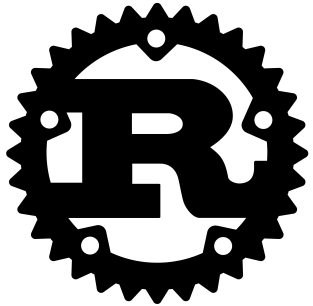


RUST

Introduzione al linguaggio

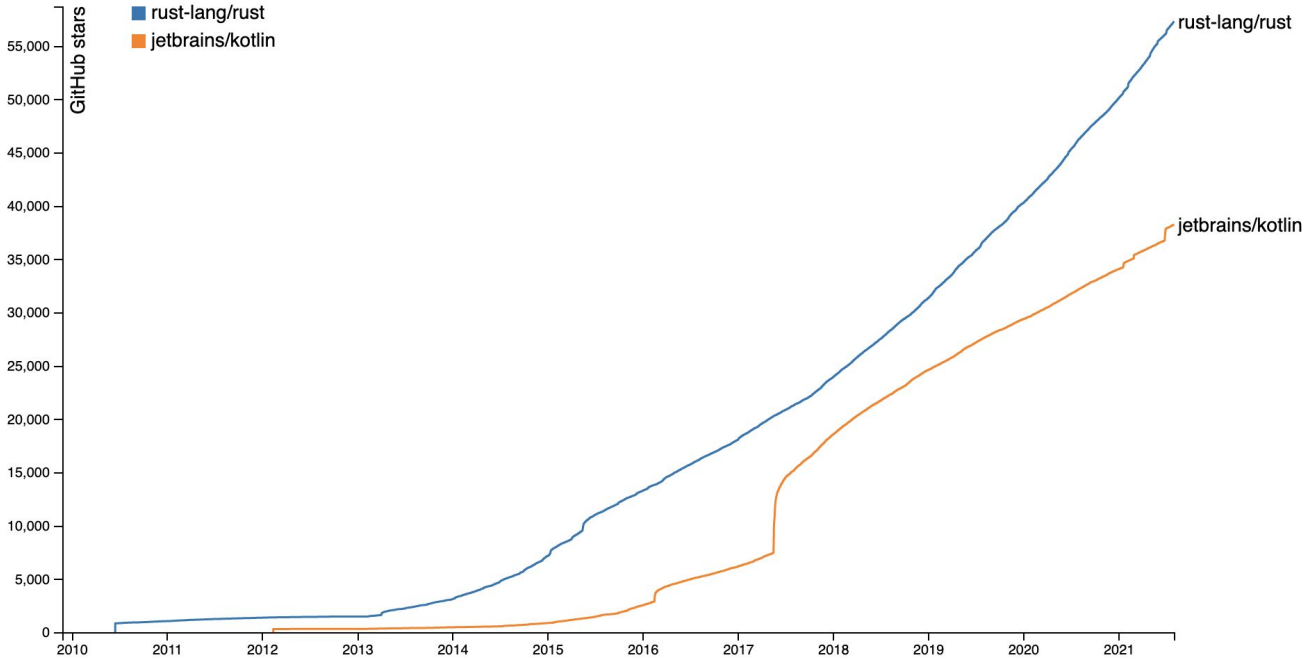
Introduzione

- Rust è un linguaggio di programmazione moderno focalizzato sulla **correttezza**, **velocità** e sul supporto alla **programmazione concorrente**
  - Tali obiettivi vengono perseguiti mantenendo ad un livello minimo le librerie di supporto in fase di esecuzione, in particolare senza usare tecniche di garbage collection né fare assunzioni sulla struttura dell'ambiente di esecuzione diverse da quelle del linguaggio C
  - Questo rende un programma scritto in Rust adatto ad essere eseguito in una varietà di contesti, dai sistemi embedded al kernel di un sistema operativo, dalle applicazioni lato server all'implementazione dei browser o di loro moduli
- Rust è un linguaggio **staticamente** e **fortemente tipato**, adatto alla programmazione di sistema
  - Tutti i tipi sono noti in fase di compilazione
  - Un sofisticato motore di inferenza viene usato per validare le proprietà dei tipi nel contesto del programma e ridurre significativamente il rischio di errori
  - Permette un controllo totale dell'uso della memoria e ottimizza al massimo il codice generato

Storia

- Nato nel 2006 come progetto personale di Graydon Hoare, un dipendente di Mozilla Research, nel 2009 ha ricevuto una prima sponsorizzazione da parte di Mozilla, poi è diventato open source
  - Rust 1.0 - 15 maggio 2015
  - Versioni successive rilasciate costantemente ogni 6 settimane
  - Rust 1.66.0 - 12 dicembre 2022
- Usato in produzione in molti contesti
  - Firefox contiene 3M LoC scritte in Rust (9%)
  - Dropbox usa Rust nel suo motore di sincronizzazione
  - npm usa Rust per implementare il servizio di autorizzazione del registry
  - deno, il successore di node.js, è scritto in Rust
  - Amazon Web Services usa Rust (Firecracker project) per ottenere alte prestazioni in servizi come Lambda, EC2, S3,...
  - Microsoft ha portato tutta l'API Windows in Rust (https://github.com/microsoft/windows-rs)

Evoluzione



https://github.com/dtolnay/star-history

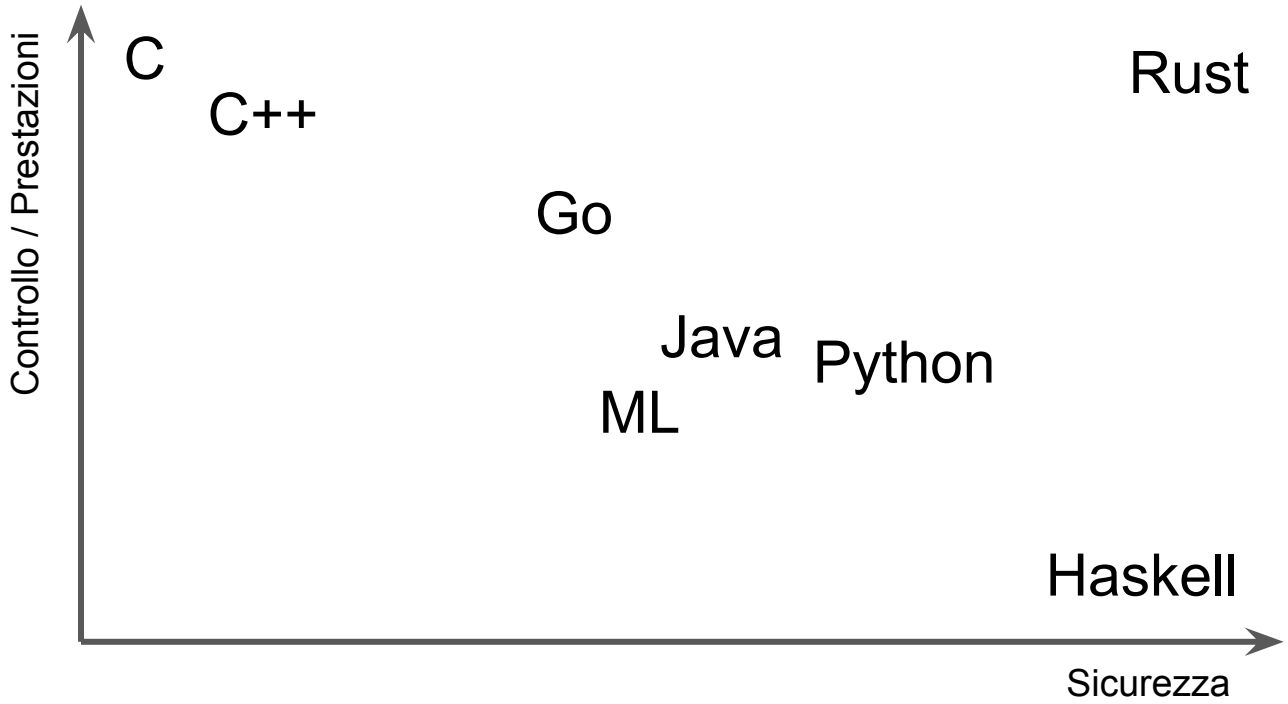
Obiettivi del linguaggio

- Offrire un linguaggio per la programmazione di sistema **privo di comportamenti non definiti**, **concorrente**, **pratico**
  - I linguaggi esistenti, ad un livello di astrazione ed efficienza simile, sono soggetti a molteplici limiti:
    - Poca attenzione alla sicurezza (safety) dei costrutti
    - Scarso supporto alla correttezza formale dell'esecuzione concorrente
    - Mancano di strumenti di corredo a supporto della creazione, condivisione e messa in campo
    - Offrono un controllo limitato sull'uso delle risorse computazionali
- Offrire **astrazioni a costo nullo** per la maggior parte degli idiomi di programmazione
  - Permettendo al programmatore di adottare lo stile preferito nella scrittura di un algoritmo (iterazione, ricorsione, chiusure, …) e garantendo la generazione del miglior codice assembler possibile, senza introdurre costi aggiuntivi per funzionalità non richieste
  - *https://boats.gitlab.io/blog/post/zero-cost-abstractions/*
- Supportare la **produttività del programmatore**
  - Offrendo costrutti di alto livello ed un ecosistema di compilazione/gestione delle dipendenze/test integrato

Presupposti di base

- Linguaggio compilato (non basato su bytecode)
- Fortemente tipizzato in fase di compilazione
- Paradigma imperativo, ma con aspetti funzionali
- Non ha né garbage collection né ambiente di supporto all'esecuzione
- Sistema dei tipi sofisticato

Confronto



vs Python

- Molto più veloce
  - Normalmente destinato a contesti differenti
- Minor consumo di memoria
- Vero multi-threading
  - Parzialmente disponibile anche in Python v.3.12
- Tipi algebrici
- Approccio all'ereditarietà differente
- Pattern matching
  - Introdotto in Python v.3.10
- Linguaggio staticamente orientato ai tipi: molti meno arresti anomali in fase di esecuzione
  - La gestione dinamica dei tipi usata da Python porta l'interprete ad accettare costrutti che si rivelano incoerenti solo in fase di esecuzione

vs Java

- Nessun overhead causato dalla JVM
  - non si verificano pause causate dal Garbage Collector
- Minor consumo di memoria
- Nessun costo di astrazione
- Approccio all'ereditarietà ed alla programmazione generica differente
- Pattern matching
- Sistema di compilazione unico
- Gestione delle dipendenze integrata

vs C/C++

- Nessun segmentation fault
- Nessun buffer overflow
- Nessun null pointer
- Nessuna corsa critica
- Sistema dei tipi più elaborato
- Approccio all'ereditarietà differente
- Processo di costruzione unificato
- Gestione delle dipendenze integrata

vs Go

- Nessuna pausa a causa del Garbage Collector
- Minor consumo di memoria
- Nessun null pointer
- Migliore gestione degli errori
- Programmazione concorrente sicura
- Sistema dei tipi più solido
- Approccio all'ereditarietà differente
- Nessun costo di astrazione
- Gestione delle dipendenze

Confronto (prestazioni *)

Total					
	Energy		Time		Mb
(c) C	1.00	(c) C	1.00	(c) Pascal	1.00
(c) Rust	1.03	(c) Rust	1.04	(c) Go	1.05
(c) C++	1.34	(c) C++	1.56	(c) C	1.17
(c) Ada	1.70	(c) Ada	1.85	(c) Fortran	1.24
(v) Java	1.98	(v) Java	1.89	(c) C++	1.34
(c) Pascal	2.14	(c) Chapel	2.14	(c) Ada	1.47
(c) Chapel	2.18	(c) Go	2.83	(c) Rust	1.54
(v) Lisp	2.27	(c) Pascal	3.02	(v) Lisp	1.92
(c) Ocaml	2.40	(c) Ocaml	3.09	(c) Haskell	2.45
(c) Fortran	2.52	(v) C#	3.14	(i) PHP	2.57
(c) Swift	2.79	(v) Lisp	3.40	(c) Swift	2.71
(c) Haskell	3.10	(c) Haskell	3.55	(i) Python	2.80
(v) C#	3.14	(c) Swift	4.20	(c) Ocaml	2.82
(c) Go	3.23	(c) Fortran	4.20	(v) C#	2.85
(i) Dart	3.83	(v) F#	6.30	(i) Hack	3.34
(v) F#	4.13	(i) JavaScript	6.52	(v) Racket	3.52
(i) JavaScript	4.45	(i) Dart	6.67	(i) Ruby	3.97
(v) Racket	7.91	(v) Racket	11.27	(c) Chapel	4.00
(i) TypeScript	21.50	(i) Hack	26.99	(v) F#	4.25
(i) Hack	24.02	(i) PHP	27.64	(i) JavaScript	4.59
(i) PHP	29.30	(v) Erlang	36.71	(i) TypeScript	4.69
(v) Erlang	42.23	(i) Jruby	43.44	(v) Java	6.01
(i) Lua	45.98	(i) TypeScript	46.20	(i) Perl	6.62
(i) Jruby	46.54	(i) Ruby	59.34	(i) Lua	6.72
(i) Ruby	69.91	(i) Perl	65.79	(v) Erlang	7.20
(i) Python	75.88	(i) Python	71.90	(i) Dart	8.64
(i) Perl	79.58	(i) Lua	82.91	(i) Jruby	19.84

*** da:** Rui Pereira, et. all. 2017. **Energy efficiency across programming languages: how do energy, time, and memory relate?** In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering* (*SLE 2017*). Association for Computing Machinery, New York, NY, USA, 256–267. DOI:https://doi.org/10.1145/3136014.3136031

Controllo

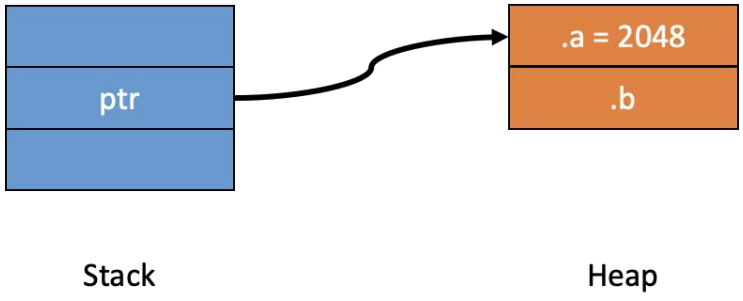
```
typedef struct Dummy { int a; int b; } Dummy;

void foo(void) {
    Dummy *ptr = (Dummy *) malloc(sizeof(struct Dummy));
    ptr->a = 2048;
    free(ptr);
}
```

Disposizione in memoria precisa

Riferimento leggero

Rilascio deterministico



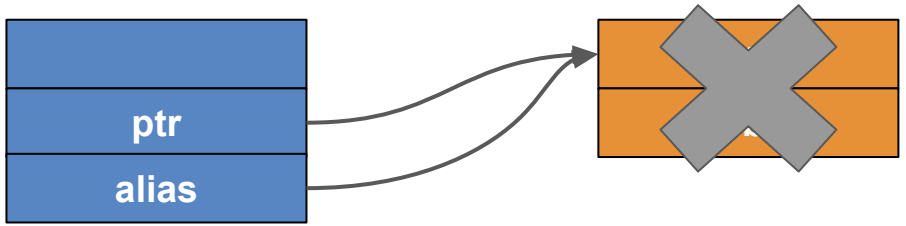
Sicurezza

```
typedef struct Dummy { int a; int b; } Dummy;

void foo(void) {
    Dummy *ptr = (Dummy *) malloc(sizeof(struct Dummy));
    Dummy *alias = ptr; //Pericolo!!
    free(ptr);
    int a = alias->a;
    free(alias);
}
```

Uso di blocco rilasciato!

Doppio rilascio!



Dangling pointer!

Sicurezza

- Rust si appoggia ad un sistema di validazione dei tipi in fase di compilazione che impedisce (per i programmi che non fanno uso delle estensioni *unsafe*) i seguenti tipi di errore:
  - **Dangling pointer** - uso di puntatori ad aree di memoria GIÀ rilasciate
  - **Doppi rilasci** - tentativo di restituire al S.O. un'area di memoria già rilasciata
  - **Corse critiche** - accesso a dati il cui contenuto è indeterminato a seguito di eventi fuori dal controllo del programma stesso (ordine di schedulazione, attese legate all'I/O, …)
  - **Buffer overflow** - tentativi di accedere ad aree di memoria contigue a quelle possedute da una variabile, ma non di sua pertinenza
  - **Iteratori invalidi** - accesso iterativo (uno alla volta) agli elementi contenuti in una collezione che viene modificata mentre l'iterazione è in corso
  - **Overflow aritmetici** (solo in modalità debug) - esecuzione di operazioni aritmetiche che, a seguito della limitata capacità di rappresentazione dei numeri binari, portano ad errori grossolani (2_000_000_000 + 2_000_000_000 = -294_967_296)
- Il linguaggio favorisce l'uso di costrutti immutabili e propone convenzioni volte a limitare il rischio di compromissione dei dati

Prestazioni

- Il compilatore ottimizza in modo aggressivo **dimensioni** e **velocità** del codice generato
  - Le strutture dati base del linguaggio sono pensate per favorire l'**uso della memoria cache**, preferendo l'uso di array a più articolate strutture basate su puntatori che limitano la località dei riferimenti, vanificando i benefici statistici offerti dalla cache
  - La politica di invocazione standard è basata su **indirizzi statici**, che possono essere ottimizzati inline, piuttosto che su meccanismi polimorfici che richiedono l'uso di invocazioni indirette, non ottimizzabili
  - La presenza di un sistema integrato per la **gestione delle dipendenze** e dei moduli facilita la condivisione del codice e l'accesso ad un vasto panorama di librerie open source che spesso offrono soluzioni allo stato dell'arte per molti problemi comuni

Linguaggio moderno

- Tipi generici ben implementati
  - Segue l'approccio simile al C++ della monomorfizzazione e non quello usato in Java della cancellazione del tipo
- Tipi algebrici e pattern
  - Supporto di tipi "prodotto" (struct e tuple) e di tipi somma (enum) (https://justinpombrio.net/2021/03/11/algebra-and-data-types.html)
  - Il costrutto *match ...* unisce il controllo dell'esaustività del dominio di un'espressione con la potenza espressiva della destrutturazione sintattica, consentendo di esprimere – in forma sintetica e facilmente leggibile – flussi di controllo molto articolati
- Strumentazione moderna
  - Il programma cargo offre supporto per automatizzare l'intero ciclo di vita del software, dalla compilazione alla gestione delle dipendenze, dall'esecuzione dei test alla profilazione

Tipi generici ben implementati

```
struct MyVec<T> {
  // ...
}

impl<T> MyVec<T> {
  pub fn find<P>(&self, predicate: P) -> Option<&T>
  where P: Fn(&T) -> bool {
    for v in self {
      if predicate(v) { return Some(v); }
    }
    None
  }
}
```

Tipi algebrici e pattern

```
enum HttpRequest {
  Get, Post(String), Put(String), Unknown
}

fn process(req: HttpRequest) {
  match req {
    HttpRequest::Get => { /* handle get request */ },
    HttpRequest::Post(data) | HttpRequest::Put(data)
    if !data.is_empty() => {
       // process data
    }
    _ => { /* manage error */ }
  }
}
```

Tipi algebrici e pattern

```
// Option<T> è un enum che può essere Some(T) o None
if let Some(f) = my_vec.find(|t| t >= 42) { /* found */ }

enum DecompressionResult {
    Finished { size: u32 },
    InputError(std::io::Error),
    OutputError(std::io::Error),
}
// errore in fase di compilazione: manca caso generico
match decompress() {
  Finished { size } => { /* analizzato con successo */ }
  InputError(e) if e.is_eof() => { /* gestisco EOF */ }
  OutputError(e) => { /*output fallisce con l'errore e*/ }
}
```

Testing e documentazione integrati

```
#[test]
fn it_works() {  assert_eq!(1 + 1, 2);
}

/// Returns one more than its argument.
///
/// ```
/// assert_eq!(one_more(42), 43);
/// ```
pub fn one_more(n: i32) -> i32 {   n + 1
}
```

Sicuro per definizione

- Il linguaggio introduce in modo esplicito il concetto di **possesso di un valore**
  - In ogni istante, ciascun valore è posseduto da una e una sola variabile
  - E' possibile trasferire il possesso di un valore ad un'altra variabile (movimento)
  - E' possibile concedere temporaneamente l'accesso ad un valore tramite l'uso di riferimenti, a patto di non modificare tale valore mentre il riferimento esiste
- Puntatori **controllati** in fase di compilazione
  - Il concetto di proprietà del dato è esteso ai dati cui si accede in modo indiretto, sia per riferimento (senza trasferimento della proprietà) che tramite puntatore
- La **sicurezza dei thread** è incorporata nel sistema dei tipi
  - Il compilatore conosce quali tipi possono essere trasferiti da un thread ad un altro e quali puntatori possono essere usati in modo sicuro tra thread
  - Il concetto di "tempo di vita" associato a ciascun valore permette di individuare in fase di compilazione quali operazioni sono lecite e quali portano ad una violazione
- Nessun **stato nascosto**
  - La presenza di errori e l'opzionalità sono modellate esplicitamente e richiedono il controllo esplicito da parte del programmatore

Possesso di un valore

```
//v possiede il valore in lettura e scrittura
let mut v = Vec::new();
//Il valore di v può essere modificato
v.push(1);

//movimento: ora v1 possiede il valore che prima era in v
let mut v1 = v;

//Il compilatore impedisce l'accesso tramite v
v.push(2);

//Ora v1 contiene [1, 3]
v1.push(3);
```

Possesso di un valore

```
//v possiede il valore in lettura e scrittura
let mut i = 12;

//r è un riferimento ad i: ha in PRESTITO il suo valore
let r = &i;

//mentre r esiste, non è lecito modificare i
i = 23;

//accedo al valore a cui r fa riferimento
println!("{}", *r);
```

Puntatori controllati in fase di compilazione

```
let v = Vec::new();

// la compilazione avviene correttamente:
println!("len: {}", v.len());

// la compilazione non avviene correttamente:
// richiede un accesso mutabile
 v.push(42);
```

Puntatori controllati in fase di compilazione

```
// Ogni valore ha un proprietario, responsabile del rilascio (RAII)
// il compilatore verifica che ci sia sempre un solo possessore
// e che non avvengano doppi rilasci
let x: Vec<i32> = Vec::new();
// y ora possiede il valore contenuto in x
let y = x;
drop(x); // illegale, y è ora il proprietario

// Nessun puntatore vive dopo le modifiche o il rilascio
// Nessun dangling pointer/use-after-free
let mut x = vec![1, 2, 3];
let first = &x[0];
let y = x;
println!("{}", *first); // illegale, first diventa invalido
//quando x è stata mossa
```

Puntatori controllati in fase di compilazione

```
let v = Vec::new();
accidentally_modify(&v);

fn accidentally_modify(v: &Vec<i32>) {
  println!("len: {}", v.len());
  // la compilazione non avviene correttamente;
  // è necessario un &mut Vec<i32>
  push(v);
}

// dichiarazione esplicita di un puntatore mutabile
 fn push(v: &mut Vec<i32>) {
  v.push(42);
}
// anche questo non verrà compilato: v non è mutabile
push(&mut v);
```

La sicurezza dei thread è incorporata nel sistema dei tipi

```
use std::cell::Rc; // reference-counted, non atomico
use std::sync::Arc; // reference-counted, atomico
// non compila:
let rc = Rc::new("not thread safe");
std::thread::spawn(move || {
    println!("I have an rc with: {}", rc); });
// compila correttamente:
let arc = Arc::new("thread safe");
std::thread::spawn(move || {
        println!("I have an arc with: {}", arc); });
// non compila:
let mut v = Vec::new();
std::thread::spawn(||{
                v.push(42); });
let _ = v.pop();
```

Nessuno stato nascosto

```
enum Option<T> {  Some(T),  None, }

enum Result<T,E> { Ok(T), Err(E), }


// r è Option<&T>, non &T –
//non può essere utilizzato senza verificare la presenza di None
let r = my_vec.find(|t| t >= 42);

//n è di tipo Result: per accedere al valore occorre verificare
//che non sia un Err(_)
let n = "42".parse();

// ? Suffisso che forza un ritorno in caso di errore

let n = "42".parse()?;
```

Controllo a basso livello

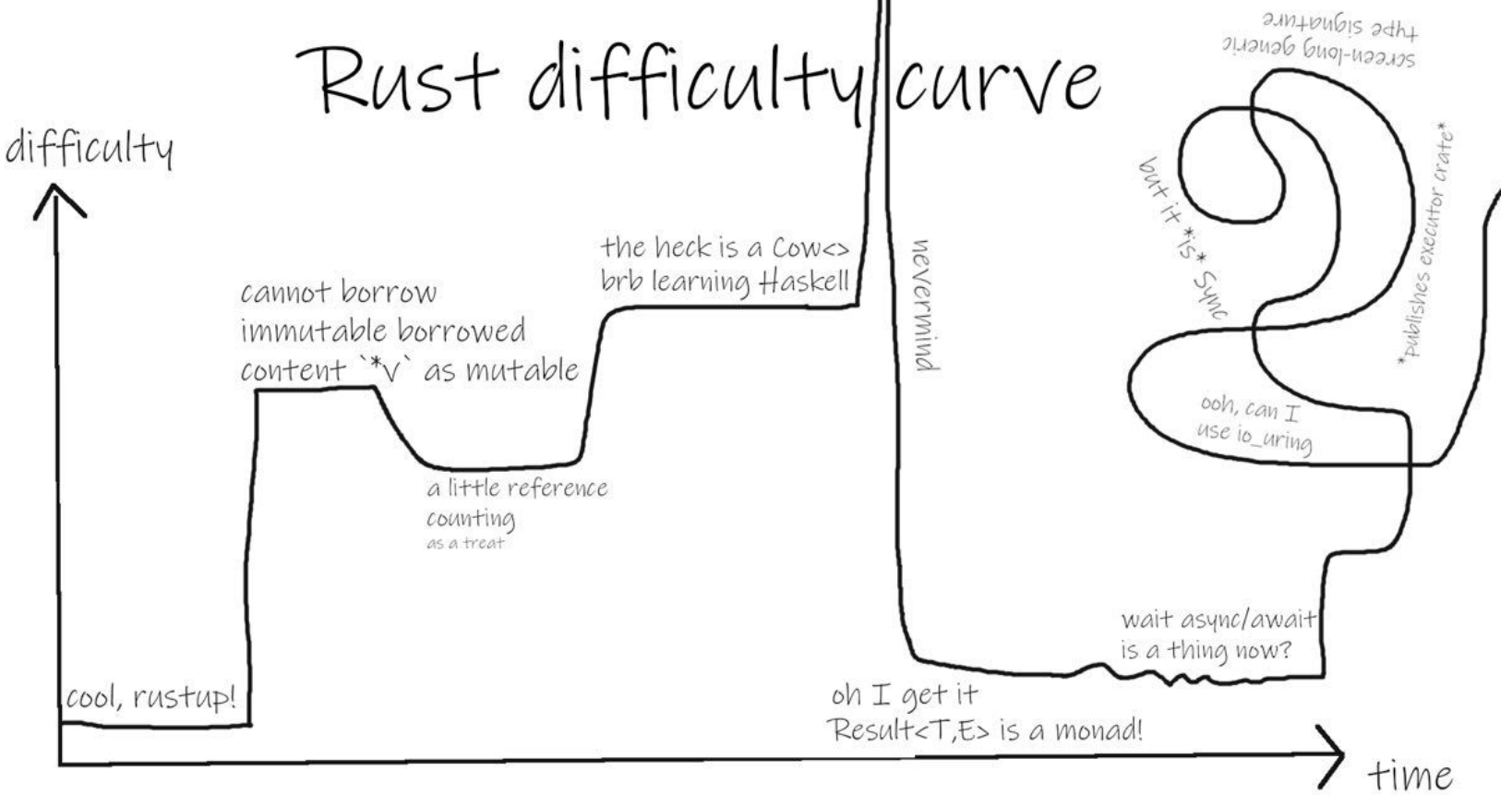
- La memoria è rilasciata non appena una variabile esce di visibilità
  - Nessuna interruzione a causa del garbage collection
- Nessuno spreco di memoria
- E' possibile invocare system call (tra cui fork/exec)
- Può essere eseguito su dispositivi senza sistema operativo
- Chiamate FFI verso altri linguaggi (C ABI)

Controllo dell'allocazione e delle chiamate dinamiche

```
// Si può allocare nello heap
let heap_x = Box::new(x);
let heap_z = vec![0; 1024];

//Si può cambiare l'allocatore
#[global_allocator]
static A: MyAllocator = MyAllocator;

// Si può attivare la chiamata dinamica(vtable):
// solo una copia di find per T
impl<T> MyVec<T> {
  pub fn find(&self, f: &dyn Fn(&T) -> bool) -> Option<&T> {
    // ...
  }
}
```



https://gist.github.com/humb1t

Installare Rust

- Si accede al sito https://rustups.rs e si seguono le istruzioni riportate
  - Per i sistemi operativi di derivazione Unix (Linux, MacOS, …) questo equivale a scaricare ed eseguire uno script che provvede a scaricare la toolchain e a configurare l'ambiente di esecuzione
  - Nel caso di Windows, vengono forniti due eseguibili, entrambi chiamati rustup-init.exe, destinati ad installare la toolchain nelle versioni a 64 o 32 bit
- E' possibile avere un assaggio della programmazione in Rust senza dover installare nulla tramite il sito https://play.rust-lang.org/
  - Offre un IDE web con la possibilità di accedere ad oltre 250 crates aggiuntivi mediante il quale è possibile scrivere, compilare ed eseguire codice Rust con alcune limitazioni (nessuna connessione di rete, memoria e disco limitati, vincoli sul tempo massimo di compilazione ed esecuzione)
  - I programmi scritti tramite il playground possono essere esportati sotto forma di GitHub GIST

Installare Rust

- Poiché il linguaggio (così come la libreria standard) viene aggiornato ogni 6 settimane, si allinea la configurazione del proprio PC con il comando
  - rustup update
- Sono disponibili diverse toolchain alternative
  - Nel caso di Windows, è possibile scegliere tra quella basata sul compilatore Microsoft (msvc) e quella basata su GCC/GDB
- L'installazione di una toolchain mette a disposizione vari strumenti, tra cui
  - **rustc** - il compilatore del linguaggio
  - **cargo** - lo strumento di gestione dei progetti e delle loro dipendenze

Cargo

- Programma ufficiale per la gestione di un package
- Offre supporto per:
  - Indicazione dei metadati di progetto (nome, versione, autore, tipo di progetto, versione del linguaggio, …)
  - Gestione delle dipendenze (uso di altre librerie) e integrazione con il sito crates.io (registro ufficiale dei package Rust)
  - Gestione dei test di unità e di integrazione
  - Esecuzione di benchmark
- Si crea un nuovo progetto con i comandi:
  - `cargo new project_name`
  - `cargo new --lib library_name`
- Si compila / esegue un progetto con i comandi:
  - `cargo build`
  - `cargo run`

Uso di librerie

- L'ecosistema di Rust è fortemente integrato
  - Chi crea una libreria può pubblicarla sul sito https://crates.io, rendendola disponibile a tutta la comunità degli sviluppatori
- Installando l'estensione cargo-edit (https://github.com/killercup/cargo-edit), si può importare una libreria pubblica con il comando
  - `cargo add <libname>`
  - Esso aggiunge, nel file Cargo.toml, una sezione chiamata [dependencies] nella quale sono indicate, una per riga, le librerie importate e la relativa versione
  - Questo fa sì che, all'atto della compilazione del package, le librerie da cui si dipende vengano automaticamente collegate al programma generato, rendendo così disponibili i relativi contenuti (tipi e funzioni)

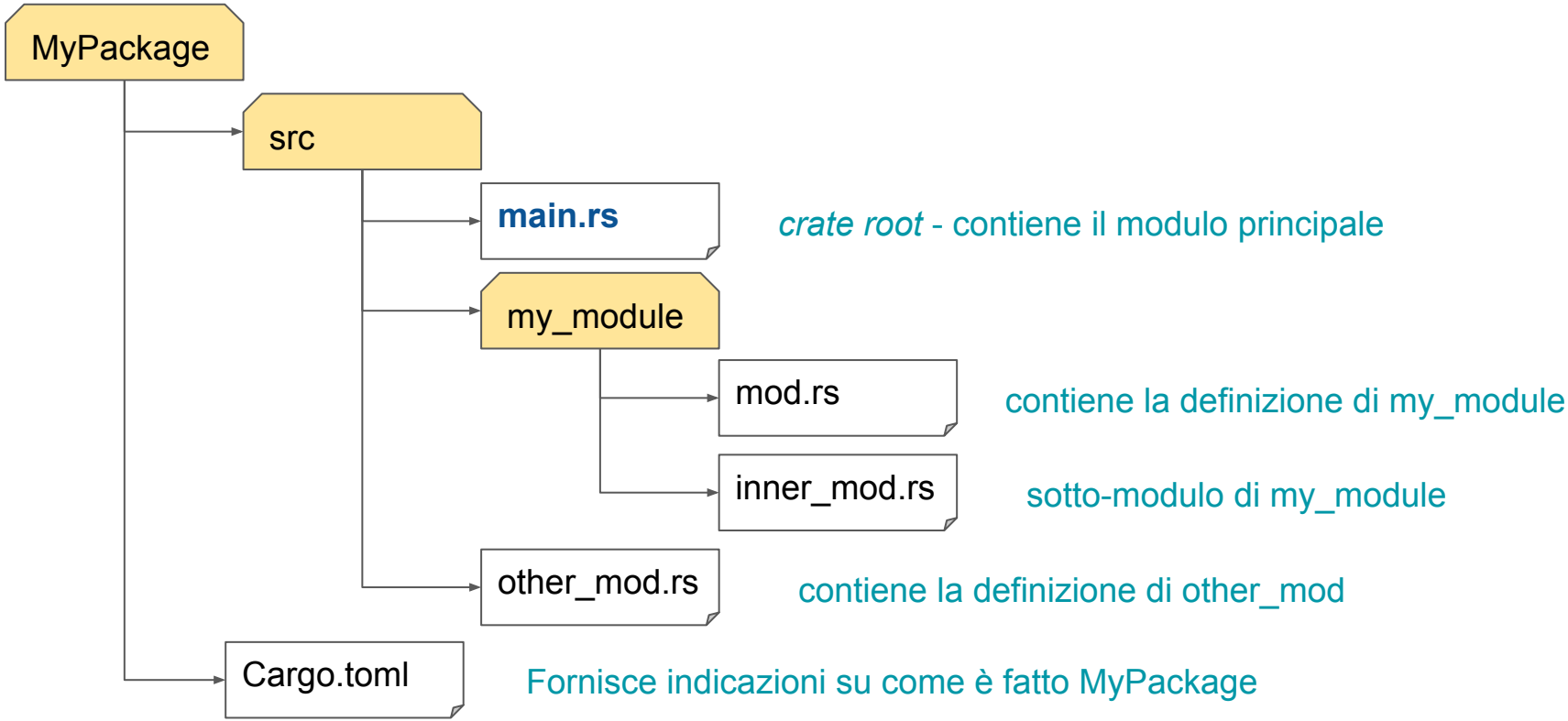
Ambienti integrati di sviluppo (IDE)

- Molti ambienti di sviluppo offrono supporto per lo sviluppo in Rust
- Tra gli altri
  - VS Code - https://marketplace.visualstudio.com/items?itemName=rust-lang.rust
  - IntelliJ Idea / **CLion** - https://plugins.jetbrains.com/plugin/8182-rust
  - Eclipse - https://www.eclipse.org/downloads/packages/release/2019-09/r/eclipse-ide-rust-developers-includes-incubating-components
  - Sublime Text - https://github.com/rust-lang/rust-enhanced
  - Atom - https://github.com/rust-lang/atom-ide-rust
  - NeoVim - https://dhghomon.github.io/easy_rust/
- IntelliJ CLion (così come Idea) è disponibile gratuitamente per gli studenti che si registrano con l'indirizzo istituzionale
  - https://www.jetbrains.com/community/education/#students

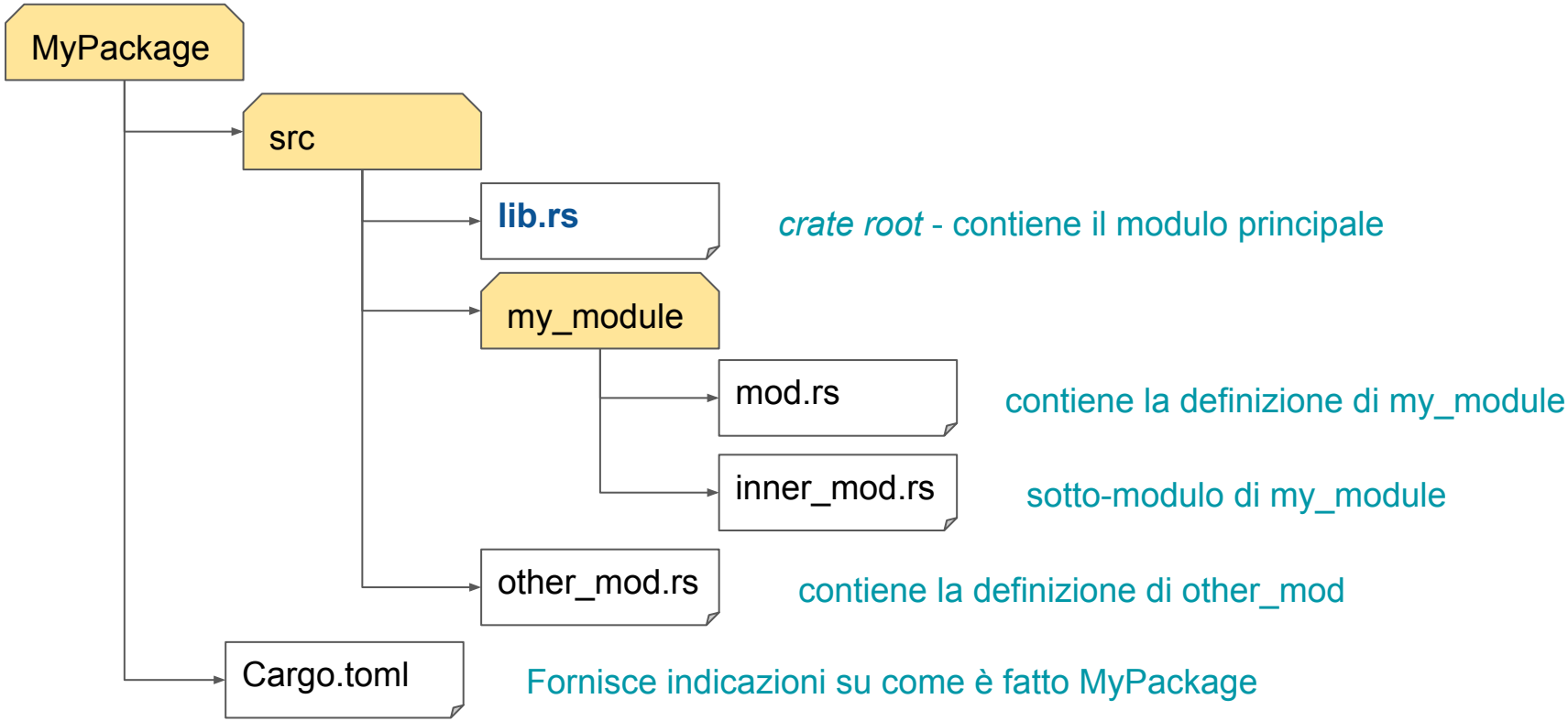
Terminologia

- **Crate**
  - Unità di compilazione che può dare origine ad un programma eseguibile (binario) o ad una libreria
  - Può contenere riferimenti a moduli contenuti in ulteriori file sorgenti: questi sono inclusi nel file corrente prima di avviarne la compilazione
- **Crate root**
  - File sorgente da cui parte il compilatore Rust per creare il modulo principale del crate
  - Un crate binario deriva da src/main.rs, una libreria deriva da src/lib.rs
- **Module**
  - Meccanismo usato per suddividere gerarchicamente il codice sorgente in unità logiche differenti e regolarne la visibilità reciproca
  - Un modulo può contenere funzioni, tipi ed altri moduli
  - Un sotto-modulo può essere contenuto nel file sorgente del suo genitore, essere scritto in un file sorgente a parte o essere memorizzato in una sottocartella della cartella in cui è ospitato il suo contenitore
  - I moduli non sono compilati individualmente, ma solo come parte di un crate che li contiene
- **Package**
  - Insieme di uno o più crates volti a fornire un insieme di funzionalità (progetto)
  - Un package è ospitato in una cartella che contiene il file Cargo.toml: esso descrive come costruire i crate di cui è composto il package
  - Un package può contenere al massimo una libreria
  - Se contiene più crate binari, questi vengono posti nella cartella src/bin

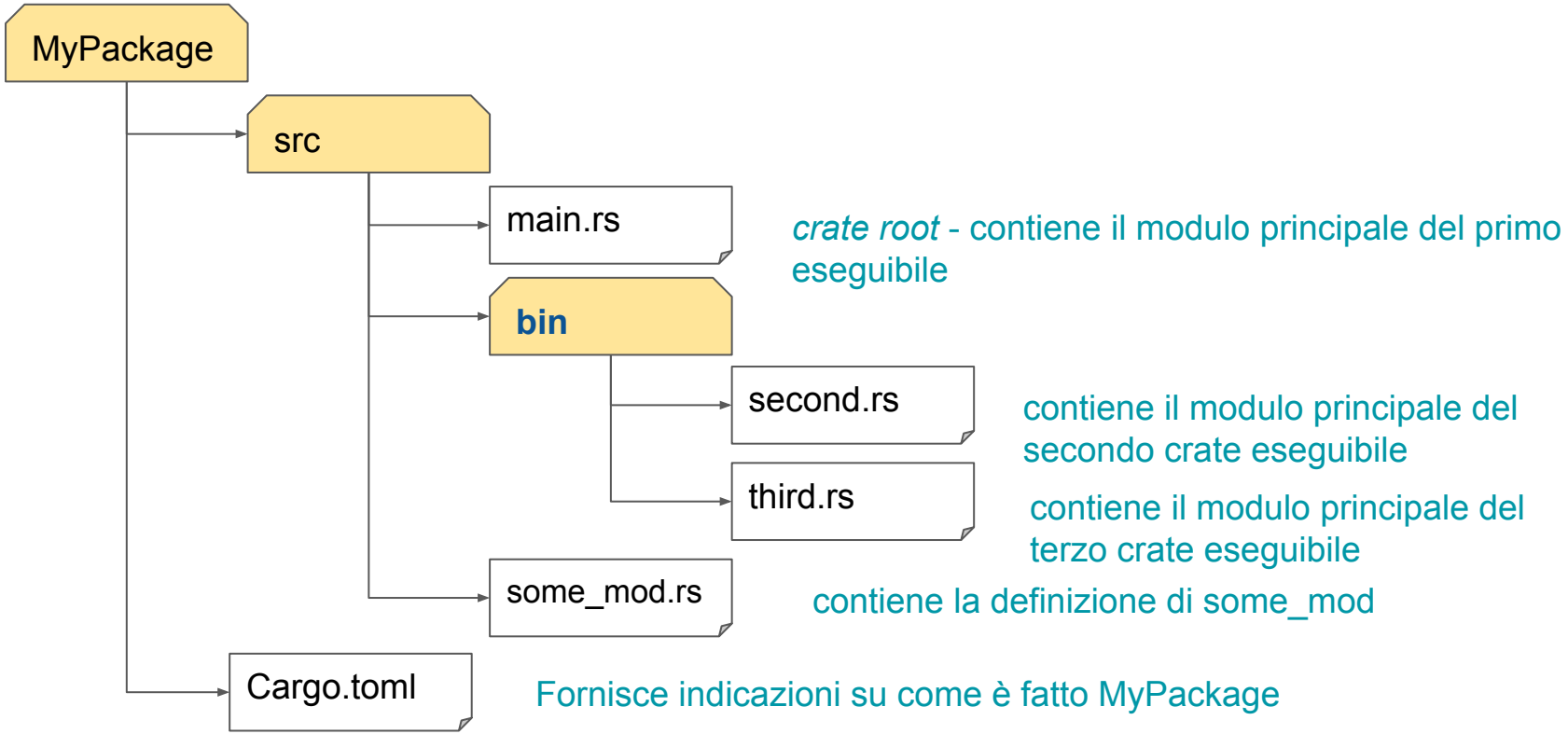
Struttura di un progetto eseguibile Rust



Struttura di un progetto di libreria Rust



Struttura di un progetto con due o più eseguibili



Sopravvivere in Rust

- Il punto di ingresso di un programma è `fn main() { ... }`
  - Normalmente si trova nel file main.rs
- Per stampare su *stdout*: `print!(...)` oppure `println!(...)`
  - Il primo parametro è una stringa di formato: per ciascuna coppia di **{}** presenti al suo interno deve essere indicato un parametro successivo, il cui contenuto sarà inserito al posto delle graffe
  - Tutto quello che non sta nelle graffe viene stampato così com'è
    `println!("Hello, {}!" , "world"); // → Hello, world!`
  - Per stampare su *stderr*: `eprint!(...)` oppure `eprintln!(...)`
- Si dichiarano le variabili con la parola chiave let
  - Nella maggior parte dei casi, il compilatore è in grado di dedurre automaticamente il tipo associato
  - `let x : i32 = 13;`
  - `println!("{}", x);`

Risorse utili

- Tour of Rust - https://tourofrust.com/
  - Tutorial interattivo disponibile molte lingue (italiano compreso) in grado di fornire i concetti essenziali della programmazione in Rust
- Rust in Easy English - https://dhghomon.github.io/easy_rust/
  - Altro tutorial interattivo, con stile più discorsivo e spiegazioni passo passo di tutti i principali concetti del linguaggio
- Rust Language Cheat Sheet - https://cheats.rs/#the-abstract-machine
  - Riassunto sintetico del linguaggio, corredato di esempi interattivi e riferimenti ad una grande quantità di risorse per gli approfondimenti
- Rust By Example - https://doc.rust-lang.org/stable/rust-by-example/
  - Raccolta ufficiale di esempi funzionanti che illustrano i diversi concetti del linguaggio e delle librerie standard
- Rust Cookbook - https://rust-lang-nursery.github.io/rust-cookbook/
  - Raccolta di esempi che presentano le buone pratiche del linguaggio applicato ad una ventina di contesti applicativi - richiede una conoscenza ragionevole del linguaggio

Risorse utili

- The Book - https://doc.rust-lang.org/book/
  - Il tutorial ufficiale del linguaggio
- The Rust Reference - https://doc.rust-lang.org/stable/reference/
  - La guida di riferimento ufficiale del linguaggio: utile per approfondire specifici concetti dopo che si è imparata la struttura generale
- Rust-learning - https://github.com/ctjhoa/rust-learning
  - Una ampia collezione curata ed aggiornata di link ad articoli e video legati all'apprendimento di Rust, suddivisi per argomento
- The Little Book of Rust Books - https://lborb.github.io/book/
  - Collezione curata ed aggiornata di libri online su Rust e sul suo ecosistema
- Rust Books - https://github.com/sger/RustBooks
  - Raccolta di riferimenti a libri cartacei (e digitali) con link ai relativi editori e/o venditori

Risorse Utili (progetti e lavori in Rust)

- Amazon Firecracker Micro-VM
  - https://github.com/firecracker-microvm/firecracker
- Redox Operating System
  - https://gitlab.redox-os.org/redox-os/redox/
- Tock (Embedded Operating System)
  - https://www.tockos.org
- Writing a OS in Rust (blog)
  - https://os.phil-opp.com/
- Bevy (simple data-driven game engine)
  - https://github.com/bevyengine/bevy
- SWC (Speedy Web Compiler)
  - https://github.com/swc-project/swc
- Jobs
  - https://rustjobs.dev/