**Emerging Programming Paradigms**

[ lezione 18 - 20 maggio ]

**Rust**

Un altro linguaggio, insieme ad Haskell, scritto da geni per geni. A differenza di Haskell, Rust è un *linguaggio di sistema*.

Lo sviluppo di Rust inizia del 2006 e viene distribuito dalla Mozilla foundation nel 2010. A quel tempo la concorrenza in Firefox era molto lenta, molti thread, molti lock, ecc. aprire molti tab rallentava parecchio il browser. L’obiettivo di Rust era creare codice sicuro e al tempo stesso efficiente. Il primo compilatore di Rust fu scritto in OCaml, che lo influenzò molto (insieme a C++), oggi è compilato in Rust stesso (utilizzando LLVM come backend, una toolchain che permette di avere un backend condiviso da frontend di vari linguaggi di programmazione).

Rust ha un **runtime molto minimale**, usa thread di sistema e non ha garbage collection. Go ad esempio invece ha un sistema di thread gestito dal linguaggio stesso (come in Erlang), che porta benefici per la concorrenza, ma appesantisce il runtime.

Il principio fondante di Rust, condiviso con C++, è lo **zero-cost abstractions**. Ovvero Rust cerca di avere più costrutti ad alto livello possibile ma solo se è possibile far si che a runtime non introducano un costo addizionale rispetto al codice che verrebbe scritto a mano dall’utente. In particolare non bisogna pagare a runtime una “penalty” per features che non si utilizzano, ad esempio in Java si paga dynamic dispatch per tutte le funzioni, anche quelle che non ne avrebbero bisogno.

Rust ha inoltre **polimorfismo parametrico** (polimorfismo di riga, ecc) tramite monomorfizzazione, quindi compila tante volte lo stesso codice istanziato sui vari tipi.

Interessante è il complesso sistema di **ownership dei dati in memoria** che viene interamente imposto a compile-time. Quindi è il compilatore che garantisce che un certo dato sia posseduto da una certa struttura dati, e che la memoria venga liberata (il dato rilasciato) al momento giusto, che non ci siano doppi accessi, accessi concorrenti, …

La **guaranteed memory safety** garantisce l’assenza di memory leaks, nessuna doppia allocazione, nessun dangling pointers ne data races. Tutto questo è garantito dal sistema di tipi.

Infine Rust lavora con concorrenza su memoria condivisa, il type system (con l’aiuto di un meccanismo di smart pointers) minimizza i problemi legati alla concorrenza, fornendo la cosiddetta **fearless concurrency**.

Alcune features *non* caratteristiche di Rust sono: le chiusure, gli ADT con pattern matching, non ha valori null (si usa il tipo option), utilizza polimorfismo parametrico bounded (templates/generics con bound tramite traits), in origine aveva classi ma furono rimosse in fretta, ora ha traits, traits objects (per il dynamic dispatch), usa i trait come bound nel polimorfismo parametrico ed ha un meccanismo di associated types (tutti casi particolari di type-classes). Infine per gestire la visibilità utilizza un meccanismo di moduli annidati (alla OCaml, ma non utilizza caratteristiche più avanzate come funtori o module types).

**Gestione della memoria**

Rust per gestire la memoria, garantendo sia l’assenza di errori di gestione della memoria che la fearless concurrency, utilizza due meccanismi:

1. un meccanismo complesso (e molto restrittivo che impone di scrivere il codice in una maniera molto rigorosa) basato su un sistema di **ownership** (di tipo RAII - Resource Allocation is Initialization, usato anche in C++, ovvero il momento in cui si alloca una risorsa è anche il momento in cui viene inizializzata e la riallocazione viene gestita dal linguaggio di programmazione). Questo meccanismo prevede che il compilatore abbia sempre la nozione di quale struttura dati mantenga il controllo su un’area di memoria, nessun altro può accedere né in lettura né scrittura a quell’area di memoria, se non “prendendola a prestito” (meccanismo di **borrowing**) in lettura o scrittura. In caso di borrowing in scrittura ad esempio, nemmeno il proprietario potrà accedere all’area di memoria fino a quando non viene restituita. Borrowing in lettura, lascia comunque la possibilità al proprietario di leggere l’area di memoria e di prestarla ad altri, solo in lettura però. Tutto questo a tempo di compilazione, è il compilatore che sa chi ha il controllo delle aree di memoria.

2. un meccanismo a runtime di **smart pointers** (presente anche in C++) per il controllo della memoria (quindi appesantisce leggermente a runtime, ma compensa con il punto 1 a compile time).

Downsides di Rust:

- curva di apprendimento estremamente ripida

- la gestione della memoria in concorrenza è un problema indecidibile, quindi da una versione all’altra il compilatore cambia molto.

**Smart pointers**

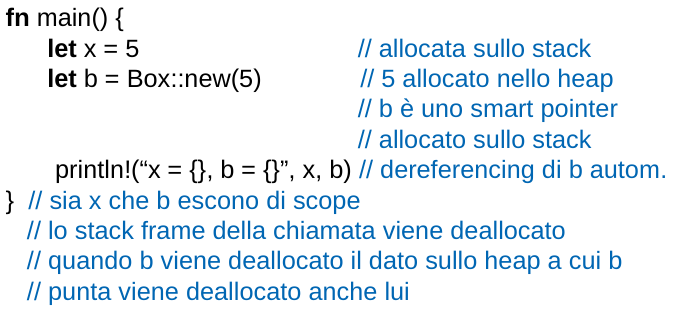
Lo smart pointer è una struttura dati che può essere user defined (in una libreria ad esempio) o system defined (definita dal runtime di un linguaggio di programmazione).

In generale è una struttura dati che implementa un certo numero di traits, che in qualche modo hanno a che fare con funzioni che gestiscano la vita delle variabili. Ad esempio esiste una funzione che è invocata quando una variabile allocata sullo stack di tipo smart pointer esce di scope; il compilatore quando produce il codice di chiusura del blocco e una variabile esce dallo scope aggiunge righe di codice che invocano la funzione definita da un trait (il cui scopo è gestire i dati aggiuntivi quando la variabile viene deallocata). Altri esempi di traits sono funzioni che vengono invocate quando si copia un puntatore, o vi si accede in lettura, in scrittura, ecc.

Uno smart pointer non è solo un puntatore è una *struttura dati generica che astrae tutte le operazioni effettuabili con un puntatore*, che possono quindi implementare una gestione differente dei side effect. Può essere visto come una monade che gestisce i side effect dati dall’uso di un puntatore.

L’idea di uno smart pointer è quella di forzare determinate politiche di gestione della memoria. Ad esempio esiste uno smart pointer che forza l’allocazione di un dato nello heap, chiamato appunto Box<T>, esiste uno smart pointer Rc<T> che fa reference counting (un dato insieme al numero di volte che viene utilizzato, è una monade), esistono smart pointer che accedono in modo atomico in lettura o scrittura in memoria, la garbage collection è un altro esempio di smart pointer, ecc.

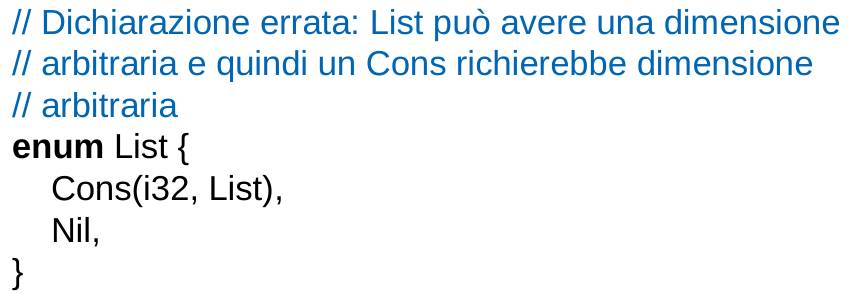
Esempio di codice Rust



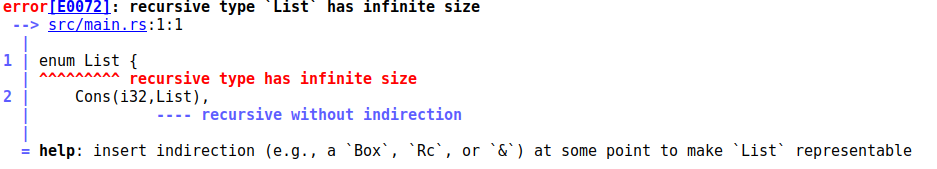
b è lo smart pointer, sullo stack, che punta al dato 5 nello heap. Nella print b è dereferenziato (ovvero si va a prendere il suo valore sullo heap). Quando il blocco viene chiuso, il compilatore inserisce del codice che chiama la funzione del trait Box che serve a deallocare b e toglie il resto dallo stack.

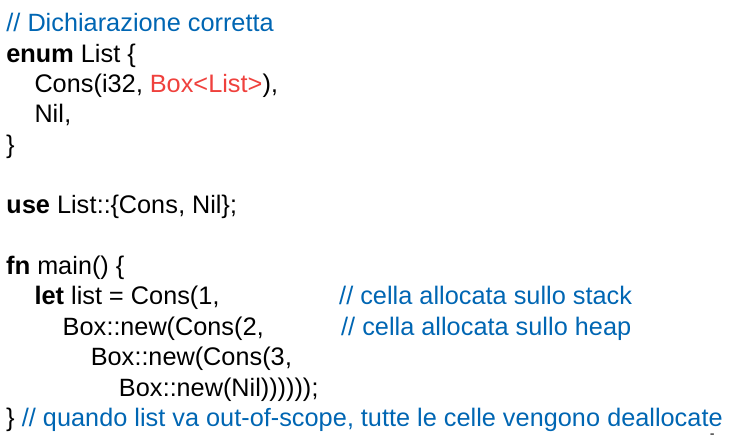
I dati in Rust usano un numero non uniforme di bytes (come in C). Box<T> permette di allocare dati di grandi dimensioni sullo heap, facendo riferimento ad essi con uno smart pointer di dimensione fissata e piccola (una parola).

**ADT Ricorsivi**



Questa definizione è *scorretta* perché, di default in Rust è tutto “unboxed”, quindi questa lista verrebbe allocata tutta consecutivamente sullo stack, quindi non avrebbe una dimensione fissata (e di conseguenza non potrebbe essere passata in input ad una funzione in quanto il compilatore non saprebbe in anticipo la lunghezza della lista).

Provando a compilare e infatti dà l’errore:  


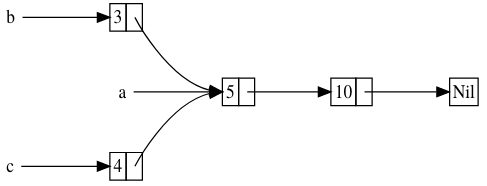


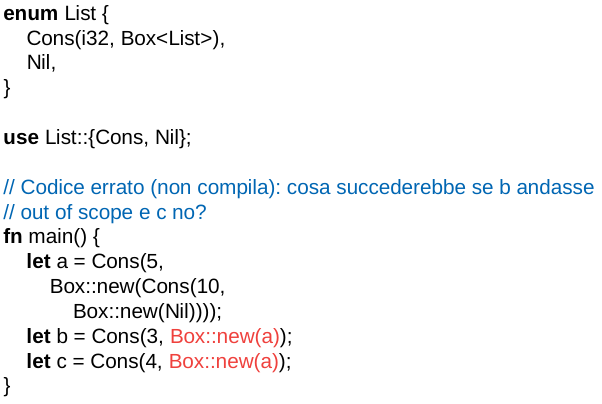
La dichiarazione corretta è invece la seguente. Si usa un puntatore alla lista allocata sullo heap (tramite lo smart pointer Box<List>). L’utilizzo è mostrato nel metodo main creando la lista [1, 2, 3]. Quando viene chiusa la main viene invocata da parte del compilatore la funzione che si occupa della deallocazione della prima Box<List> e ricorsivamente l’operazione verrà ripetuta per la cella puntata, e così via.

Il trait Box non implementa la copia, quindi non è possibile creare due puntatori allo stesso dato in memoria. Questa è una garanzia che dà il tipo di dato Box, quindi non solo garantisce il fatto che il tipo di dato verrà liberato al momento opportuno ma anche che non si avranno mai due puntatori allo stesso dato (quindi ad esempio non si avrà bisogno di lock in programmazione concorrente perché a un dato accede solo uno).

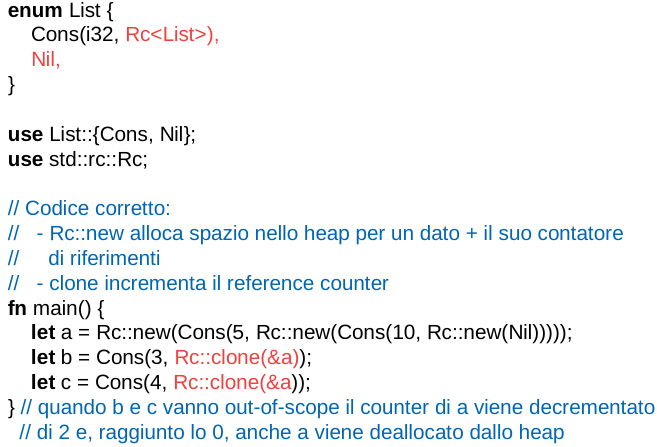
Se si vogliono due puntatori allo stesso dato allora si può utilizzare ad esempio lo smart pointer Rc<T> (reference counting). Un esempio di utilizzo è se si vuole implementare lo sharing di sotto-liste.

NOTA: lo smart pointer Rc<T> viene usato solo in ambito sequenziale, per ovviare a problemi di accesso multiplo allo stesso dato da parte di entità differenti.





Il codice sopra viene rigettato perché quando si definisce c a è già morto perchè usato da b. La soluzione corretta con Rc è la seguente:



Anche qui al termine del main vengono invocate le relative funzioni per decrementare il RC e rimuovere la lista dalla memoria.

Rust garantisce che un dato possa avere al più una reference mutable oppure un numero arbitrario di reference in sola lettura (no data races). Box<T> non ha restrizioni da questo punto di vista, perché Box viene acceduto una volta sola, ha un solo puntatore entrante. Rc<T> non è pensato per lavorare in concorrenza, quindi permette solo reference di sola lettura, lo smart pointer di tipo rc permette ad esempio clone, ma solo in lettura.

**Recap**

Un linguaggio di sistema con controllo totale di dove vengono allocati i dati (stack/heap).

Rust utilizza tutte le feature emergenti che abbiamo visto fin’ora.

Il meccanismo (il primo) di gestione della ownership viene dalla logica lineare e dall’isomorfismo di Curry-Howard.

Libra è implementato in Rust e ispirato a Rust, e (nel contesto degli smart contract) implementa lo stesso meccanismo di tipaggio di Rust (quindi il compilatore influenza il linguaggio).

**Smart pointers**

**Box**<T> un solo puntatore in sola lettura.

Se si volesse costruire una lista mutabile probabilmente si userebbe Box, in quanto essendo mutabile non è consigliabile che venga condivisa con altri.

**Rc**<T> reference counting. Utile per condividere ma permette reference solo in lettura.

Qui si inizia a vedere la complessità di rust: non c’è un solo tipo di lista! ci sono liste mutabile, non mutabili, liste mutabili però non posso modificarne la lunghezza, etc…

**RefCell**<T> verifica a runtime che ci siano o solo un *mutable reference* (un puntatore che può anche scrivere) oppure N puntatori in sola lettura. Da usare con parsimonia perchè appesantisce, meglio usare i controlli a compile-time di Rust.

------------ Breve excursus ------------

È comune dover utilizzare gli smart-pointer perché i controlli a compile time non bastano?

Dipende dalle strutture dati, se sono semplici si riesce a usare il controllo a compile-time.

Il caos nasce quando si hanno delle strutture dati né temporanee (scope locale a una funzione) né persistenti (lifetime programma) ma bensì attive solo per una parte dell’esecuzione del programma. Il sistema a compile-time si complica.

Il compilatore comunque ad ogni versione viene migliorato, i controlli a compile-time sono problemi indecidibili e il compilatore fa approssimazioni.

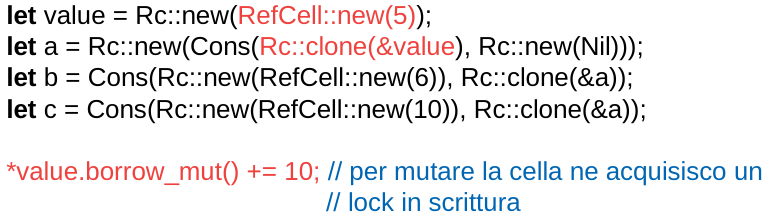
----------------------------------------

Si usa quindi Rc<RefCell<i32>> per esempio per creare un intero condivisibile (Rc) e mutabile (RefCell). Ad esempio una lista con la seguente forma:



Essendo la coda della lista condivisibile ma non mutabile (solo Rc) si possono modificare gli altri valori ma non è possibile accorciare o allungare la lista.

Esempio di utilizzo della lista e di borrowing per mutare che termina subito dopo la mutazione:



Gestire i cicli di reference

I cicli di reference sono un problema con reference counting, è necessario usare i **puntatori weak** ( Weak<T> ).

Nelle strutture cicliche il reference counting non è mai a 0 quindi avviene *memory leak*.



Per evitare questo problema si possono usare i puntatori deboli, puntatori il cui conteggio non va a sommarsi a quello del reference counter.



Cioè per creare una struttura circolare si utilizzano puntatori forti e deboli (per una lista circolare basta che uno sia debole).

Una cella viene considerata viva se ha almeno un puntatore forte entrate.

Esempio:



quando si esce dallo stack frame, il puntatore forte viene deallocato, la cella avendo RC 0 verrà considerata morta e liberata.

upgrade(r Weak<T>) -> Option<Rc<T>>

La funzione upgrade data una weak reference ritorna una Rc<T> se l’oggetto è ancora allocato, fa quindi diventare un puntatore debole un puntatore forte. C’è option perchè il puntatore può diventare forte solo se il puntatore debole puntava a qualcosa. Essendo debole poteva essere dangling.

Invece il downgrade non usa ovviamente option:

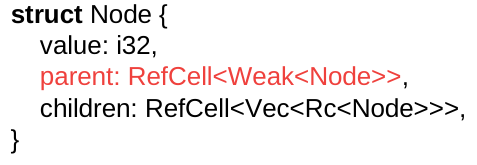
downgrade(r Rc<T>) -> Weak<T>

Esempio: **alberi**

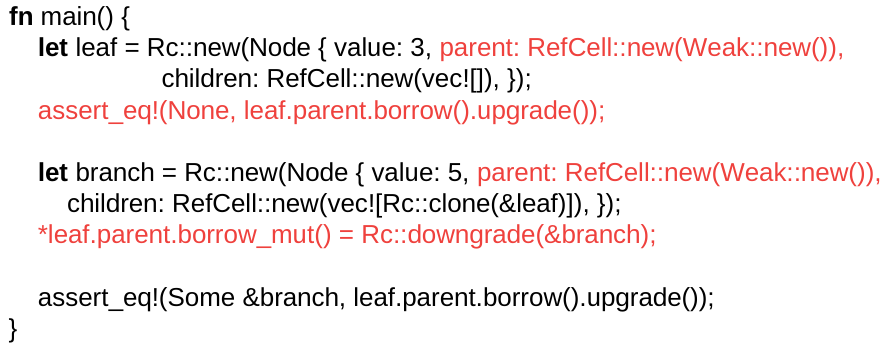
Un nodo ha due puntatori forti ai figli e un puntatore debole al padre, infatti se un nodo viene sganciato da un albero (il padre non punta più a lui), allora diventa irraggiungibile e deve essere eliminato (i figli puntano ad esso come weak, quindi non inficiano il reference counter). Anche i discendenti sono irraggiungibili e vanno reclamati.



In codice Rust:



Un nodo ha un valore intero, una weak reference (mutabile) al padre, e un vettore (mutabile) di strong reference (in sola lettura non mutabili) per i figli.



La leaf viene costruita come un puntatore contenente il valore 3, una weak reference vuota (in quando non ha nessun padre al momento) e un vettore vuoto in quanto una foglia non ha figli. Il branch è sempre un puntatore contenente il valore 5, come parent ha un weak vuoto e come figli un vettore contenente una copia (in sola lettura) del puntatore alla foglia creata in precedenza. Dopo la creazione del branch viene modificato il parent (weak) della foglia per farlo puntare al branch (la cui reference è stata downgraded per farla risultare weak in modo che combaci con il tipo del parent della foglia).

Questo esempio è un po’ difficile perchè Coen chiama Branch un nodo padre e leaf un nodo figlio. Quello che fa è creare un nodo Leaf scollegato, poi creare un Nodo Branch a cui mette Leaf come figlio e dopo imposta a Leaf il padre.

Puntatore al parent RefCell, perché così posso mutarlo, ad esempio staccarlo. Mentre Weak perché verso al padre non conta il puntatore.

Puntatori ai figli sono un vettore di puntatori forti a nodi (mutabili = RefCell, Vec = vettore, Rc = reference counter, Node = al nodo).

Posso creare una foglia con Rc::new e gli metto un nodo con valore 3, poi creo un puntatore debole al parent e lo metto in una RefCell (non punta a niente, perché non metto niente nella new = dandling) e poi creo dei figli con vec![] (creo un vettore vuoto).

La assert\_eq è per dimostrare che il programma funziona e gli *invarianti* valgono, altrimenti da errore. Quindi se faccio leaf.parent, per avere il parent, .borrow per avere un puntatore in lettura a un Weak e poi faccio upgrade per trasformarlo in forte, ottengo None. Questo è corretto perché non punta a niente.

Creo un branch, ovvero un nodo, con valore 5, dove il parent è Nan e come figlio creo un vettore che contiene solo la mia foglia, ottenuta tramite Rc::clone(&leaf), creata prima.

Assert\_eq chiede se il genitore della mia foglia, preso in prestito in sola lettura, a cui faccio upgrade, è uguale a branch.

Esempio

Anche nei linguaggi con garbage collector si possono creare situazioni dove la memoria non viene reclamata anche se dovrebbe. Ad esempio: una hash-table viene utilizzata per fare memoization, ovvero associare input a output di una funzione per non ricalcolarli ogni volta. Se il puntatore dalla hash-table all’input è strong, poiché la hash-table è sempre reachable, tutti gli input usati in passato non possono essere più reclamati anche se sono unreachable in altri modi (gli argomenti sarebbero sempre raggiungibili e quindi mai deallocati). Soluzioni: weak pointers, weak hash tables, memonoids, etc. (googlate OCaml + questi nomi per avere esempi). Per la cronaca, quello che chiama Memonoide l’ha sentito a un talk dove però li chiamavano [Ephemeron](https://en.wikipedia.org/wiki/Ephemeron).

**Smart pointers per scenari concorrenti**

In questi scenari va gestita la race condition, il trait smart pointer deve quindi anche acquisire e rilasciare lock in modo da rendere le operazioni atomiche.

**Mutex**<T> è uno smart pointer che punta a T, ma per poter usare il puntatore che fa accedere a T, è necessario passare attraverso il metodo lock(). Se qualcuno ha già fatto lock(), allora l’accesso viene sospeso.

**Arc**<T> Atomic reference counting è del tutto simile a Rc, ma è atomico e funziona bene in uno scenario concorrente, perché essendo atomico si è sicuri che acquisisca il lock prima di incrementare e decrementare il reference counter.

Non bisogna pensare Arc e Mutex come alternativi, ma vanno usati insieme. Mutex è l’ultimo “scrigno” prima di un dato, quello che protegge effettivamente il valore. Arc permette di condividere un dato in tranquillità con gestione della deallocazione automatica ma va usato insieme a Mutex per garantire che solo un agente alla volta possa mutare cosa c’è dentro.

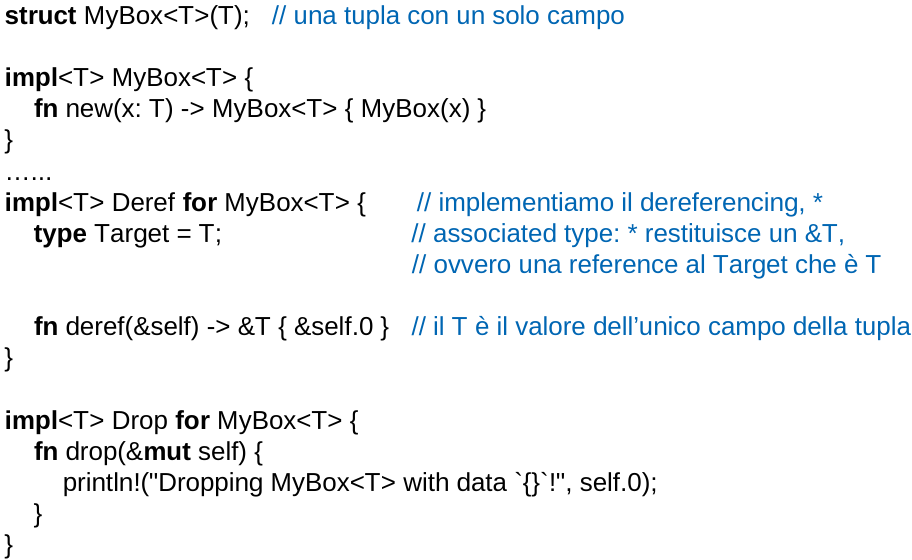
In C++ ci sono puntatori analoghi anche per gestire la concorrenza, ma per esempio non controllano che si cloni, etc… il compilatore in C++ non aiuta e garantisce meno correttezza.

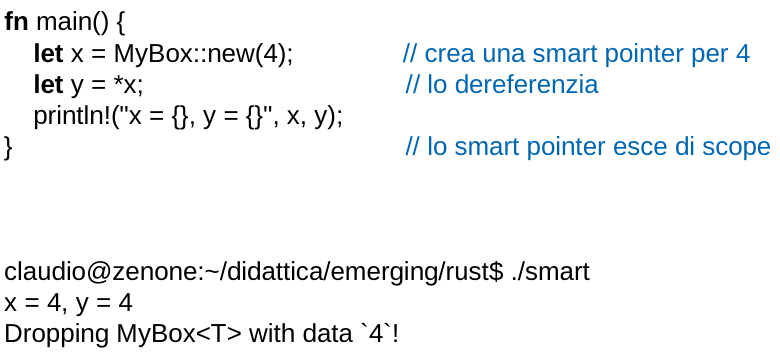
**User defined smart pointer**

Basta definire un tipo di dato che implementi certi trait degli smart pointer (a piacimento).

È possibile che per implementare certe funzionalità sia necessario bypassare i rigidi controlli di Rust, e Rust lo permette se si vuole definire uno smart pointer. Cioè si utilizza un costrutto per dire a Rust di non controllare più nulla e fidarsi. Aprire e chiudere un blocco unsafe. Dentro un blocco unsafe i puntatori sono come in C.

Esempio (senza codice unsafe):





Domanda: qual è la differenza tra borrowing e reference? Non sono cose distinte, il meccanismo linguistico per ottenere reference è il borrowing.

**Ownership + Borrowing**

> **Ownership**

Ogni cella di memoria sullo heap ha un **owner**, che è responsabile per la sua deallocazione.

Quando una cella sullo heap viene creata e un puntatore ad essa viene assegnato a una *variabile sullo stack*, quest’ultima ne diventa l’*owner*.

Quando invece il puntatore viene assegnato a *un’altra cella sullo heap*, questa ne diventa l’*owner*. Quando un owner viene deallocato (e.g. il blocco di una variabile sullo stack viene deallocato) le celle ricorsivamente possedute vengono rilasciate.

*Trasferimento di ownership*

Una cella sullo heap ha sempre uno e un solo owner, assegnamenti e passaggio come parametri della variabile/cella che ha l’ownership trasferiscono (move) l’ownership. Quando una variabile perde l’ownership, essa non può più essere utilizzata!

Solo l’owner è responsabile di deallocare una e una sola volta la cella che possiede.

Quando *assegni* o *passi come parametro* trasferisci l’ownership, che comunque rimane sempre unica. Se una variabile perde l’ownership di un cella nello heap, non può più venire utilizzata (dall’ex-owner) per scriverci/leggerci, il compilatore dà errore.

NOTA: solo le variabili sullo heap hanno un owner.

Ownership vs borrowing  
L’idea che l’ownership è un concetto più forte del borrowing. Il lifetime dell’owner diventa il lifetime dell’oggetto posseduto, cosa che non succede nel borrowing.

Ownership = chi è il responsabile della deallocazione del dato

Borrowing = uso temporaneo del dato senza prenderne l’ownership

Di ownership c’è solo un tipo, mentre di borrowing puoi scegliere se farti prestare un dato in forma mutabile o no.

References

&x è una **reference** a(l contenuto di) x. &mut x è una reference a(l contenuto di) x che permette di modificarne il contenuto. Se x ha tipo T, &x ha tipo &T e &mut x ha tipo &mut T. Prendere una reference di una variabile implica fare borrowing della variabile.

Non si hanno *data races* (anche concorrentemente):

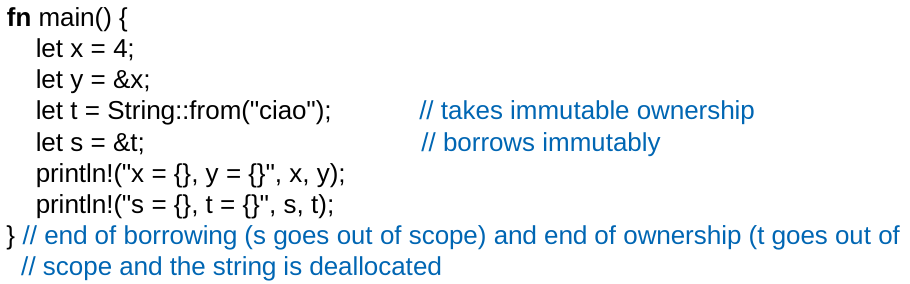
- se una variabile è borrowed mutably, nessun altro borrow è possibile e l’owner è frozen (= non può accedere alla variabile fino a quando il borrowing non è terminato)

- se l’ownership è mutable e la variabile viene borrowed, l’owner è frozen (= non può modificare la variabile fino a quando il borrowing termina).

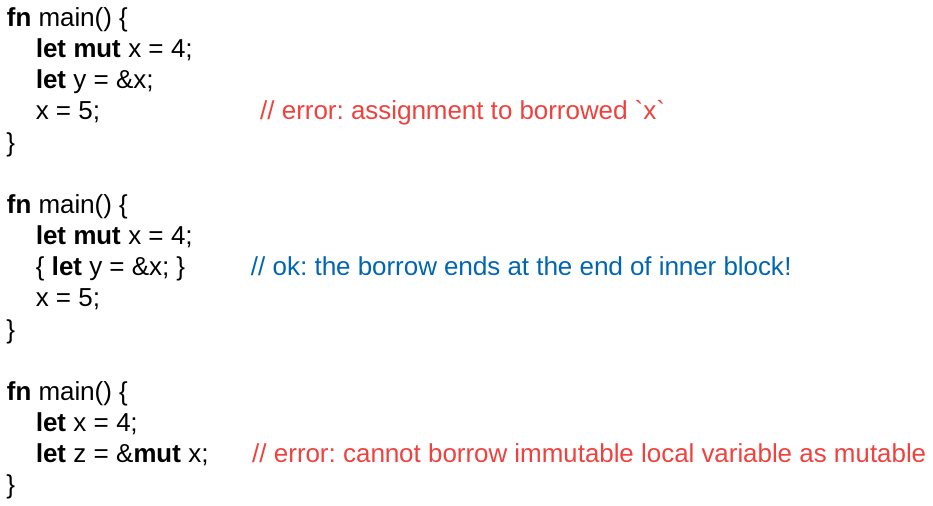
Le reference sono diverse dai puntatori e in generale sui puntatori sono possibili più operazioni. Una reference dangling non ha senso, mentre un puntatore sì e via dicendo.

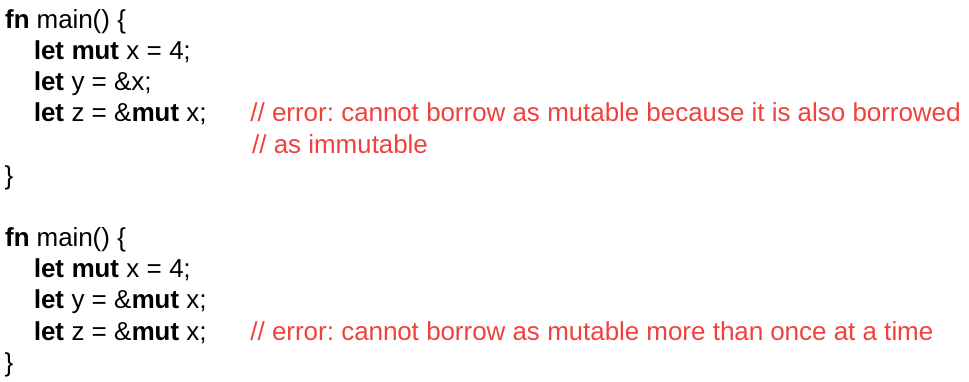
> **Borrowing**

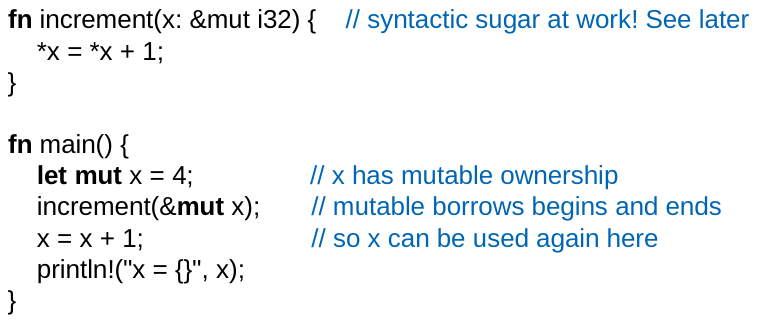
Esempio di borrowing implicito tramite reference:



Altri esempi:







**Lifetimes**

Se si crea una funzione che prende in prestito un dato, e poi lo dà in prestito ad un altra funzione il tempo totale di prestito si allunga, infatti è necessario che la funzione iniziale mantenga il dato *almeno* tanto quanto basta perché la funzione secondaria termini (per evitare dangling pointers). Viene definito in Rust il concetto di *lifetime*.

Le celle di memoria hanno un lifetime che indica quando la cella verrà deallocata dall’owner.

Il lifetime è diverso dallo scope, ad esempio nel caso in cui l’ownership venga trasferita.

Ogni reference ha sempre due lifetime:

1 - il lifetime della reference vera e propria

2 - il lifetime di ciò che è puntato

In Rust non si possono avere **dangling pointers** in quanto il linguaggio verifica che il primo lifetime sia sempre inferiore al secondo (sintassi concreta: ‘a : ‘b per ‘a,’b variabili di lifetime con significato “’a termina dopo ‘b”), cioè non è possibile che una reference viva più a lungo di ciò a cui punta.

Rust permette di scrivere codice che sia *unicamente* **polimorfo nel lifetime**. Ovvero significa che è possibile scrivere *variabili di lifetime*. Hanno la stessa sintassi di Ocaml, scritte con l’apicetto cioè alpha beta etc.. diventano ‘a, ‘b … Sono tipi polimorfi ma è anche possibile specificare la durata del lifetime di questi.

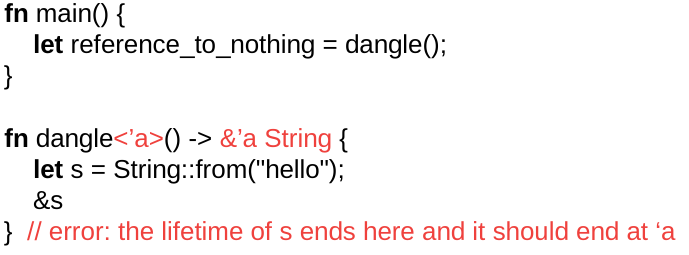
**‘static** serve per fornire un lifetime a variabili globali, che diventa quello del programma.

Posso fare polimorfismo bounded tra i lifetime delle variabili per esempio: ‘a : ‘b

vuol dire per ogni alpha per ogni beta a patto che il lifetime di alpha sia minore di quello di beta.

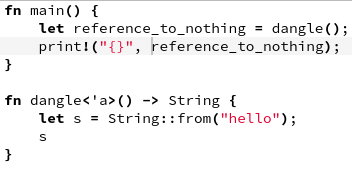
&’a i32 indica una reference che punta in modo non mutabile a un dato che è un intero che terminerà al lifetime ‘a.

Esempio:

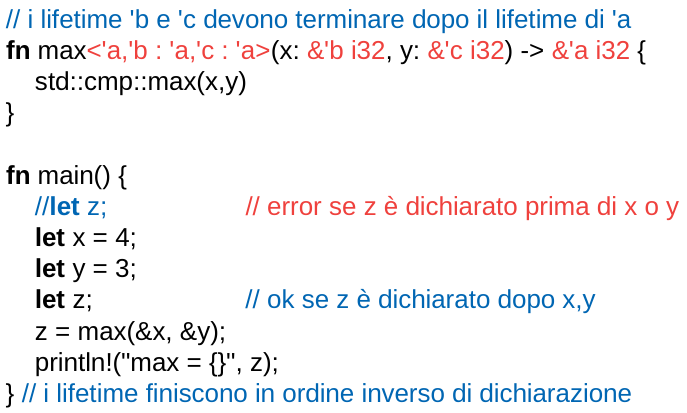


In questo esempio si prova a creare un dangling pointer con il poliformismo sui lifetime, ovvero non viene messo alcun limite alla durata del lifetime. Ma Rust dà errore in quanto capisce che diventerà dangling pointer!

L’esempio sopra dà errore, ma la seguente versione invece no:



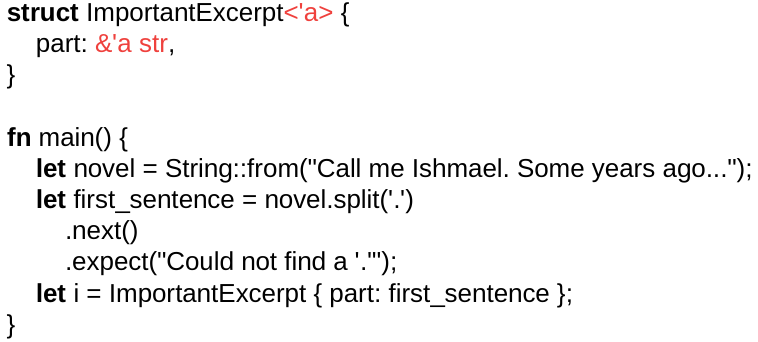
Esempio:



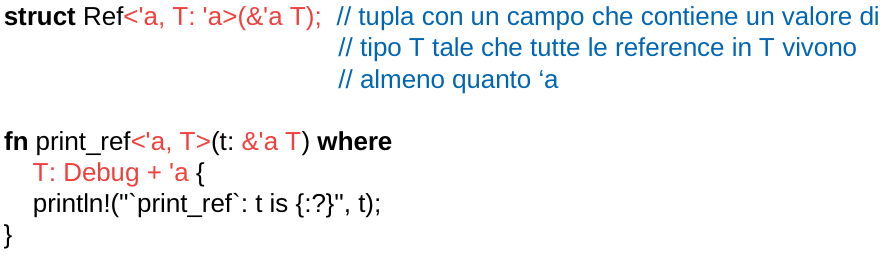
Nell’esempio sopra si vede che si può definire una funzione max (restituisce una reference al più grande dei due valori) indicando che il lifetime degli operatori deve essere maggiore di quello della variabile di risultato (perchè è pur sempre una reference, e sarebbe dangling pointer altrimenti).

NOTA: i lifetime terminano in ordine inverso di dichiarazione (per questo z va dichiarato per ultimo).

Non è sempre necessario specificare l’ordine dei lifetime come nell’esempio precedente, ad esempio:



In questo caso il compilatore controlla che ciò che è messo in reference in part non muoia prima di i, della struttura insomma.



**Slices**

Uno slice è uno smart pointer per fare borrowing (mutabile o meno) di una parte di una struttura. In quanto smart pointer le slices hanno lifetimes.

Ad es: str (string slices), &[T] (vector slices),

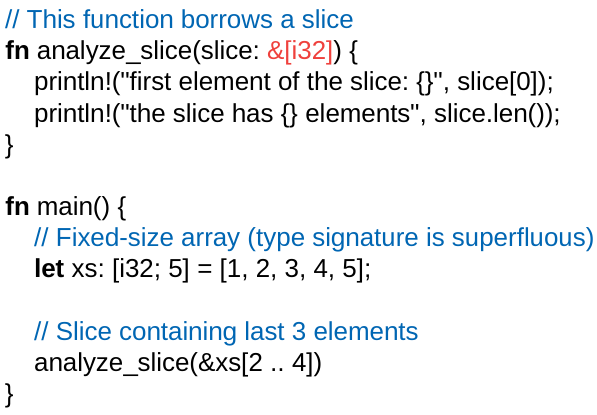
Uno slice consiste in: un puntatore + il numero di byte + capacità residua.

Sono utili ad esempio in ambito concorrente dove ogni thread prende in prestito una parte di struttura su cui deve operare scrivendo (perchè se leggesse e basta non ce ne sarebbe bisogno).

A *runtime* una slice è una tripla che contiene un puntatore all’interno del mio vettore, il numero di byte e la capacità residua. La capacità residua serve perchè scrivendo si potrebbe far crescere la struttura, centra con il paginamento fatto dal sistema operativo, quando spazio ho nella locazione di memoria assegnata al programma.

Ad esempio Rust ottimizza l’utilizzo degli string literals mettendoli tutti in un unica porzione di memoria continua e poi ogni reference è uno slice di questo. Ogni volta che si fa String::from(“qualcosa”) si fa uso di questa feature.

Esempio:



**Chiusure**

In Rust le chiuse “catturano” le variabili libere che si utilizzano all’interno, cioè non usano il sistema di Erlang o OCaml che mettono le variabili libere sullo heap (costoso) ma bensì ci si accede per borrowing mutabile/immutabile o move. Questi 3 hanno ognuno una differente sintassi.

Vi è un trasferimento di ownership se la chiusura è di tipo move: (move | params | {body}), altrimenti si effettua borrowing mutabile/immutabile.

Vi è un trait per le funzioni, uno per le chiusure move, uno per le chiusure con borrowing mutabile, uno per quelle immutabili, ecc.

“Da grandi poteri derivano grandi responsabilità”

Rust è il contrario, hai poche responsabilità perchè hai pochi poteri.

In **conclusione**

L’analisi statica di Rust aiuta a prevenire moltissimi errori comuni nella gestione esplicita della memoria e nella concorrenza con memoria condivisa.

I messaggi di errore sono spesso molto complessi da decifrare e la soluzione è spesso non ovvia (vedi Q&A sulle mailing list).

Bisogna cercare la (ri)formulazione del codice/algoritmo accettata da Rust.

È necessaria un’ottima conoscenza degli smart pointers di libreria