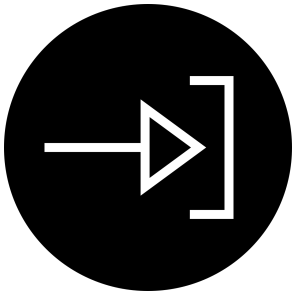


Smart Pointer

Accedere alla memoria in modo controllato

Operazioni sui puntatori

- Ogni valore manipolato da un programma è memorizzato nello spazio di indirizzamento del processo
  - L'operatore `&` (e `&mut`, in Rust) permette, in C, C++ e Rust, di ottenere l'indirizzo del primo byte in cui è memorizzato
  - Nel caso di Rust, tale operatore attiva il borrow checker che vigila sull'uso che viene fatto dell'indirizzo ottenuto, imponendo tutti i vincoli di sanità necessari a fornire le garanzie date dal modello del linguaggio
- L'operazione duale, detta dereferenza (dereferencing) o risoluzione del riferimento, trasforma un indirizzo nel corrispondente valore puntato
  - Si esprime con l'operatore `*` (e `->` in C e C++ o `.` in Rust)
  - Quando viene applicato ad un puntatore nativo o ad un riferimento Rust, il compilatore dà accesso al dato puntato

Operazioni sui puntatori

- Sia Rust che C++ permettono di ridefinire il comportamento degli operatori del linguaggio per tipi arbitrari
  - Entrambi permettono inoltre di definire tipi generici, che possono essere espansi in una molteplicità di tipi concreti, in funzione di come vengono utilizzati
- Questi meccanismi, applicati agli operatori di dereferenza abilitano la definizione di tipi che “sembrano” puntatori (dal punto di vista sintattico) ma che hanno ulteriori caratteristiche
  - Garanzia di inizializzazione e rilascio
  - Conteggio dei riferimenti
  - Accesso esclusivo con attesa
  - …
- Ciò ha portato all’introduzione del concetto di “smart pointer”
  - E alla sua diffusione sia nelle librerie standard C++, dove l’uso dei puntatori nativi è causa di errori frequenti…
  - …sia nelle librerie standard Rust, che ne hanno abbracciato l’idea per rappresentare puntatori che possiedono i dati a cui puntano (in contrapposizione ai riferimenti, che godono del solo prestito)

Uso dei puntatori

- Tramite l'uso di puntatori è possibile costruire strutture dati dinamiche (dalla topologia non prevedibile a priori e/o variabile nel tempo) come grafi, alberi, liste
  - Se questo, da un lato, offre grandi libertà al programmatore, dall'altro lo espone ad una serie di problemi legati alla difficoltà di dedurre la di correttezza del codice che lo manipola
- Le regole restrittive imposte dal borrow checker di Rust impediscono, con l'uso di soli riferimenti, la creazione di strutture cicliche
  - Ogni valore in Rust è parte di un solo albero la cui radice è contenuta in una qualche variabile
  - Questa minore capacità espressiva abilita, però, analisi più approfondite ed è alla base delle garanzie di sanità offerte dal linguaggio
- Attraverso l'uso di smart pointer come Rc<T> e Arc<T>, è possibile avere più possessori di uno stesso valore
  - Smart pointer come std::rc::Weak e std::sync::Weak offrono invece la possibilità di avere strutture cicliche, nel rispetto di alcune restrizioni

Smart pointer in C++

- **`std::unique_ptr<T>`**
  - Modella il possesso ad un valore di tipo T allocato sullo heap e rilasciato automaticamente quando il puntatore esce dal proprio scope sintattico
  - Non può essere copiato, ma solo mosso in un'altra variabile
  - Creato con la funzione std::make_unique<T>(T val)
- **`std::shared_ptr<T>`**
  - Riferimento ad un valore di tipo T allocato sullo heap insieme ad una struttura di controllo che mantiene il numero di riferimenti esistenti
  - Può essere copiato: la copia indica lo stesso blocco dell'originale, ma incrementa il conteggio dei riferimenti
  - Quanto viene distrutto, il contatore dei riferimenti viene decrementato: se raggiunge 0, il blocco viene rilasciato
  - Può essere usato con codice concorrente
  - Se si crea un grafo ciclico, il meccanismo del conteggio dei riferimenti impedisce il rilascio
  - Creato con la funzione std::make_shared<T>(T val)

unique_ptr<T>

- Internamente contiene solo un puntatore
  - La rimozione/ridefinizione dei costruttori di copia e movimento, degli operatori di assegnazione (per copia e movimento) e del distruttore garantisce che possa essere usato solo nel rispetto della sua semantica
- Se il puntatore viene riassegnato o distrutto (esce dal suo scope sintattico), il blocco viene rilasciato
  - E' anche possibile definire funzioni di rilascio custom, alternative all'invocazione della funzione delete(...)

```
{
  std::unique_ptr<int> p =
         std::make_unique<int>(5);

  int i = *p;

  *p = 7;

}
```

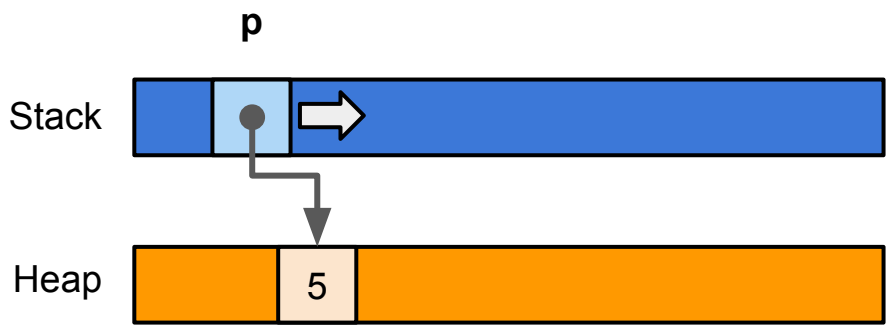
unique_ptr<T>

```
{
  std::unique_ptr<int> p =
        std::make_unique<int>(5);

  int i = *p;

  *p = 7;

}
```



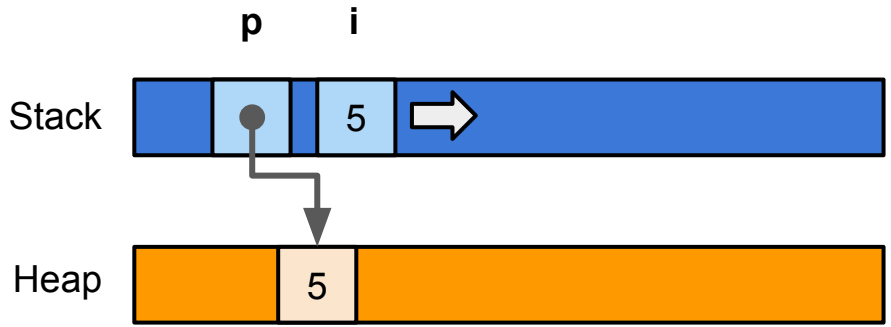
unique_ptr<T>

```
{
  std::unique_ptr<int> p =
         std::make_unique<int>(5);

  int i = *p;

  *p = 7;

}
```



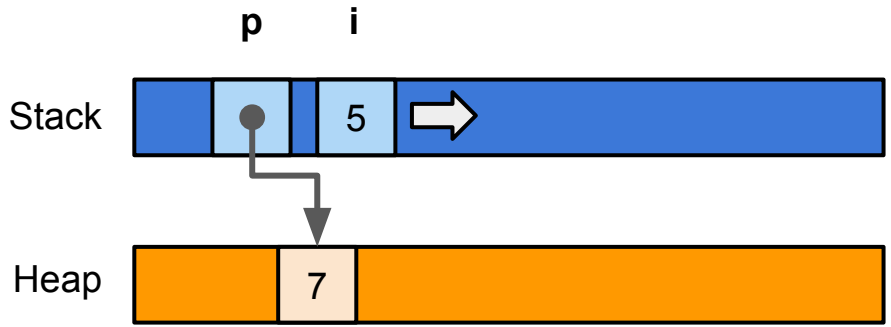
unique_ptr<T>

```
{
  std::unique_ptr<int> p =
         std::make_unique<int>(5);

  int i = *p;

  *p = 7;

}
```



unique_ptr<T>

```
{
  std::unique_ptr<int> p =
         std::make_unique<int>(5);

  int i = *p;

  *p = 7;

}
```

shared_ptr<T>

- Mantiene la proprietà condivisa a un blocco di memoria referenziato da un puntatore nativo
  - Molti oggetti possono referenziare lo stesso blocco
  - Quando tutti sono stati distrutti o resettati, il blocco viene rilasciato
- Per default, il blocco referenziato viene rilasciato tramite l'operatore delete
  - In fase di costruzione di uno shared_ptr, è possibile specificare un meccanismo di rilascio alternativo
- Un oggetto di questo tipo può anche non contenere alcun puntatore valido
  - Se è stato inizializzato o resettato al valore nullptr
- L'overhead di questa classe è significativo, conviene tenerne conto
  - La sua implementazione tipica è basata su un fat pointer, costituito da due puntatori consecutivi: il primo punto al dato, il secondo al blocco di controllo
- Viene costruito tramite la funzione std::make_shared<T>(T t)

shared_ptr<T>

- Un’implementazione tipica del blocco di controllo (metadati) contiene tre campi
  - Il contatore dei riferimenti forti
  - Il contatore dei riferimenti deboli
  - Il puntatore al dato
- Quanto viene creato un oggetto di tipo shared_ptr<T>, il contatore dei riferimenti forti vale 1, quello dei riferimenti deboli vale 0
  - Se viene effettuata una copia, il contatore dei riferimenti forti viene incrementato atomicamente
  - Quando uno shared_ptr<T> esce dal proprio scope sintattico, il contatore dei riferimenti forti viene decrementato atomicamente: se il risultato è 0, il blocco contenente il dato viene rilasciato
  - Se anche il contatore dei riferimenti deboli vale 0, viene rilasciato anche il blocco di controllo

shared_ptr<T>

```
{
  std::shared_ptr<int> p =
        std::make_shared<int>(5);

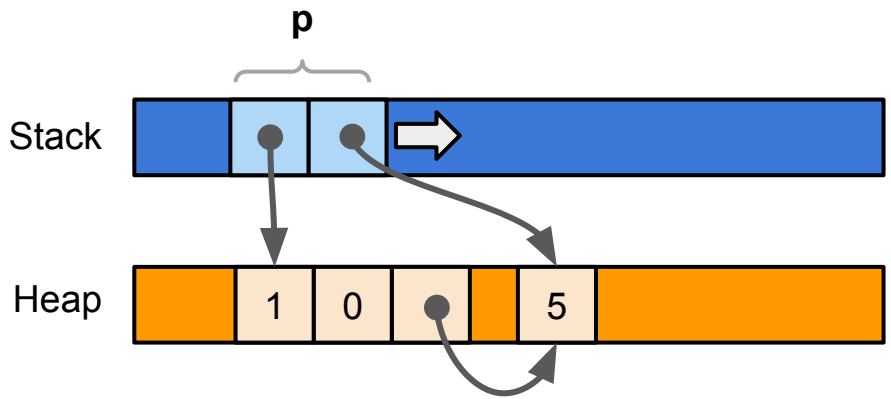
  {
    auto q = p;

    *q = 3;

  }

  *p = 7;

}
```



shared_ptr<T>

```
{
  std::shared_ptr<int> p =
         std::make_shared<int>(5);

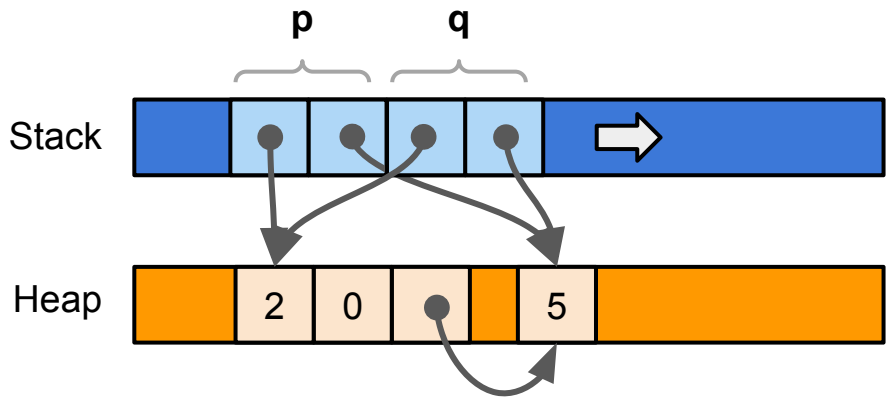
  {
    auto q = p;

    *q = 3;

  }

  *p = 7;

}
```



shared_ptr<T>

```
{
  std::shared_ptr<int> p =
         std::make_shared<int>(5);

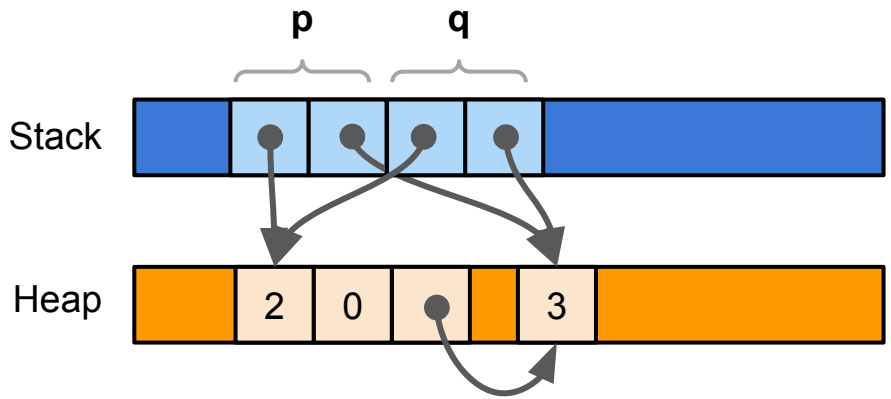
  {
    auto q = p;

    *q = 3;

  }

  *p = 7;

}
```



shared_ptr<T>

```
{
  std::shared_ptr<int> p =
         std::make_shared<int>(5);

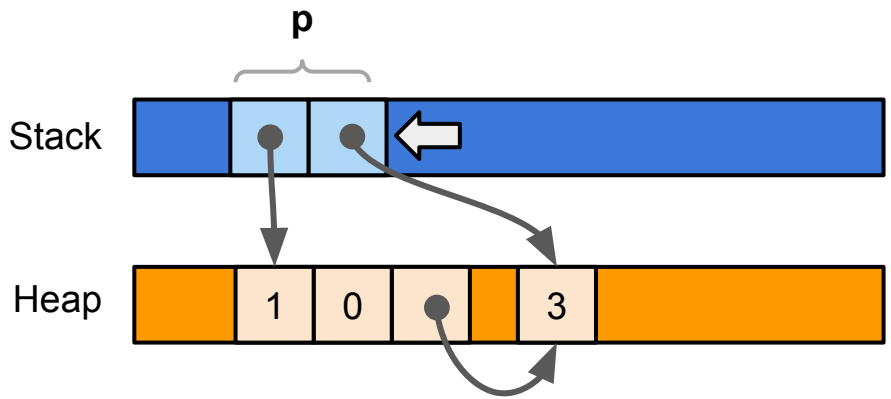
  {
    auto q = p;

    *q = 3;

  }

  *p = 7;

}
```



shared_ptr<T>

```
{
  std::shared_ptr<int> p =
         std::make_shared<int>(5);

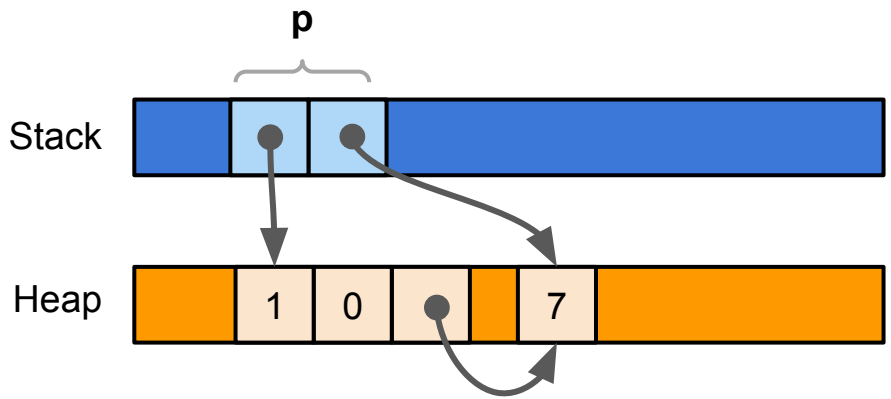
  {
    auto q = p;

    *q = 3;

  }

  *p = 7;

}
```



shared_ptr<T>

```
{
  std::shared_ptr<int> p =
         std::make_shared<int>(5);

  {
    auto q = p;

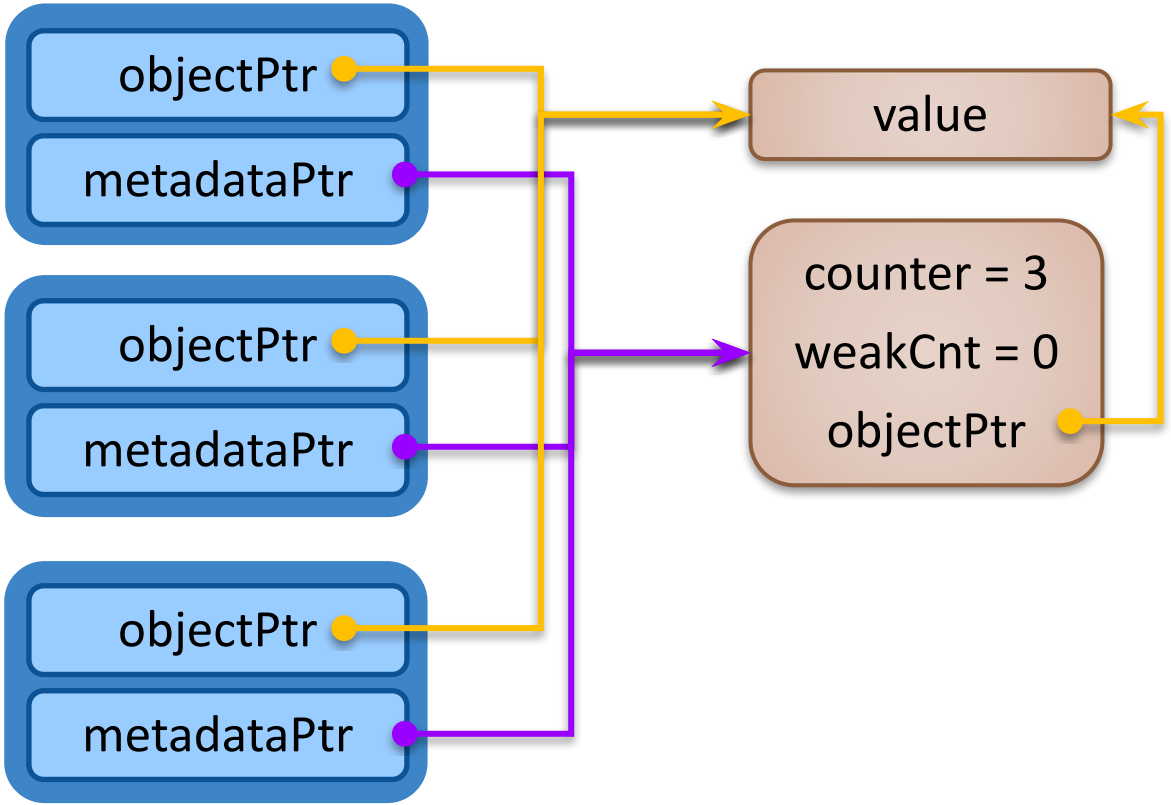
    *q = 3;

  }

  *p = 7;

}
```

shared_ptr<T>



Dipendenze cicliche

- I conteggio dei riferimenti dovrebbe garantire il rilascio della memoria in modo deterministico
  - Non appena un oggetto non ha più riferimenti viene liberato
- In alcuni casi, tuttavia, non funziona
  - Se si forma un ciclo di dipendenze (A→B, B→A) il contatore non può mai annullarsi, anche se gli oggetti A e B non sono più conosciuti da nessuno
  - Esempio tipico: lista doppiamente collegata
- Occorre evitare la creazione di cicli ricorrendo a oggetti che permettono di raggiungere la destinazione senza partecipare al conteggio dei riferimenti
  - std::weak_ptr<T>

weak_ptr<T>

- Usato per creare dipendenze cicliche senza incrementare il numero dei riferimenti esistenti
  - Per essere dereferenziato, deve essere acquisito con il metodo lock() che ritorna uno shared_ptr<T>
  - Se l'oggetto è già stato rilasciato, il puntatore ritornato è vuoto (contiene null_ptr)
- Si crea un weak_ptr<T> a partire da uno shared_ptr<T>
  - Internamente contiene un puntatore al solo blocco di controllo dello shared_ptr
  - Il contatore dei riferimenti deboli viene incrementato atomicamente
- Quando un weak_ptr viene distrutto, il contatore dei riferimenti deboli viene decrementato atomicamente
  - Se il risultato è 0 e non sono presenti riferimenti forti, il blocco di controllo viene rilasciato

Smart pointer in Rust

- Rust offre una varietà maggiore di smart pointer rispetto al C++, allo scopo di coprire ulteriori casi e definire ottimizzazioni possibili nel caso specifico di programmi puramente sequenziali piuttosto che di programmi concorrenti
  - Alcuni di questi ricalcano abbastanza fedelmente le astrazioni offerte dal C++ (Box<T>, Rc<T>, Arc<T>, Weak<T>)
  - Altri sono peculiari del modello introdotto dal linguaggio (Cell<T>, RefCell<T>, Cow<T>, Mutex<T>, RwLock<T>)
- In generale, sono realizzati mediante struct che contengono le necessarie informazioni e che implementano i tratti Deref e DerefMut
  - Quando il compilatore incontra l'espressione `*ptr` (dove il tipo di ptr implementa tali tratti) la trasforma in `* ptr.deref()` o `* ptr.deref_mut()` a seconda dei casi

I tratti Deref e DerefMut

```
trait Deref {
 type Target: ?Sized;
 fn deref(&self) -> &Self::Target;
}
trait DerefMut: Deref {
 fn deref_mut(&mut self) -> &mut Self::Target;
}
```

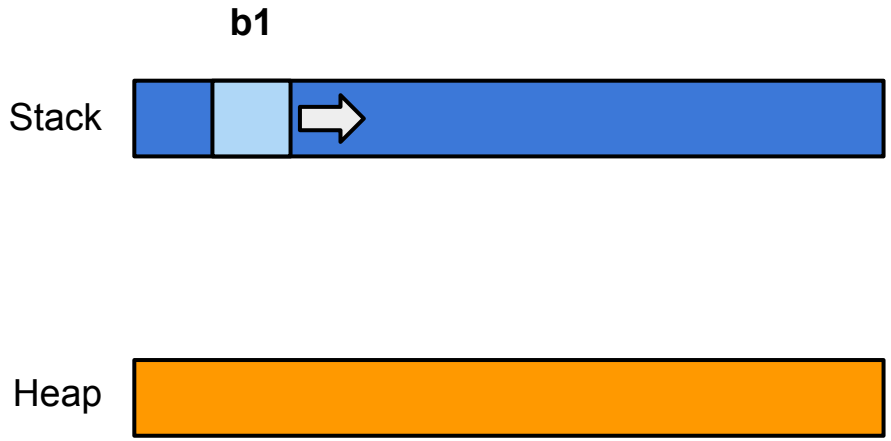
std::Box<T>

- Struttura che incapsula un puntatore ad un blocco allocato dinamicamente sullo heap all'atto della sua costruzione (tramite il metodo **`Box::new(t)`**)
  - Il dato puntato è **posseduto** da Box: quando la struttura esce dal proprio scope sintattico, il blocco sullo heap viene rilasciato automaticamente, grazie all'implementazione del tratto **`Drop`**
  - E' possibile anticipare il rilascio del blocco, invocando la funzione **`drop(b)`**
- Se la struttura viene mossa in un'altra variabile (o ritornata da una funzione), il possesso del puntatore passa alla destinazione che diventa responsabile del suo rilascio
  - Questo rende possibile ottenere cicli di vita che si estendono oltre la durata della funzione in cui il dato è stato creato
- Il tipo T può avere una dimensione non nota in fase di compilazione (ovvero non implementare il tratto Sized)
  - In questo caso, l'oggetto di tipo **`Box<T>`** si trasforma in un fat pointer formato da un puntatore seguito da un intero di dimensione **`usize`** contente la lunghezza del dato puntato
  - Analogamente, se al posto del tipo concreto T si indica un oggetto-tratto (**`dyn Trait`**), si ha un fat pointer composto da due puntatori: quello al dato sullo heap e quello a vtable del tratto

std::Box<T>

```
fn produce(odd: bool) -> Box<i32> {
    let mut b = Box::new(0);
    if odd { *b = 5; }
    return b;
}

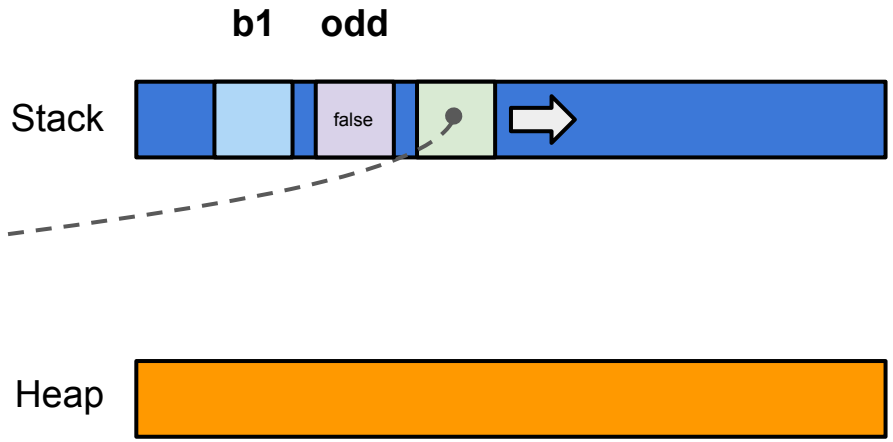
fn main() {
    let b1 = produce(false);
    println!("b1: {}", b1);
    let b2 = produce(true);
    drop(b1);
    println!("b2: {}", b2);
}
```



std::Box<T>

```
fn produce(odd: bool) -> Box<i32> {
    let mut b = Box::new(0);
    if odd { *b = 5; }
    return b;
}

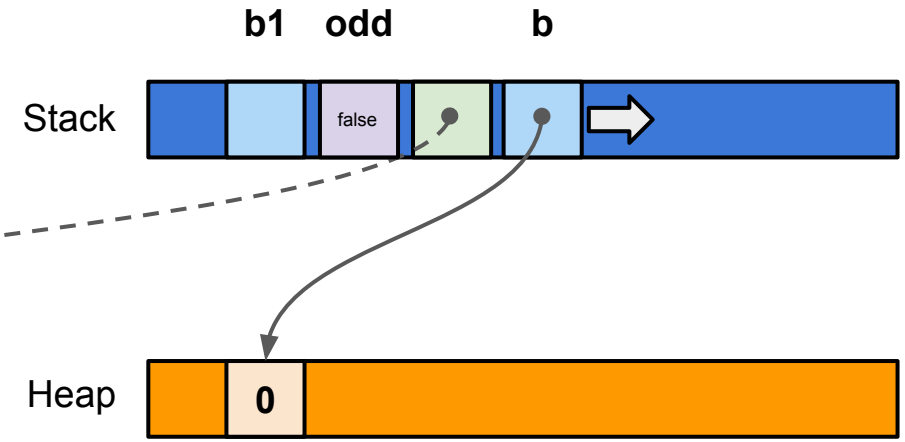
fn main() {
    let b1 = produce(false);
    println!("b1: {}", b1);
    let b2 = produce(true);
    drop(b1);
    println!("b2: {}", b2);
}
```



std::Box<T>

```
fn produce(odd: bool) -> Box<i32> {
    let mut b = Box::new(0);
    if odd { *b = 5; }
    return b;
}

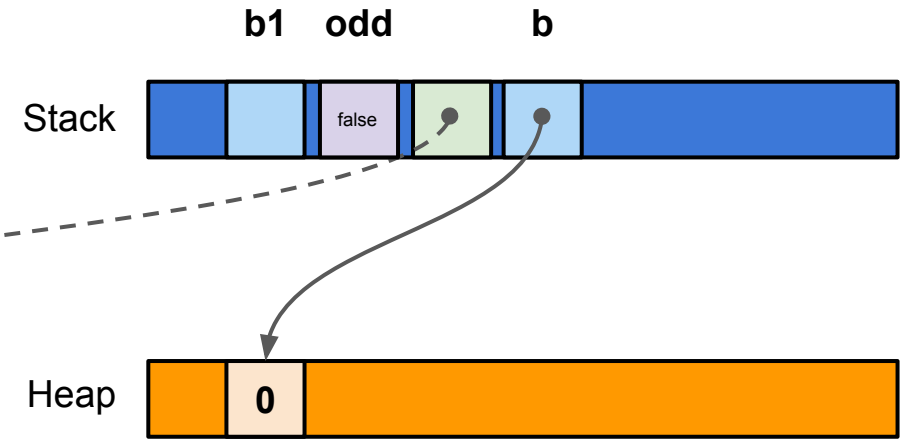
fn main() {
    let b1 = produce(false);
    println!("b1: {}", b1);
    let b2 = produce(true);
    drop(b1);
    println!("b2: {}", b2);
}
```



std::Box<T>

```
fn produce(odd: bool) -> Box<i32> {
    let mut b = Box::new(0);
    if odd { *b = 5; }
    return b;
}

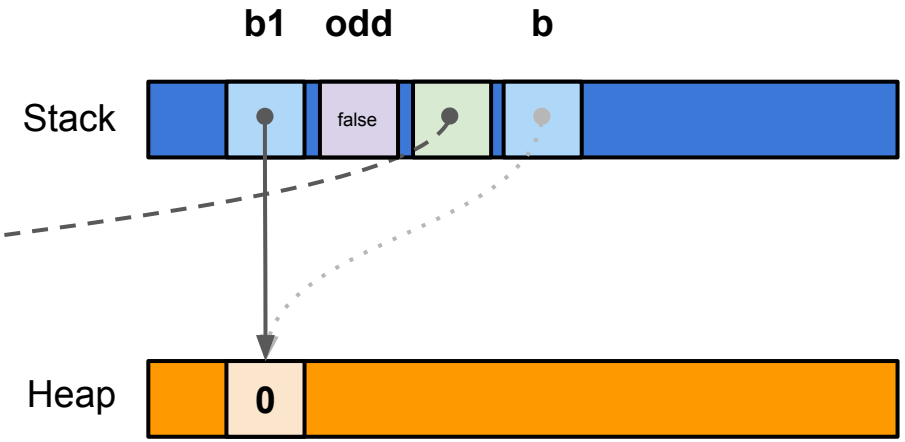
fn main() {
    let b1 = produce(false);
    println!("b1: {}", b1);
    let b2 = produce(true);
    drop(b1);
    println!("b2: {}", b2);
}
```



std::Box<T>

```
fn produce(odd: bool) -> Box<i32> {
    let mut b = Box::new(0);
    if odd { *b = 5; }
    return b;
}

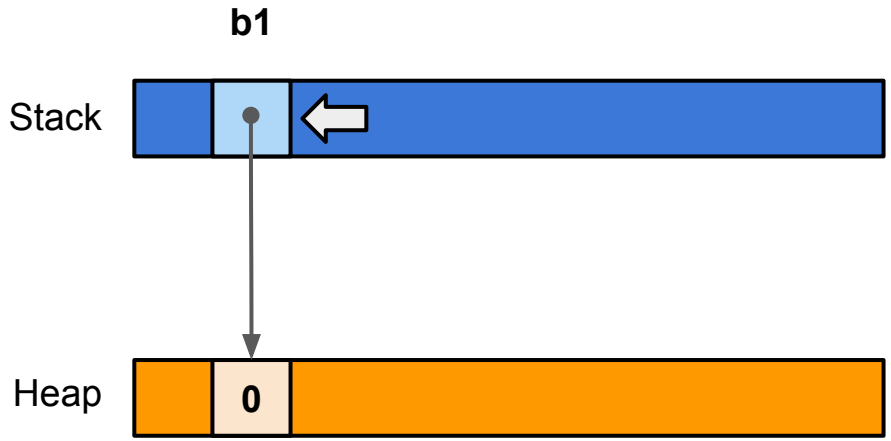
fn main() {
    let b1 = produce(false);
    println!("b1: {}", b1);
    let b2 = produce(true);
    drop(b1);
    println!("b2: {}", b2);
}
```



std::Box<T>

```
fn produce(odd: bool) -> Box<i32> {
    let mut b = Box::new(0);
    if odd { *b = 5; }
    return b;
}

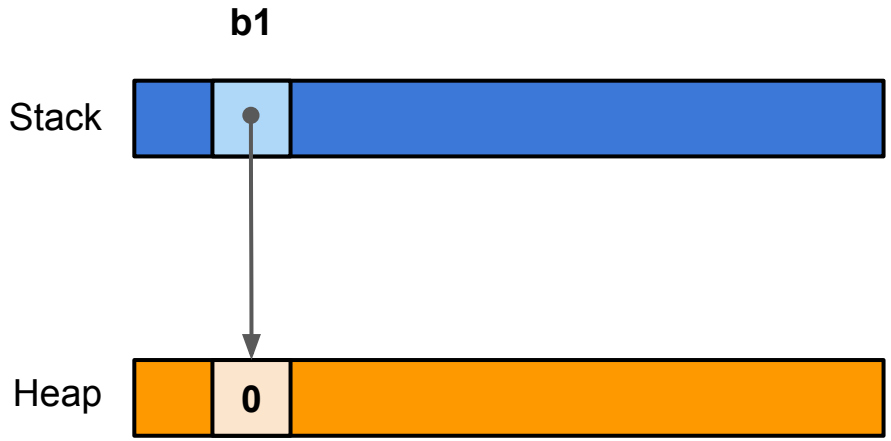
fn main() {
    let b1 = produce(false);
    println!("b1: {}", b1);
    let b2 = produce(true);
    drop(b1);
    println!("b2: {}", b2);
}
```



std::Box<T>

```
fn produce(odd: bool) -> Box<i32> {
    let mut b = Box::new(0);
    if odd { *b = 5; }
    return b;
}

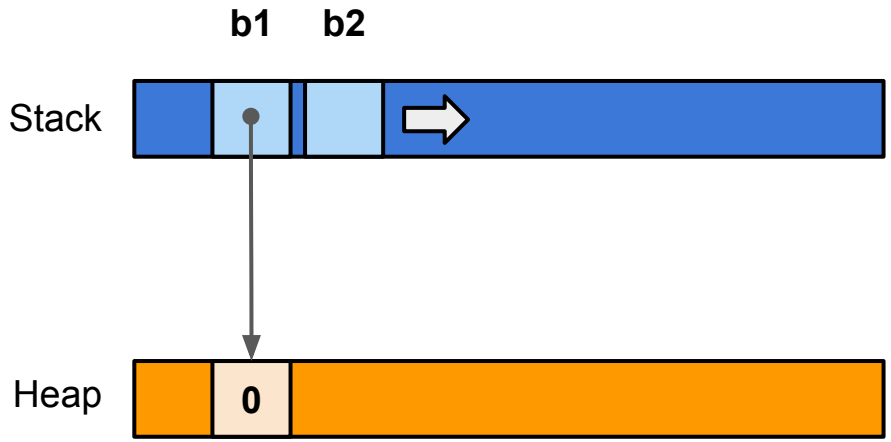
fn main() {
    let b1 = produce(false);
    println!("b1: {}", b1);
    let b2 = produce(true);
    drop(b1);
    println!("b2: {}", b2);
}
```



std::Box<T>

```
fn produce(odd: bool) -> Box<i32> {
    let mut b = Box::new(0);
    if odd { *b = 5; }
    return b;
}

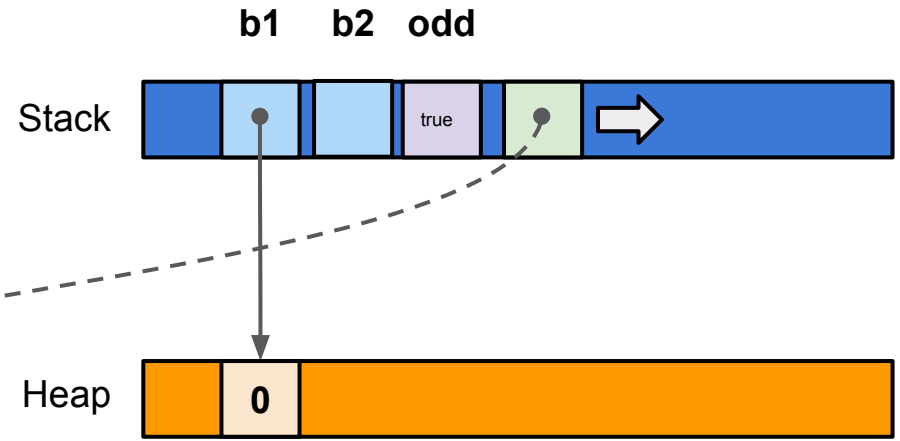
fn main() {
    let b1 = produce(false);
    println!("b1: {}", b1);
    let b2 = produce(true);
    drop(b1);
    println!("b2: {}", b2);
}
```



std::Box<T>

```
fn produce(odd: bool) -> Box<i32> {
    let mut b = Box::new(0);
    if odd { *b = 5; }
    return b;
}

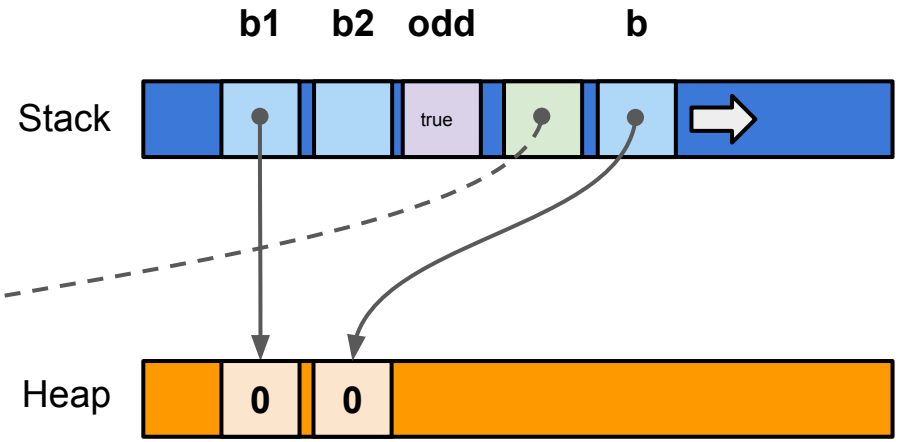
fn main() {
    let b1 = produce(false);
    println!("b1: {}", b1);
    let b2 = produce(true);
    drop(b1);
    println!("b2: {}", b2);
}
```



std::Box<T>

```
fn produce(odd: bool) -> Box<i32> {
    let mut b = Box::new(0);
    if odd { *b = 5; }
    return b;
}

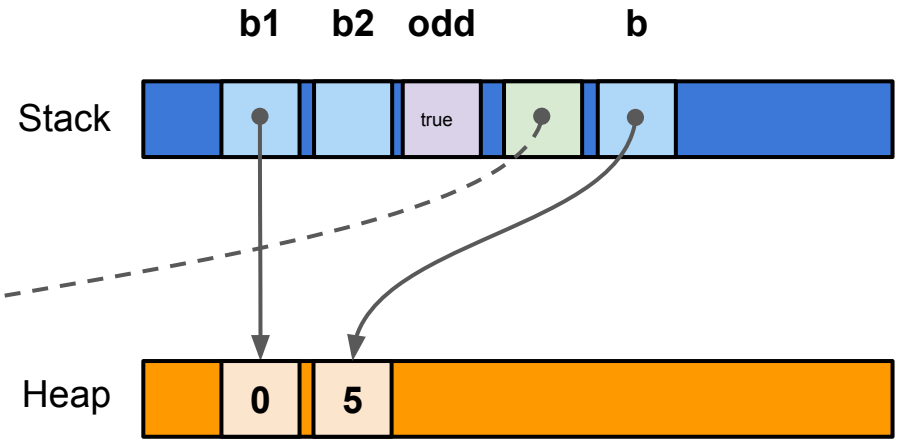
fn main() {
    let b1 = produce(false);
    println!("b1: {}", b1);
    let b2 = produce(true);
    drop(b1);
    println!("b2: {}", b2);
}
```



std::Box<T>

```
fn produce(odd: bool) -> Box<i32> {
    let mut b = Box::new(0);
    if odd { *b = 5; }
    return b;
}

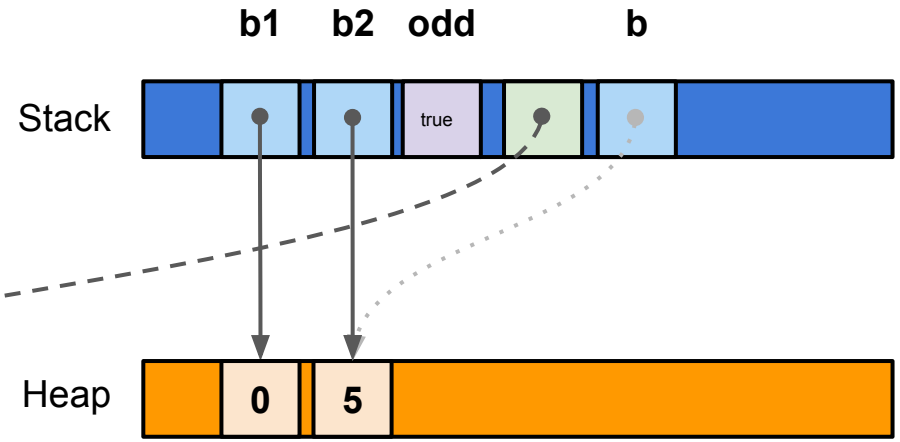
fn main() {
    let b1 = produce(false);
    println!("b1: {}", b1);
    let b2 = produce(true);
    drop(b1);
    println!("b2: {}", b2);
}
```



std::Box<T>

```
fn produce(odd: bool) -> Box<i32> {
    let mut b = Box::new(0);
    if odd { *b = 5; }
    return b;
}

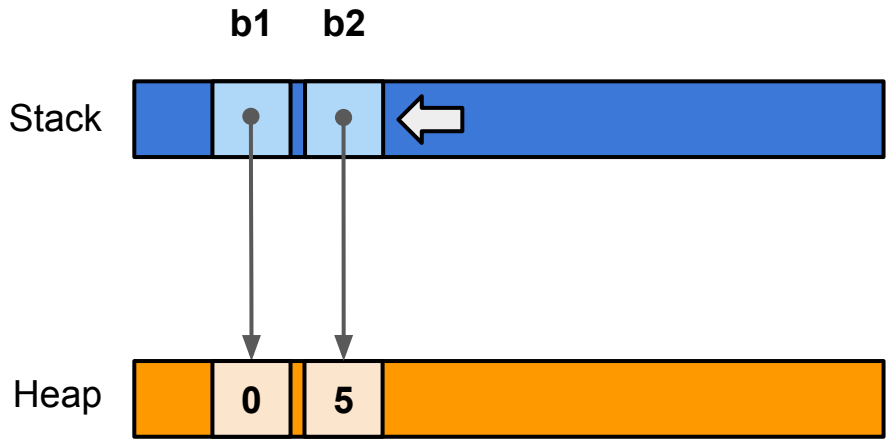
fn main() {
    let b1 = produce(false);
    println!("b1: {}", b1);
    let b2 = produce(true);
    drop(b1);
    println!("b2: {}", b2);
}
```



std::Box<T>

```
fn produce(odd: bool) -> Box<i32> {
    let mut b = Box::new(0);
    if odd { *b = 5; }
    return b;
}

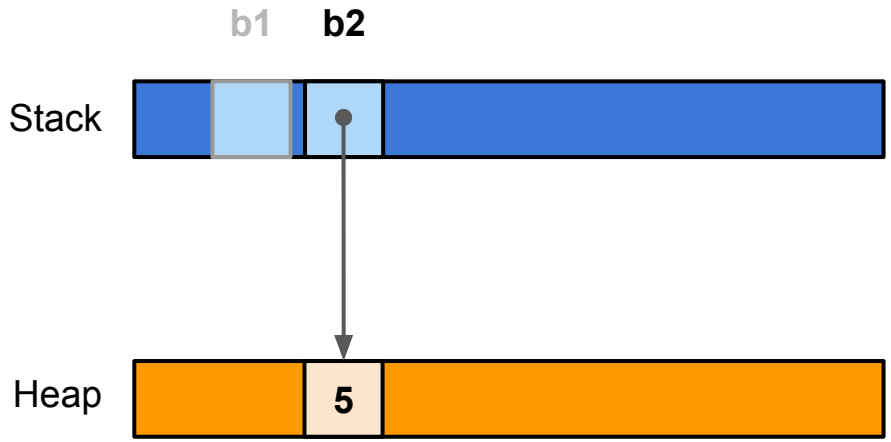
fn main() {
    let b1 = produce(false);
    println!("b1: {}", b1);
    let b2 = produce(true);
    drop(b1);
    println!("b2: {}", b2);
}
```



std::Box<T>

```
fn produce(odd: bool) -> Box<i32> {
    let mut b = Box::new(0);
    if odd { *b = 5; }
    return b;
}

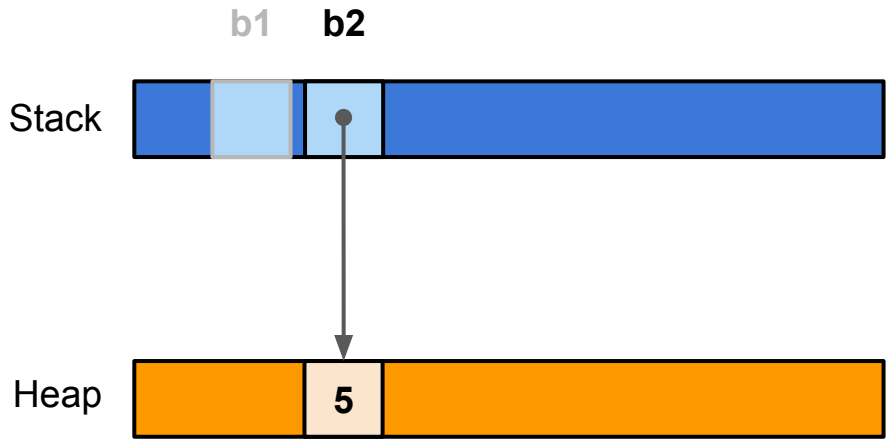
fn main() {
    let b1 = produce(false);
    println!("b1: {}", b1);
    let b2 = produce(true);
    drop(b1);
    println!("b2: {}", b2);
}
```



std::Box<T>

```
fn produce(odd: bool) -> Box<i32> {
    let mut b = Box::new(0);
    if odd { *b = 5; }
    return b;
}

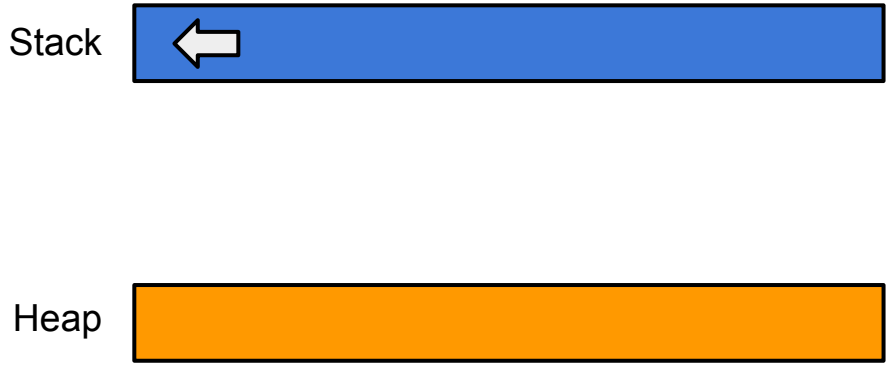
fn main() {
    let b1 = produce(false);
    println!("b1: {}", b1);
    let b2 = produce(true);
    drop(b1);
    println!("b2: {}", b2);
}
```



std::Box<T>

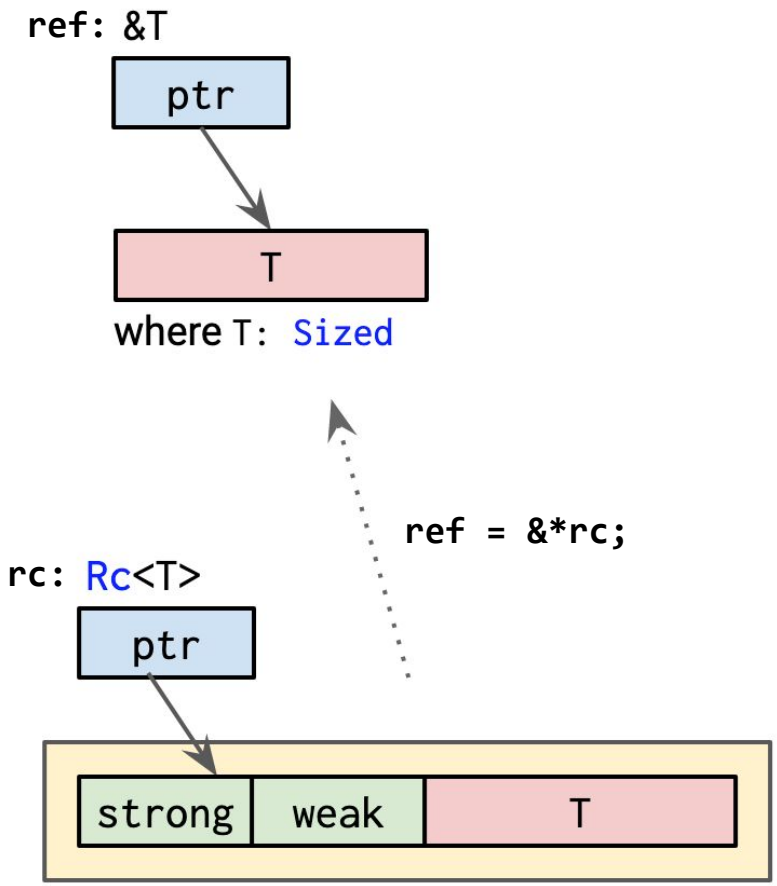
```
fn produce(odd: bool) -> Box<i32> {
    let mut b = Box::new(0);
    if odd { *b = 5; }
    return b;
}

fn main() {
    let b1 = produce(false);
    println!("b1: {}", b1);
    let b2 = produce(true);
    drop(b1);
    println!("b2: {}", b2);
}
```



std::rc::Rc<T>

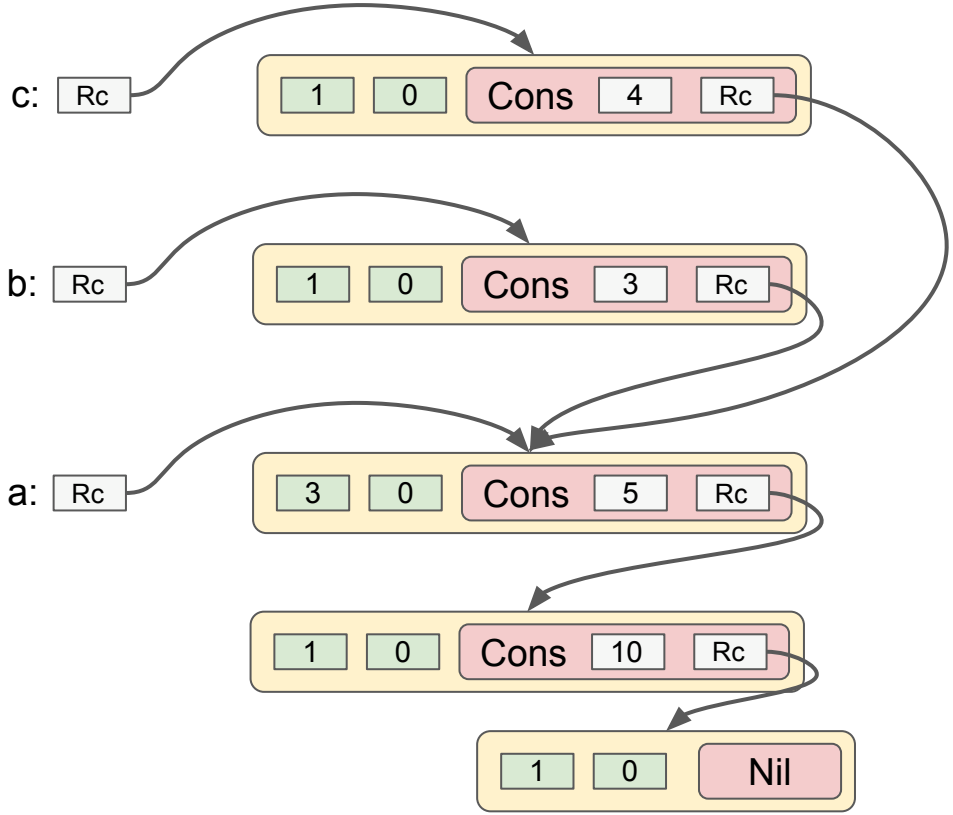
- Nelle situazioni in cui occorre disporre di più possessori di uno stesso dato immutabile, è possibile usare questo smart pointer
  - Internamente mantiene una copia del dato e due contatori: il primo indica quante copie del puntatore esistono, il secondo quanti riferimenti deboli sono presenti
  - Ogni volta che questo puntatore viene clonato, il primo contatore viene incrementato
  - Quando il puntatore esce dal proprio scope, il contatore viene decrementato: se il risultato è 0, il blocco viene rilasciato
- Rc<T> si presta a realizzare alberi e grafi aciclici
  - Non può essere usato da più di un thread



std::rc::Rc<T>

```
enum List {
    Cons(i32, Rc<List>),
    Nil,
}
use crate::List::{Cons, Nil};
use std::rc::Rc;

fn main() {
    let a = Rc::new(
       Cons(5,
         Rc::new(
           Cons(10, Rc::new(Nil)))));
    let b = Rc:new(
               Cons(3, Rc::clone(&a));
    let c = Rc:new(
               Cons(4, Rc::clone(&a));
}
```



std::rc::Rc<T>

- Per evitare problemi di omonimia con i metodi contenuti nel dato incapsulato, tutti i metodi di Rc sono dichiarati con la sintassi
  - `pub fn strong_count(this: &Rc<T>) -> usize`
  - Chiamando **`this`** (e non **`self`**) il parametro che indica l'istanza, non è possibile utilizzare la notazione puntata per invocare i metodi, ma occorre richiamarli nella forma estesa `Rc::<T>::strong_count(&a)`
- Per motivi di efficienza, l'operazione di incremento e decremento sui campi privati strong_count e weak_count non è *thread-safe*
  - Per questo motivo, non è possibile utilizzare questo smart pointer in un contesto concorrente
  - Esiste un'altra classe, trattata in seguito, che supera questo limite: **std::sync::Arc<T>**

std::rc::Weak<T>

- Se si costruisse, usando **`Rc<T>`**, una sequenza circolare di puntatori, la memoria allocata non potrebbe più essere rilasciata
  - Come nel caso di **`shared_ptr`** in C++, la catena dei puntatori terrebbe in vita tutti i blocchi, garantendo che il conteggio dei riferimenti valga almeno 1
- E' possibile creare una struttura con dipendenze circolari utilizzando il tipo **`Weak<T>`**
  - Esso è una versione di **`Rc`** che contiene un riferimento senza possesso al blocco allocato
- Si crea un valore di tipo **`Weak<T>`** a partire da un valore di tipo **`Rc<T>`** con il metodo **`Rc::downgrade(&rc)`**
  - Se il valore originale è ancora in vita (ovvero se **`strong_counter()`** > 0), è possibile costruire un nuovo valore di tipo **`Rc<T>`** invocando il metodo **`upgrade()`**
  - Esso ritorna un valore di tipo **`Option<Rc<T>>`**

std::rc::Weak<T>

```
use std::rc::Rc;

let five = Rc::new(5);

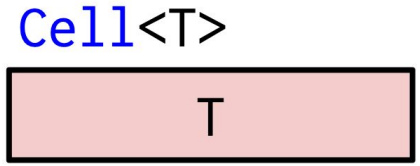
let weak_five = Rc::downgrade(&five);

let strong_five: Option<Rc<_>> = weak_five.upgrade();
assert!(strong_five.is_some());

// Destroy all strong pointers.
drop(strong_five);
drop(five);

assert!(weak_five.upgrade().is_none());
```

std::cell::Cell<T>



- Il *borrow checker* garantisce, in fase di compilazione, che dato un valore di tipo T in ogni momento valgano i seguenti invarianti, mutuamente esclusivi
  - Non esista alcun riferimento al valore al di là del suo possessore
  - Esistano uno o più riferimenti immutabili (`&T`) - aliasing
  - Esista un solo riferimento mutabile (`&mut T`) - mutabilità
- Esistono situazioni in cui l'analisi statica eseguita in fase compilazione è troppo restrittiva
  - Il modulo std::cell offre alcuni contenitori che consentono una mutabilità condivisa e controllata
  - E' possibile cioè avere più riferimenti al valore pur essendo in grado di mutarlo
  - I tipi offerti possono funzionare solo in contesti **non concorrenti** (basati su singolo thread)
- La struct **`std::cell::Cell<T>`** implementa la mutabilità del dato contenuto al suo interno attraverso metodi non richiedono la mutabilità del contenitore
  - Si dice che Cell implementa un meccanismo di *interior mutability*

std::cell::Cell<T>

```
use std::cell::Cell;
struct SomeStruct {
    a: u8,
    b: Cell<u8>,
}

let my_struct = SomeStruct {
    a: 0,
    b: Cell::new(1),
};
// my_struct.a = 100;
// ERRORE: `my_struct` è immutabile

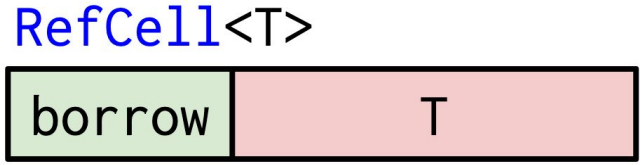
my_struct.b.set(100);
// OK: anche se `my_struct` è immutabile, `b` è una Cell e può essere modificata

assert_eq!(my_struct.b.get(), 100);
```

std::cell::Cell<T>

- Il metodo `get(&self) -> T` che restituisce il dato contenuto al suo interno
  - A condizione che T implementi il tratto Copy
- Il metodo `take(&self) -> T` restituisce il valore contenuto, sostituendolo con il risultato dell'invocazione di `Default::default()`
  - A condizione che T implementi il tratto Default
- Il metodo `replace(&self, val:T) -> T` sostituisce il valore contenuto nella cella con quello passato come parametro e lo restituisce come risultato
  - Questo metodo non pone restrizioni sul tipo di dato
- Il metodo `into_inner(&self) -> T` consuma la cella e restituisce il valore contenuto
  - Anche in questo caso, può essere usato con ogni tipo di dato

std::cell::RefCell<T>



- `Cell<T>` non consente di creare riferimenti al dato contenuto al suo interno
  - Ma solo di inserire, estrarre o sostituire il valore
- La struct `std::cell::RefCell<T>` rappresenta un blocco di memoria a cui è possibile accedere attraverso particolari smart pointer che simulano il comportamento di riferimenti condivisi e mutabili
  - Ma la cui compatibilità con le regole del borrow checker è stabilita in fase di esecuzione e non di compilazione
  - Eventuali tentativi di violazione delle regole generano una condizione di panic, comportando la terminazione del thread corrente
- Il metodo `borrow(&self) -> Ref<'_, T>` restituisce uno smart pointer che implementa il tratto `Deref<T>`
  - Oppure provoca un panic se è già presente un riferimento mutabile
- Il metodo `borrow_mut(&self) -> RefMut<'_, T>` restituisce uno smart pointer che implementa il tratto `DerefMut<T>`
  - Oppure provoca un panic se è già presente un riferimento semplice

std::cell::RefCell<T>

```
use std::cell::RefCell;

let c = RefCell::new(5);

{
    let m = c.borrow_mut();
    assert!(c.try_borrow().is_err());
    *m = 6;
}

{
    let m = c.borrow();
    assert!(c.try_borrow().is_ok());
    assert!(*m == 6);
}
```

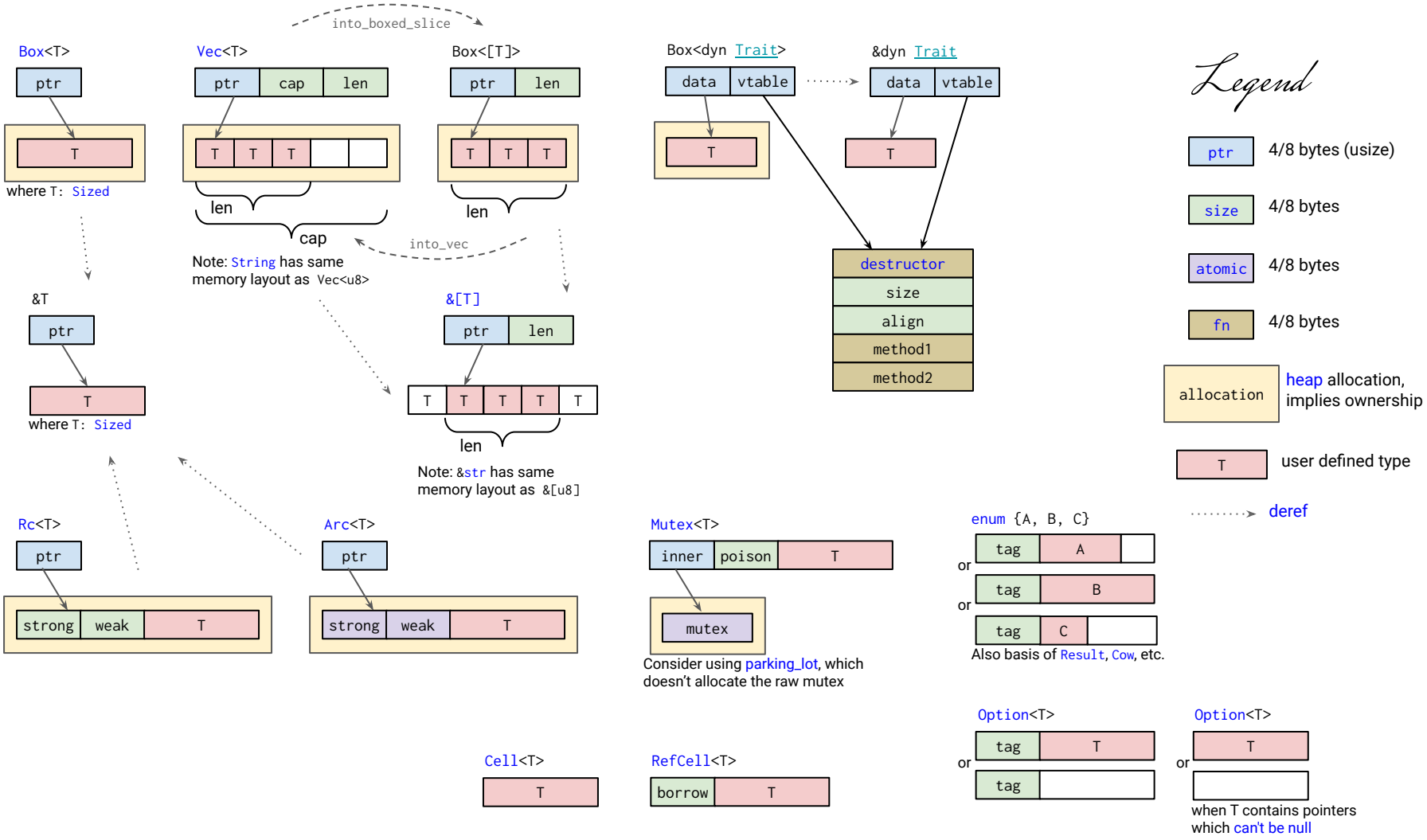
std::borrow::Cow<’a, B>

- Smart pointer che implementa il meccanismo clone on write
  - Se ci cerca di modificare il dato contenuto, e questo è condiviso, il dato viene clonato: si prende possesso della copia e si effettua la modifica, lasciando l’originale invariato
  - Se il dato che si vuole modificare era già posseduto, non avviene nessuna clonazione e si opera la modifica direttamente
- Implementato sotto forma di enumerazione
  - ```
    pub enum Cow<'a, B>
    where B: 'a + ToOwned + ?Sized,
     {
        Borrowed(&'a B),
        Owned(<B as ToOwned>::Owned),
    }
    ```
- Si istanzia attraverso il metodo Cow::from(...)
  - Il compilatore sceglie, in base al tipo di dato fornito, se collocare il valore nella variante Owned o Borrowed
    ```

Smart pointer e metodi

- L'argomento **`self`** di un metodo può anche avere come tipo **`Box<Self>`**, **`Rc<Self>`**, o **`Arc<Self>`**
  - In tal caso, il metodo può essere solo invocato a partire dal corrispondente tipo di puntatore
  - L'invocazione del metodo passa la proprietà del puntatore al metodo stesso
- A differenza di quanto accade con i riferimenti, non è disponibile una forma abbreviata per la sintassi di self
  - Il cui tipo deve essere dichiarato in modo esplicito, come nel caso dei parametri ordinari

```
impl Node {
  fn append_to(self: Rc<Self>, parent: &mut Node) {
    parent.children.push(self);
  }
}
```



Per saperne di più…

- Understanding smart pointers in Rust
  - https://blog.logrocket.com/smart-pointers-rust/
- Understanding Rust smart pointers
  - https://medium.com/the-polyglot-programmer/undestanding-rust-smart-pointers-660d59715ab9
- Rust Smart Pointers Tutorial
  - https://www.koderhq.com/tutorial/rust/smart-pointer/
- Smart Pointers in Rust: What, why and how?
  - https://dev.to/rogertorres/smart-pointers-in-rust-what-why-and-how-oma