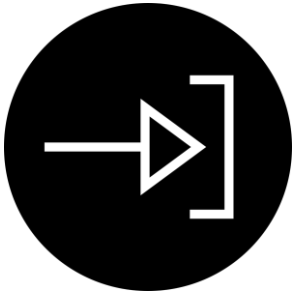


Smart Pointer

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)

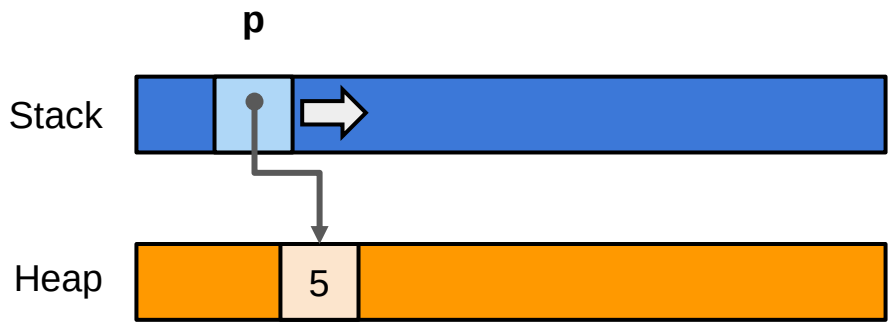
unique_ptr<T>

- 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;

}
```

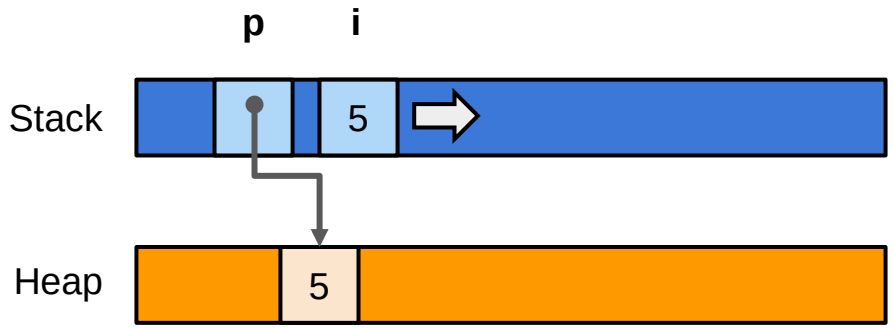
unique_ptr<T>

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

  int i = *p;

  *p = 7;

}
```



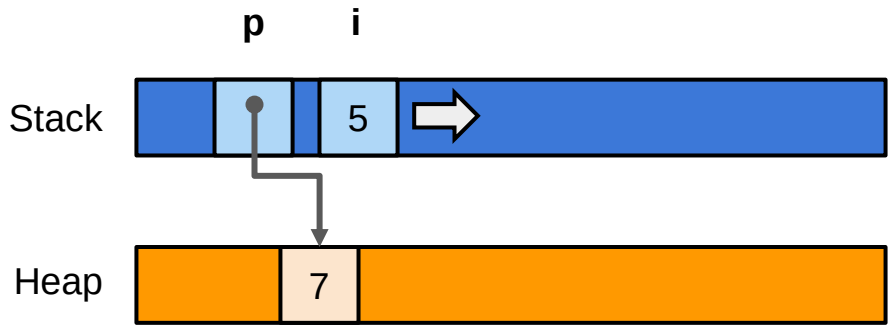
unique_ptr<T>

```
{
  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>

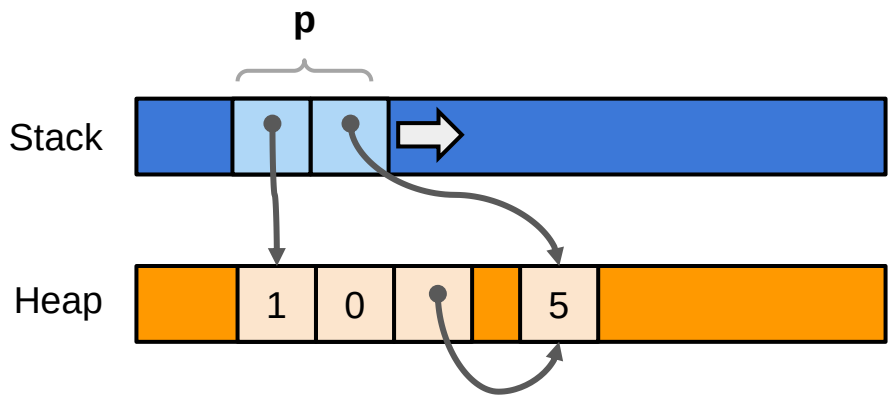
- Mantiene la proprietà condivisa a un blocco di memoria referenziato da un puntatore nativo
  - Molti oggetti possono referenziare lo stesso blocco
  - Quando tutti sono stati distrutti o resettati, il blocco viene rilasciato
- Per default, il blocco referenziato viene rilasciato tramite l'operatore 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

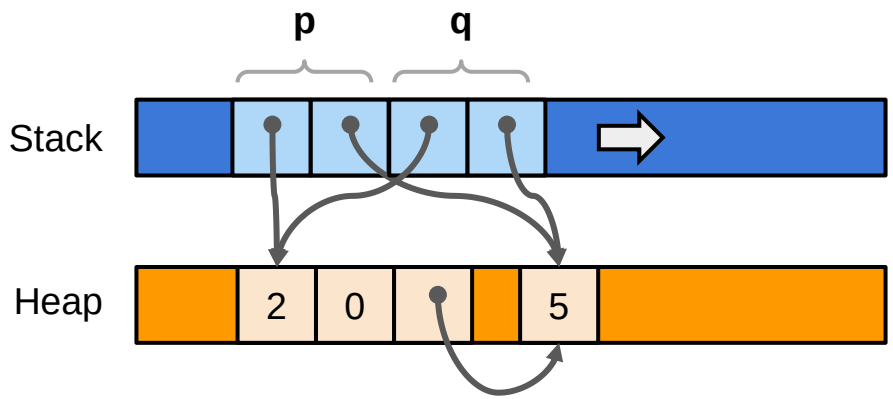
shared_ptr<T>

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

  {
    auto q = p;

    *q = 3;

  }

  *p = 7;

}
```



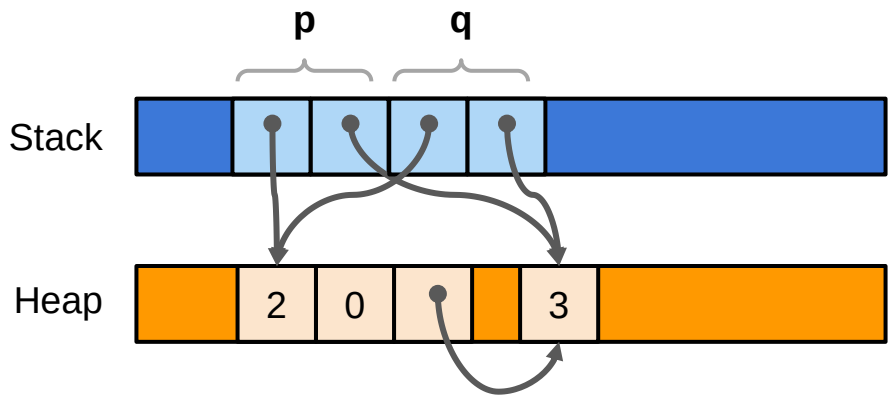
shared_ptr<T>

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

  {
    auto q = p;

    *q = 3;

  }

  *p = 7;

}
```



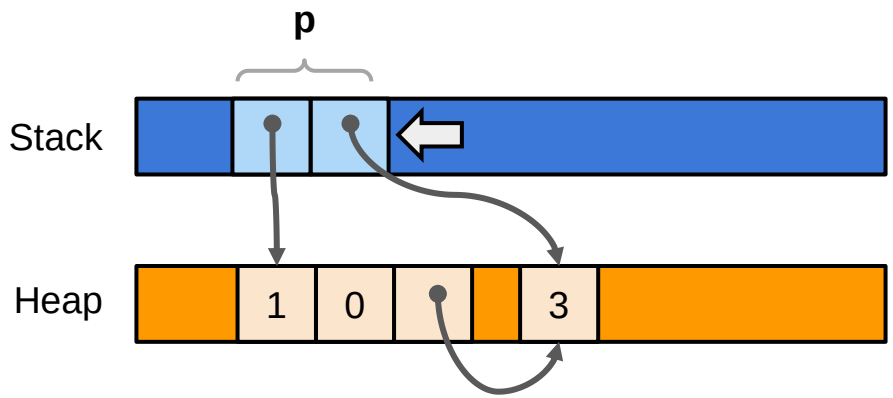
shared_ptr<T>

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

  {
    auto q = p;

    *q = 3;

  }

  *p = 7;

}
```



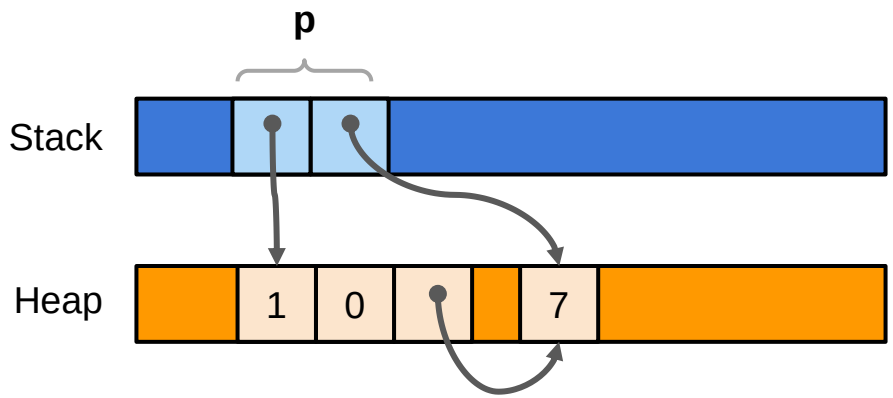
shared_ptr<T>

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

  {
    auto q = p;

    *q = 3;

  }

  *p = 7;

}
```



shared_ptr<T>

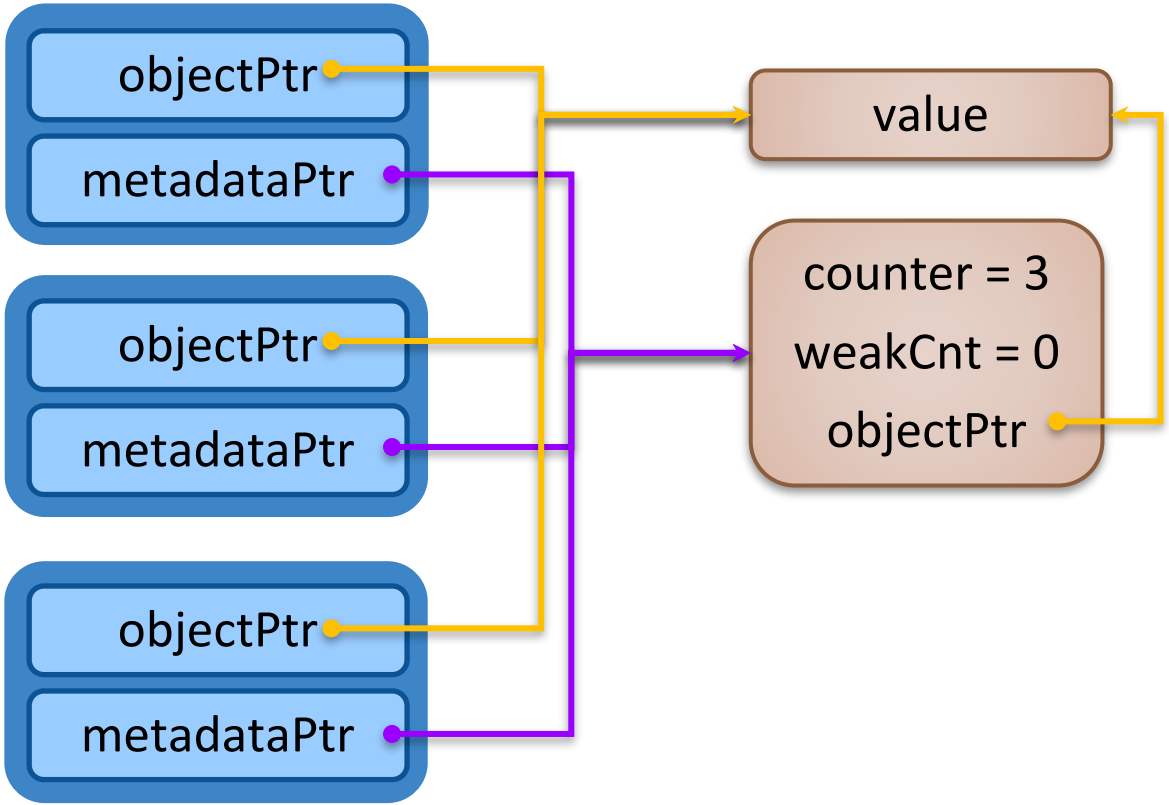
```
{
  std::shared_ptr<int> p =
         std::make_shared<int>(5);

  {
    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

- l 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

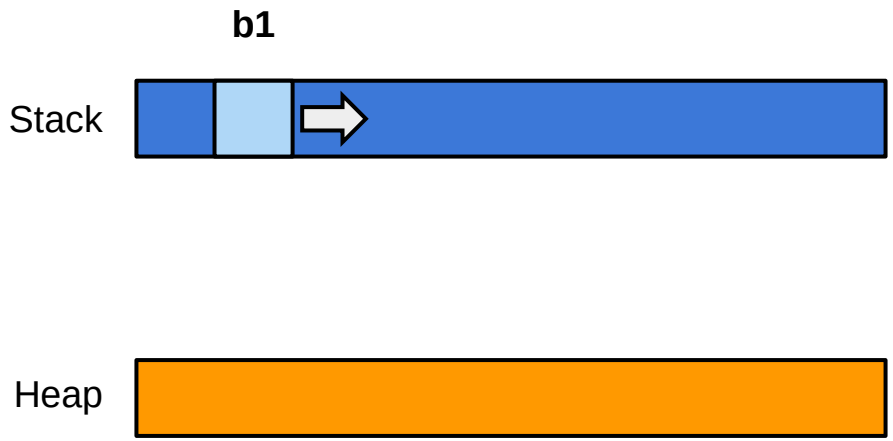
```
trait Deref {
 type Target: ?Sized;
 fn deref(&self) -> &Self::Target;
}
trait DerefMut: Deref {
 fn deref_mut(&mut self) -> &mut Self::Target;
}
```

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

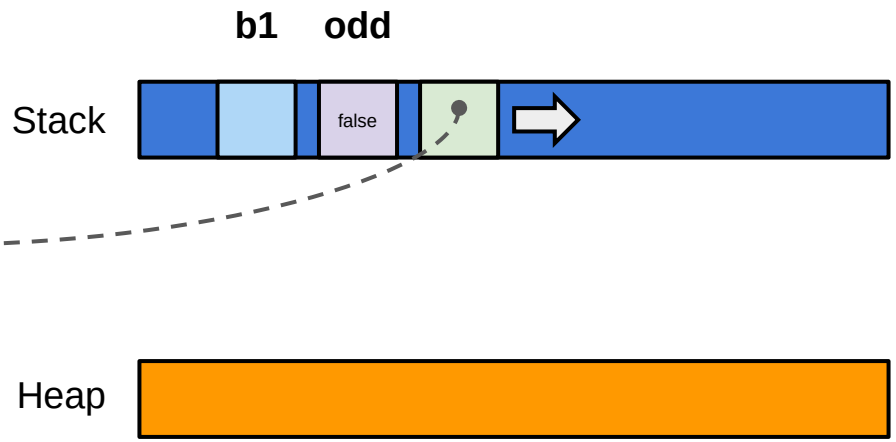
std::Box<T>

```
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);
}
```



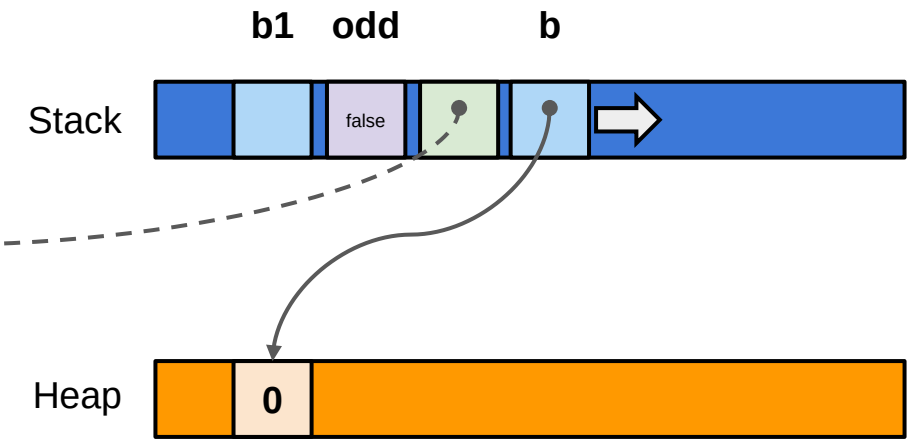
std::Box<T>

```
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);
}
```



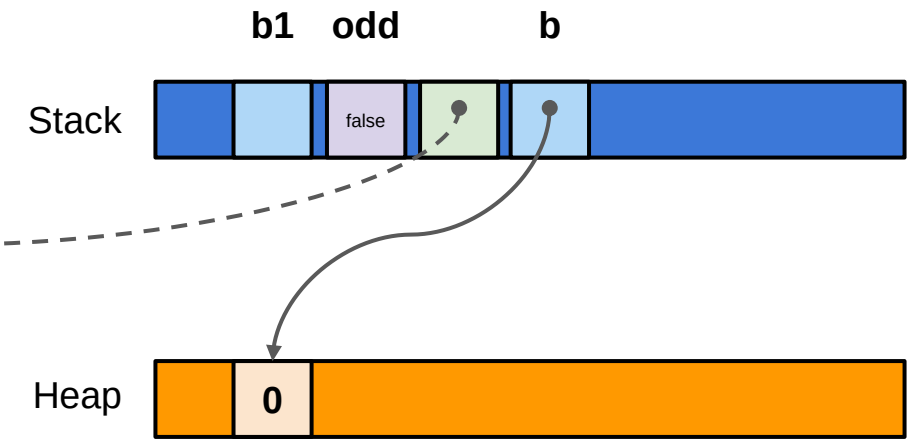
std::Box<T>

```
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);
}
```



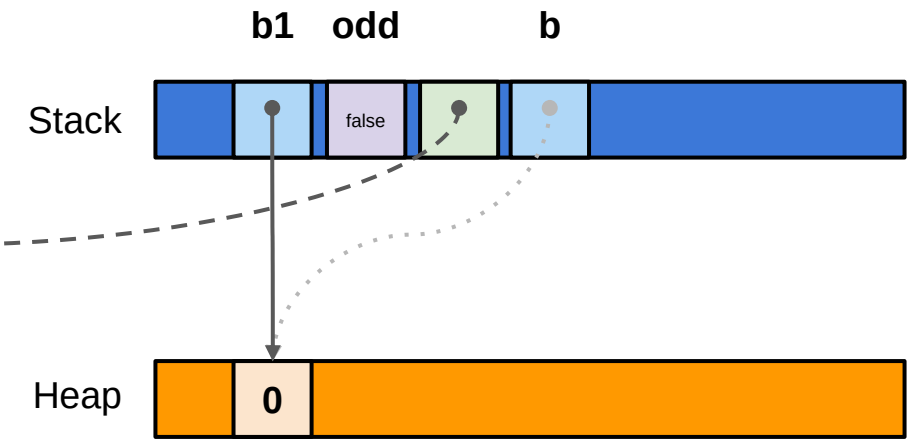
std::Box<T>

```
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);
}
```



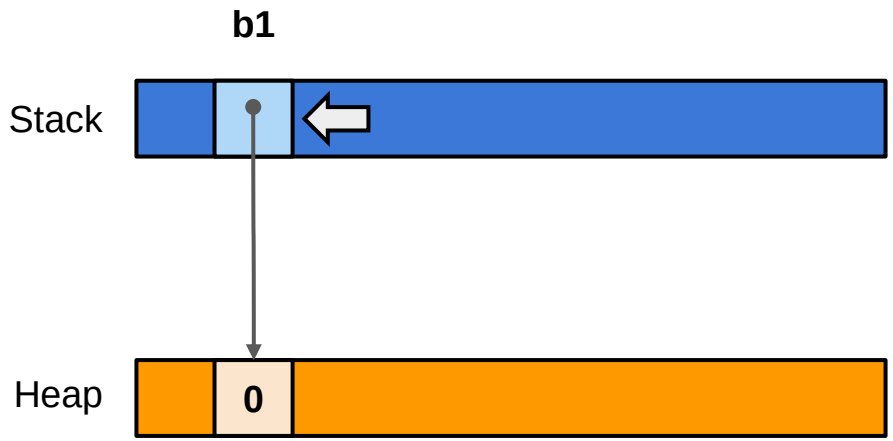
std::Box<T>

```
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);
}
```



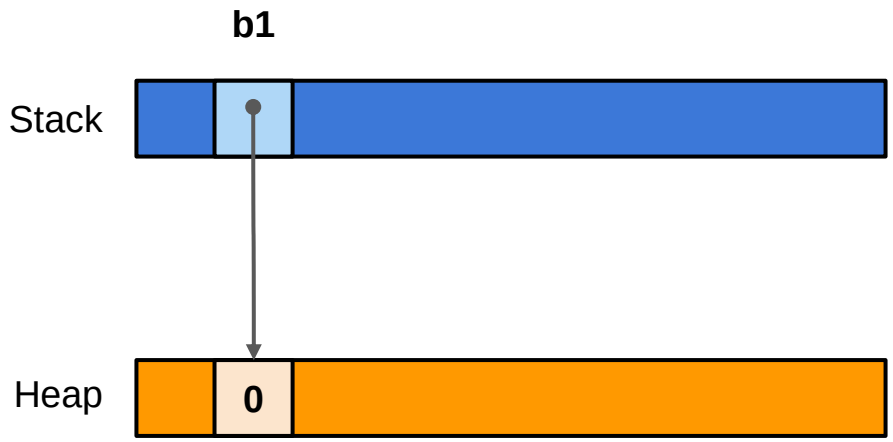
std::Box<T>

```
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);
}
```



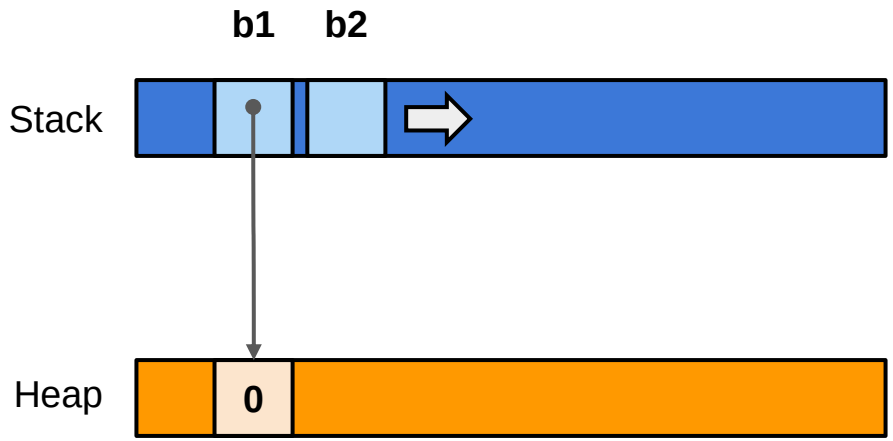
std::Box<T>

```
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);
}
```



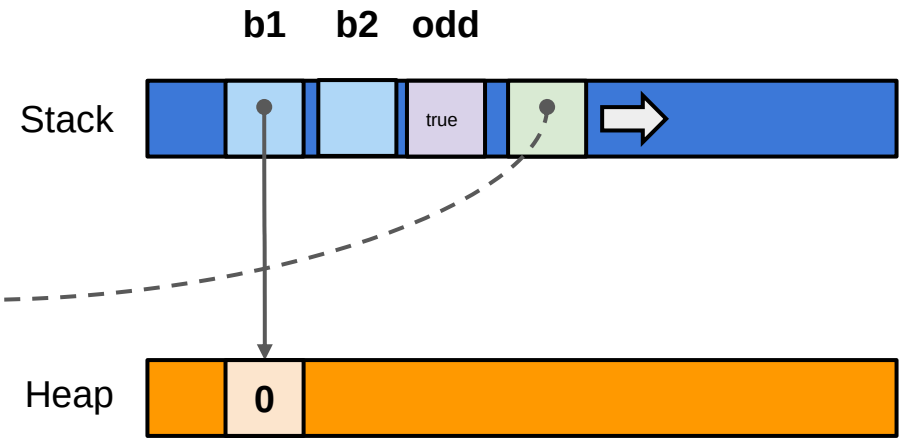
std::Box<T>

```
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);
}
```



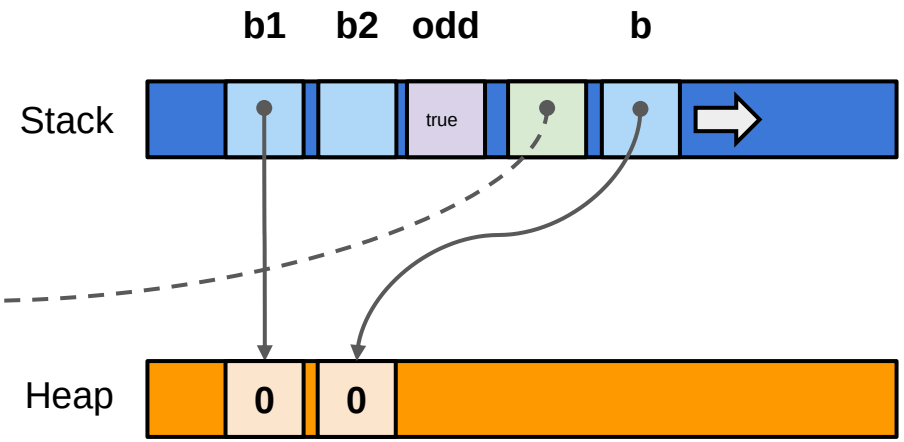
std::Box<T>

```
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);
}
```



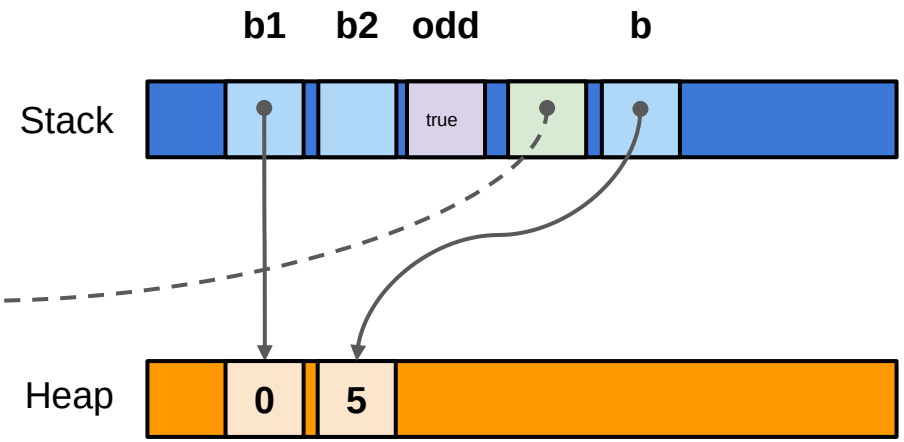
std::Box<T>

```
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);
}
```



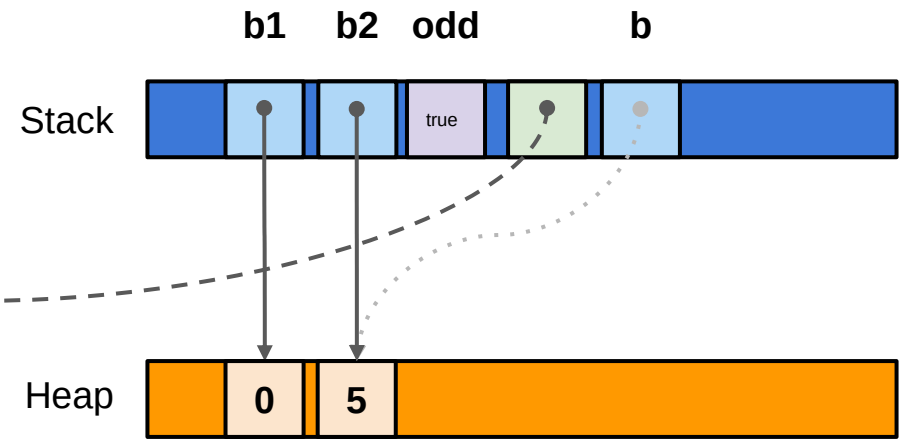
std::Box<T>

```
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);
}
```



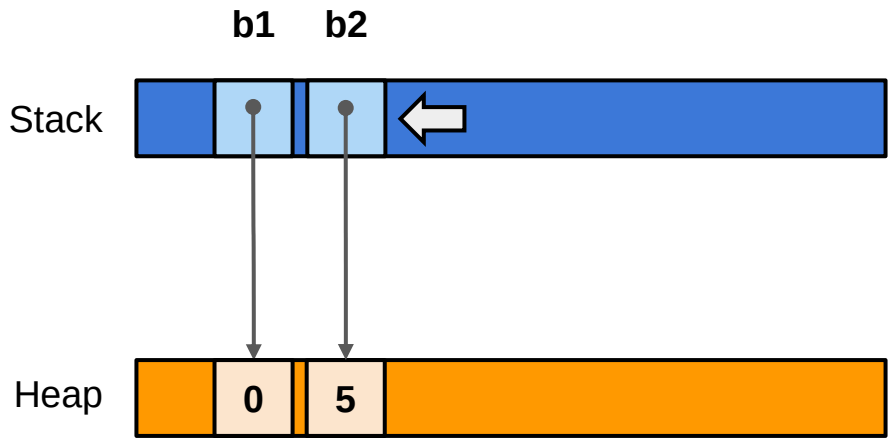
std::Box<T>

```
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);
}
```



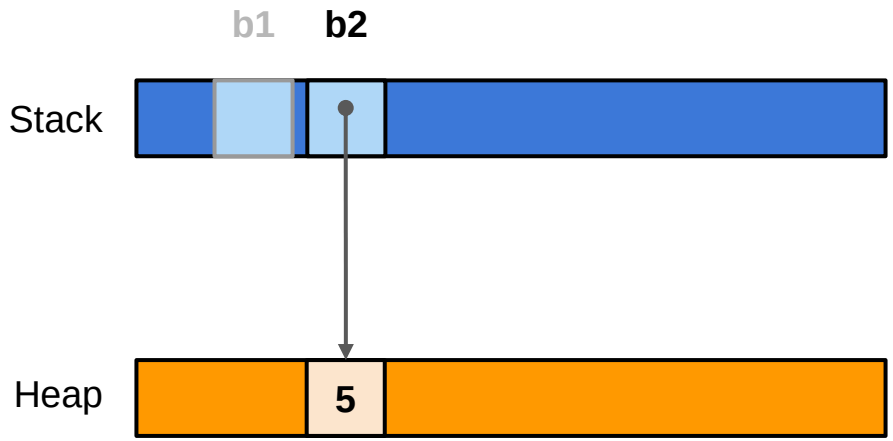
std::Box<T>

```
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);
}
```



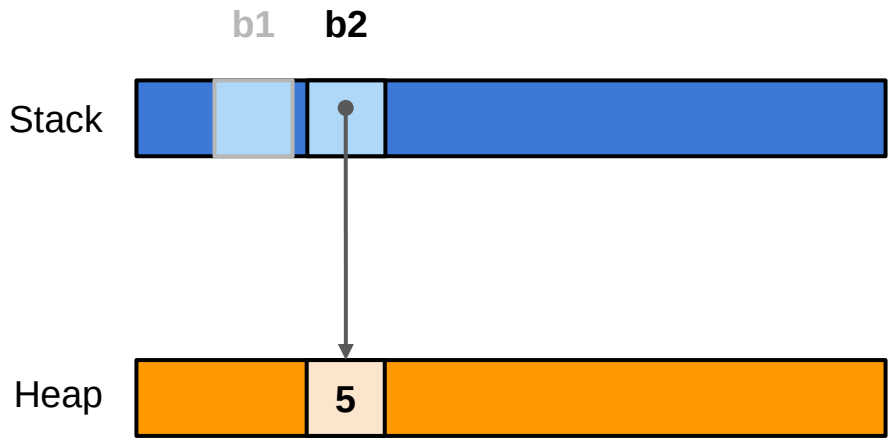
std::Box<T>

```
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);
}
```



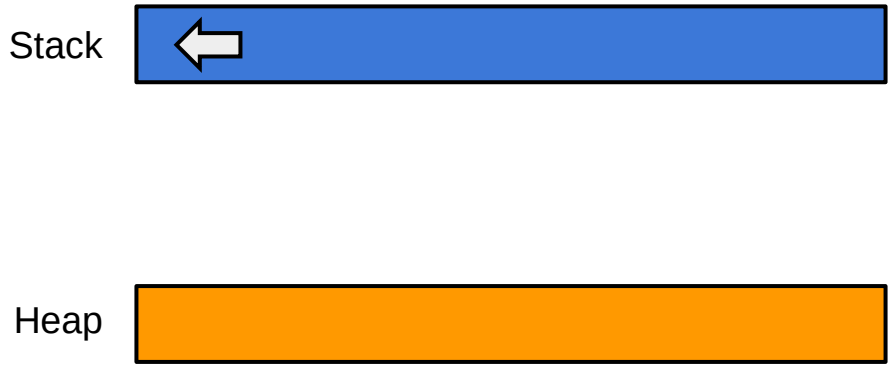
std::Box<T>

```
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);
}
```

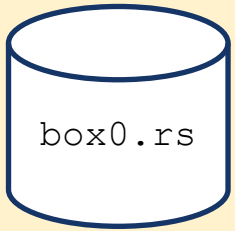


std::Box<T>

```
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);
}
```



```
fn main() {
    // Allocazione di un intero sulla heap con Box.
    let mut numero = Box::new(10);

    println!("Valore iniziale: {}", numero);

    // Modifica del valore all'interno del Box.
    *numero = 20;

    println!("Valore modificato: {}", numero);
}
```



Tipi Ricursivi

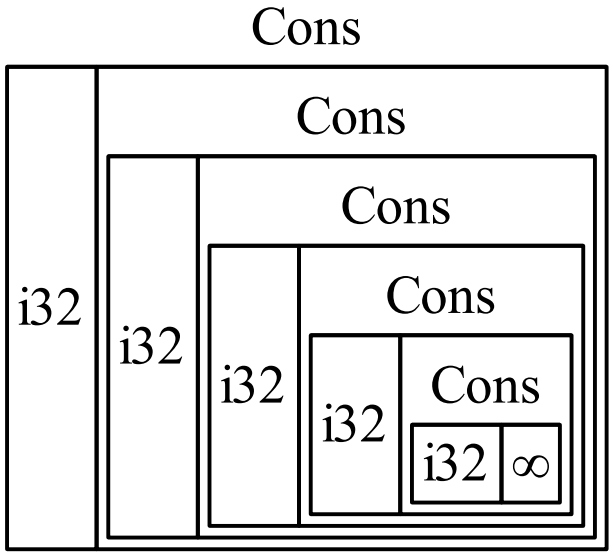
- A titolo di esempio consideriamo la sequente struttura dati chiamata *Cons List (construct list),* struttura dati che deriva dal mondo LISP

```
enum List {
    Cons(i32, List),
    Nil,
}
```

```
(1, (2, (3, Nil)))
```

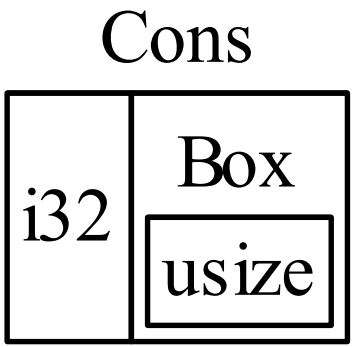
```
enum List {
    Cons(i32, List),
    Nil,
}

fn main() {
    let list = Cons(1, Cons(2, Cons(3, Nil)));
}
```


```
error[E0072]: recursive type `List` has infinite size
 --> src\main.rs:1:1
  |
1 | enum List {
  | ^^^^^^^^^
2 |     Cons(i32, List),
  |               ---- recursive without indirection
```

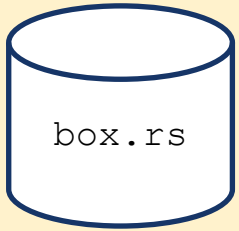


- Utilizzando il Box, il compilatore riesce a definire la dimensione della struttura dati.

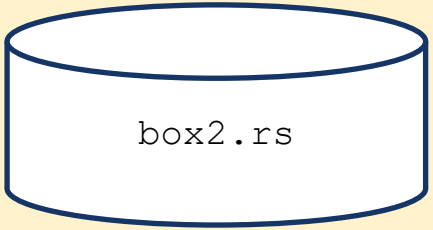
```
enum List {
    Cons(i32, Box<List>),
    Nil,
}
```



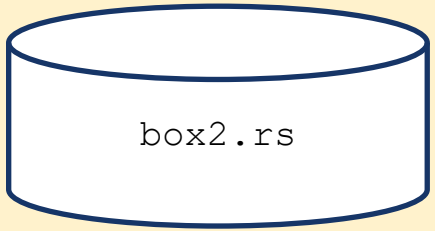
```
use crate::List::{Cons, Nil};

enum List {
    Cons(i32, Box<List>),
    Nil,
}

fn main() {
    let a = Cons(3, Box::new(Nil));
    let b = Cons(2, Box::new(a));
    let c = Cons(1, Box::new(b));
    let head = Cons(0, Box::new(c));

    // equivalente a
    // let list = Cons(0, Cons(1, Cons(2, Cons(3, Nil))));

    let mut current_node = &head;

    while let Cons(value, next) = current_node {
        println!("Value: {}", value);
        current_node = next;
    }
}
```



```rust
// Definizione di una struttura per un albero binario
#[derive(Debug)]
enum BinaryTree {
    Empty,
    Node(i32, Box<BinaryTree>, Box<BinaryTree>),
}
impl BinaryTree {
    // Metodo per creare un nuovo albero binario
    fn new() -> Self { BinaryTree::Empty }
    // Metodo per inserire un valore nell'albero binario
    fn insert(&mut self, value: i32) {
        match *self {
            BinaryTree::Empty => *self = BinaryTree::Node(value,
Box::new(BinaryTree::Empty), Box::new(BinaryTree::Empty)),
            BinaryTree::Node(ref mut data, ref mut left, ref mut right) => {
                if value <= *data {
                    left.insert(value);
                } else {
                    right.insert(value);
                       }
                }
            }
     }
}
```

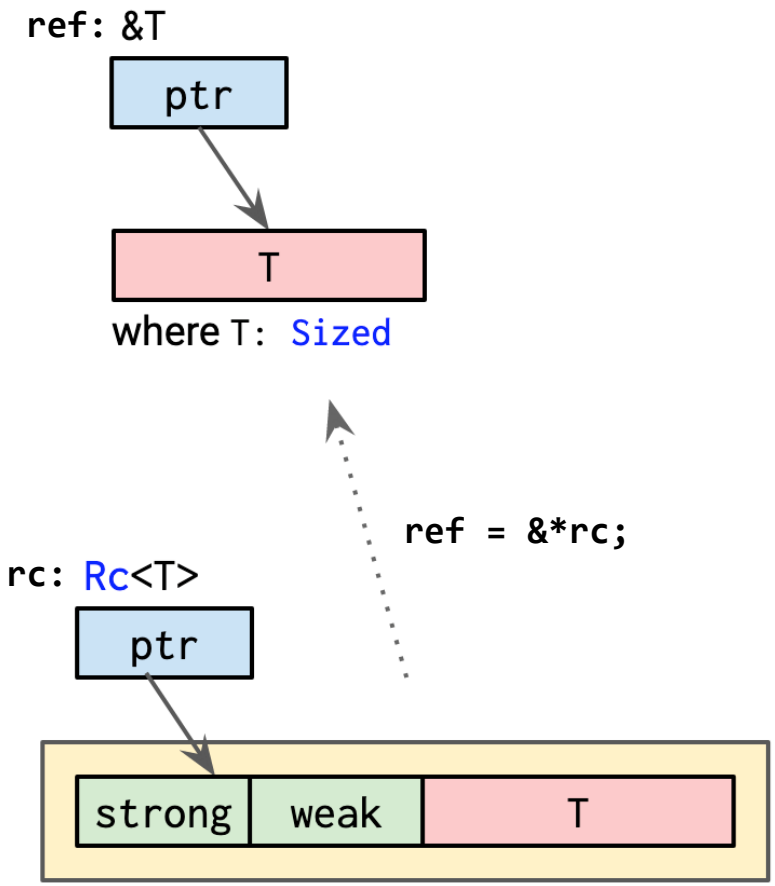


```
fn main() {
    // Creazione di un nuovo albero binario
    let mut tree = BinaryTree::new();

    // Inserimento di alcuni valori nell'albero binario
    tree.insert(5);
    tree.insert(3);
    tree.insert(7);
    tree.insert(1);
    tree.insert(4);
    tree.insert(6);
    tree.insert(8);

    println!("{:?}", tree);
}
```



std::rc::Rc<T>

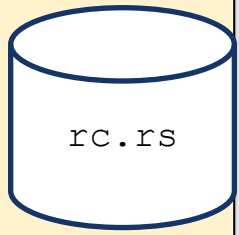
- Nelle situazioni in cui occorre disporre di più possessori di uno stesso dato **immutabile**, è possibile usare questo smart pointer
  - Internamente mantiene una copia del dato e due contatori: il primo indica quante copie del puntatore esistono, il secondo quanti riferimenti deboli sono presenti
  - 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
  - Può essere usato solo in contesti **single-thread**



std::rc::Rc<T>

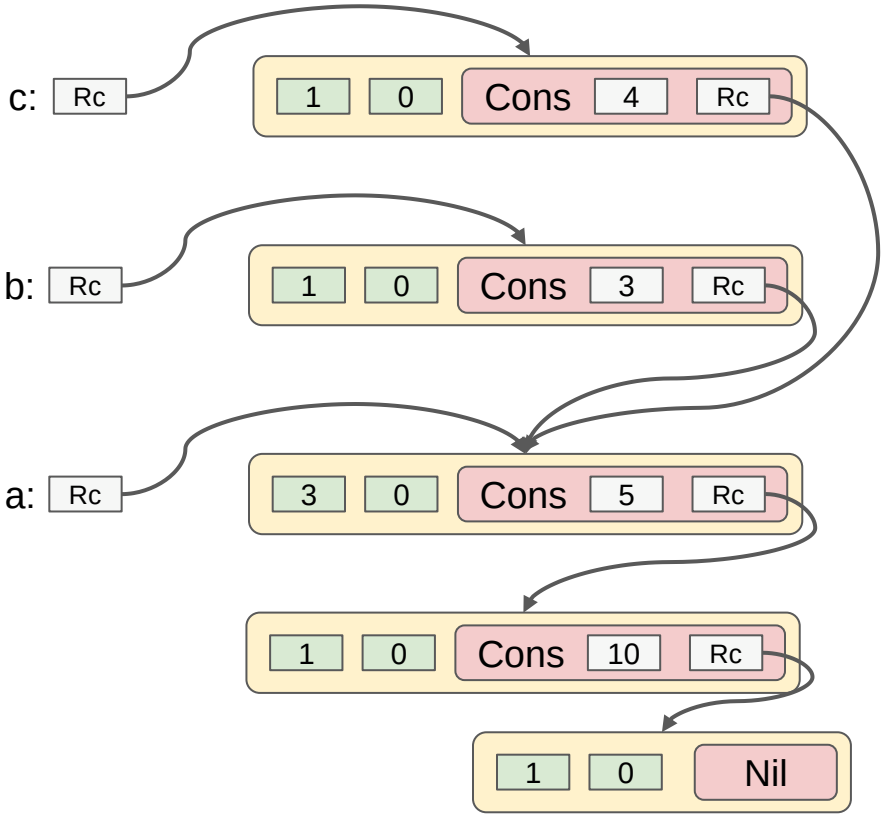
- Per evitare problemi di omonimia con i metodi contenuti nel dato incapsulato, tutti i metodi di Rc sono dichiarati con la sintassi
  - `pub fn strong_count(this: &Rc<T>) -> usize`
  - Chiamando **`this`** (e non **`self`**) il parametro che indica l'istanza, non è possibile utilizzare la notazione 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>**

```rust
use std::rc::Rc;
fn main() {
    // Creiamo un valore Rc contenente una stringa.
    let rc_esempio = Rc::new("Esempio Rc".to_string());
    {   // Cloniamo rc_esempio per creare un nuovo puntatore allo stesso valore.
        let rc_a = Rc::clone(&rc_esempio);
        println!("Conteggio referenze di rc_a: {}", Rc::strong_count(&rc_a));
        {   // Cloniamo ancora per creare un altro puntatore.
            let rc_b = Rc::clone(&rc_a);
            println!("Conteggio referenze di rc_b: {}", Rc::strong_count(&rc_b));
            println!("Conteggio referenze di rc_a: {}", Rc::strong_count(&rc_a));

            // I due Rc sono uguali se i loro valori interni sono uguali.
            println!("rc_a e rc_b sono uguali: {}", rc_a.eq(&rc_b));

            // Possiamo usare direttamente i metodi del valore interno.
            println!("Lunghezza del valore dentro rc_a: {}", rc_a.len());
        }
        // Quando rc_b esce dallo scope, il conteggio delle referenze diminuisce.
        println!("Conteggio referenze di rc_a dopo che rc_b è uscito: {}",
Rc::strong_count(&rc_a));
    }
    println!("Conteggio referenze di rc_esempio dopo che rc_a è uscito: {}",
Rc::strong_count(&rc_esempio));
    // Quando rc_esempio esce dallo scope, rc_esempio e il valore vengono eliminati.
}
```



```
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)));
}
```

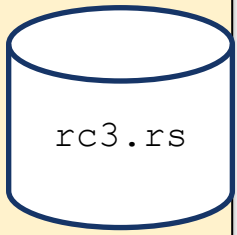
rc2.rs



Mutabilità del dato puntato da Rc

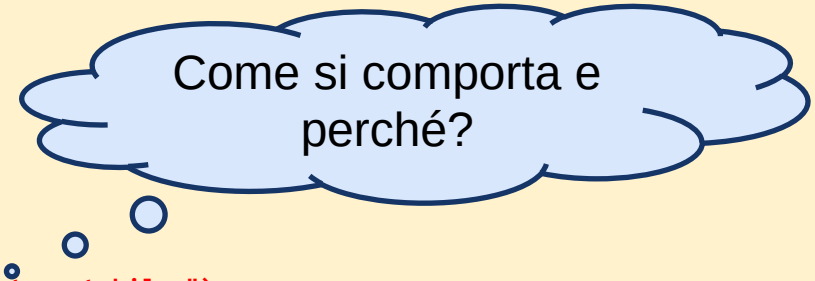
- Il dato puntato da Rc è mutabile solo se non esistono altri riferimenti al dato.
- Il metodo da utilizzare è **`get_mut(this: &Rc<T>) -> Option<& mut T>`**
  - Se non esistono altri puntatori allo stesso dato, get_mut restituirà `Some(& mut T)` permettendo di modificare il valore contenuto
  - Altrimenti restituirà `None`, poiché non è possibile avere più riferimenti mutabili allo stesso valore.

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

fn main() {
    let mut valore = Rc::new(5);
    {
        println!("Valore: {:?}", valore);
        let copia = Rc::clone(&valore);
        match Rc::get_mut(&mut valore) {
            Some(v) => *v += 10, // Modifica il valore se è possibile.
            None => println!("Non è possibile ottenere un riferimento mutabile."),
        }
    }
    // Prova a ottenere un riferimento mutabile.
    match Rc::get_mut(&mut valore) {
        Some(v) => *v += 10, // Modifica il valore se è possibile.
        None => println!("Non è possibile ottenere un riferimento mutabile."),
    }

    println!("Valore: {:?}", valore);
}
```



```rust
use std::rc::Rc;
#[derive(Debug)]
struct Node {
    value: i32,
    children: Vec<Rc<Node>>,
}
fn main() {
    let nipote1 = Rc::new(Node {
        value: 3,
        children: vec![],
    });
    let nipote2 = Rc::new(Node {
        value: 6,
        children: vec![],
    });

    let padre = Rc::new(Node {
        value: 9,
        children: vec![Rc::clone(&nipote1), Rc::clone(&nipote2)],
    });

    let nonno = Rc::new(Node {
        value: 27,
        children: vec![Rc::clone(&padre)],
    });
    println!("{:#?}", nonno);

}
```

```rust
use std::rc::Rc;
#[derive(Debug)]
struct Node {
    value: i32,
    children: Vec<Rc<Node>>,
}
fn main() {
    let mut nipote1 = Rc::new(Node {
        value: 3,
        children: vec![],
    });
    let nipote2 = Rc::new(Node {
        value: 6,
        children: vec![],
    });

    let padre = Rc::new(Node {
        value: 9,
        children: vec![Rc::clone(&nipote1), Rc::clone(&nipote2)],
    });

    let nonno = Rc::new(Node {
        value: 27,
        children: vec![Rc::clone(&padre)],
    });

    match Rc::get_mut(&mut nipote1) {
        Some(v) => v.children.push(Rc::clone(&nonno)),
        None => println!("Non è possibile ottenere un riferimento mutabile."),
    }

    println!("{:#?}", nonno);

}
```



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
  - Se ciò a cui punto è ancora vivo ci arrivo,
  - se ciò a cui punto non è più vivo, non ci arrivo
  - dunque **non partecipo al processo di tenuta in vita**, ma se l'oggetto è vivo ci posso arrivare.
- 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>>`**

- Un puntatore `Weak` può essere generato solo a partire da un `Rc`, attraverso il metodo `downgrade`
- Un `Weak` non è direttamente dereferenziabile. Se faccio `*w` di un oggetto `Weak`, non ottengo l'accesso al campo. Per poter accedere al dato a cui quel Weak punta devo fare `upgrade`
  - Se il puntatore `Rc` è ancora vivo, il `Weak` si tramutata in un `Rc`, in modo da poterlo possedere, almeno temporaneamente. Quando si finisce, si può fare il `drop` di `Rc`, che sparisce, ma rimane la variabile di tipo `Weak` che può eventualmente essere ricondotta ad un `Rc`.

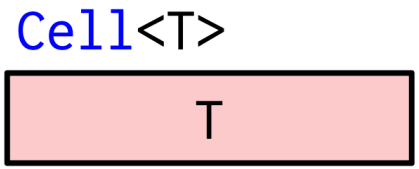
```
use std::rc::Rc;
fn main() {
    let five = Rc::new(5);
    let ten = Rc::clone(&five);

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

    println!("Conteggio referenze strong di five: {}", Rc::strong_count(&five));
    println!("Conteggio referenze weak di five: {}", Rc::weak_count(&five));

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

    println!("Conteggio referenze strong di five: {}", Rc::strong_count(&five));
    println!("Conteggio referenze weak di five: {}", Rc::weak_count(&five));
    drop(strong_five);
    println!("Conteggio referenze strong di five: {}", Rc::strong_count(&five));
    println!("Conteggio referenze weak di five: {}", Rc::weak_count(&five));
    drop(ten);
    drop(weak_five);
    println!("Conteggio referenze strong di five: {}", Rc::strong_count(&five));
    println!("Conteggio referenze weak di five: {}", Rc::weak_count(&five));
}
```

```rust
use std::rc::Rc;
fn main() {
    let five = Rc::new(5);

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

    println!("{:?}", five);
    drop(five);

    let strong_five: Option<Rc<_>> = weak_five.upgrade();
    println!("{:?}", strong_five);
}
```

std::cell

- 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)

std::cell::Cell<T>



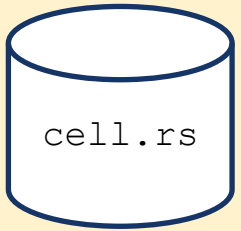
- La struct **`std::cell::Cell<T>`** implementa la mutabilità del dato contenuto al suo interno (**memorizzato nello stack**) attraverso metodi che non richiedono la mutabilità del contenitore
- Si dice che Cell implementa un meccanismo di *interior mutability (*nell'ambito di una *exterior immutability*): un dato conosciuto con una reference può essere modificato (facendo una operazione di swap)
- Le Cell permettono di bypassare le regole di analisi statica fatta dal borrow checker del compilatore:
  - in un programma single thread, si può creare una variabile di tipo Cell, conosciuta come reference semplice, ma sostituire il dato X con il dato Y
  - se un dato di tipo Cell, gli altri sapendo che quella è una Cell che può cambiare improvvisamente, devono leggerla e non devono farsi delle copie per i fatti loro.

std::cell::Cell<T>

- Una istanza di `Cell<T>` viene creata con il metodo `new(value: T) -> Cell<T>`
- Il metodo `set(&self, val: T)` imposta il valore contenuto nella cella con il valore passato come parametro
- 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::Cell<T>

```
use std::cell::Cell;
#[derive(Debug)]
struct SomeStruct {
    a: u8,
    b: Cell<u8>,
}
fn main() {

    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
    println!("{:?}", my_struct);

}
```



```rust
use std::cell::Cell;
#[derive(Debug)]
struct SomeStruct {
    a: u8,       // il campo a è soggetto a tutti i vincoli del borrow checker

    b: Cell<u8>, // il campo b è soggetto ad un rilassamento di questi vincoli
}
fn main() {

    let my_struct = SomeStruct {
        a: 0,
        b: Cell::new(1),
    };


    let copy_struct = &my_struct;
    my_struct.b.set(100);

    // OK: anche se esiste un Reference, `b` è una Cell e può essere modificata

    println!("{:?}", my_struct);
    println!("{:?}", copy_struct);
}
```

```
use std::cell::Cell;
struct Contatore {
    conteggio: Cell<u32>,
}
impl Contatore {
    fn new() -> Contatore {
        Contatore {
            conteggio: Cell::new(0),
        }
    }
fn incrementa(&self) {
        let conteggio_attuale = self.conteggio.get();
        self.conteggio.set(conteggio_attuale + 1);
    }
fn decrementa(&self) {
        let conteggio_attuale = self.conteggio.get();
        self.conteggio.set(conteggio_attuale - 1);
    }
fn leggi(&self) -> u32 {
        self.conteggio.get()
    }
}
fn main() {
    let contatore = Contatore::new();
    contatore.incrementa();
    contatore.incrementa();
    println!("Conteggio: {}", contatore.leggi());
    contatore.decrementa();
    println!("Conteggio: {}", contatore.leggi());
}
```

```rust
use std::cell::Cell;

fn main() {
    // Creiamo una Cell contenente un valore iniziale.
    let cell = Cell::new(42);

    // Utilizziamo il metodo take per estrarre il valore dalla Cell.
    let taken_value = cell.take();
    println!("Valore estratto dalla Cell: {}", taken_value);

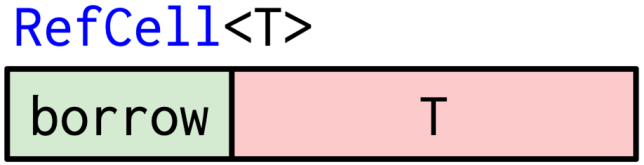
    // Utilizziamo il metodo replace per sostituire il valore all'interno della Cell
    // e ottenere quello precedente.

    let previous_value = cell.replace(99);
    println!("Valore precedente nella Cell: {}", previous_value);

    // Utilizziamo il metodo into_inner per estrarre il valore contenuto nella Cell.
    let inner_value = cell.into_inner();
    println!("Valore estratto dalla Cell con into_inner: {}", inner_value);

    // la cella è stata mossa e dunque non è più accessibile
    // let taken_value = cell.take();
}
```

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 (in fase di run time) generano una condizione di panic, comportando la terminazione del thread corrente
- Oltre a incapsulare il dato, ha un campo che dice se ce l'ha qualcuno in lettura o scrittura
  - Se non ce l'ha nessuno, posso accedere al dato
  - Se ce l'ha qualcuno in lettura e
    - Viene chiesto in lettura, si segna che siamo in n+1 ad avercelo in lettura
    - Viene chiesto in scrittura, da un panic
  - Se ce l'ha qualcuno in scrittura, e viene chiesto in lettura o scrittura, da un panic.

std::cell::RefCell<T>

- `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
- `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
- `try_borrow(&self) -> Result<Ref<T>,BorrowError>`: restituisce un risultato contenente un riferimento immutabile al valore, se possibile, senza panicare.
- `try_borrow_mut(&self) -> Result<RefMut<T>,BorrowMutError>`: restituisce un risultato contenente un riferimento mutabile al valore, se possibile, senza panicare.
- `into_inner(self) -> T`: consuma la RefCell e restituisce il valore contenuto al suo interno.
- `get_mut(&mut self) -> &mut T`: restituisce un riferimento mutabile al valore contenuto nella RefCell se non ci sono altri riferimenti attivi.

```
use std::cell::RefCell;
fn main() {
    let mut c = RefCell::new(5);
    println!("Il dato è {:?}", c);
    {
        *c.get_mut() += 5;

        println!("Il dato aggiornato è {:?}", c);

        let mut m = c.borrow_mut();

        if (c.try_borrow().is_err())
        {
            println!("Non posso fare un altro prestito"); // Si entra qua
        }
        *m = 6;
        println!("Il dato è {:?}", m);
    }
    if (c.try_borrow().is_ok())
    {
        println!("Posso fare un altro prestito");
        let m = c.borrow();
        println!("Il dato è {:?}", m);
        if (c.try_borrow_mut().is_ok())
        {
            println!("Posso fare un altro prestito"); // non si entra qua
            let mut m2 = c.borrow_mut();
        }
        drop(m);
        let val = c.into_inner();
        println!("Valore Finale dopo la distruzione del RefCell{:?}", val);

    }
}
```

Combinazione di RefCell e Rc

- Un Rc<T> che tiene un RefCell<T> corrisponde ad un dato che può avere molti proprietari e che può essere mutato.

```
#[derive(Debug)]
enum List {
    Cons(Rc<RefCell<i32>>, Rc<List>),
    Nil,
}

use List::{Cons, Nil};
use std::rc::Rc;
use std::cell::RefCell;

fn main() {
    let value = Rc::new(RefCell::new(5));

    let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));

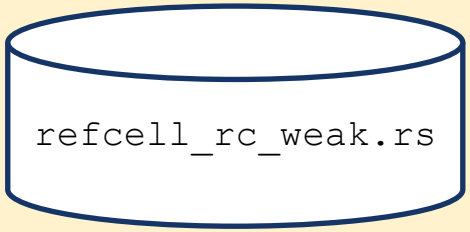
    let b = Cons(Rc::new(RefCell::new(6)), Rc::clone(&a));
    let c = Cons(Rc::new(RefCell::new(10)), Rc::clone(&a));

    println!("a before = {:?}", a);
    println!("b before = {:?}", b);
    println!("c before = {:?}", c);

    *value.borrow_mut() += 10;

    println!("a after = {:?}", a);
    println!("b after = {:?}", b);
    println!("c after = {:?}", c);
}
```

Grafo ciclico




```
use std::rc::{Rc,Weak};
use std::cell::RefCell;
#[derive(Debug)]
struct Node {
    value: i32,
    parent: RefCell<Weak<Node>>,
    children: RefCell<Vec<Rc<Node>>>,
}
fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        children: RefCell::new(vec![]),
        parent: RefCell::new(Weak::new()), }); // even the Weak pointer has an associated new() fn
    println!("leaf: strong = {} weak = {}", Rc::strong_count(&leaf), Rc::weak_count(&leaf) );
    { // create a new scope within our main function
        let root = Rc::new(Node{ // create a branch where the previously created leaf is our child
            value:5,
            children: RefCell::new(vec![Rc::clone(&leaf)]),
            parent: RefCell::new(Weak::new()), });
        // 1. borrow a mutable reference of the leaf's parent
        // 2. dereference it
        // 3. set it equal to a weak reference to the root node
        *leaf.parent.borrow_mut() = Rc::downgrade(&root);

        println!("root: strong = {} weak = {}", Rc::strong_count(&root), Rc::weak_count(&root));
        println!("leaf: strong = {} weak = {}", Rc::strong_count(&leaf), Rc::weak_count(&leaf));
        println!("leaf parent = {:#?}", leaf.parent.borrow().upgrade());
    } // parent goes out of scope even though the child has a reference to it
    // the parent now equals None! it's gone out of scope even though we had a reference to it!
    println!("leaf parent = {:#?}", leaf.parent.borrow().upgrade());
    println!("leaf: strong = {} weak = {}", Rc::strong_count(&leaf), Rc::weak_count(&leaf));
}
```

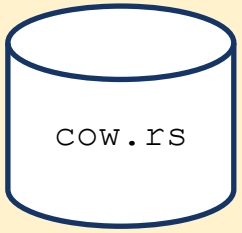
std::borrow::Cow<'a, B>

- Smart pointer che implementa il meccanismo **clone on write**
  - Se si 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),   // Questo indica che il valore è preso in prestito
                        // e non può essere modificato
     Owned(<B as ToOwned>::Owned), // Questo indica che il valore è di proprietà
                                   // e può essere modificato.
    }
    ```
    ```

std::borrow::Cow<’a, B>

- Il metodo **`from()`** crea un nuovo Cow da un riferimento a dati presi in prestito; Il compilatore sceglie, in base al tipo di dato fornito, se collocare il valore nella variante Owned o Borrowed
- Il metodo **`to_mut()`** restituisce un riferimento mutabile ai dati.
- Il metodo **`into_owned()`** restituisce i dati come proprietà
- Il metodo **`as_ref()`** restituisce un riferimento ai dati presi in prestito
- Il metodo **`as_mut()`** restituisce un riferimento mutabile ai dati.

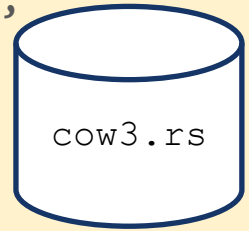
```rust
use std::borrow::Cow;
fn abs_all<'a>(input: &'a mut Cow<'a, [i32]>)  -> Cow<'a, [i32]>{
    // Check if the input slice contains any negative values
    if input.iter().any(|&x| x < 0) {
        // If yes, create a new vector with the absolute values
        let mut output = Vec::new();
        for &x in input.iter(){
            output.push(x.abs());
        }
        // Return the vector as an owned slice
        print!("Owned => ");
        Cow::Owned(output)
    } else {
        // If no, return the input slice as a borrowed slice
        print!("Borrowed => ");
        Cow::Borrowed(input)
    }
}
fn main() {
    let slice = [0, 1, 2];
    let mut input = Cow::from(&slice[..]);
    let mut output = abs_all(&mut input); // No clone occurs because input doesn't need to be mutated.
    println!("{:?}", output);

    let slice = [-1, 0, 1];
    let mut input = Cow::from(&slice[..]); // Clone occurs because input needs to be mutated.
    output = abs_all(&mut input);
    println!("{:?}", output);

    let mut input = Cow::from(vec![-1, 0, 1]); // No clone occurs because input is already owned.
    output = abs_all(&mut input);
    println!("{:?}", output);
}
```



```rust
use std::borrow::Cow;

fn modify_if_condition(content: Cow<str>) -> Cow<str> {
    if content.len() < 10 {
        Cow::Owned(content.into_owned().to_lowercase())
    } else {
        content
    }
}

fn main() {
    let long_string = String::from("This is a long string");
    let short_string = String::from("SHORT");

    let modified_long = modify_if_condition(Cow::from(&long_string));
    let modified_short = modify_if_condition(Cow::from(&short_string));

    println!("{}", modified_long); // Stampa "This is a long string"
    println!("{}", modified_short); // Stampa "short"
}
```

Utilizzando il puntatore Cow, evitiamo il clonaggio quando non è necessario modificare i dati

```rust
use std::borrow::Cow;

fn main() {
    let mut borrowed_data: Cow<str> = Cow::Borrowed("Salve Mondo!");

    let mut mutable_data = borrowed_data.to_mut();
    mutable_data.make_ascii_uppercase();

    println!("Mutated data: {}", mutable_data);
    println!("Original data: {}", borrowed_data);
}
```

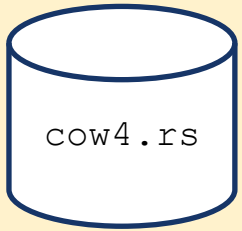


Chiamiamo to_mut() su un'istanza Cow<str> presa in prestito.

Se Cow<str> è nella variante Borrowed, to_mut() lo converte nella variante Owned, consentendo l'accesso modificabile ai dati.

Otteniamo quindi un riferimento mutabile (&mut str) ai dati sottostanti ed eseguiamo operazioni mutabili su di esso.

```rust
use std::borrow::Cow;

fn main() {
    let borrowed_data: Cow<str> = Cow::Borrowed("Salve ");

    let mut owned_data: String = borrowed_data.into_owned();
    owned_data.push_str("Mondo!");

    println!("Owned data: {}", owned_data);
}
```



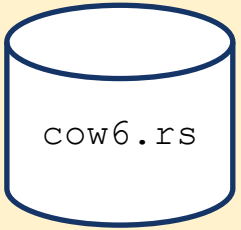
Chiamiamo into_owned() su un'istanza Cow<str> presa in prestito.

Se Cow<str> è nella variante Borrowed, into_owned() clona i dati sottostanti in una String posseduta.

Otteniamo la stringa posseduta e possiamo usarla.

```rust
use std::borrow::Cow;

fn main() {
    // Creazione di una stringa owned
    let owned_string = "Hello, World!".to_string();
    let cow_from_owned = Cow::from(owned_string);
    cow_from_owned.to_uppercase();
    println!("{:?}", cow_from_owned); // Output: Cow::Owned("Hello, World!")

    // Creazione di una stringa borrowed
    let borrowed_string = "Rust is awesome!";
    let cow_from_borrowed = Cow::from(borrowed_string);
    println!("{:?}", cow_from_borrowed); // Output: Cow::Borrowed("Rust is awesome!")

    // Creazione di una stringa owned da una stringa borrowed
    let another_borrowed_string = "Strings everywhere!";
    let cow_from_owned_borrowed= Cow::from(another_borrowed_string.to_string());
    println!("{:?}", cow_from_owned_borrowed); // Output: Cow::Owned("Strings everywhere!")
}
```

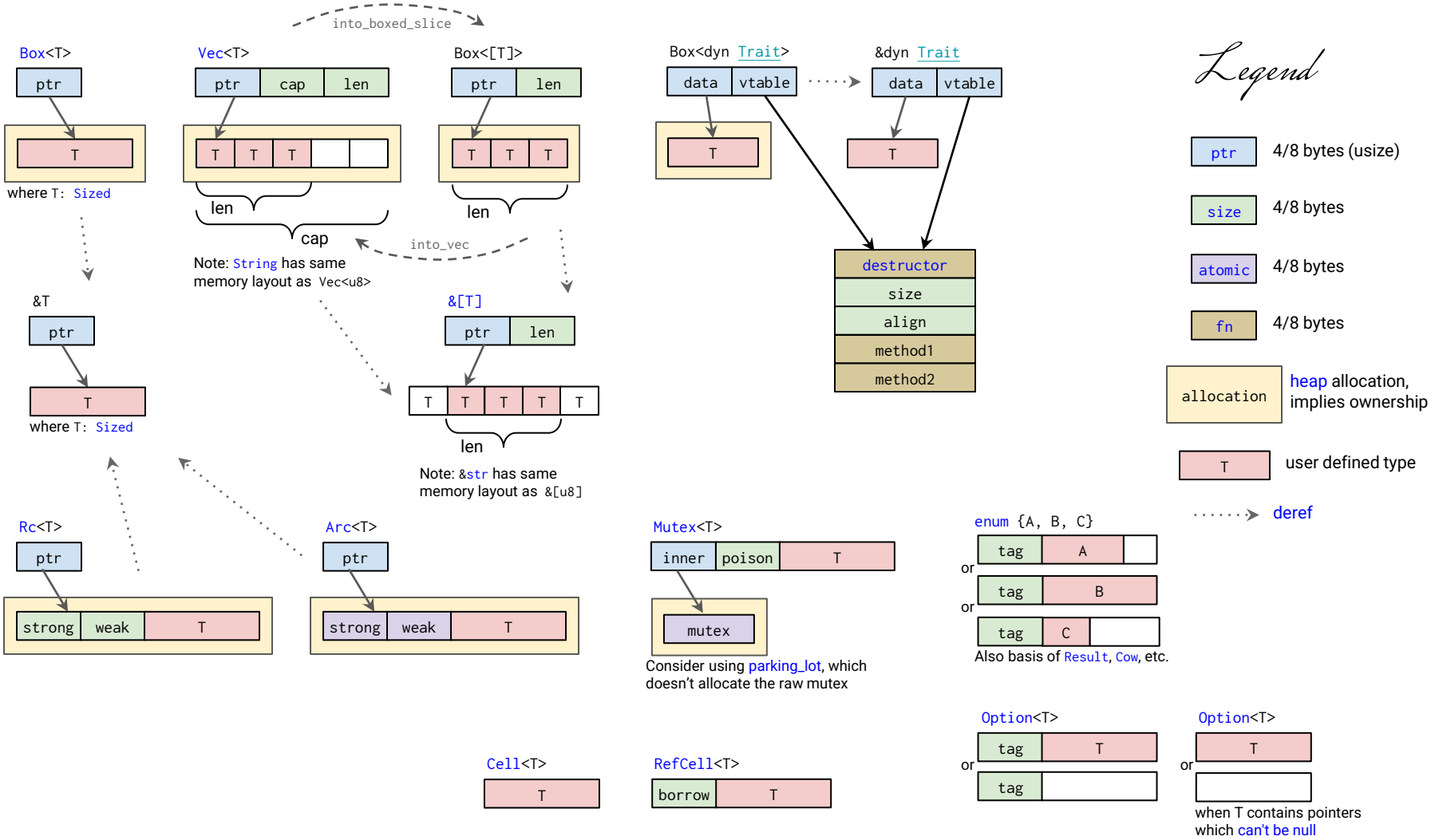
```rust
use std::borrow::Cow;
struct Data {
    content: String,
}
fn modify_data(data: &mut Data) -> Cow<str> {
    // Manipoliamo direttamente i dati senza clonarli
    data.content.push_str(", Rust!");
    Cow::from(&data.content)
}
fn main() {
    let mut external_data = Data {
        content: String::from("Hello"),
    };
    // Utilizzando il puntatore, evitiamo il clonaggio dei dati esterni
    let modified_data = modify_data(&mut external_data);
    println!("{}", modified_data); // Stampa "Hello, Rust!"
}
```



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);
  }
}
```



Rust container cheat sheet, by Raph Levien, Copyright 2017 Google Inc., released under Creative Commons BY, 2017-04-21, version 0.0.4

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