



Lifetime

Riferimenti ed esistenza in vita

Riferimenti e funzioni

- Se una funzione riceve un parametro di tipo riferimento (mutabile o meno), il tempo di vita del riferimento diventa parte integrante della firma della funzione
 - E' necessario infatti che tutte le operazioni eseguite dalla funzione sul riferimento **siano compatibili** con la validità del dato memorizzato al suo interno
 - `fn f(p: &i32) { ... } ⇒ fn f<'a>(p: &'a i32) { ... }`
 - Il compilatore provvede in molti casi a effettuare autonomamente la riscrittura indicata (*lifetime elision*)
- Se la funzione riceve **più riferimenti**, può essere necessario indicare se il loro tempo di vita sia vincolato al più breve o se siano disgiunti
 - Nel primo caso si usa un solo identificatore `fn f<'a>(p1: &'a i32, p2:&'a i32) { ... }`
 - Nel secondo si usano etichette diverse `fn f<'a, 'b>(p1: &'a i32, p2:&'b i32) { ... }`
- Nel caso in cui la funzione sia generica, le meta-variabili di tipo vengono indicate dopo gli identificatori del tempo di vita
 - Questi ultimi non partecipano al processo di monomorfizzazione

Riferimenti e funzioni

- Uno dei casi più frequenti in cui il compilatore non riesce a dedurre il corretto tempo di vita è legato a funzioni che **restituiscono un riferimento**, estratto da una delle strutture dati ricevute in ingresso
 - In tale caso occorre annotare il tipo restituito con la corretta etichetta
 - `fn f<'a, 'b>(p1: &'a Foo, p2:&'b Bar) -> &'b i32 { /*...*/ return &p2.y; }`
- Se la funzione memorizza il riferimento ricevuto in ingresso in una struttura dati, il compilatore deduce che **il tempo di vita della struttura** in cui il riferimento è memorizzato **deve essere incluso** o coincidente con **il tempo di vita del riferimento**
 - Se questo non avviene, il compilatore identifica l'errore e impedisce alla compilazione di avere successo

Riferimenti e funzioni

```
fn f(s: &str, v: &mut Vec<&str>) {  
    v.push(s);  
}
```

error: lifetime mismatch

```
1 | fn f(s: &str, v: &mut Vec<&str>) {  
  |           ----          ----  
  |           |  
  |           these two types are declared with different lifetimes...  
2 |     v.push(s);  
  |           ^ ...but data from `s` flows into `v` here
```

Riferimenti e funzioni

```
fn f<'a>(s: &'a str, v: &'a mut Vec<&str>) {  
    v.push(s);  
}
```

error: explicit lifetime required in the type of `v`

```
1 | fn f<'a>(s: &'a str, v: &'a mut Vec<&str>) {  
    |                                     ----- help: add explicit  
lifetime `a` to the type of `v`: `&'a mut Vec<&'a str>`  
  
2 |     v.push(s);  
    |           ^ lifetime `a` required
```

Riferimenti e funzioni

```
fn f<'a>(s: &'a str, v: &'a mut Vec<&'a str>) {  
    v.push(s);  
}
```

```
fn main() {  
    let mut v: Vec<&str> = Vec::new();  
    {  
        let s= String::from("abc");  
        v.push(&s);  
    }  
    println!("{:?}", v);  
}
```

error: `s` does not live long enough

```
9 |         v.push(&s);  
   |               ^^ borrowed value does not live long enough  
10 |     }  
   |     - `s` dropped here while still borrowed  
11 |     println!("{:?}", v);  
   |                   - borrow later used here
```

Riferimenti e funzioni

- Se una funzione ha due o più riferimenti in ingresso e un riferimento in uscita, il compilatore non è in grado di decidere (senza eseguire il codice) da quale parametro provenga il riferimento in uscita
 - L'inserimento degli identificatori nella firma della funzione risolve questa ambiguità

```
fn f<'b, 'c>(x: &'b S, y: &'c S) -> &'c u8 { ... }

let v1 = S(1);
let mut v2 = S(2);

let r = f(&v1, &v2); // Sappiamo che r è basato su v2,
                    // che rimane nello stato di prestito
                    // mentre v1 viene rilasciato e può evolvere
let v2 = v1;        // Questa operazione crea un problema!
print_byte(r);      // Qui finisce il prestito di r (e di v2)
```

Riferimenti e funzioni

- Lo scopo degli identificatori relativi al ciclo di vita è **duplice**:
 - Per il codice che **invoca** la funzione (chiamante), essi indicano su **quale**, tra gli indirizzi in ingresso, è basato il risultato in uscita
 - Per il codice **all'interno** della funzione (chiamato), essi garantiscono che vengano restituiti solo indirizzi cui **è lecito accedere** per (almeno) il tempo di vita indicato
- All'atto dell'invocazione, gli identificatori forniti dal programmatore (o inseriti automaticamente dal compilatore, quando possibile) sono legati all'effettivo intervallo minimo (espresso come insieme di linee di codice) nel quale il valore da cui il prestito è stato preso debba restare bloccato per non violare le assunzioni su cui la funzione è basata
 - Tentativi di modificare il valore originale da cui il prestito è preso prima che il tempo di vita sia trascorso portano ad errori di compilazione che costituiscono la base della robustezza del sistema di possesso e prestito offerto da Rust

Riferimenti e strutture dati

- Lo stesso tipo di ragionamento vale se il dato viene salvato all'interno di una qualsiasi struttura dati
 - Rust verifica che il valore a cui si punta abbia un tempo di vita maggiore o uguale del tempo di vita della struttura dati
 - Questo richiede di esplicitare il tempo di vita della struttura rispetto al tempo di vita dei riferimenti in essa contenuti
- Se una struttura che contiene riferimenti è contenuta, a sua volta, in un'altra struttura, anche quest'ultima deve avere il tempo di vita specificato in modo esplicito

```
struct User<'a> {  
    id: u32,  
    name: &'a str,  
}
```

```
struct Data<'a> {  
    user: User<'a>,  
    password: String,  
}
```

Elisione dei tempi di vita

- Rust, per default, provvede ad assegnare, ad ogni riferimento presente in una struttura dati o tra i parametri di una funzione, un tempo di vita distinto
 - Tali tempi di vita si propagano al codice che fa uso di tale struttura dati e, in assenza di ambiguità, non richiede di esplicitare gli identificatori dei tempi di vita

```
struct Point {  
    x: &i32,  
    y: &i32,  
}  
  
fun scale(r: &i32, p: Point) -> i32 {  
    r * (p.x * p.x + p.y * p.y)  
}
```

```
struct Point {  
    x: &'a i32,  
    y: &'b i32,  
}  
  
fun scale<'a,'b,'c> (r: &'a i32,  
                    p: Point<'b, 'c>) -> i32 {  
    r * (p.x * p.x + p.y * p.y)  
}
```

Elisione dei tempi di vita

- Se una funzione **restituisce un riferimento** o un tipo che contiene - direttamente o indirettamente - un riferimento, occorre disambiguare il tempo di vita del valore ritornato
 - Se c'è un solo parametro in ingresso dotato di tempo di vita, Rust assume che quello debba essere il tempo di vita del risultato
 - Se sono presenti più parametri in ingresso con tempo di vita, tocca al programmatore esplicitare quale debba essere il tempo di vita da associare al risultato
- Nel caso di metodi che accedono a **self** tramite un riferimento, Rust assume che il tempo di vita da associare al risultato **sia quello del riferimento a self**
 - Questo parte dal presupposto che se un metodo di un oggetto restituisce un dato preso a prestito (borrow), questo sia stato preso dai dati posseduti dall'oggetto stesso



Chiusure

Funzioni lambda

Funzioni e chiusure

- La programmazione in stile funzionale introduce il concetto di **funzioni di ordine superiore**
 - Funzioni i cui parametri e/o il cui tipo di ritorno sono a loro volta una funzione
- Questo richiede, ad un linguaggio, la possibilità di trattare una funzione come un tipo dati qualsiasi, consentendo di memorizzarla in una variabile
 - Nel linguaggio C questo è possibile modellando la variabile come un puntatore a funzione, usando la sintassi
TipoRitornato (* v) (p1: Tipo₁, ..., pn: Tipo_n);
che definisce **v** come puntatore ad una funzione che accetta n parametri e restituisce un valore dei tipi indicati
 - In C++, è possibile anche modellare la variabile come oggetto funzionale (ovvero che definisce il metodo ***operator() (...)*** e che può avere uno stato)
 - In Rust è possibile assegnare ad una variabile il puntatore ad una funzione (che avrà come tipo ***fn(T₁, ..., T_n) -> U***) o assegnare un valore che implementa un tratto funzionale: ***FnOnce, FnMut, Fn***

Puntatori a funzione

- Qualunque sia la natura del dato assegnato, si utilizza la variabile che contiene il puntatore come se fosse essa stessa una funzione
 - Sia il tipo di ritorno che tutti i tipi degli argomenti devono corrispondere a quanto dichiarato nella definizione della variabile (e della funzione)

```
double f1(int i, double d) {  
    return i * d;  
};
```

C++

```
double (*ptr)(int, double);
```

```
ptr = f1;    //identico a ptr = &f1;
```

```
ptr(2, 3.14); // restituisce 6.28
```

```
fn f1(i: i32, d: f64) -> f64 {  
    return i as f64 * d;  
}
```

Rust

```
let ptr: fn(i32, f64) -> f64;
```

```
ptr = f1; //assegno il puntatore
```

```
ptr(2, 3.14); // chiamo la funzione
```

Oggetti funzionali

- In C++ esiste un ulteriore tipo invocabile: il «funtore» o «oggetto funzionale»
 - Istanza di una qualsiasi classe che abbia ridefinito la funzione membro `operator()`

```
class FC {  
public:  
  
    int operator() (int v) {  
        return v*2;  
    }  
};
```

C++

```
{  
    FC fc;  
    int i= fc(5);  
    // i vale 10  
    i=fc(2);  
    // i vale 4  
}
```

C++

Oggetti funzionali

- È possibile includere più definizioni di `operator()`
 - Devono avere tipi differenti nell'elenco dei parametri, così da essere distinguibili
- Un oggetto funzionale può contenere variabili membro
 - Queste possono essere utilizzate all'interno delle funzioni `operator()` per tenere traccia di uno stato
 - Il comportamento non è più quello di una funzione pura (il cui output è sempre lo stesso a parità di input, come succede con le funzioni matematiche)
- Da un punto di vista dell'implementazione, il compilatore introduce, come per tutti i metodi, un ulteriore parametro nascosto (`this`) all'elenco dei parametri formali
 - Tale parametro fornisce l'accesso alla componente di stato dell'oggetto funzionale (le variabili membro dell'istanza della classe)

Oggetti funzionali

```
class Accumulatore {
    int totale;                //variabile membro
public:
    Accumulatore():totale(0){}
    int operator()(int v){
        totale += v;
        return v;
    }
    int totale() { return totale; }
};

void main() {
    Accumulatore a;            //istanza l'oggetto funzionale
    for (int i=0; i<10; i++)
        a(i);                  //invoca int operator() (int v)
    std::cout << a.totale() << std::endl; //stampa 45
}
```

C++

Funzioni lambda

- La notazione legata agli oggetti funzionali è particolarmente verbosa
 - Per questo motivo, i linguaggi moderni mettono a disposizione un concetto offerto dal paradigma funzionale: le funzioni lambda
- Una funzione lambda è una funzione anonima costituita da un blocco di codice espresso in forma letterale
 - Tale forma dipende dal linguaggio di programmazione ed è oggetto delle forme più disparate
- In **Javascript**, si utilizza la notazione freccia:
 - `const f = (v) => v + 1 // funzione che restituisce un valore incrementato`
- In **Kotlin** si racchiude l'espressione tra parentesi graffe:
 - `val f = { v: Int -> v + 1 }`
- In **C++** si usa una notazione ancora più complessa:
 - `auto f = [](int v) -> int { return v + 1; }`
- In **Rust** si racchiudono i parametri formali tra `|` `|`:
 - `let f = | v | { v + 1 }`

Funzioni lambda

- Una volta definita ed assegnata ad una variabile, una funzione lambda può essere invocata trattando la variabile come se fosse una funzione
 - Ovvero facendo seguire, al nome della variabile la lista degli argomenti racchiusi in parentesi tonde: ad esempio, `f(5);`
- E' possibile passare una funzione lambda come argomento di una funzione da invocare o utilizzare una funzione lambda come valore di ritorno di una funzione
 - La sintassi con cui si indica il tipo ritornato (una funzione che accetta certi tipi come parametri e restituisce un certo tipo di valore), a seconda dei linguaggi, può essere più o meno leggibile

C++

```
int (*ret_fun())(int) {  
    return [](int i) { return i+1; }  
}
```

Rust

```
fn ret_fun() -> fn(i32) -> i32 {  
    return |x|{ x+1 };  
}
```

Chiusure

- Il corpo di una funzione lambda può fare riferimento alle variabili che sono visibili nel contesto in cui è definita, acquisendone un riferimento, una copia o il possesso completo, in base al linguaggio e alla sintassi usata
 - Tali variabili, che compaiono nel corpo della funzione lambda sono dette **variabili libere**
- La funzione lambda così ottenuta viene detta **chiusura**
 - In quanto racchiude, al proprio interno, (una copia de) i valori catturati (quelli contenuti nelle variabili libere), rendendoli disponibili quando sarà successivamente invocata
- In C++, il compilatore trasforma una chiusura in un oggetto funzionale
 - Esso contiene, come variabili istanza, i valori delle variabili libere e definisce come `operator()` il corpo della funzione lambda
- In Rust, il compilatore trasforma una chiusura in una tupla
 - Avente tanti campi quante sono le variabili libere
 - Tale tupla implementa uno dei tratti funzionali previsti dal linguaggio: **FnOnce**, **FnMut**, **Fn**

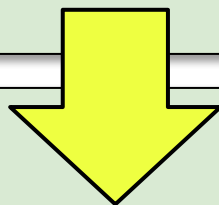
Cattura delle variabili in C++

- La notazione lambda è introdotta da una coppia di parentesi quadre
 - Al loro interno è possibile elencare variabili locali il cui valore o il cui riferimento si vuole rendere disponibili nella funzione
- Cattura per valore
 - `[x, y] (int i) { return (i-x) / (y-x); }`
 - Viene effettuata una copia dei valori all'interno dell'oggetto funzionale
 - La funzione λ potrà essere invocata anche quando tali variabili saranno uscite dallo scope
- Cattura per riferimento
 - `[&x, &y] (int i) { return (i-x) / (y-x); }`
 - Eventuali cambiamenti al contenuto delle variabili catturate, successivi alla creazione della funzione λ influenzano il comportamento della funzione
 - Attenzione a riferimenti pendenti!
- Cattura mista
 - `[x, &y] (int i) { return (i-x) / (y-x); }`
 - Viene catturato "x" per valore e "y" per riferimento

Cattura delle variabili in C++

```
{  
    int i = ...;  
    auto f = [i] (int v) { return v+i; };  
}
```

C++



```
{  
    int i = ...;  
    class __lambda_6_13 { //Nome univoco assegnato dal compilatore  
        int i;           //Variabile istanza catturata  
    public: inline int operator()(int v) const { //Firma della funzione  
        return v + i;    //Corpo della funzione  
    }  
    public: __lambda_6_13(int _i): i{_i}{} //Costruttore  
};  
__lambda_6_13 f = __lambda_6_13{i};  
}
```

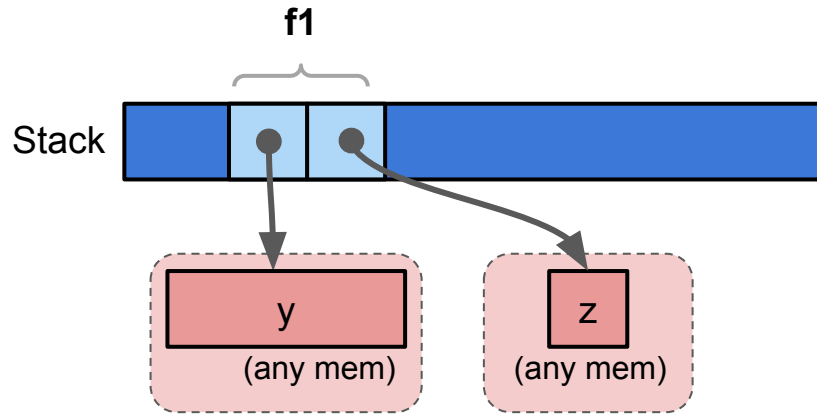
C++

Cattura delle variabili in Rust

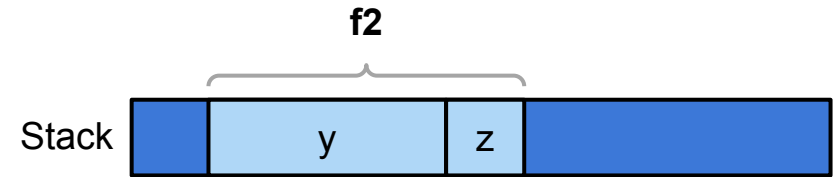
- Per default in Rust, tutte le variabili libere che compaiono nel corpo di una funzione lambda sono **catturate per riferimento**
 - Il compilatore, automaticamente, crea un prestito in lettura (&)
 - Se occorre modificare il contenuto delle variabili catturate (acquisendole con **&mut**), occorre dichiarare la funzione lambda come mutabile (**let mut f = |...| {...};**)
 - Il borrow checker, come al solito, verifica che tali riferimenti siano coerenti tra loro e con il tempo di vita dei valori cui si riferiscono
- Se occorre, è possibile indicare che la funzione lambda deve acquisire il **possesso dei valori** contenuti nelle variabili libere
 - Lo si fa anteponendo alla definizione della funzione lambda la parola chiave **move**
 - **let f = move |...| {...};**

Cattura delle variabili in Rust

```
let f1 = | x | { x + y.f() + z };
```



```
let f2 = move | x | { x + y.f() + z };
```



I tratti funzionali

- Rust definisce tre tratti funzionali che possono essere implementati **solo** tramite chiusure
 - Quale tratto venga implementato, dipende da **cosa** e **come** viene catturato

```
trait FnOnce<Args> {  
    type Output;  
    fn call_once(self, args: Args) -> Self::Output;  
}  
  
trait FnMut<Args>: FnOnce<Args> {  
    fn call_mut(&mut self, args: Args) -> Self::Output;  
}  
  
trait Fn<Args>: FnMut<Args> {  
    fn call(&self, args: Args) -> Self::Output;  
}
```

Tratti funzionali

- Una chiusura implementa il tratto **FnOnce<Args>** se consuma uno o più valori come parte della propria esecuzione
 - Pertanto, potrà essere invocata una sola volta

```
let range = 1..10;
let f = || range.count();
let n1 = f(); // 10
let n2 = f(); // Errore di compilazione: l'intervallo è stato consumato
```
- Una chiusura che implementa il tratto **FnMut<Args>** può essere invocata più volte, ma ha catturato una o più variabili in modo esclusivo (**&mut**)
 - Questo tipo di chiusura produce effetti collaterali ed ha pertanto uno **stato**
 - Esecuzioni successive con gli stessi parametri in ingresso possono dare risultati differenti

```
let mut sum = 0;
let mut f = |x| sum += x;
f(5); // ok, sum: 5
f(7); // ok, sum: 12
```

Tratti funzionali

- Una chiusura che implementa il tratto **Fn<Args>** si limita ad accedere in sola lettura alle variabili libere (&)
 - Finché la funzione esiste, le variabili libere sono in prestito condiviso e non possono essere cambiate
 - Pertanto, la chiusura può essere invocata un numero qualsiasi di volte e produce, a parità di argomenti, sempre lo stesso risultato

```
let s = "hello";  
let f = | v | v < s;  
f("world");    //false  
f("bye");      //true
```

<https://huonw.github.io/blog/2015/05/finding-closure-in-rust/>

Funzioni di ordine superiore

- E' possibile implementare una funzione che accetta come parametro una funzione lambda ricorrendo alla programmazione generica
 - Ogni funzione lambda forma infatti un tipo a sé, la cui unica istanza è la funzione lambda stessa
 - Tuttavia, esse implementano tutte almeno uno dei tratti funzionali: è quindi possibile definire una funzione generica che accetta come parametro di ingresso un'istanza del tipo F , soggetta al vincolo che tale tipo deve implementare uno dei tratti funzionali
- Quale specifico tratto richiedere dipende dalla natura del codice che si intende scrivere:
 - $\text{FnOnce}(\text{Args})$ accetterà qualsiasi chiusura, ma questa potrà essere invocata una sola volta
 - $\text{FnMut}(\text{Args})$ e $\text{Fn}(\text{Args})$ sono via via più restrittivi sulle operazioni che è lecito eseguire all'interno della funzione λ e più ampi nell'uso della funzione di ordine superiore

```
fn higher_order_function<F, T, U>(f: F) where F: Fn(T) -> U {  
    // ... codice che usa f(...)  
}
```


Funzioni di ordine superiore

- Analogamente, è possibile scrivere una funzione che restituisce una chiusura
 - Se la chiusura ritornata cattura qualche variabile, occorre fare attenzione al fatto che, normalmente, ciò implica memorizzare un riferimento all'interno della chiusura stessa
 - Questo fatto potrebbe imporre restrizioni sul tempo di vita del dato catturato non facilmente compatibili con il fatto che la chiusura ritornata deve sopravvivere alla funzione stessa (e quindi a tutte le sue variabili locali e ai suoi argomenti)
 - In questa situazione, si usa comunemente il modificatore **move** per trasferire il possesso di tali dati alla chiusura stessa
 - Può, inoltre, essere necessario richiedere che il dato catturato sia clonabile, se l'esecuzione della chiusura ritornata tendesse a consumare il dato e non si volesse ridurla ad **FnOnce<Args>**

```
fn function_generator<T>(v: T) -> impl Fn()->T where T: Clone {  
    return move || v.clone();  
}
```

Funzioni di ordine superiore

```
fn generator(prefix: &str) -> impl FnMut() -> String {  
    let mut i = 0;  
    let b = prefix.to_string();  
    return move || {i+=1; format!("{}",b,i)}  
}  
  
fn main() {  
    let mut f = generator("id_");  
    for _ in 1..5 {  
        println!("{}",f());  
    }  
}
```



id_1
id_2
id_3
id_4