



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) {    C++  
    return i * d;  
};  
  
double (*ptr)(int, double);  
  
ptr = f1; //identico a ptr = &f1;  
  
ptr(2, 3.14); // restituisce 6.28
```

```
fn f1(i: i32, d: f64) -> f64 { Rust  
    return i as f64 * d;  
}  
  
let ptr: fn(i32, f64) -> f64;  
  
ptr = f1; //assegno il puntatore  
  
let num = ptr(2, 3.14);  
// chiamo la funzione
```

```
fn add(a: i32, b: i32) -> i32 { a + b }
fn subtract(a: i32, b: i32) -> i32 { a - b }
fn multiply(a: i32, b: i32) -> i32 { a * b }

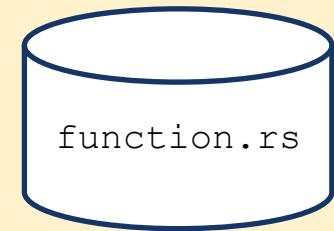
fn main() {
    let (x, y) = (10, 5);

    // Selezioniamo la funzione in base a una condizione.
    let operation_name = "add";

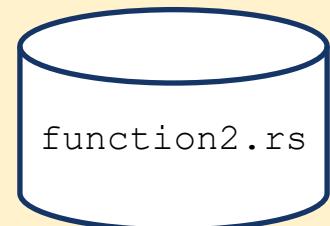
    // Assegnamo il puntatore alla funzione corrispondente.
    let operation: fn(i32, i32) -> i32 = match operation_name {
        "add" => add,
        "subtract" => subtract,
        "multiply" => multiply,
        _ => panic!("Operazione non supportata"),
    };

    let result = operation(x, y);

    println!("Il risultato dell'operazione '{}' è: {}", operation_name, result);
}
```



```
fn add_one(x: i32) -> i32 {  
    x + 1  
}  
  
fn do_twice(f: fn(i32) -> i32, arg: i32) -> i32 {  
    f(arg) + f(arg)  
}  
  
fn main() {  
    let answer = do_twice(add_one, 5);  
    println!("The answer is: {}", answer);  
}
```



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**

Terminologia



- **Funzioni di ordine superiore:** funzioni i cui parametri e/o il cui tipo di ritorno sono a loro volta una funzione
- **Funzione anonima:** funzione definita senza un nome e il loro uso è diretto (ossia la dichiarazione viene scritta dentro il codice)
 - Sintassi più concisa: il compilatore è in grado di inferire (ossia di riconoscere) i tipi degli argomenti e del valore di ritorno
- **Funzione lambda:** sinonimo di funzione anonima, ossia rappresenta la funzionalità di RUST di implementare funzioni anonime
- **Chiusura:** Funzione lambda (ossia anonima) che può catturare variabili (detti **parametri liberi**) dell'ambiente circostante
- Le variabili libere sono catturate:
 - per riferimento (default)
 - per movimento (utilizzando la keyword `move`)

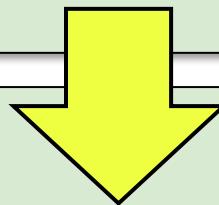
Cattura delle variabili in C++

- 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};  
}
```

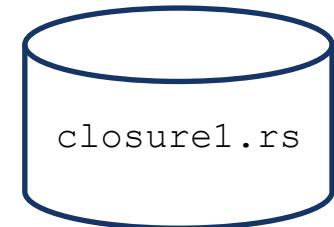
C++

Cattura delle variabili in Rust

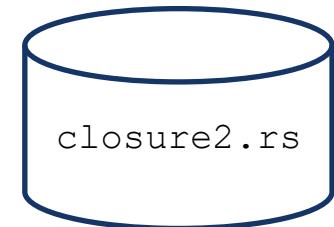
- Per default in Rust, tutte le variabili libere che compaiono nel corpo di una funzione lambda sono **catturate per riferimento**
 - Il compilatore, automaticamente, crea un prestito in lettura (`&`)
 - Se occorre modificare il contenuto delle variabili catturate (acquisendole con `&mut`), occorre dichiarare 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 |...| {...};`

Esempi di Chiusure

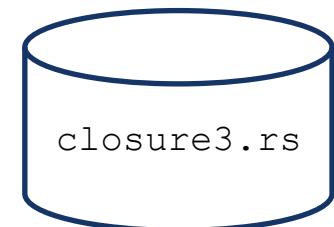
```
fn main() {  
    let add = |num1, num2| num1 + num2;  
    println!("La somma è: {}", add(1, 2));  
}
```



```
fn main() {  
    let factor = 2;  
    let multiply = |n| n * factor;  
    println!("Il risultato è: {}", multiply(5));  
}
```

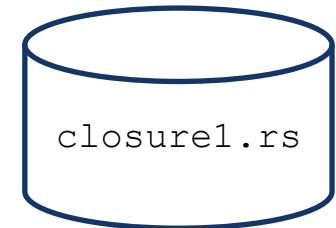


```
fn main() {  
    let multiply = |x: i32, y: i32| -> i64 {  
        (x * y).into() };  
    println!("{}", multiply(3, 4)); // prints 12  
}
```

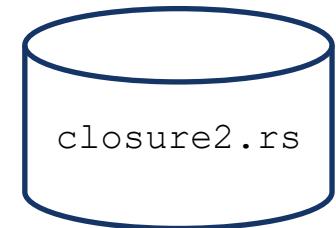


Quali sono le variabili libere?

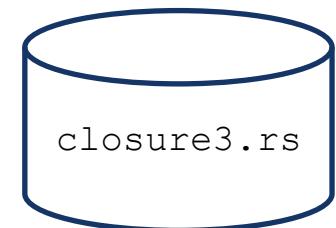
```
fn main() {  
    let add = |num1, num2| num1 + num2;  
    println!("La somma è: {}", add(1, 2));  
}
```



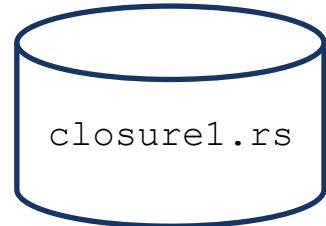
```
fn main() {  
    let factor = 2;  
    let multiply = |n| n * factor;  
    println!("Il risultato è: {}", multiply(5));  
}
```



```
fn main() {  
    let multiply = |x: i32, y: i32| -> i64 {  
        (x * y).into() };  
    println!("{}", multiply(3, 4)); // prints 12  
}
```

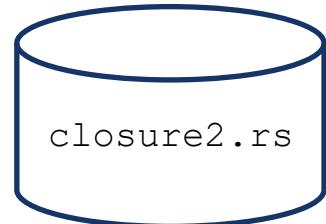


```
fn main() {  
    let add = |num1, num2| num1 + num2;  
    println!("La somma è: {}", add(1, 2));  
}
```

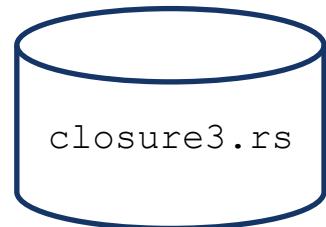


```
fn main() {  
    let factor = 2;  
    let multiply = |n| n * factor;  
    println!("Il risultato è: {}", multiply(5));  
}
```

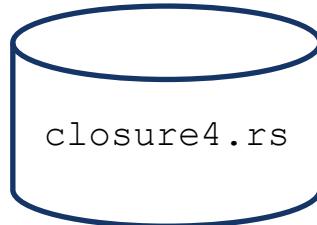
Variabile libera
catturata



```
fn main() {  
    let multiply = |x: i32, y: i32| -> i64 {  
        (x * y).into() };  
    println!("{}", multiply(3, 4)); // prints 12  
}
```



```
fn main() {  
    let mut numbers = vec![5, 2, 9, 1, 7];  
  
    numbers.sort_by(|a, b| a.cmp(b));  
  
    println!("Sorted numbers: {:?}", numbers);  
}
```

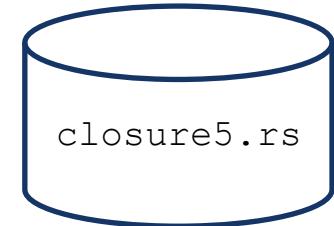


```
#[derive(Debug, PartialEq, PartialOrd)]
struct Person {
    name: String,
    age: u8,
}

fn main() {
    let mut people = vec![
        Person { name: String::from("Alice"), age: 30 },
        Person { name: String::from("Bob"), age: 24 },
        Person { name: String::from("Carol"), age: 29 },
    ];

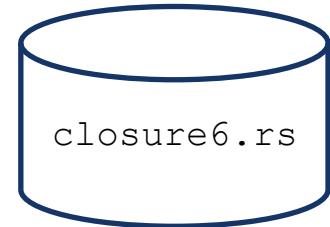
    // Ordina le persone per età usando una chiusura
    people.sort_by(|a, b| a.age.cmp(&b.age));

    println!("{:?}", people);
}
```



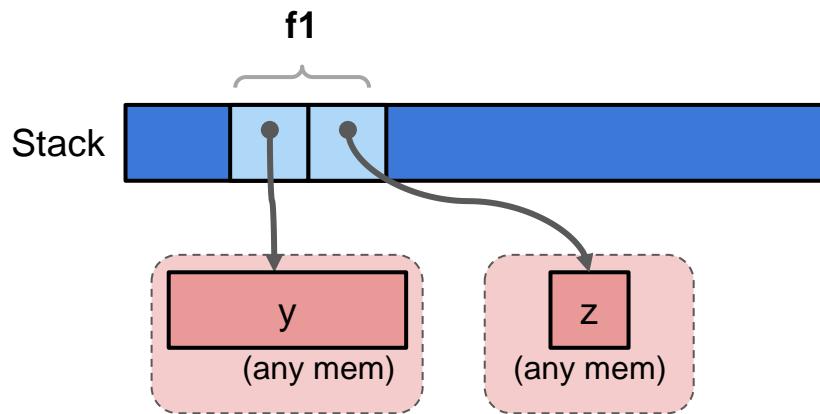
closure5.rs

```
fn main() {  
    let numbers = vec![1, 2, 3, 4, 5, 6];  
    let even_numbers: Vec<_> =  
numbers.into_iter().filter(|&x| x % 2 == 0).collect();  
    println!("{}:?", even_numbers); // Stampa: [2, 4, 6]  
}
```

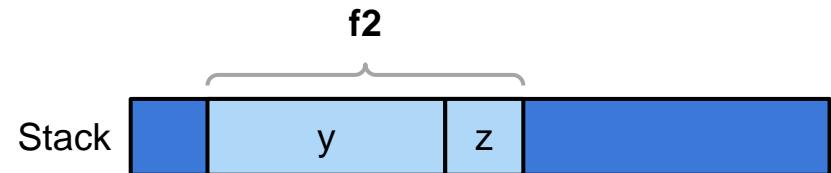


Cattura delle variabili in Rust

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



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



```
fn main() {  
    let mut count = 0;  
  
    let mut increment = move || {  
        count += 1; // Incrementiamo la variabile catturata  
        println!("Il conteggio è: {}", count);  
    };  
  
    increment();  
  
    increment();  
}
```



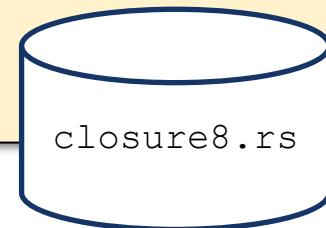
```
fn main() {  
  
    let mut count = 0;  
  
    let mut increment = move || {  
        count += 1; // Incrementiamo la variabile catturata  
        println!("Il conteggio è: {}", count);  
    };  
  
    increment();  
  
    increment();  
    println!("Hello, {}", count);  
}  
}
```

Il conteggio è: 1
Il conteggio è: 2
Hello, 0

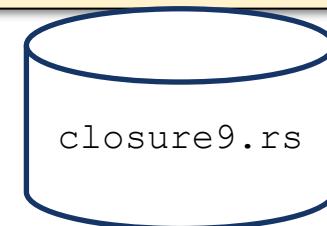


Come mai funziona così?

```
fn main() {  
    let numbers = vec![1, 2, 3, 4, 5];  
  
    let print_numbers = move || {  
        println!("I numeri sono: {:?}", numbers);  
        // La chiusura usa il vettore catturato  
        // numbers è stato spostato (moved) dentro la chiusura  
    };  
  
    // non può più essere usato qui  
    //println!("I numeri sono: {:?}", numbers);  
    // Questo darebbe errore di compilazione  
  
    print_numbers(); // Chiamiamo la chiusura  
}
```



```
fn main() {  
    let mut count = 0;  
  
    let mut increment_n = |n| {  
        count += n; // Incrementiamo la variabile catturata  
        println!("Il conteggio è: {}", count);  
    };  
  
    increment_n(10); // Stampa: Il conteggio è: 10  
  
    increment_n(5); // Stampa: Il conteggio è: 15  
  
}
```

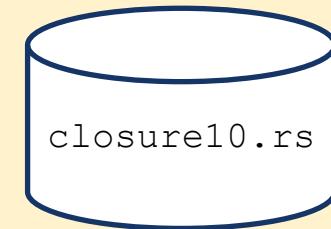


```
fn main() {  
    let mut count = 0;  
  
    let mut increment_n = |n| {  
        count += n; // Incrementiamo la variabile catturata  
        println!("Il conteggio è: {}", count);  
    };  
  
    increment_n(10);  
    println!("{}", count);  
    increment_n(5);  
    println!("{}", count);  
}
```



Il compilatore non
compila!
Come mai?

```
fn main() {  
    let mut count = 0;  
  
    let mut increment_n = |n| {  
        count += n; // Incrementiamo la variabile catturata  
        println!("Il conteggio è: {}", count);  
    };  
  
    increment_n(10);  
    increment_n(5);  
    println!("{}", count);  
}
```



Questo lo compila!

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

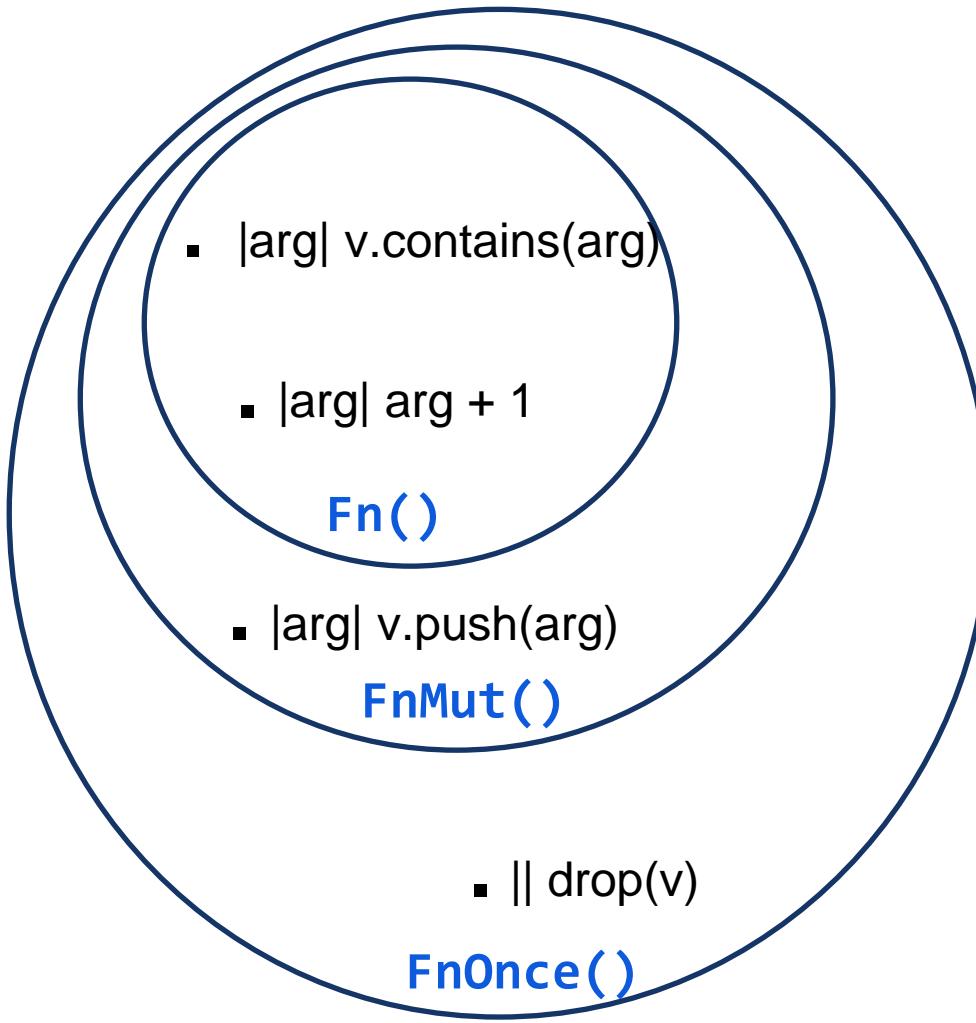
```
trait FnOnce<Args> {
    type Output;
    fn call_once(self, args: Args) -> Self::Output;
}

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

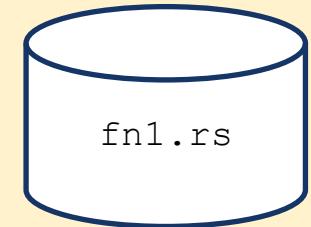
trait Fn<Args>: FnMut<Args> {
    fn call(&self, args: Args) -> Self::Output;
}
```

Tratti funzionali

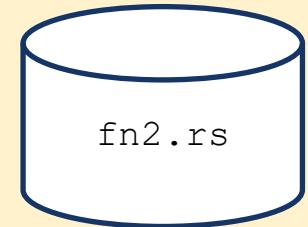
- Una chiusura che implementa il tratto **Fn<Args>** si limita ad accedere in sola lettura alle variabili libere (**&**)
 - Finché la funzione esiste, le variabili libere sono in prestito condiviso e non possono essere cambiate
 - Pertanto, la chiusura può essere invocata un numero qualsiasi di volte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



```
fn main() {  
    let greeting = "Ciao";  
  
    // Definiamo una closure che non modifica variabili esterne  
    let greet = || {  
        println!("{}", greeting);  
    };  
  
    // Chiamiamo la closure  
    greet();  
    greet();  
    greet();  
}  
}
```



```
fn main() {  
    // Una chiusura che accetta un parametro e implementa il tratto Fn  
    let mut x = 10;  
  
    let aggiungi_a_x = |y: i32| {  
        println!("Il risultato di x + y è: {}", x + y);  
    };  
  
    aggiungi_a_x(5); // Chiamiamo la chiusura con il parametro 5  
    aggiungi_a_x(7); // Chiamiamo la chiusura con il parametro 7  
    aggiungi_a_x(7); // Chiamiamo la chiusura con il parametro 7  
}
```

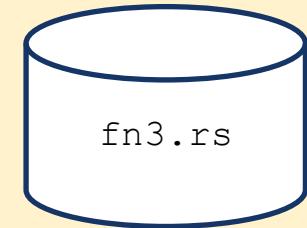


```
fn main() {
    let base_number = 10;

    // Definiamo una closure che non modifica variabili esterne
    let square = |x: i32| {
        let result = x * x;
        println!("Il quadrato di {} è: {}", x, result);
    };

    square(base_number);
    square(5);

}
```

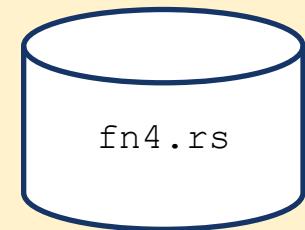


```
fn main() {
    let data = vec![1, 2, 3, 4, 5];

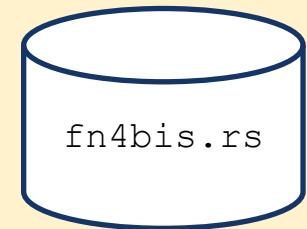
    // Definiamo una closure che può essere chiamata solo una volta
    let process_data = move || {
        println!("Elaborazione dei dati in corso...");
        let sum: i32 = data.iter().sum();
        println!("La somma dei dati è: {}", sum);
    };

    // Chiamiamo la closure
    process_data();
}

}
```



```
fn main() {  
    let data = vec![1, 2, 3, 4, 5];  
  
    // Definiamo una closure che può essere chiamata solo una volta  
    let process_data = move || {  
        println!("Elaborazione dei dati in corso...");  
        let sum: i32 = data.iter().sum();  
        println!("La somma dei dati è: {}", sum);  
    };  
  
    // Chiamiamo la closure  
    process_data();  
  
    process_data();  
}
```



```
fn main() {
    let mut data = vec![1, 2, 3, 4, 5];

    // Definiamo una closure che può essere chiamata solo una volta
    let mut process_data = move || {
        data.push(6);
        println!("Elaborazione dei dati in corso...");
        let sum: i32 = data.iter().sum();
        println!("La somma dei dati è: {}", sum);
    };

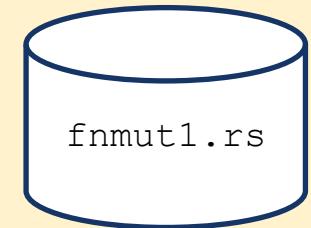
    // Chiamiamo la closure
    process_data();
    data.push(7);
}
```



```
fn main() {
    let mut counter = 0;

    // Definiamo una closure che può essere chiamata più volte
    let mut increment_counter = || {
        counter += 1;
        println!("Il contatore è ora: {}", counter);
    };

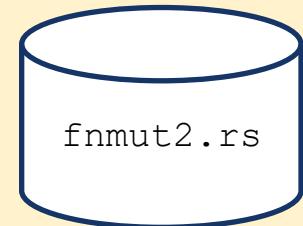
    // Chiamiamo la closure più volte
    increment_counter();
    increment_counter();
    increment_counter();
}
```



```
fn main() {
    let mut x = 10;

    // Una chiusura che implementa il tratto FnMut e accetta un parametro
    let mut aggiungi_a_x = |y: i32| {
        x += y;
        println!("Il nuovo valore di x è: {}", x);
    };

    aggiungi_a_x(5); // Chiamiamo la chiusura con il parametro 5
    aggiungi_a_x(7); // Chiamiamo la chiusura con il parametro 7
}
```

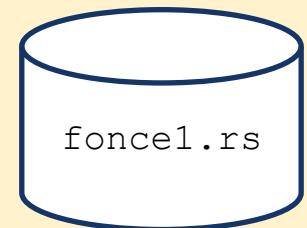


```
fn main() {
    let vec = vec![1, 2, 3, 4, 5];

    // Definiamo una chiusura che prende il possesso del vettore vec
    let consume_vector = move || {
        // Consumiamo il vettore stampando tutti i suoi elementi
        for num in vec {
            println!("{}", num);
        }
    };

    // Chiamiamo la chiusura. Dopo questa chiamata, non possiamo più usarla.
    consume_vector();

    // Tentare di chiamare nuovamente la chiusura genererebbe un errore di
    // compilazione.
    // consume_vector();
}
```



Si veda il programma
fn4.rs

```
fn main() {
    let mut data = vec![1, 2, 3, 4, 5];

    // Definiamo una closure che può essere chiamata solo una volta
    let mut process_data = move || {
        println!("Elaborazione dei dati in corso...");
        let sum: i32 = data.iter().sum();
        println!("La somma dei dati è: {}", sum);
        drop(data);
        // Consumiamo i dati per evitare ulteriori elaborazioni
    };

    // Chiamiamo la closure
    process_data();

    // Tentiamo di chiamare nuovamente la closure
    // Questo produrrà un errore di compilazione perché la closure è stata consumata
    //process_data();
}
```



```
fn main() {  
    let range = 1..10;  
    let f = || range.count();  
    let n1 = f();  
    let n2 = f();  
}
```



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:
 - 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

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

```
fn consume_closure<F>(f: F)
    where
        F: FnOnce() -> String,
{
    println!("La closure dice: {}", f());
}

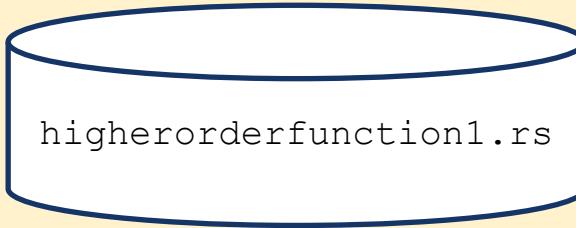
fn main() {
    let text = "Hello, world!".to_string();

    let printer = || text;

    consume_closure(printer);

    // La seguente linea sarebbe errore se decommentata,
    // perché printer non può essere usata più di una volta.
    // consume_closure(printer);

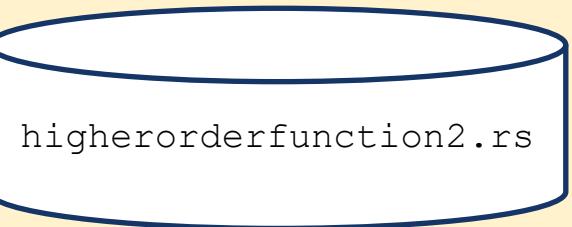
}
```



higherorderfunction1.rs

```
fn call_with_ten<F>(func: F) -> i32
    where
        F: Fn(i32) -> i32,
{
    func(10)
}

fn main() {
    let venti = |x| x * 2;
    let trenta = |x| x * 3;
    println!("10 x 2 = {}", call_with_ten(venti));
    println!("10 x 3 = {}", call_with_ten(trenta));
}
```



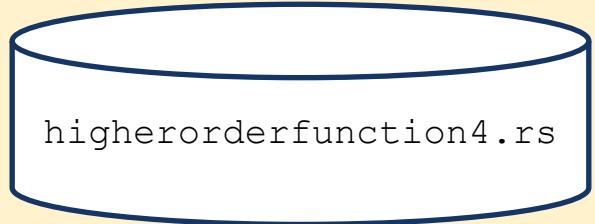
```
fn apply_function<F> (input: i32, function: F) -> i32
    where
        F: Fn(i32) -> i32,
{
    function(input)
}

fn main() {
    let doubled = apply_function(10, |n| n * 2);
    println!("{}", doubled);
}
```

higherorderfunction3.rs

```
fn call_twice<F>(mut closure: F)
    where
        F: FnMut(),
{
    closure();
    closure()
}

fn main() {
    let mut i = 0;
    call_twice(|| i += 1 );
    println!("{}", i);
}
```



higherorderfunction4.rs

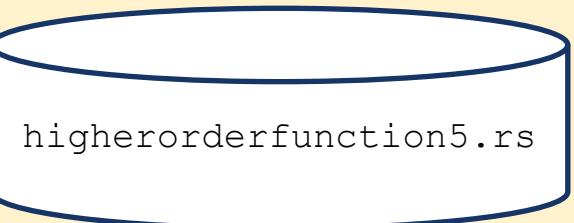
Funzioni di ordine superiore

- 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>**

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
```

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

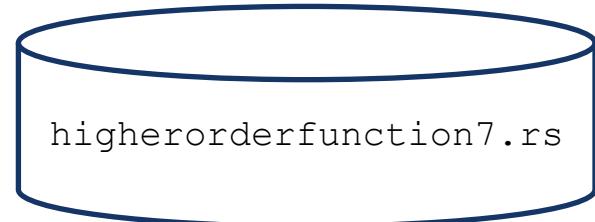
fn main() {
    // Crea una funzione che restituisce il numero 42
    let generate_42 = function_generator(42);
    // Crea una funzione che restituisce la stringa "hello"
    let generate_hello = function_generator("hello".to_string());

    // Esempio con un tipo più complesso come un vettore
    let vec = vec![1, 2, 3];
    let generate_vec = function_generator(vec);

    println!("Numero: {}", generate_42()); // Stampa "Numero: 42"
    println!("Stringa: {}", generate_hello()); // Stampa "Stringa: hello"
    println!("Vettore: {:?}", generate_vec()); // Stampa "Vettore: [1, 2, 3]"
}
```

higherorderfunction6.rs

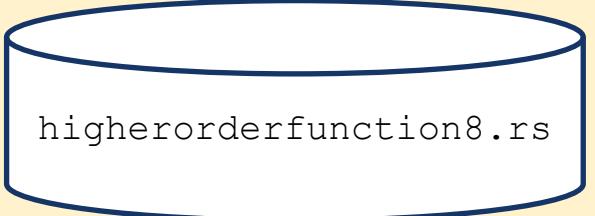
```
fn crea_messaggio(s: String) -> impl FnOnce() -> String {  
    move || s  
}  
  
fn main() {  
    let messaggio = crea_messaggio("Ciao Rust!".to_string());  
    println!("{}", messaggio());  
}
```



higherorderfunction7.rs

```
fn crea_contatore() -> impl FnMut() -> i32 {
    let mut contatore = 0;
    move || {
        contatore += 1;
        contatore
    }
}

fn main() {
    let mut incrementa = crea_contatore();
    println!("Il primo valore è: {}", incrementa());
    println!("Il secondo valore è: {}", incrementa());
}
```



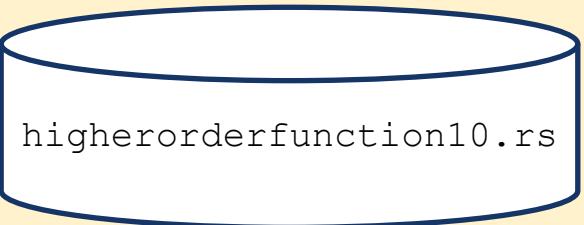
higherorderfunction8.rs

```
fn genera_contatore() -> impl Fn(i32) -> i32 {  
    let contatore = 0;  
    move |incremento: i32| contatore + incremento  
}  
  
fn main() {  
    let conto = genera_contatore();  
    println!("Il risultato è: {}", conto(3));  
    println!("Il risultato è: {}", conto(10));  
}
```



```
fn compose<F, G>(f: F, g: G) -> impl Fn(i32) -> i32
    where
        F: Fn(i32) -> i32,
        G: Fn(i32) -> i32,
{
    move |x| g(f(x))
}

fn main() {
    let add_then_double = compose(|n| n + 1, |n| n * 2);
    println!("{}", add_then_double(5));
    println!("{}", add_then_double(10));
}
```





Per saperne di più

- Functional Programming using Rust
 - <https://medium.com/coderhack-com/functional-programming-using-rust-3776c10cf6>
 - Introduzione alla programmazione funzionale in Rust con esempi
- Functional Programming In Rust: Unlocking Expressive Code
 - <https://marketsplash.com/tutorials/rust/rust-functional-programming/>
 - Rivisitazione degli aspetti funzionali presenti in Rust correddati da semplici esempi
- Easy Rust - Closures
 - https://dghomon.github.io/easy_rust/Chapter_37.html
 - Presentazione del concetto di chiusura in Rust con approccio narrativo
- A guide to closures in Rust
 - <https://hashrust.com/blog/a-guide-to-closures-in-rust/>
 - Trattazione approfondita del concetto di chiusura in Rust con esempi e indicazioni di come il compilatore tratti tale tipo di dato