Rust

Rust è un linguaggio di sistema (come C/C++, il compilato può girare sul bare metal, senza il supporto di un sistema operativo).

Nasce all'interno della Mozilla foundation, con forti influenze da OCaml e C++. Proprio in OCaml viene scritto il suo primo compilatore, anche se ora il codice viene compilato in Rust stesso (con LLVM come backend).

Come tutti i linguaggi a livello di sistema **non è presente un runtime** (in realtà è presente un runtime minimo), infatti non utilizza system threads e non ha un garbage collector.

Inoltre, è presente una **zero-cost abstraction**, con polimorfismo parametrico via monomorfizzazione. Il complesso sistema di ownership dei dati in memoria è interamente imposto a compile-time.

Viene garantita la **memory safety**, non avendo memory leaks, double deallocation, dangling pointes e data races.

Per concludere, è presente un sistema di **fearless concurrency**, dove il type system (insieme agli smart pointers) minimizzano i problemi legati alla concorrenza.

Sono inoltre presenti anche alcune features aggiuntive:

- Chiusure.
- Algebraic Data Types e Pattern Matching.
- No NULL values, vengono usati *option types*.
- Polimorfimo parametrico bounded, usando templates/generics con bounds con i traits.
- Traits, inizialmente con classi (poi rimosse) + Trait Objects, per dynamic dispatch + Trait Bounds + Associated Types.
- Moduli annidabili.

Rust è diventato famoso nel tempo per la sua **gestione della memoria**.

In Rust sono presenti due meccanismi ortogonali, che nascono per essere indipendenti ma finiscono per interagire l'uno con l'altro. I due meccanismi sono:

1. Complesso sistema di **ownership** con **borrowing** in **lettura** e/o **scrittura**.
2. **Smart pointers**, qualora il primo meccanismo diventi troppo complesso da gestire.

Primo meccanismo (ownership + borrowing)

Ogni cella di memoria sullo heap ha un owner, che è responsabile per la sua deallocazione.

Quando una cella sullo heap viene creata e un puntatore a esso viene assegnato a una **variabile sullo stack**, quest'ultima ne diventa l'**owner**.

Quando invece il puntatore viene assegnato a **un'altra cella sullo heap**, questa ne diventa l'**owner**.

Quando un owner viene **deallocato** (ad esempio, il blocco di una variabile sullo stack viene dellocato) **le celle RICORSIVAMENTE possedute vengono rilasciate**. Il codice per la deallocazione viene inserito alla fine del blocco (delimitato dalle parentesi graffe).

Una cella sullo **heap** ha sempre **uno e un solo owner**. **Assegnamenti** e **passaggio come parametri** della variabile/cella che ha l'ownership trasferiscono (**move**) l'ownership.

Quando una variabile perde l'ownership, essa non può più essere utilizzata!

Si osservino ora alcuni esempi di codice.

In questo primo esempio è possibile osservare il funzionamento del sistema di ownership:

```
fn main() {
 // x è unboxed, quindi sullo stack
 let x = 4;

 // s sullo stack punta a una stringa nello heap
 let s = String::from("ciao");

 let y = x;

 let t = String::from("ciao");

 println!("x = {}, y = {}", x, y);
 println!("s' = {}, t = {}", s, t);
}
```

`let y = x;` copia il valore 4 da `x` ad `y`. La variabile `x` rimane viva!

Vediamo ora una seconda versione modificata:

```
fn main() {
 // x è unboxed, quindi sullo stack
 let x = 4;

 // s sullo stack punta a una stringa nello heap
 let s = String::from("ciao");

 let y = x;

 let t = s;

 println!("x = {}, y = {}", x, y);
 println!("s' = {}, t = {}", s, t);
}
```

In questo caso, con `let t = s;`, l'ultima riga è errata. `s` non ha più l'ownership, è diventata una variabile morta (l'ownership è di `t`)!

Si osservi ora un secondo esempio, vedendo il passaggio di ownership attraverso l'utilizzo di funzioni:

```
fn main() {
 let s1 = gives_ownership();        // ownership taken
 let s2 = String::from("hello");    // ownership taken
 let s3 = takes_and_gives_back(s2); // s2 looses ownership; s3 takes it
} // the strings pointed to by s1 and s3 are deallocated

fn gives_ownership() -> String {            // String is a pointer!
 let some_string = String::from("hello");   // allocated here
 some_string                                // ownership transferred
}

fn takes_and_gives_back(a_string: String) -> String { // ownership taken
 a_string                                        // ownership transferred
}
```

`gives_ownership()` alloca una nuova stringa nello heap e ne trasferisce l'ownership ad `s1`.

`takes_and_gives_back(...)` prende momentaneamente l'ownership della stringa, per poi restituirla ad `s3`.

In questo caso, alla fine del blocco verrà aggiunto dal compilatore il codice per deallocare `s1` ed `s3`.

Si introduce ora il concetto di **References**.
Questo sistema può essere visto come un sistema di prestiti della ownership.

`&x` è una reference a `x` (o al suo contenuto).
`&mut x` è una reference a `x` (o al suo contenuto) che permette di modificarne il contenuto.

Se `x` ha tipo `T`, `&x` ha tipo `&T` e `&mut x` ha tipo `&mut T`.

Prendere una reference di una variabile implica fare **borrowing** della variabile. Questo permette di **non** avere **data races** (anche concorrentemente), infatti:

- Se una variabile è borrowed **mutably**, **nessun altro borrow è possibile** e l'owner è **frozen** (non può accedere alla variabile fino a quando il borrowing termina).
- Se l'ownership è **mutable** e la variabile viene borrowed, l'owner è **frozen** (non può modificare la variabile fino a quando il borrowing termina).

Si osservino ora alcuni esempi di codice.

In questo primo esempio l'obiettivo è osservare il sistema di borrowing:

```
fn main() {
 let x = 4;
 let y = &x;

 let t = String::from("ciao");      // takes immutable ownership
 let s = &t;                        // borrows immutably

 println!("x = {}, y = {}", x, y);
 println!("s = {}, t = {}", s, t);
}
 // end of borrowing (s goes out of scope) and end of ownership
 // (t goes out of scope and the string is deallocated)
```

Il compilatore controlla partendo dal basso verso l'alto ed aggiunge il codice necessario.

Vediamo ora vari esempi di borrowing (con errori di compilazione e non):

```
fn main() {
 let mut x = 4;
 let y = &x;
 x = 5;             // error: assignment to borrowed ‘x‘
}
```

In questo caso, il compilatore restituirà un errore!

```
fn main() {
 let mut x = 4;
 { let y = &x; }    // ok: the borrow ends at the end of inner block!
 x = 5;
}
```

In questo caso, il compilatore non restituirà alcun errore, avendo terminato il borrow all'interno di uno scope interno.

```
fn main() {
 let x = 4;
 let z = &mut x; // error: cannot borrow immutable local var. as mutable
}
```

In questo caso, non si può fare borrowing mutabile di una variabile locale immutabile.

```
fn main() {
 let mut x = 4;
 let y = &x;
 let z = &mut x;    // error: cannot borrow as mutable because it is
                    // also borrowed as immutable
}
```

In questo caso, non è possibile effettuare più prestiti (è già presente un prestito immutabile).

```
fn main() {
 let mut x = 4;
 let y = &mut x;
 let z = &mut x;    // error: cannot borrow as mutable more
                    // than once at a time
}
```

In questo caso, non è possibile effettuare più prestiti mutabili.

```
fn increment(x: &mut i32) { // syntactic sugar at work! See later
 *x = *x + 1;
}

fn main() {
 let mut x = 4;             // x has mutable ownership
 increment(&mut x);         // mutable borrows begins and ends
 x = x + 1;                 // so x can be used again here
 println!("x = {}", x);
}
```

In questo caso, l'ownership di `x` viene passata alla funzione `increment(...)`. Terminata la funzione, l'ownership tornerà alla variabile `x` che potrà essere nuovamente utilizzata.

Nota Bene: il compilatore modifica le euristiche valide di versione in versione. Alcuni di questi esempi potrebbero avere esiti differenti in base alla versione utilizzata!

Si introduce ora il concetto di **Lifetimes**.
Finora era sempre la parentesi graffa chiusa che determinava la morte dell'uso di una variabile. In realtà, le celle di memoria hanno un **lifetime** che indica quando la cella verrà deallocata dall'owner.

Il lifetime è diverso dallo scope, come, ad esempio, nel caso in cui l'ownership venga trasferita.

Ogni reference ha di fatto **due** lifetime: quello della reference e quello di ciò a cui la reference punta.

Questo risolve il problema dei **dangling pointers**: Rust verifica che il primo lifetime sia

sempre inferiore al secondo (sintassi concreta: ‘a : ‘b per ‘a,’b variabili di lifetime con significato “’a termina dopo ‘b”).

L’unico termine costante di tipo lifetime è `‘static` (vivo fino al termine del programma, ad esempio una variabile globale). Le variabili di lifetime vengono indicate con `‘a`, `‘b`, ..., che stanno ad indicare alpha, beta, ...

I lifetime vengono utilizzati in due contesti:

- Template astratti su variabili di lifetime + bound (polimorfismo bounded).
- Reference tipate con il lifetime (ad esempio, `&’a i32` reference a un `i32` di lifetime `‘a`).

E’ presente un meccanismo di **elisione**: ove non necessari i lifetime si possono non esplicitare!

Si osservino ora alcuni esempi di codice.

In questo primo esempio l’obiettivo è osservare il classico problema dei dangling pointers:

```
fn main() {
 let reference_to_nothing = dangle();
}

fn dangle<’a>() -> &’a String {
 let s = String::from("hello");
 &s
} // error: the lifetime of s ends here and it should end at ‘a
```

La funzione `dangle()` restituisce un riferimento ad una stringa.
E’ quindi presente un errore, dato che il lifetime di `s` termina alla fine della funzione (lifetime *now* arbitrario), ma dovrebbe terminare a lifetime `’a`.

```
// i lifetime ’b e ’c devono terminare dopo il lifetime di ’a
fn max<’a,’b : ’a,’c : ’a>(x: &’b i32, y: &’c i32) -> &’a i32 {
 std::cmp::max(x,y)
}

fn main() {
 //let z;           // error se z è dichiarato prima di x o y

 let x = 4;
 let y = 3;

 let z;             // ok se z è dichiarato dopo x,y
 z = max(&x, &y);
 println!("max = {}", z);
} // i lifetime finiscono in ordine inverso di dichiarazione
```

In questo caso, la variabile `z` va dichiarata dopo le variabili `x` e `y`.