



# 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 senza il bisogno dell'intervento del programmatore (*lifetime elision*)

# 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
- 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
  - Occorre annotare il tipo restituito con la corretta etichetta
- 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) -> &'a i32 { ... }`
  - Nel secondo si usano etichette diverse `fn f<'a, 'b>(p1: &'a i32, p2:&'b i32) -> &'a i32 { ... }`

```
fn confronta(str1: &str, str2: &str) -> &str {
    if str1.len() > str2.len() { str1 } else { str2 }
}

fn main() {
    let s1 = String::from("hello");
    let s2 = String::from("world");
    let risultato;
    risultato = confronta(&s1, &s2);

    println!("La stringa più lunga è: {}", risultato);
}
```

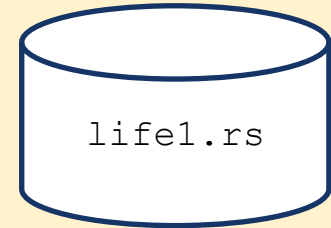


```
1 | fn confronta(str1: &str, str2: &str) -> &str {
  |               ----          ^ expected named lifetime parameter
  |
= help: this function's return type contains a borrowed value, but the
signature does not say whether it is borrowed from `str1` or `str2`
help: consider introducing a named lifetime parameter

1 | fn confronta<'a>(str1: &'a str, str2: &'a str) -> &'a str {
  |               +++++          ++          ++          ++
```

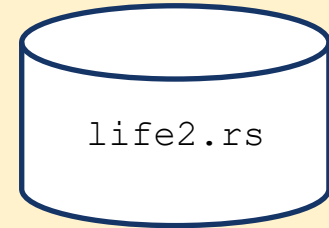
- Le annotazioni devono essere riportate solo nella signature della funzione e non nel corpo della funzione.

```
fn confronta<'a> (str1: &'a str, str2: &'a str) -> &'a str {  
    if str1.len() > str2.len() {  
        str1  
    } else {  
        str2  
    }  
}  
  
fn main() {  
    let s1 = String::from("hello");  
    let s2 = String::from("world!");  
    let risultato;  
    risultato = confronta(&s1, &s2);  
  
    println!("La stringa più lunga è: {}", risultato);  
}
```



- Se il riferimento restituito dipende solo da un parametro, occorre specificare l'annotazione solo per il parametro per cui c'è la dipendenza.

```
fn stampa<'a> (str1: &'a str, str2: &str) -> &'a str {  
    println!("{}",str2);  
    str1  
}
```



```
fn main() {  
    let s1 = String::from("Viva i lifetimes");  
    let s2 = String::from("Questa è una stringa di benvenuto");  
  
    let risultato;  
    risultato = stampa(&s1, &s2);  
  
    println!("{}", risultato);  
}
```

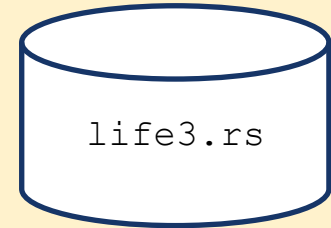
- Annotazione equivalente alla precedente

```
fn stampa<'a, 'b> (str1: &'a str, str2: &'b str) -> &'a str {  
    println!("{}",str2);  
    str1  
}  
  
fn main() {  
    let s1 = String::from("Viva i lifetimes");  
    let s2 = String::from("Questa è una stringa di benvenuto");  
  
    let risultato;  
    risultato = stampa(&s1, &s2);  
  
    println!("{}", risultato);  
}
```



- Nel caso in cui la funzione sia generica, le meta-variabili di tipo vengono indicate dopo gli identificatori del tempo di vita.

```
fn merge_slices<'a, T>(first: &'a [T], second: &'a [T]) -> Vec<&'a T>
    where T: Copy {
    let mut merged: Vec<&'a T> = Vec::new();
    for item in first.iter().chain(second.iter()) {
        merged.push(item);
    }
    merged
}
```



```
fn main() {
    let first_slice = &[1, 2, 3, 4];
    let second_slice = &[5, 6, 7, 8];

    let merged_result = merge_slices(first_slice, second_slice);

    println!("Merged slices: {:?}", merged_result);
}
```



```
struct S(u8);

fn f<'a>(x: &S, y: &'a S) -> &'a u8 {
    &y.0
}

fn print_byte(byte: &u8) {
    println!("Byte: {}", byte);
}

fn main() {
    let v1 = S(1);
    let mut v2 = S(2);

    let r = f(&v1, &v2);

    v2 = v1;

    print_byte(r);
}
```



```
struct S(u8);
```

```
fn f<'a>(x: &S, y: &'a S) -> &'a u8 {  
    &y.0  
}
```

```
fn print_byte(byte: &u8) {  
    println!("Byte: {}", byte);  
}
```

```
fn main() {  
    let v1 = S(1);  
    let mut v2 = S(2);  
  
    let r = f(&v1, &v2);  
  
    v2 = v1;  
  
    print_byte(r);  
}
```



```
15 |         let r = f(&v1, &v2);  
    |                                     --- `v2` is borrowed here  
  
16 |  
17 |         v2 = v1;  
    |         ^^^^^^^ `v2` is assigned to here but it was  
    |         already borrowed  
18 |  
19 |         print_byte(r);  
    |                       - borrow later used here
```

```
struct S(u8);

fn f<'a>(x: &S, y: &'a S) -> &'a u8 {
    &y.0
}

fn print_byte(byte: &u8) {
    println!("Byte: {}", byte);
}

fn main() {
    let v1 = S(1);
    let v2 = S(2);

    let r = f(&v1, &v2);

    let v2 = v1;

    print_byte(r);
}
```



```
struct S(u8);

fn f<'a>(x: &S, y: &'a S) -> &'a u8 {
    &y.0
}

fn print_byte(byte: &u8) {
    println!("Byte: {}", byte);
}

fn main() {
    let v1 = S(1);
    let mut v2 = S(2);

    let r = f(&v1, &v2);
    print_byte(r);
    v2 = v1;
}
```



# 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 funzioni

- 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`

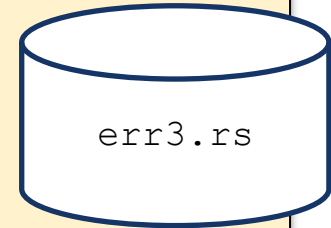
```
1 | fn f<'a>(s: &'a str, v: &'a mut Vec<&str>) {  
    |                                     ----- help: add explicit  
lifetime `a` to the type of `v`: `&'a mut Vec<&'a str>`  
  
2 |     v.push(s);  
    |           ^ lifetime `a` required
```



# 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);  
}
```

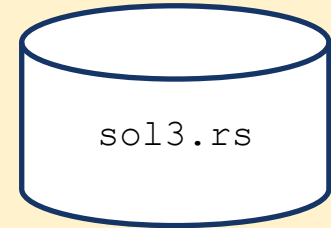


error: `s` does not live long enough

```
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

```
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);  
    }  
}
```



# 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

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

```
struct Contenitore<'a> {  
    dati: &'a str,  
}
```

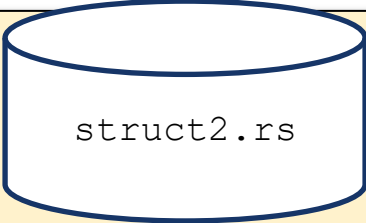
```
fn main() {  
    let dati = String::from("Dati importanti");  
    let cont = Contenitore { dati: &dati };  
    println!("{}", cont.dati);  
}
```



struct1.rs

```
struct Contenitore<'a> {  
    dati_esteri: &'a str,  
    dati_interni: String,  
}
```

```
fn main() {  
    let esterno = String::from("dati esterni");  
    let interno = String::from("dati interni");  
    let cont = Contenitore { dati_esteri: &esterno, dati_interni: interno };  
    println!("Dati esterni: {}, Dati interni: {}", cont.dati_esteri,  
cont.dati_interni);  
}
```



struct2.rs

- Il lifetime deve essere specificato per garantire che il riferimento content sia valido e non punti a una memoria non allocata.

```
struct TextWindow<'a> {  
    content: &'a str,  
}  
impl<'a> TextWindow<'a> {  
    fn new(content: &'a str) -> Self {  
        TextWindow { content }  
    }  
    fn display(&self) {  
        println!("Text window content: {}", self.content);  
    }  
}  
fn main() {  
    let text_window;  
    {  
        let my_text = "Hello, world!";  
        text_window = TextWindow::new(my_text);  
    }  
    text_window.display();  
}
```



struct3.rs

# 'static

- Se la funzione restituisce un riferimento ad una variabile locale che contiene una stringa costante, bisogna riportare l'annotazione di lifetime `'static` che indica che il riferimento restituito ha una durata statica, cioè per l'intera durata del programma.

```
fn crea_stringa() -> &'static str {  
    let s = "hello";  
    s  
}
```

```
fn main() {  
    let s = crea_stringa();  
    println!("{}", s);  
}
```



# Riferimenti e strutture dati

- 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,  
}
```

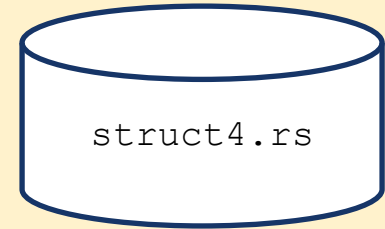


```
#[derive(Debug)]
struct User<'a> {
    id: u32,
    name: &'a str,
}

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

fn main() {
    let user = User { id: 1, name: "Alice" };
    let data = Data { user, password: String::from("password123") };

    println!("Data: {:?}, {:?}", data.user, data.password);
}
```



# 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

# Regole del compilatore

1. Ogni parametro che è un riferimento ha il suo parametro di lifetime:
  - Una funzione con un parametro ha un parametro di lifetime ('a)
  - Una funzione con due parametri ha due parametri di lifetime ('a e 'b)
2. Se c'è un solo parametro di lifetime in input, quel parametro è assegnato a tutti i parametri di lifetime di output
3. Se ci sono più parametri di lifetime di ingresso, ma uno di loro è &self o &mut self (poiché si tratta di un metodo) il parametro di lifetime di self è assegnato a tutti i parametri di lifetime di output
  - 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

```
fn trova_primo(s: &str, target: char) -> Option<&str> {  
    for (i, c) in s.chars().enumerate() {  
        if c == target {  
            return Some(&s[..i]);  
        }  
    }  
    None  
}  
  
fn main() {  
    let s = String::from("hello");  
    let result;  
    {  
        let r = trova_primo(&s, 'l');  
        result = r.unwrap();  
    }  
    println!("{}", result); // Stampa "he  
}
```



```
fn get_first_element(arr: &[i32]) -> &i32 {  
    if arr.is_empty() {  
        panic!("Array is empty");  
    }  
    &arr[0]  
}
```

```
fn main() {  
    let array = [1, 2, 3, 4];  
    println!("First element: {}", get_first_element(&array));  
}
```



```
fn get_element<T>(data: & [T], index: usize) -> Option<&T> {  
    data.get(index)  
}  
  
fn main() {  
    let data = &[1, 2, 3, 4, 5];  
    let index = 2;  
  
    match get_element(data, index) {  
        Some(element) => println!("Element at index {}: {}", index, element),  
        None => println!("Index out of bounds"),  
    }  
}
```



```
struct Example {  
    data: i32,  
}  
  
impl Example {  
    fn get_data_ref(&self) -> &i32 {  
        &self.data  
    }  
}  
  
fn main() {  
    let ex = Example { data: 42 };  
    let data_ref = ex.get_data_ref();  
    println!("Data reference: {}", data_ref);  
}
```



```
struct Example {  
    data1: i32,  
    data2: i32,  
}  
impl Example {  
fn get_data_ref(&self, other: &i32) -> &i32 {  
    if self.data1 > *other {  
        &self.data1  
    } else {  
        &self.data2  
    }  
}  
}  
fn main() {  
    let ex = Example { data1: 42, data2: 20 };  
    let other_data = 30;  
    let data_ref = ex.get_data_ref(&other_data);  
    println!("Data reference: {}", data_ref);  
}
```





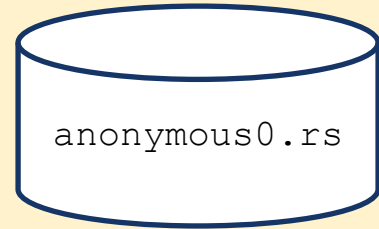
```
struct Example {  
    data1: i32,  
    data2: i32,  
}  
impl Example {  
fn get_data_ref<'a>(&'a self, other: &'a i32) -> &'a i32 {  
    if self.data1 > *other {  
        &self.data1  
    } else {  
        &other  
    }  
}  
}  
fn main() {  
    let ex = Example { data1: 42, data2: 20 };  
    let other_data = 30;  
    let data_ref = ex.get_data_ref(&other_data);  
    println!("Data reference: {}", data_ref);  
}
```



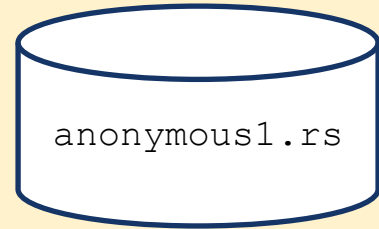
# Anonymous Lifetime

- Nel caso di riferimenti inseriti in strutture date, se la definizione del metodo permette di ricavare in modo implicito la dipendenza dei temi di vita, si può utilizzare l'annotazione di *anonymous lifetime* (<'\_>).

```
struct Persona<'a> {  
    nome: &'a str,  
}  
  
impl Persona<'_> {  
    fn saluta(&self) {  
        println!("Ciao, sono {}", self.nome);  
    }  
}  
  
fn main() {  
    let nome = String::from("Mario");  
    let persona = Persona { nome: &nome };  
    persona.saluta();  
}
```



```
struct Worker<'a> {  
    name: &'a str,  
    id: u32,  
}  
  
impl Worker<'_> {  
    fn new(name: &str, id: u32) -> Worker{  
        Worker { name, id }  
    }  
  
    fn get_name(&self) -> &str {  
        self.name  
    }  
}  
  
fn main() {  
    let name = "Alice";  
    let id = 1001;  
  
    let worker = Worker::new(name, id);  
  
    println!("Worker name: {}", worker.get_name());  
}
```

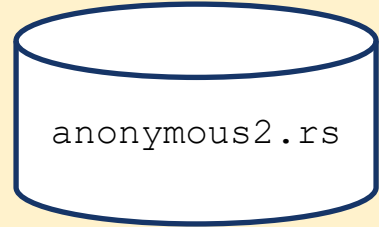


```
trait Biography{  
    fn biography(&self) -> & str;  
}
```

```
struct Person<'a> {  
    name: &'a str,  
    age: u32,  
    bio: &'a str,  
}
```

```
impl Biography for Person<'_> {  
    fn biography(&self) -> & str {  
        self.bio  
    }  
}
```

```
fn main() {  
    let name = "Alice";  
    let age = 30;  
    let bio = format!("{}", name, age);  
    let person = Person { name, age, bio: &bio };  
    println!("Biography: {}", person.biography());  
}
```

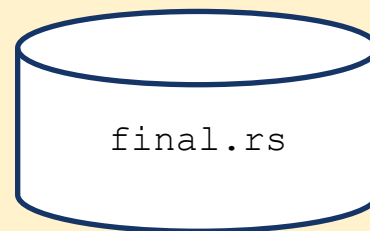


# Trait Bound, Generic Type e Lifetimes

```
use std::fmt::Display;

fn longest_with_an_announcement<'a, T>(x: &'a str, y: &'a str, ann: T) -> &'a str
    where
        T: Display,
{
    println!("Announcement! {}", ann);
    if x.len() > y.len() {
        x
    } else {
        y
    }
}

fn main() {
    let x = "hello";
    let y = "world!";
    let ann = "Important message";
    let longest_str = longest_with_an_announcement(x, y, ann);
    println!("The longest string is: {}", longest_str);
}
```



# Per saperne di più



- Diving Deep: How Rust Lifetimes Work
  - <https://medium.com/@luishrsoares/diving-deep-how-rust-lifetimes-work-2692633ab344>
  - Articolo di approfondimento sui lifetimes