Sicurezza della memoria: Garbage Collection e Borrow-checking

Dangling Pointers/References

I riferimenti possono diventare **dangling** (o anche **wild**) quando fanno riferimento a una destinazione non valida. Questi includono puntatori a memoria deallocata o riallocata (ad esempio, dopo averla liberata). In tutti questi casi, il dereferenziamento di un puntatore dangling può portare a comportamenti inaspettati (**undefined behaviour**), poiché la posizione di memoria referenziata contiene dati inattesi.

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

Di solito, i puntatori vengono chiamati **wild**, quando sono **non inizializzati**. Questa categoria è più facile da individuare, osservando gli accessi a variabili non inizializzate.

Al contrario, l'individuazione dei riferimenti dangling è più difficile e richiede di monitorare e ragionare su tutte le possibili combinazioni di codice in cui passano i valori dei puntatori (a volte fare tale analisi è impossibile, ad esempio nelle librerie compilate).

Tombstones

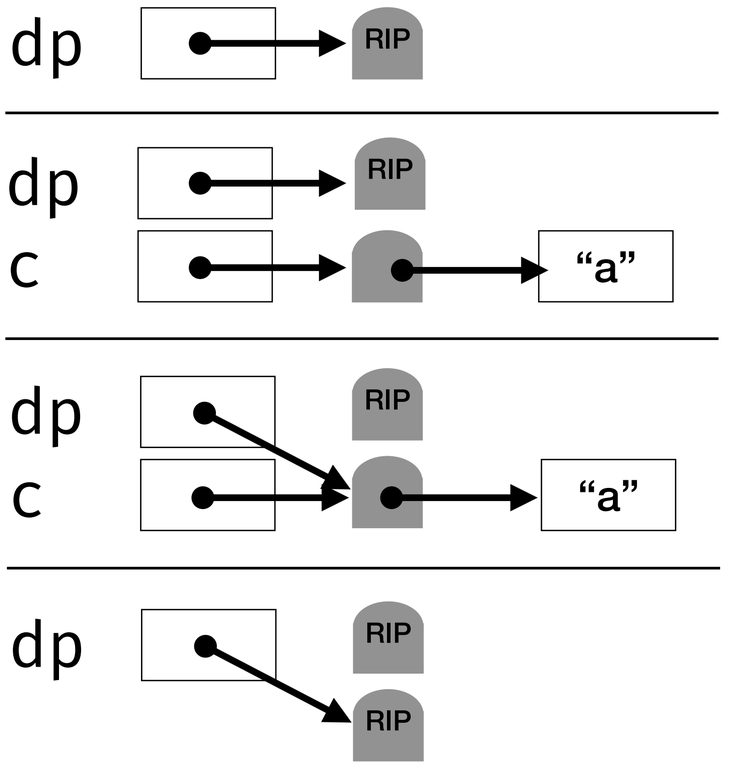
Un modo per gestire i riferimenti dangling sono le **tombstones** (lapidi).

Essenzialmente, le tombstones funzionano associando **a ogni allocazione** a cui accede un puntatore una parola aggiuntiva allocata in memoria, chiamata tombstone.

All'inizializzazione, la tombstone contiene l'indirizzo dell'elemento allocato, mentre il puntatore stesso riceve l'indirizzo della tombstone.

Tutte le dereferenziazioni dei puntatori diventano **accessi a due hop**, in cui si accede prima alla tombstone e poi al puntatore dell'oggetto.

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



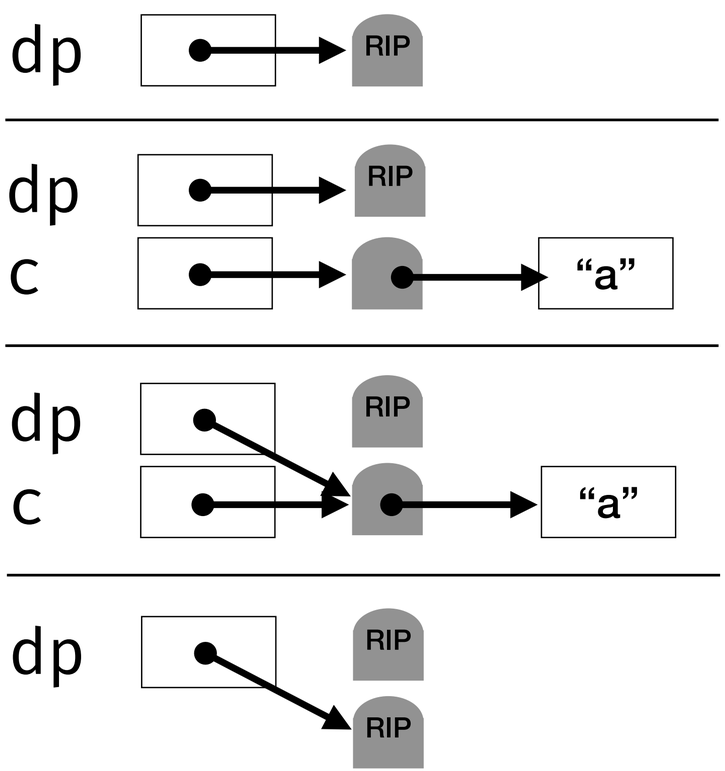
Tombstones

L’accesso a due hop **tiene sotto controllo tutte le possibili duplicazioni del puntatore, che puntano tutte alla stessa tombstone**.

Finché la posizione di memoria dell'oggetto puntato è allocata, la sua tombstone vi punta.

Quando si dealloca l'oggetto, lo si segna come "morto" con una parola dedicata nella sua tombstone/lapide (da cui il nome) e si solleva un errore in caso di dereferenziazione successiva.

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



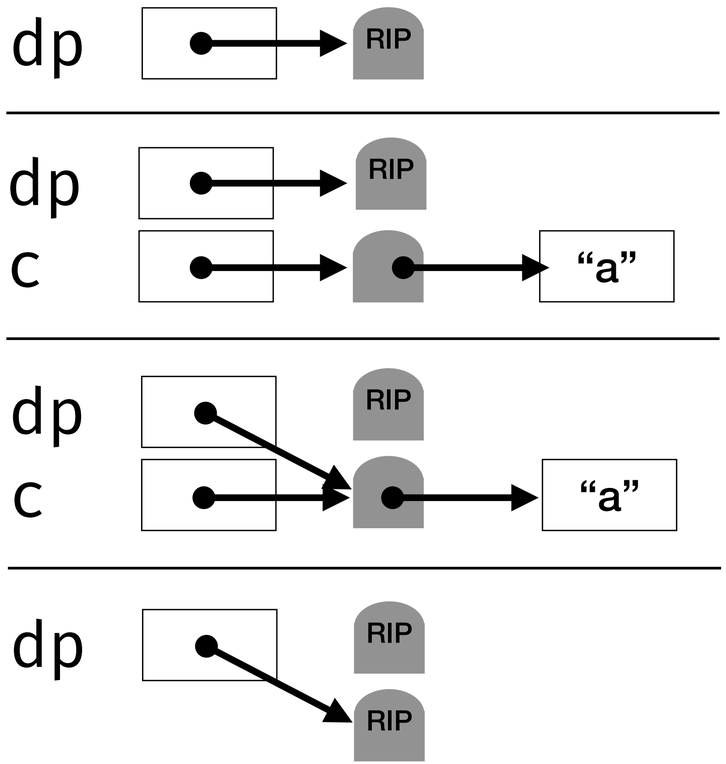
Tombstones

Anche se semplice, il meccanismo delle tombstones ha un conto salato da pagare.

Dal punto di vista dell'efficienza, stiamo **duplicando tutte le dereferenziazioni** dei puntatori (l'accesso a due hop) e spenderemo un po' di tempo anche per creare la tombstone.

Dal punto di vista dello spazio, per ogni puntatore nel nostro programma abbiamo una **posizione di memoria occupata dalla sua tombstone** (sia per l'heap che per lo stack), che **non possiamo recuperare** (a meno che non impieghiamo alcune tecniche intelligenti di conteggio dei riferimenti), con la possibilità di esaurire lo spazio nel "cimitero".

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



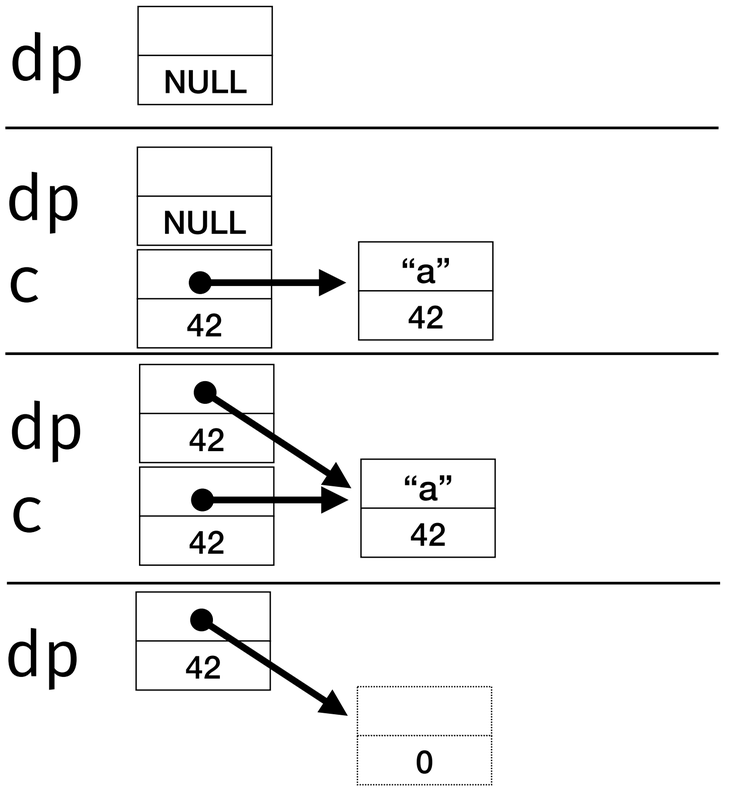
Locks and Keys

I lock-and-keys sono un'alternativa alle tombstones, ma solo per i puntatori allo heap. Ogni volta che **allochiamo un oggetto sull'heap, lo associamo a un "lock"** costituito da una parola casuale in memoria. Un puntatore consiste in una coppia: l'indirizzo effettivo e una “chiave" (key), cioè una parola in memoria inizializzata al valore del blocco dell'oggetto puntato.

Quindi, ogni volta che si **dereferenzia un puntatore, si controlla che** la chiave possa “aprire” il lock, cioè **che le due parole coincidano**. Questo significa anche che, quando assegniamo il puntatore, l'assegnazione copia sia il puntatore che la chiave, utilizzata per verificare la corrispondenza.

Al momento della deallocazione, cancelliamo l'oggetto in memoria e impostiamo il suo blocco su un valore canonico (al di fuori del dominio delle chiavi) e invalidiamo ogni possibile dereferenziazione successiva, che, come nelle tombstones, solleverebbe un errore.

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



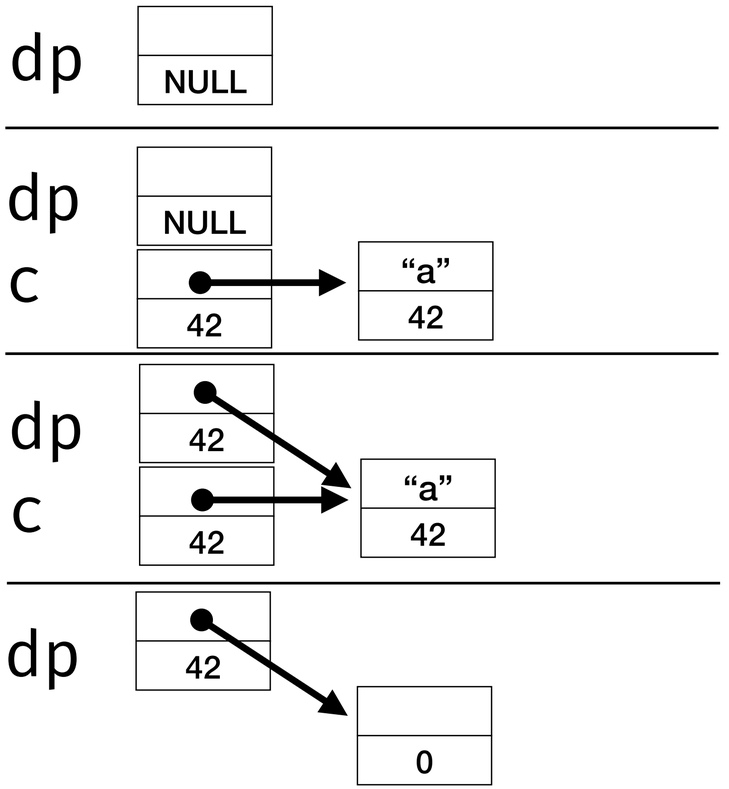
Locks and Keys

Può accadere che allocazioni successive possano utilizzare l'area di memoria precedentemente usata come lock (per un altro lock o per altri scopi), ma è statisticamente improbabile che un errore non venga rilevato perché la cella di memoria attuale contiene esattamente il valore del lock precedentemente presente.

Locks-and-keys ha un costo significativo. In termini di spazio, costano anche **più delle tombstones**, poiché **è necessaria una parola aggiuntiva per ogni puntatore**, anche se sia i lock che le chiavi vengono deallocati insieme all'oggetto o al puntatore di cui fanno parte.

Dal punto di vista dell'efficienza, locks-and-keys determinano **costi alla creazione**, **all'assegnazione** e alla **dereferenziazione**, poiché ora si esegue sempre un controllo di equivalenza

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



Garbage Collection

Se non vogliamo gestire manualmente la deallocazione della memoria nello heap, abbiamo bisogno di un meccanismo per gestire automaticamente la deallocazione degli oggetti non più utilizzati.

Il termine generale per uno di questi meccanismi è "garbage collection”—introdotto per la prima volta in LISP (intorno al 1960) ma presente in molti linguaggi moderni come Javascript, Python e Java—e chiamiamo "garbage collector" un'implementazione di tale meccanismo.

Da un punto di vista logico, le operazioni di un garbage collector comprendono:

- **Garbage detection**: rilevare se gli oggetti in memoria sono in uso o meno;
- **Garbage collection**: rilasciare la memoria occupata dagli oggetti non utilizzati.

Seguendo la nomenclatura, chiamiamo "garbage" gli oggetti non in uso che il garbage collector può rilevare e raccogliere.

Garbage Collection

Poiché la garbage collection è uno dei principali fattori determinanti per le prestazioni dei linguaggi con gestione automatica della memoria (ad esempio, il runtime di Java fornisce 7 diversi algoritmi di garbage collection, adattati a diversi scenari di esecuzione), le tecniche moderne sono diventate piuttosto performanti e sofisticate. Qui vedremo quelle *più comuni*.

In generale, il funzionamento della garbage collection *dipende dal funzionamento della detection*—e la detection e la collection vengono spesso eseguite in stretta successione temporale.

Inoltre, diventa **più facile lavorare sugli oggetti in memoria se il collector conosce la loro forma/limiti** e, allo stesso modo, **quali posizioni di un oggetto corrispondono a puntatori**—poiché il garbage collector deve seguire questi puntatori per rilevare se la deallocazione di un oggetto rende garbage altri oggetti puntati da quest'ultimo.

Possiamo fornire queste informazioni sia staticamente che dinamicamente (con i soliti compromessi tra espressività e prestazioni) lasciando che il compilatore/runtime associ ogni oggetto a un descrittore del suo tipo, riportando, ad esempio, la dimensione e gli offset della posizione dei puntatori. In seguito, a runtime, il garbage collector può utilizzare questa associazione per semplificare l'attraversamento degli oggetti in memoria.

Garbage Collection • Reference Count

Il modo più semplice per identificare il garbage è trovare gli oggetti che non hanno puntatori ad essi. La tecnica dei **contatori di riferimenti** segue questa definizione ed è probabilmente il modo più elementare per realizzare un garbage collector.

Quando si alloca un oggetto nello heap, il runtime inizializza anche un contatore di riferimenti (un intero, inaccessibile al programmatore) di quell'oggetto e si assicura di mantenere il contatore di ogni oggetto sincronizzato con il numero di puntatori attivi a quell'oggetto.

Pertanto, al momento della creazione dell'oggetto, il suo contatore ha valore 1. Il runtime **aumenta il contatore ogni volta che assegniamo il suo puntatore a una variabile puntatore** e lo **diminuisce ogni volta che perde una variabile puntatore**, ad esempio perché la variabile puntatore è uscita dallo scope o perché l'abbiamo riassegnata a un altro puntatore.

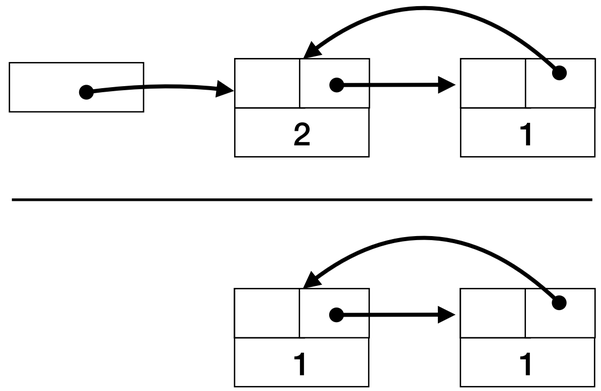
Garbage Collection • Reference Count

Quando un contatore di riferimento raggiunge lo 0, il suo oggetto diventa garbage e possiamo deallocarlo dalla memoria.

Come già detto, questa azione potrebbe non riguardare solo l'oggetto in questione, ma anche altri oggetti (puntati dall'oggetto), i cui contatori devono essere diminuiti.

Pertanto, quando si contrassegna un oggetto come garbage (cioè si imposta il suo puntatore a 0), occorre prima eseguire una visita dell'oggetto e decrementare tutti i puntatori a cui fa riferimento (ed eventualmente contrassegnare gli oggetti puntati come garbage, se i loro contatori scendono a 0).

Poiché si tratta di una visita ricorsiva della struttura di memoria a partire dall'oggetto, una delle principali limitazioni di questa tecnica è **l'impossibilità di gestire strutture ricorsive**; non si tratta di un problema dell'algoritmo, ma piuttosto di una limitazione della definizione di garbage del conteggio dei riferimenti.

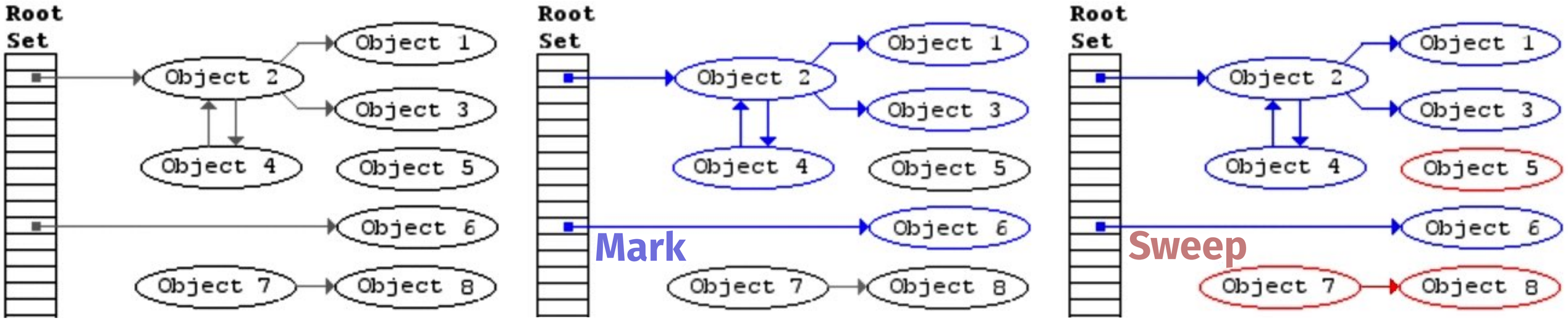


Garbage Collection • Mark and Sweep

La tecnica mark-and-sweep prende il nome dal modo in cui esegue detection e collection.

Mark indica che la detection utilizza contrassegni per identificare il garbage, seguendo due fasi. Prima, attraversa l'heap e contrassegna tutti gli oggetti come garbage. Seconda, attraversa lo stack, segue i puntatori attivi agli oggetti nello heap e rimuove i contrassegni precedenti da quelli visitati (ricorsivamente);

Sweep è una visita piatta dell'heap per raccogliere tutti gli oggetti marcati come garbage.



Garbage Collection • Mark and Sweep

A differenza dei garbage collector con reference counting, mark-and-sweep non solo non è incrementale ma **non libera la memoria nel momento in cui diventa garbage**.

Al contrario, è il runtime che lo invoca quando lo ritiene utile, ad esempio quando l'heap sta esaurendo la memoria disponibile.

La garbage collection con mark-and-sweep è un esempio di **stop-the-world garbage collection**, in cui il garbage collector deve arrestare completamente l'esecuzione del programma per assicurarsi di vedere tutti gli oggetti, cioè che nessun nuovo oggetto venga allocato e che nessun oggetto esistente diventi irraggiungibile mentre il garbage collector è in esecuzione.

Le tecniche mark-and-sweep (e stop-the-world) sono **più efficienti** e **facili da implementare** rispetto a quelle incrementali—che rallentano i programmi e richiedono più memoria per le loro funzioni di bookkeeping. Tuttavia, hanno **due svantaggi** principali. In primo luogo, la pausa ha un **effetto negativo** sulle prestazioni e **sulla reattività del programma**, rendendo il garbage collection stop-the-world **inadatto a programmi altamente interattivi**. In secondo luogo, senza politiche di allocazione della memoria che contrastino la frammentazione della stessa, **le prestazioni** della fase di marcatura **dipendono dalle dimensioni dell'heap** (prima fase di marcatura) **e dello stack** (seconda fase di marcatura).

Inversione dei puntatori: contrassegnare gli oggetti con un utilizzo minimo della memoria

La fase di marcatura è naturalmente ricorsiva e l'implementazione ovvia richiede uno stack la cui profondità massima è proporzionale alla catena più lunga di oggetti nello heap. Tuttavia, se il runtime decide di invocare il garbage collector perché il programma ha esaurito la memoria allocata, potremmo non avere lo spazio necessario per contenere lo stack di esplorazione—ricordiamo che, di solito, lo stack e l'heap crescono dalle estremità opposte di un vettore lineare, quindi, esaurire l'heap significa esaurire anche lo stack.

Per ridurre al minimo la memoria richiesta da questo passaggio, possiamo **codificare lo stack nei campi già esistenti dello heap**, utilizzando una tecnica chiamata "**inversione dei puntatori**".

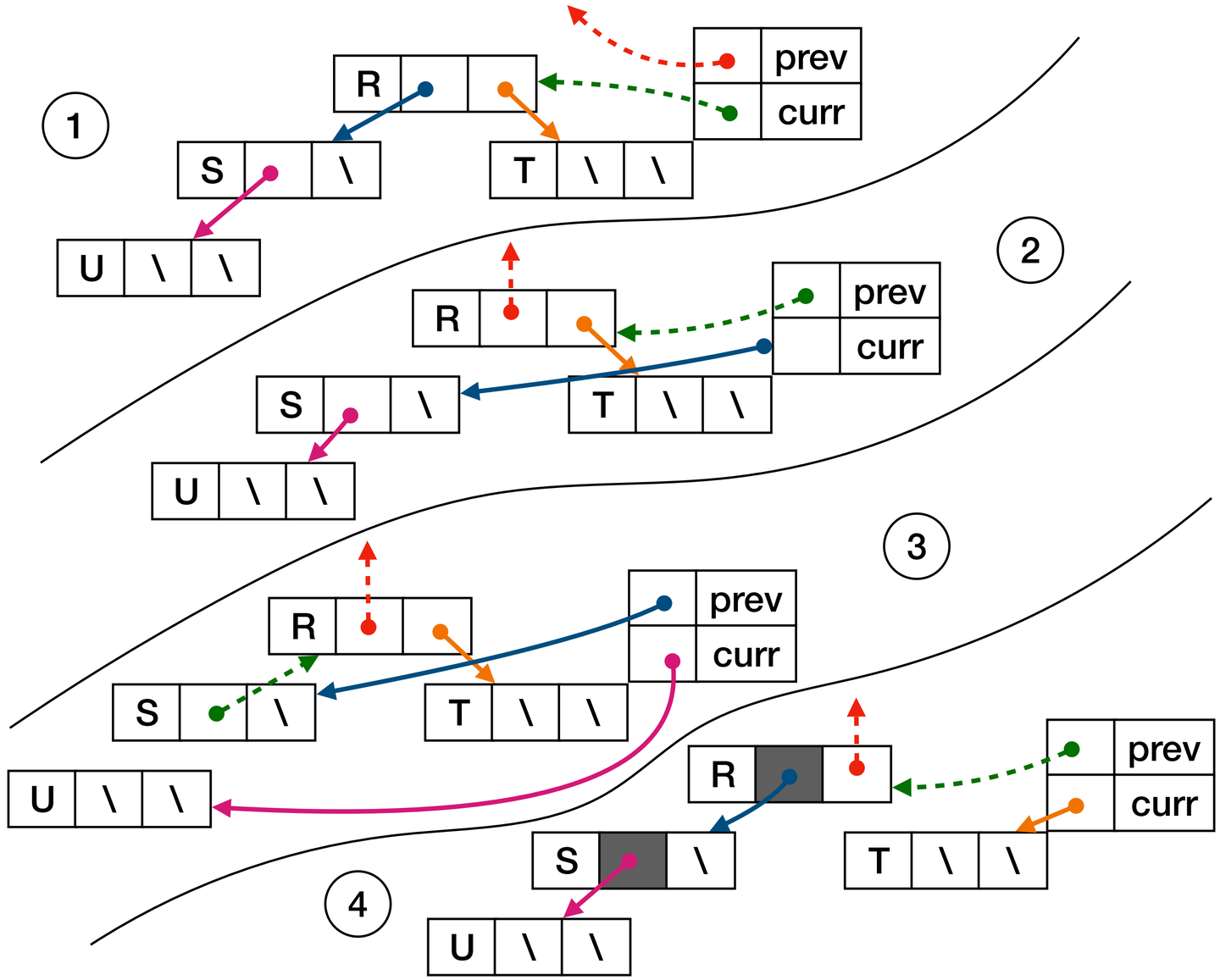
In particolare, poiché il garbage collector esplora il percorso di un dato oggetto, può invertire i puntatori che segue, in modo che ognuno di essi punti indietro al blocco precedente, invece di andare avanti al successivo.

Inversione dei puntatori: contrassegnare gli oggetti con un utilizzo minimo della memoria

La tecnica richiede solo due puntatori per funzionare: il puntatore **curr**, che indica l'oggetto attualmente in esame, e il puntatore **prev**, che indica l'oggetto che ha preceduto il corrente nella visita.

Quando il garbage collector passa da un oggetto all'altro, cambia il puntatore che segue per riferirsi all'oggetto precedente e, quando ritorna ad un oggetto visitato, ripristina il puntatore invertito al valore precedente (**curr**).

Per evitare loop nelle visite, il raccoglitore contrassegna i puntatori visitati/invertiti (ombreggiati, a destra), per distinguerli dai puntatori non visitati.

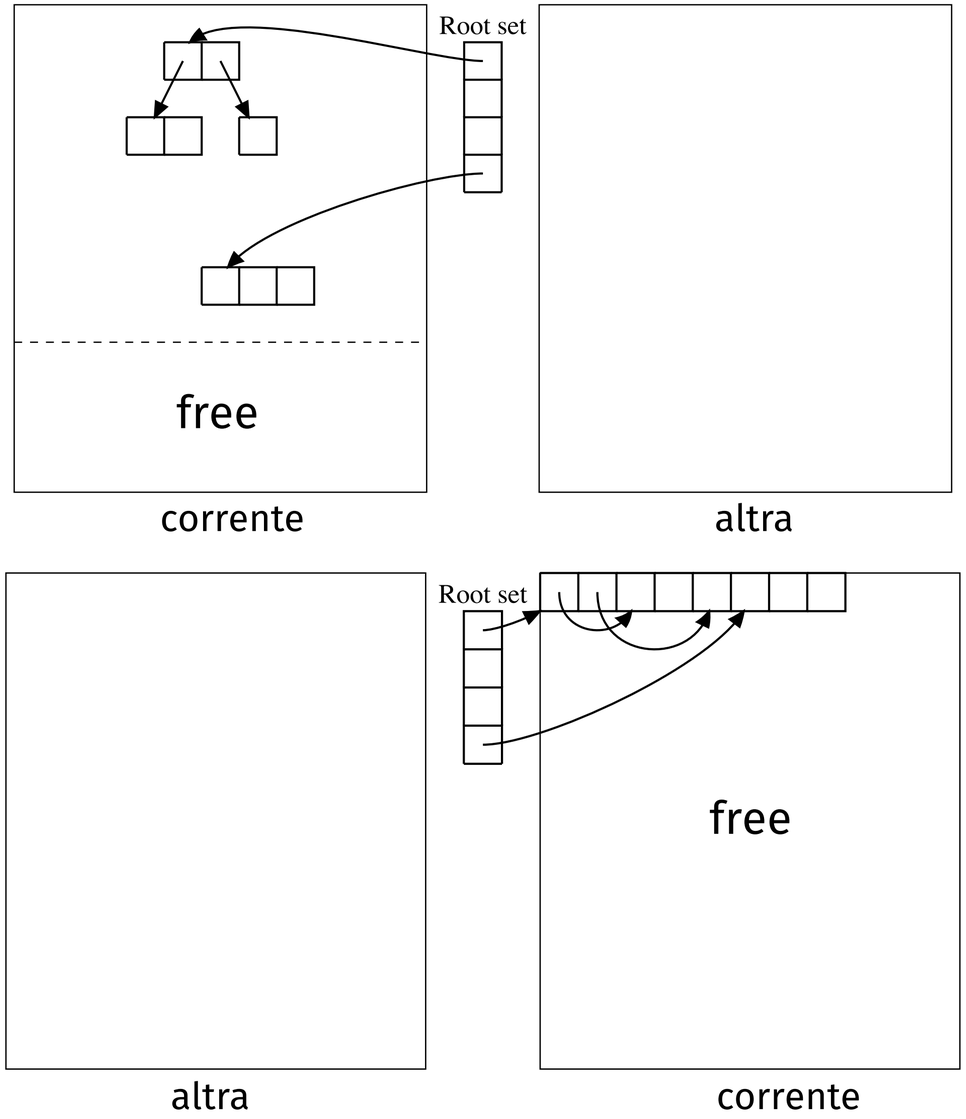


Stop and Copy

Un'evoluzione della tecnica mark-and-sweep è "stop and copy", che consente di ottenere la pulizia dello heap **eliminando sia la prima fase della marcatura che lo sweep**.

La tecnica funziona **dividendo l'heap in due regioni di uguale dimensione**. Tutte le allocazioni avvengono nella metà "corrente", mentre l'altra è vuota. Poi, quando la metà "corrente" è quasi piena, il garbage collector esplora la metà "corrente", a partire dall'insieme delle radici nello stack, e **copia ogni oggetto raggiungibile in modo contiguo nell'altra metà**. Quando il garbage collector termina l'esplorazione, scambia i puntatori dalla metà "corrente" e all'altra.

Sebbene questo metodo dimezzi la quantità di memoria heap disponibile, **il tempo richiesto da un raccoglitore stop-and-copy è proporzionale alla quantità di oggetti non garbage** nell'heap e **l'aumento della quantità di memoria disponibile diminuisce la frequenza di invocazione del garbage collector** e, quindi, il costo totale della gestione della memoria.



Memory Sicurezza della memoria attraverso il borrow-checking

Il borrow-checking è una tecnica (presente, ad esempio, in Rust) che cerca di trovare un equilibrio tra la sicurezza dei linguaggi con garbage collection (e.g., Java) e il controllo fornito dai linguaggi con gestione della memoria diretta (C).

Il borrow-checking funziona limitando il modo in cui i programmi possono utilizzare i puntatori, in modo che, se il compilatore consente la compilazione di un programma, sappiamo che è privo di errori come i puntatori dangling e wild, double free e errori simili di utilizzo della memoria.

La tecnica si basa sulla definizione, nel linguaggio, di una proprietà di possesso (**ownership**), dove ogni valore del programma ha un **unico proprietario** (una variabile) **che ne determina la durata** (**lifetime**) in memoria. Abbiamo già visto un concetto simile con i valori allocati nello stack e gli scope, dove, quando una variable esce dallo scope, liberiamo anche il valore relativo.

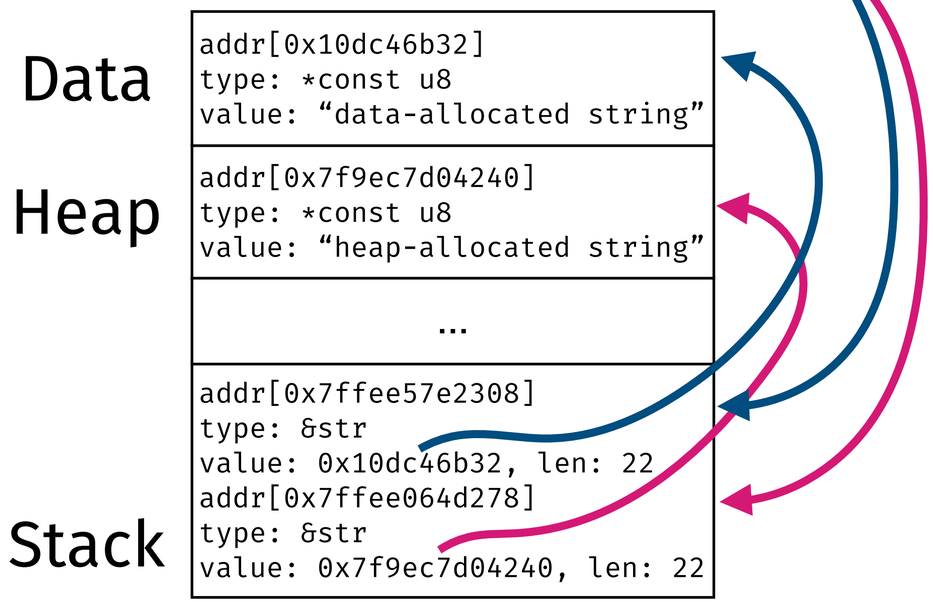
Breve promemoria sull'allocazione della memoria (con Rust)

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

In entrambi i casi, il compilatore collega la deallocazione degli oggetti nello stack e nello heap al "lifetime" del proprietario.

Nell'esempio sopra, `s1` è il proprietario della stringa "stack-allocated..." e `s2` è il proprietario della stringa "heap-allocated...".

Quando queste variabili terminano il proprio lifetime, qui, quando finiamo l'esecuzione dell'operazione myFn, possiamo eliminare entrambi i valori dalla memoria (sia nello stack che nello heap).

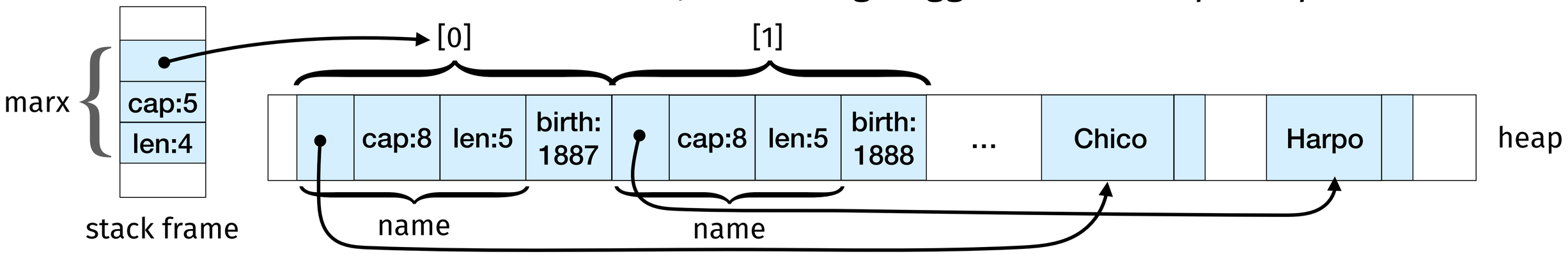


Catene di ownership

Come visto, possiamo annidare strutture di dati una dentro l'altra, formando catene

```
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 });
```

Anche in questo caso, quando `marx` esce dallo scope, il controllore di proprietà sa che è sicuro rimuoverlo dallo stack, con tutti gli oggetti dello heap che possiede.



Estendere la Ownership

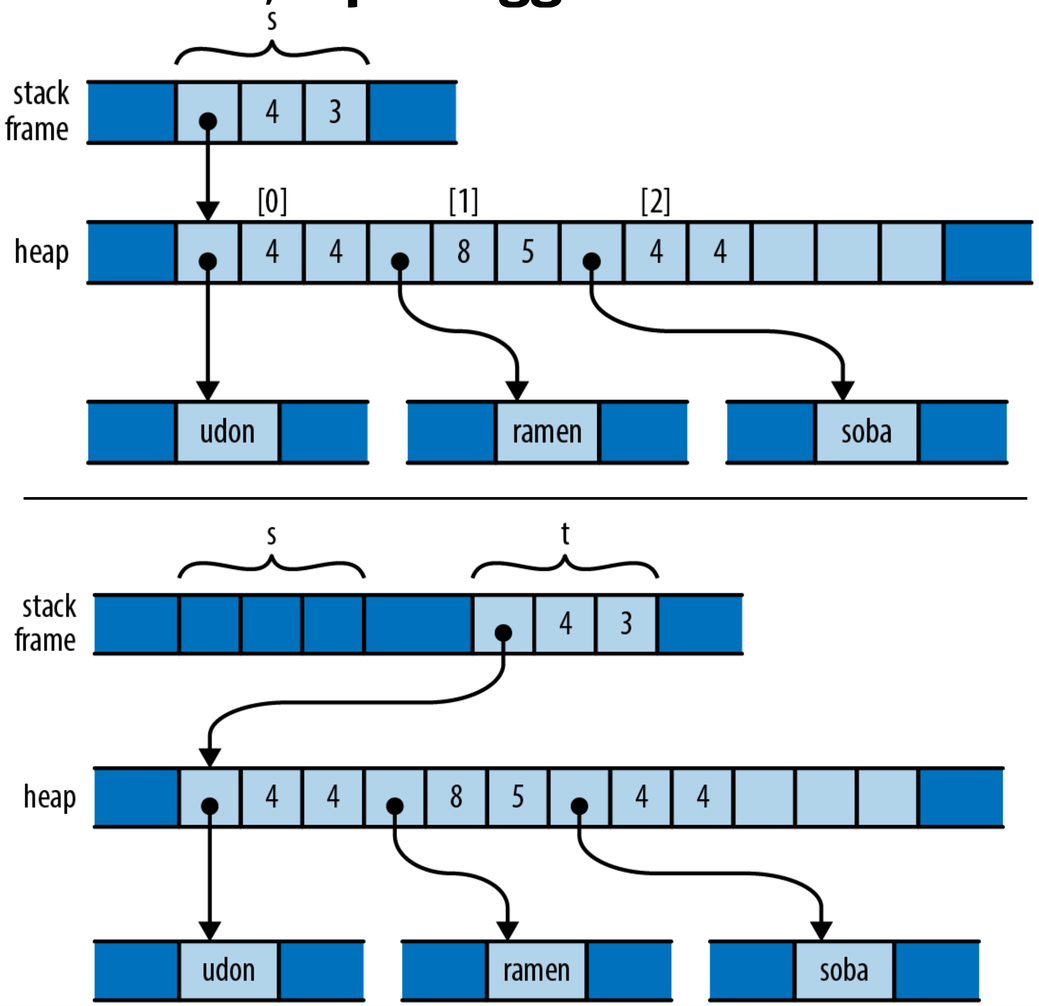
Il concetto di ownership che abbiamo visto finora è piuttosto semplice: si tratta essenzialmente di un albero in cui, deallocando la radice, sappiamo di poter deallocare tutti i suoi sottonodi. Sebbene questo ci impedisca di costruire grafi circolari, possiamo tranquillamente estendere questa idea in diverse direzioni:

- Possiamo **passare** (**move**) **la ownership** dei valori da un proprietario ad un altro;
- Possiamo dare maggiore libertà dalle regole di ownership ad alcuni tipi semplici, come gli interi, i floats e i caratteri;
- Possiamo fornire **tipi speciali di puntatore con conteggio dei riferimenti** (Rc e Arc), che consentono ai valori di avere più proprietari, con alcune restrizioni.
- Possiamo permettere ai proprietari di "**prendere in prestito un riferimento**" (**borrow**) a un valore, a patto di limitare i riferimenti a puntatori non proprietari con durata limitata.

Passare la proprietà (Move)

In Rust, per la maggior parte dei tipi (a parte alcune eccezioni, discusse in seguito), operazioni come l'**assegnazione di un valore** a una variabile, il **passaggio di un valore** a una funzione o la **restituzione di un valore** non eseguono alcuna copia, ma spostano la proprietà del valore.

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



Il semplice programma qui sopra non viene compilato in Rust. Il motivo è che prima assegniamo a `s` il vettore, poi lo **passiamo di proprietà** a `t` e ora `s` è una **variabile non inizializzata** (non possiede alcun valore), rendendo la seconda assegnazione scorretta.

Passare (ancora) la proprietà

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

Come ci si può aspettare, il **passaggio** (di proprietà) **dei valori interagisce con i costrutti del flusso di controllo**.

Ad esempio, il codice a sinistra non viene compilato poiché (sebbene sia impossibile accedervi) abbiamo un ramo condizionale (`if`) che sposta il valore posseduto da x in a, in modo che l'ultima istruzione assegnerebbe una variabile non inizializzata.

Il principio generale è che, **se è possibile che una variabile abbia spostato la proprietà del proprio valore** (e non è certo che da allora le sia stato assegnato un nuovo valore) **consideriamo la variabile non inizializzata**.

Passare (ancora) la proprietà

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

Un caso interessante è quello delle **collezioni indicizzate**, in cui si **potrebbe assegnare il** (spostare la proprietà del) **contenuto di alcuni elementi di un vettore**.

Tuttavia, per poterlo fare, dovremmo **segnarci**, nell’esempio, che il primo **elemento** del vettore v **è diventato non inizializzato** e tenere traccia di questa informazione finché il vettore non viene deallocato. In generale, questa non è una strada percorribile per un controllo statico, a meno che non si limiti fortemente il tipo di espressioni che si possono usare per accedere ai vettori.

In effetti, se fossero ammesse solo le costanti, potremmo tenere traccia dei valori non inizializzati; tuttavia, con le espressioni generiche, non possiamo prevedere quali elementi di un dato vettore diventino non inizializzati a causa di uno spostamento.

Per questo motivo, non permettiamo di assegnare (spostare) elementi dalle collezioni, ma lasciamo che il programmatore faccia **riferimento** o possa **copiare** i suoi valori.

Tipi Copia

Sebbene la semantica del movimento (di proprietà) renda efficiente il passaggio dei valori, ci sono alcuni casi in cui è più semplice (per il programmatore) **passare i valori "per valore"**, cioè facendone una copia.

Questo è il caso dei **tipi Copy**, cioè dei valori di tipi che sono abbastanza efficienti (o utili) da poterne fare una copia piuttosto che spostarne la proprietà. Questo è ciò che accade, in Rust, con i numeri (interi e float), che sono solo schemi di bit in memoria, senza alcuna risorsa nello heap o dipendenza da qualcosa di diverso dai byte che li compongono—quindi è relativamente efficiente e, soprattutto, sicuro creare copie indipendenti piuttosto che spostarli come qualsiasi altro tipo di variabile. Lo stesso vale per i tipi chars e bool, così come per le tuple o gli array di dimensioni fisse dei tipi Copy.

Per fare un controesempio, **String non è un tipo Copy**, perché possiede un array allocato in heap. Oltre ai tipi Copy standard, gli utenti possono definire i loro tipi Copy struct (à la C), purché questi utilizzino solo tipi Copy nei loro campi.

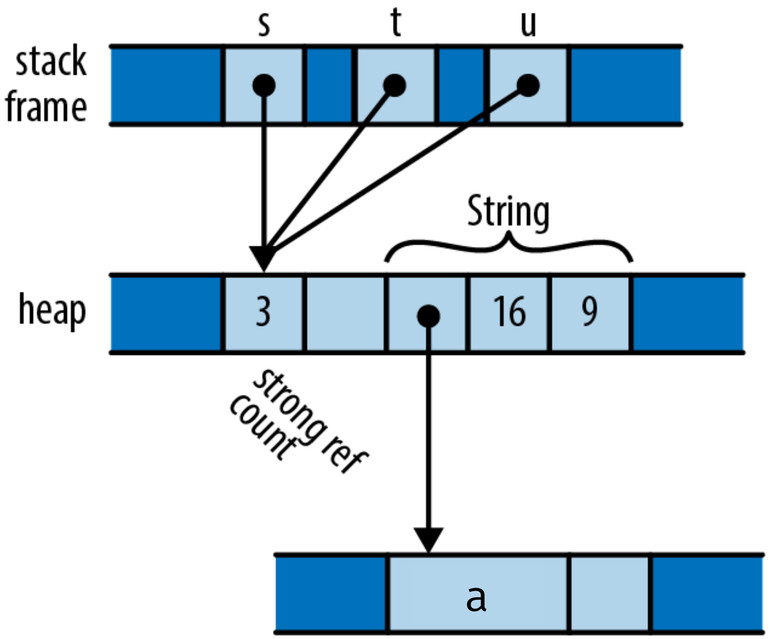
Rc e Arc: Proprietà Condivisa

Se nel caso generale è sufficiente che i valori abbiano proprietari unici, a volte è necessario condividere la proprietà dei valori tra un insieme di proprietari, ad esempio, con chi può leggere il valore in modo sicuro, perché è immutabile.

Per questi casi, Rust fornisce i tipi "puntatore con conteggio dei riferimenti", chiamati **Reference Count** (**Rc<T>**) e "conteggio atomico dei riferimenti" (**Arc<T>**).

Intuitivamente, i tipi Rc/Arc sono abbastanza simili (il secondo aggiunge alcuni controlli per rendere il conteggio dei riferimenti sicuro per la concorrenza) e tengono traccia del numero di riferimenti a un valore per determinare se il valore è ancora in uso o meno. Se ci sono zero riferimenti a un valore, il valore può essere rimosso senza che nessun riferimento venga invalidato.

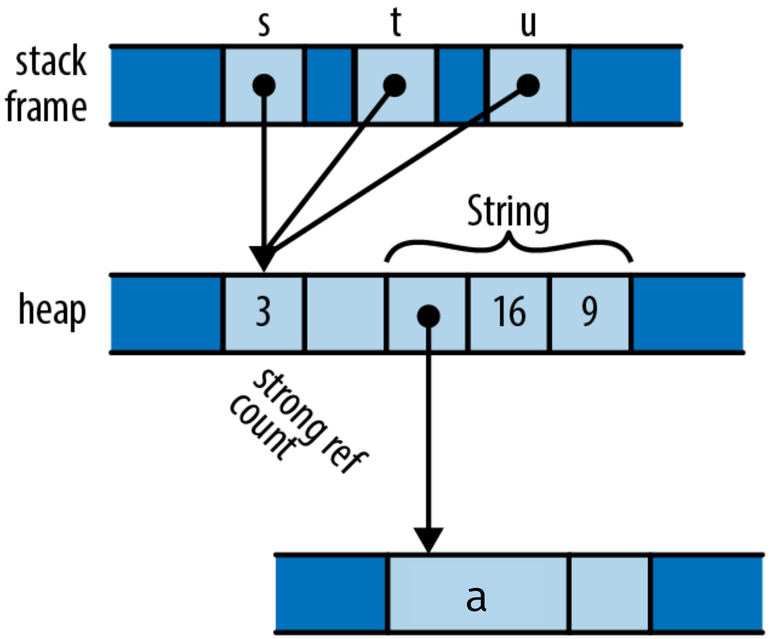
```
let s: Rc<String> = Rc::new("a".to_string());
let t: Rc<String> = s.clone();
let u: Rc<String> = s.clone();
```



Rc e Arc: Proprietà Condivisa

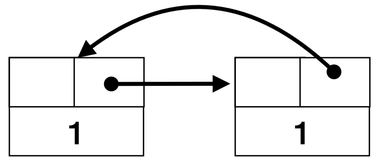
Nell'esempio, ognuno dei tre puntatori Rc<String> si riferisce allo stesso blocco di memoria, che contiene un contatore di riferimenti e lo spazio per la stringa.

Ai puntatori Rc si applicano le solite regole di proprietà: ogni clone incrementa il contatore, mentre ogni deallocazione di un puntatore lo diminuisce, finché l'ultimo esistente non viene deallocato e il runtime dealloca anche la Stringa referenziata nell'heap.




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

Rc e Arc: Proprietà Condivisa



Come abbiamo discusso per i garbage collector, un problema ben noto del reference counting è che non è possibile deallocare strutture ricorsive.

Anche i tipi con conteggio dei riferimenti possono soffrire di questo problema e Rust rende tali situazioni esplicite e rare.

Infatti, non possiamo creare un ciclo senza che, a un certo punto, un valore più vecchio punti a un valore più recente, il che implica che almeno il valore più vecchio sia mutabile. Questo è il motivo per cui **i puntatori Rc mantengono immutabili i loro referenti**. Naturalmente, in alcuni casi è necessario creare strutture cicliche in memoria. Rust soddisfa questa esigenza con le le **Reference Cells**, che forniscono un'interfaccia per consentire ai valori referenziati di cambiare, a costo di rinunciare alle garanzie statiche sull'uso dei puntatori per i controlli di runtime (che potrebbero far fallire il programma, se il riferimento è usato in modo improprio).

Prendere in prestito (borrowing) i riferimenti

I tipi di puntatori visti finora sono owners per cui, quando rimuoviamo il proprietario rimuoviamo anche il valore riferito. Rust ha anche dei puntatori non proprietari, chiamati **riferimenti**, che non hanno alcun effetto sulla vita dei valori riferiti e, al contrario, dipendono da essi. Il mantra per questo tipo di variabili è

I riferimenti non devono mai sopravvivere ai loro valori riferiti.

Capire il motivo per tale necessità è molto diretto. Dato che eliminiamo un valore quando il suo proprietario viene rimosso, se un riferimento al valore vivesse più del proprietario del valore stesso, il riferimento sarebbe dangling (punterebbe ad una cella di memoria di cui non conosciamo il contenuto).

Prendere in prestito (borrowing) i riferimenti

Per questo motivo, nel linguaggio di Rust, **la creazione di un riferimento a un valore si chiama "prendere in prestito" (borrowing) il valore**.

Un riferimento dà accesso a un valore senza influenzarne la proprietà e può essere di due tipi:

- **i riferimenti condivisi** (**shared references**) consentono all'utente di **leggere ma non di modificare** il valore. Trattandosi di un riferimento in sola lettura, possiamo tranquillamente avere tutti i riferimenti condivisi a un valore che vogliamo. Questa è la forma più semplice di riferimento, poiché utilizza la stessa sintassi vista, ad esempio, in C, dove `&e` genera un riferimento condiviso al valore contenuto nella variabile;
- I **riferimenti mutabili** (**mutable references**) consentono all'utente di **leggere e modificare** il valore, ma rendono impossibile avere altri riferimenti a quel valore attivi allo stesso tempo. La sintassi per generare un riferimento mutabile è `&mut` e genera un riferimento mutabile al valore contenuto nella variabile e.

In Rust i riferimenti non sono mai `NULL`. L'assenza del valore `NULL` per i puntatori deriva dai controlli effettuati dal compilatore, che si assicura che le variabili non vengano mai utilizzate prima di essere inizializzate (non ci sono puntatori wild) e dal fatto che non esiste un modo (non-sicuro) per far sì che il linguaggio generi riferimenti pericolosi.

Lifetimes

Il compilatore di Rust convalida i “prestiti” ragionando sulle lifetime delle variabili; essenzialmente **si** assicura che **nessun riferimento sopravviva al proprietario da cui prende in prestito il valore**.

```
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
```

Nell'esempio sopra, il compilatore si lamenta perché x accederebbe a una posizione di memoria non controllata, dato che il valore riferito da y viene deallocato una volta uscito dallo scope interno.

Quindi, le **interazioni tra le variabili (prestiti) non modificano i lifetime delle variabili, ma impongono vincoli tra queste che il compilatore deve controllare**. Nell'esempio, x=&y impone il vincolo che il tempo di vita di y debba essere uguale a quello di x, il che fa scattare l'errore del compilatore.

In realtà, il controllo sui lifetimes effettuato da Rust è più raffinato di questa regola grossolana, e considera le lifetimes di x e y correlate fintanto che x è assegnato a y e abbiamo bisogno di leggere x (come nell'esempio, in cui cerchiamo di stamparlo)—se eliminassimo la lettura di x il programma verrebbe compilato.

Annotazioni delle Lifetime

Sebbene il compilatore di Rust sia abbastanza intelligente nel dedurre le lifetimes (un po' come accade con l'inferenza di tipo), in alcuni casi è necessario fornire informazioni sulle lifetimes, attraverso annotazioni che chiariscono situazioni ambigue.

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

Per esempio, sotto, dato che non è chiaro, il compilatore chiede al programmatore di indicare se il riferimento restituito dalla funzione snd deve considerare come lifetime quella di a di b.

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

Annotazioni delle Lifetime

```
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 );
}
```

Per risolvere questo problema, dobbiamo usare i **parametri di lifetime** che (per certi versi, ricordano i parametri di tipo del polimorfismo universale, e in effetti) in Rust, queste annotazioni sono a livello di tipo.

Rust inferisce autonomamente le lifetimes delle variabili, insieme ai tipi. E.g., quando creiamo una variabile, scrivendo `let x = 5`, il sistema di tipi assegna il tipo `int` alla variabile x e un nuovo parametro lifetime `'l`, che diventa vincolato da come usiamo il valore riferito.

Annotazioni delle Lifetime

```
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 );
}
```

Le **annotazioni delle lifetime rendono esplicite le durate di variabili** correlate.

Ad esempio, nella struct S a fianco, dichiariamo due parametri di lifetime, dove uno ('a) identifica la durata di un campo (x), mentre l'altro ('b) identifica le durate di due campi (y e z).

Questo rende il nostro esempio compilabile, perché la lifetime di x può sopravvivere a quella di y e z, dato che sono distinte. Senza questa annotazione, Rust (di base) considererebbe x, y e z associate alla stessa lifetime, dando luogo a un errore di compilazione.

Annotazioni delle Lifetime

Il concetto delle annotazioni delle lifetimes è simile per le funzioni, dove a ogni tipo di variabile e al tipo di ritorno è associata una durata diversa, ad es,

```
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 );
}
```

Annotazioni delle Lifetime

Nel nostro esempio, poiché la funzione `snd` decide a tempo di esecuzione quale riferimento, tra a e b, restituire, chiariamo che devono condividere la 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 );
}
```

Concretamente, la funzione ha solo un parametro di durata, `'l`, che lega insieme (cioè vincola ad essere uguali) le durate di a e b.

Il **significato dell'annotazione del tipo di ritorno** con `'l` è leggermente diverso: non significa che a, b e il valore di ritorno di `snd` devono avere tutti la stessa durata, ma piuttosto che **il riferimento restituito viene preso in prestito dall'argomento con la stessa annotazione** (in questo caso, a o b).

Annotazioni delle Lifetime

Per chiarire ulteriormente, vediamo un'estensione dell'esempio visto prima.

Ricordiamo che il vincolo di durata su snd è che il riferimento restituito non deve superare quelli di a e di 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 );
}
```

L'esempio infrange questo vincolo, dato che la lifetime di r supera quello di b (quello di a va bene).

```
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
```

Riferimenti Mutabili

I valori presi in prestito (**borrowed**) da riferimenti condivisi (**shared references**) sono in sola lettura. Al contrario, un valore preso in prestito da un riferimento mutabile è raggiungibile esclusivamente tramite quel riferimento. Inoltre, per tutta la lifetime di un riferimento mutabile non deve esistere un altro percorso utilizzabile verso il suo valore o verso qualsiasi valore raggiungibile da esso. **Gli unici riferimenti la cui lifetime può sovrapporsi a un riferimento mutabile sono quelli presi in prestito dal riferimento mutabile stesso.**

```
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 );
}~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Nell'esempio, violiamo la regola per i riferimenti mutabili (ultima istruzione): prendiamo in prestito sia un riferimento mutabile che immutabile a `greet`.

Ovviamente, viceversa, possiamo vedere la violazione dal lato del riferimento condiviso: poiché prendiamo in prestito un riferimento condiviso a `greet`, questo deve essere in sola lettura, quindi il prestito di un riferimento mutabile è illegale.

Recap: Ownership e Shared e Mutable References