# I05-SmartPointer

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

Se disponiamo di un puntatore, possiamo effetture la deferenziazione applicando l’asterisco e accedendo quindi al contenuto della cella a cui quel puntatore fa riferimento.

Possiamo insegnare al compilatore che ci sono dei tipi che vogliamo fare sembrare puntatori:

* Si comporta come puntatore ma fa cose in più
  + Garanzia di inizializzazione e rilascio
  + Conteggio dei riferimenti
  + Accesso esclusivo con attesa

## Immagine che contiene testo, schermata, linea, diagramma Descrizione generata automaticamenteNote su c++

* **Unique\_ptr** → puntatore che non può essere copiato, esiste in **unica copia**
  + Simile a ref mut
  + Immagine che contiene testo, schermata, linea, diagramma

    Descrizione generata automaticamenteImmagine che contiene testo, schermata, Carattere, linea

    Descrizione generata automaticamenteImmagine che contiene schermata, linea, testo, diagramma

    Descrizione generata automaticamenteImmagine che contiene schermata, testo, linea, diagramma

    Descrizione generata automaticamentePrendo i suoi **dati** e lascio vuoto dall’altra parte
* **Shared\_ptr** → dealloco solo quando nessuno lo usa già
  + Immagine che contiene testo, schermata, Carattere, numero

    Descrizione generata automaticamentePossono morire nell’ordine che vogliono, quando cnt=0, nessuno è più interessato → posso liberare quel blocco di memoria
    - **Ptatore a dato**
    - **Struttura di controllo**
      * Con conteggio dei riferimenti hard
      * Con conteggio dei riferimenti weak
      * Puntatore al dato stesso
  + *Immagine che contiene testo, schermata, linea, Carattere

    Descrizione generata automaticamenteProblemi con i cicli → rischio che non si riesca a libeare*

Immagine che contiene testo, schermata, linea, Blu elettrico

Descrizione generata automaticamenteImmagine che contiene testo, schermata, linea, diagramma

Descrizione generata automaticamenteImmagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamenteImmagine che contiene testo, schermata, linea, Carattere

Descrizione generata automaticamenteImmagine che contiene schermata, linea, diagramma, Carattere

Descrizione generata automaticamente  
Hard link = 0 → distruggi il dato

* Weak link è anche lui a 0? → distruggi il blocco di controllo

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

Nota: in caso di cicli, questi si tengono in vita da soli e non si riescono a liberare.

**Weak:ptr**

Serve a creare dipendenze cicliche che incrementano il contatore di weak.

Non può essere usato, arriva solo al blocco di cotrollo ma **non posso accedere**. Se il weak ptr muore, decrementa il contatore.

**WeakPtr può essere promosso a shared** → va nel blocco di controlo, verifica che il dato esista ancora → crea temporanemanete uno shared pointer :

* Ptatore a controllo lo prendo da me
* Ptatore al dato dal blocco di controllo

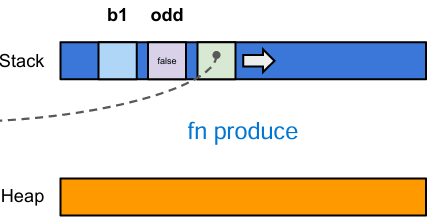
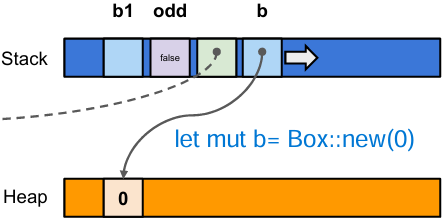
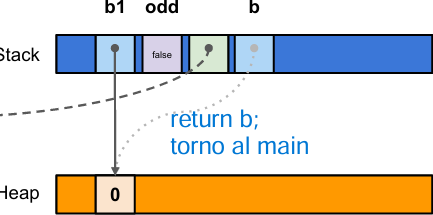
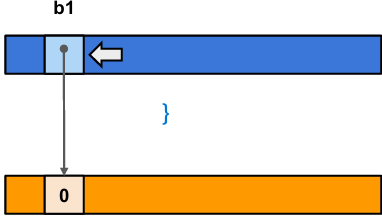
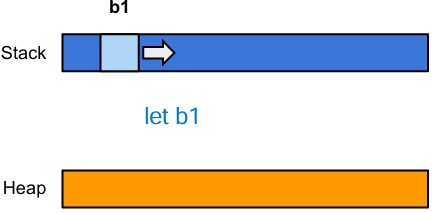
## RUST

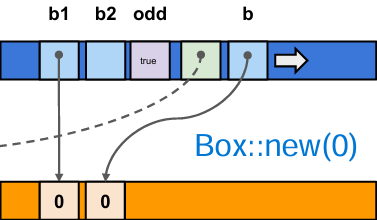
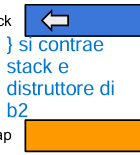
#### **std::Box**

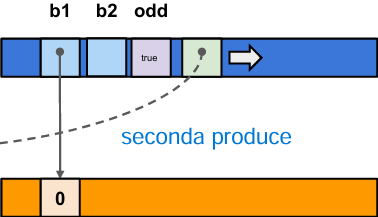
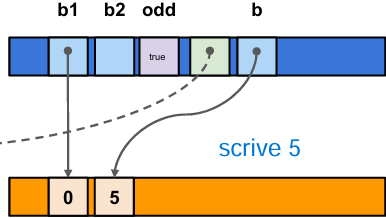
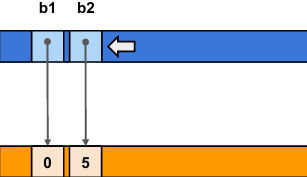
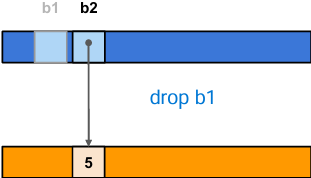
Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamenteSimile a unicPointer in c++ → singolo puntatore che punta necessariamente sull’heap

* Creabile attraverso Box∷new (t)
* **Prende possesso** del valore che gli passiamo come parametro e ne trasferisce il possesso al box stesso che lo conserva all’interno dello heap finchè oggetto non esce da context
  + quando la struttura esce dal proprio scope sintattico, il blocco sullo heap viene rilasciato automaticamente, grazie all’implementazione del tratto Drop
* In caso di movimento, **non implementa copy**, puntatore viene spostato al nuovo e vecchio box inaccessibile



Ritornato al main, possesso preso da b1

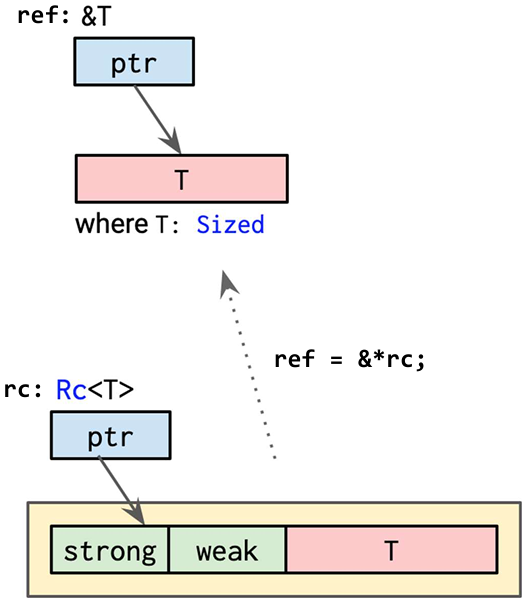


Nota sul tipo T che si passa alla BOX:

* 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, si trasforma in un **fat pointer** formato da un puntatore seguito da un intero di dimensione usize contente la lunghezza del dato puntato
* 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

***Nota****: Drop b1 → b1 chiama il suo distruttore. Rilascia il blocco e svuota heap*

#### **std::rc::Rc<T>**

Nelle situazioni in cui occorre disporre di **più possessori di uno stesso dato immutabile**.

Utile per alberi e grafi aciclici

*nota: Non può essere usato da più di un thread*

**Internamente**:

* copia del dato
* contatore puntatori esterni
  + si incrementa in caso di clone
  + decrementa se esce da scope
* contatore riferimenti deboli

&\*rc → \*rc mi fa puntare al dato, & mi da il riferimento a dove sta il dato, ottenendo così l’informazione di dove sio trova il mi dato.

Qaundo **creo un oggetto di tipo rc**:

Strong → 1

Weak → 0

**Implementa**:

**Clone** → incrementa strong di 1

**Drop** → decrementa strong di 1

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Queste funzioni sono state create con un “**this**” (non si chiamano self perché si chiamasse self, si potrebbe fare un self.strongcount)

* 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) -> 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::::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
  + *Per superare questo limite, si usa la classe* std::sync::Arc

RC∷**downgrade** → crea un Weak<T> : prende l’Rc che aveva precedentemente e invece di incrementare il tratto strong, incrementa il tratto weak.

*NOTA: Se si costruisse, usando Rc, 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*

* + - *Esso è una versione di Rc che contiene un riferimento senza possesso al blocco allocato*

RC∷**downgrade** → crea un Weak<T> : prende l’Rc che aveva precedentemente e invece di incrementare il tratto strong, incrementa il tratto weak.

#### **std::rc::Weak<T>**

Immagine che contiene testo, schermata, Carattere, documento

Descrizione generata automaticamente L’oggetto weak non implementa deref. → ritorna sempre un weak

* Se lo si vuole far diventare strong → devo chiamare **upgrade()** → ritorna un Option<Rc<T≫
  + Se strong > 0, incrementa strong e ritorna un nuovo sharePointer
  + Altrimenti no
* Al **drop**(five, tutto viene rilasciato), rimane solo il blocco di controllo da 16 byte e ho un contatore 0;1 → rimane weak\_five che punta alla sequenza 0,1 e non può fare upgrade in quanto strong==0

Nota:

* Weak pointer mi danno la possibilità di chiudere dei cicli
  + Perché non danno fastidio a strong
* Rc Incapsula Il Dato T Che E’ Immutabile
* Nota: se il valore originale è ancora in vita (strongCounter>0), è possibile costruire un nuovo valore di tipo Rc<T> invocando il metodo upgrade() che ritorna un valore Option<Rc<T>>>

*Il borrow checker garantisce, in fase di compilazione, che dato un valore di tipo T in ogni momento valgano i seguenti aspetti:*

* *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*

*Ci sono alcune situazioni in cui mi serve modificare un dato di cui ho solo il riferimento normale→ &T*

#### **std::cell::Cell**

implementa la mutabilità del dato contenuto al suo interno attraverso metodi non richiedono la mutabilità del contenitore → interior mutability

Modulo che offre alcuni contenitori che consnentono una mutabilità condivisa e controllata

* È possibile avere più riferimenti mutabili ad un valore mutabile
* I tipi offerti possono funzionare solo in contesti non concorrenti (basati su singolo thread

Rust mi fa inserire all’interno di una cella questo dato. T deve essere sized.

Immagine che contiene testo, schermata, Carattere, documento

Descrizione generata automaticamente

→ compilatore li interpreta diversamente

Pur avendo accesso ad un dato immutabile Cell, **posso cambiare il dato al suo interno (u8).**  
in questo caso il tipo T (u8) implementava copy → posso chiamare get

**Metodi** di Cell:

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

**Penalizzazioni** di Cell:

* Non mi permette di avere un riferimento a cosa c’è dentro (non implementa tratto ref)
  + Posso prendere possesso del suo contenuto, posso copiarlo ma non posso avere il riferimento del contenuto.

#### **REFCELL**

In alcune sItuazioni c’è bisogno in runtime di incapsulare in una cella e avere un riferimento.

**Nota**: la valutazione che ci sia solo un riferimento è solitamente fatta a compileTime, per RefCell viene fatta in RunTime con:

**Metodo Borrow** → per ottenere un riferimento semplce in lettura

* + se non c’è riferimento mut in giro, ok posso dartelo: ti do uno smartPointer che usi come fosse un ref ma che ha il tratto drop integrato → quando quella cosa che ti ho dato finisce, elimino e decremento conteggio di riferimenti semplici

**Metodo Borrow mut** → per ottenere riferimento mutabile → se chiedo riferimento mutabile e c’è già un riferimento mutabile, panica

**Dunque**:

* + - *Se c’è un riferimento semplice e chiedo mutabile → panic*
    - *Se c’è mutabile e chiedo semplice → panic*
    - *Se c’è mutabile e chiedo mutabile → panica*

*Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente*

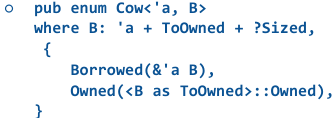
*borrow\_mut → flag da 0 va a -1*→ tryborrow → se ok va, altrimenti errore  
Dentro c, ci sarà flag -1 e m=6 → non posso accedere  
} → butta, a questo punto flag=0, m=6

→ in lettura

#### **std::borrow::Cow<a’, B> → Copy On Write**

Nel caso in cui un dato può essere conosciuto da tanti e qualcuno ogni tanto lo vuole cambiare solo per sé.

* Quando uno fa una modifica, effettua una clone per lui solamente con il nuovo valore





#### **SMART POINTER E METODI**

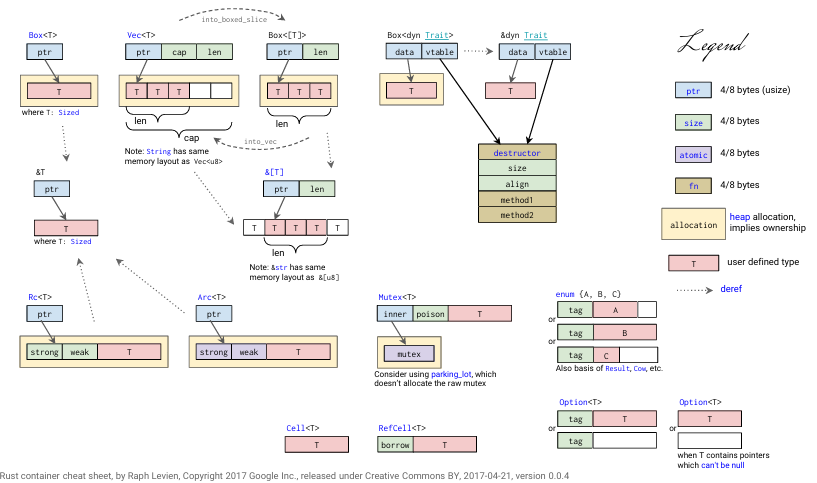
L’argomento self di un metodo può anche avere come tipo Box<Self>, Rc<Self> o Arc<Self>

* Implementano Ref che mi restituiscono il dato puntato

→ bisogna specificare la sintassi

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente



Appunti eliminati

Box: puntatore che punta ad un blocco sullo Heap

* Blocco di dimensione fissa
  + Sxe punta a slice → fatPointer
  + Se punta ad oggetto tratto → FatPointer(Vtable, rif)

Vec: smart pointer ad un blococ che può espandersi o cotnrarsi

Arc e Rc si differiscono per l’istruzione macchina usate per far eincremento o descremento del puntatore

* Dd
* Atomic in base a cosa c’è

CELL REFCEL : mi permettono di avere un ref semplice in cui il dato cambia

MUTEX: mi da la possibilità di trasformare un riferimento in sola lettura in una cosa mutabile, facendo in modo che se sono in due a provare a fare cose: uno entra e l’altro aspetta

RwLock : Operazioni compatibili, possono avvenire insieme (lettura) mentre se avviene scrittura, tutto bloccato

Cow: mi permette di gestire contenoieanemanet deid ati che alcune volte richiedono una copia e altre volte no

Immagine che contiene testo, Carattere, schermata, documento

Descrizione generata automaticamenteTutti questi sono dei dati che si comportano come puntatori ma non lo sono, possiamo usarli perché implementano Deref e DerefMut