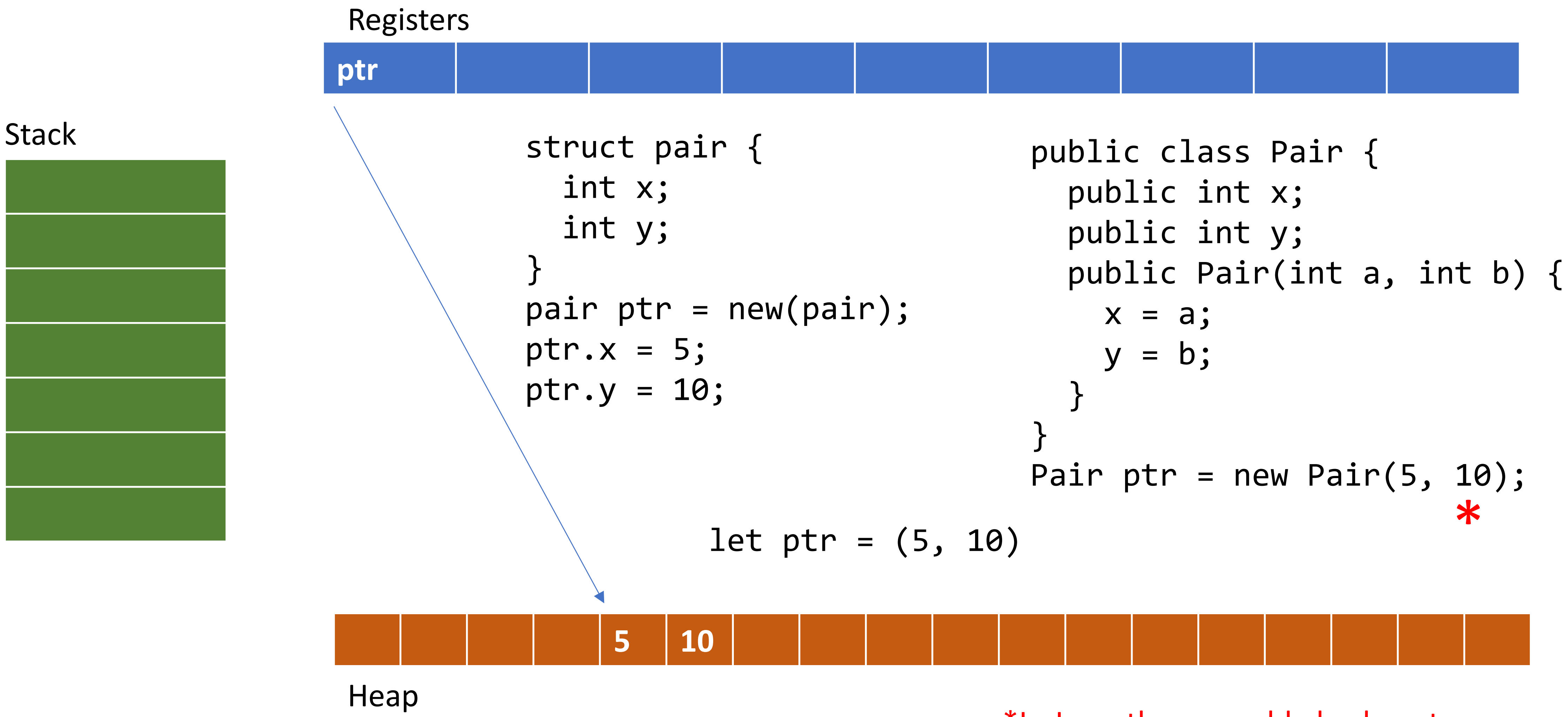
CS443: Compiler Construction

Lecture 24: Memory Management & Garbage Collection

Stefan Muller

Memory layout



```
struct pair {
  int x;
  int y;
}
pair ptr = new(pair);
ptr.x = 5;
ptr.y = 10;
```

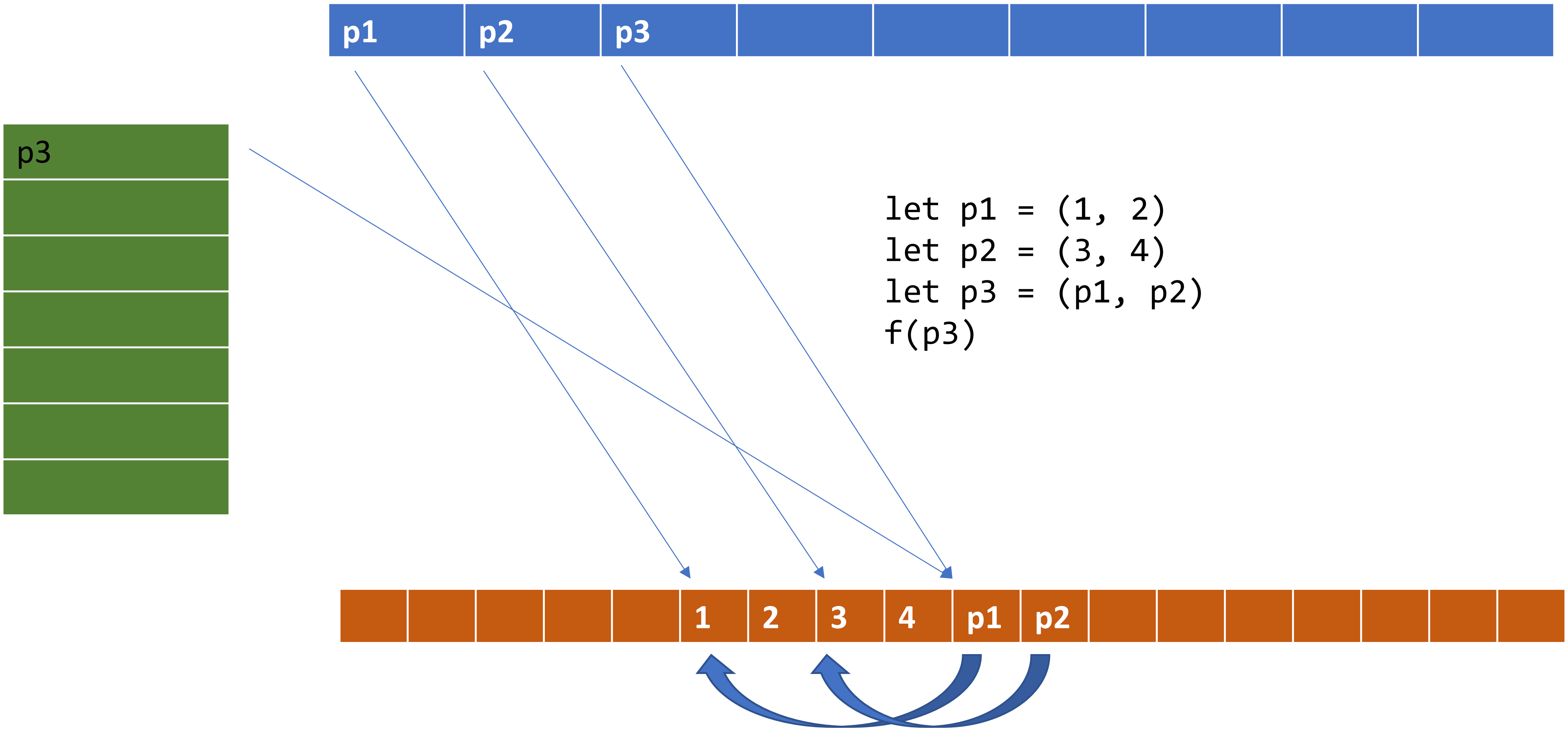
```
public class Pair {
  public int x;
  public int y;
  public Pair(int a, int b) {
    x = a;
    y = b;
  }
}
Pair ptr = new Pair(5, 10);
```

*

```
let ptr = (5, 10)
```

*In Java, there would also be a tag

Objects can be nested



```
let p1 = (1, 2)
let p2 = (3, 4)
let p3 = (p1, p2)
f(p3)
```

Memory management answers two questions

- How do we allocate memory?
- What do we do with it when we’re done?

MM breaks down into two basic strategies

- Manual – programmer says when to allocate (`malloc/new`) and free (`free/drop`)
  - Good control
  - Might forget to free/free twice/use after free

- Automatic – free memory automatically when no longer needed
  - (“Garbage collection”)
  - Some runtime overhead

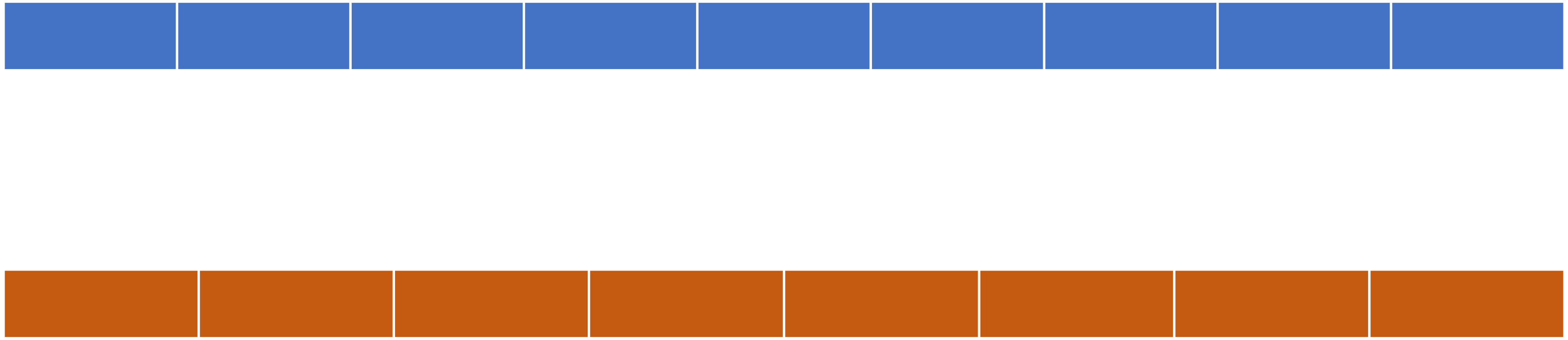
What about Rust?

- Still manual, the compiler just inserts calls to `drop` when variables go out of scope (definitely can't be used any more)
- Overly conservative, but prevents errors with free.
- Manual doesn't have to mean awful!

Manual Memory Management

```
Let
pair a = pair(x, y);
mean
pair a =
malloc(sizeof(pair));
a.fst = x;
a.snd = y;
```

```
pair a = pair(1, 2);
pair b = pair(3, 4);
pair c = pair(5, 6);
pair d = pair(a, b);
d.snd = c;
free(b);
pair e = pair(7, 8);
```



Manual Memory Management

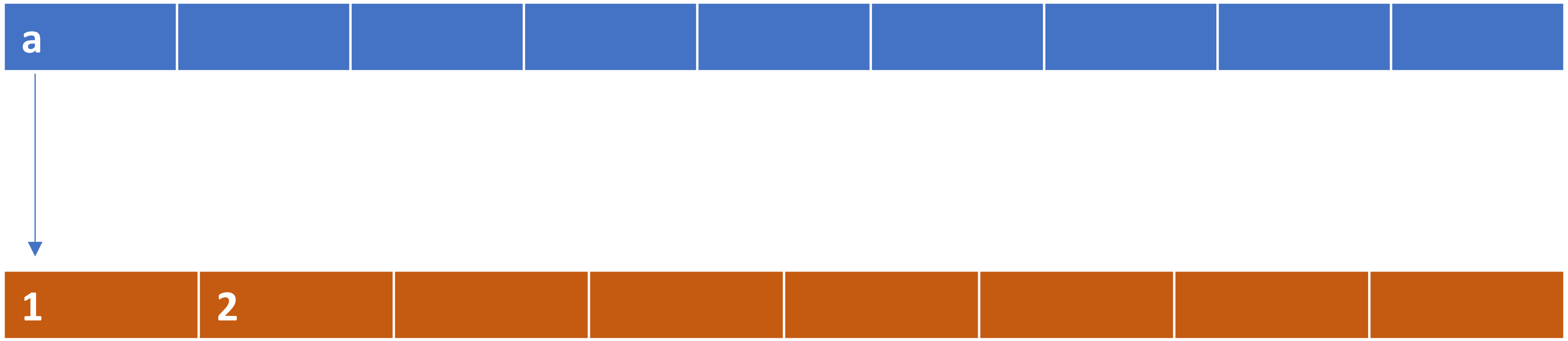
Let

```
pair a = pair(x, y);
```

mean

```
pair a =
malloc(sizeof(pair));
a.fst = x;
a.snd = y;
```

```
pair a = pair(1, 2);
pair b = pair(3, 4);
pair c = pair(5, 6);
pair d = pair(a, b);
d.snd = c;
free(b);
pair e = pair(7, 8);
```



Manual Memory Management

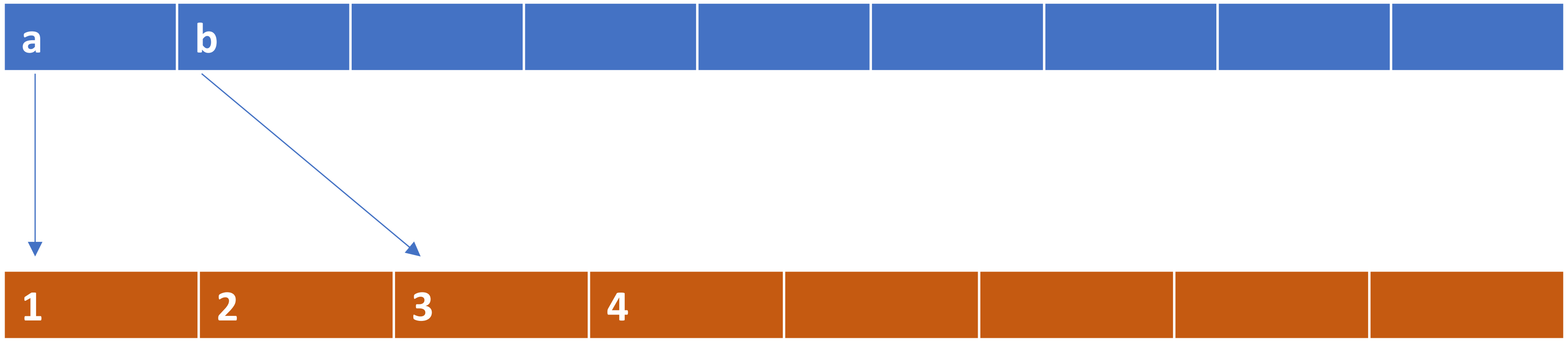
Let

```
pair a = pair(x, y);
```

mean

```
pair a =
malloc(sizeof(pair));
a.fst = x;
a.snd = y;
```

```
pair a = pair(1, 2);
pair b = pair(3, 4);
pair c = pair(5, 6);
pair d = pair(a, b);
d.snd = c;
free(b);
pair e = pair(7, 8);
```



Manual Memory Management

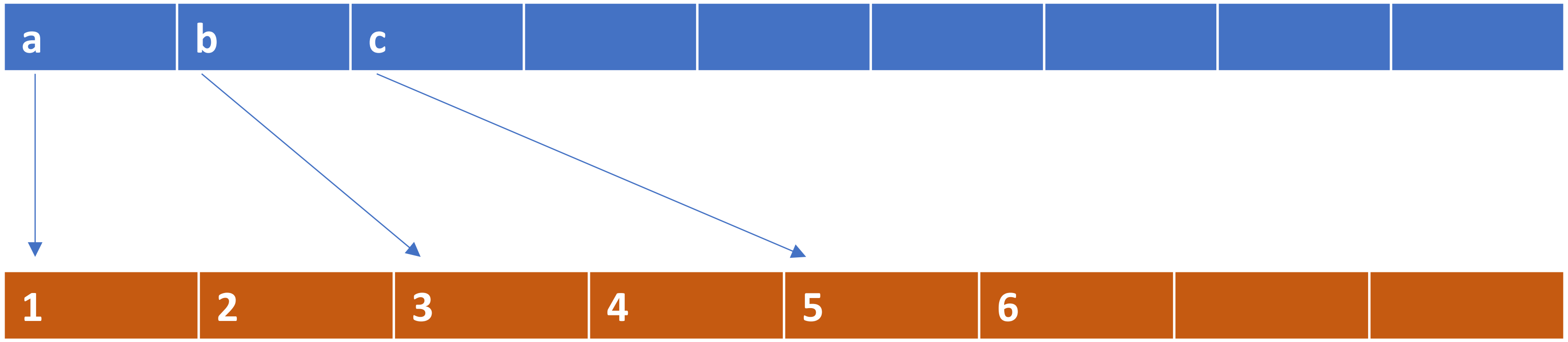
Let

```
pair a = pair(x, y);
```

mean

```
pair a =
malloc(sizeof(pair));
a.fst = x;
a.snd = y;
```

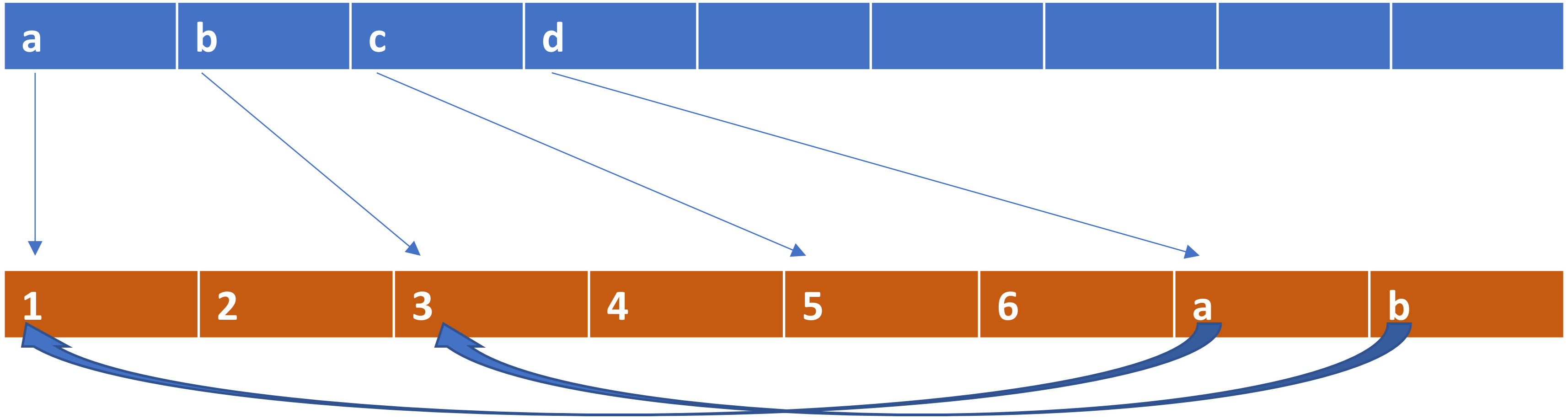
```
pair a = pair(1, 2);
pair b = pair(3, 4);
pair c = pair(5, 6);
pair d = pair(a, b);
d.snd = c;
free(b);
pair e = pair(7, 8);
```



Manual Memory Management

```
Let
pair a = pair(x, y);
mean
pair a =
malloc(sizeof(pair));
a.fst = x;
a.snd = y;
```

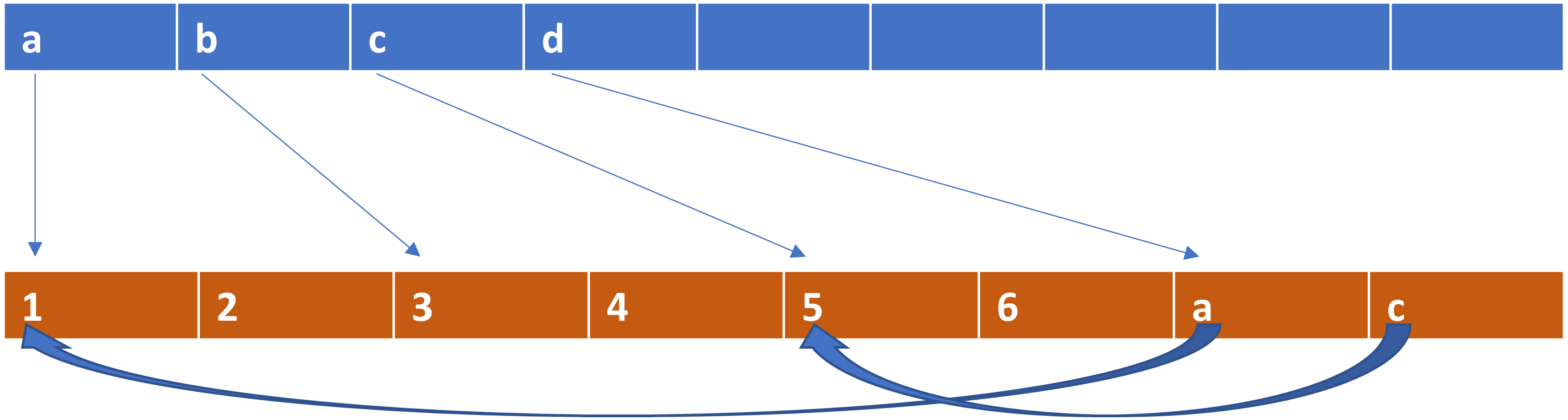
```
pair a = pair(1, 2);
pair b = pair(3, 4);
pair c = pair(5, 6);
pair d = pair(a, b);
d.snd = c;
free(b);
pair e = pair(7, 8);
```



Manual Memory Management

```
Let
pair a = pair(x, y);
mean
pair a =
malloc(sizeof(pair));
a.fst = x;
a.snd = y;
```

```
pair a = pair(1, 2);
pair b = pair(3, 4);
pair c = pair(5, 6);
pair d = pair(a, b);
d.snd = c;
free(b);
pair e = pair(7, 8);
```



Manual Memory Management

Let

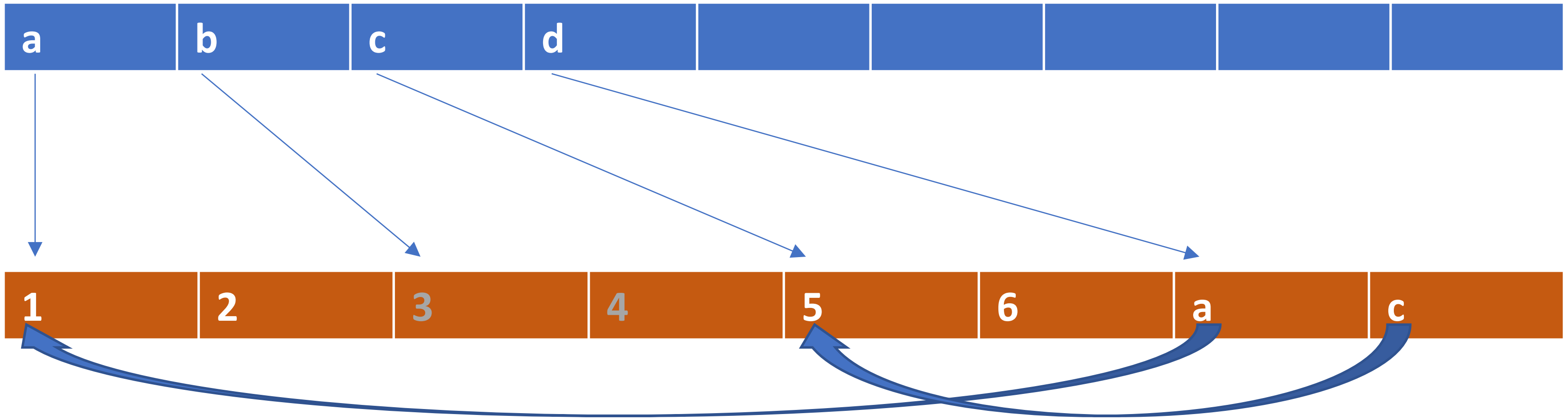
```
pair a = pair(x, y);
```

mean

```
pair a =
malloc(sizeof(pair));
a.fst = x;
a.snd = y;
```

```
pair a = pair(1, 2);
pair b = pair(3, 4);
pair c = pair(5, 6);
pair d = pair(a, b);
d.snd = c;
free(b);
pair e = pair(7, 8);
```

b still valid, still points to same loc, but can reuse memory



Manual Memory Management

Let

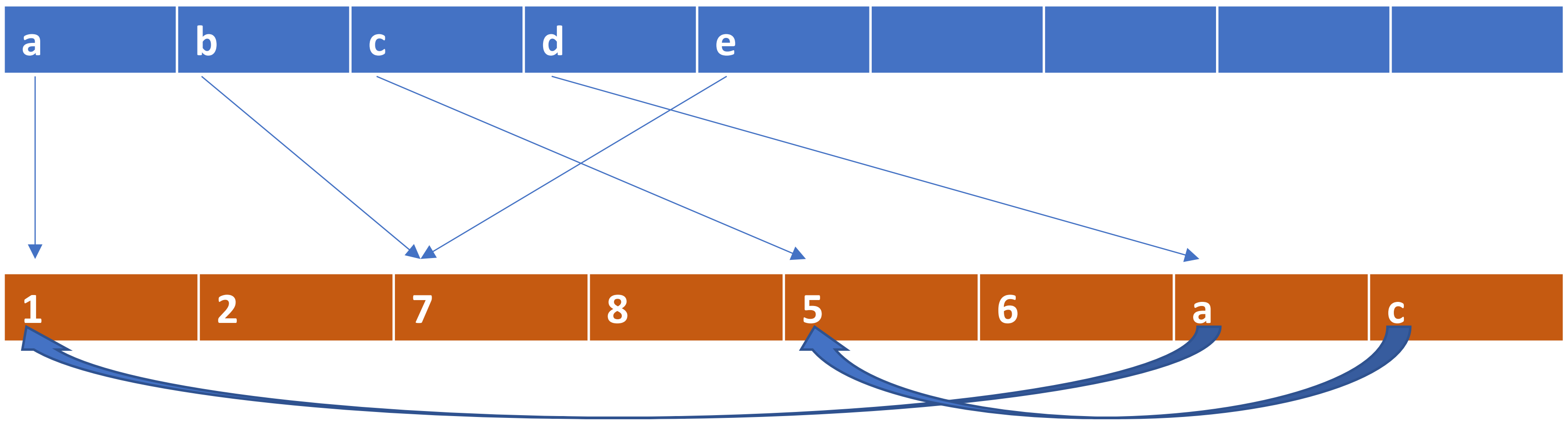
```
pair a = pair(x, y);
```

mean

```
pair a =
malloc(sizeof(pair));
a.fst = x;
a.snd = y;
```

Need to reuse that space—*fragmentation*

```
pair a = pair(1, 2);
pair b = pair(3, 4);
pair c = pair(5, 6);
pair d = pair(a, b);
d.snd = c;
free(b);
pair e = pair(7, 8);
```



Manual pros and cons

- Pros
  - Space-efficient
  - `free` is cheap
  - Lots of control

- Cons
  - `malloc` is expensive (and hard to implement)!
  - Lots of control

Functional languages allocate *a lot*

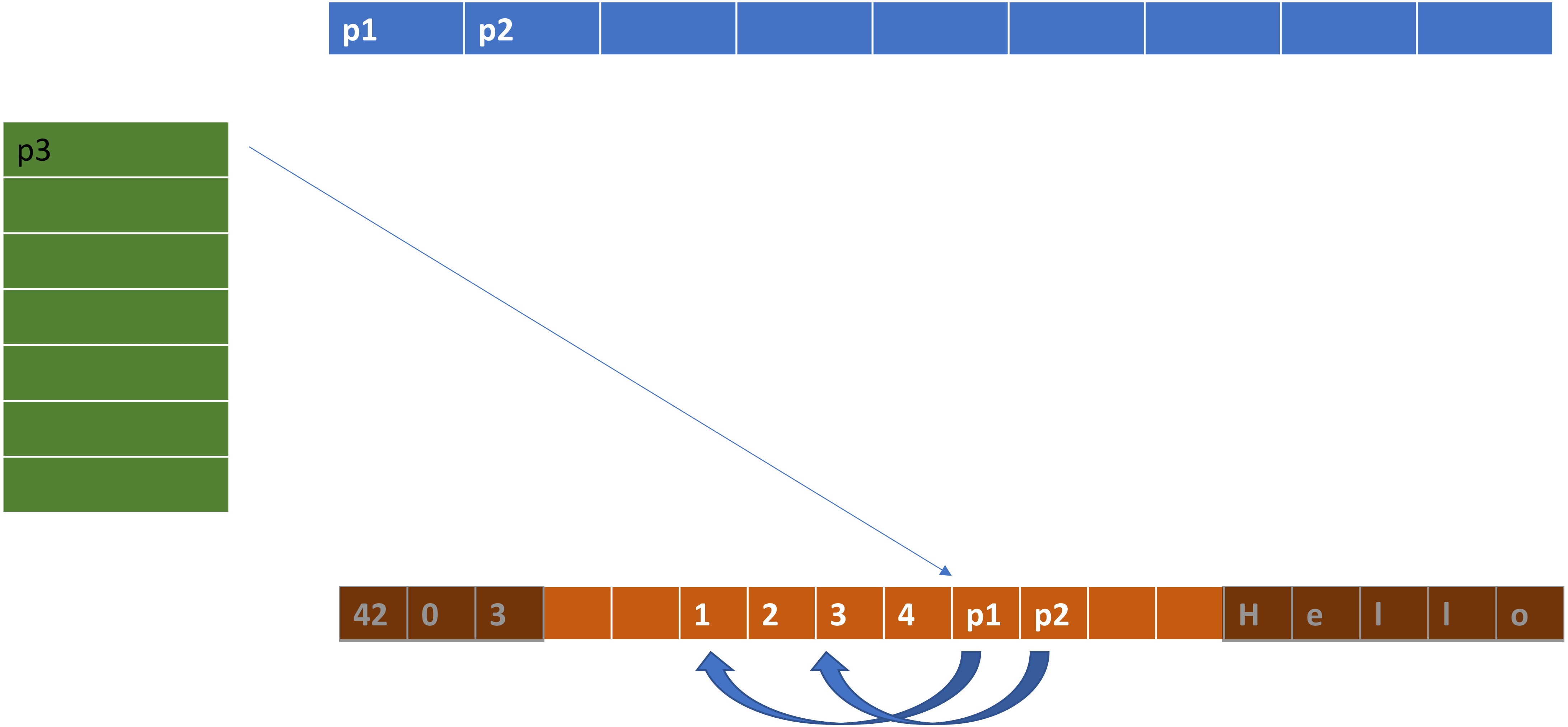
```
__list __ctemp52 = new(__list);
__ctemp52.list_tl = __env;
__ctemp52.list_hd = ((int)(__list)__ctemp51.list_hd);
__env = __ctemp52;
__list __ctemp53 = new(__list);
__ctemp53.list_tl = __env;
__ctemp53.list_hd = ((int)__ctemp51.list_tl);
__env = __ctemp53;
__clos __ctemp54 = ((__clos)__lookup(4, __env));
__clos __ctemp55 =
  ((__clos(*)(int, __list))__ctemp54.clos_fun)(((int)__lookup(1, __env)),
  __ctemp54.clos_env);
__clos __ctemp56 = __ctemp55;
__pair __ctemp57 =
  ((__pair(*)(__list, __list))__ctemp56.clos_fun)(((__list)__lookup(0,
  __env)), __ctemp56.clos_env);
__pair __ctemp58 = __ctemp57;
__list __ctemp59 = new(__list);
__ctemp59.list_tl = __env;
__ctemp59.list_hd = ((int)__ctemp58.pair_fst);
__env = __ctemp59;
__list __ctemp60 = new(__list);
__ctemp60.list_tl = __env;
__ctemp60.list_hd = ((int)__ctemp58.pair_snd);
```

And also can you imagine having to free everything manually?

Reachability and garbage

- Root set: Anything immediately reachable (registers, stack)
  - e.g., local variables, arguments
- Reachable (“live”): any objects (transitively) pointed to by root set
- Garbage (“dead”): any allocated objects not reachable

Objects not reachable from roots are dead/garbage



Knowing what points to what isn’t as easy as it sounds

- In C:

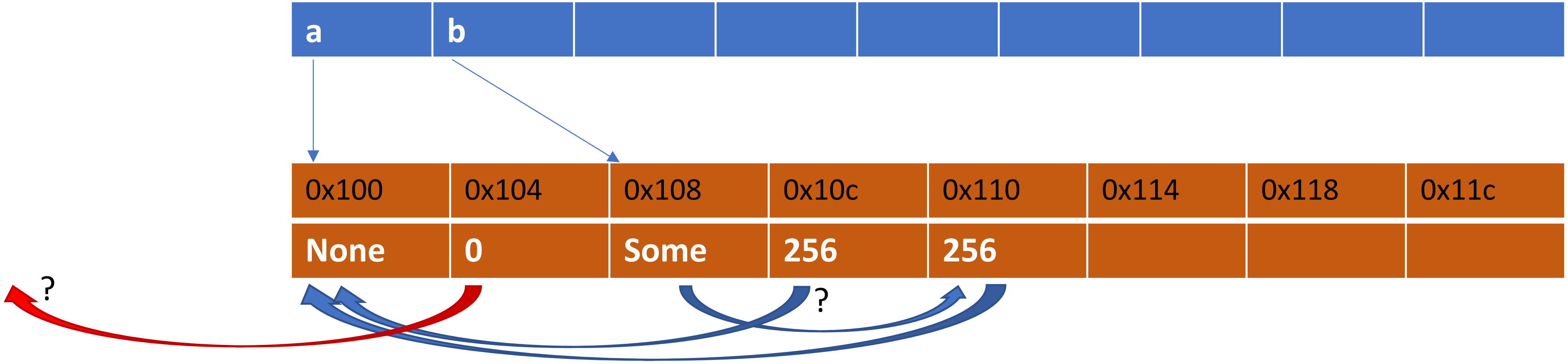
```
int *p = (int *)0xdeadbeef;
*p = 5;
```

Garbage collection won’t work well in C

Knowing what points to what isn’t as easy as it sounds

In ML

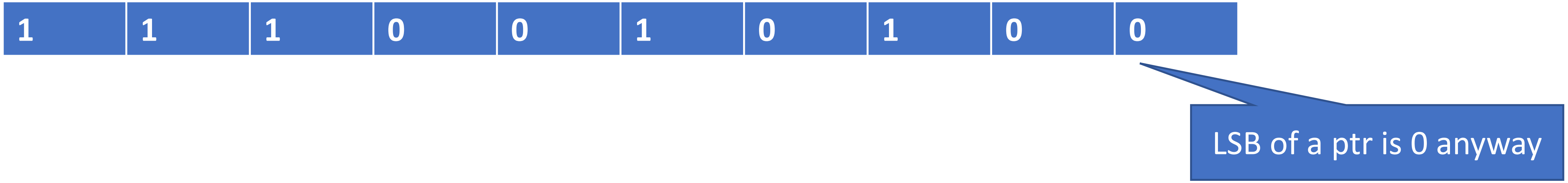
```
let a = (None, 0)
let b = (Some a, 256)
```



a	b							

0x100	0x104	0x108	0x10c	0x110	0x114	0x118	0x11c
None	0	Some	256	256			

OCaml’s clever hack: use the LSB to indicate integer or pointer

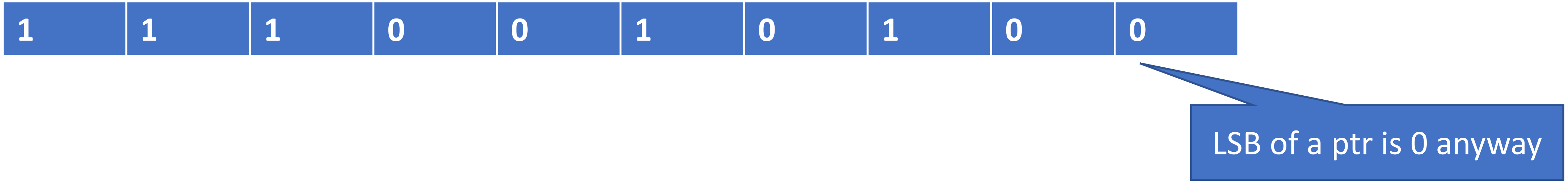


1	1	1	0	0	1	0	1	0	0

LSB of a ptr is 0 anyway

```
(1, 2, x + 1)
```

OCaml’s clever hack: use the LSB to indicate integer or pointer



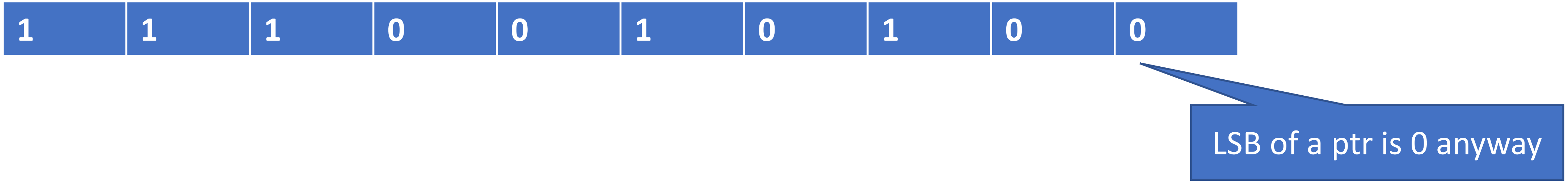
1	1	1	0	0	1	0	1	0	0

LSB of a ptr is 0 anyway

```
(1, 2, x + 1)
```

```
12d43:    48 c7 00 03 00 00 00       movq   $0x3,(%rax)
12d4a:    48 c7 40 08 05 00 00       movq   $0x5,0x8(%rax)
12d51:    00
12d52:    48 83 c3 02                add    $0x2,%rbx
12d56:    48 89 58 10                mov    %rbx,0x10(%rax)
12d5a:    48 83 c4 08                add    $0x8,%rsp
12d5e:    c3                         retq
```

OCaml’s clever hack: use the LSB to indicate integer or pointer



1	1	1	0	0	1	0	1	0	0

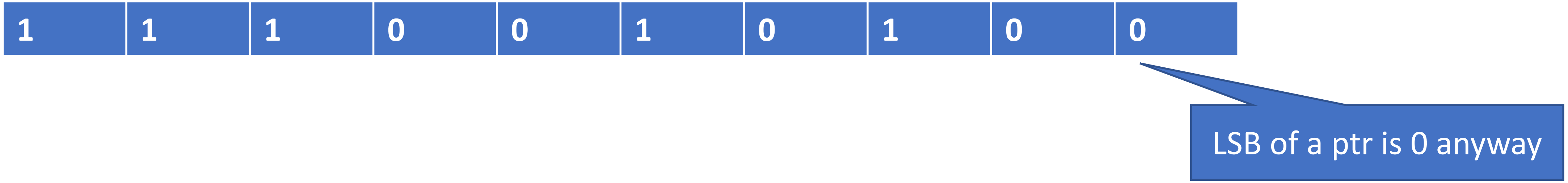
LSB of a ptr is 0 anyway

3 = 1 << 1 + 1

```
(1, 2, x + 1)
```

```
12d43:    48 c7 00 03 00 00 00       movq   $0x3,(%rax)
12d4a:    48 c7 40 08 05 00 00       movq   $0x5,0x8(%rax)
12d51:    00
12d52:    48 83 c3 02                add    $0x2,%rbx
12d56:    48 89 58 10                mov    %rbx,0x10(%rax)
12d5a:    48 83 c4 08                add    $0x8,%rsp
12d5e:    c3                         retq
```

OCaml's clever hack: use the LSB to indicate integer or pointer



1	1	1	0	0	1	0	1	0	0

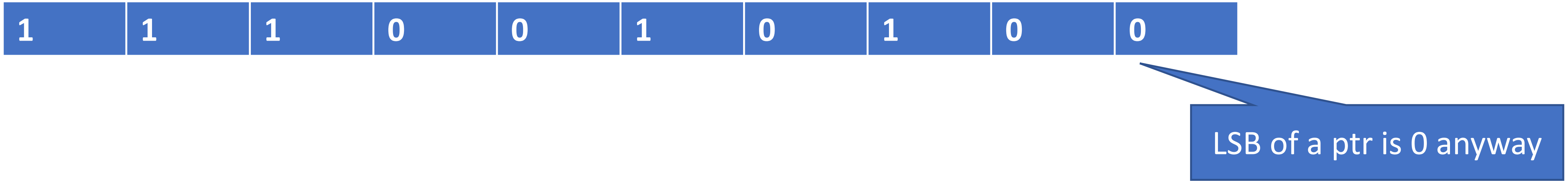
LSB of a ptr is 0 anyway

5 = 2 << 1 + 1

```
(1, 2, x + 1)
```

```
12d43:    48 c7 00 03 00 00 00      movq   $0x3,(%rax)
12d4a:    48 c7 40 08 05 00 00      movq   $0x5,0x8(%rax)
12d51:    00
12d52:    48 83 c3 02               add    $0x2,%rbx
12d56:    48 89 58 10               mov    %rbx,0x10(%rax)
12d5a:    48 83 c4 08               add    $0x8,%rsp
12d5e:    c3                        retq
```

OCaml’s clever hack: use the LSB to indicate integer or pointer



1	1	1	0	0	1	0	1	0	0

LSB of a ptr is 0 anyway

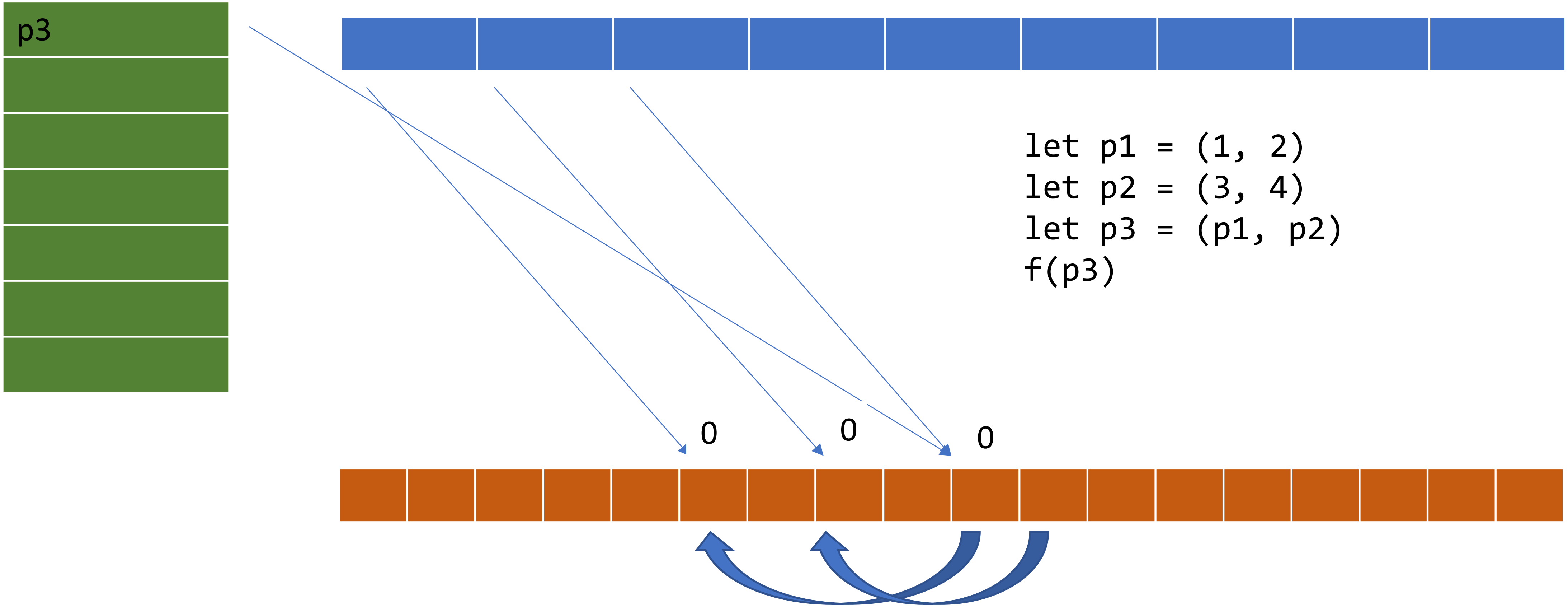
(x << 1 + 1) + (1 << 1) = (x + 1) << 1 + 1

```
(1, 2, x + 1)
```

```
12d43:    48 c7 00 03 00 00 00     movq   $0x3,(%rax)
12d4a:    48 c7 40 08 05 00 00     movq   $0x5,0x8(%rax)
12d51:    00
12d52:    48 83 c3 02              add    $0x2,%rbx
12d56:    48 89 58 10              mov    %rbx,0x10(%rax)
12d5a:    48 83 c4 08              add    $0x8,%rsp
12d5e:    c3                       retq
```

GC Strategy #1: Reference counting

- Idea: keep track of how many references every object has



```
let p1 = (1, 2)
let p2 = (3, 4)
let p3 = (p1, p2)
f(p3)
```

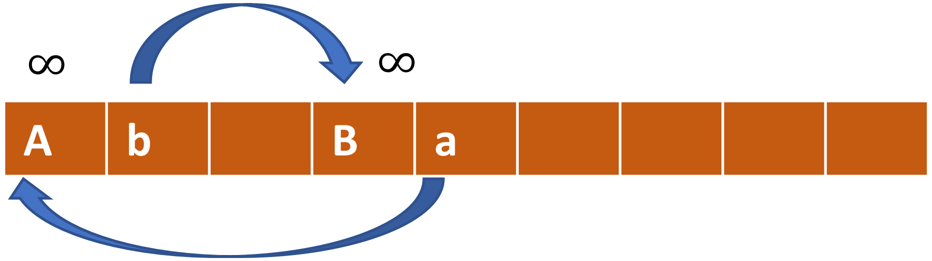
Reference counting pros

- Simple, intuitive
- Garbage collected immediately

Reference counting cons

- Cyclic data structures

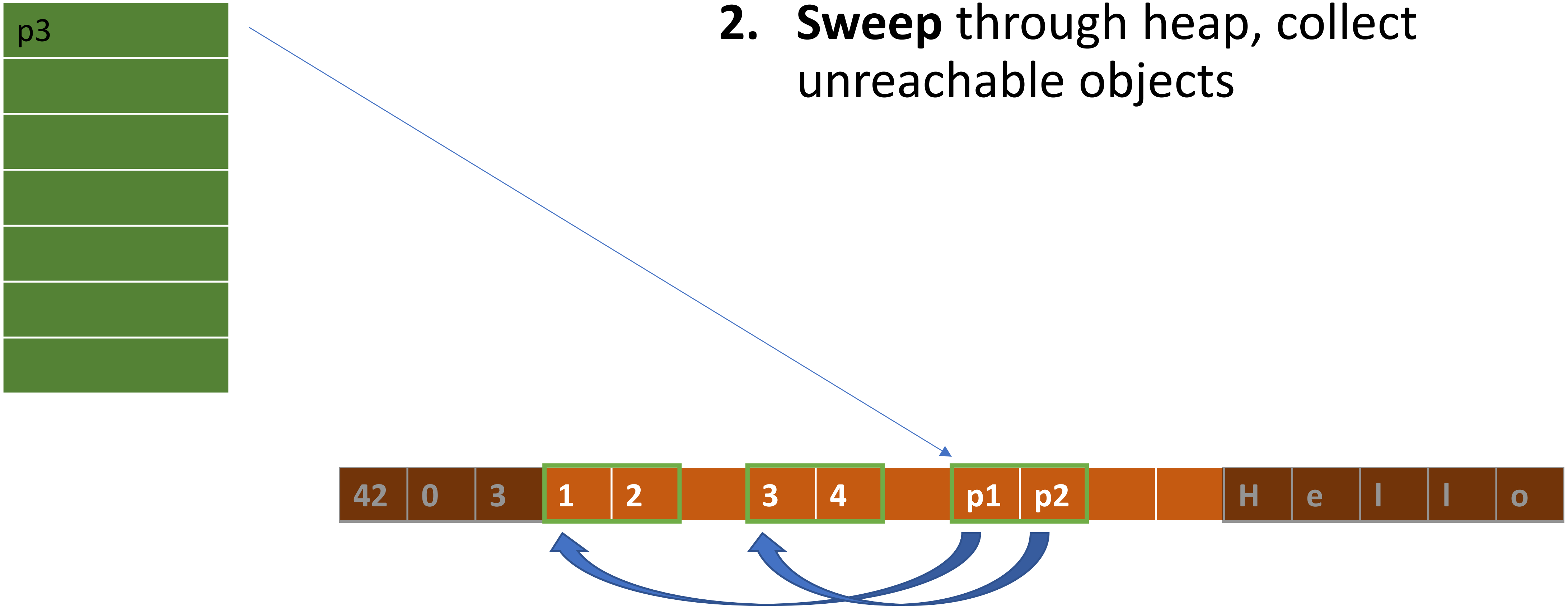
```
a = new A();
b = new B();
A.b = b;
B.a = a;
```



- Updating counts can be expensive

GC Strategy #2: Mark and sweep

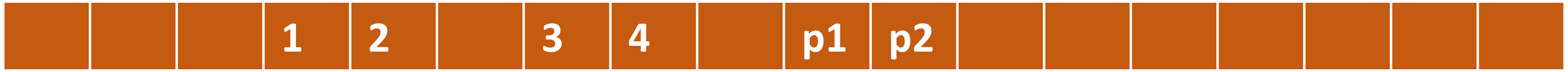
1. **Mark** reachable objects
2. **Sweep** through heap, collect unreachable objects



Mark and Sweep pros and cons

- Pros:
  - Works on cyclic references
  - Just traverse references once

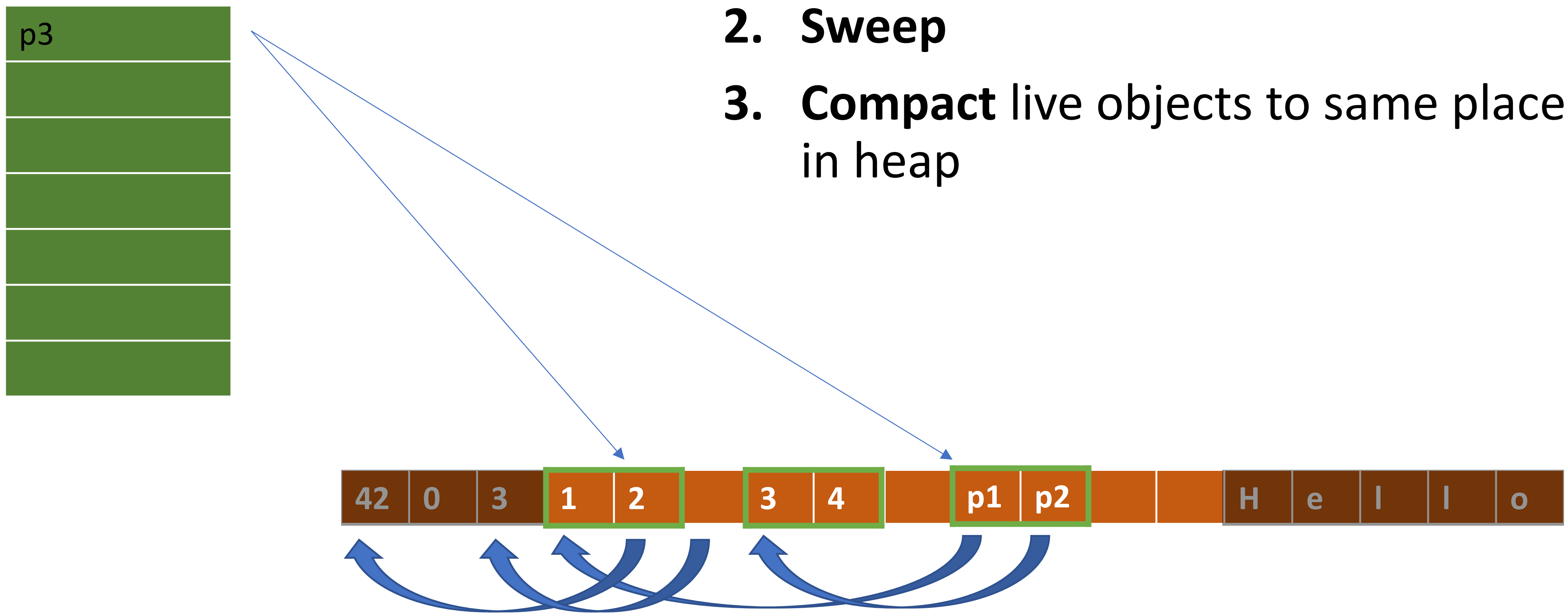
- Cons:
  - Have to sweep through whole heap (can optimize)
  - Fragmentation



1 2 3 4 p1 p2

GC Strategy #2½: Mark and compact

1. **Mark**
2. **Sweep**
3. **Compact** live objects to same place in heap



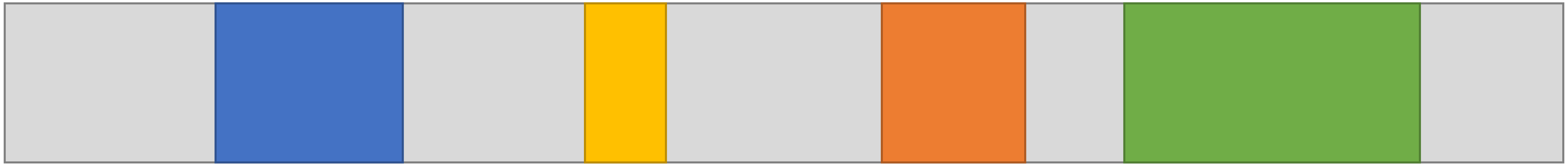
Mark and compact pros and cons

- Pros:
  - Fragmentation solved

- Cons:
  - Have to update pointers

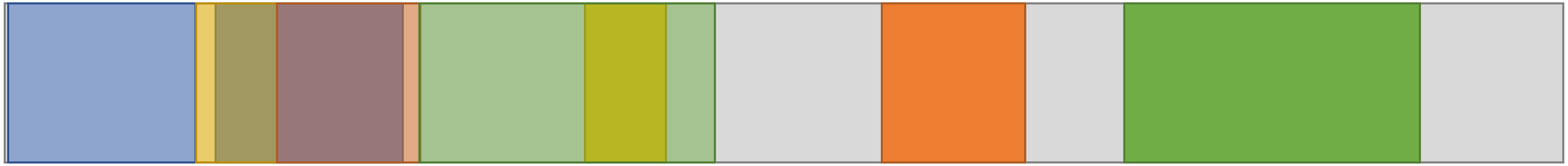
Implementing Compaction (#1): Keep a “forwarding pointer” in each object

1. Compute new locations of objects
2. Update all pointers
3. Move



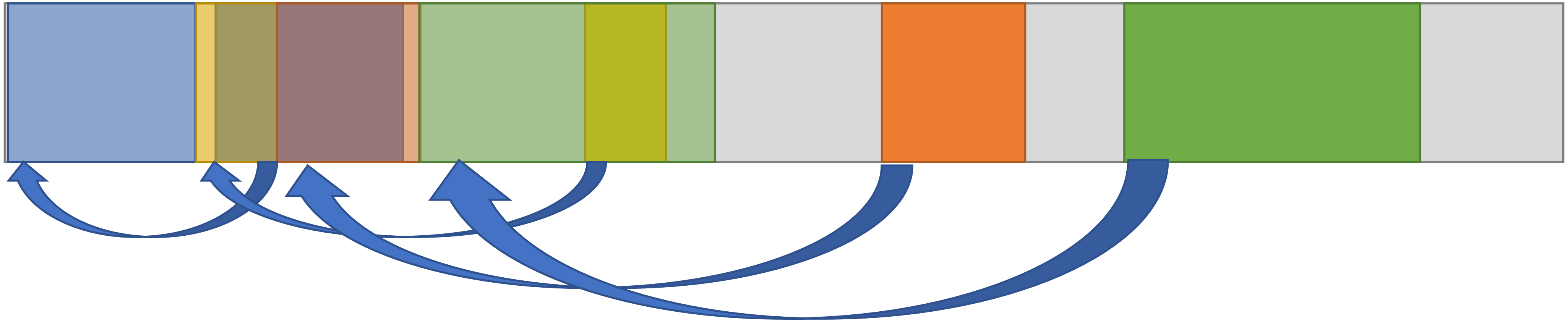
Implementing Compaction (#1): Keep a "forwarding pointer" in each object

1. Compute new locations of objects
2. Update all pointers
3. Move



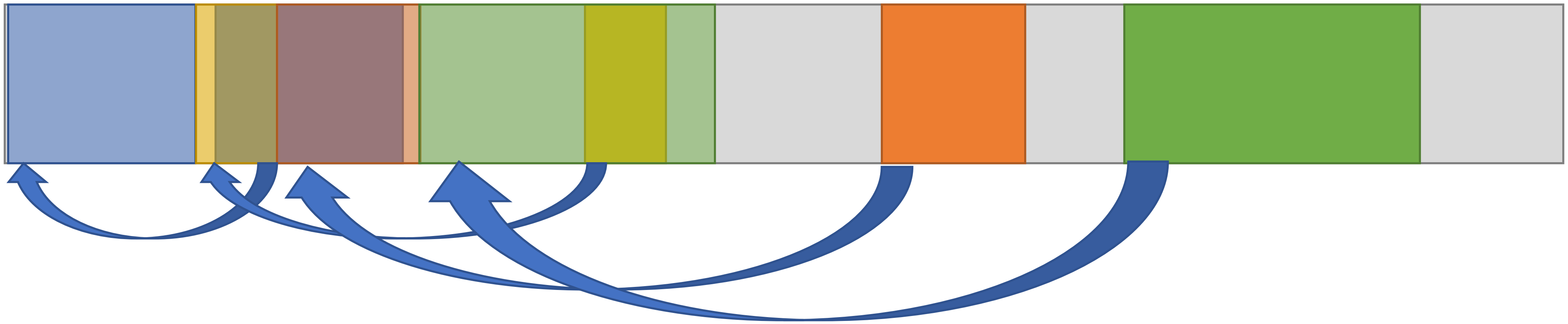
Implementing Compaction (#1): Keep a "forwarding pointer" in each object

1. Compute new locations of objects
2. Update all pointers
3. Move



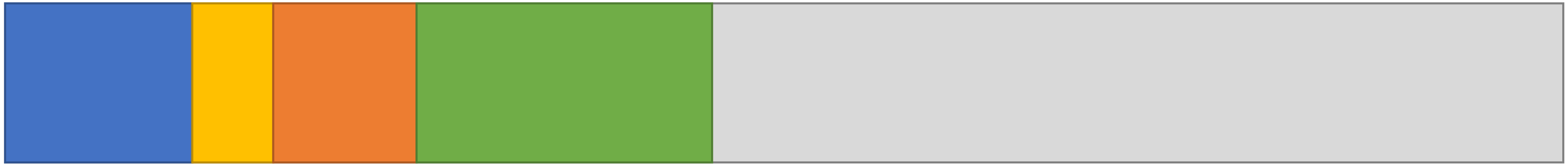
Implementing Compaction (#1): Keep a "forwarding pointer" in each object

1. Compute new locations of objects
2. Update all pointers
3. Move



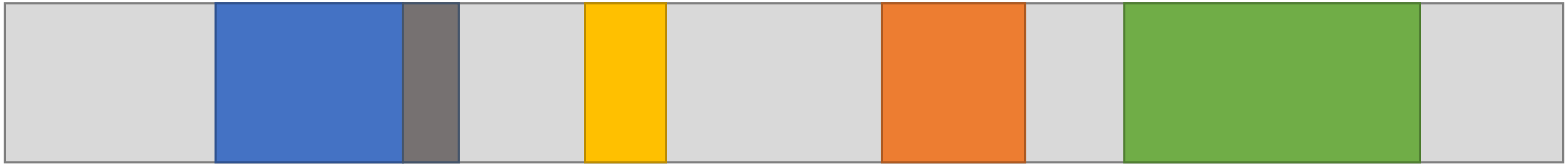
Implementing Compaction (#1): Keep a "forwarding pointer" in each object

1. Compute new locations of objects
2. Update all pointers
3. Move



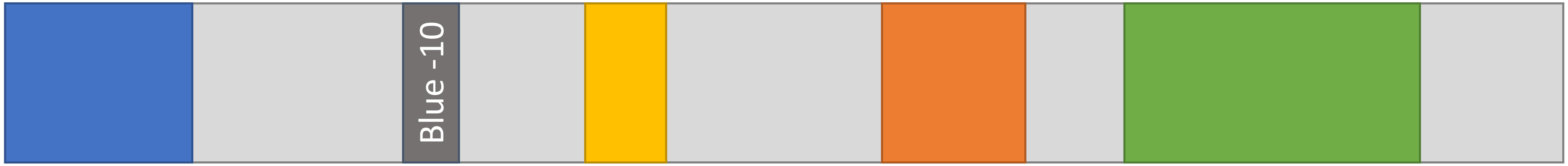
Implementing Compaction (#2): Keep a table in free space

- Table maps **groups of consecutive objects** to new offsets



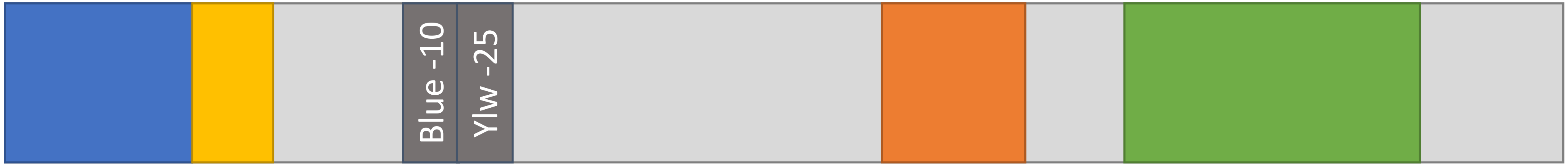
Implementing Compaction (#2): Keep a table in free space

- Table maps **groups of consecutive objects** to new offsets



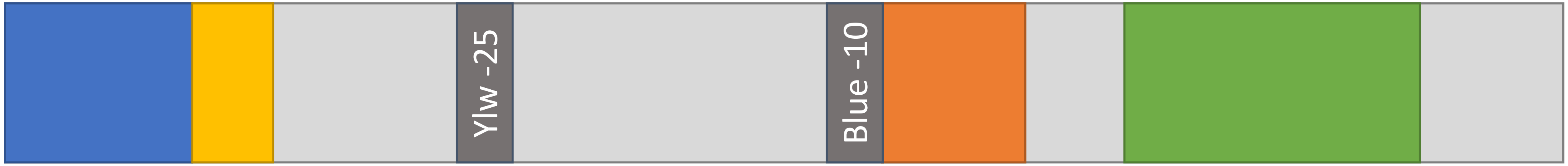
Implementing Compaction (#2): Keep a table in free space

- Table maps **groups of consecutive objects** to new offsets



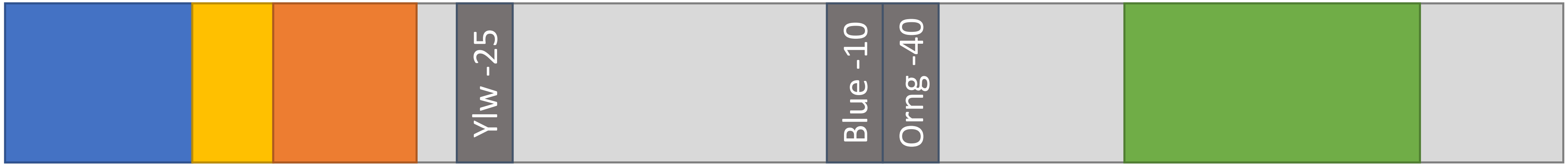
Implementing Compaction (#2): Keep a table in free space

- Table maps **groups of consecutive objects** to new offsets
- “Roll” the table into free space if needed



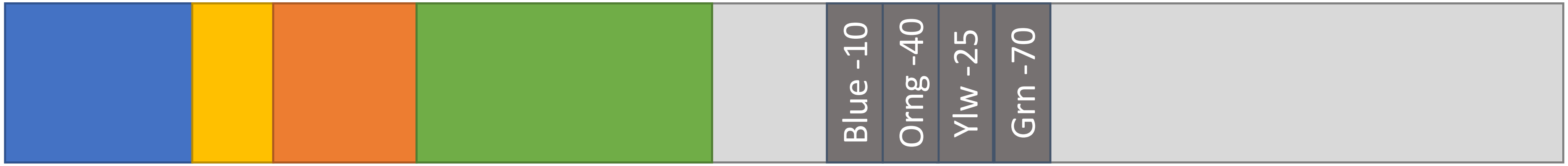
Implementing Compaction (#2): Keep a table in free space

- Table maps **groups of consecutive objects** to new offsets
- “Roll” the table into free space if needed



Implementing Compaction (#2): Keep a table in free space

- Table maps **groups of consecutive objects** to new offsets
- “Roll” the table into free space if needed
- Need to sort the table at the end



Compacting allows for **really** fast allocation

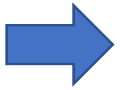
- “Bump allocation”
  - Heap pointer points to end of heap
  - To allocate N bytes:
    - Increment (“bump”) heap pointer by N
    - If we pass the end of the heap, trigger a GC
    - Return old value of heap pointer

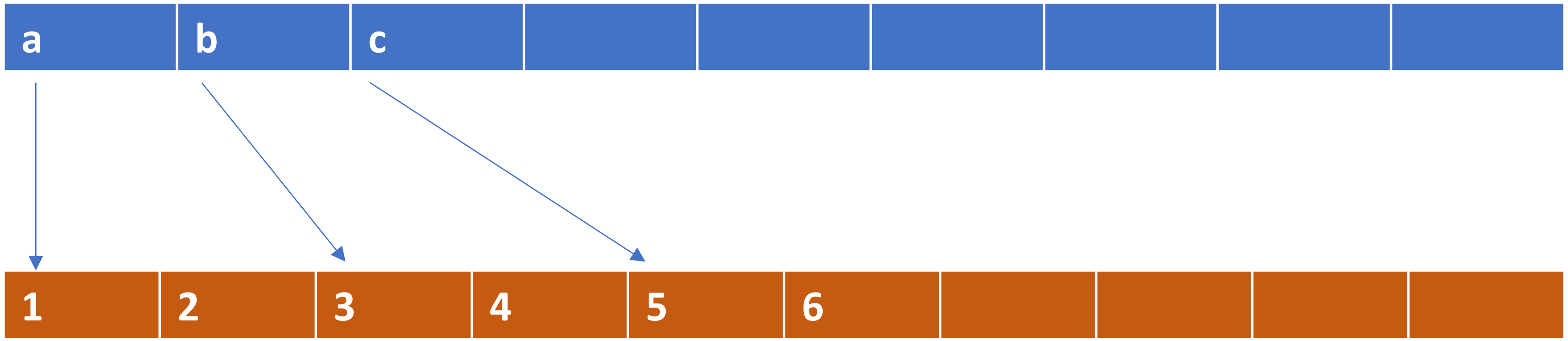
… yes. That’s it. That’s how we implement `malloc`

```
__malloc:
  lw t0,heapptr           # t0 = heap ptr
  lw t2,heapend           # t2 = end of heap
  add t1,t0,a0            # t1 = heap ptr + Nbytes
  blt t2,t1,__eom         # check if t1 > heap limit
  sw t1,heapptr           # heap ptr += Nbytes
  addi a0,t0,0            # a0 = old heap ptr
  jalr zero,ra,0          # return
__eom:                    # trigger GC
  …
```

Bump allocation

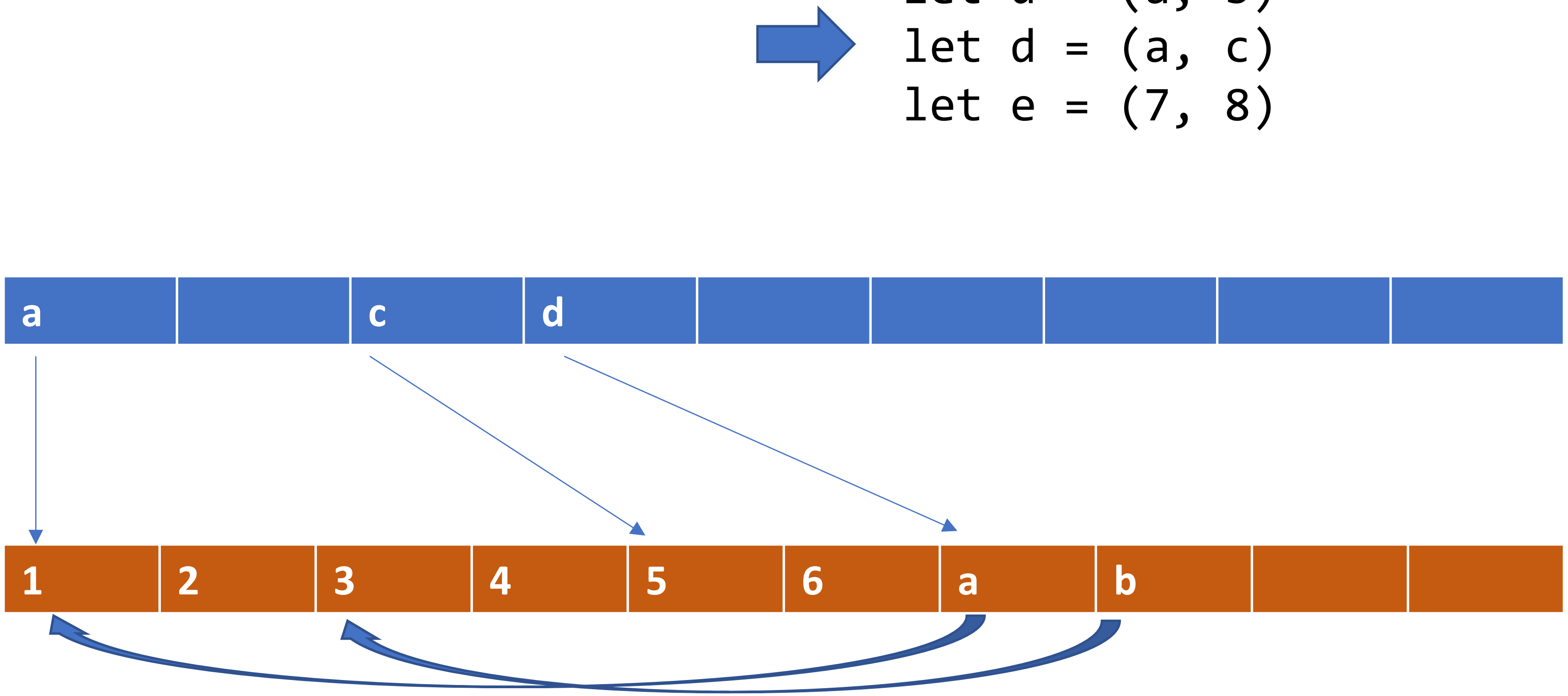
```
let a = (1, 2)
let b = (3, 4)
let c = (5, 6)
let d = (a, b)
let d = (a, c)
let e = (7, 8)
```





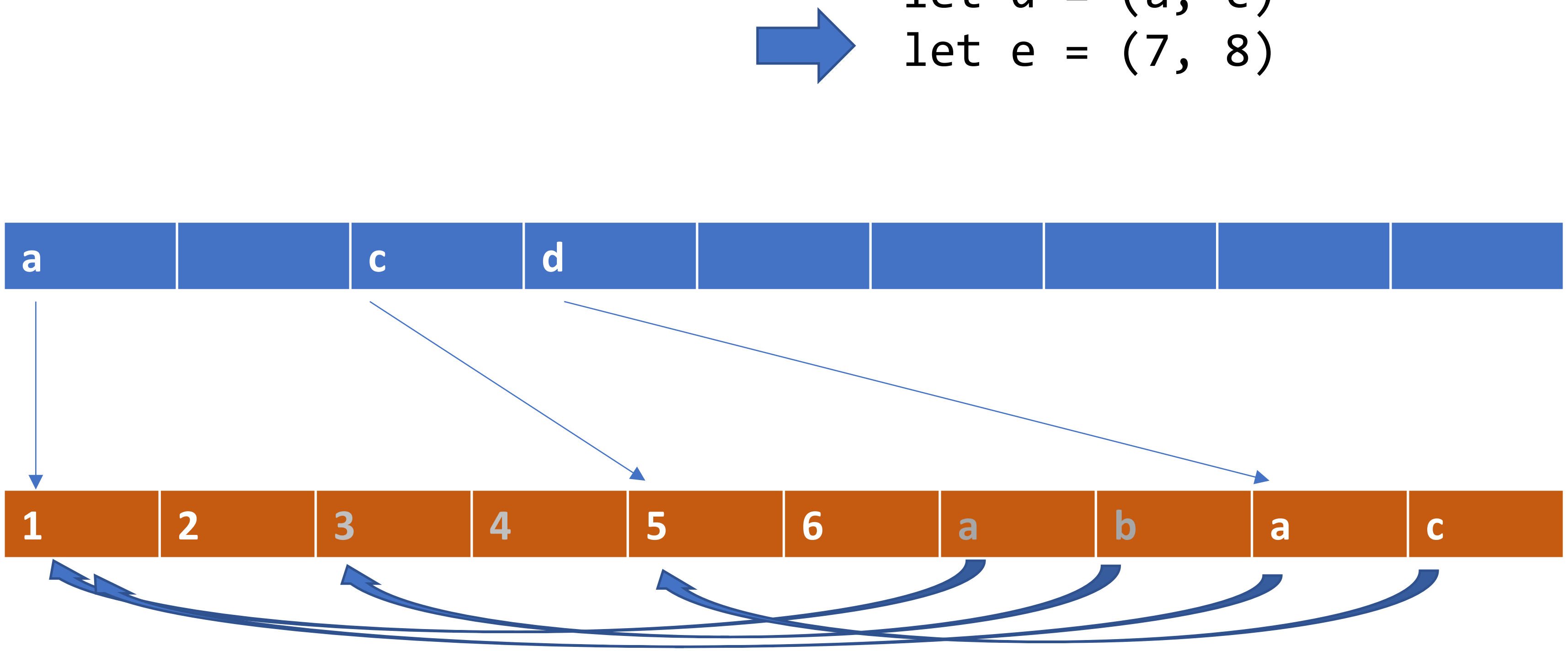
Bump allocation

```
let a = (1, 2)
let b = (3, 4)
let c = (5, 6)
let d = (a, b)
let d = (a, c)
let e = (7, 8)
```



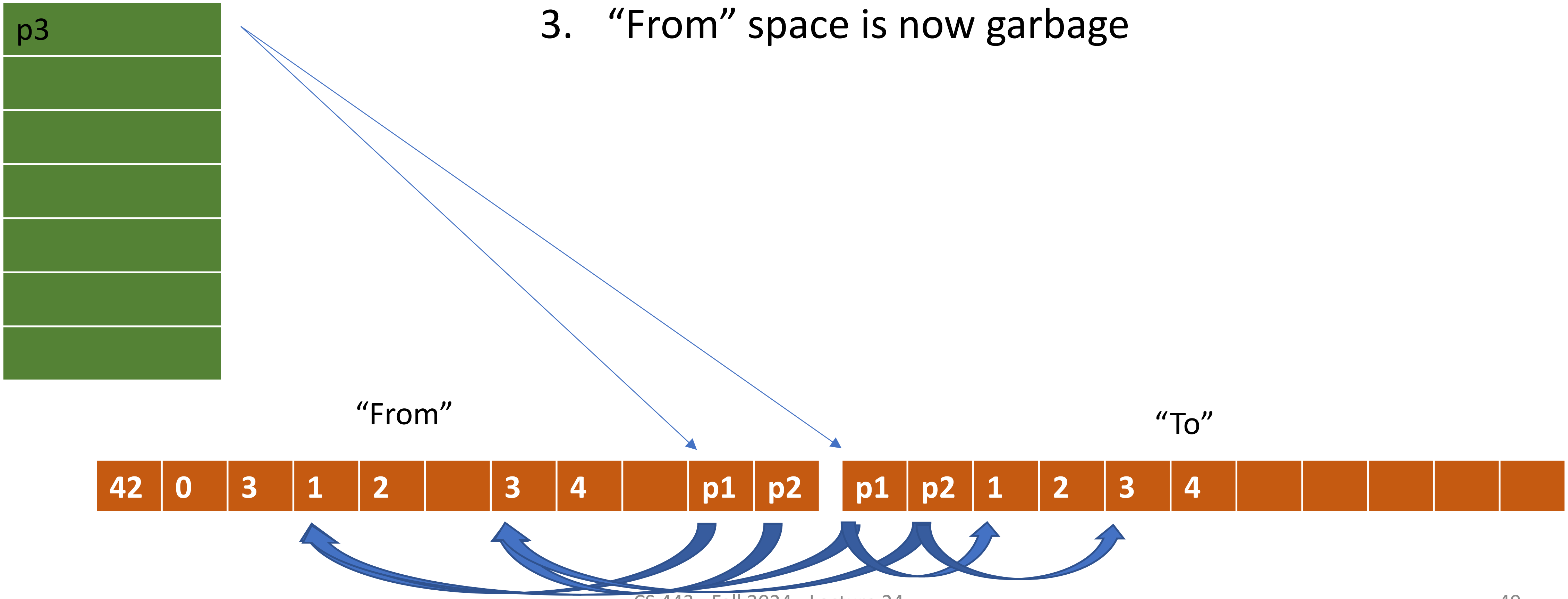
Bump allocation

```
let a = (1, 2)
let b = (3, 4)
let c = (5, 6)
let d = (a, b)
let d = (a, c)
let e = (7, 8)
```



GC Strategy #3: Copying

1. Divide heap into “from” space and “to” space
2. **Copy** live objects into “to” space
3. “From” space is now garbage

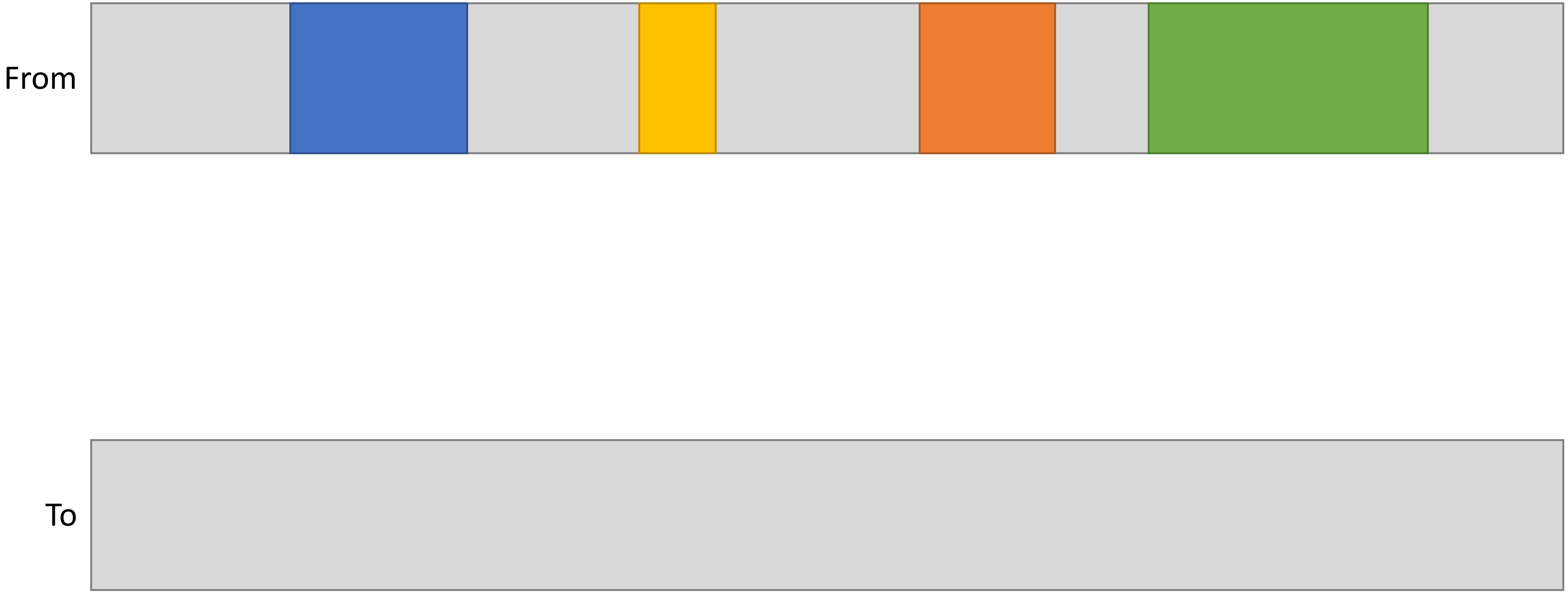


Copying pros and cons

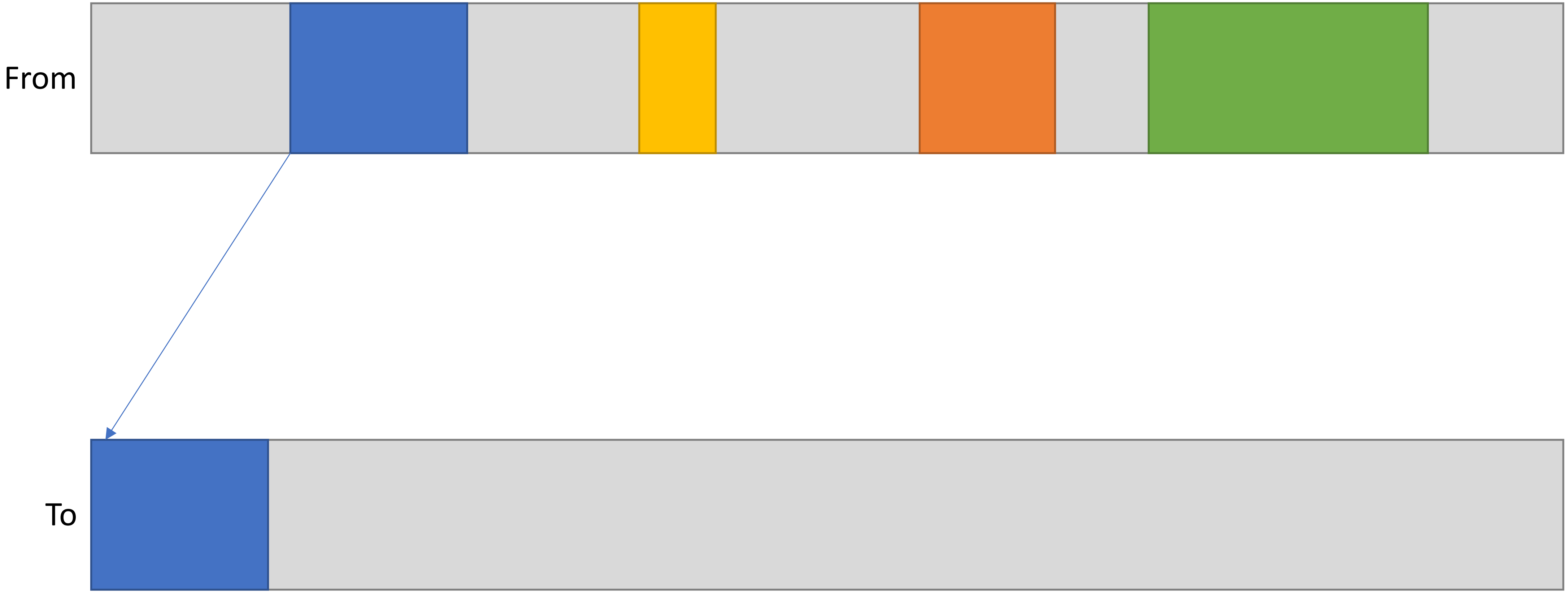
- Pros
  - No traversing of whole heap
  - No fragmentation

- Cons
  - Heap size basically cut in half
  - Have to move pointers

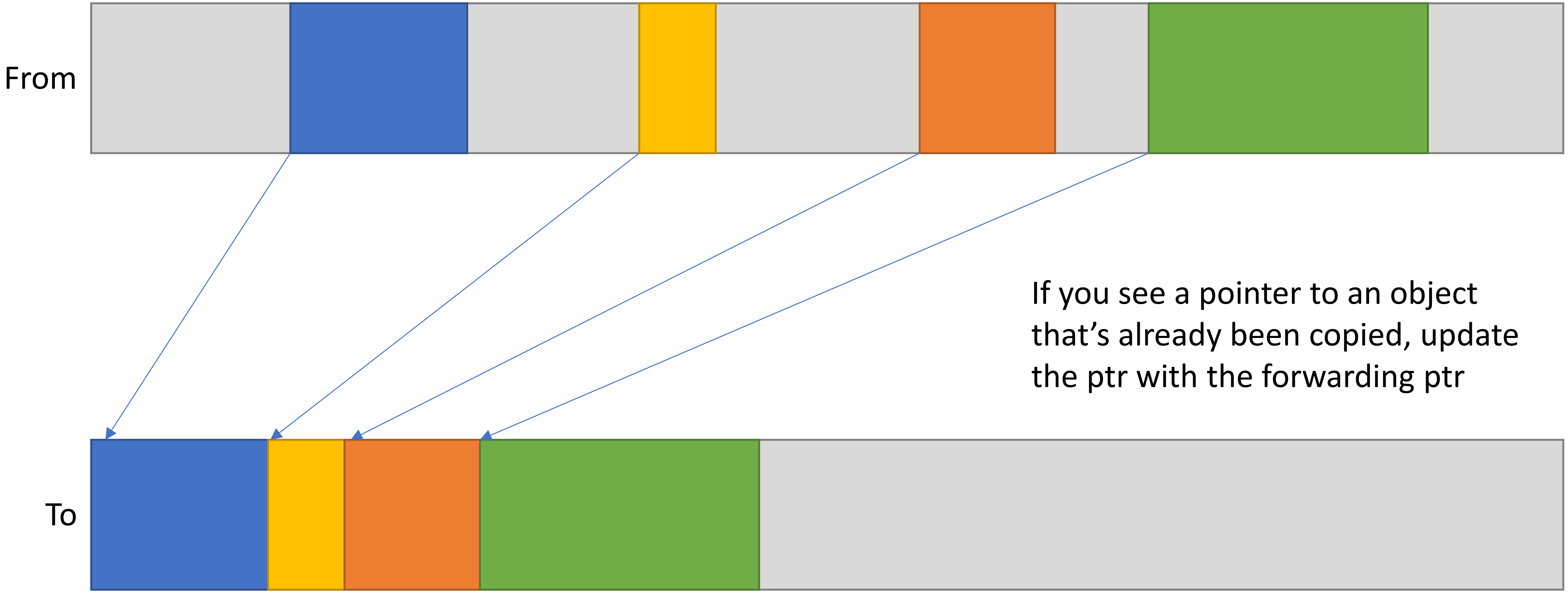
Copying Implementation: Just turn the from space into forwarding pointers



Copying Implementation: Just turn the from space into forwarding pointers

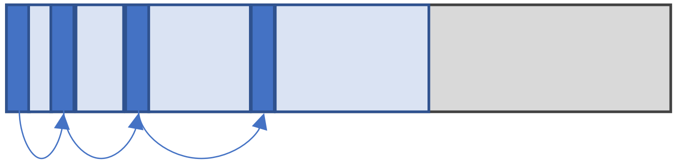


Copying Implementation: Just turn the from space into forwarding pointers



If you see a pointer to an object that’s already been copied, update the ptr with the forwarding ptr

Another side benefit of copying



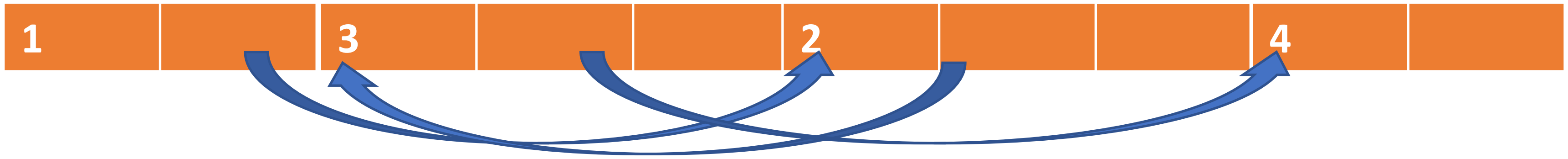
```
let rec list l n =
  if n <= 0 then l
  else
   list ((List.length
             (List.init (n mod 5) id))::l)
        (n – 1)

let l = list [] 10000
do_n_times 3 (fun _ -> traverse l)
(* Do other stuff *)
do_n_times 3 (fun _ -> traverse l)
```

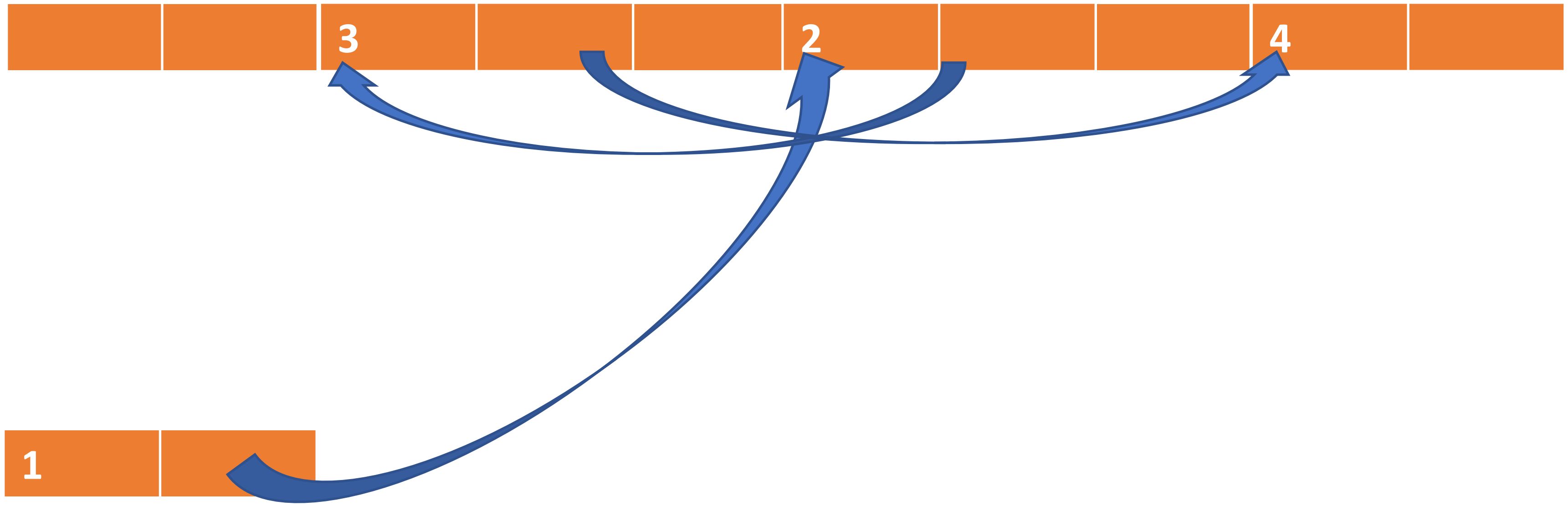
What happened here?

```
Traversed list in 0.00016s
Traversed list in 0.00016s
Traversed list in 0.00016s
Starting new major GC cycle
Traversed list in 0.00007s
Traversed list in 0.00006s
Traversed list in 0.00006s
```

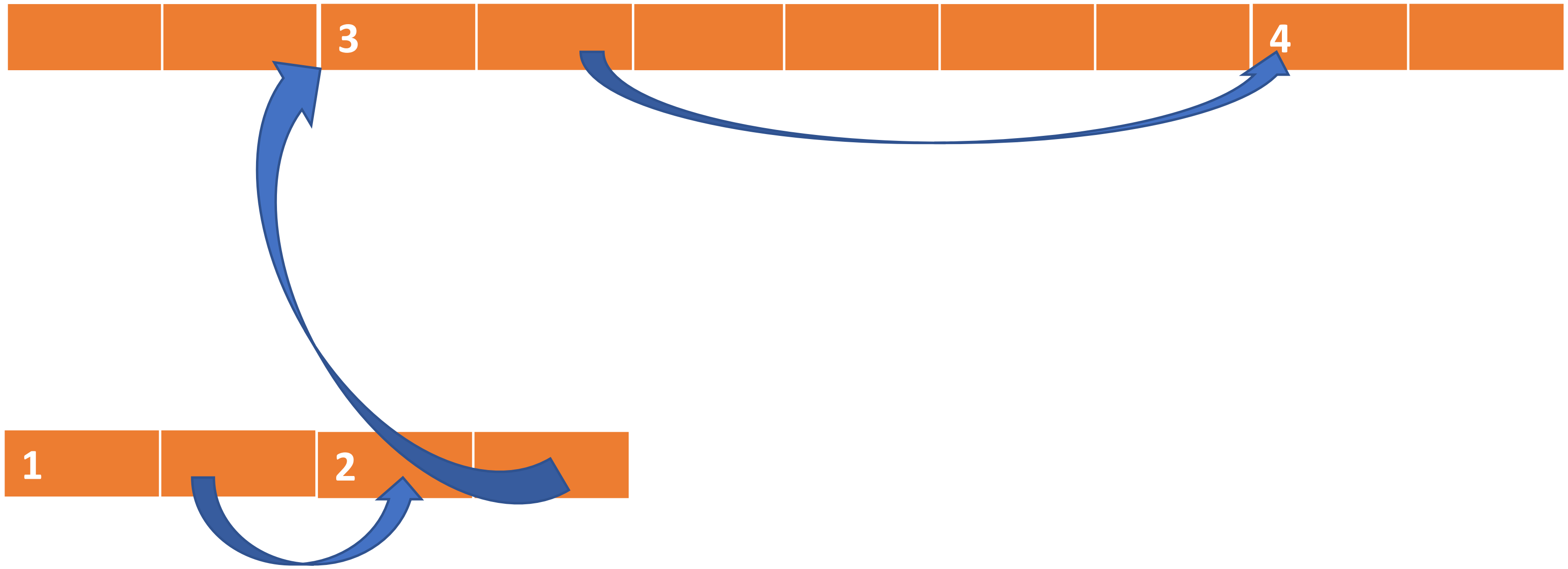
Another side benefit of copying



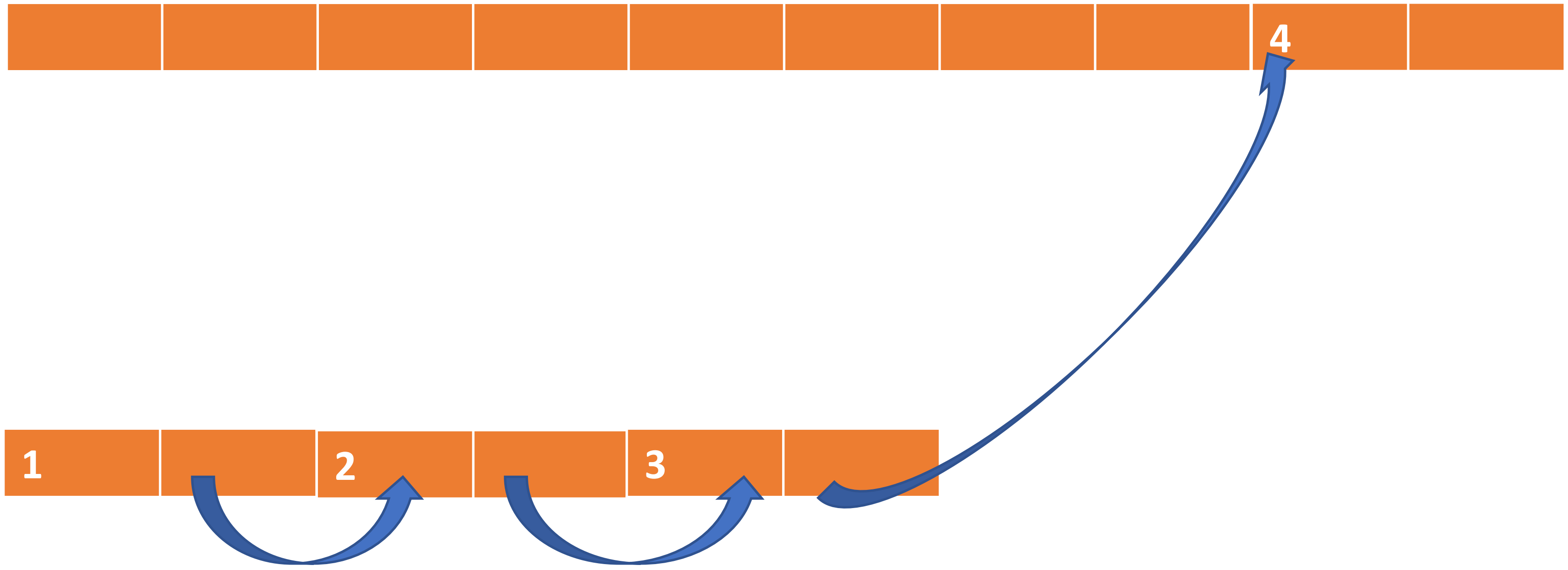
Another side benefit of copying



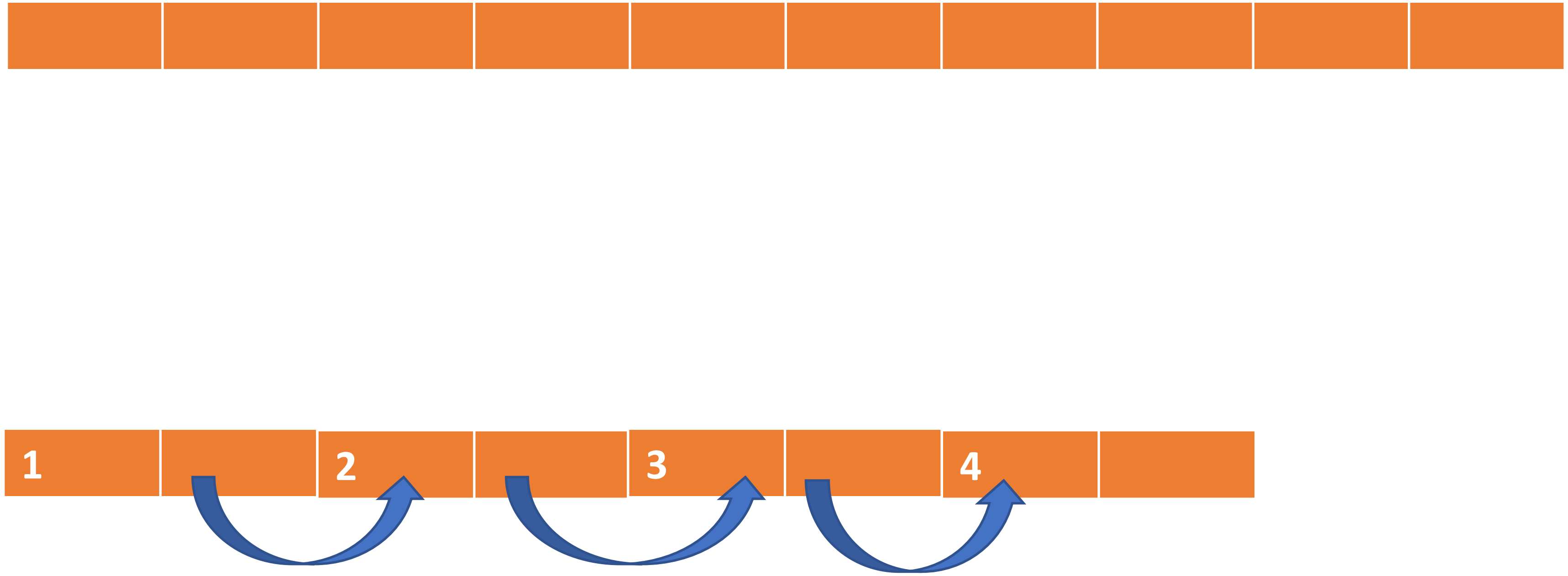
Another side benefit of copying



Another side benefit of copying

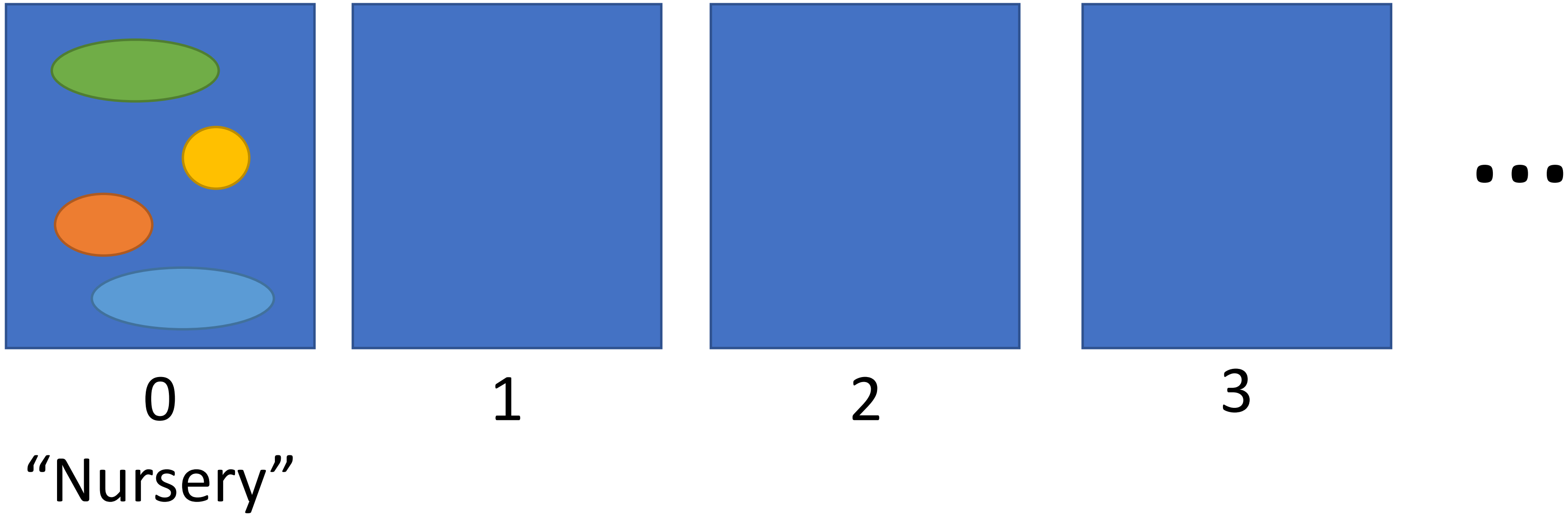


Another side benefit of copying



Generational garbage collection

- Idea: “most objects ‘die young’”
- Separate heap into areas called *generations*
- Collect younger generations more aggressively/frequently



Efficiency

- Most GCs we have discussed are “stop the world”
  - Stop program, do a collection
  - *Pause time:* amount of time a program must wait for the collector

- To reduce pause time, many real-world GCs are *concurrent* or *incremental* (do small amounts of work as the program runs)

In practice, pause times are pretty short

- Don’t let people tell you GC makes it totally impractical to use functional languages for real code

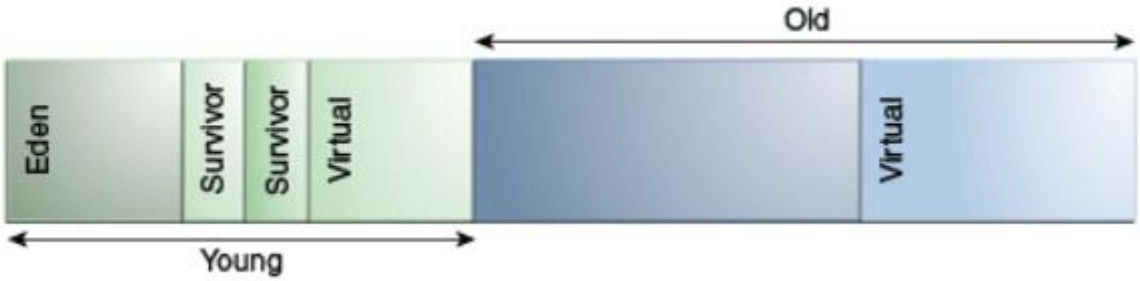
```
GC type        time ms     number          bytes           bytes/sec
-------------    -------    -------    --------------- ---------------
copying          3,063         37      2,111,703,368       689,423,253
mark-compact         0          0                  0                 -
minor                0         11              4,520                 -
total time: 19,902 ms
total GC time: 3,472 ms (17.4%)
max pause time: 433 ms
total bytes allocated: 15,794,832,336 bytes
max bytes live: 140,663,592 bytes
max heap size: 1,125,367,808 bytes
```

3472 ms / 37 = 93ms avg.

OCaml

- Two generations: *minor heap* and *major heap*
  - Allocate large objects directly into major heap
  - “Minor collections” frequent
  - “Major collections” when necessary
- Major collections are (concurrent) mark-compact
  - Not to be confused with *parallel* GC (GC runs on multiple threads to reduce pause time)

Java (HotSpot JVM)



- Generational
  - Eden (nursery)
  - Live objects copied from Eden to one of two “survivor” spaces
  - Copying collection used to copy between survivor spaces
  - After a certain number of copies, moved to “old” generation
- Several different collection strategies available for different applications

Python

- Reference Counting*
    - Generational
    - Cycles?
    - It’s complicated