02_3 LIFETIME

Un valore inzia ad esistere all'interno del blocco in cui è definitito e, se non viene mosso, cessa di esistere insieme al suo blocco. Se di questo valore recuperiamo il riferimento, il riferimento potrà esistere e essere acceduto in vari modi solo nell'intervallo di tempo in cui il valore, a cui il riferimento si riferisce, esiste.

Se recupero il **riferimento di un valore e lo passo ad una funzione**:

- Riferimento ha una durata — un solo identificatore `fn f<'a>(p1: &'a i32, p2:&'a i32) { ... }`
  - Internamente il compilatore si segna i vincoli legati al tempo di vita
- Se funzione riceve 2+ riferimenti, occorre che le azioni svolte dalla funzione siano compatibili ai tempi di vita dei vari riferimenti. — si usano etichette diverse `fn f<'a, 'b>(p1: &'a i32, p2:&'b i32) { ... }`

Quando la funzione generica viene usata, il compilatore costruisce delle versioni specifiche per i tipi con cui la funzione è stata utilizzata → **MONOMORFIZZAZIONE**.

Nota: gli indicatori del tempo di vita non partecipano alla monomorfizzazione, sono solo utili affinchè il borrow checker possa capire quali sono le **conseguenze dei puntatori** che stiamo passando e possa quindi verificare che non facciamo schifezze.

Se la **funzione ritorna un riferimento**, il compilatore non riesce a dedurre il tempo di vita. In questo caso è essenziale dire da dove è preso questo intero di cui è tornato il riferimento.

In questo caso specifichiamo che il tempo di vita è legato al secodo parametro → così il compilatore comprende quanto potrebbe durare questo prestito.

```
fn f<'a, 'b>(p1: &'a Foo, p2:&'b Bar) -> &'b i32 { /*…*/ return &p2.y; }
```

Se invece di distinguere a' e b', avessi a' e a' allora il tempo di vita sarebbe ridotto al minore dei due a'.

Se il compilatore si accorge che il risultato della funzione è usato, ci blocca.

*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*

Esempi:

```
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
```

Il problema è che la stringa potrebbe essere non più valida e il vettore lo sia ancora → per evitare questo problema bisogna garantire che la stringa viva più del vettore.

```
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
```

Gli sto dicendo che il vettore esiste per il tempo di quella stringa ma il compilatore vuole garanzie che il contenuto del vettore sia anche valido per tutto quel tempo di vita.

Modo corretto:

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

Ma **problema**:

```
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
```

Il vettore dura più di s, non si può mettere s dentro.

Nel caso in cui ci sono **due riferimenti in ingresso e uno solo in uscita**, **bisogna specificare** al compilatore a quale tempo di vita è legato il riferimento ritornato.

```
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)
```

Non posso accedere al risultato oltre la durata di v2.
Non posso modificare v2 fintanto che faccio accesso al risultato *(penultima riga da cancellare altrimenti non compila)*

Lo **scopo degli identificatori** relativi al ciclo di vita è duplice:

- X codice che **invoca la funzione** → indicano su quale indirizzo è basato il risultato in uscita
- X codice interno alla **funzione** → garantiscono che vengano restituiti solo indirizzi a cui è lecito accedere per il tempo di vita indicato.

Quando avviene l'invocazione, il compilatore verifica che gli identificatori che abbiamo passato esistano per almeno un tempo di vita minimo.

Strutture dati: se io salvo un riferimento con un tempo di vita all'interno di una struttura dati, questa struttura dati eredita il tempo di vita più breve tra quelli dei riferimenti che presenta al suo interno.

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

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

Nota: nel caso in cui una struttura diventa parte di una struttura più grande, la più grande sarà limitata dal tempo di vita di quella inserita al suo interno.

Elisione dei tempi di vita: se la funzione prende un solo riferimento in ingresso e un solo riferimento in uscita, il compilatore deduce in automatico il tempo di vita.

Se sono presenti più riferimenti e non si specifica il tempo di vita, il compilatore immagina che i tempi di vita siano indipendenti tra di loro →il programmatore dovrebbe specificare.

```
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)
}
```

Per non avere problemi:

Se una funzione restituisce un riferimento o un tipo che contiene un riferimento, **bisogna specificare** il tempo di vita del riferimento.
Se una funzione prendere mut sel o self tramite riferimento, per il compilatore il tempo di vità è quello del riferimento a self (Se non specificato). *(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)*

Nota aggiuntiva: esiste il **lifetime static** → riferimento che dura certamente quanto l'intero programma (es: costanti). Inoltre, se una variabile posside interamente il suo valore (non ha riferimenti) → quella variabile ha come tempo di vita static.

CHIUSURE *(funzione lambda)*

Nota: programmazione può essere di tipo imperativo (azioni hanno effetti collaterali che si propagano sulle nostre variabili) oppure funzionale (pone le radici in matematica → non ha effetti collaterali).

Una funzione mangia dei dati e produce un risultato:

- Operano su valori
- Operano su altre funzioni
  - Per fare ciò linguaggio deve permettere di salvare all'interno delle variabili una funzione.

Sui dati si possono fare diverse operazioni, mentre le funzioni posso semplicemente chiamarle.

In Rust è possibile assegnare ad una variabile il puntatore ad una funzione (che avrà come tipo `fn(T1, …, Tn) -> U`) o assegnare un valore che implementa un tratto funzionale: `FnOnce`, `FnMut`, `Fn`

Let f= alpha → f diventa un puntatore alla funzione alpha**, f è diventato un alias di alpha**; *volendo si può fare in modo che f possa assumere alpha o beta in base ad un evento → decide cosa salvare dentro f e indipendentemente chiamo f (alpha e beta devono avere lo stesso numero e tipo di parametri e ritorno).*

Ad una variabile, oltre che le funzioni**, posso asegnare dei tratti** che danno all'oggetto che gli implementa il comportamento di una funzione (FnOnce, FnMut, Fn)

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

double (*ptr)(int, double);

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

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

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

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

ptr = f1; //assegno il puntatore

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

A questo punto chiamare ptr o f1 è la stessa cosa

Posso assegnare un valore differente in base ad una condizione presente nel codice → a questo punto chiamo sempre la mia variabile e chiama la funzione desiderata.

C++

C++
```
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
}
```

Nota sul c++: quando introduco una classe, posso inserire tra i suoi metodi anche operator(). Creo una cosa che uso come fosse una funziona senza che sia una funzione.

All'interno della mia classe posso definire più operator per diversi tipi.

*Siccome è una classe, può avere delle **variabili membro** → possono essere usate per ricordare il comportamento della mia funzione → posso trasformare la mia funzione in impura: funzione con uno stato, rciorda la sua storia. Es: Accumulatore.*

***Nota**: se creo due oggetti di tipo accumulatore questi saranno distinti tra loro → posso avere più storie in contemporanea cosa che non sarebbe possibile usando variabili globali.*

Posso anche usare il mio stato per cambiare il valore di ritorno.

C++
```
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
}
```

- 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 }`

Le funzioni lambda non hanno un nome, lo prendono dalla variabile a cui sono assegnate.

Funzioni lambda

Una volta definita ed assegnata ad una variabile, può essere invocata trattao la variabile come fosse una funzione. Inoltre, una funzione lambda può essere passaa come argomento ad una funzione o ritornata da una funzione.

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

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

Nel corpo di questa funzione, possono essere citate delle variabili che siano visibili nel punto in cui la definizione avviene → congeliamo nel corpo della nostra funzione dei valori che prendiamo attorno a noi (nel punto in cui siao stati definiti) → la lambda acquisisce questi valori e rimangono con lei. → VARIABILI LIBERE

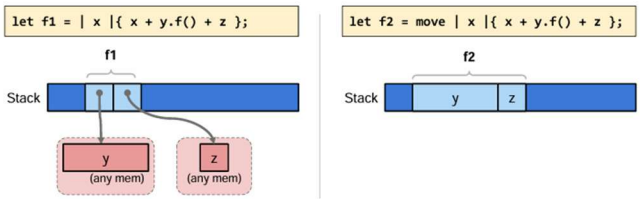
In Rust il compilatore **trasforma una chiusura in una tupla**

- Avente tanti campi quante sono le variabili libere
- Tupla che implementa uno dei tratti funzioni previsti da Rust: FnOnce, FnMut, Fn

--codice rust

Cattura delle variabili in Rust:

Di default viene inserito il riferimento semplice al dato, nel caso in cui si voglia che il dato originale abbia un tempo di vita diverso, bisogna governare in un'altra maniera → se si aggiugne la parola chiave **move**, invece di avere il reference si prende proprio la variabile → si **acquisisce il possesso alla variabile**.



Esistono **3 tratti FUNZIONALI**: *(quelli più sotto hanno bisogno dell'implementazione di quelli più sopra)*

- **Fn once** → quando la chiusura verrà chiamata distruggerà un suo pezzo; dunque, è chiamabile una volta sola
  - Se consuma 1o+ valori nella sua esecuzione
  - Es: se chiamo self, prende possesso e restituisce altro
- **Fn mut** → ha la possibilità di essere chiamato più volte e ha la possibilità di mdoificare se stesso
  - Cattura in modo esclusivo 1+ variabili
  - Es: metodo che prende &mut self
  -
- **Fn** → può essere chiamate infinite volte senza effetti particolari
  - Accedo in sola vettura alle variabili libere o se ne prendo possesso
    - *Si guarda ma non si modifica (come se si fa &self)*

```
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;
}
```

- 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
```

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
```

Funzioni di ordine superiore

Posso implementare funzioni che accettino come parametro altre funzioni. Posso passare oggetti T che implementa il tratto Fn.

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

- Le funzioni lambda implementano almeno un tratto funzionale:
  - FnOnce(Args) accettetterà qualsiasi chiusura, ma questa potrà essere invocata una sola volta
  - FnMut(Args) e Fn(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

--codice

Slide non fatte:

- 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() };
}
```

```
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
```