

Inizieremo a discutere le caratteristiche di Rust relativamente al problema che affronteremo in modo particolare nel dettaglio, cioè il discorso della memory safety in Rust. Rust è un linguaggio di programmazione che è stato sviluppato recentemente e ha tra i suoi promotori la mozilla foundation e vuole affrontare un problema che abbiamo affrontato della prima parte e del corso, cioè il discorso della memory safety e andiamo ricordiamo qual è il problema, quali sono le il perché questo problema è un problema interessante per chi progetta linguaggi di programmazione e anche per chi li usa e anche per la sicurezza, per la programmazione di applicazioni sicure ea vedere poi andrò a vedere come Rust lo affronta

# Memory Safety

A smooth introduction  
to RUST

```

#include <stdio.h>

struct point_t { int x; int y; };

int main() {
    struct point_t p;
    p.x = 5;
    p.y = 65;

    struct point_t p2;

    int* ptr = (int*) &p;
    printf("%d\n", *ptr);          // 5
    printf("%d\n", *(ptr + 1));    // 65
    printf("%c\n", *((char*)(ptr + 1))); // A
    printf("%d\n", *((int*)(&p2 + 1))); // 5

    return 0;
}

```

questo programma C definisce una struct che viene utilizzata per fare la rappresentazione di due punti nel piano cartesiano, quindi l'ascissa e l'ordinata il main, quello che fa crea un puntatore a questa struttura, assegna ai campi, quindi alla ascissa, e all'ordinata il valore 5 e il valore 65 per quanto riguarda l'ordinata, poi crea un altro puntatore sempre alla struttura e fa un po' di operazione su questo puntatore, in modo particolare va a dereferenziare p e va a assegnare questo valore dereferenziato quindi va a rivedere l'oggetto puntato da p e lo assegna a questo nuovo puntatore che abbiamo introdotto a questo punto usa questo puntatore per accedere alla struttura, ad esempio, vede il primo elemento che ha 5 correttamente il secondo elemento usando l'aritmetica dei puntatori diventa 65, poi fa una operazione, cioè acceda all'elemento, fa il casting, la vede come carattere e dovrebbe stampare il carattere a e poi usa il fatto che p2 con la aritmetica dei puntatori gli posso incrementare con un valore, cioè punta all'elemento iniziale del della struttura, punta al primo elemento e viene fuori 5.

C C

Run ►

```
1 #include <stdio.h>
2
3 struct point_t { int x; int y; };
4
5 int main() {
6     struct point_t p;
7     p.x = 5;
8     p.y = 65;
9
10    struct point_t p2;
11
12    int* ptr = (int*) &p;
13    printf("%d\n", *ptr); // 5
14    printf("%d\n", *(ptr + 1)); // 65
15    printf("%c\n", *((char*)(ptr + 1))); // A
16    printf("%d\n", *((int*)(&p2 + 1))); // 5
17
18    return 0;
19 }
```

```
► clang-7 -pthread -lm -o main main.c
► ./main
5
65
A
5
►
```

Le ipotesi sul comportamento del linguaggio di programmazione C ci hanno fatto dedurre una che quello che ci aspettiamo perché? se mi permettete, questa battuta, questo programma è spettacolare e istruttivo. E' spettacolare perché anche terrificante da un altro punto di vista, perché? quello che noi possiamo fare è prendere un puntatore valido dichiarato a un oggetto che è una struttura con un tipo dichiarato correttamente, andiamo a prendere un elemento quindi a entrare dentro questa struttura e facciamo un cast a un tipo differente di una diversa rappresentazione in memoria con diversa dimensione e poi, utilizzando l'aritmetica dei puntatori, possiamo operare sulla struttura, andando a ragionare sul fatto di come all'interno della struttura i campi sono memorizzati e quindi qual è il layout in memoria degli oggetti che abbiamo creato e poi possiamo anche operare sulle posizioni relative da puntatori che sono stati inseriti sulla stack, quindi è importante perché ci permette di far vedere come, utilizzando delle ipotesi sulla rappresentazione interna, e utilizzando il linguaggio a queste ipotesi, riusciamo a fare delle operazioni particolarmente poco eleganti.

**It's worth appreciating how horrendous this program is.**

**We can take a valid pointer to a real value (a point), cast it to a different type of a different size and perform arbitrary arithmetic on the pointer.**

**We simultaneously rely on multiple assumptions about the size of types, the layout of structs, and even the relative layout of two variables on the stack!**

# Memory: ideal properties

- **Memory safety:** memory pointers always point to valid memory (allocated and of the correct type/size).
- **Memory safety is a *correctness* issue:** a memory unsafe program may crash or produce nondeterministic output depending on the bug.
- **Memory containment:** memory does not leak: if a piece of memory is allocated, either it is reachable from the root set of the program, or it will be deallocated eventually.
- **Memory containment is a *performance* issue:** a leaky program may eventually run out of

Quando abbiamo introdotto questo problema ho fatto un po' di esempio e abbiam detto va bene tutto questo è un problema di memory safety, perché abbiamo dei problemi di corruzione della memoria con comportamenti che dipendono da come poi la memoria viene corrotta. Quando abbiamo introdotto il problema lo abbiamo fatto in termini di una nozione di memory safety che diceva quello che li viene scritto, cioè che i puntatori alla memoria devono sempre portare a un oggetto che è stato allocato correttamente in memoria avendo le informazioni corrette col tipo associato dal puntatore e la dimensione corretta. Allora da questo punto di vista la memory safety per come l'abbiamo presentata è un problema di correttezza ovvero è nel modello del linguaggio di programmazione, i puntatori devono sempre avere una coerenza rispetto al tipo e l'uso di un comportamento all'interno di un progetto deve comportare delle regole di coerenza. Se noi rompiamo queste regole di coerenza, insomma, le regole del sistema di tipo allora il sistema che abbiamo implementato può produrre dei comportamenti che sono dipendenti da quello che c'è in una memoria di cui non abbiamo controllo, quindi possono creare dei bug e possono avere dei comportamenti non deterministici. Dal lato della sicurezza e il fatto che l'attaccante usando questi bug può prendere il controllo del programma e operare in modo fraudolento rispetto agli obiettivi del programma che chi aveva progettato aveva in mente. Oltre alla safety c'è anche un problema di memory containment, perché tutte le caratteristiche che abbiamo visto fino a questo momento dalla taint analysis, dall' information flow e dall'history dependent access control che abbiamo esaminato nel dettaglio nelle scorse settimane, quello che vogliono affrontare è che lo stato di esecuzione del programma inclusa la memoria, non è tale da fare del leakage dell'informazione se vi ricordate, questi sono gli aspetti che abbiamo visto nell'ultimo periodo. Tecnicamente in termini di gestione della memoria vuol dire che se io voglio avere una memoria che non fornisce anche dei leak di memory, quindi non ha la possibilità di essere corrotta da questo punto di vista vuol dire che se io vado a vedere astrattamente la memoria ho che la memoria dinamica è raggiungibile da uno degli elementi che sta nel root set, quindi dall' entità che sono attive in qualunque momento in esecuzione il programma o a un certo punto sarà deallocata e restituito alla memoria. Quindi non c'è del leakage della memoria, dovuto a un problema di gestione, notate che il memory containment non è tanto un problema di correttezza ma un problema di performance. Perché se io ho un leak di memoria quindi ho della memoria che non è utilizzata, se il programma continua a scrivere memoria, non avrà più spazio per fare richiesta alla memoria quindi a un certo punto ci sarà un problema, si bloccherà per run out.

è molto meglio che il sistema di implementazione, la macchina virtuale di realizzazione del linguaggio che si occupano di gestire la memoria allora il linguaggio di programmazione che hanno il garbage collector allora a questo punto è esattamente il run time che invoca il garbage collector e a questo punto si preoccupa di restituire la memoria non utilizzata allo heap e succede che dal punto di vista del linguaggio di programmazione la memoria è un astrazione di un supporto di memorizzazione perché chi programma ha il linguaggio e la macchina virtuale del linguaggio che deve affrontare i problemi di gestione della memoria è un dettaglio di implementazione. Vuol dire che la memory safety con garbage collector è garantito esattamente da questa astrazione. Il memory containment dice non c'ho del leakage e quindi il sistema di gestione della memoria mi permette di restituire, identifica quindi tutti quelle componenti di memoria sono diventati garbage, quindi non più raggiungibili non più utilizzabili perché associati a strutture attive e le restituisce alla memoria per poter essere utilizzato allora il memory containment è garantito dal garbage collector che ha ricordo della memoria raggiungibile.

**In garbage-collected languages memory safety is guaranteed for all data allocated on the heap within the language runtime, assuming a correct implementation of the garbage collector.** These languages abstract away memory as an implementation detail.

**Memory containment is guaranteed for tracing garbage collectors (like Mark&Sweep), but not necessarily for reference counting garbage collectors in the presence of cycles.**

A (not null)  
reference

Null References: the billion dollar mistake

<https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/>

Il garbage collector è esattamente il meccanismo che ci occorre nella struttura dei linguaggi di programmazione, quindi nelle macchine virtuali dei linguaggi di programmazione per avere un' astrazione della memoria e allo stesso tempo avere dei sistemi che sono in grado di avere un overhead accettabile a tempo di esecuzione e garantire la memory safety e la memory containment. Ovviamente ci sono delle situazioni dove la scelta di garbage collector hanno dei vincoli sull'implementazione e coloro che devono implementare delle applicazioni particolari hanno delle obiezioni all'uso del garbage collector.

# Garbage Collection

- GC makes programmers significantly more productive and less bug-prone while imposing an acceptable runtime overhead.
- Garbage collector is not a panacea for memory management.
  - Programmers for embedded devices prefer to carefully craft programs for minimal memory footprint when their device has only 64KB of RAM.
  - Garbage collectors involve some kind of unpredictable stop-the-world overhead, which can be killer for apps that have functionality that needs to be always running.
  - Careful control over memory layouts can provide avoid cache thrashing by improving spatial locality. Low-level memory control can often reduce accidental/unnecessary memory allocations.

Ad esempio se voi dovete programmare un controllore per un qualche dispositivo embedded e avete 64k di ram beh a questo punto quello che volete volete gestire la memoria da programma perché volete ottimizzare, avete poca memoria e avete un problema di performance elevato e quindi dovete ottimizzare a mano la gestione della memoria per gli obiettivi che avete. cioè non non vi interessa avere questa astrazione similmente per tutte quelle applicazioni che hanno dei tempi di risposta brevi e devono essere molto stringenti anche un overhead minimo dovuto alla chiamata del garbage collector fa la differenza quindi vogliono avere delle soluzioni ad hoc. Non solo perchè questo fatto di avere l'astrazione della memoria a livello del linguaggio di programmazione non piace, ad esempio a coloro che fanno high performance computing perché ad esempio, vogliono utilizzare i dati che sono in cache e la località e la spazialità la vicinanza di dati nella rappresentazione ovviamente in memoria, per poter avere un controllo più fine sulla performance di esecuzione e quindi anche loro fanno delle scelte differenti vuol dire che questo discorso nel garbage collector a tutti quegli aspetti positivi che vi dicevo, ma ne ha anche di critici.

il progetto di Rust voleva utilizzare a basso livello la memoria, cioè avere un controllo di basso livello della memoria per affrontare esattamente alcune di quelle proprietà che dicevamo prima relativamente all' efficienza, però lo voleva utilizzare senza avere il garbage collector, quindi proprio voleva non avere il garbage collector ma allo stesso tempo voleva avere una proprietà di correttezza, quindi, di memory safety e lo garantiva, lo garantiva con un meccanismo di tipo elegante, anche in presenza di costrutti per la concorrenza allora andiamo a fare un po di storia breve, il progetto è iniziato nel 2006, nell 2010 ha avuto un finanziamento dalla mozilla foundation e ora c'è un mucchio di discussione da parte dei professional che lavorano nell information technology, sull'uso e sulle caratteristiche positive di Rust. In alcuni ad esempio negli stati uniti, in alcune università Rust è diventato il primo linguaggio di programmazione.

Started in 2006 by Graydon Hoare

Sponsored by Mozilla in 2010

Now: most loved programming language in Stack Overflow

Key properties: Type safety despite use of concurrency and manual memory management

La caratteristica tipica è il discorso della ownership e lifetime che sono esattamente le caratteristiche che permettono di comprendere e di gestire manualmente la memoria poi, dal punto di vista del modello a oggetti, usa i traits come meccanismo per gestire gli oggetti. Per chi conosce java sono delle specie di interfacce. Quando io introduco una variabile per default è immutable, vi ricordate le strutture dati immutable hanno uno dei vantaggi rispetto alla struttura mutable perché non introducono i data race, quindi il fatto di avere strutture immutabili è una caratteristica peculiare del linguaggio, e anche questo viene derivato moltissimo dalla programmazione funzionale. Ha strutture di tipo oltre a vari tipi elementari, ha le tuple, gli oggetti e altre strutture di tipo, ma soprattutto ha il pattern matching alla ml, vuol dire che ho dei meccanismi che mi permettono di andare all'interno della struttura seguendo dei pattern di regolarità anche questo è derivato da i linguaggi funzionali come anche il meccanismo di type inference. Chi programma può annotare il programma con le dichiarazioni di tipo, può anche non annotarlo ed è il sistema che mi inferisce il tipo. Ha il polimorfismo con i generici, quindi quello che si chiama il polimorfismo parametrico da sottotipo e poi ha il meccanismo chiave della programmazione funzionale, ovvero le funzioni sono entità di ordine superiore, quindi possono essere utilizzati come valori e stanno dappertutto. E' un linguaggio sostanzialmente multi paradigma con moltissime delle caratteristiche della programmazione funzionale che molti di voi hanno visto per la prima volta all'interno di questo corso, con ocaml.

---

## Lifetimes and Ownership: Key feature for ensuring safety

---

### Traits as core of object(-like) system

---

### Variable default is immutability

---

### Data types and pattern matching

---

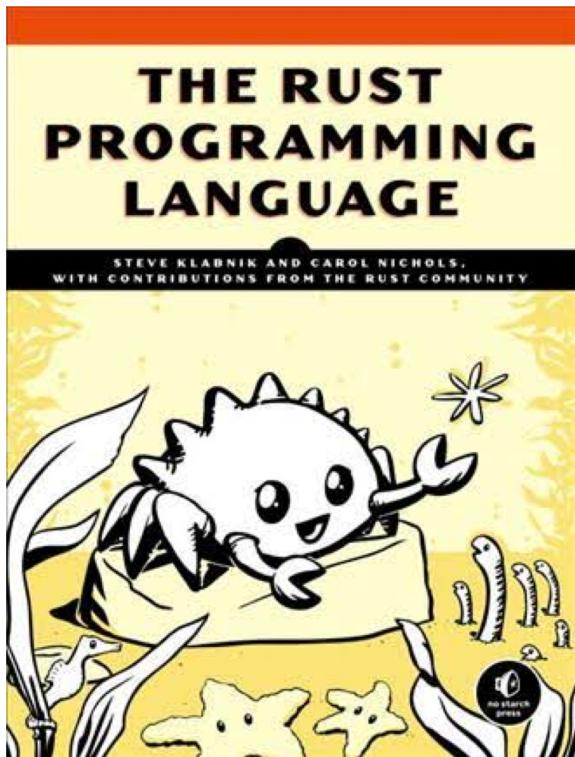
### Type inference

---

### Generics (parametric polymorphism)

---

### First-class functions



- Rust book free online
  - <https://doc.rust-lang.org/book/>
  - **We will follow it in these lectures**
- More references via Rust site
  - <https://www.rust-lang.org/en-US/documentation.html>
- Rust Playground (REPL)
  - <https://play.rust-lang.org/>

## FUNCTIONS

```
// comment
fn main() {
    println!("Hello, world!");
}
```

Hello, world!

Dichiaro una funzione che non ha parametri, restituisce void come risultato e stampa. Si capisce subito che abbiamo un meccanismo di bootstrapping, quindi abbiamo tra le varie funzioni che compongono il programma ne abbiamo una particolare che si chiama main e questa è quella che attiva il programma. Le altre funzioni sono visibili a partire da main.

Questa è una funzione ricorsiva che prende e qui stiamo annotando con le dichiarazioni dei tipi o poi andremo più avanti a vedere quali sono i tipi stiamo annotando il parametro n che è un intero a 32 bit, restituisce come risultato un intera a 32 bit quindi da un'idea della rappresentazione due alla 32-1.

```
fn fact(n:i32) -> i32
{
    if n == 0 { 1 }
    else {
        let x = fact(n-1);
        n * x
    }
}

fn main() {
    let res = fact(6);
    println!("fact(6) = {}", res);
}

fact(6) = 720
```

# OCAML VS RUST

```
# let f (m : int) (n : int) : int = fn f(m: i32, n: i32) -> i32 {
  let mm = 2 * m in
    mm + n
}
val f : int -> int -> int = <fun>
fn main() {
# let main () =
  Printf.printf "%d" (f 3 5) ;
  val main : unit -> unit = <fun>
  main () ;
  println!("{}", f(3, 5));
}
# main ();
11- : unit = ()
```

RUST is expression-oriented.

The semicolon in **let x = y; z** has basically the role of the **in** keyword in Ocaml:

**it defines an expression, not a statement as in C**

In Rust, vedete che non abbiamo il currying, è il meccanismo di prendere i parametri uno alla volta. per questo prima dicevo f è una funzione in ocamel che prende un intero e restituisce come risultato una funzione da interi a interi quindi il meccanismo del currying che ci permette di avere funzione con più parametri in Rust non l'abbiamo. In modo particolare abbiamo la possibilità di dichiarare variabili locali. Non c'è l'in che ci dice io ho un let dichiaro un'espressione che poi userò nel corpo del let, bene il punto e virgola che abbiamo in in Rust subito dopo la dichiarazione di un let di una variabile è sostanzialmente equivalente all' in di Ocaml. Da questa prima analisi abbiamo che di fatto abbiamo un meccanismo che è orientata alle espressioni, quindi allo stile della programmazione funzionale e al restituire dei valori come la programmazione funzionale in termini di funzione opportunamente tipata.

Il fatto che le espressioni non sono comandi lo si vede bene da da questo esempio vedete, questo è un ifthenelse innestato uno dentro l'altro.

## IF expressions (not statements)

```
fn main() {  
    let n = 5;  
    if n < 0 {  
        print!("{} is negative", n);  
    } else if n > 0 {  
        print!("{} is positive", n);  
    } else {  
        print!("{} is zero", n);  
    }  
}
```

5 is positive

A questo punto, andiamo a vedere meglio il discorso del let allora quando io dichiaro una variabile con un let ho un blocco dichiaro la portata della dichiarazione nell'espressione che corrisponde al corpo del let. In questo caso vedete ad esempio sto dichiarando una variabile di tipo intero, la inizializzo a zero e poi voglio incrementare a di 1. Quello che abbiamo a sinistra da un errore di compilazione perché da un errore di compilazione, perché le variabili per default sono immutable, quindi se sono immutable non possono essere modificate, non possono essere messe all'interno di assegnamenti per cui a questo punto il sistema si irrita pesantemente e non compila. Se noi vogliamo avere invece una variabile modificabile, cioè una variabile che a cui io posso poi operare tramite assegnamenti devo dichiararlo espressamente con il marcatore mut. Quello sulla destra abbiamo lo stesso programma da un punto di vista di forma, ma dove questo programma compila perché stiamo dichiarando una variabile locale mutabile a di tipo intero inizializzato a 0 e nel corpo del let stiamo incrementando di uno il valore di a quindi da questo punto di vista il compilatore di Rust non ha nulla da eccepire proprio per il fatto che è mutabile.

## LE I

- By default, Rust variables are immutable
  - Usage checked by the compiler
- **mut** is used to declare a resource as mutable.

```
fn main() {  
    let a: i32 = 0;  
    a = a + 1;  
    println!("{}" , a);  
}
```

```
fn main() {  
    let mut a: i32 = 0;  
    a = a + 1;  
    println!("{}" , a);  
}
```

Compile error

## LET BY EXAMPLES

```
{  
  let x = 37;  
  let y = x + 5;  
  y  
}//42
```

```
{  
  let x = 37;  
  x = x + 5;//err  
  x  
}
```

```
{ //err:  
  let x:u32 = -1;  
  let y = x + 5;  
  y  
}
```

```
{  
  let x = 37;  
  let x = x + 5;  
  x  
}//42
```

```
{  
  let mut x = 37;  
  x = x + 5;  
  x  
}//42
```

```
{  
  let x:i16 = -1;  
  let y:i16 =  
x+5;  
  y  
}//4
```

Redefining a variable *shadows* it (like OCaml)

Assigning to a variable only allowed if **mut**

Type annotations must be consistent (may override defaults)

## QUIZ #1: What does this evaluate to?

```
{ let x = 6;
  let y = "hi";
  if x == 5 { y } else { 5 };
  7
}
```

- A. 6
- B. 7
- C. 5
- D. Error

Non in termini ovviamente di Rust, ma dell'esperienza del linguaggio di programmazione. Da errore perché l'if non è tipato bene non sappiamo quale sia il tipo controllandolo staticamente e si è il problema il problema dei tipi, ma il problema è dovuto al fatto che tipicamente i due campi, il ramo then e il ramo else hanno tipi incompatibili perché il ramo then è una stringa il ramo else è un intero vi ricordate che la filosofia del ifthenelse visto come espressione è di avere il tipo del ramo then e else uguali, affinche il tipo sia compatibile, l'abbiamo visto in ocaml, abbiamo visto anche l'altro giorno quando abbiamo affrontato il sistema di tipi effetto per l' history dependence access controll che la filosofia quando uso nel linguaggio di programmazione l'ifthenelse come espressione il tipo del ramo else del ramo the devono essere identici

## QUIZ #1: What does this evaluate to?

```
{ let x = 6;
  let y = "hi";
  if x == 5 { y } else { 5 };
  7
}
```

- A. 6
- B. 7
- C. 5
- D. Error – if and else have incompatible types

## Mutation and Loop

```
fn fact(n: u32) -> u32 {  
    let mut x = n;  
    let mut a = 1;  
    loop {  
        if x <= 1 { break; }  
        a = a * x;  
        x = x - 1;  
    }  
    a  
}
```

infinite loop  
(break out)

# Looping constructs

- **while ...., for ..., loop ...**
- These looping constructs are *expressions*
  - They return the final computed value
  - unit, if none

`break` may take an expression argument, which is the final result of the loop

```
let mut x = 5;
let y = loop {
    x += x - 3;
    println!("{}" , x); // 7 11 19 35
    x % 5 == 0 { break x; }
};
print!("{}" , y); //35
```

La cosa importante è che tutti i costrutti di loop sono delle espressioni, vuol dire che restituiscono valori e quindi restituiscono il valore calcolato in fondo, in modo particolare ad esempio vedete nell'esempio precedente usavamo il valore in fondo dell'accumulatore per restituire il valore però ad esempio potremmo scrivere che stiamo dichiarando non variabile mutable stiamo dicendo che y è uguale al risultato dell'espressione che ha un loop al suo interno. Il loop al suo interno incrementa x col valore precedente decrementato di 3, poi stampa il valore di x, se x modulo 5 è uguale a 0, chiama un break e li abbiamo un'espressione e il valore di quell'espressione viene restituito quando viene eseguito il break e quindi è il risultato finale del loop e questo è il motivo per cui può essere messo in un qualcosa che è un'espressione che assegna un valore.

## Quiz #2

```
fn main() {  
    let mut x = 0;  
    for i in 1..6 {  
        let x = x+i;  
        println!("{}", x);}  
    x = x+ 100;  
    println!("{}", x)  
}
```

**What this evaluates to?**

The screenshot shows a Rust playground interface. At the top, there are buttons for 'RUN' (highlighted in red), 'DEBUG', 'STABLE', and three dots. Below the editor, the command line output is displayed:

```
Compiling playground v0.0.1 (/playground)
Finished dev [unoptimized + debuginfo] target(s) in 4.71s
Running `target/debug/playground`
```

The code in the editor is:

```
fn main() {
    let mut x = 0;
    for i in 1..6 {
        let x = x+i;
        println!("{}", x);
        x = x+100;
        println!("{}", x)
    }
}
```

The output window shows the following results:

```
1
2
3
4
5
100
```

viene stampato 100 perché c'è dentro il for un let, quindi viene dichiarata una nuova variabile che quindi non va ad aggiornare la variabile quella fuori dal for, quindi quando arriviamo là abbiamo  $0+100$ . Abbiamo un meccanismo di shadow, quindi abbiamo lo stesso nome ma in realtà sono due istanze diverse della stessa variabile quando usciamo dal blocco del for perdiamo il riferimento alla variabile interna e a quel punto l'unico riferimento che abbiamo è ancora di quello esterno che inizialmente è uguale a zero che era mutable e infatti questo è il motivo per cui si stampa cento

# Scalar type

- Integers
    - `i8, i16, i32, i64, isize`
    - `u8, u16, u32, u64, usize`
  - Characters (unicode)
    - `char`
  - Booleans
    - `bool = { true, false }`
  - Floating point numbers
    - `f32, f64`
  - Note: arithmetic operators (+, -, etc.) overloaded
- 
- Machine word size
- Defaults (from inference)

# tuples

## Tuples

- n-tuple **type** (*t<sub>1</sub>*, ..., *t<sub>n</sub>*)
  - **unit** () is just the 0-tuple
- n-tuple **expression** (*e<sub>1</sub>*, ..., *e<sub>n</sub>*)
- Accessed by pattern matching or like a record field

```
fn main() {  
    let tup = (500, 6.4, 1);  
  
    let (x, y, z) = tup;  
  
    println!("The value of y is: {}", y);  
}
```

```
fn main() {
    let x: (i32, f64, u8) = (500, 6.4, 1);

    let five_hundred = x.0;

    let six_point_four = x.1;

    let one = x.2;

    println!("The value of five_hundred is: {}", five_hundred);
    println!("The value of six_point_four is: {}", six_point_four);
    println!("The value of one: {}", one);
}
```

RUN ►

...

DEBUG ▾

STABLE ▾

...

```
4  let five_hundred = x.0;
5
6  let six_point_four = x.1;
7
8  let one = x.2;
9
10 println!("The value of five_hundred is: {}", five_hundred);
11
12 println!("The value of six_point_four is: {}", six_point_four);
13
14 println!("The value of one: {}", one);
15
16 }
17
```

:::

Execution

Standard Error

```
Compiling playground v0.0.1 (/playground)
Finished dev [unoptimized + debuginfo] target(s) in 1.57s
Running `target/debug/playground`
```

Standard Output

```
The value of five_hundred is: 500
The value of six_point_four is: 6.4
The value of one: 1
```

A differenza del C il tipo di un array non è il puntatore alla prima posizione dell'array, ma tiene conto anche della dimensione dell'array e quindi da un errore a tempo di esecuzione proprio perché va a controllare la dimensione dell'array stesso.

## Arrays

### Standard operations

- [Creating](#) an array (can be mutable or not)
  - But must be of fixed length
- [Indexing](#) an array
- [Assigning](#) at an array index

```
let nums = [1,2,3];
let strs = ["Monday", "Tuesday", "Wednesday"];
let x = nums[0]; // 1
let s = strs[1]; // "Tuesday"
let mut xs = [1,2,3];
xs[0] = 1; // OK, since xs mutable
let i = 4;
let y = nums[i]; //fails (panics) at run-time
```

- Rust provides a way to **iterate over a collection**
  - Including arrays

```
let a = [10, 20, 30, 40, 50];
for element in a.iter() {
    println!("the value is: {}", element);
}
```

- `a.iter()` produces an **iterator**, like a Java iterator

# RUST VS OCAML

AGAIN



Rust also has strings, algebraic  
data types, recursive types, and  
polymorphism just like OCaml.

```
// Polymorphic record for points. Equivalent to type 'a point = {x : 'a; y : 'a}.
```

```
struct Point<T> {
    x: T, y: T
}
```

il fatto che `ToString` è un trait vuol dire che deve essere implementata e semplicemente vedete la cosa che fa è trasformare in stringa le informazioni associate una cosa che abbiamo in tutti i linguaggi oggetti

```
// Polymorphic function. T: ToString means "all types T where T has a to_string
// method." ToString is a trait, which we will discuss in depth next class.
```

```
fn point_print<T: ToString>(p: Point<T>) {
    println!("({}, {})", p.x.to_string(), p.y.to_string());
}
```

```
// Polymorphic recursive sum type for lists. Note that we have to wrap the recursive reference in a Box<...>.  
// This ensures the type has a known size in memory.
```

```
enum List<T> {  
    Nil,  
    Cons(T, Box<List<T>>)  
}  
  
fn main() {  
    let p1: Point<i32> = Point { x: 1, y: 2 };  
    let p2: Point<f32> = Point { x: 3.1, y: 4.2 };  
    point_print(p1); // (1, 2)  
    point_print(p2); // (3.1, 4.2)  
    // Rust has type inference, so it can infer the type of `l` as List<i32>.  
    let l = List::Cons(5, Box::new(List::Nil));  
    // Rust has match statements just like OCaml.  
    let n = match l {  
        List::Cons(n, _) => n,  
        List::Nil => -1  
    };  
    println!("{}", n); // 5  
}
```

Il tipo lista è un tipo ricorsivo è generico rispetto al tipo degli elementi ha un valore che è la lista vuota, ha un costruttore cons che prendo un elemento di tipo p e poi prende un qualcosa che si chiama box box che prende sostanzialmente la definizione ricorsiva del tipo quindi il box è utilizzato per wrappare all'interno della rappresentazione della memoria che cosa la struttura ricorsiva quindi per avere la consistenza di rappresentazione del tipo. Questa è quindi la differenza con ocaml è una differenza che proprio ha a che vedere sulla gestione della memoria

RUN ▶

...

DEBUG ▾

STABLE ▾

...

```
21 let p1: Point<i32> = Point { x: 1, y: 2 };
22 let p2: Point<f32> = Point { x: 3.1, y: 4.2 };
23 point_print(p1); // (1, 2)
24 point_print(p2); // (3.1, 4.2)
25
26 // Rust has type inference, so it can infer the type of `l` as List<i32>.
27 let l = List::Cons(5, Box::new(List::Nil));
28 // Rust has match statements just like OCaml.
29 let n = match l {
30     List::Cons(n, _) => n,
31     List::Nil => -1
32 };
33 println!("{}", n); // 5
34 }
```

⋮

Execution

Standard Error

Standard Output

```
Compiling playground v0.0.1 (/playground)
Finished dev [unoptimized + debuginfo] target(s) in 2.77s
Running `target/debug/playground`
```

```
(1, 2)
(3.1, 4.2)
5
```

Adesso ci cominciamo ad avvicinare alle caratteristiche relativamente alla safety dell'uso della memoria. Sullo stack ci stanno sostanzialmente tutti quei dati che hanno una dimensione e un tempo di vita che sono conosciute a tempo di compilazione quindi ci stanno ad esempio i parametri locali delle funzioni, perché ci sono le dichiarazioni locali, dalla dichiarazione del tipo riesce a inferire la rappresentazione in memoria del tipo di dato, qual è la portata? il tempo di vita di quella dichiarazione è il tempo di vita della funzione, quando la funzione viene immessa sulla stack, viene eseguita la particolare attivazione quando la funzione restituisce il controllo al chiamante viene deallocated, quindi si fa un'operazione di push, poi si fa un'operazione di pop del record di attivazione dallo stack e a questo punto non sono più attive.

# Data allocation in memory: Stack vs Heap

## The stack

- constant-time, automatic (de)allocation
- Data size and lifetime must be known at compile-time
- Function parameters and locals of known (constant) size

## The heap

- Dynamically sized data, with non-fixed lifetime
- Slightly slower to access than stack;
- via a pointer

Vuol dire che sia la allocazione che la deallocazione hanno un tempo costante, corrispondono al tempo che occorre al prologo per fare l'operazione di push del nuovo record di attivazione sullo stack e all' epilogo che sono le parti dove viene compilata una funzione, ci metto un prologo e un epilogo che si preoccupa esattamente di fare la deallocazione è un tempo costante che dipende da sostanzialmente da quali sono le strutture presenti nel run time. Nello heap ci stanno delle strutture dati dinamiche che non è prevedibile a tempo di compilazione sapere la dimensione, ad esempio una lista boh dipende da cosa succede durante l'esecuzione di questa lista e che hanno un tempo di vita non fissabile che dipende esattamente dal flusso di esecuzione, non dipende dalla struttura del flusso, tipicamente si accede tramite un puntatore che è leggermente meno efficiente di accedere a un elemento sullo stack perché uno utilizza il solito meccanismo indirizzo di base più offset, quindi uno dall' offset e dall' indirizzo di base è molto efficiente, però anche i puntatori nelle macchine moderne non è che costano così tanto l'accesso.

# RUST: safe memory management

- Rust heap memory management: **without GC**
  - Type checking ensures **no dangling pointers**
  - unsafe idioms are disallowed
- Key features: **ownership** and **lifetimes**
  - Data has a single owner.
  - Immutable aliases OK, but mutation only via owner or single mutable reference
  - How long data is alive is determined by a lifetime

# Ruke of ownership

1. Each value in Rust has a variable that's its **owner**
2. There can only be **one owner at a time**
3. When the **owner goes out of scope** the value will be **dropped (freed)**

Rust fa esattamente queste operazioni per la gestione della memoria usa e le informazioni di tipo e usa le informazioni tipo non solo per associare la struttura di memoria da associare all' entità, ma usa le informazioni di tipo per gestire manualmente lo heap senza fare ricorso al garbage collector. Sostanzialmente usa queste informazione di ownership Le regole di ownership ogni valore in Rust ha una variabile che è il suo proprietario, quindi vuol dire quando io dichiaro una variabile e un valore associato a quella variabile è il proprietario e ce'è un solo proprietario alla volta. Le regole di Rust dicono che è quando io definisco il proprietario, ogni valore in memoria ha un unico proprietario ad ogni istante dell'esecuzione. Quando il proprietario esce dallo scope, quindi dalle regole sue di visibilità, il valore di cui lui era proprietario viene restituito alla memoria, quindi ha questa nazione forte di un ownership. Essendo soggetta alle regole di scopo che sono presenti come negli altri linguaggi di programmazione, quando una variabile esce dallo scopo automaticamente viene restituito alla memoria. Il fatto di avere un unico proprietario è la caratteristica che permette la gestione della memoria senza il garbage collector.

# Scope rules ... as usual

```
fn main() {  
    { // s is not valid here, it's not yet declared  
let s = "hello";  
    // s is valid from this point forward // do stuff with s  
    }  
    // this scope is now over, and s is no longer valid  
}
```

**When s comes *into scope*, it is valid.**

**It remains valid until it goes *out of scope*.**

Ie regole di scope sono quelle usuali, cioè andiamo a vedere questo esempio, vedete dichiaro una funzione del main fintanto che non faccio una dichiarazione della s non è entrata nello scopo non è valida appena s è entrata nello scope, diventa valida a questo punto è valida fintanto che vale lo scope di quel let, quando usciamo dallo scope di quel let a questo punto non è più valida e quindi può essere restituita alla lista libera il valore associato, ma questo vale per tutti i linguaggi di programmazione.

# String: dynamically sized mutable data

```
{  
    let mut s = String::from("hello"); // namespace  
    // like Java package)  
    s.push_str(", world!"); // appends to s  
    println!("{}", s); // prints hello, world!  
} // s's data is freed by calling s.drop()
```

- The contents of string **s** is allocated on the heap
  - Pointer to data is internal to the **String** object rep
  - When appending to **s**, the old data is freed and new data is allocated
- **s** is the *owner* of this data
  - When **s** goes out of scope, its **drop** method is called, which frees the data

Abbiamo una struttura che è una stringa che è di dimensione dinamica mutable allora il fatto di mettere i due punti due punti e subito dopo la stringa dice che noi sono le operazioni del name space che sono esattamente il modo che ho in Rust di importare i package come in Java scriverei import e stiamo dicendo che stiamo importando dal package delle stringhe il letterale hello e lo associamo a s che è mutable possiamo anche ad esempio fare la concatenazione, quindi possiamo usare push string dal package delle stringhe per concatenare a s la parola world per fare la stampa di hello world. Siamo all'interno di un let, quindi vuol dire che questa è una struttura dati dinamica, l'abbiamo allocata sullo heap a questo punto usciamo da questo scope, vuol dire che a questo punto, alla fine di questo blocco s non è più visibile perché è uscita dallo scope allora a questo punto viene liberata viene restituita la lista libera non perché noi come programmatore qui scriviamo free(s) ma perché il sistema automaticamente invoca delle primitive, che sono primitive di sistema associate all' implementazione delle strutture dati immutable e di dimensione variabile, cioè quelle che vanno sullo heap in modo particolare la primitiva drop che ha esattamente il significato della free, cioè viene restituita la memoria occupata dalla lista libera. L'idea è che in questo modo si associa al dato stringa un unico proprietario che è s e quando si esce dallo scope quindi s non è più visibile, si restituisce il valore

allora andiamo a vedere com'è fatta la gestione della memoria in Rust benissimo succede in tutti i linguaggi di programmazione, ho bisogno di memoria, chiedo memoria a questo punto la gestione della richiesta è fatta con una new o con un qualcosa simile nel new che è presente in altri linguaggi di programmazione, non cambia assolutamente niente il punto è, come cambia la gestione della restituzione della memoria dinamica allora questo è fatto in modo diverso, cioè nei linguaggi con garbage collector il sistema è un' astrazione la memoria, il programmatore non deve scrivere niente, è il sistema che si preoccupa di attivarsi, di fare la traccia della mappatura dello heap e a quel punto, restituire il garbage. Nei linguaggi tipo il C, uno ha la free, ma questo dà problemi di dangle reference, di liberazione del garbage, non identificazione del garbage e di problemi di puntatori non completamente usati correttamente.

# Memory usage

**The memory must be requested from the memory allocator at runtime (standard in PL!!!)**

**We need a way of returning this memory to the allocator when we're done with our dynamic data.**

Returning memory is done differently in programming languages.

**In languages with a *garbage collector (GC)*, the GC keeps track and cleans up memory that isn't being used anymore, and the programmer don't need to think about it.**

**Without a GC, it's responsibility of the programmer to identify when memory is no longer being used and call code to explicitly return it, just as we did to request it.**

**Doing this correctly has historically been a difficult programming problem.**

If we forget, we'll waste memory (**garbage**). If we do it too early, we'll have an invalid variable (**dangling references**).

Issue: **We need to pair exactly one allocate with exactly one free.**

# Memory usage: drop

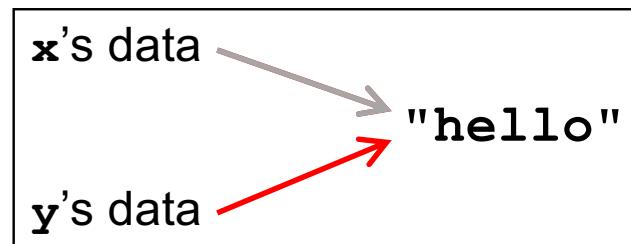
- When a variable goes out of scope, Rust calls a special run-time function.
- This function is called [drop](#),
- Rust calls drop automatically at the closing of blocks.

L'idea è di associare un unico ownership e usare il drop all'atto di quando si esce dallo scope, all'uscita del blocco, quindi, si chiama il drop che restituisce memoria alla lista libera.

- Heap allocated data is copied by reference

```
let x = String::from("hello");
let y = x; //x moved to y
```

- Both **x** and **y** point to the same underlying data



*Avoids double  
free()!*

- A move leaves only one owner: **y**

```
println!("{} , world!", y); //ok
println!("{} , world!", x); //typechecking fails
```

Allora adesso andiamo a vedere però cosa succede con l' aliasing, perché il problema del garbage collector è che se ho un unico cammino di accesso e come programmatore mi assicuro che ha un unico cammino di accesso a quella struttura dati io faccio la free e faccio la free per bene il problema è cosa succede quando io genero aliasing? Andiamo a vedere che cosa succede nei meccanismi di creazione dell' aliasing abbiamo due modi di crearli, allora il primo si chiama by reference o per persone che vengono da una conoscenza di java è quella che si chiama la shallow copy. Supponiamo di avere creato, la stringa hello dal namespace delle stringhe, poi supponiamo di dichiarare un'altra stringa y e associare il valore di x a y, allora a questo punto, quello che dice Rust è che il valore associato a x è mosso, si muove ed è dato a y. Vuol dire che il proprietario che prima era x è diventato y allora se andiamo a vederlo dal punto di vista dei cammini di accesso, il cammino acceso sono esattamente quelli che vediamo ora, cioè vediamo x che ha un cammino di accesso in grigio a hello e poi y che ha un cammino di accesso in rosso a hello. E' in rosso perché a questo punto il proprietario è y, cioè il fatto di aver fatto questa shallow copy quindi ho copiato i puntatori, però facendo la copia del puntatore ho anche spostato il proprietario, quindi il proprietario non è più x ma il proprietario è diventato y allora a questo punto quando io vado a accedere con la y il compilatore mi dice: perfetto tu lo puoi fare, ma se cerco di accedere con la x a quello che prima era puntato dalla x e che ora è sempre puntato dall' x ma x non è più la proprietaria del dato, il compilatore fallisce cioè il sistema dei tipi me lo fa controllare a tempo di compilazione, non a tempo di esecuzione

The screenshot shows a development environment with a code editor and a terminal window.

**Code Editor:**

```
1 fn main() {  
2     let s1 = String::from("hello");  
3     let s2 = s1;  
4     println!("{} world!", s1);  
5     println!("{} world!", s2);  
6 }
```

**Terminal (Execution Output):**

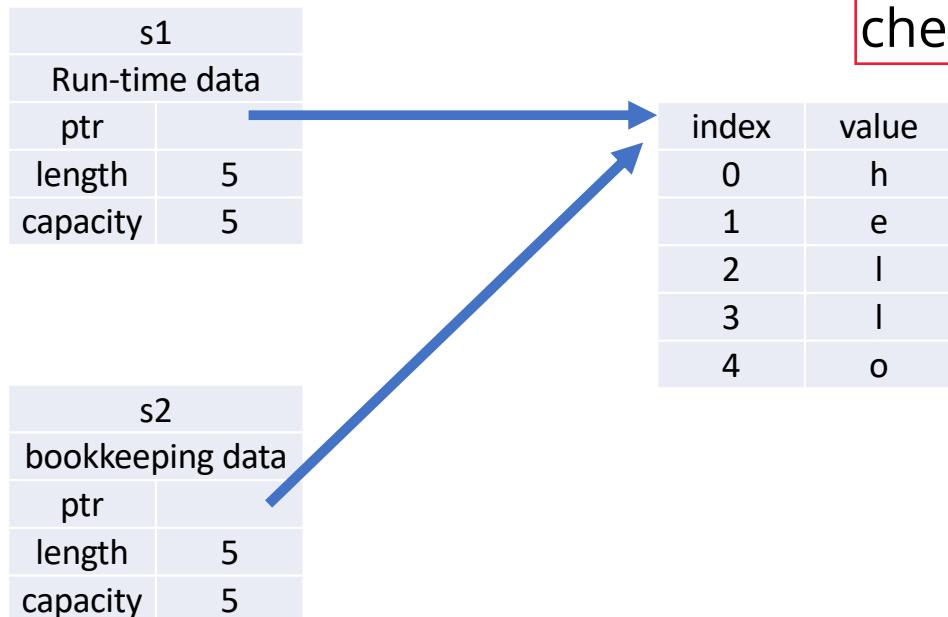
```
Execution  
1 let s1 = String::from("hello");  
2     -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait  
3 let s2 = s1;  
4     -- value moved here  
5     println!("{} world!", s1);  
6         ^^ value borrowed here after move  
  
error: aborting due to previous error  
  
For more information about this error, try `rustc --explain E0382`.  
error: could not compile `playground`  
  
To learn more run the command again with --verbose
```

# Run time representation

| s1       |   | Run-time data |       |
|----------|---|---------------|-------|
| ptr      |   | index         | value |
| length   | 5 | 0             | h     |
| capacity | 5 | 1             | e     |
|          |   | 2             | l     |
|          |   | 3             | l     |
|          |   | 4             | o     |

Ho un valore sullo stack un po' più grasso rispetto da quello che uno si aspetta e poi sull'heap vado a mettere l'indice e la posizione del letterale, unicode che corrisponde ad h e l l o, questa è la rappresentazione di s1

# Run time representation



quando io vado a fare l'operazione di shallow copy ho s2 ed s1 che puntano alla stessa struttura sull'heap. Vuol dire che i valori sullo stack sono copiati cioè i valori associati alla rappresentazione del puntatore sono copiati e stanno sulla stack, ma i valori che invece erano sullo heap non sono copiati, sono condivisi solo che è cambiato l'owner.

the pointer, the length, and the capacity **are copied**  
the data on the heap that the pointer refers to **are not copied**

RUN ►

DEBUG ▾

STABLE ▾

```
1 fn main() {
2     let s1 = String::from("hello");
3     let s2 = s1;
4
5     println!("{} , world!", s1);
6 }
```

⋮⋮⋮

### Execution

```
error[E0382]: borrow of moved value: `s1`
--> src/main.rs:5:28
|
2 |     let s1 = String::from("hello");
|         -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait
3 |     let s2 = s1;
|             -- value moved here
4 |
5 |     println!("{} , world!", s1);
|             ^`value borrowed here after move`
```

error: aborting due to previous error; 1 warning emitted

# Remark

- **shallow copy** is usually used to explain the the concept of copying the pointer, length, and capacity without copying the data.
- Rust also invalidates the first variable,
  - instead of being called a shallow copy, it's known as a **move**
  - If we *do* want to **deeply copy** the heap data, not just the stack data, we can use a common method called **clone**.

Rust usa una cosa che nei linguaggi di programmazione è noto, si chiama shallow copy, quindi vuol dire che quando uno fa un operazione con puntatori sta in realtà copiando il valore e a questo punto l'effetto netto è che ho degli alias alla struttura. In Rust si chiama move. Nel linguaggio di programmazione voi sapete che esiste anche una deep copy, cioè l'entità il meccanismo che vuole separare esattamente due valori e copiarli. Allora in Rust si ottiene con un meccanismo che si chiama il cloning

# Deep copy retains ownership

Make clones (copies) to avoid ownership loss

```
let x = String::from("hello");
let y = x.clone(); //x no longer moved
println!("{} , world!", y); //ok
println!("{} , world!", x); //ok
```

associamo a y non x ma un clone di x quindi questo vuol dire che x continua a essere l'owner del valore poi abbiamo una copia di x esplicita e questa copia di x ha come owner y, quindi abbiamo due copie a questo punto che hanno la loro locazione di memoria differente sullo heap, due owner differenti e due cammini di accesso differenti, infatti quando voglio cercare di accedere a y mi dice ok come prima, ora quando cerco di accedere a x mi continua a dire ok perché? perché non viene persa l'ownership del dato

RUN ►

...

DEBUG ▾

STABLE ▾

...

```
1 fn main() {
2     let s1 = String::from("hello");
3     let s2 = s1.clone();
4
5     println!("s1 = {}, s2 = {}", s1, s2);
6 }
```

⋮⋮⋮

Execution

Standard Error

```
Compiling playground v0.0.1 (/playground)
Finished dev [unoptimized + debuginfo] target(s) in 1.42s
Running `target/debug/playground`
```

Standard Output

```
s1 = hello, s2 = hello
```

BUT ...

RUN ►

DEBUG ▾

STABLE ▾

...

```
1 fn main() {  
2     let x = 5;  
3     let y = x;  
4  
5     println!("x = {}, y = {}", x, y);  
6 }
```

se ne guarda bene di darmi errore perché? perché per i valori scalari primitivi viene fatta automaticamente la deep copy.

Standard Error

```
Compiling playground v0.0.1 (/playground)  
Finished dev [unoptimized + debuginfo] target(s) in 0.95s  
Running `target/debug/playground`
```

Standard Output

```
x = 5, y = 5
```

## Primitives copied automatically

- **i32, char, bool, f32, tuples of these types, etc.**

```
let x = 5;
let y = x;
println!("{} = 5!", y); //ok
println!("{} = 5!", x); //ok
```

*The reason is that types such as integers that have a known size at compile time are stored entirely on the stack, so copies of the actual values are quick to make.*

il passaggio dei parametri è un meccanismo di assegnamento dinamico del valore del parametro attuale al nome parametro formale, quindi, anche in quel caso abbiamo la gestione dell' ownership, esattamente con gli assegnamenti, come succede con gli insegnamenti proprio il fatto che il passaggio dei parametri è un assegnamento

# Parameter passing

- The parameter passing is an assignment of the value of the actual parameter to the formal parameter
- The move and copy concepts apply as well!!

La domanda che ci siamo posti alla fine della scorsa lezione era, che cosa succede per il passaggio dei parametri? perché il passaggio dei parametri altro non è che un assegnamento di natura dinamica che avviene tra il parametro formale e il valore del parametro attuale, perché nel corpo della funzione o della procedura o del metodo invocato a questo punto, il parametro attuale sarà il meccanismo che viene utilizzato per accedere al valore del parametro attuale, quindi, la nozione di move, quindi il passaggio della ownership e la nozione di copia, ovvero il fatto di fare invece una move fare una deep copy del valore si applicano esattamente al concetto di passaggio dei parametri, dobbiamo solo vedere col meccanismo del passaggio di parametri come questo avviene ed è una cosa che succede in tutti i linguaggi di programmazione il fatto di avere delle modalità di passaggio dei parametri che sostanzialmente influenzano il meccanismo di assegnamento dei valori.

sotto abbiamo la definizione di un'altra funzione che è sostanzialmente la funzione identità, la funzione identità qui è soltanto per far vedere che abbiamo un trasferimento di ownership dal chiamante al chiamato e poi anche da chiamato al chiamante al momento del ritorno del passaggio del parametro questo è il senso di questo esempio non significativo per quanto calcola ma per quali sono le caratteristiche del passaggio dei parametri. A questo punto quello che succede quando noi invochiamo da rust la funzione identità con parametro s1 abbiamo la nozione di movimento, per cui a questo punto vuol dire che il parametro formale, l'argomento s è diventato il possessore del valore della stringa s1 che è hello, quindi abbiamo esattamente la move. Quando restituisce il controllo al chiamante abbiamo il passaggio, il movimento opposto, cioè dal chiamato al chiamante abbiamo questo trasferimento di ownership a s2 a questo punto una volta che abbiamo fatto questa operazione utilizziamo s2. La cosa più semplice in questo esempio dovendo stare in una pagina è fare una stampa? Ma cosa succede ad s1? s1 è ancora all'interno dello scope bene, s1 quando lo invochiamo il compilatore fallisce dice che c'è un errore proprio per il fatto che abbiamo trasmesso l'identità e quindi non possiamo più accedere tramite s1. Le informazioni che ci dà il compilatore di rust sono significative ad esempio ci dice che stiamo facendo un trasferimento di ownership con s1 a questo punto, con la chiamata abbiamo che il valore è stato spostato dal parametro attuale s1 al parametro formale che se non ricordo male è s e a questo punto abbiamo un trasferimento indietro pertanto quando noi cerchiamo di accedere con s1 il compilatore ci dice, guarda, tu hai trasferito e hai perso l'ownership del valore e quindi "ti attacchi" il fatto che c'è scritto "borrow of moved value" guarda hai programmato male le regole del trasferimento sono tali che a questo punto non puoi più muovere s1.

## On a call, ownership passes from:

- argument to called function's parameter
- returned value to caller's receiver

```
fn main() {  
    let s1 = String::from("hello");  
    let s2 = id(s1); //s1 moved to arg  
    println!("{}" ,s2); //id's result moved to s2  
    println!("{}" ,s1); //fails  
}  
fn id(s:String) -> String {  
    s // s moved to caller, on return  
}
```

RUN ►

DEBUG ▾

STABLE ▾

...

```
1 fn main() {
2     let s1 = String::from("hello");
3     let s2 = id(s1);
4     println!("{}" ,s1);
5 }
6
7 fn id(s:String) -> String {
8     s
9 }
```

⋮⋮⋮

### Execution

```
|  
= note: `#[warn(unused_variables)]` on by default  
  
error[E0382]: borrow of moved value: `s1`  
--> src/main.rs:4:15  
|  
2 |     let s1 = String::from("hello");  
|         -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait  
3 |     let s2 = id(s1);  
|             -- value moved here  
4 |     println!("{}" ,s1);  
|             ^^ value borrowed here after move
```

```

fn main() {
    let s = String::from("hello"); // s comes into scope

    takes_ownership(s);        // s's value moves into the function...
                                // ... and so is no longer valid here

    let x = 5;                 // x comes into scope

    makes_copy(x);            // x would move into the function,
                            // but i32 is Copy, so it's okay to still
                            // use x afterward

} // Here, x goes out of scope, then s. But because s's value was mo
   // special happens.

```

```

fn takes_ownership(some_string: String) {
    println!("{}", some_string);
} // Here, some_string goes out of scope and `drop` is called. The backing
   // memory is freed.

```

```

fn makes_copy(some_integer: i32) {
    println!("{}", some_integer);
} // Here, some_integer goes out of scope. Nothing special happens.

```

per quanto riguarda i valori primitivi l'assegnamento e quindi anche il passaggio dei parametri fanno una deep copy quindi c'è una copia dal valore del parametro formale a come viene utilizzato nel parametro attuale, sono due mondi separati visto che c'è una deep copy. Con la takes\_ownership(s) abbiamo il trasferimento la move da s al parametro formale e a questo punto fa perdere l' ownership, quando viene restituito il controllo, abbiamo perso la ownership del valore associato a s, la cosa non vale per l'intero 5, perché? E' un valore primitivo, per cui quando facciamo il passaggio dei parametri facciamo una deep copy a questo punto è vero che il valore associato a x viene trasferito, ma si mantiene la ownership di quel valore e a questo punto si può utilizzare in seguito il valore associato a x, perché x non ha perso l' ownership di quel valore e quindi questo vi fa vedere la differenza tra il fatto di passare ed effettuare una copia e il fatto di trasmettere la ownership.

```
fn main() {  
    let s1 = gives_ownership();      // gives_ownership moves its return value into s1  
    let s2 = String::from("hello");  // s2 comes into scope  
    let s3 = takes_and_gives_back(s2); // s2 is moved into takes_and_gives_back, which also  
                                    // moves its return value into s3  
} // Here, s3 goes out of scope and is dropped. s2 goes out of scope but was  
// moved, so nothing happens. s1 goes out of scope and is dropped.  
  
fn gives_ownership() -> String {      // gives_ownership will move its return value into the calling function  
  
    let some_string = String::from("hello"); // some_string comes into scope  
  
    some_string                      // some_string is returned  
}  
  
fn takes_and_gives_back(a_string: String) -> String {  
  
    a_string // a_string is returned and moves out to the calling function  
}
```