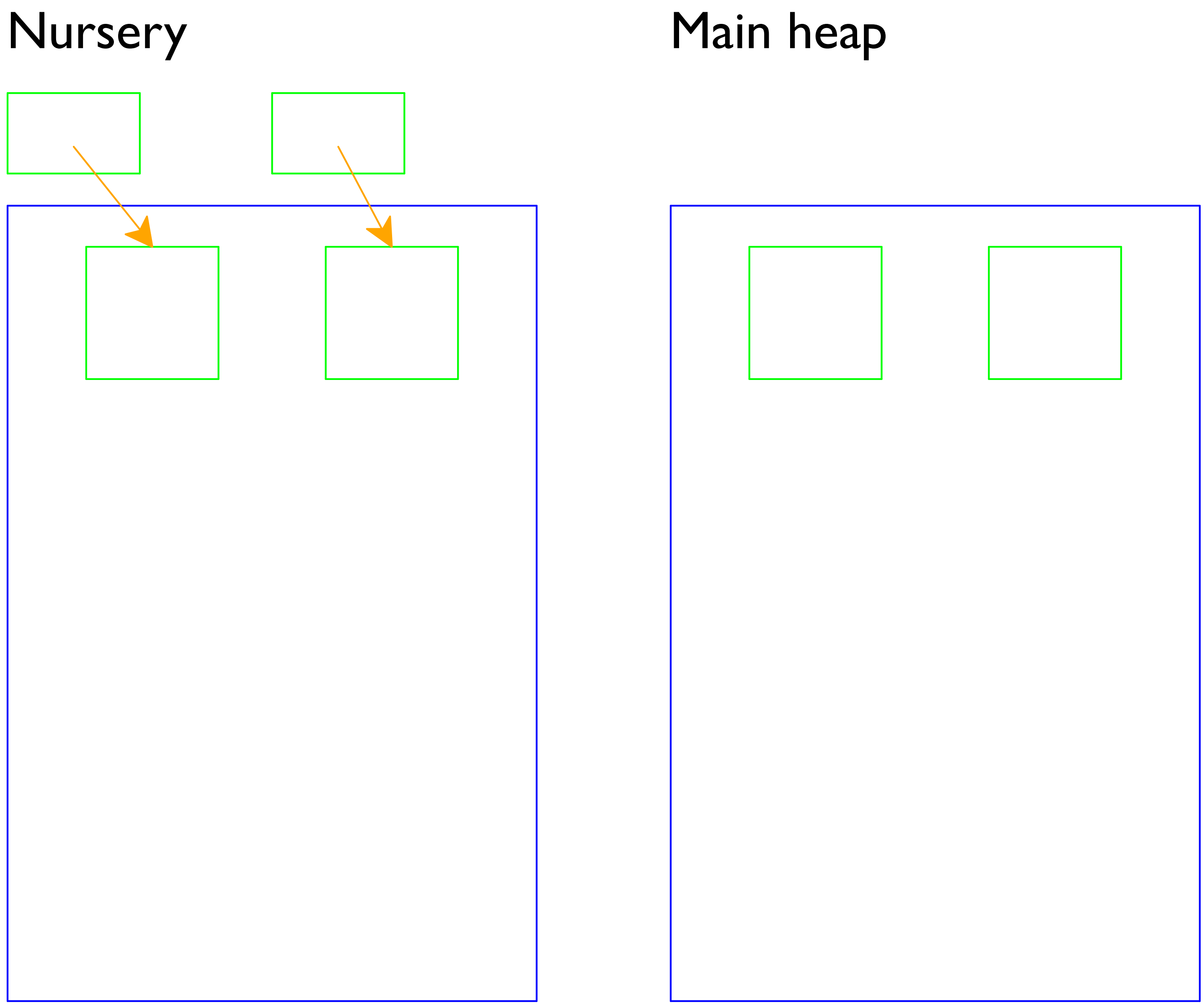
Generational Garbage Collection



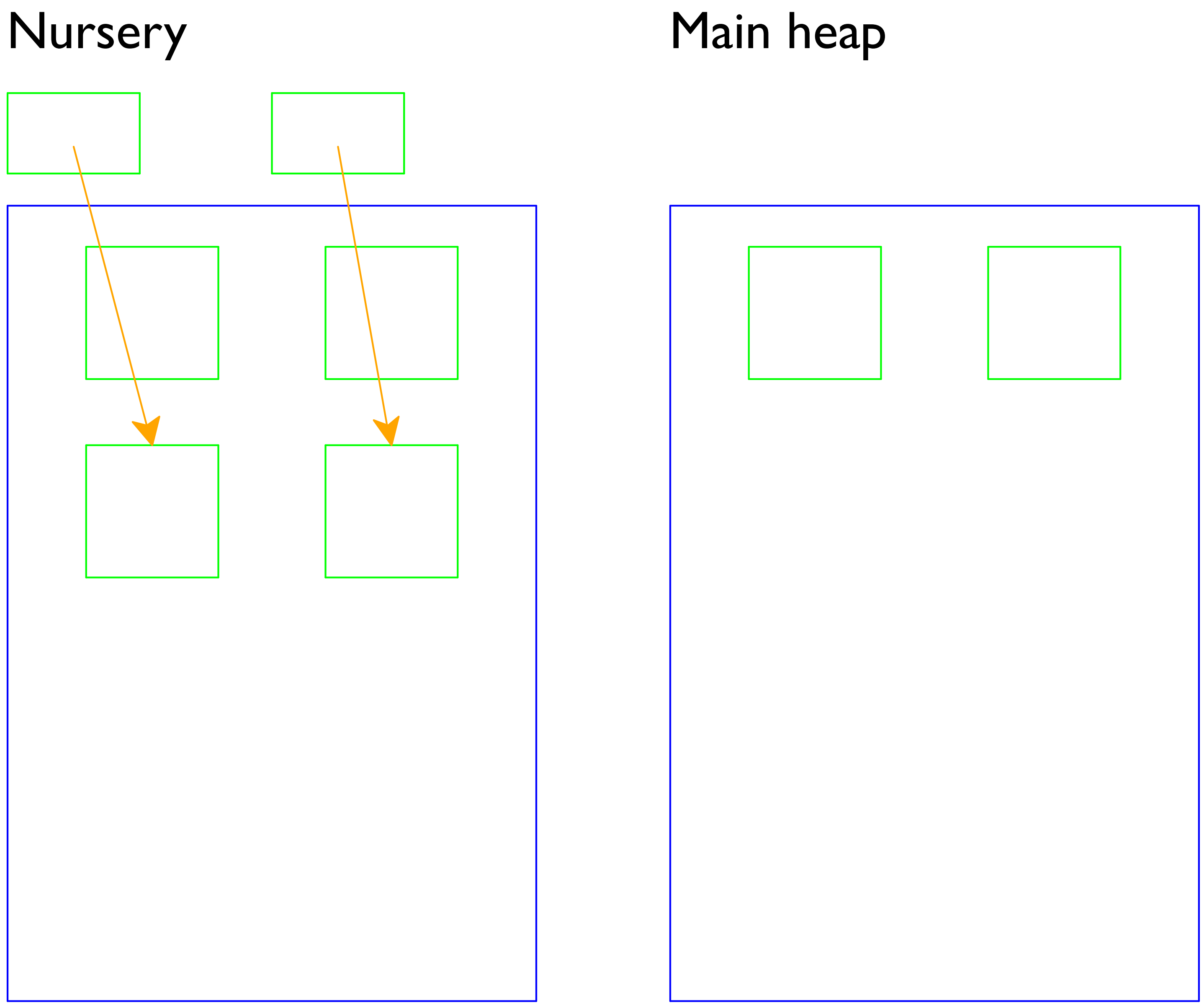
Generational Garbage Collection



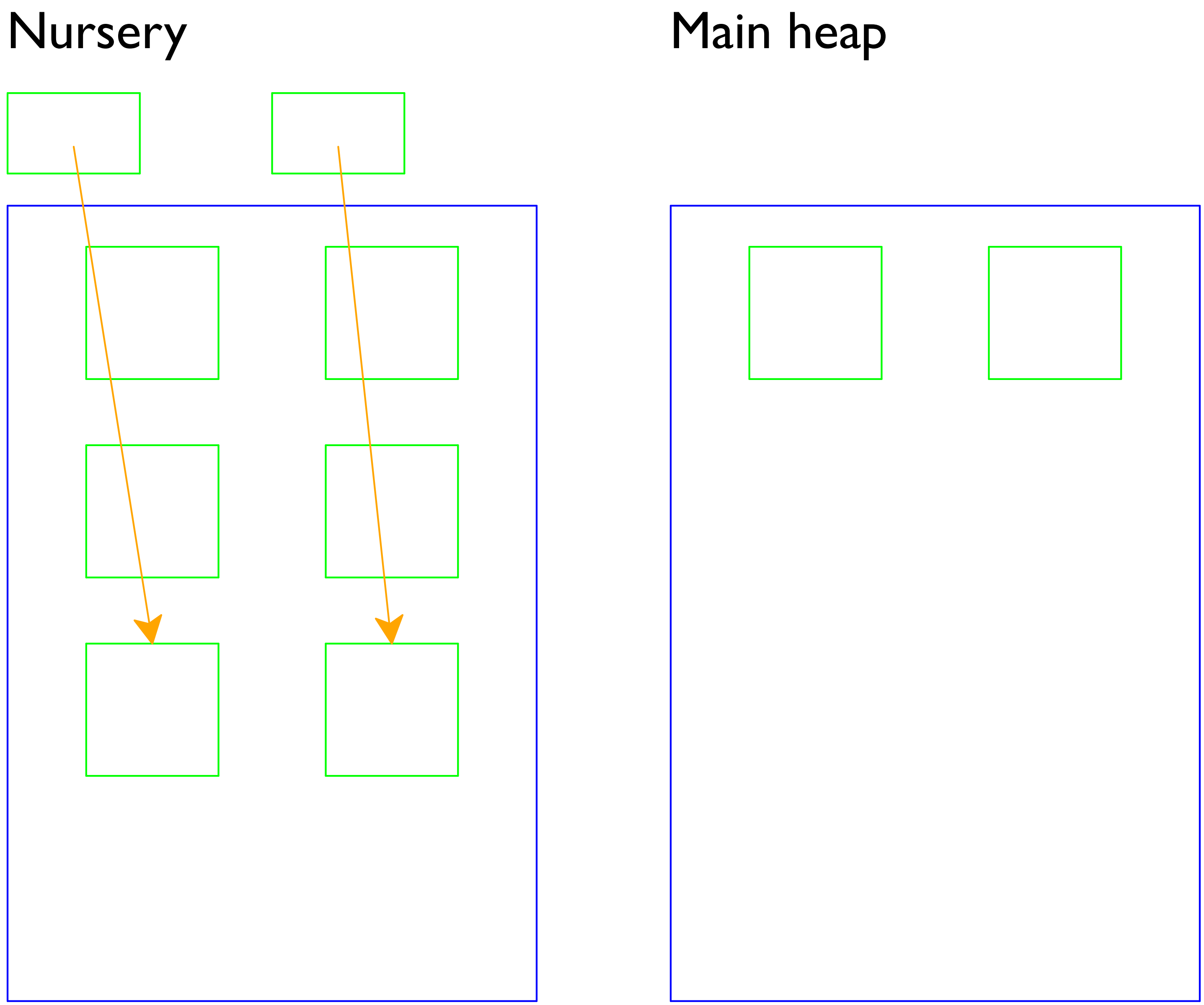
Generational Garbage Collection



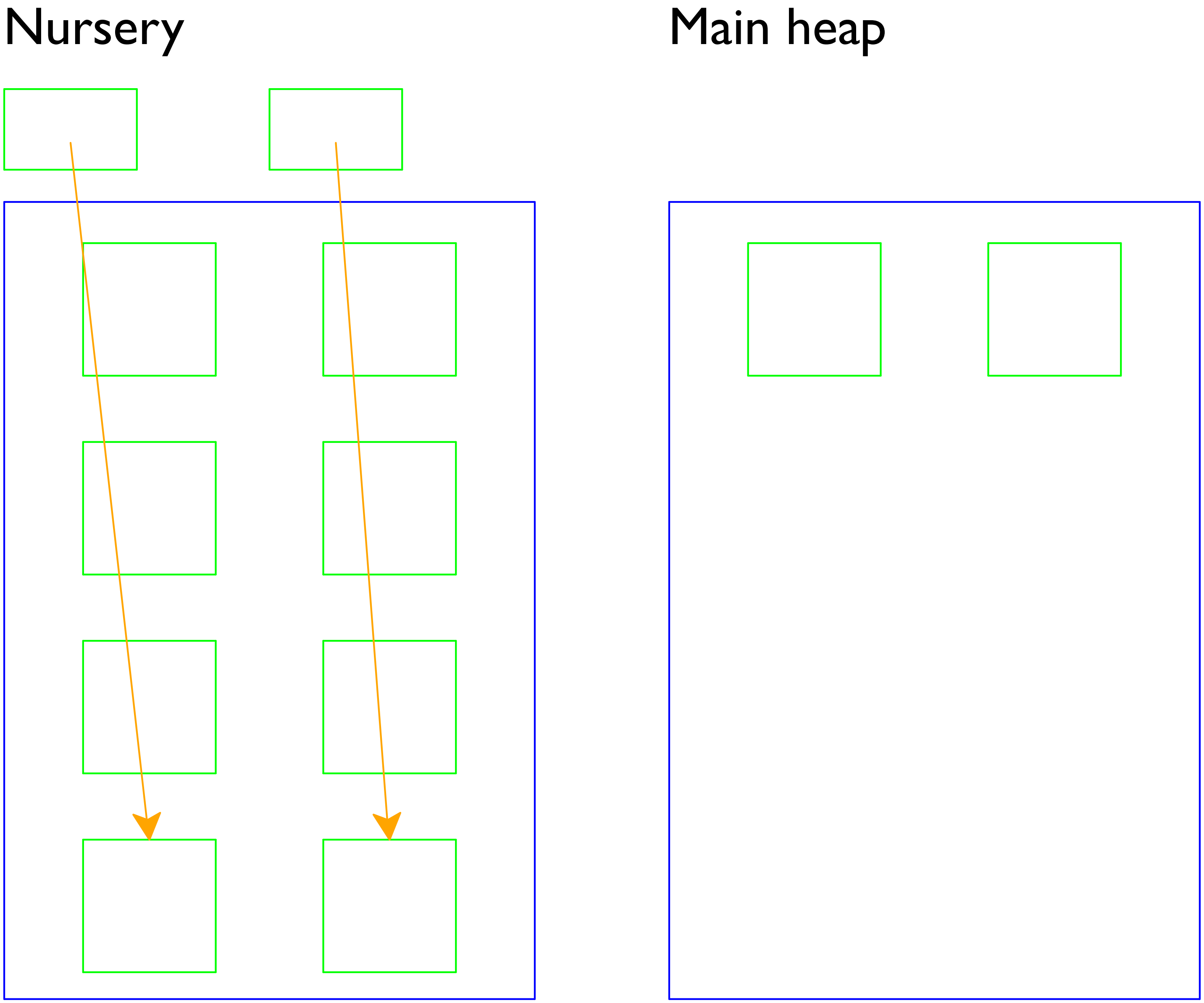
Generational Garbage Collection



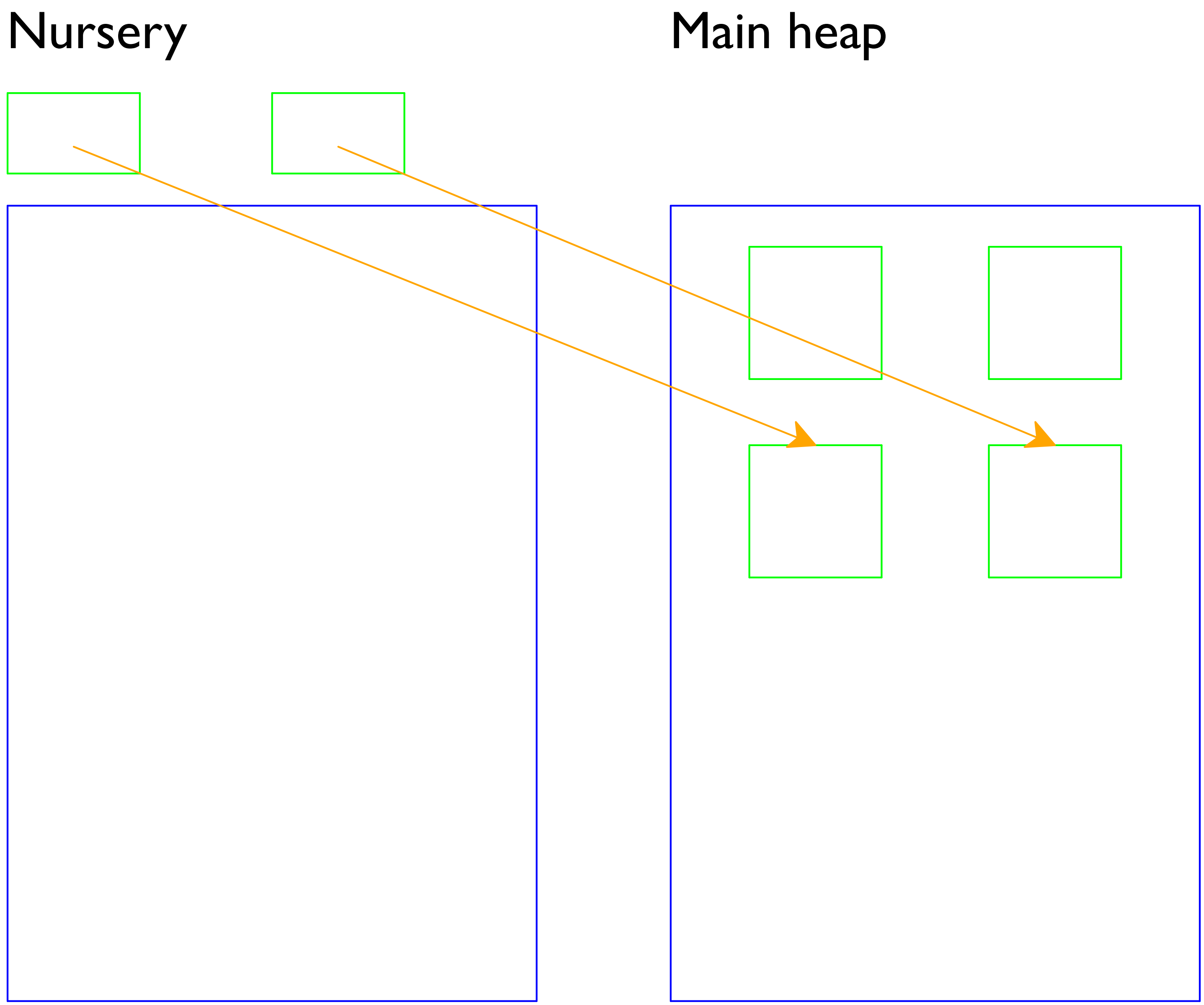
Generational Garbage Collection



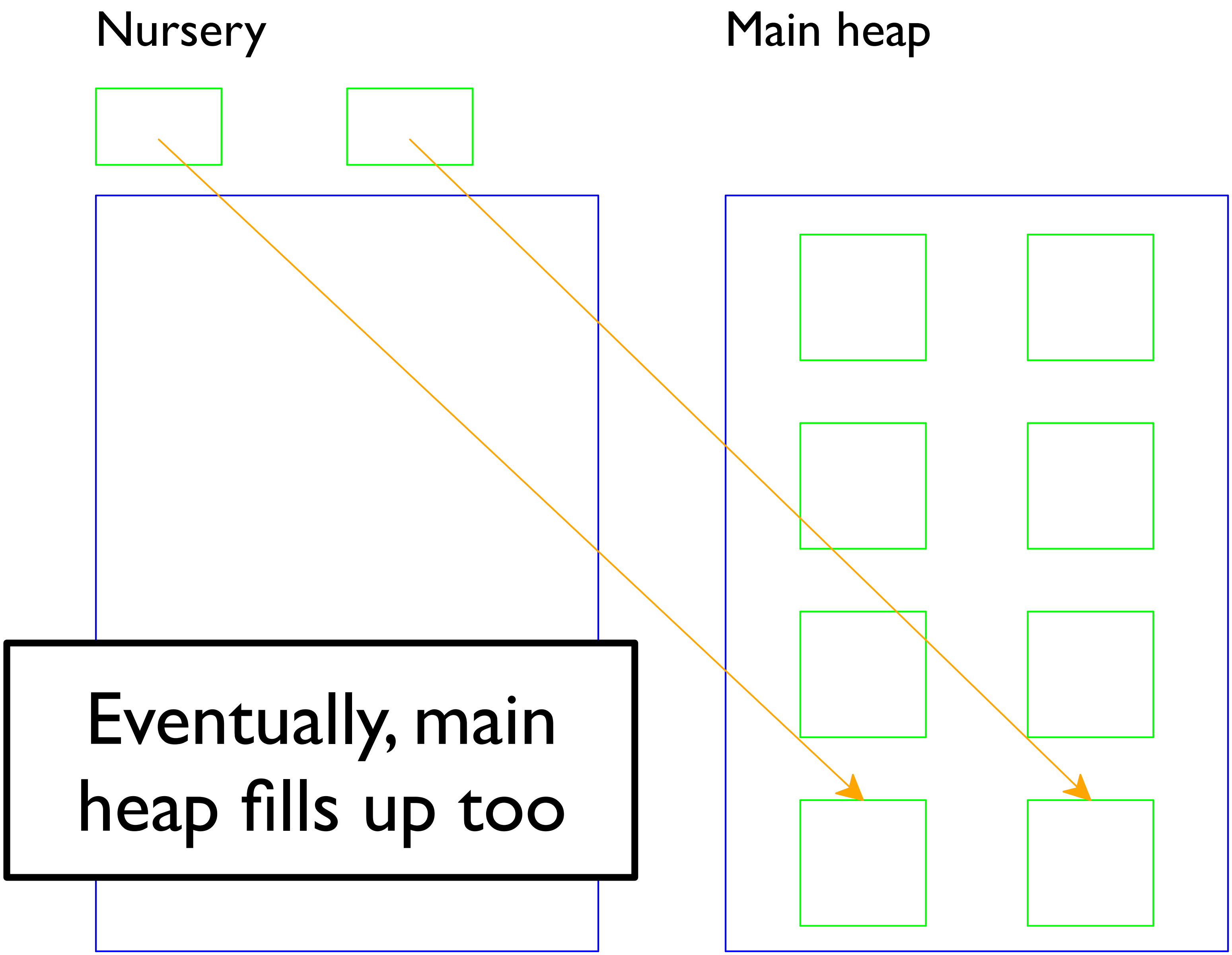
Generational Garbage Collection



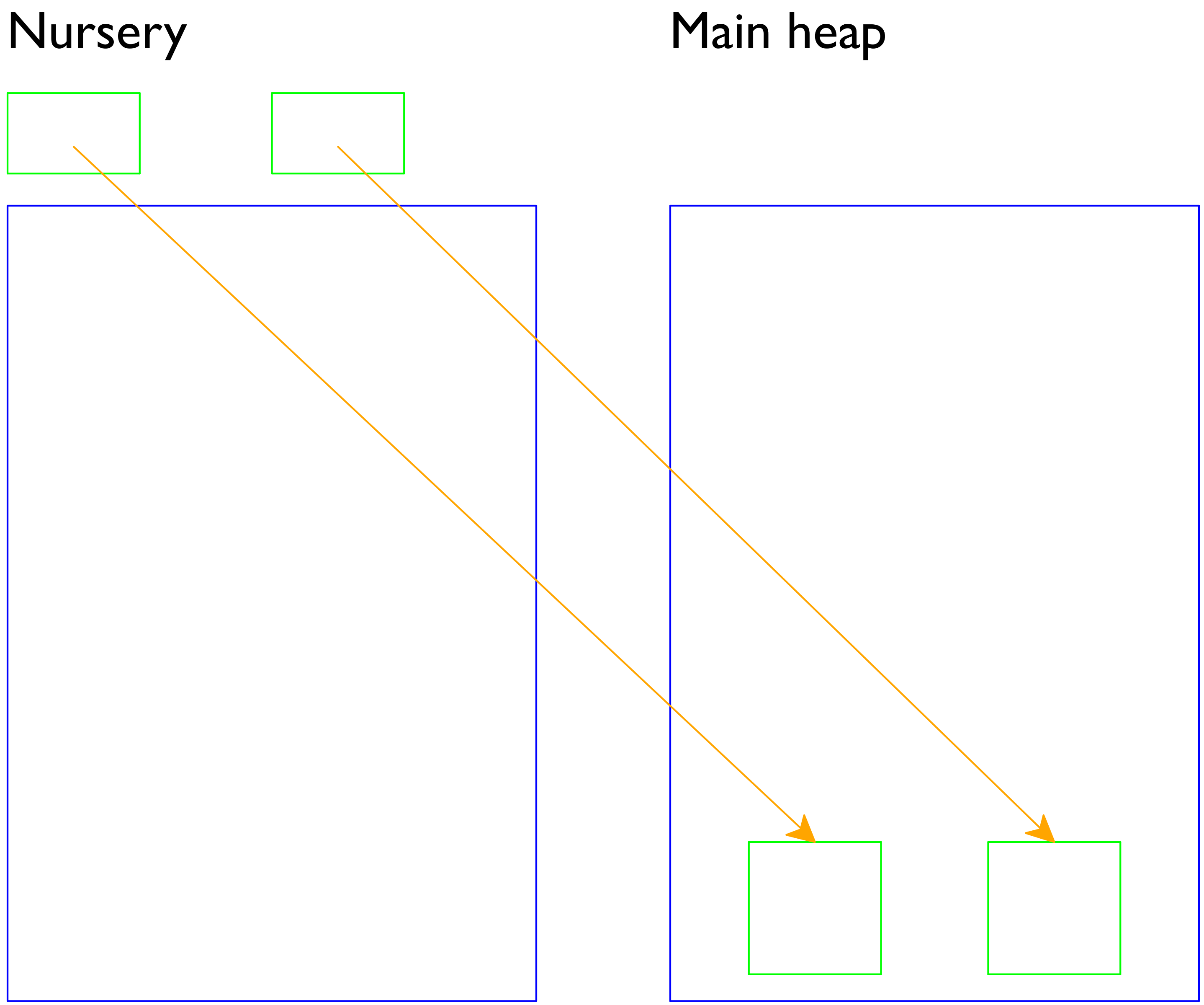
Generational Garbage Collection



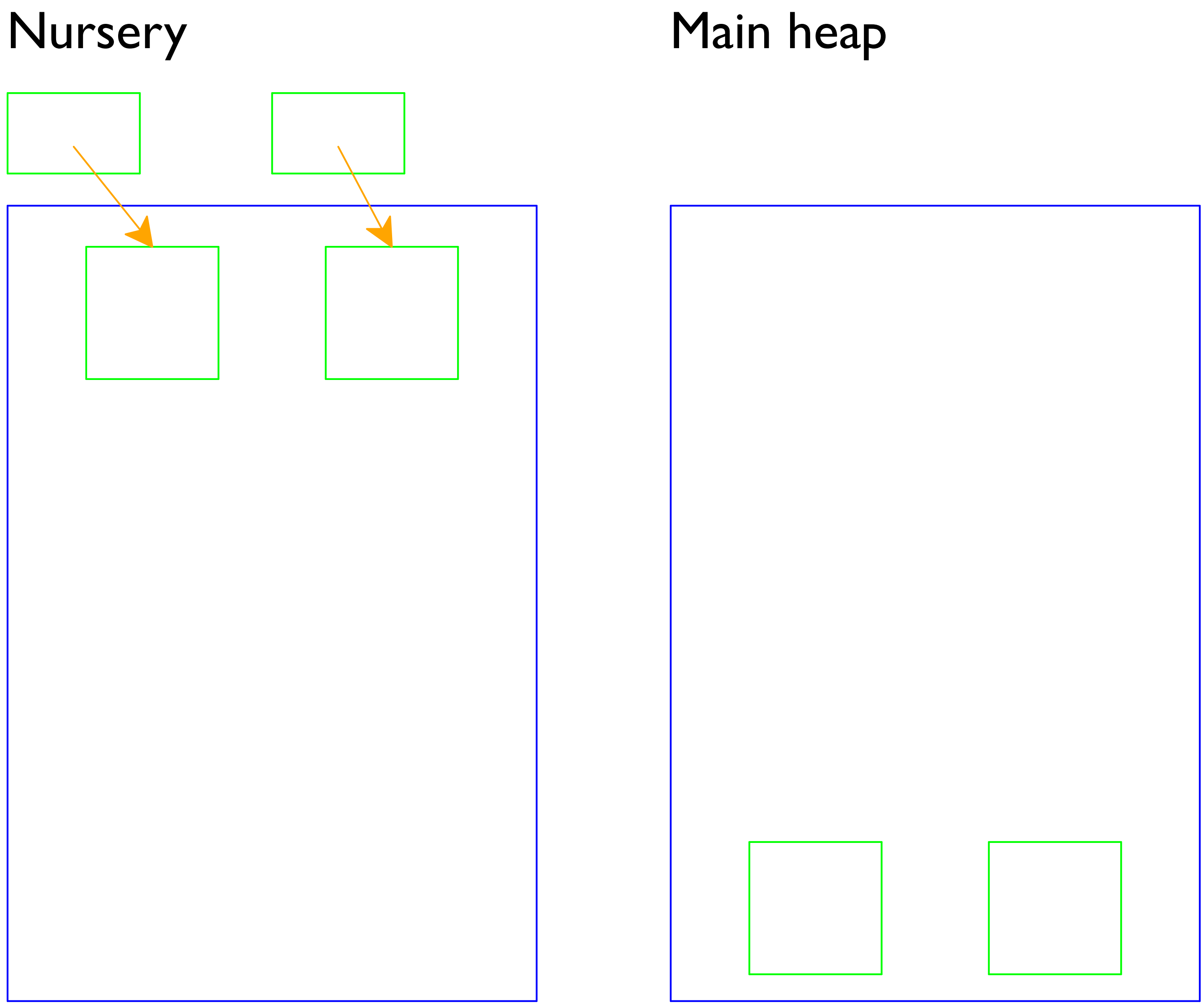
Generational Garbage Collection



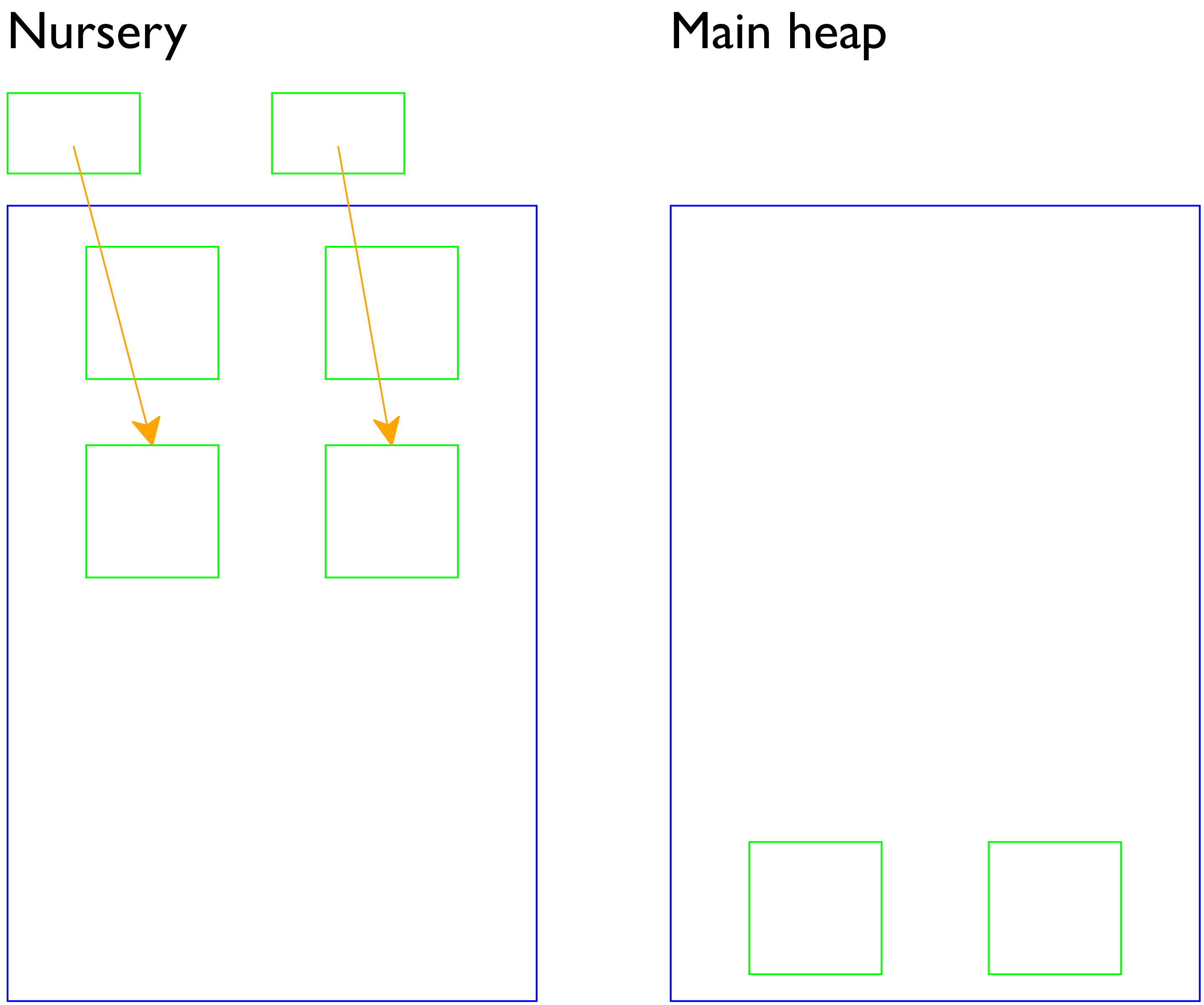
Generational Garbage Collection



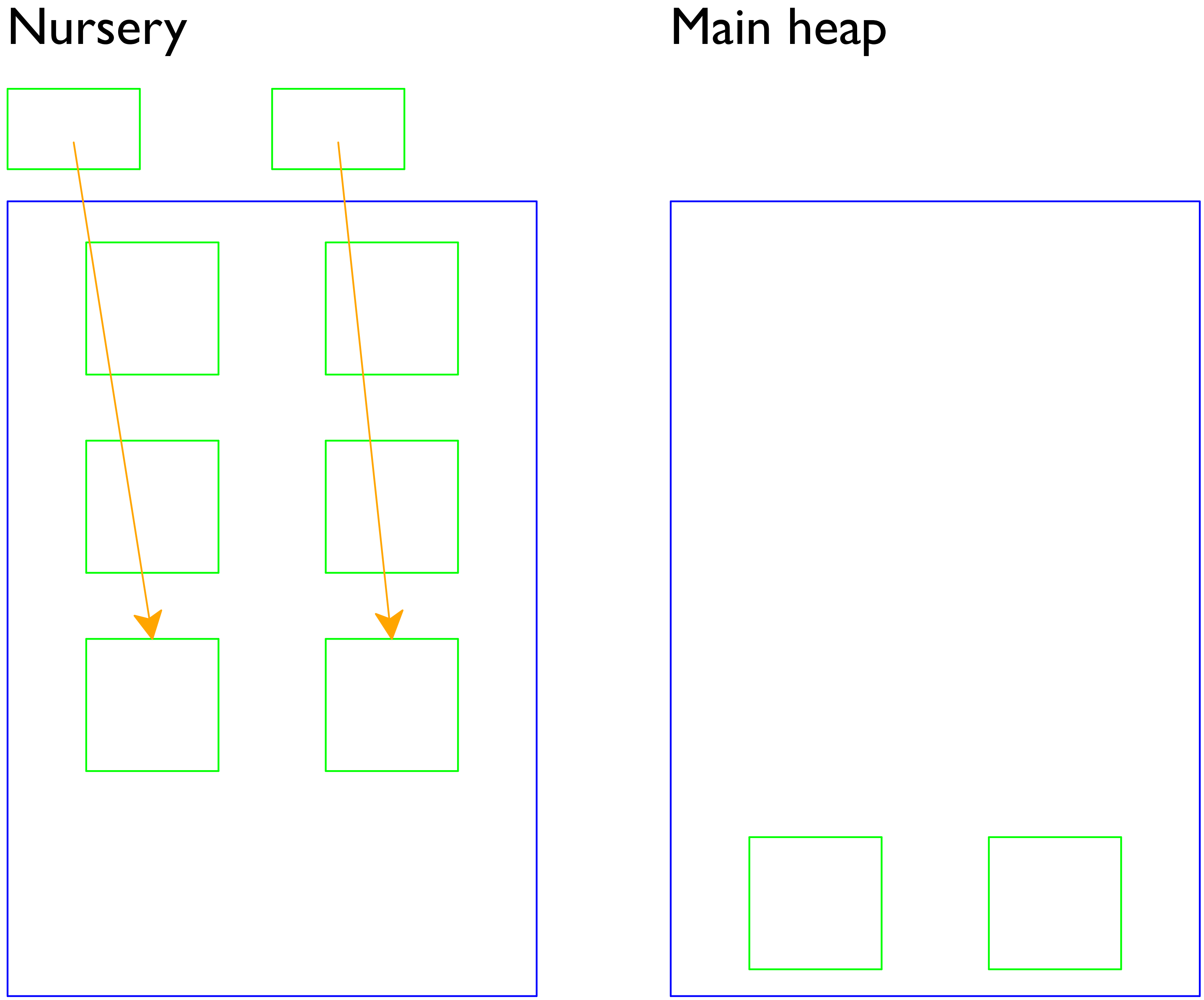
Generational Garbage Collection



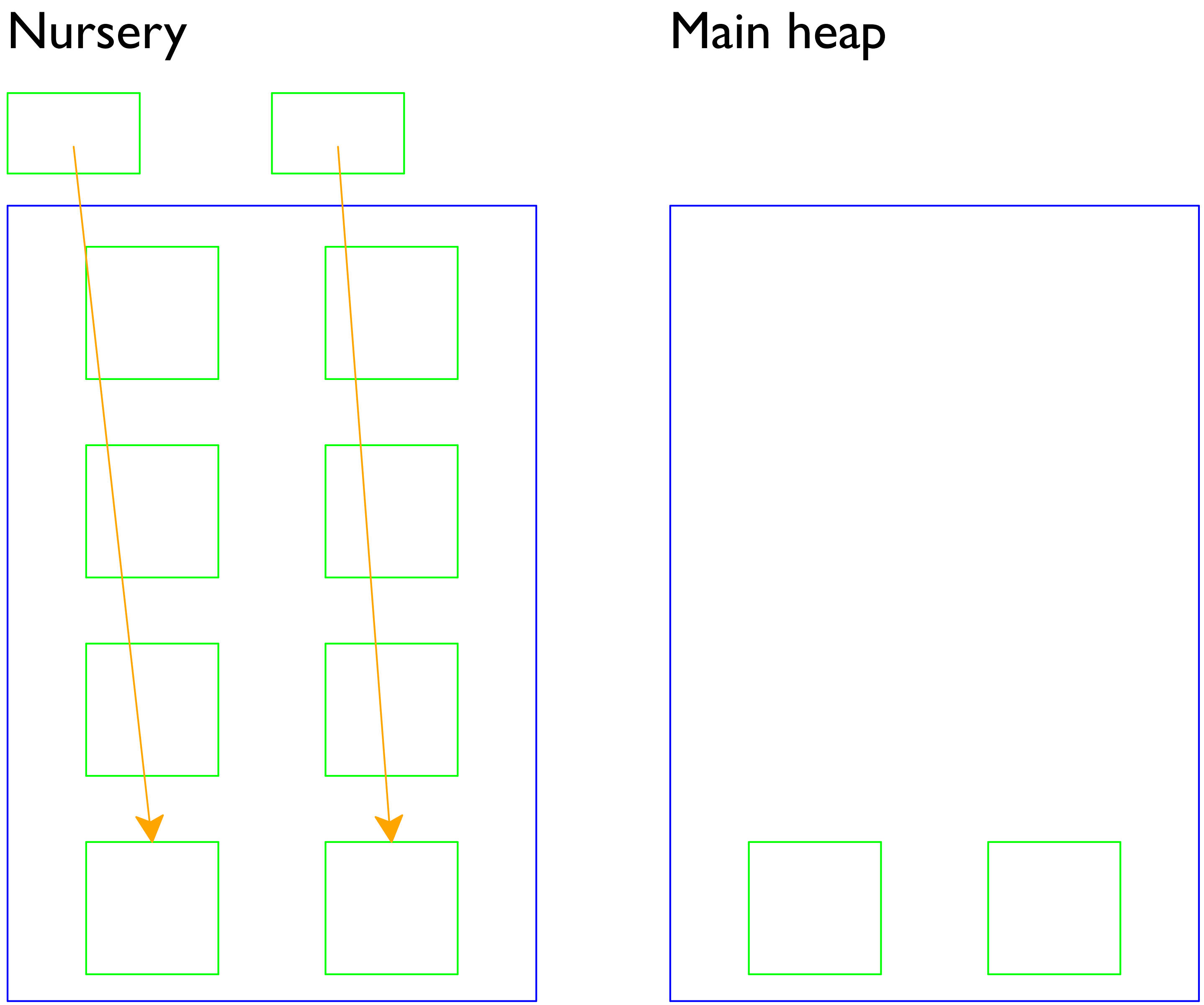
Generational Garbage Collection



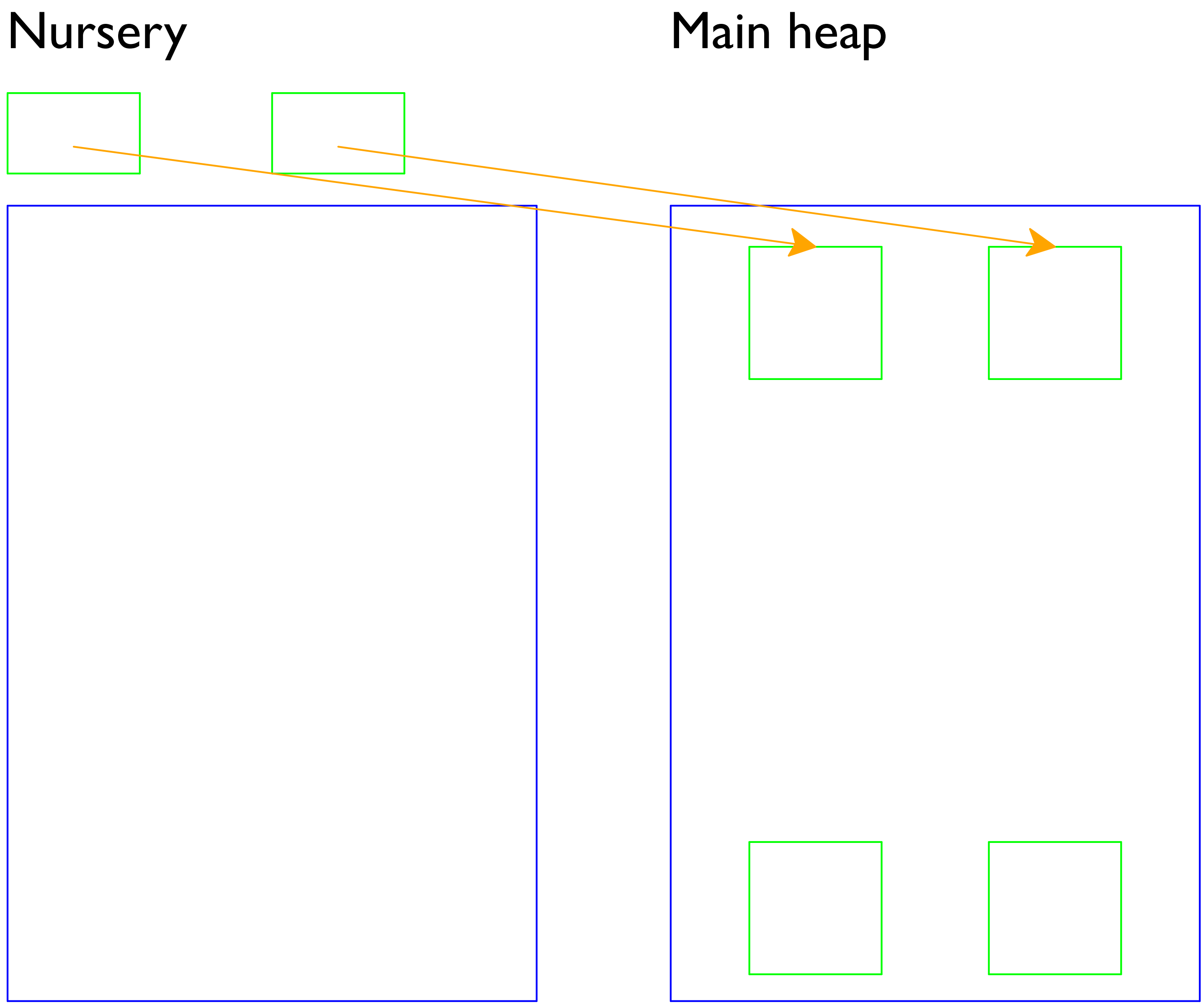
Generational Garbage Collection



Generational Garbage Collection



Generational Garbage Collection



Generational Garbage Collection

If the generational hypothesis holds

- Then we'll collect the nursery pretty often
- But only collect the main heap rarely

Collecting the nursery is cheap

- Can keep it fairly small, so not many objects
- Will be mostly dead objects, and copying GC has cost proportional to live objects!
- → can afford to do it often; pauses will be short

Collecting the main heap is slow

- It will be large; needs to hold all our data
- A lot of it will be live, and will need to be marked/copied
- → but that's ok, don't do it very often

The Snag

- When we GC the nursery, what do we use as roots?
- Want to use registers, globals, etc. Sure.
- But we may also have pointers to nursery objects from the main heap!
  - These nursery objects may be live too!
- Only matters if the object in the main heap is live
  - But can't know unless we mark all of main heap
  - Which is what we were trying to avoid in the first place!

The Solution

- Track pointers from main heap to nursery
  - And just assume they're live, conservative
  - So treat them as roots
- How do we keep track? Write barriers
  - Can only get a pointer from main heap to nursery by using mutation
  - Can't have an old object point to a new one naturally
    - Latter wasn't around when the old one was allocated!
  - So whenever we mutate, we look out for that case, and keep track
- Doesn't happen much in practice
  - Requires a particular mutation pattern
  - So ok to be conservative

Variants

- Can have N generations
  - Containing progressively older and older objects
- Can migrate generations only after surviving N GCs
  - Reduces the number of short-lived objects that only survive because we happened to GC during their (short) lifetime
  - Can use some spare bits in the objects to store count (bit overloading, think mark bit)

In Practice

- The vast majority of production GCs are generational in some form
  - It’s that good
- Generational hypothesis: self-fulfilling prophecy / virtuous circle
  - Generational GC is efficient because most objects are short-lived
  - Generational GC makes short-lived objects cheap
  - Programmers use more short-lived objects because they’re cheap
  - Lather, rinse, repeat

History

- Came from the Self language (late 80s, early 90s)
  - Based on work from Lieberman and Hewitt (1983)
- Self is little-known today, but hugely influential
  - JS is basically Self
  - Implementation technology (JITs, PICs, OSR, adaptive deopts, etc.) is used all over the place

Cheney on the MTA

Charlie on the MTA

- 1940s: Massachusetts Transit Association (MTA) has both entry and exit fares on trains
- 1949: Walter A. O'Brien runs for mayor of Boston
  - His campaign: get folk singers to write songs about items on his platform
  - Then blare them from a truck going around town (got fined 10$ for that)
- One such song is about Charlie, who has enough money to get on the train, but not off
  - So *he never returned*
- 1959: The song itself becomes a hit
- 2006: MBTA (former MTA) introduces the Charlie card
- 2009: I move to Boston, and finally get the Charlie/Cheney joke

Continuation-Passing Style

- We've seen continuation-passing style
  - Wrote an interpreter that way
- In CPS, the last thing a function does is *always* call another function
  - If the function is done with its own work, it calls its continuation
  - Otherwise, passes it along to whoever it calls
  - So *our functions never return*
  - They just keep calling until the end, then everything just returns all at once
- Compilers for language with higher-order functions often convert object programs to CPS inside the compiler
  - Easier to implement control (e.g., return, exceptions, etc.)
  - Makes a lot of transformations and optimizations easier

Cheney on the MTA

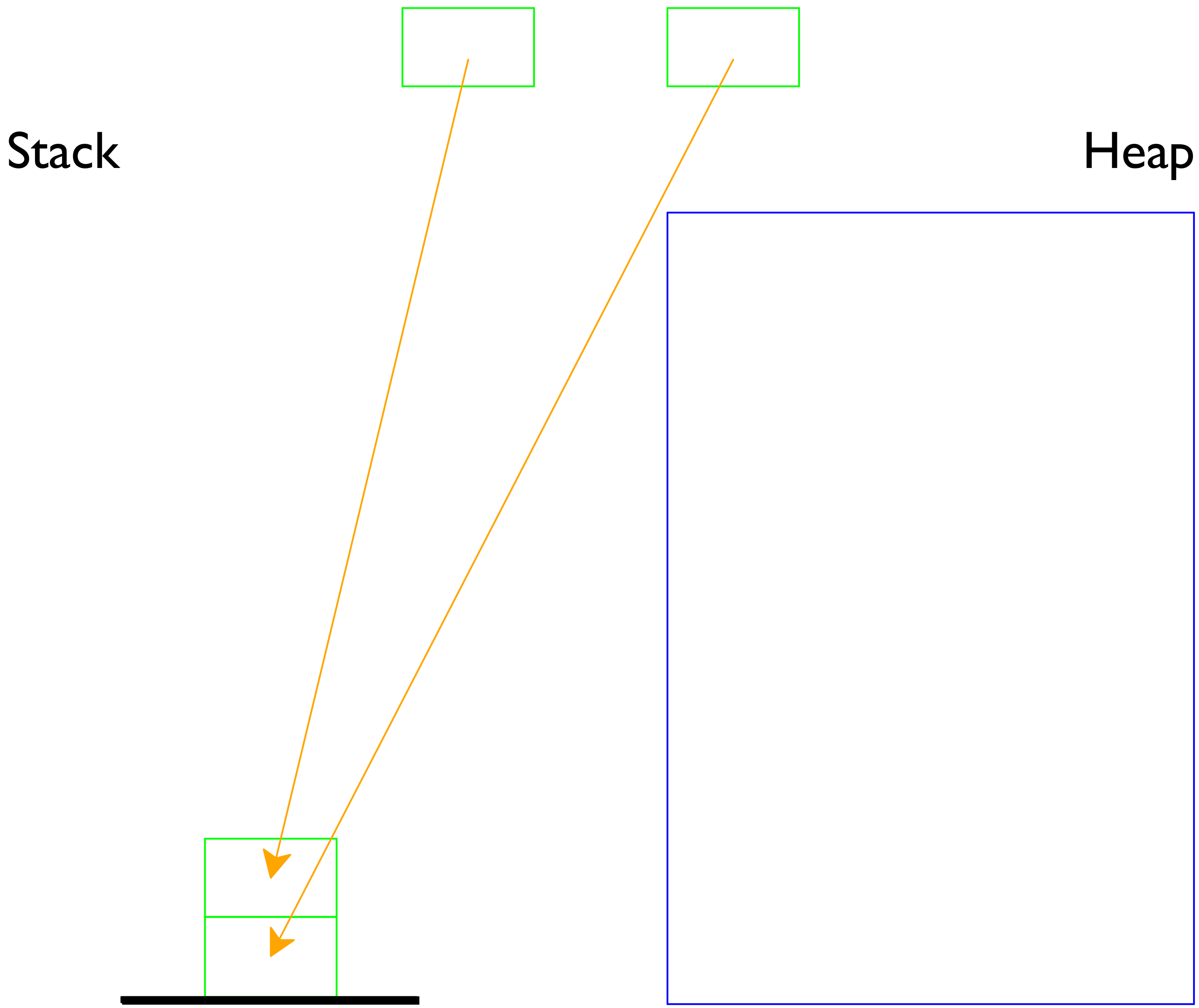
- Two-space copying collection = Cheney's algorithm
- Our object programs are in CPS
  - Their functions never return!
  - Just like Charlie!
- So their stack just keeps growing and growing
  - (In CPS, all calls are tail calls, so could reuse stack frames and solve that problem, which is how most compilers solve the problem. But not here!)

Cheney on the MTA

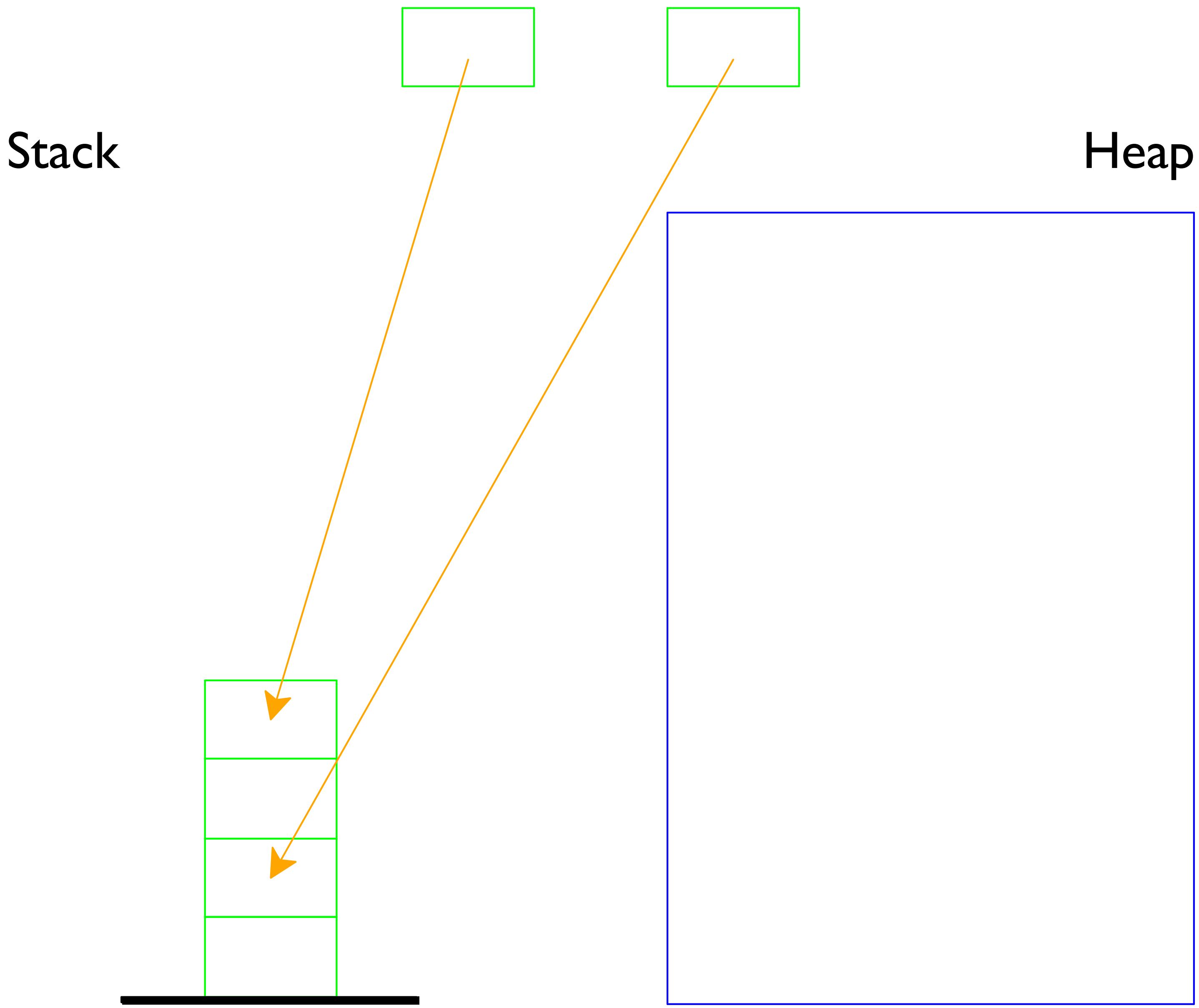
Key idea: use the stack as the nursery for a generational GC!

- Grow the stack until we run out of stack space
- Then GC, copying live objects into the heap
- Then restart the stack from 0!
  - Nothing ever returns, so we don't need return addresses!
  - And all the useful data has been copied away!
- Analogy: instead hopping on a trampoline on every function call
  - We sometimes jump over the Empire State Building
  - (Trampolines are a canonical solution for "reusing" stack frames for tail calls)

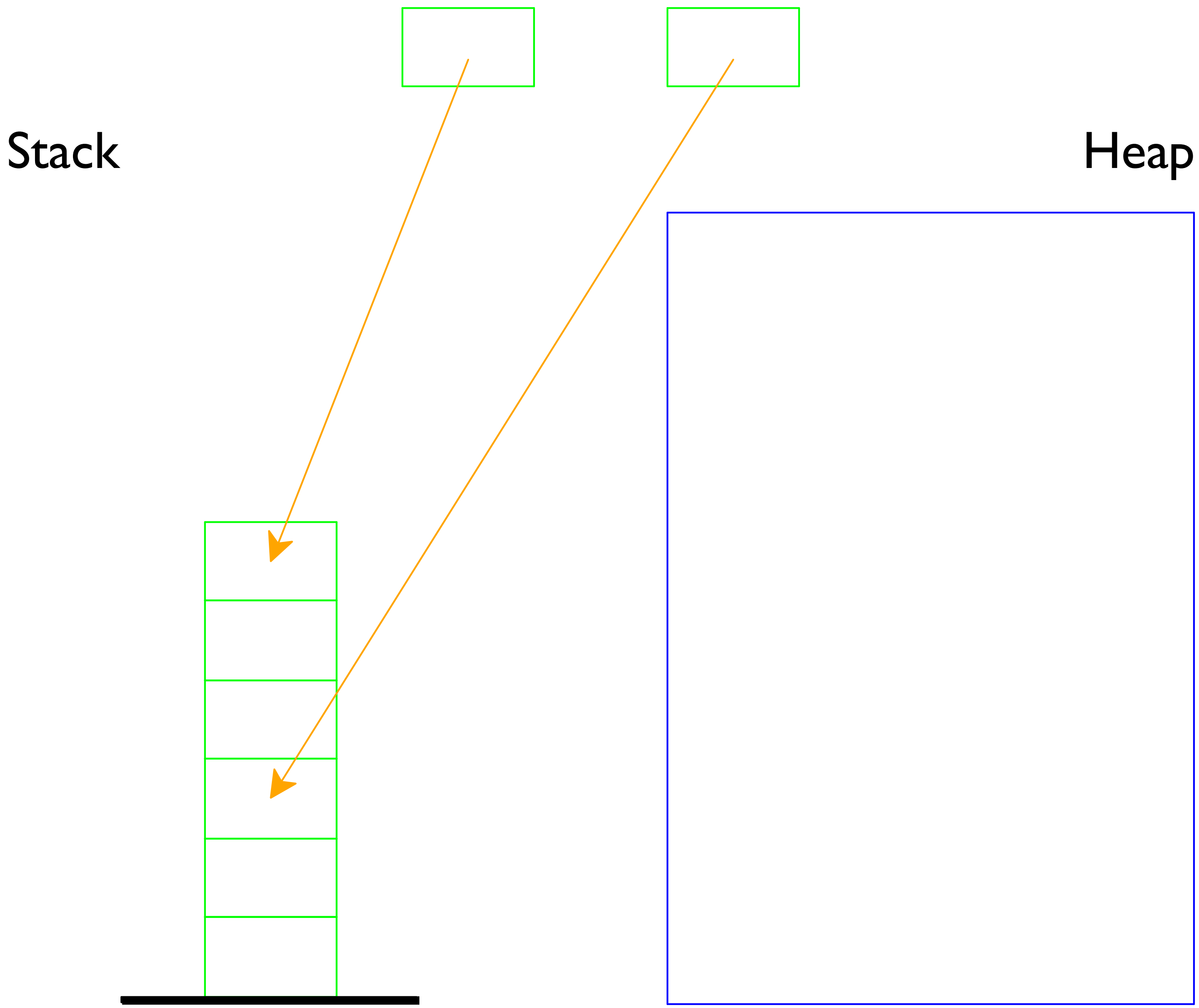
Cheney on the MTA



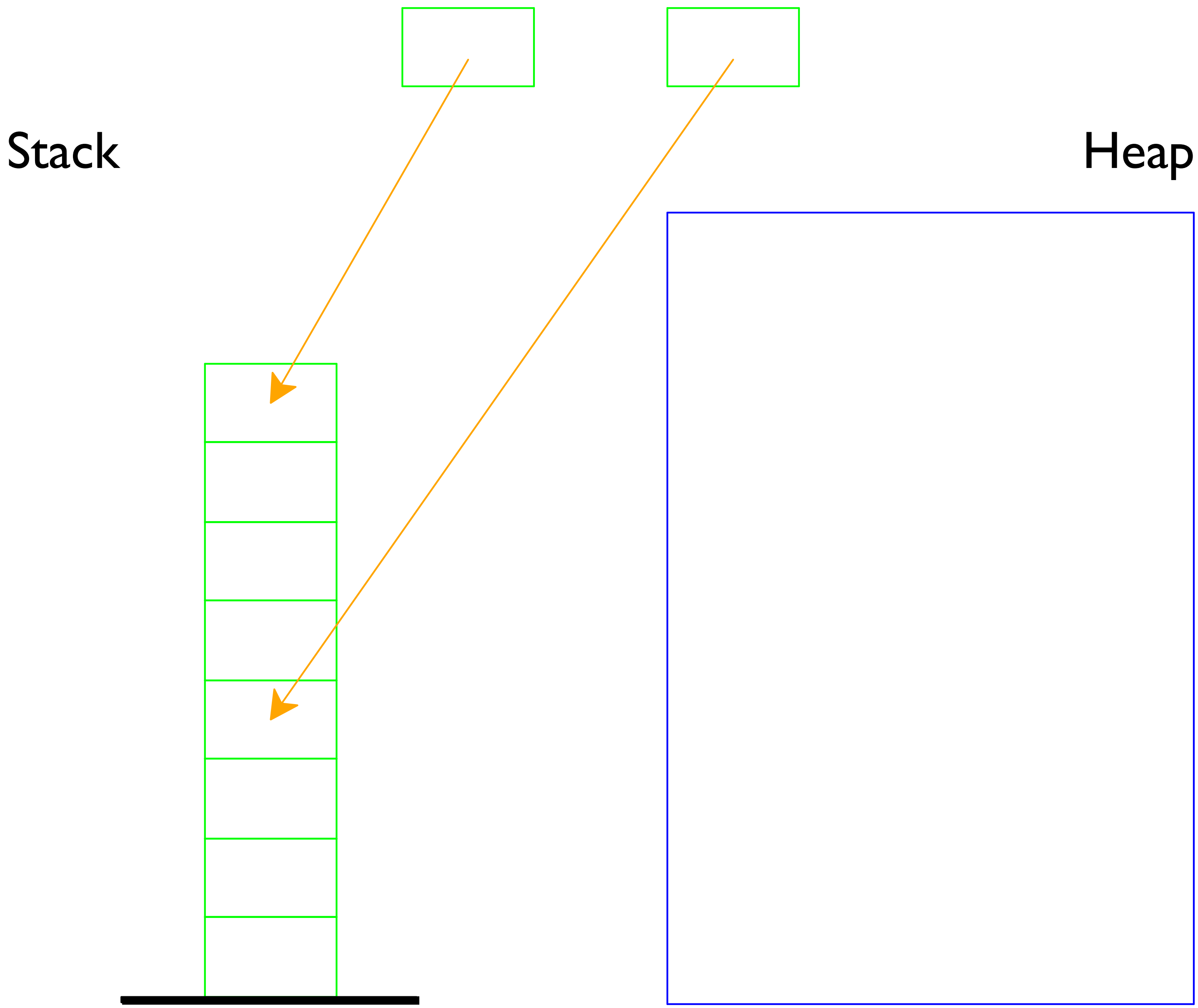
Cheney on the MTA



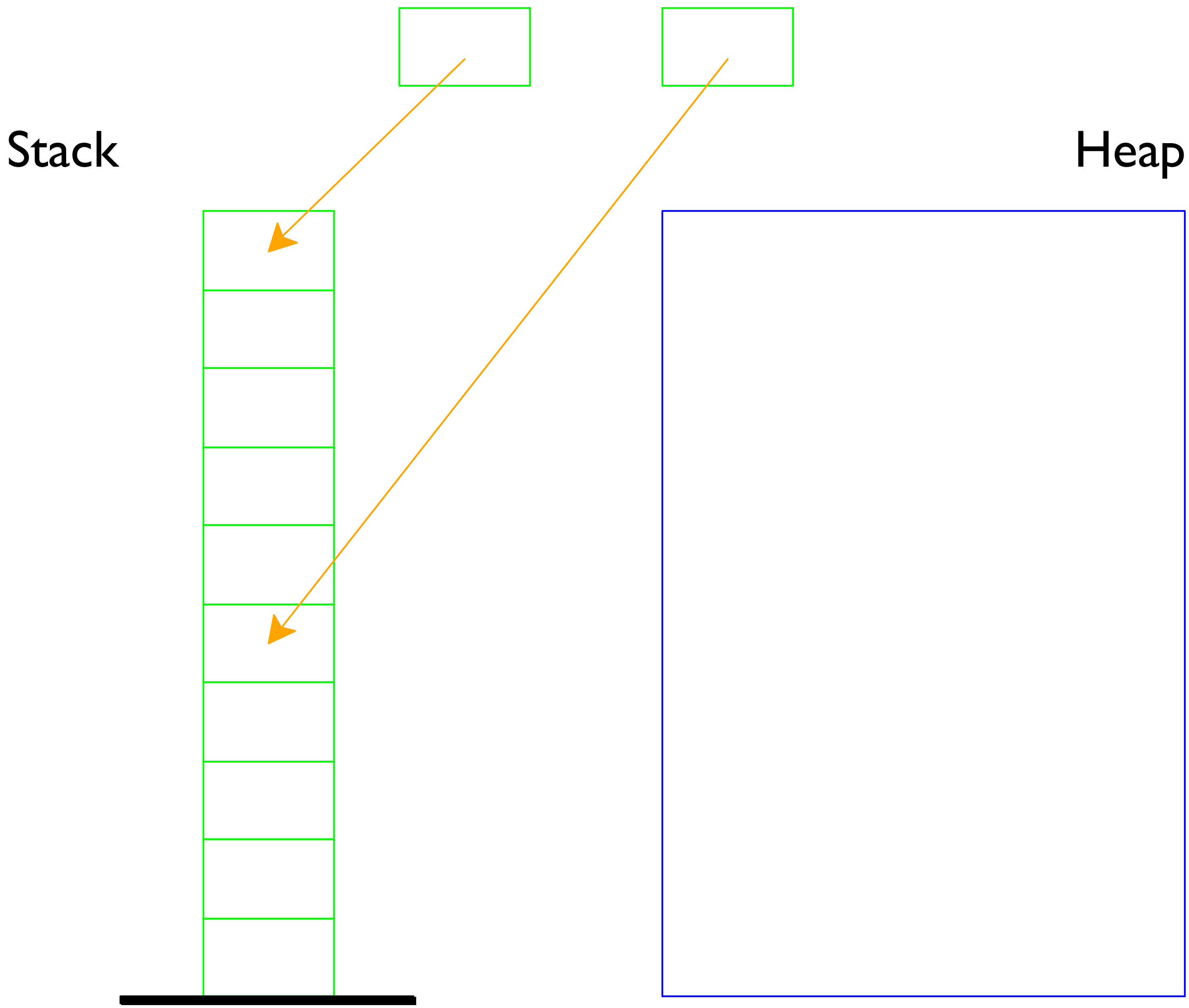
Cheney on the MTA



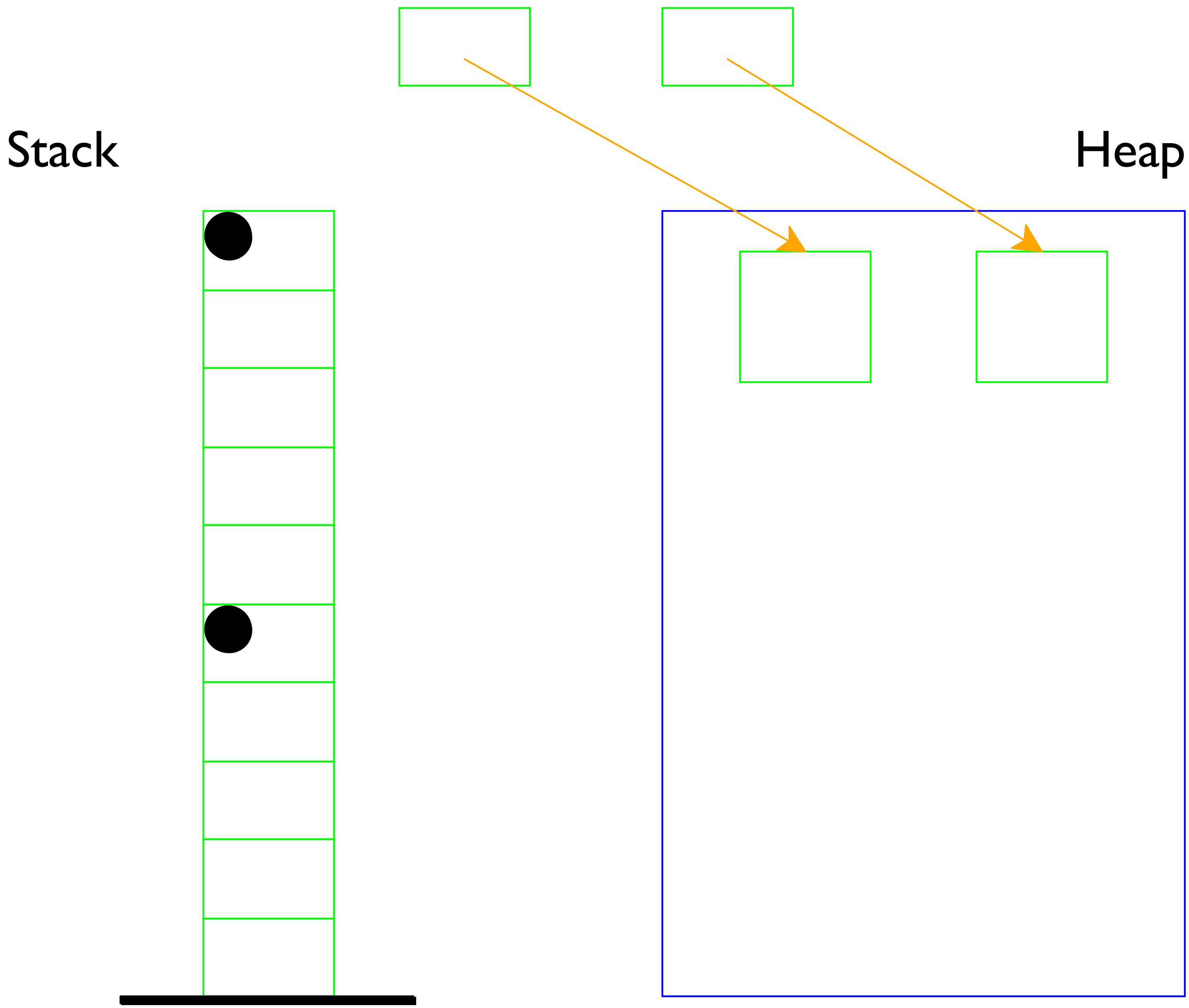
Cheney on the MTA



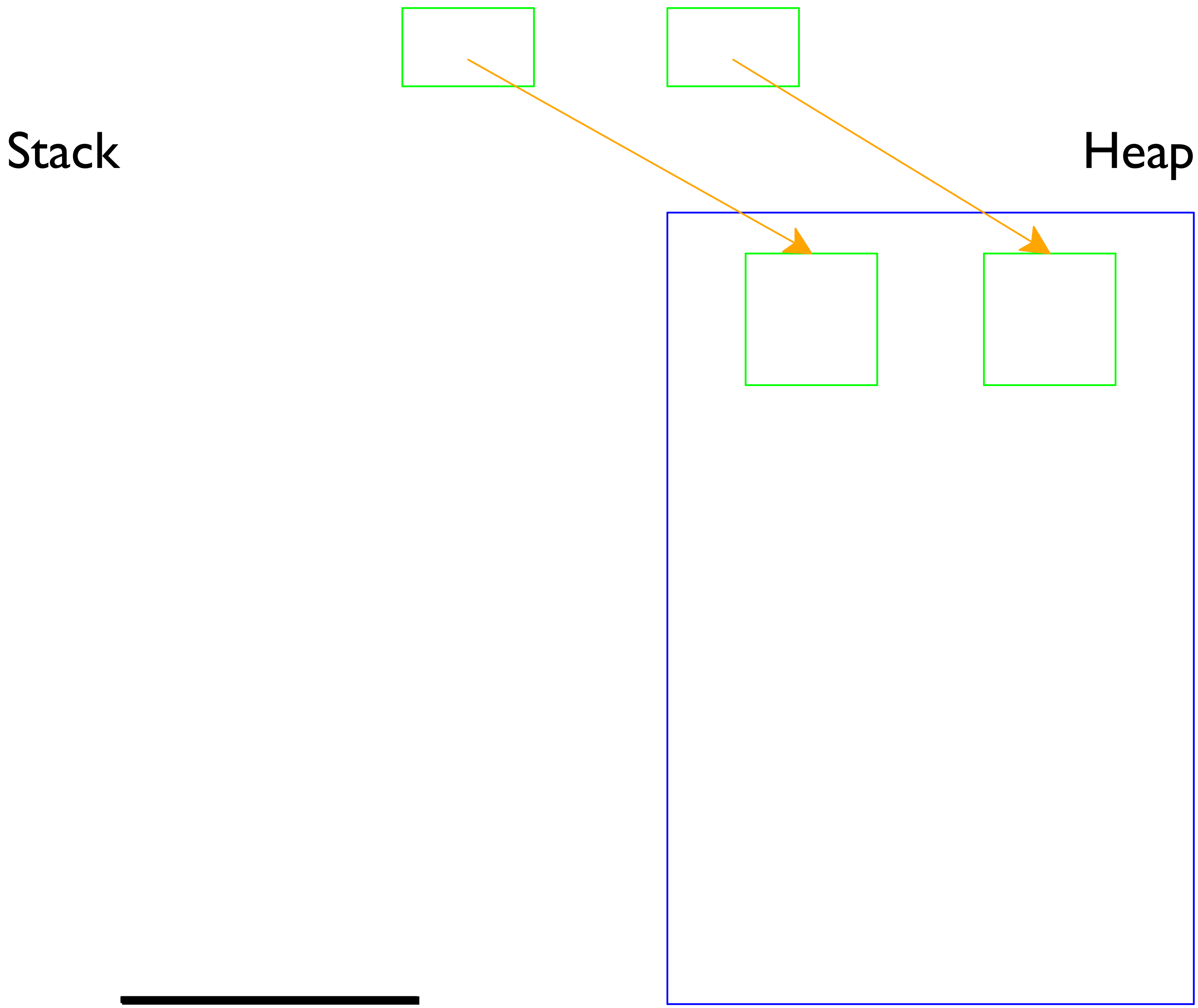
Cheney on the MTA



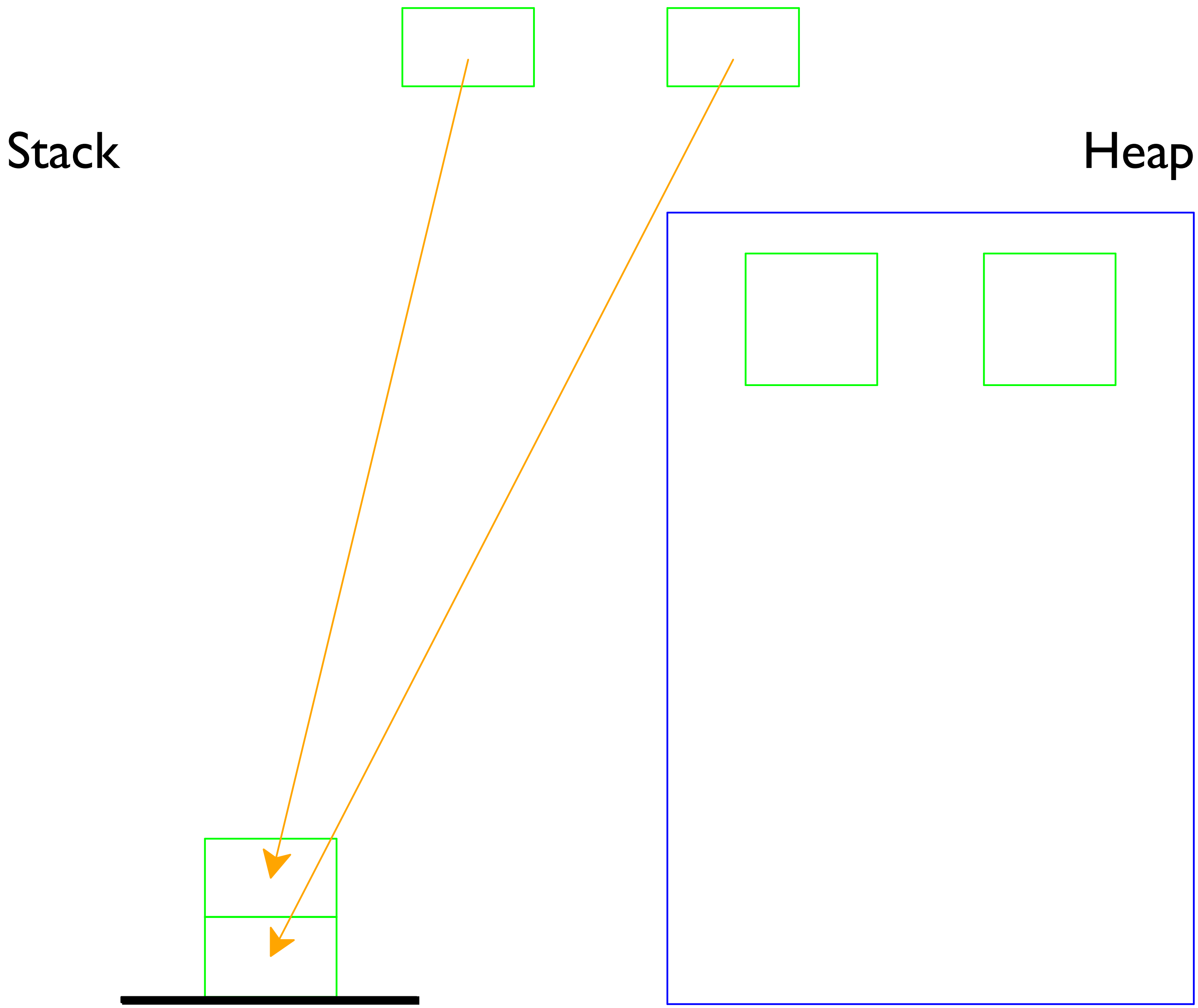
Cheney on the MTA



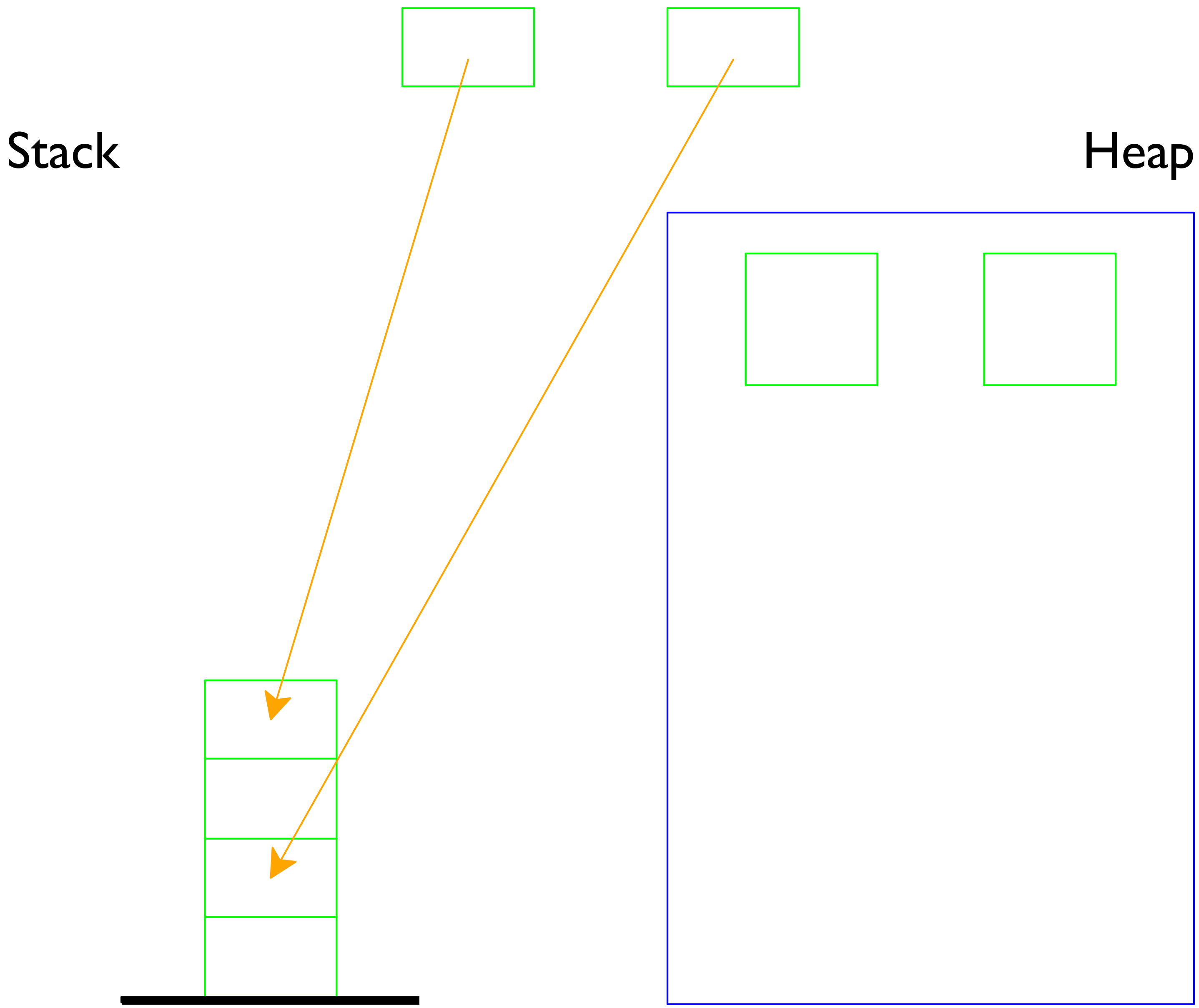
Cheney on the MTA



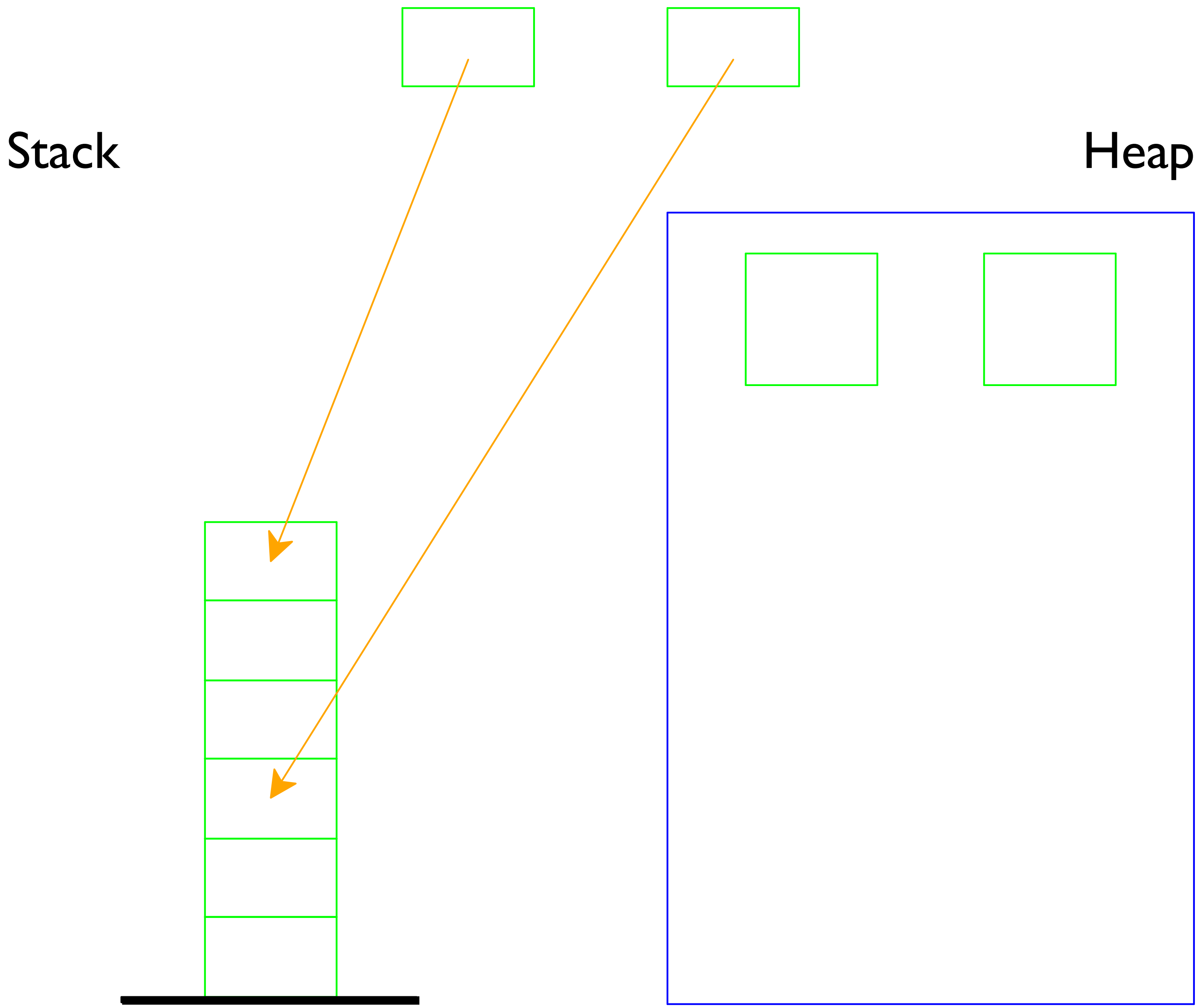
Cheney on the MTA



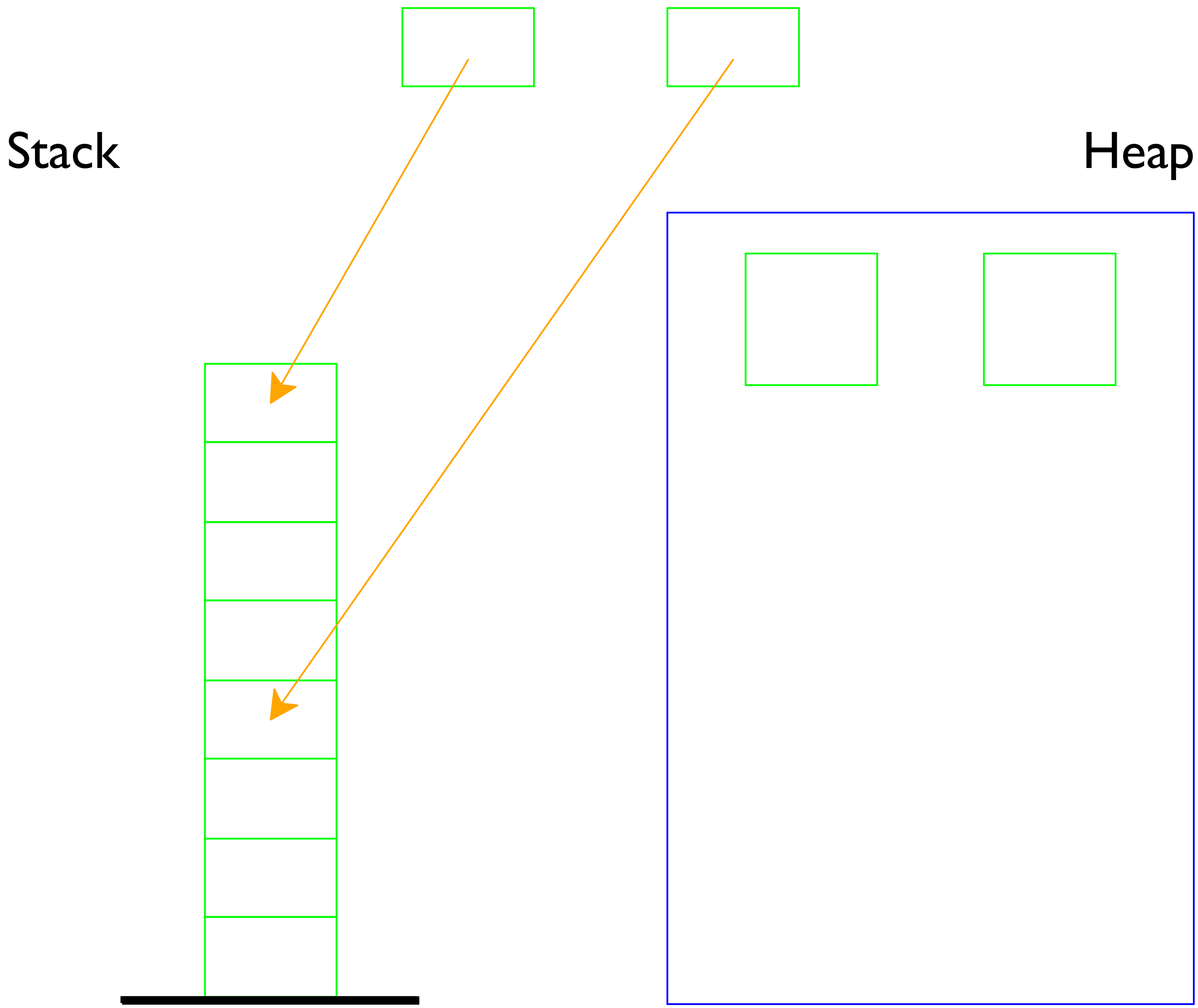
Cheney on the MTA



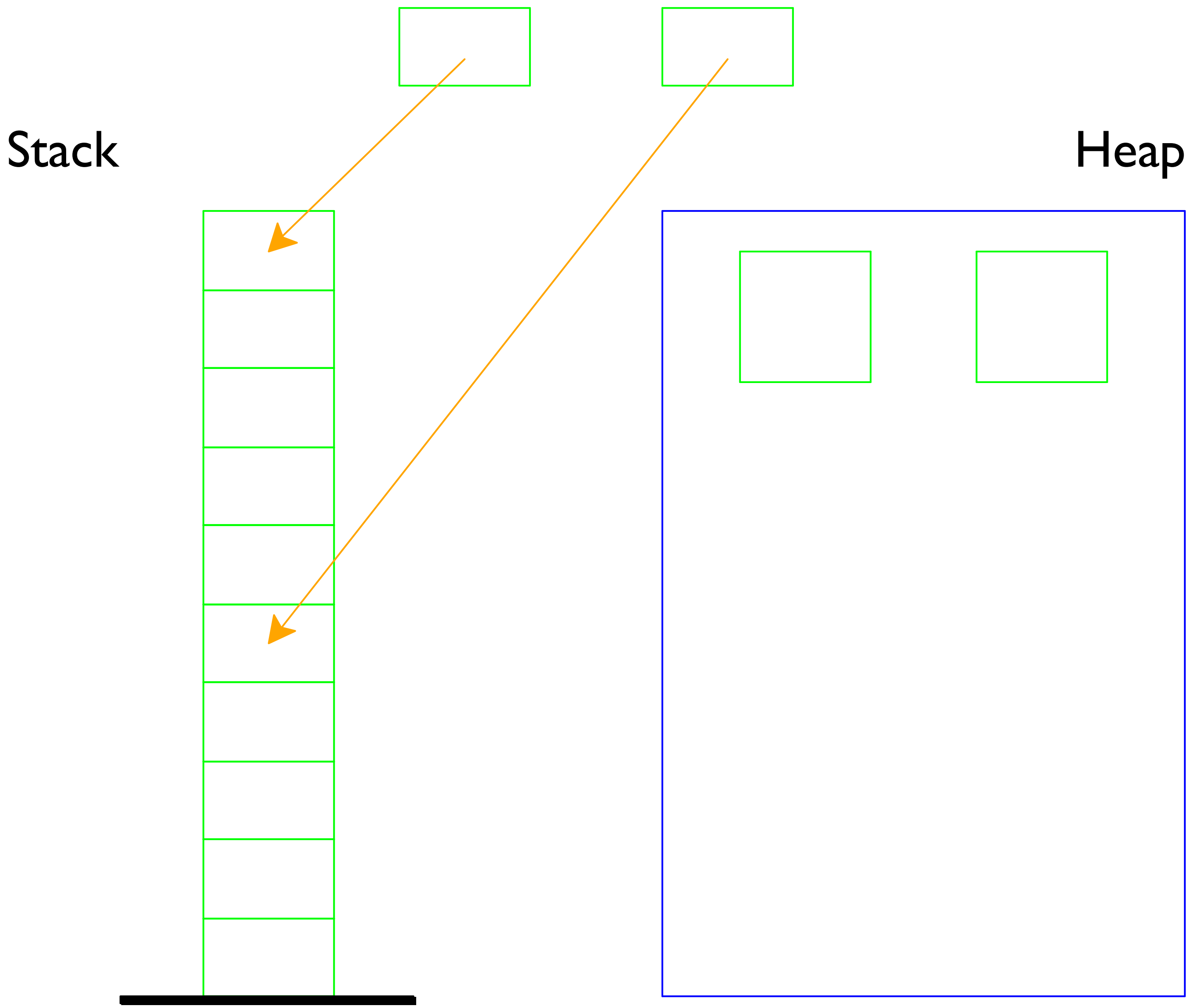
Cheney on the MTA



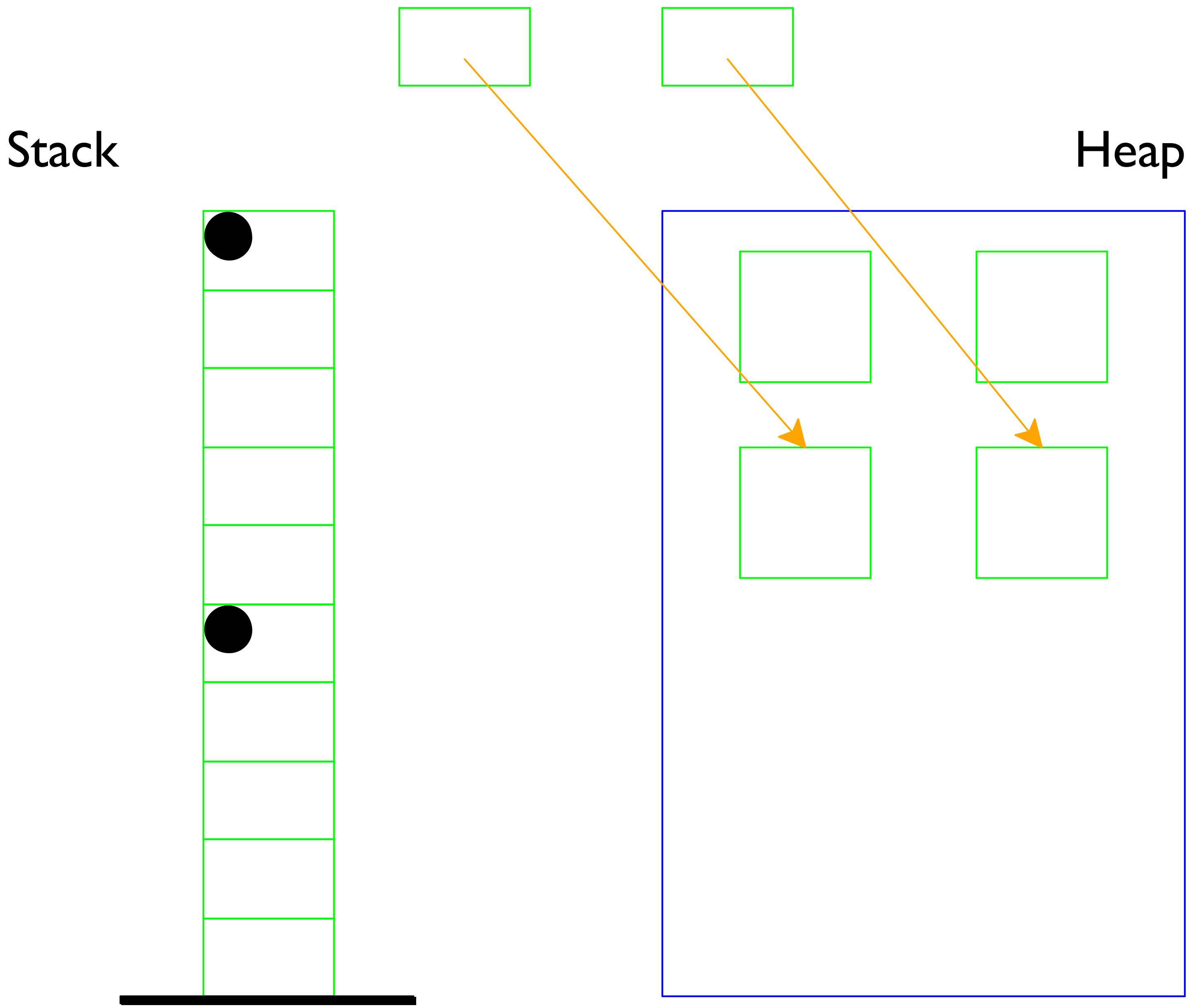
Cheney on the MTA



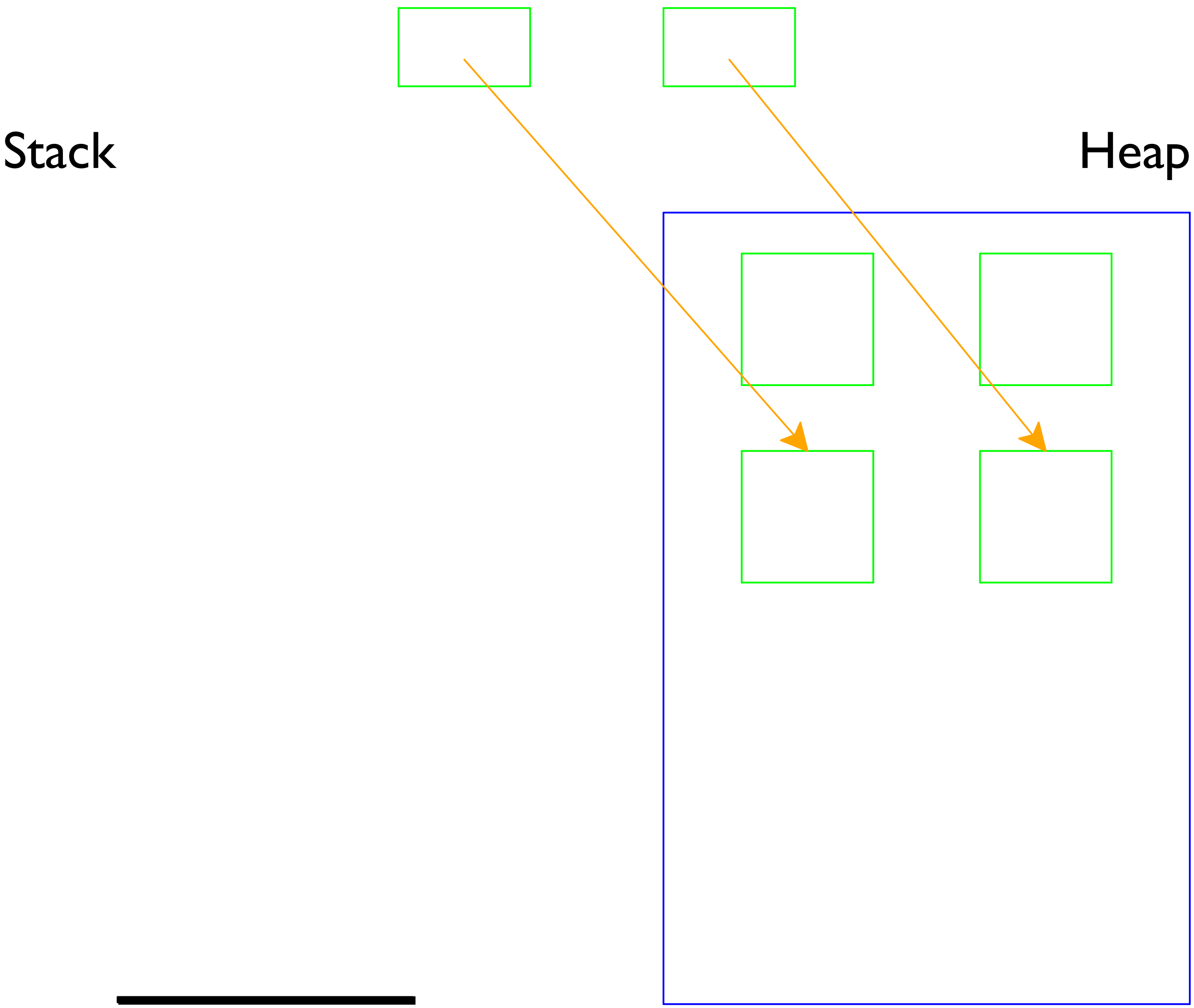
Cheney on the MTA



Cheney on the MTA



Cheney on the MTA



For More Information

- Chicken Scheme uses this implementation strategy
- CONS Should Not CONS Its Arguments, Part II: Cheney on the M.T.A., Henry Baker, 1994
  - `http://home.pipeline.com/~hbaker1/CheneyMTA.html`