Memory Safety: Garbage Collection and Borrow-checking

Dangling Pointers/References

References can become **dangling** (also called **wild**) when they reference an invalid destination. These include pointers to deallocated memory as well as reallocated one (e.g., after we freed it). In all these cases, dereferencing a dangling pointer can lead to **unpredictable behaviour**, since the referenced memory location contains unexpected data.

```
{
 char *dp = NULL;
 {
  char c = “a”;
  dp = &c;
 }
}
```

Usually, wild pointers—intended as **uninitialised pointers**—are easier to catch, by looking at accesses to uninitialised variables.

On the contrary, detecting dangling references is harder and entails tracking and reasoning on all the possible combinations of code where pointer values go through (sometimes this is even impossible, e.g., in compiled libraries).

Tombstones

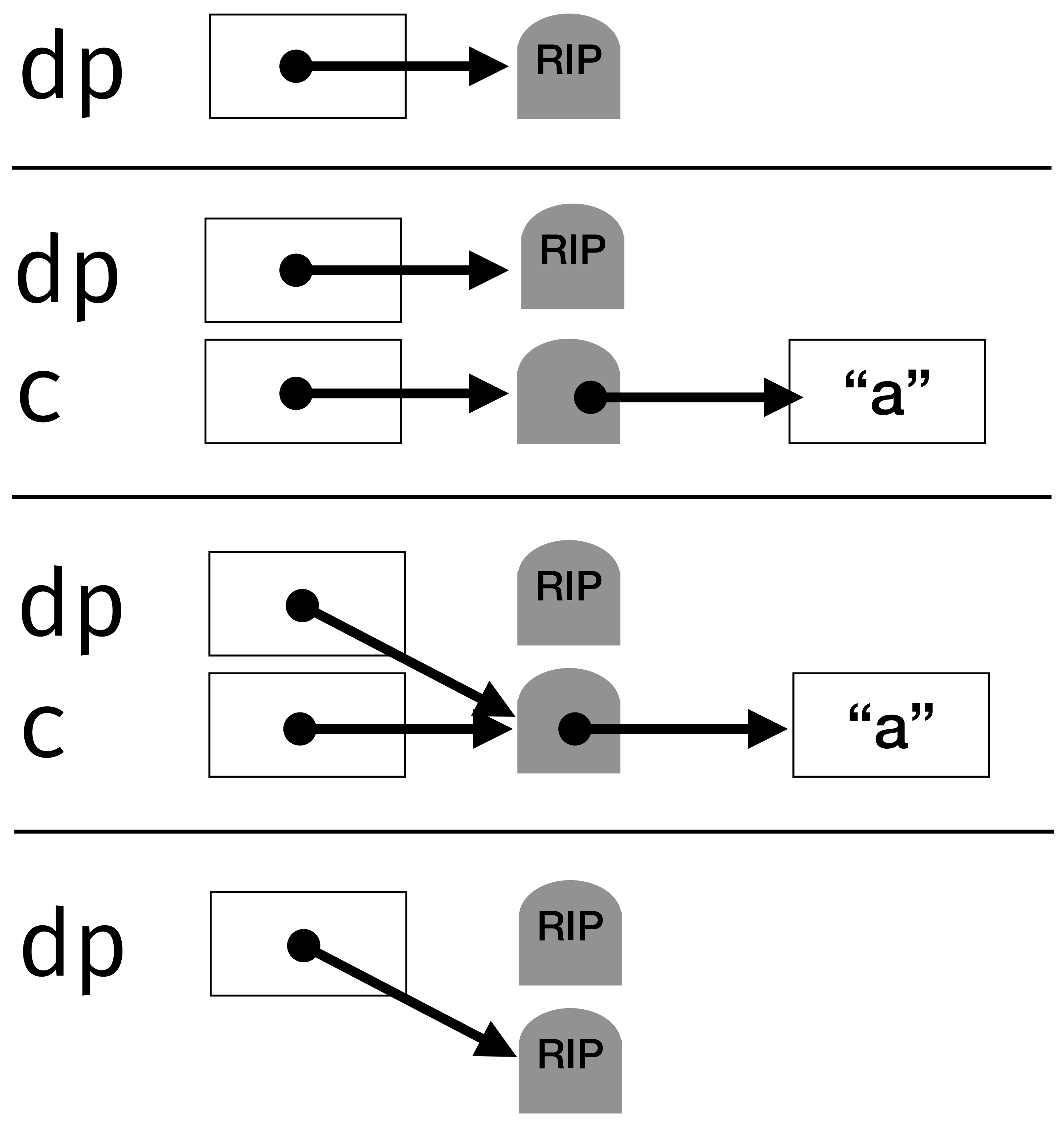
A way to handle dangling references are **tombstones**.

Essentially, tombstones work by associating **every allocation** accessed by a pointer with a **companion** additional word allocated on memory, called the **tombstone**.

At initialisation, the tombstone contains the address of the allocated element, while the pointer itself receives the address of the tombstone.

All pointer dereferences become **two-hop lookups**, where we first access the tombstone and then the pointer of the object.

```
{
 char *dp = NULL;
 {
  char c = “a”;
  dp = &c;
 }
}
```



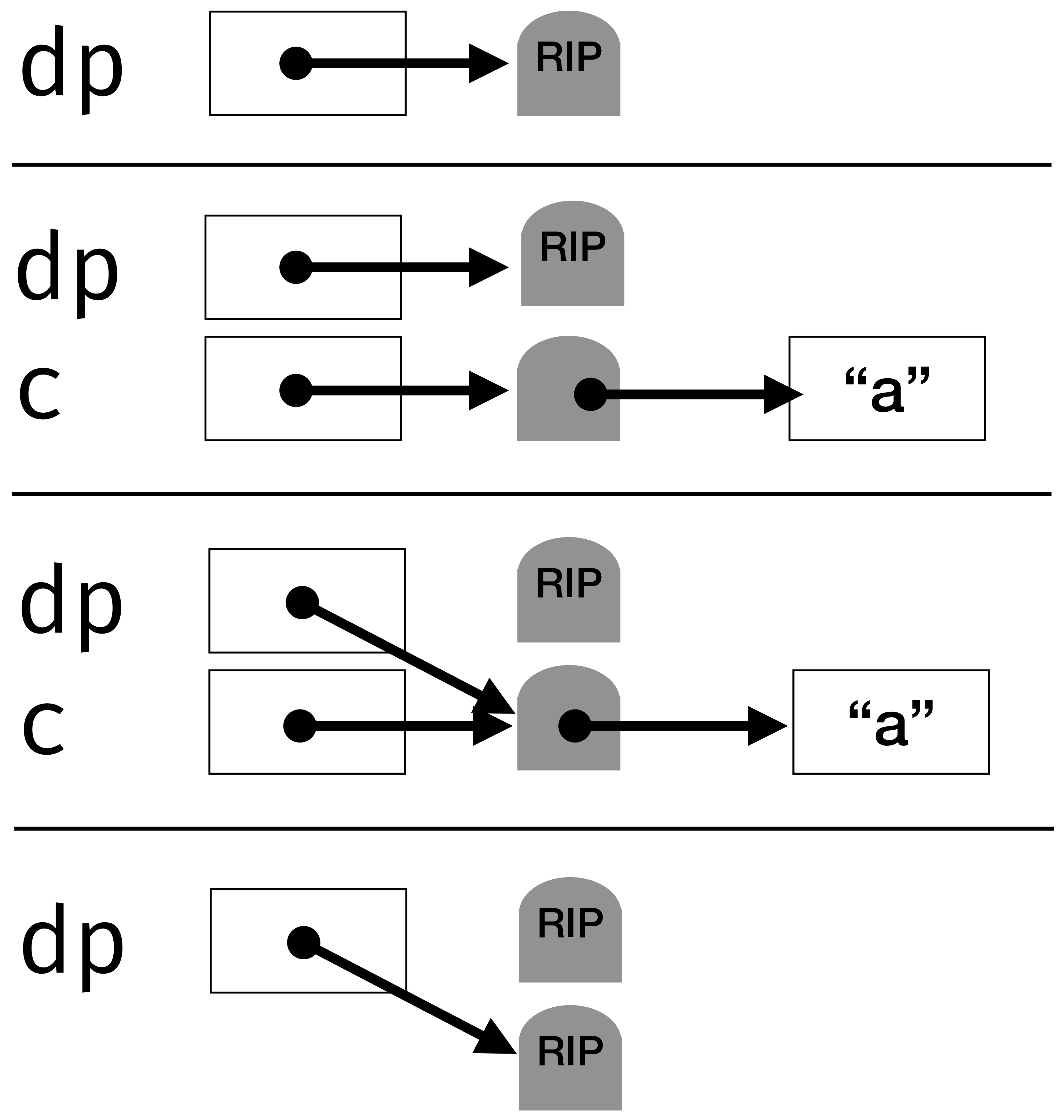
Tombstones

The two-hop lookup **keeps all possible duplications of the pointer in check, as they all point to the same** tombstone.

As long the the memory location of the pointed object is allocated, its tombstone points to it.

When we deallocate the object, we mark it as “dead” with a dedicated word in its tombstone (hence, the name) and raise an error at any successive dereferencing.

```
{
 char *dp = NULL;
 {
  char c = “a”;
  dp = &c;
 }
}
```



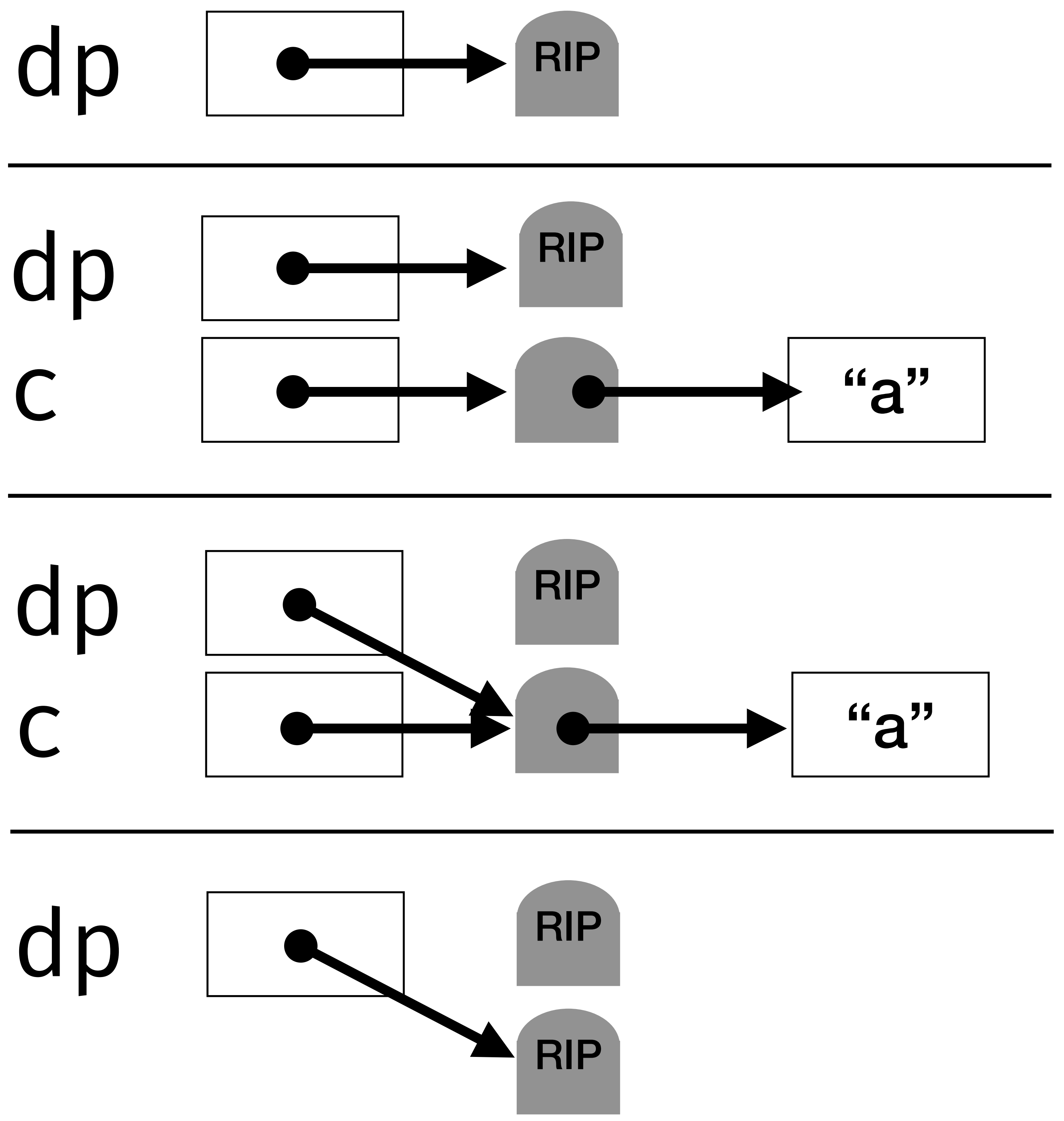
Tombstones

Although simple, the tombstone mechanism has a hefty bill to pay.

Efficiency-wise, we are **duplicating** all pointer **dereferences** (the two-hop lookup) and will spend some time also in creating the tombstone.

Space-wise, for each pointer in our program we have a memory **location occupied by its tombstone** (both for the heap and the stack), which we **cannot reclaim** (unless we employ some clever reference-counting techniques), resulting in the possibility of running out of space in the graveyard.

```
{
 char *dp = NULL;
 {
  char c = “a”;
  dp = &c;
 }
}
```



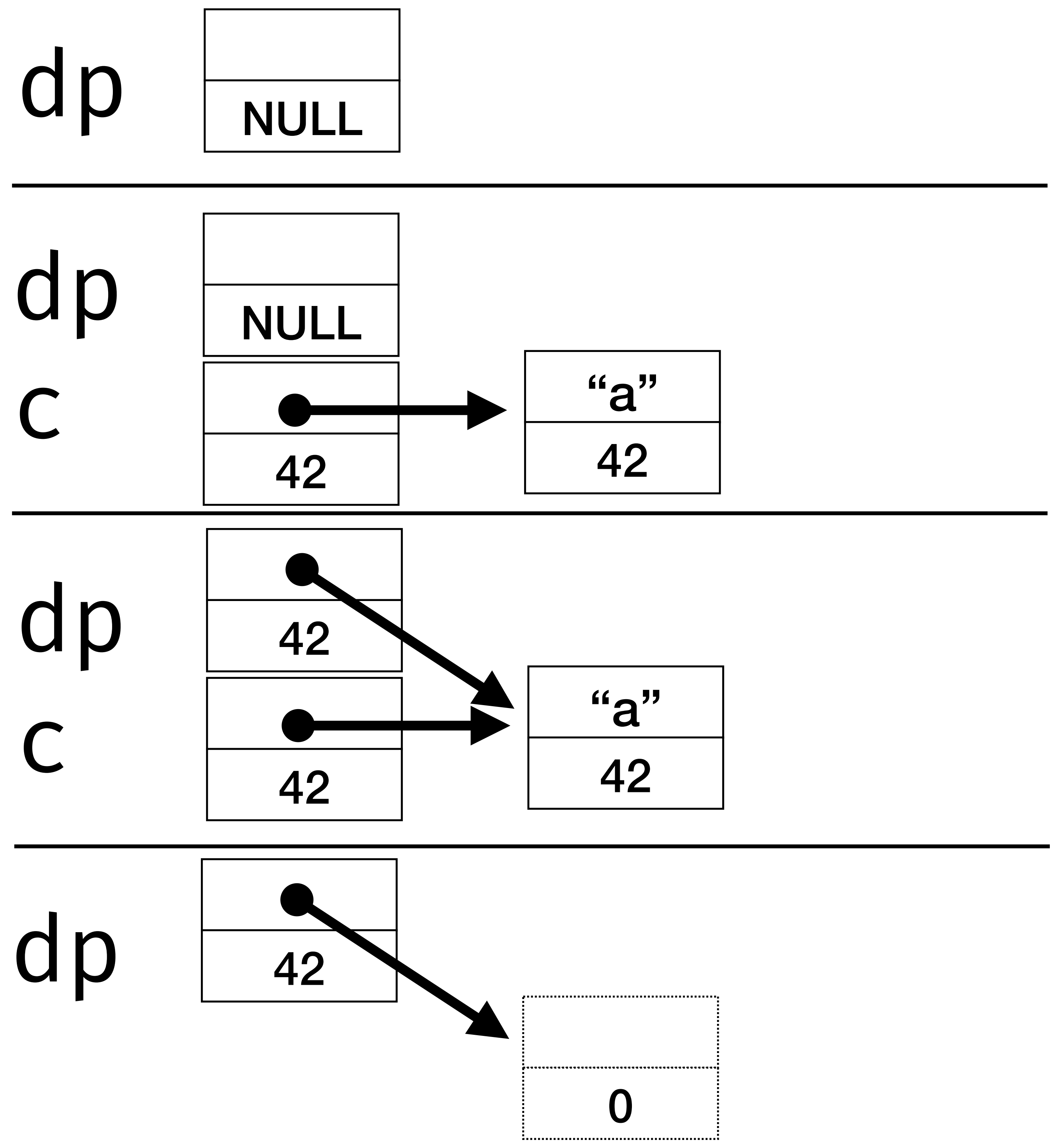
Locks and Keys

Locks and keys are an alternative to tombstones, but only for pointers to the heap: every time we **allocate an object on the heap**, we **associate** it **to a "lock"** made of a memory word that stores a random value. Now, a pointer consists in a pair: the actual address and a "key", i.e., a memory word initialised to the value of the lock of the pointed object.

Then, each time we **dereference a pointer**, we **check that the key can open the lock**, i.e., that the two words coincide. This also means that, when we assign the pointer, the assignment copies both the pointer and the key, used to check the correspondence.

At deallocation, we both delete the object in memory and set its lock to a canonical value (outside the domain of keys) and invalidate any possible successive dereferencing, which, like in tombstones, would raise an error.

```
{
 char *dp = NULL;
 {
  char c = “a”;
  dp = &c;
 }
}
```

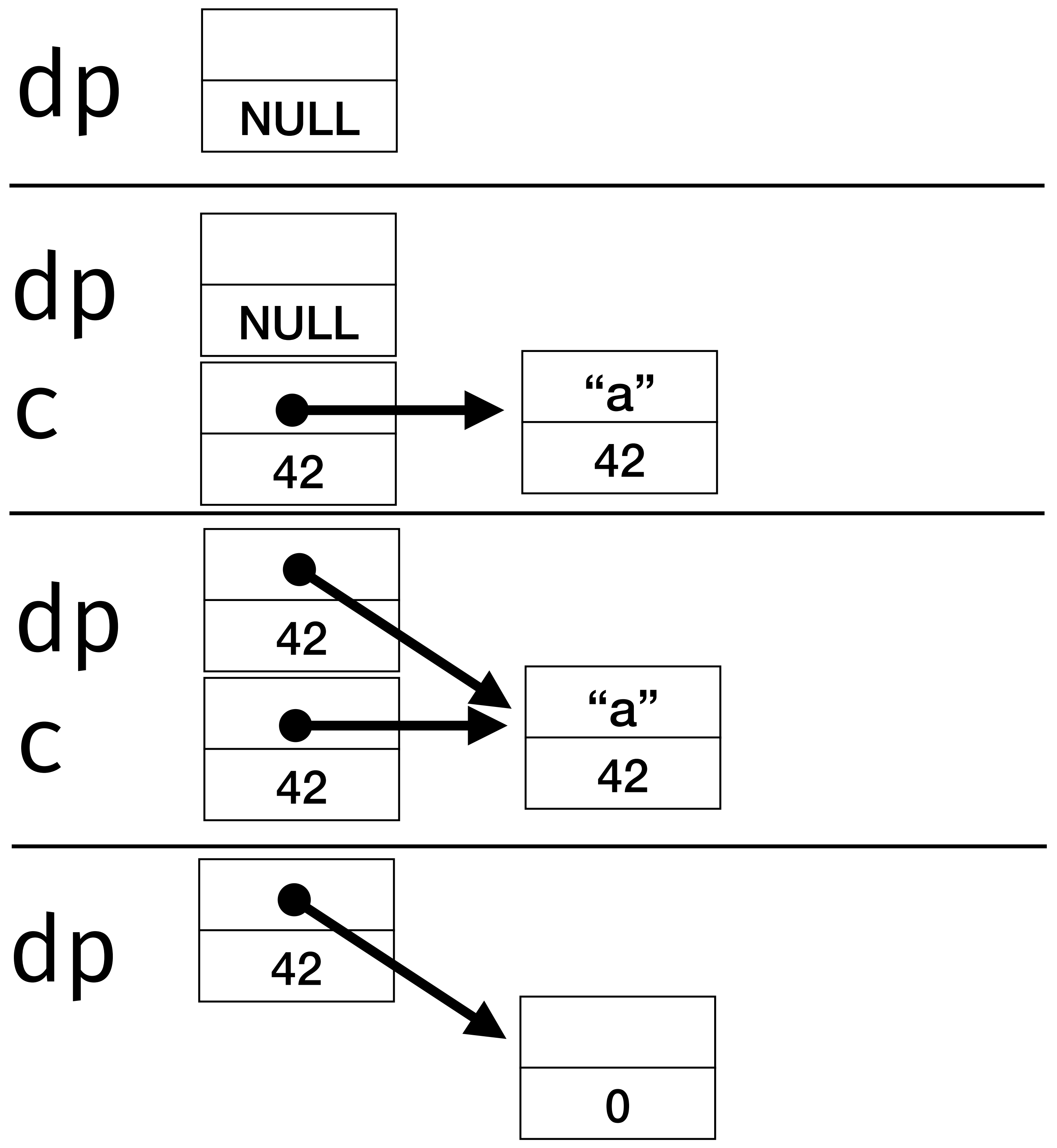


Locks and Keys

It can happen that successive allocations could use the memory area previously used as a lock (for another lock or for other purposes), however it is statistically unlikely that an error will not be detected because a former lock happens to have the same value it had before its cancellation.

Locks and keys also have a significant cost. In terms of **space**, they cost even **more than tombstones**, since an **additional word** is **needed for each pointer**—although both locks and keys are deallocated together with the object or pointer they are part of. From the point of view of the **efficiency**, locks and keys determine costs both at creation, assignment, and dereferencing, since now we **always perform an equivalence check**.

```
{
 char *dp = NULL;
 {
  char c = “a”;
  dp = &c;
 }
}
```



Garbage Collection

In languages without explicit memory deallocation on the heap, we need a mechanism to automatically handle the deallocation of objects in the heap no longer in use. The general term for one such mechanism is “garbage collection”—introduced for the first time in LISP (around 1960) but present in many modern languages such as Javascript, Python, and Java—and we call “garbage collector” an implementation of said mechanism.

From a logical point of view, the operations of a garbage collector include:

1. **Garbage detection**: to detect whether objects in memory are in-use or not;
2. **Garbage collection**: release the memory occupied by not in-use objects.

Following the nomenclature, we call “garbage” those objects not in-use that the garbage collector can detect and collect.

Garbage Collection

Since garbage collection is one of the main determinants in the performance of garbage-collected languages (e.g., the Java runtime provides 7 different garbage-collection algorithms, tweaked for different execution scenarios), real-world, modern techniques are quite performant and sophisticated. Here, we will see the most common ones.

In general, how garbage collection works depends on how detection works—and garbage detection and garbage collection are frequently performed in close temporal succession.

Moreover, it becomes **easier to work on the objects in memory** if the collector knows their shape/**boundaries** and, similarly, **what locations of an object correspond to pointers**—since the garbage collector needs to follow those pointers to detect whether deallocating an object would make other objects, pointed by the latter, garbage.

We can provide this information both statically and dynamically (with the usual expressiveness vs performance trade-offs) by letting the compiler/runtime associate each object to a descriptor of their type, reporting, e.g., the size and the offsets of the location of pointers. Then, at runtime, the garbage collector would use this association to simplify the traversal of the objects in memory.

Garbage Collection • Reference Count

The simplest way we can identify garbage is by finding those objects that have no pointers to them. The technique of **reference counters** follows this definition and is probably the most basic way to realise a garbage collector.

When we allocate an object on the heap, the runtime also initialises a reference counter (an integer, inaccessible to the programmer) of that object and it makes sure to keep the counter of each object synchronised with the number of active pointers to that object.

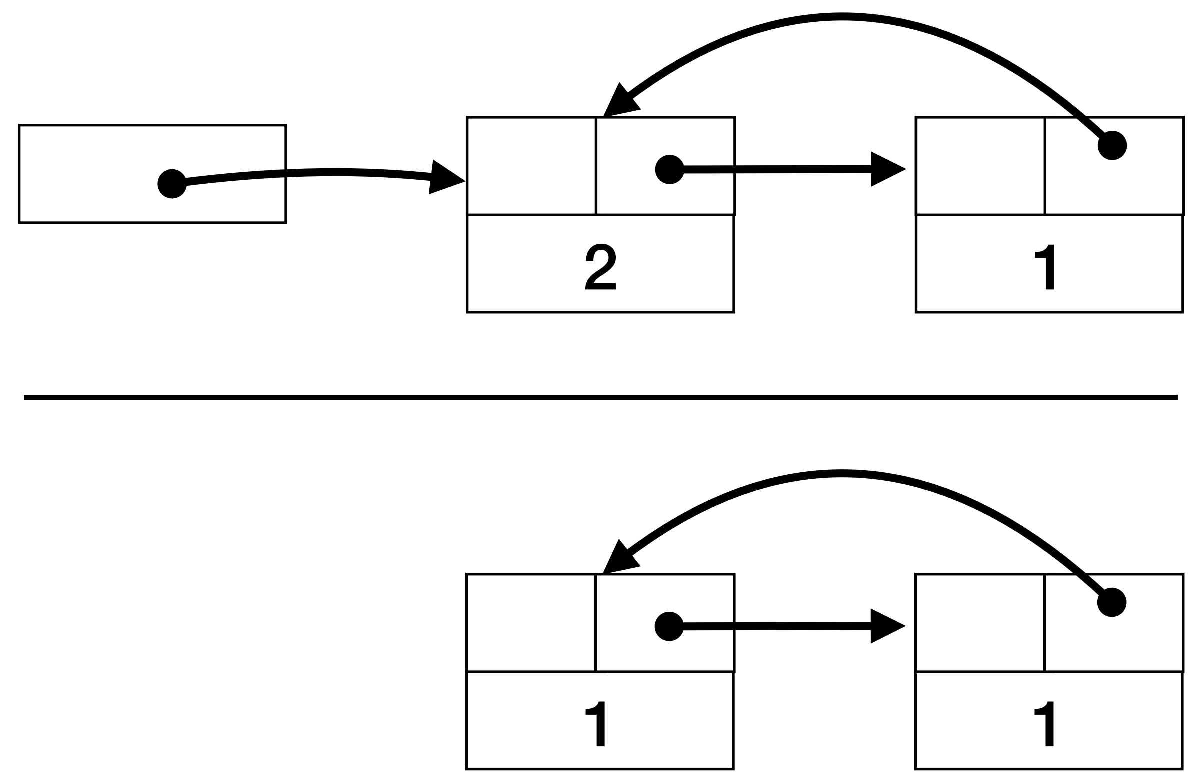
Hence, at the time of object creation, its counter has value 1. Then, the runtime **increases** the counter each time we assign its pointer to a pointer variable and **decreases** the counter each time it loses a pointer variable, e.g., because the pointer variable went out of scope or we re-assigned it to another pointer.

Garbage Collection • Reference Count

When a reference counter reaches 0, its object becomes garbage and we can deallocate it from memory.

As mentioned, this action might not just affect the object in question and rather concern other objects (pointed by the object), whose counters we need to decrease.

Hence, when marking an object as garbage (i.e., setting their counters to 0), we first need to perform a visit of the object and decrement all the pointers it refers to (and possibly marked the related objects as garbage).



Since this is a recursive visit of the memory structure from the object, one of the main limitations of this technique is its **impossibility to deal with recursive structures**—this is not a problem of the algorithm but rather a limitation of the definition of garbage of reference counting.

Garbage Collection • Mark and Sweep

Mark and sweep takes its name from how it performs detection and collection.

Mark indicates that detection uses markings to identify garbage, following two steps: first, it traverses the heap and marks all objects as garbage, second, it traverses the stack, it follows the active pointers to the objects in the heap, and it removes the previous markings from the ones we visit (recursively);

Sweep results in a flat visit of the heap to collect all marked objects.

Root Set
Object 1
Object 2
Object 3
Object 4
Object 5
Object 6
Object 7
Object 8
Root Set
Object 1
Object 2
Object 3
Object 4
Object 5
Object 6
Mark
Object 7
Object 8
Root Set
Object 1
Object 2
Object 3
Object 4
Object 5
Object 6
Sweep
Object 7
Object 8

Garbage Collection • Mark and Sweep

Unlike reference-count garbage collectors, mark-and-sweep collectors are not incremental—they do not free memory as soon as they detect garbage.

On the contrary, it is the runtime that invokes them when it deems it useful, e.g., when the heap is exhausting its available memory.

Mark-and-sweep garbage collectors are also an example of **stop-the-world garbage collection**, where the garbage collector needs to completely halt the execution of the program to make sure to see all objects, i.e., that no new objects are allocated and no existing objects become unreachable while the collector is running.

Mark-and-sweep (and stop-the-world) techniques are **more performant** and **easier to implement than incremental ones**—which slow programs and demand more memory for their bookkeeping functions. However, they have two major drawbacks. First, the pausing has a detrimental effect on the performance and responsiveness of the program, making stop-the-world garbage collection unsuitable for highly interactive programs. Second, without memory-allocation policies that contrast memory fragmentation, the performance of the mark phase depends on the size of both the heap (first mark phase) and the stack (second mark phase).

Pointer reversal: marking objects with minimal memory usage

The mark phase is naturally recursive and the obvious implementation needs a stack whose maximum depth is proportional to the longest chain through the heap. However, if the runtime decides to invoke the collector because the program exhausted its allocated memory, we might not have the space to hold the stack needed by the collector to work—remember that, usually, stack and heap grow from the opposite ends of a linear vector, so, exhausting the heap means we exhausted also the stack.

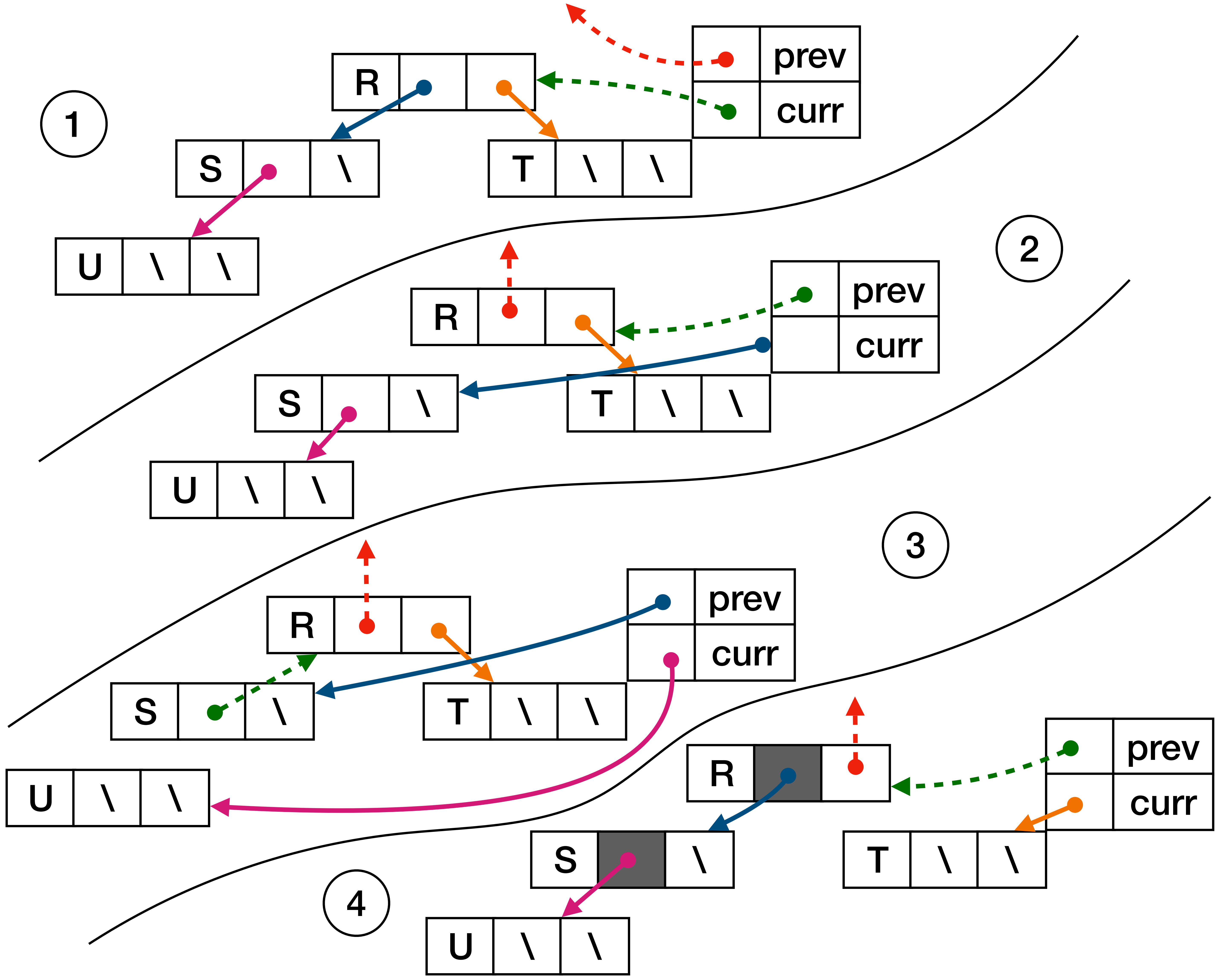
To minimise the memory required by this step we can **encode the stack in the already-existing fields in the heap,** using a technique called "pointer reversal". Specifically, since the collector explores the path from a given object, it can reverse the pointers it follows, so that each points back to the previous block instead of going forward to the next.

Pointer reversal: marking objects with minimal memory usage

The technique requires just two pointers to work: the **curr** pointer, which indicates the object currently under examination, and the **prev** pointer, which indicates the object that preceded the current in the visit.

As the garbage collector moves from one object to the next, it changes the pointer it follows to refer back to the previous object. When it returns to a visited object, it restores the pointer (to the **curr** value).

To avoid loops in visits, the collector marks the visited/reversed pointers (shaded, on the right), to distinguish them from unvisited pointers.

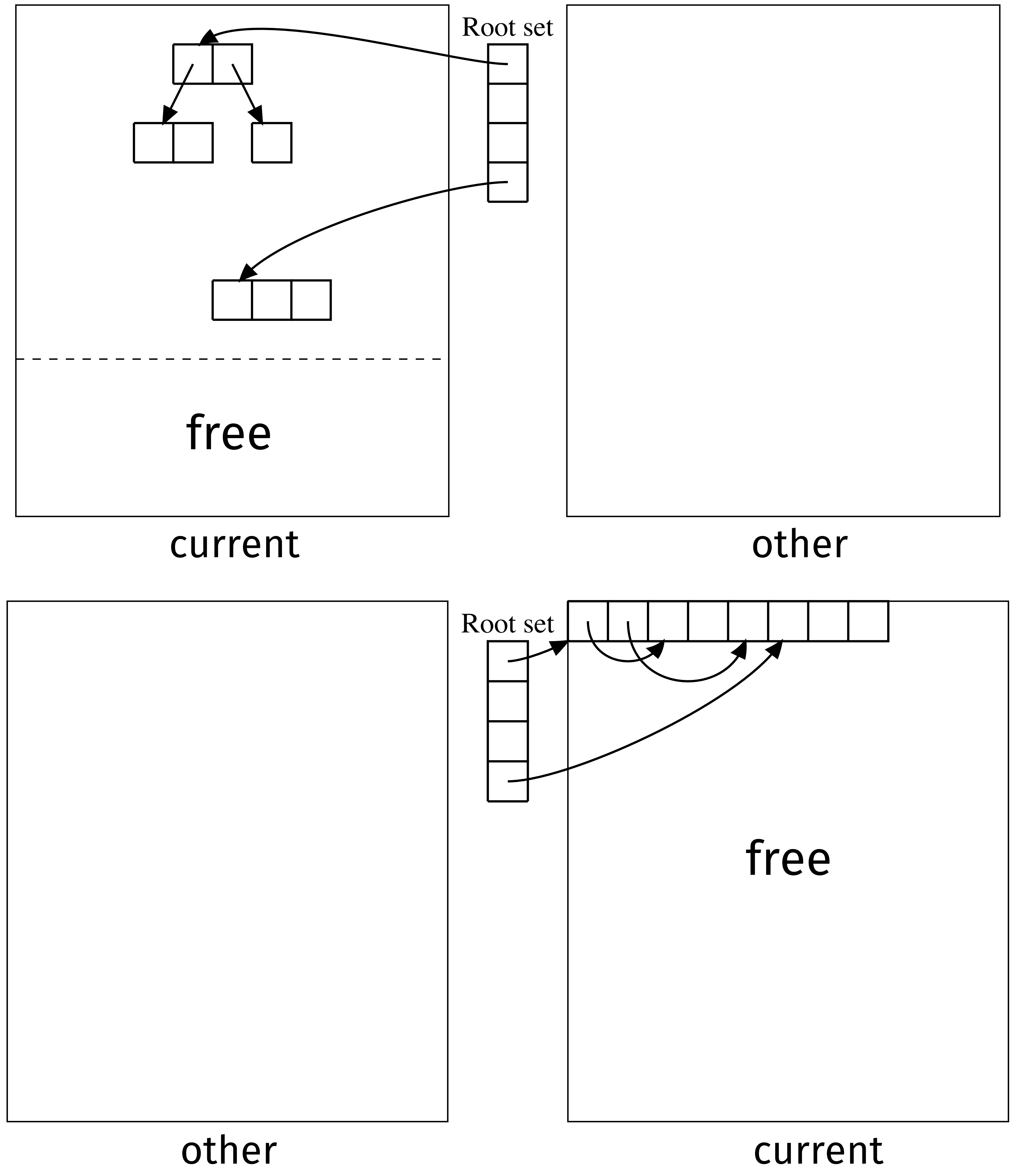


Stop and Copy

An evolution of the mark and sweep technique is “stop and copy”, which achieves compaction while simultaneously **eliminating the first phase of the marking and the sweep**.

The technique works by **dividing the heap into two regions of equal size**. All allocations happen in the “current” half while the “other” is empty. Then, when the “current" half is almost full, the collector explores the “current” half, from the root set, and **copies each reachable object contiguously into the “other” half**. When the collector finishes its exploration, it swaps its pointers to the “current” and “other” halves.

While this method halves the amount of available heap memory, **the time required by a stop-and-copy collector is proportional to the amount of non-garbage objects** on the heap and **increasing** the **amount of memory available decreases the frequency of invocation of the collector** is called and, thus, the total cost of memory management.



Memory safety via borrow-checking

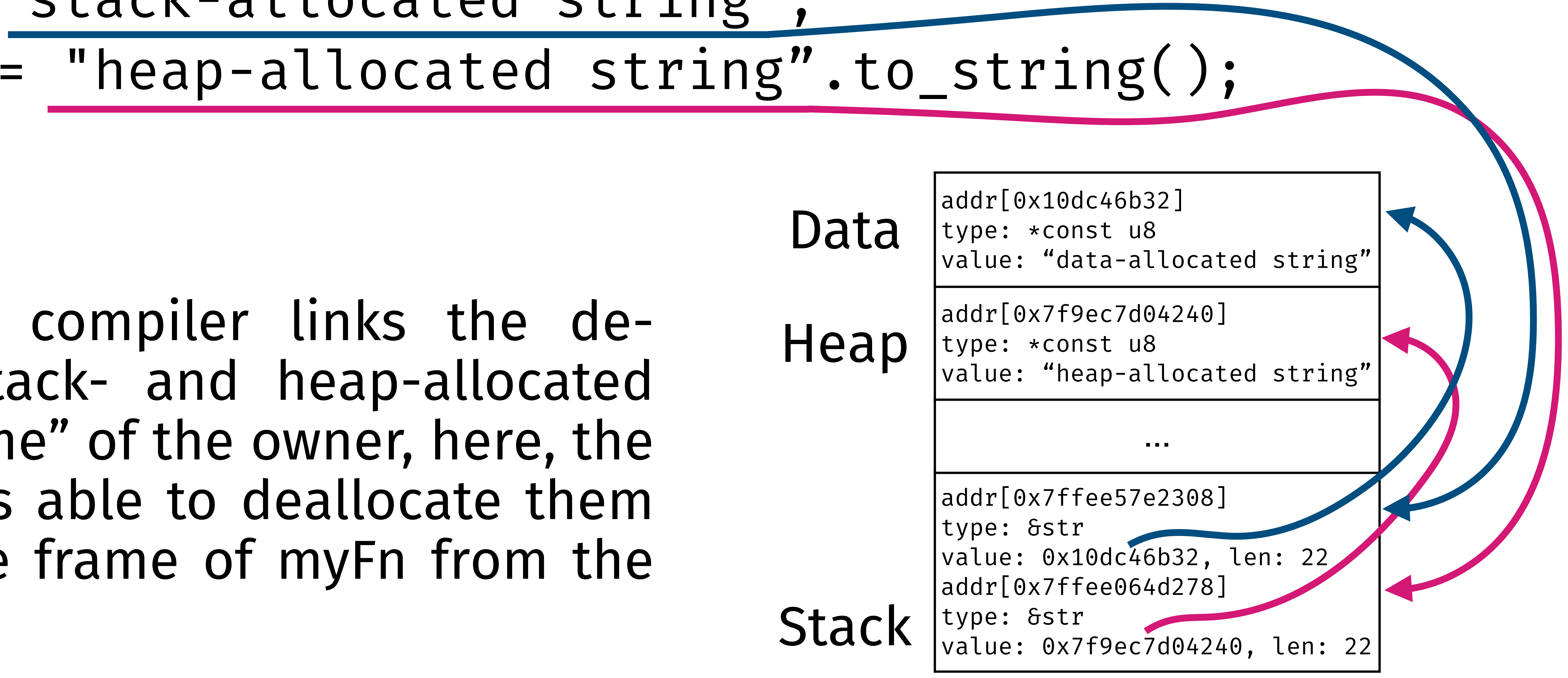
Borrow-checking is a technique (as present in, e.g., Rust) that tries to strike a balance between the safety of garbage-collected languages (Java) and the control provided by languages with memory-management constructs (C).

Borrow-checking works by restricting how programs can use pointers, so that, if the compiler allows the compilation of a program, then we know it is free from errors such as dangling and wild pointers, double frees, and similar memory-safety errors.

The technique hinges on the definition, in the language, of **ownership**, such that every value in a program has a **single owner** (its variable) that determines its **lifetime**. We already saw a similar concept with stack-allocated values and scopes, where, when the owner goes out of scope (and freed) also its owned values do.

Brief reminder of memory allocation (with Rust)

```
fn myFn() {
 let s1: &str = "stack-allocated string";
 let s2: String = "heap-allocated string".to_string();
}
```



```
addr[0x10dc46b32]
type: *const u8
value: "data-allocated string"

addr[0x7f9ec7d04240]
type: *const u8
value: "heap-allocated string"

...

addr[0x7ffee57e2308]
type: &str
value: 0x10dc46b32, len: 22
addr[0x7ffee064d278]
type: &str
value: 0x7f9ec7d04240, len: 22
```

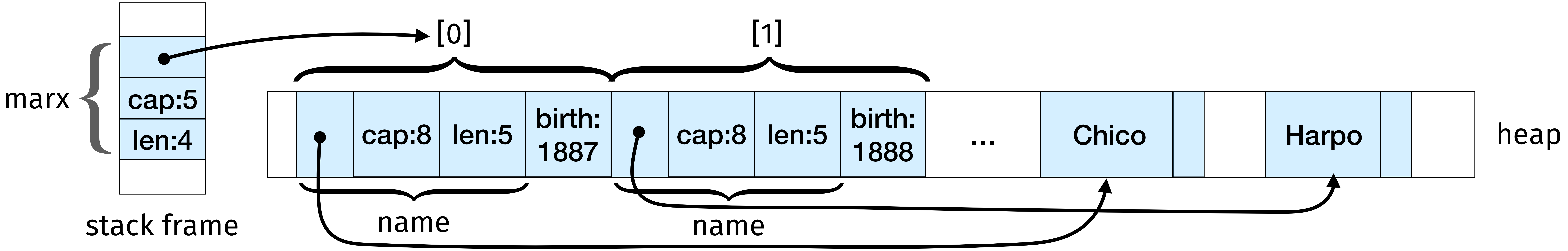
In both cases, the compiler links the de-allocation of the stack- and heap-allocated objects to the "lifetime" of the owner, here, the method myFn, and is able to deallocate them when we remove the frame of myFn from the stack.

Chains of ownership

As seen, we can nest data structures within each other, forming chains of ownership.

```
struct Person { name: String, birth: i32 }
let mut marx = Vec::new();
marx.push(Person { name: “Chico”.to_string()  , birth: 1887 });
marx.push(Person { name: “Harpo”.to_string()  , birth: 1888 });
marx.push(Person { name: “Groucho".to_string(), birth: 1890 });
marx.push(Person { name: “Gummo”.to_string()  , birth: 1893 });
marx.push(Person { name: “Zeppo”.to_string()  , birth: 1901 });
```

Also in this case, when marx goes out of scope, the ownership checker knows that it is safe to remove it from the stack, with all the objects in the heap it owns.



Extending Ownership

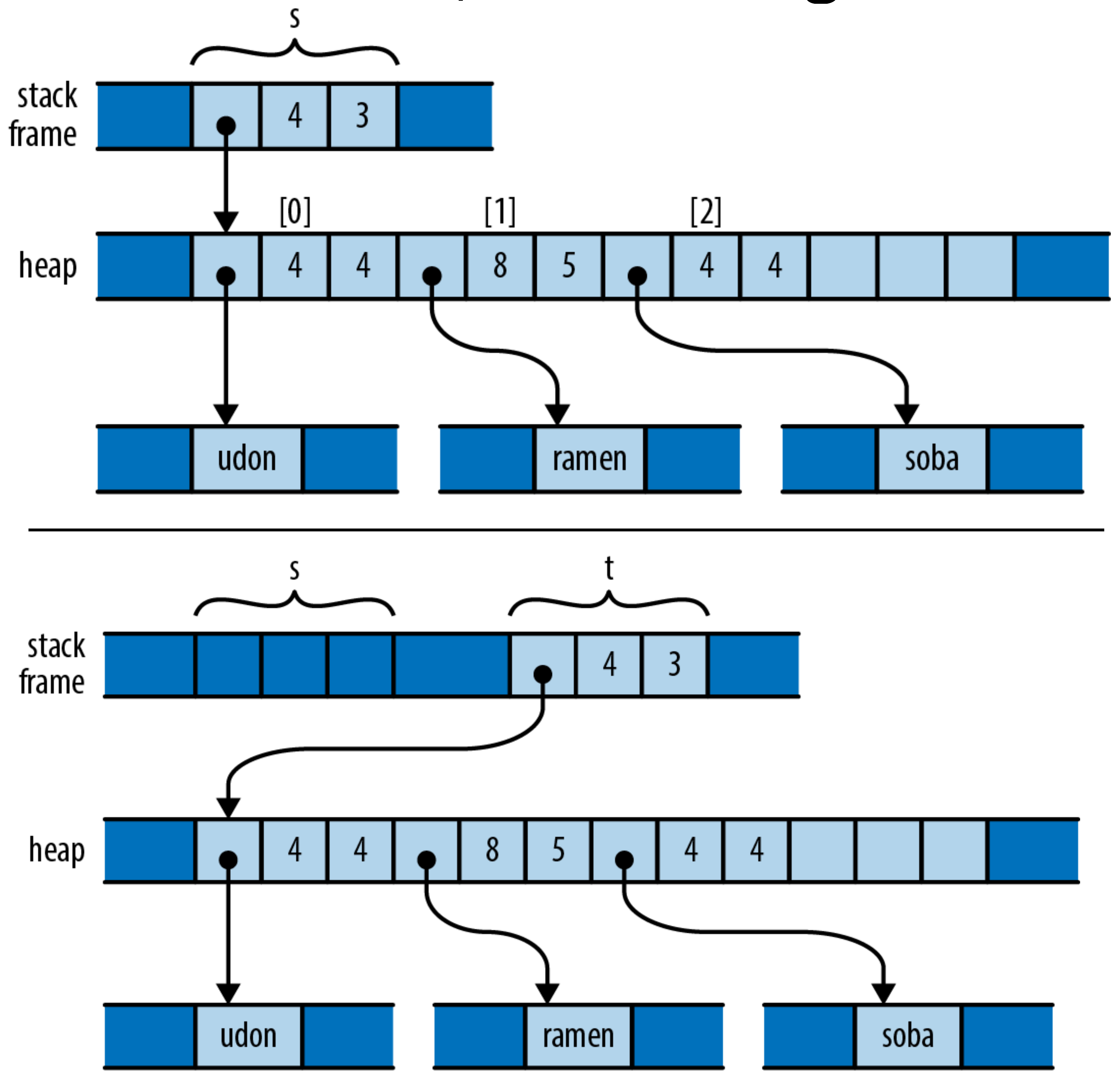
The concept of ownership we have seen so far is quite simple: it is essentially a tree where, by deallocating the root, we know we can deallocate all its subnodes. While this prevents us from building circular graphs of ownership, we can safely extend this idea in several directions:

- We can **move** the ownership of values from one owner to another;
- We can give more freedom from the ownership rules to some simple types, like integers, floating-point numbers, and characters;
- We can provide **special reference-counted pointer types** (Rc and Arc), which allow values to have multiple owners, under some restrictions.
- We can let owners “**borrow a reference**” to a value, as long as we restrict references to non-owning pointers with limited lifetimes.

Moves

In Rust, for most types (besides some exceptions, discussed afterwards), operations like **assigning a value** to a variable, **passing a value** to a function, or **returning a value** do not perform any copy but rather **move the ownership** of the value.

```
let s = vec![ “udon”, “ramen”, “soba” ];
let t = s;
let u = s;
```



For example, the simple program above would not compile in Rust. The reason is that first we assign to s the vector, then, we pass it to t, so, now s is an **uninitialised variable** (it owns no value) and the second assignment would not make sense.

Further Moves

```
let x = vec![10, 20, 30];
if false {
 let a = x;
} else {
 let c = 2;
}
let c = x;
```

Of course, **moves interact with control-flow constructs**.

For example, the code on the left would not compile, since (although it is impossible to enter it) we have a conditional branch (if) that moves the value owned by x to a, so that the last instruction would assign an uninitialised variable.

The general principle is that, if it is **possible** for a **variable** to have had its **value moved** away and it is **not certain** that it has been **given a new value** since, we consider the variable uninitialized.

Further Moves

```
let v = vec![“a”.to_string()];
let first = v[0];
```

An interesting case is that of **indexed collections**, where we could assign (move) the content of some of the elements in, e.g., a vector, into a variable.

However, to allow this, we would **need to remember** that, now, the first **element of the vector** v **has become uninitialized**, and track that information until the vector is deallocated. In general, this is not a feasible path, unless we severely limit what kind of expressions we can use to access vectors.

Indeed, if we just allowed constants, we could keep track of what values become uninitialised, however, with general expressions, we cannot predict what elements of any given vector become uninitialised due to a move.

So we do not allow assigning (moving) elements out of collections, but rather let the programmer **reference** or or **copy** its values.

Copy Types

While the move semantics makes it clean and cheap to pass values around, there are some cases where it is easier (for the programmer) **to pass values "by value"**, i.e., by making a copy of it.

This is the case for **Copy types**, i.e., values of types that are cheap (or useful) enough to make a copy of, rather than moving their ownership. This is what happens, in Rust, with numbers (integers and floats), which are just patterns of bits in memory, without any heap resources or dependencies on anything other than the bytes it comprises, so it is relatively **cheap** and, above all, **safe to create independent copies** of the values rather than moving them like any other type of variable. This is the same with chars and bool types, as well as tuples or fixed-size arrays of Copy types.

To make a counterexample, String is not a Copy type, because it owns a heap-allocated array. Besides standard Copy types, users can define their **struct Copy types**, as long as these **use only Copy types** in their fields.

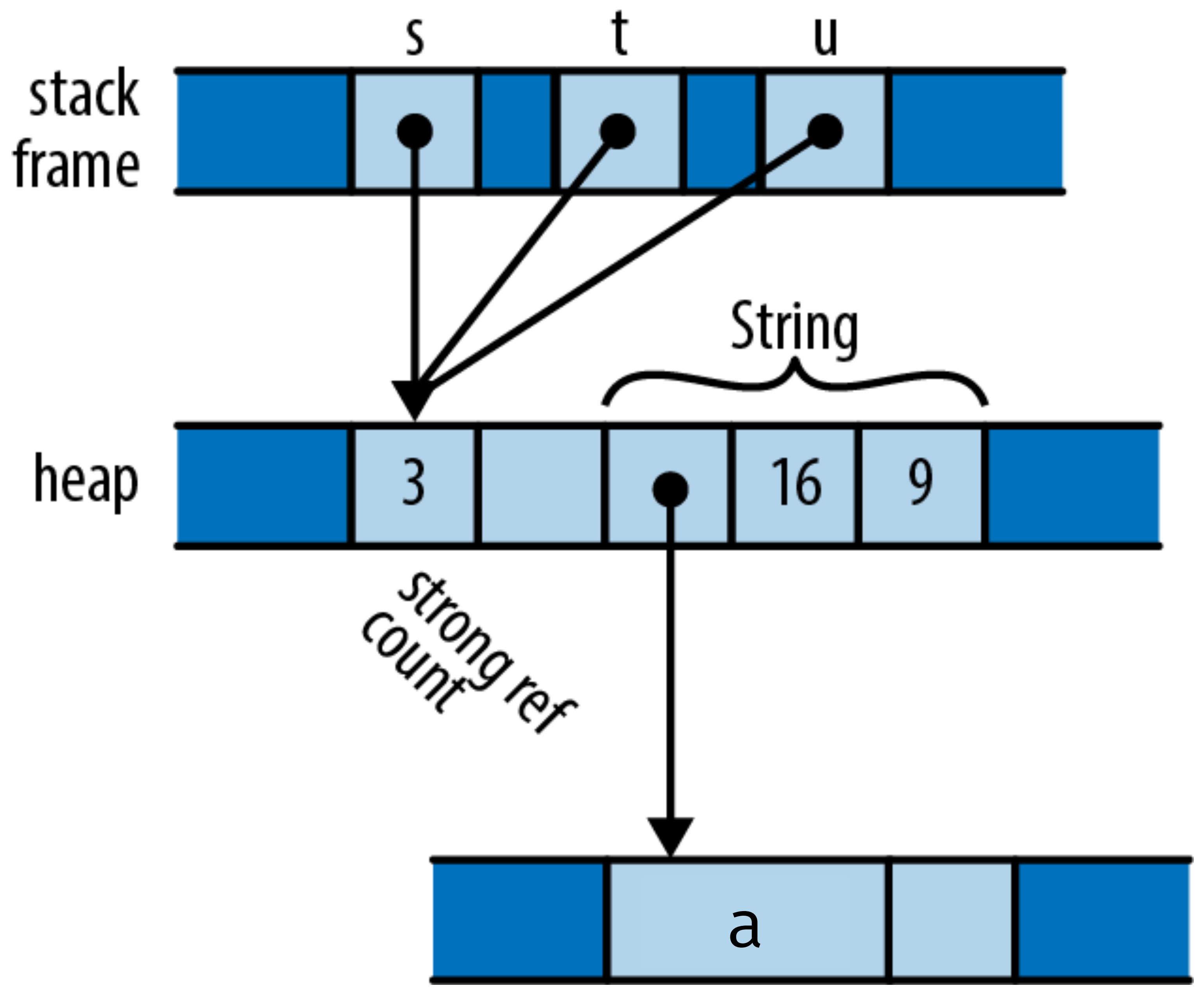
Rc and Arc: Shared Ownership

While for the general case it is enough for values to have unique owners, sometimes we need to share the ownership of values among a set of owners, e.g., who can safely read the value, because it is immutable.

For these cases, Rust provides the **reference-counted pointer types**, called **Reference Count (Rc<T>)** and **atomic reference count (Arc<T>)**.

Intuitively, Rc/Arc types are quite similar (the second adds some checks to make reference-counting thread safe) and they keep track of the number of references to a value to determine whether or not the value is still in use. If there are zero references to a value, the value can be cleaned up without any references becoming invalid.

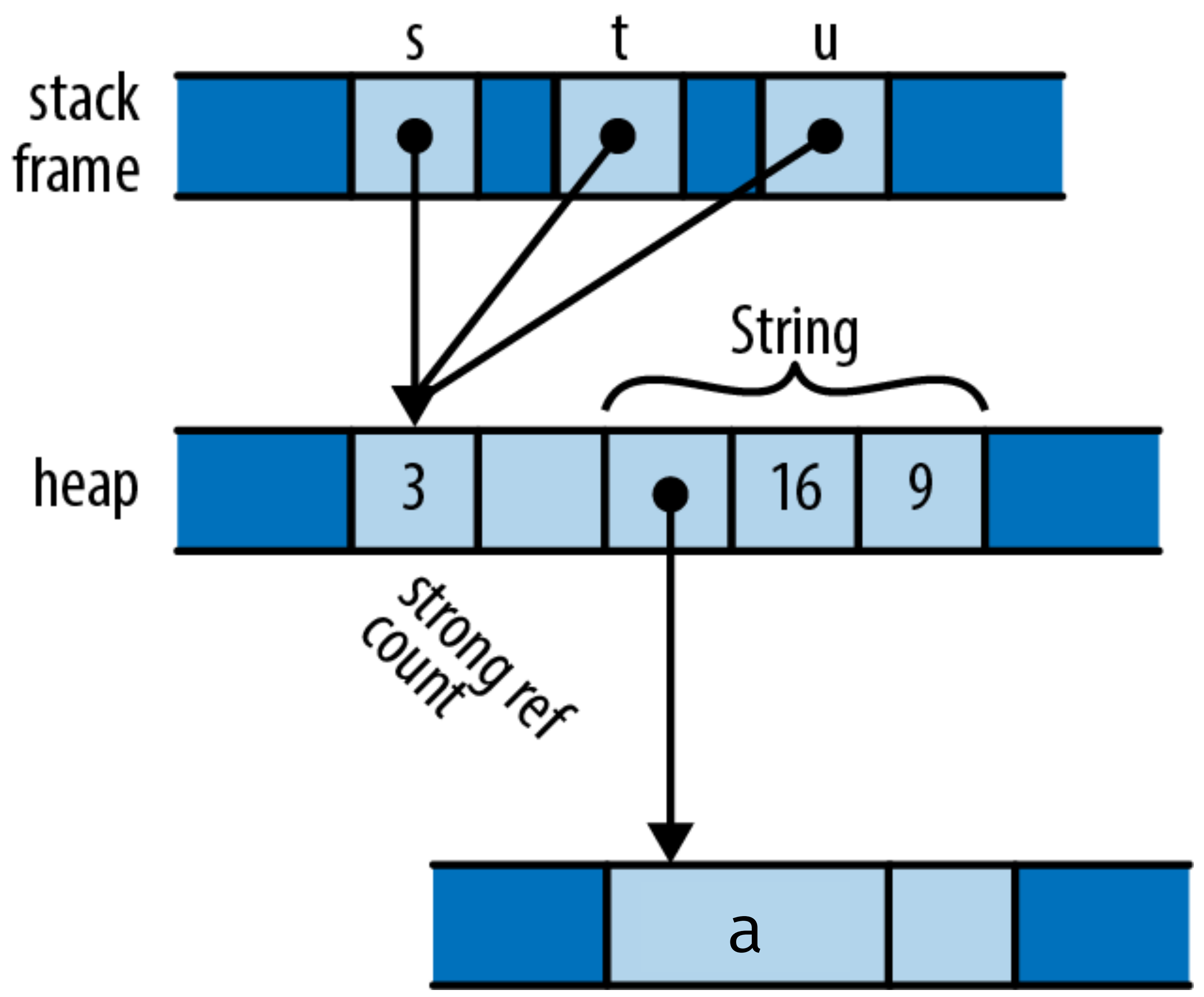
```
let s: Rc<String> = Rc::new("a".to_string());
let t: Rc<String> = s.clone();
let u: Rc<String> = s.clone();
```



Rc and Arc: Shared Ownership

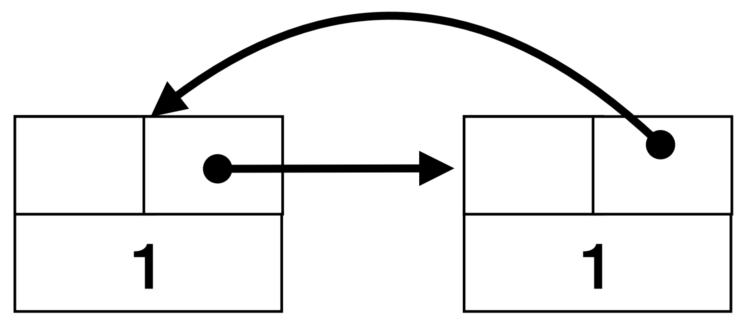
In the example, each of the three `Rc<String>` pointers refers to the same block of memory, which holds a reference count and space for the String.

The usual ownership rules apply to the Rc pointers themselves: each clone increment the counter while each deallocation of a pointer decreases, until the last extant is deallocated and the runtime also deallocates the referenced String in the heap.



```
let s: Rc<String> = Rc::new("a".to_string());
let t: Rc<String> = s.clone();
let u: Rc<String> = s.clone();
```

Rc and Arc: Shared Ownership



As we discussed for garbage collectors, one well-known problem with reference counting is that we cannot deallocate graphs of reference-counted values.

Also reference-count types can suffer from this problem, and Rust makes such situations explicit and rare.

Indeed, we cannot create a cycle without, at some point, making an older value point to a newer value, which implies that at least the the older value is **mutable**. This is the reason why Rc pointers hold their referents **immutable**. Of course, there are some cases that require us to create cyclical structures in memory. Rust accommodates this need with Reference Cells, which provide an interface to allow referenced values to change to the cost of renouncing static guarantees on the usage of pointers for runtime checks (which might make the program fail, if the reference is used improperly).

Borrowing References

The pointer types seen until now are owning pointers: when the owner is dropped, the referent goes with it. Rust also has non-owning pointers, called **references**, which have no effect on their referents' lifetimes but rather depend on it: references must never outlive their referents. This is why, in Rust-speak, creating a reference to some value is called **borrowing the value**, with the motto "what is borrowed must eventually return".

A reference gives access to a value without affecting its ownership and can be of two kinds:

- **shared references** let the user read but not modify its referent. Since it is a read-only reference, we can safely have as many shared references to a value as we like. This is the simplest form of referencing, as uses the same syntax seen, e.g., in C, `&e` generates a shared reference to the value held in variable `e`, so if `e` has the type `T`, then `&e` has the type `&T`, and `&T` is always a subtype of `Copy`.
- **mutable references** let the user both read and modify the value, but it makes it impossible to have any other references to that value active at the same time. The syntax to generate a mutable reference is `&mut e`, which generates a mutable reference to the value held in variable `e`. Types-wise we have `&mut T`, which are not subtypes of `Copy`.

Reference are never `NULL` in Rust. The absence of the `NULL` value for pointers comes from the checks performed by the compiler, which makes sure that variables are never used until they are initialised (there are no wild pointers) and the fact that there is no (safe) way to have the language generate custom references (e.g., from a given integer).

Lifetimes

The Rust compiler validates borrows by reasoning on the lifetimes of variables; essentially making sure that **no reference outlives its referent**.

```
let x: &str;
{
 let y = "a";
 x = &y;
}
println!( "{}", x );
```

```
  |
6|  x = &y;
  |       ^^ borrowed value does not live long enough
7|  }
  |  - `y` dropped here while still borrowed
8|  println!( "{}", x )
  |                  - borrow later used here
```

In the example above, the compiler complains because x would access invalid memory, since the value referred by y is deallocated once out of the inner scope.

The **interactions between variables do not change lifetimes** but rather **impose constraints** for the compiler to check. In the example, x=&y sets the constraint that the lifetime of y should be as long as that of x, which triggers the error raised by the compiler. Actually, the check on lifetimes done by Rust is more refined than this coarse rule, and it makes the lifetime of x and y related as long as x is assigned to y and we need to read x (as in the example, where we try to print it). Indeed, if we remove the reading of x the program would compile (i.e., since x is never used, we can get rid of it after the assignment).

Lifetime Annotations

While the Rust compiler is quite smart in inferring lifetimes (like type inference), in some cases we need to provide information on lifetimes—through lifetime annotations—to clarify ambiguous situations.

```
fn snd(a: &str, b: &str) -> &str {
 return if true { a } else { b }
}
fn main() {
 let a = "a";
 let b = "b";
 snd( &a, &b );
}
```

For example, in the example on the left, the compiler needs to know whether the reference returned from the function borrows from a or b, so the compiler asks the programmer to explicitly add this constraint.

```
  |
1 | fn snd(s1: &str, s2: &str) -> &str {
  |            ----      ----     ^ expected named lifetime parameter
  |
```

Lifetime Annotations

```
struct S<'a,'b> {
 x: &'a i32,
 y: &'b i32,
 z: &'b i32
}
fn main() {
 let x = 10;
 let r;
 {
  let y = 20;
  {
   let s = S {
    x: &x,
    y: &y,
    z: &y
   };
   r = s.x;
  }
 }
 println!( "{}", r );
}
```

To solve this issue, we need to use **lifetime parameters**—in a way, similar to type parameters.

In Rust, these annotations go at the level of types and, as mentioned, Rust does already a good job at inferring lifetimes, along with types: when **we create a variable of a certain** (inferred) **type S it also has a fresh lifetime parameter 'l, which becomes constrained by how we use the referred value**.

Lifetime annotations can make lifetimes of related variables explicit, e.g., in the struct S in the example, we declare it to have two lifetime parameters, where one (‘a) identifies the lifetime of one field (x), while the other (‘b) identifies the lifetimes of two files (y and z).

This make our example compile, where the lifetime of x can outlive that of y and z, since they are distinct. Without this annotation, x, y, and z would have shared the same lifetime, resulting in a compilation error.

Lifetime Annotations

The concept is similar for functions, where each type of each variable and the return type have associated a different lifetime, i.e.,

```
fn snd(a: &str, b: &str) -> &str {
 return if true { a } else { b }
}
fn main() {
 let a = “a"; let b = "b"; snd( &a, &b );
}
```

```
fn snd<‘c,’d,’e>(a: &’c str, b: &’d str) -> &’e str {
 return if true { a } else { b }
}
fn main() {
 let a = “a"; let b = "b"; snd( &a, &b );
}
```

Lifetime Annotations

In our example, since the function snd decides at runtime which reference, between a and b, to return, we clarify that they must share the same lifetime.

```
fn snd<‘l>(a: &’l str, b: &’l str) -> &’l str {
 return if true { a } else { b }
}
fn main() {
 let a = “a"; let b = "b"; snd( &a, &b );
}
```

Concretely, the function just has one lifetime parameter, `‘l` that binds together (i.e., constraints to be equal) the lifetimes of a and b.

The meaning of the annotation of the return type with `‘l` is slightly different: this does not mean that a, b, and the return value of snd must all have the exact same lifetime but rather than **the returned reference is borrowed from the argument with the same annotation** (in this case, either a or b).

Lifetime Annotations

To further clarify, let us look at an extension of the example saw before.

Remind that the lifetime constraint on snd is that the returned reference must not outlive those of a and b.

```
fn snd<'l>(a: &’l str, b: &'l str) -> &'l str {
  return if true { a } else { b }
}
fn main() {
 let r: &str;
 let a = “a”.to_string();
 {
  let b = “b”.to_string();
  r = snd( &a, &b );
 }
 println!( "{}", r );
}
```

The example breaks this constraints, since the lifetime of r outlives that of b (the one of a is fine).

```
9 |r=snd(&a,&b);
  |         ^^ borrowed value does not live long enough
10|}
  |- `b` dropped here while still borrowed
11|println!( "{}", r );
  |- borrow later used here
```

Mutable References

As we have seen, values borrowed by **shared references** are read-only. On the contrary, a value borrowed by a **mutable reference** is reachable exclusively via that reference. Moreover, across the lifetime of a mutable reference there must be no other usable path to its referent or to any value reachable from there. The only references whose lifetimes may overlap with a mutable reference are those borrowed from the mutable reference itself.

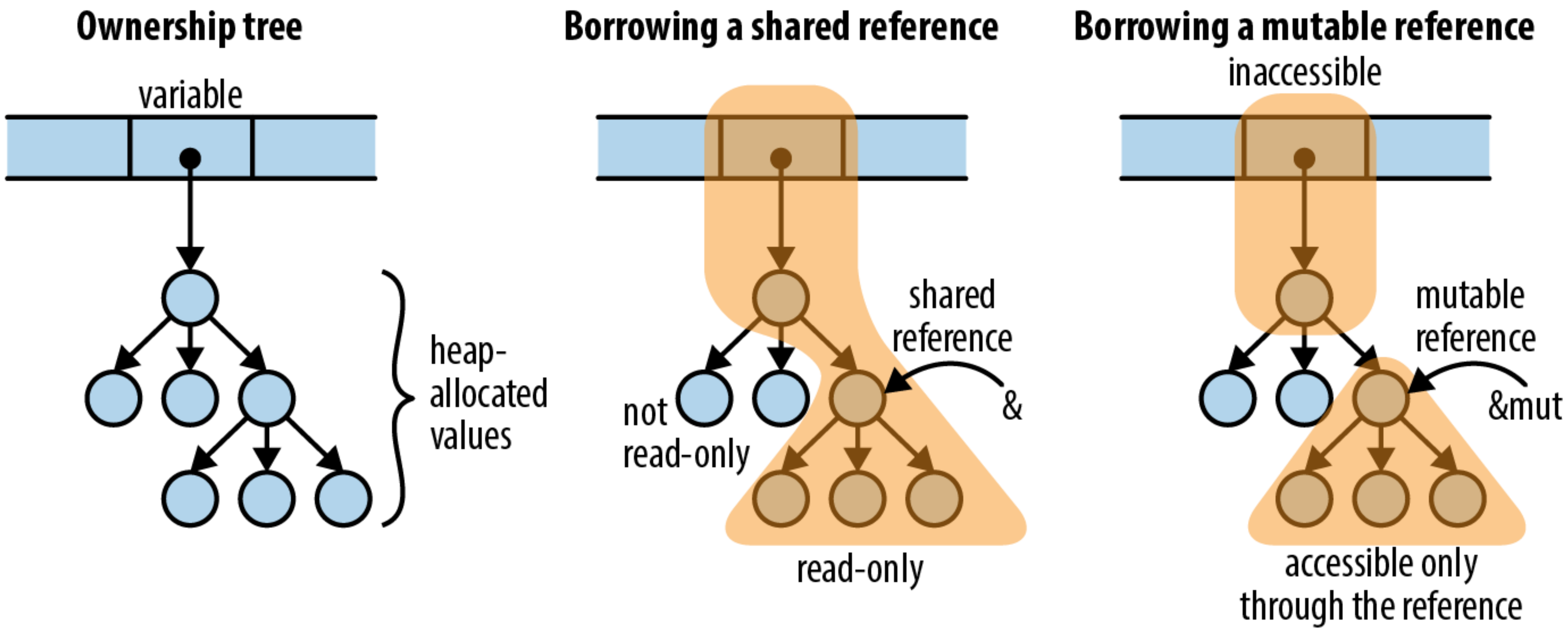
```
fn concat(l: &mut String, r: &String) {
 l.push_str( r );
}

fn main() {
 let mut greet = "Hello".to_string();
 let subject = "World".to_string();
 concat(&mut greet, &subject );
 concat(&mut greet, &greet );
}~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

In the example, we violate the rule for mutable references (last instruction): we borrow both a mutable and immutable reference to `greet`.

Of course, vice versa, we can see the violation from the side of the shared reference: since we borrow a shared reference to `greet`, it must be read-only, so the borrowing of a mutable reference is illegal.

Recap: Ownership and Shared and Mutable References


Ownership tree
variable
heap-allocated values
Borrowing a shared reference
shared reference
&
not read-only
read-only
Borrowing a mutable reference
inaccessible
mutable reference
&mut
accessible only through the reference