RUST

*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*
- *Linguaggio compilato e non basato su bbytecode*

*Rust è un linguaggio **staticamente e fortemente tipato/ tipizzato** (quando introduco una variabile questa è legata ad un tipo), adatto alla programmazione di sistema*

- *Tutti i **tipi sono noti in fase di compilazione** (se non specificato, dedotto dal valore)*
- *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*

***Note** per programmazione di sistema*

- *Privo di comportamenti non definiti*
- *Con supporto corretto alla programmazione concorrente*
- *Linguaggio pratico, spiega l'errore preso*
- *Supporto alle dipendenze*
- *Offrire astrazione a costo nullo → se ti basta c'è versione gratis, altrimenti puoi pagare per avere cose extra*

Problemi **risolti** in RUST:

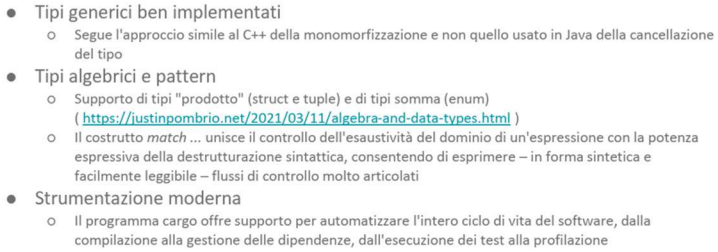
- Dangling pointer → puntatori ad areee di memoria già rilasciate
- Doppi rilasci
- Corse critiche → accesso a dati il cui contenuto può dipendere da schedulazione o altri eventi esterni al programma stesso
- Buffer overflow → tentativo di accedere ad aree contigue a quelle possedute ma non di pertinenza
- Iteratori invalidi → accesso iterativo ad una collezione che viene modificata nel mentre
- Overflow aritmetici in fase di debug

I puntatori si portano dietro l'etichetta del tempo di vita che viene validato dal compilatore quindi non posso accedere a zone di memoria non valida.

- **Fat pointer**: puntatori che si portano dietro inizio e dimensione

Il **compilatore ottimizza in modo aggressivo** la dimensione di codice che viene compilato e la velocità. → togliendo tutto e massimizzando performance tenendo conto che l'architettura dei calcolatori è **basato su cache**

- Predilige uso di array → principio di località dei riferimenti
- Evitare doppi salti in quanto causano cache miss
- Maggior parte di ottimizzazioni sono basate su indirizzi statici
- Gestione delle dipendenze, sistema di test integrato, gestione dei moduli → garantisce migliori performance



POSSESSO DI UN VALORE → SICURO

Diritto di accedere ad un dato e dovere di rilasciarlo alla fine della sua vita → **un dato è posseduto da una e una sola variabile** e se quella variabile cessa di esistere, il valore muore subito.

Rust introduce il concetto di **MOVIMENTO** → trasferire il possesso di un valore ad un'altra variabile

- il valore può muoversi nel caso in cui non ci sia nessuno che lo conosca.

È anche possibile concedere temporaneamente l'accesso in sola lettura ad un valore tramite l'uso di riferimenti

Dato send: dato può essere ceduto ad un altro thread
Dato sink:

In Rust non ci sono le eccezioni → le funzioni che possono fallire ritornano un tipo algebrico

- Ok
- Error

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)

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);`

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