



UNIVERSITÀ
DEGLI STUDI
FIRENZE

Scuola di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Informatica

Tesi di Laurea

ANALISI DEL LINGUAGGIO RUST E DELLA
SUA CRESCENTE ADOZIONE NEI SISTEMI
OPERATIVI

ANALYSIS OF RUST AND ITS INCREASING
ADOPTION IN OPERATING SYSTEMS

FRANCESCO BIRIBÒ

Relatore: *Rosario Pugliese*

Anno Accademico 2024-2025

INDICE

Elenco delle figure	3
1 Introduzione	7
1.1 Contestualizzazione	7
1.2 Argomento	8
1.3 Obiettivi	8
1.4 Struttura	9
1.5 Annuncio dei risultati	9
2 Rust	11
2.1 Contesto e motivazioni	11
2.2 Origine e primo sviluppo	13
2.3 Diffusione e adozione iniziali	13
2.4 Espansione e interesse crescenti	14
2.5 Supporto istituzionale e professionale	15
3 Gestione della memoria in Rust	17
3.1 Ownership	17
3.2 Borrowing	21
3.3 Lifetime	24
4 Sistemi Operativi	29
4.1 C: motivazioni e caratteristiche	29
4.2 Rust contro C	32
4.2.1 Gestione delle risorse	34
4.2.2 Sicurezza della memoria	43
4.2.3 Prestazioni	56
4.2.4 Complessità del codice	60
5 Progetti e applicazioni reali	73
5.1 Kernel di Windows 11	73
5.2 Rust for Linux	74
5.2.1 Moduli in mainline	76
5.2.2 Moduli outside mainline	78
5.2.3 Impatto del progetto	80
5.3 Red Hat: Nova	82
5.4 Ubuntu: sudo-rs	83
5.5 Redox OS	84
5.6 Considerazioni sui progetti analizzati	86
6 Conclusione	89

2 INDICE

6.1	Riassunto	89
6.2	Considerazioni critiche e personali	90
	Bibliografia	91

ELENCO DELLE FIGURE

Figura 1	<i>Double free</i> in C	46
Figura 2	Tentativo di <i>double free</i> in Rust	46
Figura 3	<i>Access after free</i> in C	49
Figura 4	Tentativo di <i>access after free</i> in Rust	49
Figura 5	<i>Buffer overflow</i> in C	51
Figura 6	<i>Buffer overflow</i> in Rust	51
Figura 7	<i>Buffer overread</i> in C	53
Figura 8	<i>Buffer overread</i> in Rust	53
Figura 9	<i>Uninitialized memory access</i> in C	55
Figura 10	<i>Uninitialized memory access</i> in C con previa <i>free</i> . .	56
Figura 11	<i>Access after free</i> dovuto all'assenza di <i>lifetime</i> in C .	62
Figura 12	Tentativo di <i>access after free</i> in Rust	62
Figura 13	Traduzione di una <i>macro</i> in C	68
Figura 14	Limitazioni delle <i>macro</i> C	69
Figura 15	Compilazione di una <i>macro</i> in Rust	71
Figura 16	Errore durante la compilazione di una <i>macro</i> in Rust	71

"Simplicity is prerequisite for reliability"
— *Edsger Dijkstra*

INTRODUZIONE

1.1 CONTESTUALIZZAZIONE

Negli ultimi anni il linguaggio di programmazione *Rust* ha suscitato un crescente interesse, attirando l'attenzione di sviluppatori e aziende. Una delle motivazioni principali risiede nel suo approccio alla gestione della memoria dinamica, da sempre un tema cruciale nello sviluppo di sistemi di basso livello.

Storicamente, si sono affermati due approcci alla gestione della memoria: quello *manuale*, adottato dai primi linguaggi di programmazione come l'assembly e, successivamente, dal C e quello *automatico*, introdotto nel 1959 da Lisp, il primo linguaggio a integrare la *Garbage Collection* (GC).

L'approccio manuale rende il programmatore interamente responsabile della gestione della memoria, esponendo a una vasta gamma di errori dovuti al fattore umano; la GC, invece, solleva il programmatore da tale incarico, ma introduce un overhead non trascurabile e riduce la trasparenza e il controllo sulla memoria.

Nel contesto delle applicazioni di basso livello, soprattutto dei sistemi operativi, controllo diretto e prestazioni sono requisiti fondamentali; per questo, l'approccio manuale rappresenta nella pratica l'unica soluzione percorribile. È in questo contesto che si inserisce Rust, un linguaggio sviluppato per garantire una gestione sicura della memoria dinamica, senza impiegare GC, e, quindi, senza compromettere le prestazioni.

In questa tesi verrà analizzato Rust, con l'obiettivo di comprendere le motivazioni alla base della sua crescente diffusione e di valutare gli strumenti effettivamente offerti.

1.2 ARGOMENTO

La presente ricerca prende in esame la crescente popolarità di Rust nell'ambito della programmazione di basso livello, in particolare nello sviluppo di sistemi operativi.

Il linguaggio ha una premessa chiara: prevenire interi classi di problemi legati alla gestione della memoria a livello di compilazione, fornendo allo stesso tempo prestazioni, in termini di velocità d'esecuzione, paragonabili a C e C++.

Vi sono, per tale motivo, varie iniziative che mirano a inserire il linguaggio come valida opzione nello sviluppo di sistemi operativi, ma la sua integrazione solleva questioni fondamentali, sia da un punto di vista di tradizione (in quanto storicamente C è un pilastro nello sviluppo di sistemi operativi), che di spesa, in termini di tempo, per imparare un nuovo linguaggio e per mantenere contemporaneamente due flussi di sviluppo paralleli (C e Rust).

1.3 OBIETTIVI

L'obiettivo principale di questa tesi è offrire una panoramica sul linguaggio di programmazione Rust, esponendo quelle che sono le motivazioni dietro il suo sviluppo e la sua popolarità; successivamente, verrà analizzato l'approccio alternativo di gestione della memoria implementato dal linguaggio.

Un altro obiettivo è quello di stabilire se, nella teoria, Rust rappresenti una valida opzione per la programmazione di basso livello. Verranno innanzitutto stabiliti i requisiti fondamentali di un linguaggio per lo sviluppo di sistemi operativi; successivamente Rust verrà confrontato con C, standard *de facto* per la programmazione di basso livello, sulla base di gestione delle risorse, della memoria e degli errori, complessità della sintassi e prestazioni, fornendo a tale scopo anche esempi di codice.

Come aspetto finale, verranno proposti progetti concreti che mostrano come la popolarità del linguaggio non sia solo teorica, ma rappresenti uno strumento pratico e in grado di portare risultati concreti.

L'analisi combina aspetti teorici (descrizione dei meccanismi ed esempi di codice) con lo studio di casi concreti, per evidenziare sia i principi alla base del linguaggio sia le sue applicazioni pratiche.

1.4 STRUTTURA

Questa tesi è strutturata in quattro parti principali, ognuna con il rispettivo capitolo, dal due al cinque.

Nel capitolo due, *'Rust'* 2, verrà fornita una panoramica sul linguaggio di programmazione Rust, con particolare attenzione sulle motivazioni che ne hanno determinato lo sviluppo e alla base della sua crescente popolarità negli ultimi anni.

Nel capitolo tre, *'Gestione della memoria in Rust'* 3, verrà esaminato il modello di gestione della memoria implementato da Rust, con un'analisi approfondita dei meccanismi alla base; verranno inoltre forniti listati di codice che mostrano esempi del loro funzionamento.

Nel capitolo quattro, *'Sistemi Operativi'* 4, verranno esposti gli strumenti necessari che un linguaggio di programmazione dovrebbe avere per essere impiegato nella programmazione di sistema, con riferimento al linguaggio C. Successivamente, quest'ultimo verrà confrontato con Rust sotto gli aspetti di gestione della memoria, delle risorse e degli errori, complessità del codice e prestazioni, per mostrare come Rust metta a disposizione strumenti fondamentali per essere considerato una valida alternativa, teorica, nella programmazione di sistema.

Nel capitolo cinque, *'Progetti e applicazioni reali'* 5, verranno presentati progetti e applicazioni concreti che mostrano, nella pratica, come Rust non sia solo un linguaggio promettente dal punto di vista teorico, ma rappresenti un valido strumento, capace di ottenere risultati concreti, nella programmazione di basso livello.

1.5 ANNUNCIO DEI RISULTATI

Nella prima parte della tesi, viene evidenziato come l'approccio di gestione della memoria di Rust permetta di prevenire errori critici tipici dei linguaggi con approccio manuale, senza compromettere le prestazioni.

Il confronto con C evidenzia vantaggi in termini di sicurezza e mantenibilità, a fronte di una sintassi più complessa e una curva di apprendimento più ripida e lenta.

Nella seconda parte, verranno presi in considerazione progetti concreti (*Rust for Linux*, *Redox OS*, il kernel di Windows 11 e *sudo-rs*), evidenziando che Rust è già impiegato in contesti di sistemi operativi e che la sua

adozione procede, seppur gradualmente, in maniera significativa.

RUST

In questo capitolo verrà offerta una panoramica sul linguaggio di programmazione Rust, con particolare attenzione riguardo le motivazioni che ne hanno determinato lo sviluppo e alla base della sua crescente popolarità.

Verranno inoltre introdotti alcuni concetti fondamentali riguardanti il modello di gestione della memoria di Rust, che costituiranno il fulcro del capitolo tre: *‘Gestione della memoria in Rust’* 3.

In chiusura, saranno presentati alcuni progetti attuali che evidenziano come Rust rappresenti una valida alternativa nel contesto della programmazione di sistemi.

Le informazioni contenute in questo capitolo sono tratte principalmente dal seguente articolo[\[15\]](#).

Rust è un linguaggio di programmazione che, negli ultimi anni, ha ottenuto un’ottima reputazione tanto tra gli sviluppatori quanto tra le aziende. Il suo principale punto di forza risiede nella capacità, considerata a lungo un ideale solo teorico, di produrre codice che sia simultaneamente ad alte prestazioni e sicuro, in particolare sotto l’aspetto di gestione della memoria.

Che cosa rende Rust *speciale*, rispetto ad altri linguaggi di programmazione? Principalmente, il suo approccio innovativo e alternativo alla gestione della memoria, che evita sia la necessità di gestione manuale da parte dello sviluppatore che la dipendenza da meccanismi di automatici, come *garbage collection*.

2.1 CONTESTO E MOTIVAZIONI

Rust è spesso definito come un linguaggio di basso livello che offre astrazioni di alto livello a costo quasi nullo. La caratteristica principale

che lo distingue dalla maggior parte degli altri linguaggi è il meccanismo di gestione della memoria dinamica.

Tradizionalmente esistono due approcci principali alla gestione della memoria:

- **Gestione manuale:** tipica di linguaggi considerati di livello più basso, come C e C++, progettati per dare al programmatore il massimo controllo sulle risorse, compresa la memoria. Secondo questo approccio è il programmatore a decidere quando allocare e quando deallocare la memoria dinamica. Il codice risultante generalmente è molto performante in termini di velocità d'esecuzione, ma questo avviene a un costo: non si hanno garanzie sulla sicurezza e sull'integrità della memoria. L'accesso e la gestione diretti della memoria possono portare problemi come *unfreed memory*, *access after free*, *double free*, *dangling pointers* e non solo.
- **Gestione automatica:** tipica di linguaggi di livello più alto come Java, Python e C#, progettati per facilitare la gestione della memoria dinamica. È infatti gestita da un componente runtime, noto come *Garbage Collector*: la memoria viene allocata implicitamente quando vengono creati oggetti, e liberata quando questi non sono più validi, in maniera trasparente agli occhi del programmatore. Il GC tiene traccia degli oggetti allocati e libera la loro memoria quando questi non sono più referenziati. Il codice prodotto generalmente è libero dalla maggior parte degli errori legati alla memoria, ma questo avviene a un costo: il codice relativo al GC viene incluso insieme al codice dell'applicazione, generando un file eseguibile di dimensioni maggiori; inoltre, il GC richiede risorse computazionali aggiuntive per compiere la sua funzione, eventualmente rallentando, se non bloccando momentaneamente, l'esecuzione dell'applicazione.

Non esiste un approccio 'oggettivamente corretto': durante lo sviluppo di un'applicazione la scelta sull'approccio utilizzato spesso ricade sulle specifiche esigenze. In applicazioni di basso livello, quali lo sviluppo di sistemi operativi, dove le prestazioni sono fondamentali, la scelta si riduce, nella pratica, solamente all'approccio manuale, rendendo linguaggi come C e i suoi derivati lo standard.

Il creatore di Rust si pose come obiettivo sviluppare un linguaggio che eliminasse il compromesso tra prestazioni e sicurezza della memoria, fornendo un'alternativa agli approcci tradizionali.

2.2 ORIGINE E PRIMO SVILUPPO

Rust nacque inizialmente come progetto personale di Graydon Hoare, sviluppatore presso l'azienda Mozilla, nel 2006. Il progetto nacque in seguito alla frustrazione di Hoare per i frequenti malfunzionamenti di software critici, spesso dovuti a problematiche di gestione della memoria¹.

L'obiettivo del progetto era molto ambizioso: sviluppare un linguaggio che consentisse di scrivere codice che fosse contemporaneamente *veloce e sicuro*, principalmente sotto l'aspetto di gestione della memoria.

Il nome '*Rust*' fu ispirato da un particolare tipo di fungo (chiamato appunto *Rust*, ruggine in inglese), definito dal creatore stesso come '*sovra-ingegnerizzati per la sopravvivenza*', metafora per la resilienza e l'attenzione alla sicurezza che il linguaggio mirava a ottenere.

Le prime versioni di Rust adottavano un sistema *Garbage Collection* integrato nel linguaggio, dotato di una sintassi esplicita e dedicata. Tuttavia, dopo i primi esperimenti, il team di sviluppo decise di rinunciare completamente al GC, ritenendolo incompatibile con gli obiettivi di performance e controllo sulla memoria a cui mirava il linguaggio.

Fu così introdotto un nuovo modello, noto come '*modello di ownership*', il quale si fonda sui concetti di possesso (*ownership*) e prestito (*borrowing*). Questo modello definisce regole rigorose sul trasferimento e l'utilizzo dei valori, cercando di prevenire gli errori legati alla memoria già durante la fase di compilazione, senza necessità di un *runtime* dedicato (come il GC).

Questo modello divenne la base del sistema di memoria sicura senza GC che ancora oggi contraddistingue Rust.

2.3 DIFFUSIONE E ADOZIONE INIZIALI

Dopo il rilascio della prima versione stabile, nel 2015, Rust iniziò a essere adottato in contesti reali, seppure con una diffusione iniziale veramente limitata. Alcune aziende cominciarono a sperimentare con il linguaggio, per riscrivere componenti critici, mirando a ottenere un miglioramento in termini di sicurezza e prestazioni:

- **Mozilla**, promotrice del linguaggio, adottò Rust nella realizzazione del motore di rendering CSS *Servo*, dal quale derivano alcuni

¹ Secondo quanto riportato nell'articolo[15], la frustrazione di Hoare era dovuta ai continui crash del software (presumibilmente scritto in C) dell'ascensore del palazzo in cui risiedeva, al ventunesimo piano.

componenti integrati nel motore di rendering di *Firefox*, *Gecko*;

- **Facebook** riscrisse in Rust parte degli strumenti interni dedicati alla gestione della propria codebase.

Questi primi esempi segnarono una fase di *‘adozione esplorativa’*, in cui Rust cominciava a essere percepito come una possibile alternativa ai linguaggi esistenti.

2.4 ESPANSIONE E INTERESSE CRESCENTI

Nel biennio 2020–2021 si verificò una prima vera e propria espansione dovuta a una maggiore stabilità del linguaggio. Le prime adozioni, da parte di Mozilla e Facebook, giocarono un ruolo cruciale: mostrarono le effettive capacità di Rust.

Come conseguenza, sempre più aziende cominciarono a introdurre codice Rust nella propria codebase, principalmente per i componenti critici sotto gli aspetti di prestazioni e sicurezza:

- **Dropbox** migrò il motore di sincronizzazione da Python a Rust, ottenendo un significativo miglioramento delle prestazioni;
- **Fusion Engineering**² utilizzò Rust per riscrivere i componenti relativi al sistema di controllo dei droni, precedentemente scritti in C++;
- **Discord** riscrisse la pipeline di encoding video e porzioni di backend relative alla gestione dell’autenticazione, dei messaggi e delle notifiche³;
- **Amazon** iniziò a introdurre Rust nei servizi della piattaforma AWS, rafforzando il legame tra il linguaggio e l’ambiente cloud.

Tuttavia, in questo periodo, la maggior parte degli sviluppatori Rust contribuivano al progetto senza un supporto economico diretto. I membri principali del team erano in gran parte impiegati in altre aziende (Mozilla, Amazon, Huawei e Microsoft) e si dedicavano a Rust come attività collaterale.

² Si tratta di un’azienda che sviluppa sistemi software di controllo e gestione per droni. Fondata da Mara Bos, membro del *Rust Project* e co-leader del *Rust Library Team*.

³ Secondo dichiarazioni di Discord stessa, questa modifica ha generato un’incremento delle prestazioni fino a 10 volte.

2.5 SUPPORTO ISTITUZIONALE E PROFESSIONALE

A partire dal 2021, Rust entrò in una nuova fase, contraddistinta da un crescente supporto industriale. Diverse aziende del settore tecnologico, in particolare alcune *Big Tech*, iniziarono a finanziare in modo diretto lo sviluppo del linguaggio, offrendo compensi economici ai membri della community che divennero, a tutti gli effetti, sviluppatori Rust a tempo pieno.

L'obiettivo di questi finanziamenti era quello di permettere agli sviluppatori di dedicarsi in modo continuativo al progetto, garantendo codice stabile, aggiornato e mantenibile.

Questo periodo segnò un passaggio cruciale: da un progetto guidato dalla passione della community a un ecosistema supportato anche da risorse industriali. Di conseguenza, Rust cominciò a consolidarsi come una scelta sempre più pragmatica per applicazioni ad alte prestazioni e con forti requisiti in termini di sicurezza della memoria.

Negli ultimi anni, Rust ha trovato crescente impiego anche a livello di sistema operativo, in particolare in ambito kernel. Tra gli esempi più rilevanti si trovano l'integrazione di Rust nel kernel di Windows 11 da parte di Microsoft, il progetto *Rust for Linux*, lo sviluppo di driver come *Nova* da parte di Red Hat o il tentativo di rendere *sudo* più sicuro da parte di Ubuntu, fino ad arrivare allo sviluppo di interi sistemi operativi in Rust, come *Redox OS*⁴.

⁴ Questi progetti verranno esplorati approfonditamente nel capitolo cinque: *'Progetti e applicazioni reali'* [5](#).

GESTIONE DELLA MEMORIA IN RUST

In questo capitolo viene esaminato l'approccio implementato da Rust per gestire la memoria dinamica. In particolare, verranno approfonditi concetti chiave come *Ownership*, *Borrowing* e *Lifetime*, illustrandone il funzionamento. L'obiettivo è evidenziare come Rust possa garantire la sicurezza della memoria senza sacrificare le prestazioni.

Le informazioni presentate in questo capitolo sono tratte principalmente dalla documentazione ufficiale di Rust[7].

L'approccio implementato da Rust per gestire la memoria dinamica è noto come 'modello di ownership' (*Ownership Model*). Il modello si fonda sui concetti di *ownership*, *borrowing* e *lifetime*, i quali stabiliscono regole sulla gestione della memoria che devono essere rispettate affinché il codice sia compilabile. Il rispetto di tali regole costituisce un prerequisito per la compilazione. Il compilatore controlla che ogni vincolo sia soddisfatto e procede alla compilazione solo in caso positivo. Il modello garantisce un uso corretto e sicuro della memoria già a tempo di compilazione, evitando l'introduzione di overhead durante l'esecuzione¹.

3.1 OWNERSHIP

Il primo concetto alla base del modello di ownership di Rust è l'*ownership*² stessa, la quale definisce vincoli sui legami tra variabili e valori. Rust introduce il concetto di *proprietario*: ogni valore ha un proprietario, ovvero la variabile che lo possiede in un determinato istante.

¹ È opportuno precisare che un minimo overhead viene comunque introdotto, sotto forma di invocazione automatica di una funzione dedicata alla deallocazione.

² Sebbene innovativo, questo concetto non è unico di Rust, il quale ha infatti preso spunto da un pattern ampiamente utilizzato in C++: *RAII* (Resource Allocation Is Initialization), secondo il quale gli oggetti acquisiscono le risorse di cui hanno bisogno (ovvero vengono inizializzati) al momento della loro allocazione.

Si consideri, come esempio, la seguente assegnazione: $a = 5$. Nella maggior parte dei linguaggi di programmazione, l'istruzione verrebbe interpretata come *'a vale 5'*. In Rust il significato è diverso: *'a possiede il valore 5'* o *'a è il proprietario di 5'*.

Stabilire i proprietari è fondamentale per implementare un meccanismo di deallocazione della memoria deterministico e prevedibile: chi libera la memoria relativa a un determinato valore? Il suo proprietario.

Le regole di *ownership*, quindi, stabiliscono i vincoli sui legami tra valori e proprietari. Esse sono tre:

- Ogni valore ha un proprietario (owner);
- Può esserci un solo proprietario per volta (per ogni valore);
- Quando il proprietario di un valore esce dallo scope³, il relativo valore viene scartato.

In Rust la memoria viene liberata automaticamente quando il proprietario di un valore esce dallo scope. Per implementare questo, Rust invoca una funzione speciale, *drop*, in corrispondenza della fine di ogni scope. La funzione viene invocata su ogni variabile che esce dallo scope ed è il proprietario di un valore.

Grazie alle regole di *ownership*, Rust può garantire che la deallocazione della memoria avvenga sempre in modo corretto: poiché ogni valore ha un solo proprietario, non vi è ambiguità riguardo a chi sia responsabile della sua deallocazione.

- L'esistenza di un proprietario garantisce che non vi siano valori allocati ma non più referenziati;
- L'unicità del proprietario impedisce che un'area di memoria venga deallocata più volte;

Le regole di *ownership* influenzano diversi aspetti del linguaggio, in particolare la condivisione dei valori tra variabili, l'assegnazione, il passaggio di parametri alle funzioni e operazioni analoghe.

Per comprendere a fondo il comportamento del modello di *ownership*, è utile esaminare come Rust gestisce i diversi tipi di dato. I tipi di dato si distinguono tra *semplici* e *complessi*:

³ Lo scope rappresenta un range nel programma all'interno del quale un oggetto è valido. Solitamente è delimitato da una coppia di parentesi graffe.

- I tipi semplici hanno dimensione fissa, nota a tempo di compilazione, e vengono quindi allocati nello stack;
- I tipi complessi, invece, hanno dimensione variabile o non determinabile a tempo di compilazione e vengono quindi allocati nell'heap.

In relazione alla gestione della memoria, Rust definisce due comportamenti distinti per l'assegnazione: *Copy* e *Move*.

- *Copy*: l'intera memoria del valore viene duplicata tramite una copia bitwise; questo comportamento è adottato di default per i tipi semplici;
- *Move*: il valore viene trasferito da una variabile all'altra, invalidando quella di origine; questo comportamento è adottato di default per i tipi complessi.

I seguenti esempi illustrano il comportamento dell'assegnamento nei tipi semplici, che implementano *Copy*, e in quelli complessi, che implementano *Move*:

Listing 1: Comportamento di Copy

```
fn main() {  
    let x: i32 = 5;  
    let y: i32 = x;  
    println!("x: {}, y: {}", x, y);  
}
```

Listing 2: Comportamento di Move

```
fn main() {  
    let x: String = String::from("Sono in Heap!");  
    let y: String = x;  
    println!("x: {}, y: {}", x, y); // <-- Errore di compilazione  
}
```

Nel listato 1 si osserva che `x` resta valido dopo l'assegnamento, poiché `i32` implementa *Copy*, trattandosi di un tipo semplice, allocato nello stack.

Nel listato 2, invece, l'assegnamento provoca il trasferimento dell'ownership, che viene trasferita a `y`, invalidando `x`. `String` è infatti un tipo complesso, allocato nell'heap, che non implementa *Copy* ma utilizza il comportamento di *Move* per impostazione predefinita.

Come visto nel listato 2, l'assegnamento di un tipo complesso trasferisce l'*ownership*, rendendo invalido il riferimento originale. Questo comportamento, tuttavia, può risultare limitante in alcuni scenari.

Per ovviare a tale limitazione, Rust fornisce un meccanismo per effettuare copie profonde anche con tipi che utilizzano *Move*: il tratto *Clone*⁴.

Il tratto *Clone* permette di definire esplicitamente come realizzare la copia profonda di un valore. L'invocazione del metodo `clone()` genera un nuovo valore, indipendente dal primo, ma con lo stesso contenuto.

Si consideri il seguente esempio, che mostra il comportamento di *Clone*:

Listing 3: Comportamento di *Clone*

```
fn main() {
    let x: String = String::from("Sono in Heap!");
    let y: String = x.clone();
    println!("x: {}, y: {}", x, y); // Entrambi i riferimenti sono
                                   validi
}
```

Nel listato 3 si osserva che sia *x* che *y* sono validi in seguito all'assegnamento, in quanto la chiamata a `clone()` ha generato una nuova stringa, con lo stesso contenuto, ma indipendente dalla prima.

Un'ulteriore limitazione del meccanismo di *ownership* si presenta durante l'invocazione di una funzione, in particolare, nel passaggio di un valore come parametro. Rust gestisce il passaggio di un parametro a una funzione in maniera analoga agli assegnamenti: le stesse regole di *Copy* e *Move* continuano ad applicarsi.

Si consideri il seguente esempio che illustra il comportamento di Rust nel passaggio a funzione di un valore di tipo semplice e di uno di tipo complesso:

Listing 4: Trasferimento di *ownership* nelle chiamate a funzione

```
fn my_func(first: i32, second: String) {
    println!("Parametro 'first' {}", first);
    println!("Parametro 'second' {}", second);
}
```

⁴ La clonazione può introdurre costi in termini di prestazioni, in quanto richiede operazioni di lettura e scrittura nella memoria *Heap*.

```
fn main() {
    let x: i32 = 5;
    let y: String = String::from("move!");
    my_func(x, y);
    println!("Variabile originale x: {}", x); // <-- Ok
    println!("Variabile originale y: {}", y); // <-- Errore di
        compilazione: ownership trasferita
}
```

Nel listato 4 si osserva che, passando un valore di tipo semplice come `x` (che implementa `Copy`), la variabile rimane valida anche dopo la chiamata alla funzione. Al contrario, passando `y`, di tipo `String` (che implementa `Move`), l'*ownership* viene trasferita al parametro `second`. Terminata l'esecuzione di `my_func`, `second` esce dallo scope e il valore viene deallocato, rendendo `y` una variabile non più valida.

Tale comportamento può risultare limitante in molte situazioni pratiche, dove si desidera utilizzare un valore senza trasferirne la proprietà. Per risolvere questo problema, Rust introduce un secondo meccanismo fondamentale: il *borrowing*.

3.2 BORROWING

Dopo l'*ownership*, il secondo concetto fondamentale su cui si fonda il modello di gestione della memoria di Rust è il *borrowing*. Questo meccanismo consente la condivisione sicura dei dati, permettendo di accedere temporaneamente a un valore senza acquisirne la proprietà.

Rust implementa concretamente il *borrowing* tramite il concetto di *reference*. Una *reference* può essere vista come un puntatore: rappresenta un indirizzo che può essere seguito per accedere al valore memorizzato a tale indirizzo; tale valore appartiene a un'altra variabile.

In generale, le variabili in Rust, sia di tipo primitivo che *reference*, sono immutabili per default: non possono essere modificate una volta assegnato un valore. Nel caso sia necessario modificare una variabile o una *reference*, è possibile utilizzare la parola chiave `mut` durante la sua dichiarazione, indicando che tale variabile (o *reference*) può essere modificata durante l'esecuzione.

Il meccanismo che si occupa di controllare che le regole di *borrowing* vengano rispettate è noto come *Borrow Checker*. Le regole di *borrowing* sono le seguenti:

- In ogni istante può esistere una sola reference mutabile a un dato valore;
- Se esiste almeno una reference immutabile, allora non possono esistere reference mutabili;
- Tutte le reference devono essere sempre valide.
- Una reference mutabile può essere creata solo da una variabile dichiarata come `mut`.⁵

Si considerino i seguenti esempi, che illustrano violazioni delle regole di borrowing. Essi hanno lo scopo di mostrare come tali comportamenti non siano ammessi dal Borrow Checker, generando errori in fase di compilazione.

Listing 5: Reference mutabile a variabile immutabile

```
fn main() {
    let x: u8 = 5;
    let mut y: &mut u8 = &mut x;
    *y += 1;
}
```

Listing 6: Due reference mutabili alla stessa variabile

```
fn main() {
    let mut x: u8 = 5;

    let first_ref = &mut x;
    let second_ref = &mut x;

    *first_ref += 1;
    *second_ref += 1;

    println!("Il valore di x ora \\'e {}", x);
}
```

Listing 7: Coesistenza reference mutabile e immutabile

```
fn main() {
    let mut x: u8 = 5;
```

⁵ Questo vincolo è spesso dato per implicito, ma è fondamentale nel comportamento del *Borrow Checker*.


```

let mutable_ref = &mut x;
let immutable_ref = &x;

*mutable_ref += 1;

println!("Il valore di x ora \\'e {}", immutable_ref);
}

```

Listing 8: Reference non valida: *dangling reference*

```

fn main() {
    let dangling_reference: &u8;

    {
        let x: u8 = 5;
        dangling_reference = &x;
    }

    println!("Il valore che x aveva \\'e {}", *dangling_reference);
}

```

Nel listato 5 si tenta di ottenere un riferimento mutabile a una variabile immutabile, violando un vincolo fondamentale del borrow checker. Questo genera un errore di compilazione:

```

error[E0596]: cannot borrow 'x' as mutable, as it is not
    declared as mutable
3 |     let mut y: &mut u8 = &mut x;
  |                               ^^^^^^

```

Nel listato 6 si tenta di ottenere due riferimenti mutabili simultaneamente, violando la prima regola di borrowing, come indica il seguente errore:

```

error[E0499]: cannot borrow 'x' as mutable more than once
    at a time
4 | let first_ref = &mut x;
  |               ----- first mutable borrow
5 | let second_ref = &mut x;
  |               ^^^^^^ second mutable borrow

```

Il listato 7 mostra invece il tentativo di ottenere contemporaneamente una reference mutabile e una immutabile, violando la seconda regola di borrowing:

```

error[E0502]: cannot borrow 'x' as immutable because it is
    also borrowed as mutable
4 | let mutable_ref = &mut x;
  |                  ----- mutable borrow
5 | let immutable_ref = &x;
  |                  ^^ immutable borrow

```

Infine, nel listato 8 si presenta una *dangling reference*, che viola la terza regola di borrowing:

```

error[E0597]: 'x' does not live long enough
5 |     let x: u8 = 5;
  |         - binding 'x' declared here
6 |     dangling_reference = &x;
  |                         ^^ borrowed value does
   not live long enough
7 | }
  | - 'x' dropped here while still borrowed

```

Come evidenziato dal listato 8, alcune violazioni delle regole di *borrowing* non riguardano soltanto la mutabilità o il numero di reference presenti, ma coinvolgono anche la durata nel tempo di una reference rispetto al valore referenziato.

Rust gestisce queste relazioni temporali tramite il concetto di *lifetime*, che consente al compilatore di determinare se una reference sarà valida finché necessaria, evitando il rischio di *dangling references*.

Il *Borrow Checker*, quindi, sfrutta anche il sistema delle *lifetime* per garantire che la condivisione dei valori avvenga sempre in modo sicuro e coerente. Questo meccanismo sarà approfondito nella prossima sezione.

3.3 LIFETIME

Il sistema delle *lifetime* rappresenta il terzo concetto fondamentale su cui si basa il modello di ownership di Rust. Questo meccanismo viene utilizzato dal *Borrow Checker* per determinare la validità temporale di un prestito (*borrow*).

Nella maggior parte dei casi, le *lifetime* sono implicite e dedotte automaticamente dal compilatore⁶, in questo caso, vengono definite *lifetime generiche*. In contesti particolari, tuttavia, è necessario esplicitarle,

⁶ Per questo motivo, durante lettura di codice Rust, le annotazioni di *lifetime* sono spesso non visibili.

specialmente quando può crearsi ambiguità riguardo la durata di una reference, ad esempio quando si lavora simultaneamente con reference che appartengono a scope differenti.

Si consideri, a riguardo, il seguente esempio, che mostra come le *lifetime* implicite non siano sufficienti in alcuni contesti:

Listing 9: Limitazione delle *lifetime* generiche

```
fn biggest(a: &u8, b: &u8) -> &u8 {  
    if *a > *b { a } else { b }  
}
```

Nel listato 9 si tenta di restituire un riferimento all'intero più grande tra i due parametri forniti. Sebbene il codice possa sembrare corretto, il *Borrow Checker* lo rifiuta, in quanto non può garantire la validità della reference restituita nel tempo. Il problema nasce dal fatto che le reference *a* e *b* potrebbero riferirsi a variabili definite in scope differenti, con durate di vita (*lifetime*) diverse. Il compilatore, con le *lifetime generiche*, cerca di assegnare la stessa durata ad entrambe le reference, generando ambiguità.

Per esplicitare la durata di vita di una reference si utilizza un'annotazione di *lifetime* posta prima del tipo. Questa è rappresentata da un apostrofo seguito da un identificativo; per convenzione si usa una sola lettera, seguendo l'ordine alfabetico ('a', 'b e così via). Riferendoci all'esempio del listato 9, di seguito si presenta la versione corretta, con *lifetime esplicite*:

Listing 10: *Lifetime* esplicite

```
fn biggest<'a>(a: &'a u8, b: &'a u8) -> &'a u8 {  
    if *a > *b { a } else { b }  
}
```

Nel listato 10 viene dichiarata una singola *lifetime*, 'a, che indica al *Borrow Checker* che i parametri *a* e *b* devono avere la stessa durata di vita. Di conseguenza, se le *lifetime* dei parametri differiscono, la compilazione sarà rifiutata.

È importante fare una precisazione: esplicitare una *lifetime* **non modifica** l'effettiva durata di una variabile. Si tratta solamente di un'indicazione semantica che informa il compilatore dei vincoli temporali che devono essere rispettati tra le *reference* coinvolte. Il *Borrow Checker* utilizza queste annotazioni per verificare la validità dei prestiti nel tempo e può rifiutare il codice se i vincoli non sono coerenti.

A tale scopo, si consideri il seguente esempio, che mostra come il *Borrow Checker* possa rifiutare una reference che non rispetti le *lifetime* specificate:

Listing 11: Limitazioni delle *lifetime*

```
fn biggest<'a>(a: &'a u8, b: &'a u8) -> &'a u8 {
    if *a > *b { a } else { b }
}

fn main (){
    let result;
    let smaller = 1;
    {
        let bigger = 2;
        result = biggest(&smaller, &bigger);
    }
    println!("Il piu grande e: {}", result);
}
```

Nel listato 11, la funzione `biggest` viene invocata con due reference che hanno *lifetime* differenti. In particolare, `bigger` viene definita in uno scope interno, e scade prima che possa essere utilizzata la reference restituita da `biggest`. Il *Borrow Checker* si accorge di questa incoerenza e segnala un errore di compilazione:

```
error[E0597]: 'bigger' does not live long enough
 9 |   let bigger = 2;
   |       ----- binding 'bigger' declared here
10 |   result = biggest(&smaller, &bigger);
   |                               ^^^^^^^ borrowed value does
   |                               not live long enough
```

In contesti dove le *lifetime* non vengono esplicitamente annotate, il compilatore applica automaticamente delle regole note come *lifetime elision rules*, che permettono di dedurre in modo implicito i vincoli temporali tra le *reference*. Queste regole sono tre:

- A ciascun parametro che è una reference viene assegnata una *lifetime* distinta;
- Se c'è una sola reference tra i parametri di ingresso, la sua *lifetime* viene assegnata automaticamente al valore di ritorno (se anch'esso è una reference);

- Se ci sono più reference in ingresso, ma una di esse è `&self` o `&mut self`, allora la *lifetime* di `self` viene propagata alle eventuali reference in uscita.

Infine, esiste una *lifetime* speciale, `'static`, il quale indica una reference valida per l'intera durata del programma. Questa *lifetime* tipicamente viene utilizzata per dati costanti o risorse allocate a tempo indeterminato.

Nel prossimo capitolo: *'Sistemi Operativi'* [4](#) verrà mostrato come, grazie a questi concetti, Rust rappresenti un'alternativa valida per la programmazione di sistema, riuscendo a competere con linguaggi come C e C++.

SISTEMI OPERATIVI

Questo capitolo esplorerà le capacità offerte da Rust nel contesto della programmazione di sistema. In apertura verrà fornita una panoramica sul linguaggio C, standard *de facto* nel contesto programmazione di sistema, con particolare attenzione alle caratteristiche che lo rendono popolare in questo ambito.

Successivamente Rust verrà confrontato con C sulla base degli aspetti analizzati, per valutare come Rust possa presentare un'alternativa valida sul piano teorico.

Nel capitolo cinque: *'Progetti e applicazioni reali'* [5](#), l'analisi verrà spostata sul piano pratico, esplorando progetti concreti, che mostrano le potenzialità di Rust nella programmazione di basso livello.

Nel contesto dei sistemi operativi, quali lo sviluppo di kernel, driver o componenti embedded i linguaggi tradizionalmente dominanti sono l'assembly e, con maggiore rilievo, C.

Nonostante l'esistenza di alternative come C++, Lisp, Forth o Bliss, la scelta spesso ricade esclusivamente su C, come testimoniano le codebase dei sistemi operativi più popolari oggi giorno: Windows, MacOS, Linux e Android sono scritti quasi interamente nel linguaggio C.

Ma cosa rende C così popolare in questo ambito? Quali caratteristiche lo distinguono dalle alternative?

4.1 C: MOTIVAZIONI E CARATTERISTICHE

C è un linguaggio di programmazione ampiamente utilizzato nella programmazione di sistema. Il linguaggio offre un livello di astrazione molto vicino al codice macchina (o meglio, all'assembly), garantendo un certo grado di portabilità tra architetture differenti.

Si consideri, per esempio, quanto detto dal creatore del linguaggio:

'[C has] the power of the assembly language and the convenience of ... assembly language.'

— Dennis Ritchie, creatore del linguaggio C, *'Dennis Ritchie: The Shoulders Steve Jobs Stood On'*, *Wired*, 13 Oct 2011

La popolarità di C in questo ambito è da attribuire a diverse sue caratteristiche, alcune delle quali ben documentate[1]. Queste sono principalmente: compilazione, assenza di dipendenze runtime, gestione diretta della memoria, manipolazione a basso livello di bit e corrispondenza quasi diretta al codice macchina.

COMPILAZIONE C è un linguaggio compilato: il file eseguibile generato risulta molto efficiente, sotto l'aspetto della velocità d'esecuzione, rispetto a linguaggi interpretati. Questo lo rende preferibile nell'ambito di sviluppo kernel, dove le prestazioni rappresentano un aspetto cruciale.

ASSENZA DI DIPENDENZE RUNTIME C può essere impiegato per realizzare codice *'bare metal'*, eseguibile direttamente sull'hardware, senza il supporto di un sistema operativo.

C non ha esigenze runtime significative: può funzionare anche senza un allocatore di memoria fornito dal sistema operativo¹. L'unica necessità è quella di un chiamante che invochi la funzione *main*².

ACCESSO DIRETTO ALLA MEMORIA I puntatori C consentono l'accesso diretto a indirizzi di memoria arbitrari, permettendo operazioni di lettura e scrittura dirette.

Tale controllo sulla memoria è cruciale per lo sviluppo di un sistema operativo, dove è richiesto di gestire le tabelle delle pagine, i dispositivi I/O mappati sulla memoria, i controllori DMA e altri;

MANIPOLAZIONE DI BIT Molte interazioni con l'hardware avvengono tramite operazioni bitwise, come riportato da *Ada Computers*[11]. Esempi comuni sono:

- Operazioni di scrittura e lettura dei registri della CPU, come registri di flag di stato. Per esempio, in CPU x86 si trova il registro *EFLAGS*

1 Un programma che non usufruisce della memoria dinamica non richiede un allocatore. Inoltre, nello sviluppo di un sistema operativo, è il programmatore che deve realizzare il proprio sistema di allocazione.

2 In contesti a basso livello, questa chiamata può essere realizzata da un semplice bootstrap assembly.

che contiene una serie di bit di stato (*overflow*, *zero*, *carry*, *interrupt enable* e altri), il loro controllo di solito è fatto tramite l'applicazione di una maschera bitwise;

- Gestione delle periferiche mappate sulla memoria, tramite operazioni di abilitazione e disabilitazione dei singoli pin (e.g. registri *GPIO*);
- Il clock della CPU può essere gestito tramite operazioni bitwise.

Come riportato dal seguente articolo [2], il linguaggio C supporta un ampio range di operazioni bitwise, offrendo i seguenti operatori: *AND* (&), *OR* (|), *XOR* (^), *SHL* (<<), *SHR* (>>), *NOT* (!) e il complemento (~).

SOMIGLIANZA AL CODICE MACCHINA C mantiene una corrispondenza quasi *1-to-1* con codice assembly. Questa trasparenza del linguaggio è fondamentale nello sviluppo di sistemi operativi, in quanto consente di comprendere l'effetto di ogni singola istruzione. C evita strutture dati complesse o astrazioni pesanti che potrebbero mascherare il comportamento a basso livello del programma.

DEBOLEZZA DI C Tuttavia, uno dei punti di forza di C rappresenta anche una sua criticità. L'accesso diretto e non controllato della memoria, unito all'assenza di protezioni a runtime, rende semplice commettere errori potenzialmente gravi come:

- **Accessi non autorizzati alla memoria:** un programma che legge da un indirizzo di memoria arbitrario potrebbe inavvertitamente compromettere la privacy di un'altro processo; una scrittura accidentale potrebbe causare la corruzione della memoria, condivisa o utilizzata da un altro processo;
- **Saturazione della memoria:** un programma che alloca memoria senza controllo o limiti potrebbe esaurire la memoria disponibile. Se lo *swapping* è disponibile, il sistema può degradare drasticamente le prestazioni; in caso non lo fosse, può verificarsi un crash.

4.2 RUST CONTRO C

Nonostante Rust offra numerose astrazioni di alto livello³, è progettato come un linguaggio di basso livello, adatto alla programmazione di sistema. Per comprendere le capacità di Rust nella programmazione di basso livello, verrà confrontato con il linguaggio C, sulla base delle caratteristiche che rendono quest'ultimo favorevole nella programmazione di sistema.

La maggior parte delle informazioni riportate nella sezione è stata ricavata dalla documentazione ufficiale di Rust[7].

- **Compilazione** (Come C): Anche Rust è un linguaggio compilato. Il compilatore genera, in base alla piattaforma, un file direttamente eseguibile;
- **Dipendenze runtime** (Come C): Rust, per configurazione predefinita, ha dipendenze runtime minime (principalmente un allocatore di memoria). Tuttavia, come riportato in *The Embedded Rust Book*[10], tramite la direttiva `#![no_std]` è possibile escludere la libreria standard: in questa configurazione, l'unico requisito è un bootstrap che invochi la funzione `_start` per fare iniziare l'esecuzione;
- **Accesso diretto alla memoria** (Come C): Tramite una parola chiave, *unsafe*, Rust mette a disposizione cinque operazioni denominate *superpoteri non sicuri*: tra queste si trovano anche la possibilità di dereferenziare un puntatore raw (come in C) e la possibilità di eseguire codice C o assembly;
- **Manipolazione di bit** (Come C): Rust offre lo stesso livello di manipolazione dei singoli bit di C, con un'unica differenza: le operazioni bitwise in Rust sono ben definite, evitando condizioni di *Undefined Behaviour* tramite controlli statici sulle dimensioni e tipi degli operandi;
- **Somiglianza al codice macchina** (Diverso da C): Rust generalmente offre astrazioni di alto livello, inoltre il compilatore può introdurre copie e spostamenti aggiuntivi per preservare le condizioni di

³ Rust offre delle astrazioni di alto livello definite *zero cost*: feature quali *iteratori*, *generics*, *smart pointers* e meccanismi di *async/await* che vengono compilate in codice dalle prestazioni equivalenti alle controparti *low-level* scritte a mano.

ownership o inserire controlli su accessi e indici per gli *slice*⁴. Tali comportamenti, pur aumentando la sicurezza, possono produrre un codice meno ‘trasparente’ rispetto alla controparte C;

Infine, a differenza di C, Rust garantisce l’integrità e la sicurezza della memoria già a tempo della compilazione, prevenendo errori comuni legati alla gestione della memoria grazie al *modello di ownership*. In quanto i controlli effettuati dal *Borrow Checker* avvengono a tempo di compilazione, non si hanno dipendenze o overhead introdotti a runtime.

UNSAFE RUST Come già accennato, Rust mette a disposizione, tramite la parola chiave *unsafe*, cinque operazioni principali. Gli sviluppatori del progetto Rust si riferiscono a queste operazioni con l’espressione *Unsafe Superpowers*:

- Dereference di un puntatore raw;
- Invocazione di una funzione o metodo non sicuri;
- Accesso e modifica di una variabile mutabile e statica;
- Implementazione di un tratto non sicuro;
- Accesso ai campi di una *union*.

Le espressioni *Unsafe Rust* e *Unsafe Superpowers* possono risultare fuorvianti: il *Borrow Checker* esegue comunque controlli per garantire la validità delle reference.

La parola chiave permette solamente di eseguire operazioni che, per definizione, sono considerate non sicure. Il compilatore non può controllarne la sicurezza e l’integrità durante la compilazione; all’interno di un blocco *unsafe*, è il programmatore a diventare responsabile di garantire che gli accessi alla memoria siano validi.

La *best practice* riguardante i blocchi non sicuri prevede di ridurre il codice non sicuro il più possibile, utilizzandolo solo quando strettamente necessario. Inoltre, è preferibile incapsulare il codice non sicuro all’interno di astrazioni sicure, fornendo API sicure per il suo utilizzo.

⁴ Slice è un tipo primitivo in Rust. Sono un riferimento a una porzione contigua (in memoria) di elementi di una collezione. Non memorizzano dati esplicitamente, si tratta di una vista su dati esistenti.

4.2.1 Gestione delle risorse

In questa sottosezione verranno analizzati e confrontati i meccanismi offerti da C e Rust per la gestione delle risorse. In particolare verrà analizzata la gestione di: memoria dinamica, file su disco, risorse di rete e *thread* con risorse condivise⁵.

Memoria dinamica

Oltre al *modello di ownership*, Rust incapsula la gestione della memoria dinamica tramite astrazioni note come *smart pointers*, con lo scopo di evitare problematiche comuni relative all'allocazione e deallocazione manuali della memoria. In questa sezione verranno trattati solamente i meccanismi di base per l'allocazione, l'accesso e la liberazione della memoria. L'analisi dettagliata degli errori comuni sarà il fulcro della sottosezione successiva: *Sicurezza della memoria* [4.2.2](#).

In C, la libreria standard (`stdlib.h`) fornisce primitive per la gestione manuale della memoria: `malloc` e `calloc` per l'allocazione, `realloc` per il ridimensionamento e `free` per la deallocazione. Entrambe `malloc` e `calloc` richiedono come argomento la quantità di memoria da allocare, espressa in byte.

Se l'allocazione ha successo, restituiscono un puntatore all'area di memoria allocata, altrimenti restituiscono `NULL`. Tuttavia, il controllo dell'esito è lasciato al programmatore, che deve verificare esplicitamente se il puntatore restituito è valido o meno. Nella pratica, spesso, questa verifica viene omessa, con potenziali gravi conseguenze.

Analogamente, `free` si limita a tentare la deallocazione dell'indirizzo fornito, ma non restituisce nessun esito; inoltre, se viene invocata più volte sullo stesso puntatore, può generare un errore noto come *double free*.

Rust adotta un approccio diverso, basato su *smart pointers* come `Box<T>` o `Vec<T>`, che incapsulano i valori memorizzati nell'heap e ne gestiscono automaticamente il ciclo di vita. Queste astrazioni si fondano sul pattern *RAII*, unendo le operazioni di allocazione e inizializzazione per evitare l'accesso a valori non inizializzati.

Si considerino i seguenti esempi C e Rust che mostrano la gestione del

⁵ Gli esempi di codice riportati in questa sottosezione sono stati sviluppati e testati su ambiente Linux con `glibc`

completo ciclo di vita (allocazione, inizializzazione, accesso e deallocazione) di un intero nella heap:

Listing 12: Gestione ciclo di vita in memoria dinamica C

```
#include <stdlib.h>
#include <stdio.h>
int main(void) {

    int* value = malloc(sizeof(int)); // Allocazione
    if(value == NULL) return -1;

    *value = 52; // Inizializzazione

    printf("%d\n", *value); // Accesso

    free(value); // Deallocazione

    return 0;
}
```

Listing 13: Gestione ciclo di vita in memoria dinamica Rust

```
fn main() {

    let value = Box::<i32>::new(52); // Allocazione e
        inizializzazione

    println!("{}", *value); // Accesso

} // <-- Fine scope: Deallocazione automatica
```

Entrambi i listati 12 e 13 implementano la stessa logica: viene allocato spazio sufficiente per un intero nella memoria heap, successivamente viene inizializzato, stampato e infine deallocato.

Nel caso di C, le varie fasi sono distinte e scollegate: l'allocazione tramite `malloc`, l'inizializzazione e l'accesso tramite dereferenziazione e la deallocazione tramite `free`. Inoltre, è responsabilità del programmatore specificare la dimensione corretta della memoria da allocare, in questo caso tramite `sizeof(int)`. Tuttavia, non vi sono controlli per confermare che la dimensione inserita sia sufficiente o meno, rischiando di generare errori durante l'esecuzione.

Rust, al contrario, segue il paradigma *RAII*: la fase di allocazione

e di inizializzazione sono unificate nella creazione dello *smart pointer* (`Box::new`). La deallocazione avviene automaticamente al termine dello scope, prevenendo sia la mancata, che la doppia, liberazione. L'unico aspetto simile rispetto a C è l'accesso, gestito tramite *dereference*.

File

Sia Rust che C offrono strumenti per eseguire operazioni di input/output su file del filesystem. I due linguaggi si distinguono per il livello di astrazione offerto.

Nel linguaggio C, le principali operazioni di I/O vengono fornite dalle API POSIX, tramite funzioni definite nelle librerie `unistd.h` e `fcntl.h`. L'accesso è basato su *file descriptor*⁶, i quali vengono ottenuti tramite la funzione `open` e successivamente gestiti tramite `read`, `write` e `close`. L'interfaccia rispecchia la natura di basso livello del linguaggio, rappresentando un semplicissimo wrapper alle chiamate di sistema (*syscalls*). L'esito di un'operazione è determinabile attraverso il suo valore di ritorno, solitamente non negativo per indicare il successo e viceversa. La problematica principale deriva dal fatto che la gestione degli errori non è obbligatoria: sta alla discrezione del programmatore controllare la validità dei *file descriptor* o il numero di byte effettivamente letti o scritti, rispetto a quelli aspettati. Spesso, purtroppo, queste verifiche vengono omesse e, come conseguenza, vi è il rischio di letture e scritture su *file descriptor* non validi, risultando spesso in *undefined behaviour*.

Rust d'altra parte, espone una API con un livello di astrazione più alto, tramite il modulo `std::fs`. Tale modulo mette a disposizione la struttura `File`, la quale incapsula interamente la gestione dei *file descriptor*. Le operazioni di lettura e scrittura avvengono tramite funzioni espone dal modulo `std::io`, sia con bufferizzazione che senza. In generale, per letture e scritture piccole, è sufficiente usare le funzioni `read` e `write`, mentre nella maggior parte dei casi, è consigliato l'utilizzo delle strutture con supporto alla bufferizzazione: `BufReader` e `BufWriter`. Le strutture fornite da Rust integrano un meccanismo per la gestione degli errori: le operazioni che possono fallire restituiscono il tipo `Result<T, E>`, forzando di conseguenza il programmatore a gestire esplicitamente l'errore, tramite costrutti `if-else` o `match` per definire una logica di gestione dell'errore, oppure propagando l'errore tramite l'operatore `?`.

⁶ Un *file descriptor* è un intero senza segno che identifica univocamente un file nel contesto di un determinato processo.

Si considerino i seguenti esempi che mostrano un esempio di operazioni di I/O su file:

Listing 14: I/O su file in C

```
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
int main(void) {

    int read_from = open("./read_from.txt", O_RDONLY , S_IRUSR); //
        Apertura file
    int write_to = open("./write_to.txt", O_CREAT | O_WRONLY |
        O_APPEND , S_IWUSR); // Apertura file

    char buf[2048];
    ssize_t read_byte;

    while ( (read_byte = read(read_from, buf, sizeof(buf))) > 0) {
        // Lettura
        write(write_to, buf, read_byte); // Scrittura
    }

    close(write_to); // Chiusura
    close(read_from); // Chiusura
    return 0;
}
```

Listing 15: I/O su file in Rust

```
use std::fs::File;
use std::io::{Read, Write};
fn main() -> std::io::Result<()> {

    let mut read_from = File::open("read_from.txt")?; // Apertura in
        lettura
    let mut write_to = File::create("write_to.txt")?; // Creazione e
        apertura in scrittura

    let mut buf = [0u8; 2048];

    loop {
        let read_byte = read_from.read(&mut buf)?; // Lettura
```

```

    if read_byte == 0 { break }
    write_to.write_all(&buf[..read_byte]); // Scrittura
}
Ok(())
} // Fine scope: Chiusura dei file

```

La logica implementata in entrambi i listati, 14 e 15, è la stessa: dalla directory attuale vengono aperti due file, `read_from` in lettura e `write_to` in scrittura, successivamente viene letto l'intero contenuto di `read_from` e viene scritto su `write_to`, tramite blocchi di 2048 byte.

Nel listato 14, i controlli sugli errori sono stati intenzionalmente omessi per evidenziare che in C tale comportamento è ammesso (l'unico controllo effettuato è su `read`, per determinare la fine della lettura).

Nel listato 15, invece, gli errori vengono gestiti tramite l'operatore `?`, il quale propaga l'errore alla funzione chiamante, garantendo simultaneamente sicurezza e maggiore leggibilità del codice.

Socket

Entrambi i linguaggi mettono a disposizione strumenti per lo sviluppo di applicazioni di rete, in particolare per la comunicazione orientata alla connessione (e.g. socket TCP).

Nel linguaggio C, l'interfaccia di programmazione di rete è fornita principalmente dalle librerie di sistema POSIX:

- `socket`: creazione di un socket;
- `bind`: associazione di un indirizzo IP e porta;
- `listen`: messa in ascolto in attesa di connessioni;
- `accept`: accettazione di una connessione in arrivo;
- `write`: invio di dati sul socket;
- `read`: ricezione di dati dal socket;
- `close`: chiusura del socket;

Questa interfaccia, sebbene molto flessibile, è soggetta a errori comuni, principalmente dovuti alla gestione manuale dei file descriptor⁷ e alla mancata verifica dell'esito delle operazioni.

⁷ In ambienti Unix le socket vengono gestite tramite file descriptor.

Rust, invece, fornisce un'interfaccia di alto livello tramite il modulo `std::net`, che incapsula i socket TCP in due strutture: `TcpListener` per il lato server e `TcpStream` per il lato client. Le principali operazioni sono:

- `TcpListener::bind`: unisce la creazione del socket, il bind e la messa in ascolto;
- `TcpListener::accept`: accetta una connessione in ingresso e restituisce un `TcpStream`;
- `TcpStream::connect`: si connette a un `TcpListener`;
- `TcpStream::read`: riceve dati dal socket, implementando il `Trait Read`;
- `TcpStream::write`: invia dati sul socket, implementando il `Trait Write`.

Queste astrazioni integrano la gestione degli errori avvalendosi del tipo `Result<T, E>`, obbligando il programmatore a gestire esplicitamente sia il caso di successo che quello di fallimento di un'operazione.

I listati 16 e 17 mostrano la realizzazione di un server TCP in entrambi i linguaggi. La logica implementata è la stessa per entrambi: il server si mette in ascolto sulla porta 50000, accetta una connessione in ingresso, legge fino a 1024 byte dal socket e infine scrive, sempre su quest'ultimo, i dati ricevuti.

Nel listato 16 i controlli sugli errori sono stati omessi per evitare un'eccessiva complessità strutturale e facilitare la lettura, al fine di focalizzare l'attenzione sul flusso logico di una connessione TCP.

Per lo stesso motivo, e per mantenere una somiglianza strutturale con l'esempio precedente, nel listato 17 gli errori non vengono gestiti esplicitamente, ma vengono propagati al chiamante tramite l'operatore `?`.

A differenza di C, in cui la gestione degli errori viene lasciata alla discrezione del programmatore, Rust promuove un approccio più sicuro, tramite il tipo `Result` e l'operatore `?`. Questi meccanismi garantiscono la gestione dell'errore a livello di compilazione: il compilatore controlla che entrambi gli scenari (successo e fallimento) vengano gestiti esplicitamente oppure che l'errore venga propagato al chiamante tramite `?`, prevenendo scenari di *undefined behaviour*.

Listing 16: Semplice server TCP in C

```
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
int main(void) {

    int fd = socket(AF_INET, SOCK_STREAM, 0); // Creazione

    struct sockaddr_in addr = {
        .sin_family = AF_INET,
        .sin_port = htons(50000),
        sin_addr.s_addr = INADDR_ANY
    };
    bind(fd, (struct sockaddr*)&addr, sizeof(addr)); // Binding

    listen(fd, 1); // Messa in ascolto

    int client_fd = accept(fd, NULL, NULL); // Accettazione

    char buf[1024];
    ssize_t read_byte = read(client_fd, buf, sizeof(buf)); // Lettura

    write(client_fd, buf, read_byte); // Scrittura

    close(client_fd); // Chiusura
    close(fd);      // Chiusura
    return 0;
}
```

Listing 17: Semplice server TCP in Rust

```

use std::net::{TcpListener, TcpStream};
use std::io::{Read, Write};
fn main() -> std::io::Result<()> {

    let server = TcpListener::bind("0.0.0.0:50000"?; // Creazione,
        binding e messa in ascolto

    let (mut client, _) = server.accept()?; // Accettazione

    let mut buffer = [0; 1024];
    let n = client.read(&mut buffer)?; // Lettura

    client.write(&buffer[..n])?; // Scrittura

    Ok(())
} // Fine scope: Chiusura dei socket

```

Thread

La gestione dei *thread* è fondamentale per un linguaggio di programmazione moderno, in quanto consente di suddividere il lavoro su più unità esecutive e sfruttare il parallelismo in sistemi multicore o multiprocessore. Sia C che Rust offrono meccanismi per la creazione e la gestione dei *thread*, ma con approcci differenti.

La programmazione parallela rappresenta un contesto ampio e complesso, per cui una trattazione sufficientemente dettagliata richiederebbe un capitolo assestante e non rientrerebbe nell'ambito di questa trattazione, che mira a un confronto di base tra i linguaggi.

In C, la programmazione concorrente si basa tipicamente sulla libreria `pthread.h`, parte dello standard POSIX. Questa libreria offre primitive per la creazione (`pthread_create`) e sincronizzazione (`pthread_join`, `pthread_mutex` e altri) dei *thread*. Il completo controllo sui *thread*, fornito dalla libreria, comporta, però, che sia il programmatore stesso a doverne garantire una corretta gestione: sincronizzando i *thread* e garantendo un accesso corretto alle risorse condivise. L'approccio del linguaggio C, infatti, prevede semplicemente di fornire al programmatore gli strumenti necessari, ma è una sua responsabilità adoperarli correttamente: non vengono effettuati controlli sull'utilizzo corretto di *mutex* o risorse condi-

visi durante la fase di compilazione, permettendo di incorrere in errori durante l'esecuzione.

Rust, d'altra parte, tramite il modulo `std::thread`, offre un'interfaccia di livello più alto. Nonostante internamente sia basata su `pthread`, `std::thread` integra strumenti per la corretta gestione di *mutex* e accesso alle risorse condivise, tramite le strutture `Arc<T>` e `Mutex<T>`, che garantiscono la condivisione sicura di dati mutabili tra *thread*.

`Arc<T>` è uno smart pointer a conteggio di riferimento (ha un contatore interno per tenere traccia dei riferimenti) che permette la condivisione non mutabile di dati tra *thread*. `Mutex<T>` è una struttura che fornisce mutua esclusione su una risorsa per l'accesso concorrente. La combinazione `Arc<Mutex<T>>` è la forma idiomatica di Rust per la condivisione sicura di dati mutabili tra *thread*.

Vengono inoltre messi a disposizione due `Trait` fondamentali: `Send`, per indicare che è possibile trasferire la ownership dei valori di un tipo tra più *thread* e `Sync`, per indicare che per un tipo è sicuro avere reference divise su più *thread*.

Il compilatore controlla l'implementazione dei `Trait` per determinare la validità degli accessi alle risorse, riducendo la probabilità di errori di concorrenza.

In sintesi, nonostante il modello di Rust possa sembrare più complesso o verboso rispetto alla flessibilità offerta dalle librerie C, esso garantisce un accesso sicuro alle risorse condivise e, in generale, riesce a prevenire a livello di compilazione una classe di errori comuni: *data race* (accesso concorrente ai dati di cui almeno uno in scrittura).

Conclusioni

In conclusione, sotto l'aspetto della gestione delle risorse, i due linguaggi offrono strumenti analoghi, ma con differenze non trascurabili.

C permette un controllo maggiore e diretto delle risorse, ma questo avviene a un costo, come la gestione manuale di puntatori, *file descriptor* e buffer; inoltre, la maggior parte delle funzioni utilizzate per interagire con file, *socket* o *thread* si basano su API POSIX (su sistemi *POSIX-compliant*), non garantendo la compatibilità tra più piattaforme, come Windows.

D'altra parte, Rust offre un controllo più restrittivo ma semplificato, spesso incapsulando le risorse in strutture astratte, permettendo una gestione sicura delle risorse e dei relativi errori. Inoltre, a differenza di C, Rust garantisce che il codice sviluppato sia *cross-platform* in quanto

le strutture `File`, `TcpStream`, `TcpListener` e `Thread` sono indipendenti dalla piattaforma: l'interfaccia rimane la stessa, ma in base alla piattaforma specifica vengono utilizzati strumenti differenti di basso livello (e.g. i `pthread` POSIX, definiti in `pthread.h` e i `thread` Win32, definiti in `windows.h`).

4.2.2 Sicurezza della memoria

In questa sottosezione verranno analizzati alcuni tra gli errori più comuni legati alla gestione della memoria, mostrando esempi di codice scritti in C e confrontandoli, quando possibile, con le controparti in Rust.

Un aspetto interessante del modello di memoria di Rust è che alcuni errori non sono semplicemente rilevati durante l'esecuzione, o a tempo di compilazione, ma sono strutturalmente impossibili da generare: non vi è modo di scrivere codice Rust che generi tali errori. Come già accennato, il *modello di ownership* guida la struttura di un programma in Rust: spinge il programmatore a ragionare in termini di *ownership* e *lifetime*. Inoltre, l'allocazione della memoria dinamica è gestita tramite l'utilizzo di *smart pointers*, che forniscono un'astrazione sicura per la gestione della memoria dinamica.

Unfreed memory

La *unfreed memory* (memoria non liberata) rappresenta un errore comune durante la gestione della memoria dinamica tramite approccio manuale. Si ha in quei contesti in cui aree di memoria vengono allocate, senza successiva deallocazione. Le aree di memoria interessate non possono essere riutilizzate per allocazioni successive in quanto vengono considerate ancora utilizzate.

Si consideri il seguente esempio minimale C che genera un'errore di *unfreed memory*:

Listing 18: Unfreed memory in C

```
#include <stdlib.h>
int main(void) {
    void* mem = malloc(sizeof(int));
    return 0;
}
```

In Rust questo comportamento non si potrà mai realizzare, in quanto la deallocazione viene gestita implicitamente dal compilatore⁸.

Si consideri un programma in Rust che, per quanto possibile, replica il comportamento del listato 18:

Listing 19: Unfreed memory in Rust

```
fn main() {
    let mem = Box::<u32>::new(0);
}
```

Nel listato 18 viene allocata memoria nella heap sufficiente a contenere un intero, successivamente il programma termina senza liberarla⁹. Il programma viene compilato ed eseguito correttamente, nonostante la presenza di questo errore. Tuttavia, tale memoria risulta inaccessibile per ulteriori allocazioni da parte del processo durante la sua esecuzione.

Il listato 19 rappresenta il codice Rust più vicino possibile a quello riportato nel listato 18. La differenza principale è che la memoria viene deallocata automaticamente: al termine dello scope della funzione `main`, la variabile `mem` viene eliminata e la relativa memoria deallocata.

Di conseguenza, l'errore di *unfreed memory* non può manifestarsi, in quanto la deallocazione avviene in maniera trasparente agli occhi del programmatore.

Double free

La *double free* (doppia liberazione) è un'errore di gestione della memoria che si verifica quando la stessa area di memoria viene deallocata più di una volta. Ciò può causare la corruzione della memoria heap, con conseguente comportamento indefinito o crash del programma.

Nel linguaggio C, questo si verifica invocando la funzione `free` sullo stesso puntatore più volte, come mostrato nel seguente esempio:

Listing 20: Double free in C

```
#include <stdlib.h>
int main(void) {
```

⁸ Il compilatore inserisce chiamate alla funzione *drop* in corrispondenza della fine di uno scope. Questo è realizzabile in quanto il *Borrow Checker* controlla le relazioni di *ownership* e *borrowing* per determinare chi sia responsabile della deallocazione, ovvero su quali variabili invocare *drop*.

⁹ In realtà l'allocazione avviene nello spazio di memoria virtuale del processo; a termine dell'esecuzione, il sistema operativo recupera tutte le risorse allocate dal processo.

```

    void* ptr = malloc(sizeof(int));
    free(ptr);
    free(ptr);
    return 0;
}

```

Si consideri invece il seguente programma Rust che tenta di replicare lo stesso comportamento del listato 20:

Listing 21: Double free in Rust

```

fn main() {
    let mem = Box::<u32>::new(0);
    std::mem::drop(mem);
    std::mem::drop(mem);
}

```

Nel listato 20 la memoria puntata da `ptr` viene deallocata due volte. In fase di esecuzione questo può portare a un crash del programma, come mostrato nell'immagine 1.

Tuttavia, lo standard C non definisce un comportamento specifico da adottare in questo contesto, causando *undefined behaviour* nella pratica¹⁰.

Per quanto riguarda Rust, è possibile considerare due scenari:

- Il listato 19 rappresenta il comportamento idiomatico di Rust: la memoria viene deallocata automaticamente alla fine di uno *scope*, senza necessità di interventi manuali;
- Nel listato 21, invece, si tenta di deallocare esplicitamente la memoria con due chiamate consecutive a `std::mem::drop`¹¹. Tuttavia, la funzione `drop` prende possesso del valore, rendendo la reference originale, e di conseguenza la seconda chiamata, invalida. Il compilatore si accorge della violazione della terza regola di *borrowing* (*Tutte le reference devono essere valide*), impedendo la compilazione e generando l'errore riportato nell'immagine 2.

10 In ambiente Linux con `glibc`, viene rilevato l'errore causando la stampa di un messaggio e l'interruzione dell'esecuzione. In ambienti Windows, l'errore può passare inosservato o causare un crash silenzioso.

11 Questa funzione assume *ownership* del valore fornito, causandone la deallocazione una volta raggiunta la fine della funzione.

```

frank@francyy:~/Documents/thesis$ gcc double-free.c -o double-free-c
frank@francyy:~/Documents/thesis$ ./double-free-c
free(): double free detected in tcache 2
Aborted (core dumped)

```

Figura 1: *Double free* in C

```

frank@francyy:~/Documents/thesis$ rustc double-free.rs
error[E0382]: use of moved value: `mem`
  --> double-free.rs:4:20
   |
 2 |     let mem = Box::<u32>::new(0);
   |     --- move occurs because `mem` has type `Box<u32>`, which does not implement
the `Copy` trait
 3 |     std::mem::drop(mem);
   |     --- value moved here
 4 |     std::mem::drop(mem);
   |     ^^^ value used here after move

```

Figura 2: Tentativo di *double free* in Rust

Dangling pointers

Un *dangling pointer* (puntatore pendente) è un puntatore che riferisce a una locazione di memoria non più valida o che è stata liberata. Generalmente si presenta quando un'area di memoria viene deallocata, ma gli eventuali puntatori che la referenziavano non vengono aggiornati.

Si consideri il seguente esempio C che genera un *dangling pointer*¹²:

Listing 22: Dangling pointer in C

```

int main(void) {
    int* ptr;
    {
        int* val = malloc(sizeof(int));
        *val = 30;
        ptr = val;
        free(val);
    }
}

```

Nel listato 22, viene allocata memoria sufficiente per un intero e il suo indirizzo è memorizzato sia in `val` che in `ptr`. La memoria viene successivamente deallocata invocando la funzione `free` sul puntatore `val`.

¹² La presenza di uno scope interno non è funzionale all'esempio, ma non ne modifica nemmeno il risultato. È presente per garantire somiglianza strutturale con l'esempio Rust successivo.

Tuttavia, il puntatore `ptr` non viene aggiornato, continuando a riferire alla stessa area di memoria, ormai non più valida.

È importante osservare che la sola esistenza di un *dangling pointer* non causa un errore immediato. Il comportamento indefinito si manifesta solo quando si tenta di dereferenziare tale puntatore, cercando di leggere o scrivere nella memoria a cui fa riferimento.

A dimostrazione di ciò, si consideri che Rust permette la creazione di una *dangling reference*, a patto che non venga utilizzata. Il seguente esempio mostra questa possibilità:

Listing 23: Dangling reference in Rust

```
fn main() {
    let _ref: &u32;
    {
        let val = Box::<u32>::new(30);
        _ref = &*val;
    }
}
```

Il listato 23 tenta di replicare, per quanto possibile, il comportamento descritto nel listato 22: alla variabile `_ref` viene assegnato un riferimento a `val`, allocata dinamicamente in uno scope interno. All'uscita da tale scope, `val` viene deallocato, lasciando `_ref` con un riferimento non valido.

Nonostante ciò, il compilatore non genera alcun errore. Questo comportamento è giustificato dal fatto che la reference non è mai utilizzata¹³ e, di conseguenza, il *Borrow Checker* non rileva alcuna violazione delle regole di *borrowing*.

Access after free

Sebbene la sola presenza di un *dangling pointer* non causi immediatamente problemi, essa rappresenta una condizione necessaria per un errore più grave: la *access after free* (accesso post-liberazione). Questo errore si verifica quando si tenta di accedere a memoria precedentemente deallocata, dereferenziando un *dangling pointer*.

A seconda del tipo di accesso, si possono avere conseguenze differenti:

- **Lettura:** può causare il recupero di dati non validi o incoerenti, eventualmente sovrascritti da allocazioni successive;

¹³ Il *Borrow Checker* lavora in maniera *lazy*: non verifica le condizioni di *borrowing* e *lifetime* di una reference fino a che non viene utilizzata.

- **Scrittura:** può compromettere l'integrità della memoria, causando comportamenti indefiniti o crash del programma.

Si consideri il seguente esempio C che, estendendo il listato 22, genera un errore di *access after free*:

Listing 24: Access after free in C

```
#include <stdlib.h>
#include <stdio.h>
int main(void) {
    int* ptr;
    {
        int* val = malloc(sizeof(int));
        *val = 30;
        ptr = val;
        free(val);
    }
    printf("%d\n", *ptr);
}
```

Nel listato 24 si tenta di dereferenziare il puntatore `ptr` dopo che la memoria a cui faceva riferimento è stata deallocata. In questo caso, l'accesso avviene per effettuare un'operazione di lettura, che rappresenta una forma meno distruttiva rispetto a una scrittura, ma che resta comunque pericolosa: i dati letti potrebbero essere non validi o casuali, in quanto la memoria potrebbe essere stata sovrascritta da allocazioni successive.

Questo comportamento rappresenta un esempio di *undefined behaviour*, come illustrato dall'immagine 3.

Per osservare come Rust gestisce questa problematica, si consideri il seguente esempio, il quale estende il listato 23:

Listing 25: Tentativo di *access after free* in Rust

```
fn main() {
    let _ref: &u32;
    {
        let val = Box::<u32>::new(30);
        _ref = &*val;
    }
    println!("{}", _ref);
}
```

Il listato 23 tenta di replicare il comportamento del listato 24, cercando di accedere a una reference pendente per un'operazione di lettura.

Tuttavia, in questo caso viene generato un errore di compilazione, come mostrato nell'immagine 4. Il *Borrow Checker* rileva che `_ref` viene utilizzata successivamente alla deallocazione di `val`, rappresentando una violazione delle regole di *borrowing* e, di conseguenza, impedisce la compilazione.

```
frank@francyy:~/Documents/thesis$ gcc access-after-free.c -o access-after-free-c
frank@francyy:~/Documents/thesis$ ./access-after-free-c
1486727133
```

Figura 3: *Access after free* in C

```
frank@francyy:~/Documents/thesis$ rustc access-after-free.rs
error[E0597]: `*val` does not live long enough
--> access-after-free.rs:5:16
   |
4 |     let val = Box::<u32>::new(30);
   |     --- binding `val` declared here
5 |     _ref = &*val;
   |           ^^^^^ borrowed value does not live long enough
6 | }
   | - `*val` dropped here while still borrowed
7 | println!("{}", _ref);
   |           ---- borrow later used here
```

Figura 4: Tentativo di *access after free* in Rust

Access out of bounds: buffer overflow e overread

L' *access out of bounds* è un errore comune associato alla gestione manuale della memoria dinamica. Si verifica quando si tenta di accedere a una porzione di memoria al di fuori dei limiti dell'area effettivamente allocata.

In base al tipo di accesso si distingue in due varianti: nel caso di una lettura si parla di *buffer overread* (lettura oltre i limiti) mentre, nel caso di una scrittura, di *buffer overflow* (sovraccarico o sovrascrittura del buffer).

BUFFER OVERFLOW Si consideri il seguente esempio C che genera un errore di *buffer overflow*:

Listing 26: Buffer overflow in C

```

#include <stdlib.h>
#include <stdio.h>
int main(void) {
    int* vec = malloc(sizeof(int) * 3);
    printf("Allocato un vettore di 3 interi, indici validi: 0, 1,
        2\n");
    for(int i = 0; i <= 3; i++) {
        printf("Memorizzando %d all'indice %d\n", i * 10, i);
        vec[i] = i * 10;
    }
    free(vec);
    printf("Terminata la memorizzazione!\n");
    return 0;
}

```

Nel listato 26 viene allocata memoria sufficiente per un vettore di tre interi, ma successivamente vengono inizializzati quattro elementi, dall'indice 0 fino a 3. In questo caso il programma termina senza errori, viene anche eseguita la stampa finale, come mostrato dall'immagine 5.

Nonostante ciò, questo comportamento non può essere considerato una garanzia in quanto, generalmente, la conseguenza di un *buffer overflow* è un *undefined behaviour*: non è possibile determinare a priori cosa contenga la memoria oltre il limite dell'allocazione o per cosa venga utilizzata¹⁴.

Per comprendere come Rust gestisca questo errore, si consideri il seguente esempio:

Listing 27: Buffer overflow in Rust

```

fn main() {
    let mut vec: Vec<u32> = vec![0; 3];
    println!("Allocato un vettore di 3 interi, indici validi: 0, 1,
        2");
    for i in 0u32..=3u32 {
        println!("Memorizzando {} all'indice {}", i * 10, i);
        vec[i as usize] = i * 10;
    }
    println!("Terminata la memorizzazione!");
}

```

¹⁴ In generale un *buffer overflow* avviene all'interno dello spazio di indirizzamento virtuale di un processo. In questo caso, la memoria potenzialmente corrotta sarebbe del processo stesso. Tuttavia, nel caso in cui l'indirizzo di riferimento fosse al di fuori dello spazio di indirizzamento virtuale del processo verrebbe generato un'errore runtime di tipo *segmentation fault*, causando l'immediato crash del programma.

```
}

```

Il listato 27 tenta di replicare il comportamento del listato 26, allocando un vettore di tre interi e successivamente cercando di inizializzarne quattro. In questo caso la compilazione va a buon fine, ma la stampa finale non avviene: il programma genera un *panic* durante l'esecuzione. Questo avviene durante l'accesso all'indice 3, come mostrato nell'immagine 6.

Questo comportamento è dovuto al fatto che il compilatore inserisce controlli di validità sugli indici, i quali vengono eseguiti a runtime.¹⁵

```
frank@francyy:~/Documents/thesis$ gcc buffer-overflow.c -o buffer-overflow-c
frank@francyy:~/Documents/thesis$ ./buffer-overflow-c
Allocato un vettore di 3 interi, indici validi: 0, 1, 2
Memorizzando 0 all'indice 0
Memorizzando 10 all'indice 1
Memorizzando 20 all'indice 2
Memorizzando 30 all'indice 3
Terminata la memorizzazione!
```

Figura 5: *Buffer overflow* in C

```
frank@francyy:~/Documents/thesis$ rustc buffer-overflow.rs -o buffer-overflow-rust
frank@francyy:~/Documents/thesis$ ./buffer-overflow-rust
Allocato un vettore di 3 interi, indici validi: 0, 1, 2
Memorizzando 0 all'indice 0
Memorizzando 10 all'indice 1
Memorizzando 20 all'indice 2
Memorizzando 30 all'indice 3

thread 'main' panicked at buffer-overflow.rs:6:12:
index out of bounds: the len is 3 but the index is 3
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Figura 6: *Buffer overflow* in Rust

BUFFER OVERREAD Si consideri il seguente esempio C che genera un errore di *buffer overread*:

Listing 28: Buffer overread in C

```
#include <stdlib.h>
#include <stdio.h>
int main(void) {
    int* vec = malloc(sizeof(int) * 3);
```

¹⁵ Infatti, il compilatore non può sapere a tempo di compilazione quali saranno gli indici che verranno utilizzati per accedere a un vettore, quindi si limita a inserire controlli sulla loro validità prima degli accessi effettivi.

```

    for(int i = 0; i < 3; i++) vec[i] = i * 10;
    printf("Contenuto del vettore:\n");
    for(int i = 0; i < 10; i++)
        printf("Indice: %d -> Valore: %d\n", i, vec[i]);
    free(vec);
    return 0;
}

```

Nel listato 28 viene allocata memoria per un vettore di tre interi, i quali vengono correttamente inizializzati e, successivamente, si tenta di leggere dieci elementi dal vettore. In questo caso il programma termina senza errori, stampando effettivamente dieci elementi, come mostrato dall'immagine 7.

Come quanto detto per l'errore di *buffer overflow*, generalmente la conseguenza è un *undefined behaviour*: non è possibile determinare a priori cosa contenga la memoria (oltre il limite dell'allocazione) che viene letta¹⁶.

Rust gestisce questo errore in maniera analoga al *buffer overflow*. Si consideri a dimostrazione di ciò il seguente esempio:

Listing 29: Buffer overread in Rust

```

fn main() {
    fn main() {
        let vec: Vec<u32> = vec![0, 10, 20];
        println!("Contenuto del vettore:");
        for i in 0..10 {
            println!("Indice: {} -> Valore: {}", i, vec[i]);
        }
    }
}

```

Il listato 29 tenta di replicare il comportamento del listato 28, allocando un vettore di tre interi e successivamente cercando di leggerne dieci.

Anche in questo caso la compilazione termina correttamente ma, durante l'esecuzione, in particolare durante l'accesso al quarto elemento, viene generato un *panic*.

Questo comportamento può essere osservato nell'immagine 8.

16 Vangono le stesse considerazioni fatte per il *buffer overflow*: un'accesso a un indirizzo che eccede lo spazio di indirizzi virtuali del processo genera un errore runtime di tipo *segmentation fault*, causando l'immediato crash del programma

```

frank@francyy:~/Documents/thesis$ gcc buffer-overread.c -o buffer-overread-c
frank@francyy:~/Documents/thesis$ ./buffer-overread-c
Contenuto del vettore:
Indice: 0 -> Valore: 0
Indice: 1 -> Valore: 10
Indice: 2 -> Valore: 20
Indice: 3 -> Valore: 0
Indice: 4 -> Valore: 0
Indice: 5 -> Valore: 0
Indice: 6 -> Valore: 1041
Indice: 7 -> Valore: 0
Indice: 8 -> Valore: 1768189513
Indice: 9 -> Valore: 540697955

```

Figura 7: *Buffer overread* in C

```

frank@francyy:~/Documents/thesis$ rustc buffer-overread.rs -o buffer-overread-rust
frank@francyy:~/Documents/thesis$ ./buffer-overread-rust
Contenuto del vettore:
Indice: 0 -> Valore: 0
Indice: 1 -> Valore: 10
Indice: 2 -> Valore: 20

thread 'main' panicked at buffer-overread.rs:5:52:
index out of bounds: the len is 3 but the index is 3

```

Figura 8: *Buffer overread* in Rust

Uninitialized memory access

La *uninitialized memory access* (accesso a memoria non inizializzata) è un errore che si presenta quando si tenta di leggere da un'area di memoria che non è stata inizializzata. Questo può portare a comportamenti imprevedibili, in quanto i dati letti potrebbero essere casuali e potenzialmente residui da allocazioni precedenti.

Nel linguaggio C, questo errore si presenta tipicamente quando viene allocata memoria tramite la funzione `malloc` e successivamente si tenta di accedervi senza previa inizializzazione.

Si consideri il seguente esempio C che genera un errore di questo tipo:

Listing 30: Uninitialized memory access in C

```

#include <stdlib.h>
#include <stdio.h>
int main(void) {
    int* ptr = malloc(sizeof(int));
    printf("%d\n", *ptr);
    free(ptr);
    return 0;
}

```

Nel listato 30 viene allocata memoria sufficiente a contenere un intero e, senza inizializzarne il contenuto, viene tentato un accesso in lettura. Il risultato è un comportamento indefinito: il valore letto potrebbe variare in base all'ambiente di esecuzione e tra esecuzioni, senza garanzia di coerenza.

È possibile osservare un caso particolare, in cui l'accesso può sembrare produrre un risultato coerente, come mostrato nell'immagine 9¹⁷. Tuttavia, questa apparente stabilità non può essere considerata una garanzia, in quanto vi sono più fattori che possono influenzare il valore letto:

- Implementazioni specifiche dell'allocatore di memoria, che possono inizializzare la memoria allocata a zero o a un valore casuale;
- La posizione in memoria scelta dall'allocatore per il puntatore;
- Eventuali valori residui da allocazioni precedenti;
- La mappatura tra pagine virtuali e fisiche da parte del sistema operativo.

A conferma dell'inaffidabilità, si consideri il seguente esempio, che estende il listato 30:

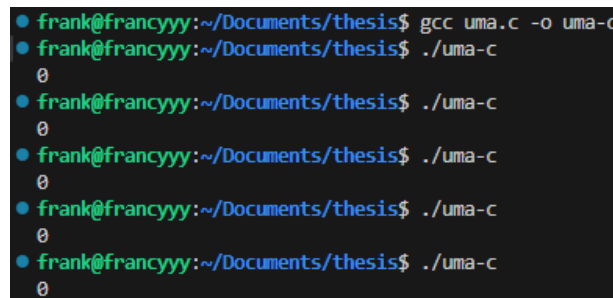
Listing 31: Uninitialized memory access in C

```
#include <stdlib.h>
#include <stdio.h>
int main(void) {
    int* ptr = malloc(sizeof(int));
    free(ptr);
    ptr = malloc(sizeof(int));
    printf("%d\n", *ptr);
    free(ptr);
    return 0;
}
```

Il listato 31 mostra come eseguire una `free` prima della `malloc` successiva può lasciare residui nella memoria. Di conseguenza il valore letto, questa volta, è diverso da zero, come si può osservare dall'immagine 10¹⁸.

17 L'esecuzione è avvenuta su Ubuntu LTS 24.04 con glibc. In questo contesto è possibile che `malloc` azzeri la memoria allocata, prima di restituirne l'indirizzo, ma non è un comportamento garantito dallo standard C.

18 Anche in questo caso, si tratta di un comportamento specifico del sistema e non garantito dallo standard.



```
frank@francyyy:~/Documents/thesis$ gcc uma.c -o uma-c
frank@francyyy:~/Documents/thesis$ ./uma-c
0
frank@francyyy:~/Documents/thesis$ ./uma-c
0
frank@francyyy:~/Documents/thesis$ ./uma-c
0
frank@francyyy:~/Documents/thesis$ ./uma-c
0
frank@francyyy:~/Documents/thesis$ ./uma-c
0
```

Figura 9: *Uninitialized memory access* in C

Rust previene questa problematica in maniera strutturale, non permettendo allocazioni dinamiche senza inizializzazione grazie all'utilizzo di smart pointers, i quali prevedono un'inizializzazione obbligatoria.

In Rust infatti, le principali primitive per l'allocazione dinamica richiedono sempre l'inizializzazione dei valori:

- **Box**: *smart pointer* impiegato per allocare un singolo valore nella heap. L'unico modo per crearne uno è tramite `Box::new`, che richiede un valore per l'inizializzazione. In *safe* Rust, non esiste un costrutto equivalente alla `malloc` in C;
- **Vec**: *smart pointer* utilizzato per una collezione dinamica di valori in heap. Anche se può essere creato vuoto (`Vec::new`), ogni accesso è verificato a runtime per evitare accessi oltre i limiti. In caso di violazioni viene generato un *panic*, terminando l'esecuzione, come è osservabile nell'immagine 8;

Di conseguenza, non è possibile sviluppare codice Rust che acceda a memoria non inizializzata senza ricorrere alla parola chiave `unsafe`¹⁹.

Conclusioni

In conclusione, sotto l'aspetto della sicurezza della memoria, Rust si distingue da C garantendo un comportamento sicuro e deterministico, prevenendo alcuni errori durante la compilazione e rilevandone altri a runtime. In C, d'altra parte, tali errori possono passare inosservati, causando *undefined behaviour* o crash del programma.

¹⁹ Rust consente l'accesso a memoria potenzialmente non inizializzata, ma soltanto all'interno di blocchi *unsafe*. Un esempio è la funzione `std::mem::MaybeUninit`, ma meccanismi come questo sono riservati a casi particolari, in cui le garanzie di sicurezza devono essere gestite manualmente dal programmatore.

```

frank@francyy:~/Documents/thesis$ gcc uma-2.c -o uma-2-c
frank@francyy:~/Documents/thesis$ ./uma-2-c
158008856
frank@francyy:~/Documents/thesis$ ./uma-2-c
1435864009
frank@francyy:~/Documents/thesis$ ./uma-2-c
1469505509
frank@francyy:~/Documents/thesis$ ./uma-2-c
1526610248
frank@francyy:~/Documents/thesis$ ./uma-2-c
1645977410

```

Figura 10: *Uninitialized memory access* in C con previa free

4.2.3 Prestazioni

In questa sottosezione verranno confrontate le prestazioni tipiche dei programmi scritti in C e Rust, basandosi su benchmark esistenti e pubblicamente disponibili.

Non verranno presentati né codice dettagliato né un’analisi sperimentale diretta, per i seguenti motivi: una valutazione accurata richiederebbe sviluppo di codice ottimizzato per entrambi i linguaggi, un set ampio di programmi da testare, differenti ambienti di esecuzione (diversi sistemi operativi, compilatori, architetture hardware) e una metodologia di confronto rigorosa, che andrebbero oltre gli obiettivi di questa trattazione.

Come accennato a inizio del capitolo, il linguaggio C è storicamente noto per le prestazioni, principalmente in termini di velocità d’esecuzione, dovute all’assenza di astrazioni complesse e al runtime minimo richiesto.

Rust, d’altra parte, mira a garantire la sicurezza della memoria senza compromettere le prestazioni, puntando a fornire velocità d’esecuzione paragonabili a C e C++.

Per offrire un confronto concreto, vengono riportati i risultati tratti da *The Computer Language Benchmarks Game*[14]²⁰, un progetto che confronta le prestazioni di diversi linguaggi di programmazione nell’esecuzione di algoritmi comuni. Il confronto si basa su quattro parametri:

- **secs**: tempo di esecuzione totale (*wall-clock time*);

²⁰ Nel sito vengono raccolti i risultati dell’esecuzione di vari algoritmi sviluppati in diversi linguaggi di programmazione. Il sito potrebbe variare nel tempo e i risultati riportati nella trattazione potrebbero non essere più i migliori e peggiori (data di riferimento 2025-07-18)

- **cpu secs**: tempo effettivo utilizzato dalla CPU, trascurando attese dovute a I/O e context switch;
- **mem**: picco di memoria RAM utilizzata dal processo;
- **gz**: dimensione del file sorgente compresso con gzip, interpretato come indice di verbosità e complessità dell'implementazione.

Verranno analizzati due benchmark distinti: uno principalmente *CPU-bound* e uno principalmente *I/O-bound*. Non viene incluso un esempio della categoria '*contentious*', in quanti tali benchmark tendono a essere influenzati da librerie esterne e dalla specifica implementazione degli algoritmi, non tanto dalle caratteristiche intrinseche del linguaggio e, per questo, non rientrano negli obiettivi della trattazione.

Confronto CPU-bound

Il benchmark *CPU-bound* considerato è *n-body*. Il problema consiste nel simulare il moto di *n* corpi che interagiscono attraverso la forza gravitazionale: a ogni passo viene calcolata la forza tra tutte le coppie di corpi, si aggiornano le velocità in base alle accelerazioni risultanti e si aggiornano le posizioni. L'algoritmo ha complessità quadratica ($O(n^2)$) ed è privo di I/O significativo: è un carico interamente *CPU-bound*.

Nella tabella 1 sono riportati i risultati del benchmark *n-body* relativi a Rust e C. Per ciascun linguaggio è stata selezionata l'esecuzione migliore e quella peggiore in termini di **cpu secs**. Sono state considerate solo le implementazioni ideomatiche, escludendo quelle che ricorrono a *SIMD scritti a mano*²¹ o a codice *unsafe* in Rust.

Dagli estratti riportati nella tabella 1 è possibile osservare che, da un punto di vista temporale, anche in un benchmark puramente *CPU-bound*, Rust riesca a raggiungere C in termini di velocità d'esecuzione, senza ricorrere a codice *unsafe*. Nei benchmark Rust ha, addirittura, superato C in velocità d'esecuzione: nel caso migliore, Rust ha impiegato circa il 30% di tempo in meno; nel caso peggiore circa il 20%.

Questo risultato supporta l'idea che Rust possa rappresentare un'alternativa a C anche in contesti ad alta intensità computazionale, pur garantendo la sicurezza della memoria.

21 Si tratta di un'ottimizzazione manuale del codice per sfruttare le istruzioni vettoriali della CPU: invece di scrivere codice 'standard', il programmatore scrive esplicitamente codice che utilizza i registri *Single Instruction, Multiple Data*.

Linguaggio	secs	cpu secs	mem (KB)	gz (B)
Esecuzione migliore				
C	4.98	4.98	2482	1186
Rust	3.46	3.46	2937	1774
Esecuzione peggiore				
C	6.89	6.89	2490	1250
Rust	5.52	5.51	3027	1483

Tabella 1: Risultati n-body per Rust e C (implementazioni ideomatiche)

Dal punto di vista dell'occupazione di risorse, invece, C si presenta vantaggioso, richiedendo una quantità inferiore di memoria durante l'esecuzione e producendo sorgenti più compatti (successivamente alla compressione).

Confronto I/O-bound

Il benchmark *I/O-bound* selezionato è *fasta*. Il problema consiste nella generazione di DNA (utilizzando i caratteri A, C, G e T) secondo pattern sia deterministici che pseudocasuali, seguita dalla loro scrittura sullo standard output. L'algoritmo presenta un carico di elaborazione minimo: rappresenta un esempio di algoritmo principalmente *I/O-bound*, in quanto la maggior parte del tempo di esecuzione è trascorso in scrittura sullo standard output. Inoltre, la presenza di generazione pseudocasuale e ripetitiva comporta un utilizzo intensivo di stringhe e buffer, rendendo il benchmark utile per verificare l'efficienza di gestione della memoria dinamica.

Nella tabella 2 sono riportati i risultati del benchmark *fasta* relativi a Rust e C. Come specificato nell'analisi CPU-bound 4.2.3, per ciascun linguaggio viene riportata l'esecuzione migliore e quella peggiore in termini di **cpu secs**. Sono state considerate solamente le implementazioni ideomatiche, escludendo quelle che ricorrono a *SIMD scritti a mano* o codice *unsafe* in Rust.

Dai risultati mostrati nella tabella 2 si osserva che non emerge un chiaro vantaggio costante tra i due linguaggi: nel caso migliore, C è circa 2.6 volte più veloce di Rust, mentre nel caso peggiore è Rust a richiedere circa la metà del tempo rispetto a C.

Linguaggio	secs	cpu secs	mem (KB)	gz (B)
Esecuzione migliore				
C	0.79	0.78	2236	1469
Rust	2.03	2.03	3199	1235
Esecuzione peggiore				
C	8.27	8.27	2195	839
Rust	4.45	4.45	3113	1240

Tabella 2: Risultati del benchmark fasta per Rust e C (implementazioni ideomatiche)

Dal punto di vista dell'occupazione di risorse, valgono ancora le considerazioni fatte durante l'analisi *CPU-bound* 4.2.3: C, generalmente, richiede una quantità minore di memoria RAM durante l'esecuzione e produce sorgenti più contenuti (una volta compressi), indicando una complessità minore dell'implementazione.

Conclusione

Dai risultati analizzati emerge che entrambi i linguaggi offrono prestazioni paragonabili, con differenze che variano in base al tipo di carico e all'implementazione specifica.

Con le opportune ottimizzazioni, sia a livello di scrittura del codice che di compilazione, Rust è in grado di raggiungere, e in alcuni casi superare 4.2.3, le prestazioni di C in termini di velocità d'esecuzione, pur garantendo sicurezza della memoria.

Tuttavia, un evidente svantaggio di Rust è rappresentato dagli elevati tempi di compilazione, specialmente rispetto a C, dovuti principalmente ai controlli effettuati dal *Borrow Checker* per determinare, e verificare, le relazioni di *ownership* e *borrowing*. Questo overhead in fase di compilazione è però compensato dalla sicurezza della memoria, garantita a tempo di compilazione, non richiedendo overhead aggiuntivi durante l'esecuzione.

Dal punto di vista della dimensione dei file eseguibili, Rust tende a generare file di dimensioni maggiori, principalmente per l'inclusione della libreria standard e dei simboli di debug²².

²² Tramite le opzioni del compilatore è possibile indicare l'ottimizzazione per lo spazio,

Infine, per quanto riguarda l'utilizzo della memoria durante l'esecuzione, entrambi i linguaggi mostrano requisiti simili, con Rust che, tendenzialmente, può richiedere una quantità leggermente superiore di RAM, dovuta principalmente all'utilizzo di astrazioni. Infatti, nonostante quest'ultime siano definite *zero-cost* in teoria, nella pratica, spesso, il compilatore non riesce a ottimizzarle pienamente e possono introdurre un minimo overhead.

4.2.4 Complessità del codice

Una critica comune nei confronti di Rust, specialmente da coloro che provengono da C, è la maggiore complessità e verbosità del linguaggio. Generalmente, la sintassi di C è considerata più semplice e minimale, principalmente per la sua natura a basso livello, la mancanza di astrazioni complesse e la presenza di un numero limitato di costrutti.

Al contrario in Rust, dovendo gestire attentamente le relazioni di *ownership*, *borrowing* e *lifetime*, il codice può risultare più complesso; questi non sono gli unici aspetti che contribuiscono a questa percezione: gestione degli errori, controllo del flusso, supporto a *generics* e *macro* sono tutti elementi che possono rendere il linguaggio più verboso e complesso rispetto a C.

Annotazioni di *lifetime*

Una delle principali differenze tra Rust e C è dovuta alla presenza del *modello di ownership*, che impone annotazioni di *lifetime* nei casi in cui non siano deducibili implicitamente. Come già accennato nel capitolo tre: 'Sistemi Operativi' 3, queste permettono al *Borrow Checker* di determinare la validità di un prestito nel tempo, rifiutando codice che viola le annotazioni.

Queste annotazioni introducono un livello ulteriore di complessità, sia sintattica che concettuale: il programmatore deve specificare quanto a lungo un riferimento sarà valido, dovendo ragionare in termini di durata dei dati nel tempo.

Tuttavia, la complessità viene compensata da una maggiore sicurezza del codice: come verrà mostrato successivamente, permette di prevenire problemi quali *access after free*.

che cerca di ridurre le dimensioni dell'eseguibile. Inoltre, in contesti di programmazione *bare-metal*, è possibile escludere la libreria standard (`#![no_std]`), in quanto non è presente un sistema operativo sottostante.

In C questo meccanismo non è presente, non esiste proprio il concetto di *lifetime* come elemento del linguaggio. In questi casi è il programmatore che deve garantire che i puntatori siano validi, senza supporto da parte del linguaggio.

Di seguito verranno riportati due esempi che mostrano come le annotazioni di *lifetime* possano prevenire *access after free*:

Listing 32: Prevenzione di *access after free* tramite *lifetime annotations*

```
fn biggest<'a>(a: &'a i32, b: &'a i32) -> &'a i32 {
    if *a > *b { a } else { b }
}
fn main (){
    let result;
    let smaller = Box::<i32>::new(1);
    {
        let bigger = Box::<i32>::new(2);
        result = biggest(&*smaller, &*bigger);
    }
    println!("Il piu grande e: {}", *result);
}
```

Listing 33: *Access after free* dovuta alla mancanza di *lifetime* in C

```
#include <stdlib.h>
#include <stdio.h>
int* biggest(int* a, int* b) {
    return (*a > *b) ? a : b;
}
int main(void) {
    int* result;
    int* smaller = malloc(sizeof(int));
    *smaller = 1;
    {
        int* bigger = malloc(sizeof(int));
        *bigger = 2;
        result = biggest(smaller, bigger);
        free(bigger);
    }
    printf("Il piu grande e: %d\n", *result);
    free(smaller);
}
```

Entrambi i listati, 32 e 33, implementano la stessa logica: definiscono una funzione `biggest` che riceve in ingresso due riferimenti (o puntatori, nel caso di C) e restituisce quello che punta al valore maggiore. La funzione `main` alloca due interi, `smaller` e `bigger`, nella heap e passa un riferimento a ciascuno a `biggest`. Per come sono inizializzati, la funzione restituisce un riferimento a `bigger`. Tuttavia, `bigger` viene deallocato prima della stampa, lasciando un *dangling pointer*.

Qua è possibile osservare la differenza tra i due linguaggi:

- In C, il compilatore produce un eseguibile senza generare alcun avviso o errore. L'esecuzione sembra funzionare, ma in realtà viene generato un *access after free*, leggendo un valore corrotto, come illustrato nell'immagine 11;
- In Rust, invece, `biggest` specifica una sola *lifetime*, 'a, indicando che tutte le reference (le due in ingresso e quella in uscita) devono essere valide per la durata di tale *lifetime*. Il *Borrow Checker* rileva che `smaller` e `bigger`, essendo definiti in scope differenti, hanno *lifetime* diverse e impedisce la compilazione. È possibile osservare questo comportamento nell'immagine 12.

```
frank@francyy:~/Documents/thesis$ gcc no-lifetimes.c -o no-lifetimes-c
frank@francyy:~/Documents/thesis$ ./no-lifetimes-c
Il piu grande e: 1453304247
```

Figura 11: *Access after free* dovuto all'assenza di *lifetime* in C

```
frank@francyy:~/Documents/thesis$ rustc lifetimes.rs -o lifetimes-rs
error[E0597]: `*bigger` does not live long enough
  --> lifetimes.rs:9:37
   |
8  |         let bigger = Box::<i32>::new(2);
   |         ----- binding `bigger` declared here
9  |         result = biggest(&*smaller, &*bigger);
   |                               ^^^^^^^^^ borrowed value does not live long enough
10 |     }
```

Figura 12: Tentativo di *access after free* in Rust

Questo è un esempio che mostra come Rust, tramite le annotazioni di *lifetime*, presenti una sintassi più complessa rispetto a C, ma allo stesso tempo più sicura.

Generics

Le espressioni *generics* o *generic programming* indicano un meccanismo di astrazione che consente di sviluppare codice indipendente da un tipo di dato specifico. Il tipo viene fornito come argomento, generalmente in fase di compilazione, consentendo il riuso del codice con diversi tipi di dato.

Il linguaggio C non supporta nativamente lo sviluppo di codice parametrico, in quanto comporterebbe un livello di astrazione superiore rispetto alla semplicità a cui il linguaggio aspira. Nonostante ciò, è possibile emulare un comportamento simile, principalmente tramite l'uso di puntatori `void*`, i quali possono riferirsi a valori di qualunque tipo. Questo meccanismo, per quanto flessibile, richiede controlli rigorosi da parte del programmatore:

- un puntatore `void*` non memorizza informazioni sulla dimensione del valore puntato, rendendo necessaria la conoscenza esplicita del tipo (solitamente memorizzandone la dimensione);
- le operazioni di accesso, sia in lettura che scrittura, devono essere effettuate tramite funzioni delicate come `memcpy`, in quanto non è possibile dereferenziare direttamente un puntatore `void*`.

Il listato 34 riporta un'estratto da un'implementazione in C di un nodo di una lista generica. L'implementazione completa è disponibile sulla piattaforma Github[16]²³. Nell'implementazione di riferimento sono state adottate alcune strategie per cercare di prevenire errori legati alla gestione manuale dei tipi e della memoria:

- **Tipo opaco:** la struttura dati è definita solo nel file sorgente, mentre l'header contiene solo una dichiarazione incompleta (*opaque type*). Questo impedisce l'accesso diretto ai campi, tra cui `element_size` di `clinkedlist`, la cui integrità è fondamentale per il corretto funzionamento della struttura;
- **Incapsulamento parziale:** la manipolazione della struttura è possibile solo tramite le funzioni fornite, obbligando l'utente a seguire un percorso logico e controllato per l'inserimento, la rimozione e la lettura dei dati. Questo permette l'introduzione di controlli interni riguardanti la validità e la coerenza dei dati.

²³ Disponibile nel repository, [C-data-structures](#), personale del candidato, seguendo il percorso: `src/linear/`. Alcuni commenti sono stati rimossi o spostati per ragioni di spazio.

Listing 34: Programmazione generica in C

```

typedef struct clinkedlist {
    node* tail;
    size_t element_count; /* Number of elements present in the list */
    size_t element_size; /* Size of the elements stored in the list */
} clinkedlist;

typedef struct node {
    void* ptr; /* Pointer to the memory location that stores the
        node's value */
    node* next; /* Pointer to the next node */
} node;

node* node_create(void* value, size_t value_size) {

    node* n = NULL;
    if (value_size <= SIZE_MAX) {

        n = (node*)malloc(sizeof(node));
        if (n) {

            n->ptr = malloc(value_size);
            if (n->ptr) {

                memcpy(n->ptr, value, value_size);
                n->next = NULL;
            }
            else {
                free(n);
                n = NULL;
            }
        }
    }
    return n;
}

```

Listing 35: Programmazione generica in Rust

```

struct Clinkedlist<T> {
    tail: Option<Box<Node<T>>>,
    element_count: usize
}

struct Node<T> {
    value: T,
    next: Option<Box<Node<T>>>
}

impl<T> Node<T> {
    fn new(value: T) -> Node<T> {
        Node {
            value,
            next: None
        }
    }
}

```

Nonostante queste precauzioni, il modello rimane fragile, per via della natura del linguaggio: l'utilizzo di `void*` permette la memorizzazione di qualsiasi valore, tra cui anche, altri puntatori; tuttavia, una eccessiva stratificazione di indirizzione (i.e. puntatori a puntatori) può compromettere la corretta deallocazione della memoria.

Rust adotta un approccio diverso, supportando la programmazione generica tramite le cosiddette *zero-cost abstractions*²⁴. Durante la fase di compilazione, il compilatore applica un processo detto *monomorfizzazione*, trasformando ogni utilizzo di una funzione o una struttura generica in una versione specifica per il tipo utilizzato.

Per ottenere un confronto sulla complessità del codice, viene riportato, nel listato 35, l'implementazione in Rust del listato 34: il supporto alla programmazione generica di Rust permette di definire strutture dati e funzioni generiche con codice sicuro e conciso.

L'implementazione risulta più semplice rispetto alla controparte C: non

²⁴ Si tratta di meccanismi e implementazioni astratti che non introducono overhead in fase di esecuzione rispetto al codice esplicito (concreto) equivalente.

è necessario gestire manualmente la dimensione dei valori o gli accessi diretti alla memoria. A tal proposito, si consideri che `Node::<T>::new()` del listato 35 è l'equivalente di `node_create` del listato 34.

Gestione degli errori

I due linguaggi si distinguono per i meccanismi impiegati per la gestione degli errori. In C, tipicamente, tale gestione è opzionale: il linguaggio non impone alcun controllo esplicito, lasciando la responsabilità al programmatore. Il meccanismo maggiormente diffuso per segnalare errori è basato sul valore di ritorno, tipicamente interi, da parte delle funzioni, secondo convenzioni informali e non uniformi²⁵.

L'assenza di vincoli espliciti sul controllo dell'esito rende facile, e comune, la mancata verifica di errori. Questo tuttavia può portare a comportamenti indefiniti, principalmente dovuti all'elaborazione di dati non validi, derivanti da chiamate a funzioni fallite, il cui risultato viene interpretato come se fosse corretto.

Come riportato dalla documentazione ufficiale[7], Rust distingue tra due classi di errori, *recuperabili* e *irrecuperabili*, fornendo meccanismi distinti per la loro gestione.

Gli errori *irrecuperabili* rappresentano situazioni dalle quali non è sicuro proseguire con l'esecuzione del programma, come l'accesso oltre i limiti di un array o il tentativo di dereferenziare un valore `None`²⁶. Rust gestisce questo tipo di errore tramite *panic*, i quali causano l'immediata interruzione del programma e rilasciano correttamente tutte le risorse allocate dal processo.

Al contrario, gli errori *recuperabili* rappresentano situazioni meno gravi, per le quali è possibile definire una logica di gestione, senza necessità di interruzione: per esempio, il tentativo di apertura di un file inesistente potrebbe essere gestito creando il file, invece di terminare l'esecuzione. Per la gestione di questo tipo di errore, Rust offre l'enumerazione `Result<T, E>`, in cui T ed E sono tipi generici:

25 Lo standard C non definisce una convenzione uniforme sugli interi da utilizzare come codice di errore. Alcune funzioni utilizzano 0 per indicare successo e ogni altro valore viene interpretato come fallimento. Per questo motivo, è sempre opportuno controllare la documentazione di una funzione, per determinare come interpretare l'esito: alcune funzioni potrebbero interpretare ogni valore non negativo con successo, come altre potrebbero utilizzare 0 per fallimento.

26 Rust utilizza l'enumerazione `Option<T>` per rappresentare un valore opzionale: `Some(T)` rappresenta la presenza di un valore, di tipo T, mentre `None` rappresenta l'assenza di un valore valido.

- **Ok(T)**: rappresenta la variante di `Result` che indica successo, contenente il valore, di tipo `T`, restituito;
- **Err(E)**: è la variante di `Result` che indica un fallimento, contenente un valore descrittivo dell'errore, di tipo `E`, generato.

A differenza del C, la gestione esplicita degli errori è obbligatoria in Rust: gli errori irrecuperabili vengono gestiti tramite `panic`; per quelli recuperabili, il compilatore impone al programmatore di gestire esplicitamente entrambe le varianti (`Ok` ed `Err`) tramite costrutti come `match` e `if let` oppure operatori come `?`, il quale propaga l'errore. Questo obbligo riduce la probabilità di errori non gestiti, rendendo il codice più sicuro e robusto rispetto all'approccio tradizionale di C.

Macro

I due linguaggi gestiscono le *macro* in maniera completamente differente. Come riportato dalla documentazione GCC[6], le *macro* in C sono frammenti di codice a cui viene associato un nome, dichiarate con la direttiva `#define`.

Durante la fase di pre-processamento (prima della compilazione), quando viene incontrato il nome di una *macro*, esso viene sostituito con il codice associato. Si tratta di una semplice sostituzione testuale, come mostrato nel seguente esempio:

Listing 36: Definizione di *macro* in C

```
#define printline(x) printf("%s\n", x)
int main(void) {
    printline("Stampa di prova");
    return 0;
}
```

Nel listato 36, viene definita la *macro* `printline`, che stampa la stringa `x`, andando a capo successivamente. All'interno della funzione `main` viene invocata `'printline("Stampa di prova")'`: questa sarà espansa in `'printf("%s\n", "Stampa di prova")'` dal pre-processore.

In quanto la sostituzione avviene durante la fase di pre-processamento, per osservare il codice generato è necessario utilizzare il flag `-E` durante la compilazione con GCC²⁷, come mostrato nell'immagine 13.

²⁷ Tramite la opzione `-E` viene indicato a GCC di fermarsi dopo la fase di pre-processamento, senza effettuare la compilazione. L'output della fase di pre-processamento verrà poi indirizzato allo standard output.

```

frank@francyy:~/Documents/thesis$ gcc -E macro.c
# 0 "macro.c"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 0 "<command-line>" 2
# 1 "macro.c"

int main(void) {
    printf("%s\n", "Stampa di prova");
    return 0;
}

```

Figura 13: Traduzione di una *macro* in C

La natura semplice delle macro C comporta delle limitazioni e falle di sicurezza, molte delle quali sono riportate sulla documentazione GCC[6]. A dimostrazione di ciò, si consideri la seguente modifica del listato 36:

Listing 37: Utilizzo scorretto di *macro* in C

```

#include <stdio.h>
#define printline(x) printf("%s\n", x)
int main(void) {
    int my_val = 150;
    printline(my_val);
    return 0;
}

```

Nel listato 37 si tenta di sfruttare la stessa macro, `printline`, per stampare il contenuto di un intero. È interessante osservare che, nonostante il mismatch dei tipi (`%s` indica di interpretare il parametro come stringa), il compilatore genera soltanto un warning, producendo lo stesso un file eseguibile. Tuttavia, tentando l'esecuzione si verifica un crash immediato dovuto a *segmentation fault*, come è osservabile dall'immagine 14.

In Rust le *macro* si distinguono in due classi: dichiarative e procedurali. Le *macro* procedurali sono molto complesse e richiederebbero un capitolo a parte, ma andrebbe oltre l'ambito di questa trattazione.

Le *macro* dichiarative, pur essendo meno complesse rispetto a quelle procedurali, sono caratterizzate comunque da una sintassi particolare, basata su *match di pattern*, permettendo perfino l'*overloading* in base al *pattern* come riportato dalla documentazione ufficiale[7]. Anche queste richiederebbero un capitolo dedicato per una spiegazione sufficientemente dettagliata.

```

frank@francyy:~/Documents/thesis$ gcc macro.c -o macro-c
macro.c: In function 'main':
macro.c:2:29: warning: format '%s' expects argument of type 'char *', but argument 2 has type 'int' [-Wformat=]
  2 | #define printline(x) printf("%s\n", x)
    |                               ^~~~~~
macro.c:5:5: note: in expansion of macro 'printline'
  5 |     printline(t);
    |     ^~~~~~
macro.c:2:31: note: format string is defined here
  2 | #define printline(x) printf("%s\n", x)
    |                               ^
    |                               |
    |                               char *
    |                               %d
frank@francyy:~/Documents/thesis$ ./macro-c
Segmentation fault (core dumped)

```

Figura 14: Limitazioni delle *macro* C

Per mostrare la maggiore complessità, e differente gestione, delle *macro* Rust rispetto alla controparte C, è sufficiente un esempio rappresentativo²⁸:

Listing 38: Definizione di *macro* dichiarativa in Rust

```

macro_rules! comp_eval {
    ($sx:expr; and $dx:expr) => {
        println!(
            "{:?} and {:?} is {:?}",
            stringify!($sx), stringify!($dx), $sx && $dx
        )
    };
    ($sx:expr; or $dx:expr) => {
        println!(
            "{:?} or {:?} is {:?}",
            stringify!($sx), stringify!($dx), $sx || $dx
        )
    };
    ($not:expr) => {
        println!(
            "not {:?} is {:?}",
            stringify!($not), !($not)
        )
    };
}

fn main() {
    comp_eval!(1 + 2 == 3; and true);
}

```

²⁸ L'esempio fornito è basato su quello riportato nella documentazione ufficiale[7], nella pagina relativa all'overloading di macro.

```

    comp_eval!(true; or false);
    comp_eval!(true);
}

```

Nel listato 38 è definita la *macro* dichiarativa `comp_eval` che, in base al pattern, espande un codice diverso:

- Se viene fornita una coppia di espressioni, `$sx` e `$dx`, separate dalla parola chiave `and`, la *macro* espande una stampa della loro congiunzione logica;
- Se le espressioni sono separate dalla parola chiave `or`, espande una stampa della loro disgiunzione logica;
- Se invece viene fornita una singola espressione, `$not`, la *macro* espande una stampa della sua negazione logica.

Il risultato dell'esecuzione di questo codice è osservabile nell'immagine 15.

Le *macro* Rust rappresentano uno strumento molto potente rispetto alle direttive pre-processore di C: permettono di definire comportamenti differenti in base alla struttura dell'invocazione, cosa non possibile con la semplice sostituzione testuale.

Un'altra differenza rispetto alle *macro* C riguarda la loro espansione: le *macro* in Rust vengono espansate a livello di compilazione e in caso di mancata corrispondenza con qualsiasi *pattern* previsto viene generato un errore di compilazione.

Ad esempio, aggiungendo l'invocazione `'comp_eval!("test")'` all'interno della funzione `main` nel listato 38, si otterrà un errore durante la compilazione in quanto il pattern fornito non corrisponde a nessuna delle regole definite nella *macro*²⁹. Il messaggio di errore è osservabile nell'immagine 16.

Conclusioni

Nonostante siano limitati, gli aspetti esaminati mettono in luce un tratto distintivo di Rust, specialmente rispetto a C: il linguaggio richiede uno sforzo iniziale maggiore, dovuto alla maggiore complessità della sintassi,

²⁹ In questo esempio il pattern fornito è una singola stringa, quindi la regola adeguata sarebbe la terza, che accetta un'unica espressione. Tuttavia, l'operatore di negazione logica non può essere applicato a una stringa, di conseguenza il compilatore rifiuta il codice, generando un errore.


```

frank@francyy:~/Documents/thesis$ rustc macro.rs -o macro-rs
frank@francyy:~/Documents/thesis$ ./macro-rs
"1 + 2 == 3" and "true" is true
"true" or "false" is true
not "true" is false

```

Figura 15: Compilazione di una *macro* in Rust

```

frank@francyy:~/Documents/thesis$ rustc macro.rs -o macro-rs
error[E0600]: cannot apply unary operator `!` to type `&'static str`
--> macro.rs:17:31
   |
17 |         stringify!($not), !($not)
   |                             ^^^^^^^ cannot apply unary operator `!`
...
25 |     comp_eval!("test");
   |     ----- in this macro invocation

```

Figura 16: Errore durante la compilazione di una *macro* in Rust

ma questo viene ripagato garantendo meccanismi di sicurezza della memoria affidabili e integrati nel linguaggio stesso.

PROGETTI E APPLICAZIONI REALI

In questo capitolo verranno presentati progetti concreti che dimostrano come Rust possa costituire una scelta valida non solo in termini teorici, ma anche nella pratica per lo sviluppo di basso livello quali driver, kernel e addirittura sistemi operativi completi.

I progetti analizzati sono in costante sviluppo ed evoluzione; le informazioni riportate in questa trattazione sono aggiornate ad Agosto 2025, ma potrebbero subire variazioni nel tempo.

Tra le applicazioni più rilevanti sono presenti: l'integrazione di Rust nel kernel di Windows 11, il progetto *Rust for Linux* (RfL) con i relativi sottoprogetti, l'implementazione di *sudo-rs* promossa da Ubuntu e il sistema operativo *Redox OS*.

5.1 KERNEL DI WINDOWS 11

L'adozione di Rust da parte di Microsoft risale al 2023, quando l'azienda ha iniziato a riprogettare sezioni critiche del kernel del sistema operativo Windows 11, come riportato dal seguente articolo[9].

Prima di analizzare quest'adozione, è utile un breve excursus storico sull'evoluzione del kernel Windows, principalmente per comprendere l'importanza di questo cambiamento.

Lo sviluppo dei sistemi Microsoft iniziò con un kernel interamente scritto in assembly (ASM8086) nelle prime versioni di *MS-DOS*. Con *MS-DOS 3.0* venne introdotto il linguaggio C, ma il primo vero kernel scritto interamente in C si ebbe con Windows NT 3.1 (**KERNEL.EXE**, 1993). Successivamente, C è rimasto il linguaggio principale per il kernel di Windows, con integrazioni C++ nelle versioni successive.

Il passaggio a Rust in Windows 11 è stato motivato da una combinazione di esigenze aziendali e del crescente interesse per il linguaggio Rust, che iniziò a essere popolare proprio in questo periodo, grazie al modello offerto: garantire la sicurezza della memoria senza sacrificare le prestazioni.

In particolare, Rust ha permesso a Microsoft di affrontare due aspetti fondamentali:

- **Sicurezza della memoria:** Windows ha una storia documentata alle spalle di errori legati alla memoria, quali *null reference*, *buffer overflow*, scritture illegali e altri. Il *modello di ownership* offerto da Rust consente di prevenire queste problematiche già a tempo di compilazione;
- **Gestione della concorrenza:** Data la complessità Windows, anche gli errori di concorrenza sono frequenti. Il *Borrow Checker* di Rust, insieme all'uso di *smart pointers* per la condivisione dei dati, garantisce integrità e un uso corretto delle risorse condivise, eliminando errori quali *data race*.

Va sottolineato che, ad oggi, si tratta di un'adozione parziale: il kernel di Windows 11 è tuttora scritto prevalentemente in C, con alcune porzioni in C++. L'obiettivo dichiarato di Microsoft è intervenire sulle sezioni chiave del kernel e sulle nuove funzionalità, valutando caso per caso l'impiego di Rust.

Nonostante si tratti di un'adozione parziale, questa scelta rappresenta un traguardo significativo per Rust: guadagnare la fiducia di un colosso tecnologico come Microsoft ha contribuito alla crescita della popolarità del linguaggio, specialmente nel panorama della programmazione di sistema.

5.2 RUST FOR LINUX

Il progetto *Rust for Linux* nasce con l'obiettivo di integrare il supporto a Rust come linguaggio di programmazione utilizzabile all'interno del kernel Linux, dimostrandone l'idoneità nello sviluppo di componenti di basso livello, principalmente driver, tradizionalmente implementati in C.

L'interesse verso Rust deriva principalmente dalle garanzie offerte dal linguaggio sotto gli aspetti di *memory-safety* e *thread-safety* senza costi di esecuzione aggiuntivi. Grazie al *modello di ownership* intere classi di errori vengono completamente prevenute a livello di compilazione, come

accessi a memoria non inizializzata, memoria non liberata, *double free* a altri ancora, comuni nei sistemi sviluppati in C¹.

Il progetto raccoglie un insieme di contributi al kernel, in gran parte rappresentato da driver per dispositivi fisici (schede di rete, NVMe) e virtuali (GPU virtuali), sviluppati interamente in Rust.

La documentazione ufficiale è consultabile dal sito di Rust for Linux [4]. Inoltre, è opportuno osservare che il progetto è in continua evoluzione, le informazioni riportate in questa trattazione sono aggiornate ad Agosto 2025, ma potrebbero variare nel futuro.

Per comprendere l'importanza di questo progetto, è utile un breve excursus sull'evoluzione dei linguaggi supportati dal kernel linux.

Il kernel Linux venne rilasciato inizialmente nel 1991 da Linus Torvalds ed era quasi interamente scritto in C, con alcune sezioni in assembly. Questa situazione rimase invariata fino al 2007, quando venne l'integrazione di C++ nel kernel.

Tuttavia, l'idea venne rifiutata da Torvalds stesso, contrario all'utilizzo di un linguaggio considerato meno trasparente e più complesso rispetto a C, come riportato dal seguente articolo di *Zeus News* [8], dove vengono esposte alcune motivazioni specifiche a riguardo.

Solo nel 2022, con la versione 6.1, venne aggiunto un primo supporto sperimentale a Rust, sufficiente per permettere agli sviluppatori di iniziare a scrivere codice Rust nel kernel e testarne l'integrazione.

Al giorno d'oggi il kernel Linux consente lo sviluppo di moduli e driver, sia in C che in Rust, definiti *out-of-tree*: moduli sviluppati e mantenuti separatamente dal codice sorgente del kernel, spesso proprietari o specifici per hardware particolare.

Il progetto *Rust for Linux* si concentra soprattutto sullo sviluppo di moduli *in-tree*, inclusi direttamente nella *mainline* del kernel (i.e. incluse nel sorgente del kernel). Tuttavia, il progetto raccoglie anche moduli *out-of-tree*, permettendo agli sviluppatori un ambiente sicuro per sperimentare e validare soluzioni prima di proporle per un'inclusione ufficiale nella *mainline*.

Un concetto chiave introdotto dal progetto è quello di *reference driver* (dri-

¹ Il modello di *ownership* viene esposto nel capitolo tre 3, mentre nel capitolo quattro 4.2.2 è possibile osservare quali errori vengono effettivamente eliminati grazie ad esso.

ver di riferimento), ovvero implementazioni in Rust che possono essere integrate nei sottosistemi senza sostituire i driver C esistenti.

Questi driver hanno diverse funzioni:

- Definire astrazioni sicure per i nuovi driver, evitando la riscrittura o duplicato codice esistente;
- Fornire un modello di riferimento per gli sviluppatori C, mostrando come un driver equivalente possa essere realizzato in Rust;
- Sfruttare le infrastrutture già presenti *in-tree* per preparare i sottosistemi a un'integrazione graduale e progressiva di Rust;
- Facilitare l'apprendimento graduale del linguaggio da parte degli sviluppatori del kernel;
- Valutare la convenienza dell'adozione: verificare quanta parte del codice possa essere scritta in modalità *safe*, quanti bug del driver originale C verrebbero effettivamente eliminati e quale sia l'impatto sulla manutenzione.

In molti casi, un *reference driver* può essere semplicemente un prototipo o un banco di prova, più che un driver destinato all'uso in produzione.

5.2.1 Moduli in mainline

Come precedentemente accennato, i moduli e driver *in mainline* fanno ufficialmente parte del kernel: sono inclusi nel sorgente e compilati insieme ad esso, risultando disponibili senza necessità di installazioni aggiuntive. La categoria *in mainline* di *Rust for Linux* comprende principalmente driver per schede di rete (*Network Interface Cards*, **AMCC QT2025** e **ASIX**) e per GPU (**Nova** e **Tyr**), oltre ad alcuni moduli di utilità.

DRIVER PER NIC Due dei principali moduli *in-tree* del progetto sono driver per dispositivi di rete, **AMCC QT2025 PHY** e **ASIX PHY**.

Il primo è un driver per l'omonimo dispositivo (transceiver **AMCC QT2025**) e fornisce un'interfaccia verso lo stack di rete del kernel, facilitando l'interazione tra il dispositivo e il sistema operativo e la gestione delle funzionalità di rete. È stato integrato nel kernel nella versione 6.12.

Il secondo è destinato ai dispositivi Ethernet del produttore **ASIX**, ma ha finalità principalmente dimostrative: è sviluppato come *reference driver*, per fornire un esempio di implementazione di un driver PHY (layer fisico dello stack ISO/OSI) in Rust. È stato introdotto nella versione 6.8.

DRIVER PER GPU Tra i moduli *in-tree* rientrano anche due driver per schede grafiche: **Nova GPU** e **Tyr GPU**.

Nova GPU è un driver per le schede grafiche NVIDIA a partire dalla serie RTX 2000, concepito come successore dell'attuale *Nouveau*. La descrizione dettagliata del driver è riportata nella sezione '*Red Hat: Nova*' 5.3.

Tyr GPU è un driver *Direct Rendering Manager (DRM)* per le GPU *Arm Mali* basate su *CSF (Command Stream Frontend)*, sviluppato come porting in Rust dell'attuale driver *Panthor* (in C). Il progetto ha un team di sviluppo che comprende ingegneri provenienti da *Collabora*, *Arm* e *Google*, e mira a fornire la stessa API per spazio utente attualmente offerta da *Panthor*, così da poter sostituire direttamente il driver nel contesto di *PanVK* (driver Vulkan).

Lo sviluppo di *Tyr* procede su due rami distinti:

- **Upstream:** Attualmente in grado di rilevare GPU su SoC (*System-on-Chip*) RK3588, leggere alcune sezioni della ROM della GPU e trasferirle allo spazio utente tramite chiamate API;
- **Downstream:** Utilizzato come *reference driver* per testare le nuove astrazioni proposte, prima della loro integrazione nella *upstream*. Attualmente è in grado di inviare piccoli pacchetti di lavoro alla GPU.

Il progetto *Tyr* è iniziato a giugno 2025 e si trova ancora in fase iniziale, instabile e fortemente sperimentale. Secondo la documentazione ufficiale (Sezione *Users – in mainline/Tyr GPU Driver*) [4], non gestisce ancora il controllo della corrente e implementa funzionalità limitate di recupero dagli errori, ma il team di sviluppo prevede di estenderle nei prossimi mesi.

DRIVER PER NULL BLOCK Il *Null Block device (/dev/null)* è un dispositivo a blocchi virtuale utilizzato principalmente per test e benchmarking: scarta tutti i dati scritti e restituisce EOF se si legge da esso, senza usufruire di memoria o spazio di archiviazione fisico.

Il driver attuale, `null_blk`, è scritto interamente in C e ha una nota storia di vulnerabilità legate alla gestione della memoria: un'analisi dei commit relativi al driver mostra che circa il 41% dei fix sono dovuti a errori di sicurezza della memoria[13]. Ciò lo ha reso un ottimo candidato per un'implementazione in Rust.

A questo scopo, è stato sviluppato `rnull`, un driver scritto interamente in safe Rust, con porzioni minime di codice `unsafe`, incapsulate in astrazioni sicure per interagire con le API C del kernel.

Attualmente `rnull` replica gran parte delle funzionalità di `null_blk`, ma non è ancora completo e manca di alcune feature presenti nell'implementazione originale.

GENERATORE DI CODICI QR PER DRM PANIC Questo modulo è destinato ai sottosistemi *Direct Rendering Manager (DRM)* e ha lo scopo di semplificare l'analisi degli *stack trace* generati in seguito a un *kernel panic*.

Il problema di base è che i messaggi di errore del *DRM* possono essere molto lunghi e poco pratici da copiare manualmente. Il modulo genera un codice QR, scansionabile con uno smartphone, contenente le informazioni sull'errore, così da ottenere rapidamente i dettagli necessari all'analisi.

L'implementazione non richiede memoria o spazio di archiviazione aggiuntivi: sfrutta lo spazio libero nel buffer già riservato al processo per memorizzare il codice QR. È stato integrato nel kernel nella versione 6.12.

5.2.2 Moduli *outside mainline*

Oltre a moduli inclusi nella *mainline*, il progetto raccoglie anche sviluppi *outside mainline*, ovvero non integrati direttamente nel kernel: pur essendo compatibili con una determinata versione del kernel, vengono mantenuti e distribuiti separatamente, richiedendo quindi installazioni aggiuntive prima di essere utilizzabili.

In questa categoria rientrano driver per dispositivi di storage (**NVMe**), GPU proprietarie (**Apple AGX**), filesystem (**PuzzleFS**) e moduli per Android.

Il progetto *Rust for Linux* privilegia lo sviluppo di moduli *in-tree*, ma ciò non rende meno rilevanti quelli *out-of-tree*: spesso si tratta di soluzioni pensate per contesti o hardware specifici; per questo, solitamente, la documentazione disponibile può risultare limitata.

ANDROID ASHMEM Ashmem (*Anonymous Shared Memory Subsystem for Android*) è un allocatore di memoria condivisa per Android, attualmente simile a **POSIX SHM** ma con un'API più semplice e basata su file.

È progettato per liberare automaticamente le regioni di memoria con-

divisa quando il sistema è sotto pressione (ovvero quando la memoria fisica si sta saturando), caratteristica che lo rende particolarmente adatto a dispositivi con risorse limitate.

DRIVER PER ANDROID BINDER Questo progetto mira a riscrivere in Rust il driver kernel per il **Binder** di Android. Il *Binder* è un componente fondamentale per la sicurezza e le prestazioni dei sistemi Android: gestisce la *IPC* (*Inter-Process Communication*) all'interno del *sandbox*² di Android, permettendo la comunicazione tra applicazioni isolate.

La sua natura critica lo rende particolarmente vulnerabile a errori legati alla gestione della memoria, per cui trarrebbe significativi vantaggi dalle garanzie offerte dal *modello di ownership* di Rust, sia intermini di sicurezza che di prestazioni.

DRIVER PER APPLE AGX Questo driver è destinato alla GPU **AGX** di Apple ed è accompagnato da binding *DRM* (*Direct Rendering Manager*) per lo spazio utente. Oltre a far parte di *Rust for Linux*, il driver rientra anche nel progetto *Asahi Linux*, il quale mira a portare supporto Linux sulle CPU Apple Silicon.

L'interesse principale verso AGX deriva proprio dal contesto di *Asahi*: la documentazione tecnica più dettagliata è presente nelle pagine ufficiali di tale progetto [3].

Attualmente lo sviluppo è focalizzato sull'implementazione di driver per *OpenGL* e *Vulkan* e sul *reverse engineering* del set di istruzioni supportato dalla GPU.

DRIVER PER NVME Questo driver rappresenta il tentativo di sviluppare un driver **NVMe PCI** interamente in safe Rust, concepito principalmente come *reference driver*. L'obiettivo è dimostrare la fattibilità di astrazioni sicure per dispositivi ad alte prestazioni, oltre che a fornire un esempio concreto per sviluppi futuri.

Allo stato attuale si trova in fase sperimentale e instabile, non adatta a un uso in produzione.

DRIVER PER FILESYSTEM PUZZLEFS **PuzzleFS** è un filesystem per container progettato per superare alcune limitazioni dell'attuale stack *OCI* (*Open Container Initiative*). Il progetto mira a ridurre la duplicazione

² Android isola l'esecuzione delle applicazioni in ambienti detti *Sandbox*, per cui ogni applicazione ha il proprio ambiente privato per la sua esecuzione, separato dalle altre.

dei dati, garantire build riproducibili, supportare il *mounting* diretto e garantire la sicurezza della memoria.

Il driver, scritto in Rust, implementa queste caratteristiche attraverso:

- **Riduzione della duplicazione:** utilizzo dell'algoritmo *FastCDC* per condividere segmenti di memoria tra i vari layer;
- **Build riproducibili:** definizione di una rappresentazione canonica del formato delle immagini;
- **Sicurezza della memoria:** ottenuta implicitamente grazie al *modello di ownership* di Rust.

5.2.3 Impatto del progetto

Tra i recenti esempi di integrazione di Rust in progetti esistenti, *Rust for Linux* è senza dubbio quello che ha avuto il maggiore impatto tecnico e mediatico, generando anche un ampio dibattito all'interno della community *open source*.

Il motivo è semplice ma fondamentale: il progetto coinvolge il kernel Linux, cuore di vari sistemi operativi alla base di gran parte dell'infrastruttura digitale odierna, sviluppato e mantenuto da una comunità vasta e con una cultura tecnica consolidata.

A confronto con:

- **Kernel di Windows 11:** Windows è un sistema proprietario sviluppato internamente da Microsoft, quindi l'integrazione di Rust non genera lo stesso dibattito pubblico né ha lo stesso impatto sulla community *open source*;
- **sudo-rs:** riscrittura in Rust del comando *sudo*, importante per la sicurezza, ma limitato a un singolo strumento, non all'intero sistema;
- **Redox OS:** sistema operativo scritto interamente in Rust, ma con adozione limitata (è considerato principalmente un progetto '*di nicchia*') rispetto a Linux, il quale è alla base di Android e della maggior parte dei server.

MOTIVAZIONI DELL'IMPORTANZA L'integrazione di Rust nel kernel Linux rappresenta il primo tentativo concreto di introdurre un linguaggio

con garanzie di sicurezza della memoria nella *mainline* del kernel. L'obiettivo è ridurre la classe di vulnerabilità legate a errori di gestione della memoria, mantenendo le prestazioni richieste da un sistema operativo.

L'importanza è amplificata dal fatto che Linux è alla base di una porzione enorme dell'ecosistema tecnologico globale: dai dispositivi mobili Android ai vari server che alimentano servizi cloud e web. Qualsiasi cambiamento strutturale al kernel ha quindi effetti su larga scala.

Infine, si tratta di un cambiamento che rompe una tradizione radicata: dal 1991 il kernel Linux è scritto quasi interamente in C, e in passato altri linguaggi (incluso C++) non sono stati accettati. L'introduzione di Rust non è solo un aggiornamento tecnico, ma anche una modifica della filosofia di sviluppo del progetto.

CONSEGUENZE SOCIO-CULTURALI L'ultima proposta per introdurre un nuovo linguaggio nel kernel risale nel 2007, quando Torvalds rifiutò l'adozione di C++. Questa storia rende l'accettazione, seppure parziale, di Rust un segnale di apertura.

In una prima occasione, Linus Torvalds ha espresso un approccio pragmatico verso Rust, riconoscendo sia le resistenze culturali che i possibili benefici tecnici:

'Rust is a very different thing, and there are a lot of people who are used to the C model. They don't like the differences, but that's OK [...] Clearly, some people just don't like the notion of Rust and having Rust encroach on their area. But we've only been doing Rust for a couple of years, so it's way too early to say Rust is a failure'.

– *Linus Torvalds, discussione alla 'Linux Kernel Mailing List', 2022*

In un'altra occasione, ha chiarito di non considerare Rust un rimpiazzo totale di C, ma uno strumento aggiuntivo, utile in casi specifici:

'I do not think Rust will take over the kernel, and I don't think anybody is even suggesting that. But I do think Rust can be a good tool for some things, and we should use the best tool for the job'.

– *Linus Torvalds, conferenza 'Linux in the Multiverse', 2024*

Queste dichiarazioni evidenziano due aspetti fondamentali: da un lato, l'esistenza di una divisione culturale tra sviluppatori più legati al modello C e sostenitori di Rust; dall'altro, la volontà di valutare Rust su basi pratiche e a lungo termine, senza pregiudizi definitivi e senza trasformarlo in una questione ideologica.

5.3 RED HAT: NOVA

Nova è un progetto sviluppato dall'azienda Red Hat, parte integrante dell'iniziativa *Rust for Linux*. Si tratta di un driver per *GSP*, un componente presente nelle schede grafiche NVIDIA di nuova generazione, dalla serie RTX 2000 in poi.

Il *GSP* è un componente hardware e firmware integrato nella GPU che consente di gestire quest'ultima come se fosse un sistema embedded. Tra le sue funzioni principali si trovano: la gestione dell'alimentazione, la gestione del clock, l'inizializzazione dell'hardware, lo scheduling delle code e il controllo termico. Permette di comunicare con la GPU come se fosse un'entità autonoma interna al sistema, sollevando la CPU da tutte le funzionalità precedentemente elencate.

Prima dell'introduzione del *GSP*, il driver open source *Nouveau*, sviluppato principalmente tramite *reverse engineering*³, soffriva di limitazioni in termini di prestazioni e stabilità rispetto ai driver proprietari forniti da NVIDIA.

Con il *GSP*, molte delle funzionalità ricostruite tramite *reverse engineering* sono ora gestite direttamente da questo processore dedicato, il quale funge da strato di astrazione che permette la comunicazione tra GPU e kernel tramite *IPC*. Questo cambiamento strutturale ha reso necessario lo sviluppo di un nuovo driver specifico.

I motivi che hanno portato Red Hat a scegliere Rust per lo sviluppo di *Nova* sono fondamentalmente tre:

- **memory-safety:** il motivo principale risiede nella garanzia di sicurezza della memoria offerta da Rust, così da prevenire in partenza la maggior parte di bug;
- **thread-safety:** le GPU sono composte da un numero elevato di *thread*, per cui il driver trae vantaggio dalla gestione sicura della concorrenza di Rust;
- **sperimentazione:** trattandosi di un nuovo driver e non di una riscrittura, vi era l'occasione per sperimentare e testare il linguaggio.

Nova è strutturato in due componenti principali:

³ NVIDIA è nota per non rilasciare pubblicamente la documentazione delle proprie GPU, principalmente per motivi di marketing, per poter offrire il proprio driver proprietario, maggiormente ottimizzato rispetto alle alternative open source.

- **nova-core**: esegue le operazioni a basso livello, come l'avvio del *GSP* e l'interazione, tramite quest'ultimo, con l'hawdware;
- **nova-DRM**: fornisce un'interfaccia astratta conforme al *DRM* (*Direct Rendering Manager*), fondamentale per la comunicazione con lo spazio utente.

Questa architettura modulare permette di combinare *nova-core* con differenti driver grafici, oltre a *nova-DRM*. Ad esempio, è possibile impiegare *VFIO* per assegnare la GPU a macchine virtuali, sfruttando la possibilità, supportata a livello firmware, di creare più *vGPU* (GPU virtuali).

Attualmente, *nova-DRM* è sviluppato principalmente come driver grafico per ambienti virtualizzati, ma può essere utilizzato anche su sistemi fisici, sebbene non sia ottimizzato per questo ambito.

5.4 UBUNTU: SUDO-RS

Sebbene non faccia parte del kernel, il comando `sudo` è uno degli strumenti più importanti e diffusi nel mondo Linux. La sua funzione è fondamentale: *Substitute User DO* consente l'esecuzione di comandi con i privilegi di un'altro utente, solitamente più elevati, come quelli di un utente ristretto o di super-utente. Proprio per questo, è uno degli strumenti più delicati dal punto di vista della sicurezza di un sistema.

Dalla sua prima introduzione fino a oggi, `sudo` è stato sviluppato nel linguaggio C. Come riportato in [12], tuttavia, Canonical ha annunciato che, a partire dalla versione 25.10 di Ubuntu, verrà introdotta una nuova implementazione del comando, scritta interamente in Rust, `sudo-rs`.

La scelta di Rust deriva dalla necessità di garantire una maggiore sicurezza della memoria e ridurre le vulnerabilità tipiche delle implementazioni in C, come accessi a memoria non valida. Il *modello di ownership* di Rust rappresenta una scelta perfetta per questo scopo, eliminando intere classi di errori pur mantenendo prestazioni comparabili a quelle di C.

Il progetto, guidato dalla *Trifecta Tech Foundation*, punta a migliorare la sicurezza di uno degli strumenti più sensibili dei sistemi Linux. Alla sua realizzazione contribuisce anche Todd Miller, storico mantenitore del comando negli ultimi trent'anni, fornendo supporto tecnico e linee guida al team di sviluppo del progetto.

Le principali vulnerabilità della versione attuale riguardano semplici errori di gestione della memoria, in particolare *use-after-free* e accessi oltre i limiti. Queste problematiche possono essere sfruttate per eseguire

attacchi di tipo *privilege escalation*⁴.

Gli obiettivi del progetto `sudo-rs` includono:

- Migliorare la gestione dell'escaping dei caratteri speciali nella shell, evitando l'esecuzione di comandi indesiderati e potenzialmente malevoli;
- Integrare il controllo sui profili di AppArmor;
- Supportare l'utilizzo di `sudoedit`;
- Garantire retrocompatibilità con kernel precedenti alla versione 5.9, come Ubuntu 20.04 LTS.

La filosofia del gruppo di sviluppo segue il principio *'less is more'*: evitare un eccesso di funzionalità iniziali per ridurre la complessità e il rischio di introdurre errori logici da risolvere in seguito.

Per questo motivo, `sudo-rs` non mira a essere un rimpiazzo totale di `sudo`: molte feature, specialmente quelle meno utilizzate o considerate meno importanti potrebbero venire non implementate inizialmente.

Il nuovo comando è già disponibile per gli utenti che desiderano testarlo⁵ e fornire feedback alla comunità.

Come nota finale, Canonical garantisce che la versione originale in C del comando continuerà a essere mantenuta e distribuita, così da lasciare all'utente la libertà di scegliere quale versione adottare, in base alle proprie preferenze o esigenze.

5.5 REDOX OS

Redox OS rappresenta una pietra miliare per il linguaggio Rust, in particolar modo nell'ambito della programmazione di sistema. Redox, come riportato dal sito ufficiale [5], è un sistema operativo general purpose Unix-like basato su microkernel; molti sistemi operativi rientrano in questa categoria (come Minix o BlackBerry QNX), ma ciò che distingue Redox risiede nell'implementazione, interamente in Rust.

⁴ La *privilege escalation* rappresenta uno scenario in cui un utente con permessi limitati (e.g. *guest*) ottiene, sfruttando le vulnerabilità di un sistema, privilegi più elevati (e.g. *root*).

⁵ È possibile installare il comando tramite il package manager `apt`, cercando `rust-sudo-rs`; successivamente, sarà disponibile come `'sudo-rs'`.

La scelta di usare solamente Rust ha l'obiettivo di mettere alla luce le capacità pratiche del linguaggio nello sviluppo sia di (micro)kernel che di programmi general purpose, fornendo un'alternativa completa a Linux o BSD.

Il progetto Redox ha riscontrato successo grazie a una combinazione di design basato su microkernel, impiego di Rust, compatibilità con POSIX e con la maggior parte dei programmi Linux/BSD grazie alla propria libreria C (anch'essa sviluppata in Rust).

DESIGN BASATO SU MICROKERNEL Un microkernel è un insieme di software minimale che offre i meccanismi necessari per implementare un sistema operativo: il microkernel da solo non è sufficiente, è necessario integrare moduli aggiuntivi per ricoprire le funzionalità di un sistema operativo.

Questo permette il caricamento, la modifica e la rimozione di moduli a runtime, senza la necessità di riavviare il sistema come conseguenza di ogni cambiamento. Inoltre, in quanto i moduli sono esterni al microkernel, essi vivono nello spazio utente, garantendo isolamento dei bug: anche se un intero modulo andasse in crash, non avrebbe conseguenze sul resto del sistema⁶ (il kernel non risentirebbe del crash).

SCRITTURA IN RUST Redox trae beneficio dai vari aspetti caratteristici del linguaggio, nonché da molte delle garanzie offerte:

- Il *modello di ownership* impone regole che prevengono la maggior parte di errori legati alla memoria, come *null reference*, *use-after-free*, *unfreed memory* e *double free*, nonché *data race*;
- Il *Borrow Checker* vieta la condivisione non sicura di risorse, imponendo vincoli rigidi per garantire l'integrità dei dati condivisi da più thread o processi;
- L'utilizzo di astrazioni sicure come `Option<T>` e `Result<T,E>` obbliga il programmatore a gestire esplicitamente sia il successo sia l'eventuale fallimento di un'operazione, evitando accessi a dati non validi;

⁶ Questo rappresenta uno dei maggiori vantaggi dell'approccio microkernel. Non vi è il rischio che l'intero sistema si corrompa in seguito al crash di una singola applicazione o modulo.

- Tutti questi meccanismi, uniti alla sintassi restrittiva del linguaggio, eliminano classi intere di bug, lasciando solo quelli legati alla logica applicativa.

Sia il microkernel che i vari driver sono implementati interamente in Rust, sfruttando le caratteristiche sopra citate e senza richiedere un ulteriore sforzo per garantire la *thread-safety* o la *memory-safety*⁷.

COMPATIBILITÀ CON POSIX Redox mira a essere compatibile con la maggior parte delle applicazioni Linux e, più in generale, a rispettare lo standard POSIX. Tale compatibilità non è di tipo binario, ma è ottenuta a livello di codice sorgente: in molti casi è sufficiente ricompilare l'applicazione per renderla eseguibile su Redox.

Questo risultato è reso possibile da *relibc*, la libreria C di Redox, scritta interamente in Rust per mantenere coerenza con la filosofia del progetto. Grazie a *relibc*, Redox supporta già numerose applicazioni fondamentali, tra cui GNU bash, Git, Ffmpeg, GCC e LLVM, rendendolo potenzialmente utilizzabile anche da un utente finale.

CONSIDERAZIONI SU REDOX È necessario osservare che Redox, nonostante sia nato come sistema operativo general purpose, è ancora in fase di sviluppo, non ancora maturo per un utilizzo quotidiano, specialmente se paragonato a colossi del calibro di Windows, MacOS o Ubuntu, i quali forniscono una buona esperienza utente già *'out of the box'*.

Ciononostante, Redox costituisce una dimostrazione concreta della maturità e dell'usabilità di Rust per lo sviluppo di sistemi operativi completi, confermandone l'applicabilità ben oltre la teoria.

5.6 CONSIDERAZIONI SUI PROGETTI ANALIZZATI

In questo capitolo, sono stati esaminati diversi progetti concreti che mostrano come Rust possa essere integrato nei sistemi operativi, sia tramite l'aggiunta di moduli specifici che tramite la riscrittura completa di componenti esistenti.

Un aspetto comune di tutti gli esempi presentati è la volontà di affrontare uno dei principali punti deboli dell'uso del C: la gestione della memoria. Il *modello di ownership* di Rust, come discusso nel capitolo tre:

⁷ È richiesto uno sforzo maggiore solo durante la fase iniziale dello sviluppo, ma una volta raggiunta la compilazione, si ha la garanzia che la memoria e la concorrenza sono gestite senza incoerenze.

‘Gestione della memoria in Rust’ 3 e mostrato nel capitolo quattro: ‘Sistemi Operativi’ 4, riesce a eliminare già a tempo di compilazione molti degli errori legati alla gestione della memoria, i quali in C emergerebbero solo a tempo di esecuzione.

Le strategie adottate, tuttavia, sono diverse: alcuni progetti optano per un’interazione graduale (come il kernel di Windows 11 e *Rust for Linux*), mentre altri per una sostituzione totale (come *sudo-rs*). Allo stesso modo, in alcuni casi l’attenzione è posta interamente sulla sicurezza del sistema (Windows 11 e *Rust for Linux*), mentre in altri viene data importanza all’esperienza dell’utente finale (*Redox OS*).

Nel complesso, queste iniziative mostrano come Rust non sia più solamente un linguaggio *promettente* e sperimentale, ma uno strumento concreto, impiegabile in contesti reali e complessi, con risultati effettivi nella direzione di maggiore sicurezza e affidabilità dei sistemi.

CONCLUSIONE

6.1 RIASSUNTO

Il secondo capitolo, ‘*Rust*’ 2, ha ripercorso le origini del linguaggio e le motivazioni alla base del suo sviluppo, evidenziando come si sia evoluto in breve tempo, da progetto personale e sperimentale, in uno strumento adottato da un ampio numero di sviluppatori e aziende.

Nel terzo capitolo, ‘*Gestione della memoria in Rust*’ 3, sono stati analizzati a fondo i meccanismi di *ownership*, *borrowing* e *lifetime*, che costituiscono il fulcro del modello di gestione della memoria del linguaggio. Questi strumenti hanno mostrato come Rust riesce a garantire simultaneamente sicurezza e controllo, aspetti fondamentali per lo sviluppo di basso livello.

Il quarto capitolo, ‘*Sistemi Operativi*’ 4, ha preso in analisi il linguaggio C come riferimento storico e pratico, ricostruendo le caratteristiche che lo hanno reso centrale nella programmazione di sistema: *compilazione*, *assenza di runtime*, *manipolazione della memoria* e dei *bit*. Rust è stato successivamente confrontato con C sia negli aspetti precedentemente elencati, sia in aspetti trasversali, quali *gestione delle risorse*, *sicurezza della memoria*, *complessità della sintassi* e *prestazioni*.

Ne è emerso un quadro in cui Rust rappresenta una valida alternativa, nella teoria, a C, a costo di una curva di apprendimento più ripida rispetto a quest’ultimo.

Infine, il quinto capitolo, ‘*Progetti e applicazioni reali*’ 5, ha illustrato progetti concreti già in corso che sperimentano l’uso del linguaggio nello sviluppo di sistemi operativi e componenti critici. Queste iniziative mostrano come il linguaggio non sia più una premessa puramente teorica, ma uno strumento concreto già in grado di produrre risultati significativi.

Nel complesso, l'analisi ha messo alla luce le potenzialità e i limiti di Rust nello sviluppo di basso livello, aspetti che verranno approfonditi nella prossima sezione, '*Considerazioni critiche e personali*' [6.2](#).

6.2 CONSIDERAZIONI CRITICHE E PERSONALI

Rust appare in grado di portare benefici significativi nello sviluppo di applicazioni di basso livello e, in particolare, di sistemi operativi. Grazie al *modello di ownership*, gli errori di gestione della memoria dinamica vengono di fatto eliminati a tempo di compilazione; inoltre le astrazioni *zero-cost* e l'impiego di *smart pointer* consentono una gestione sicura della concorrenza, prevenendo interamente le *data race*.

Tuttavia, Rust non rappresenta una soluzione magica né priva di limitazioni. La sintassi e il *Borrow Checker* richiedono tempo e dedizione per essere padroneggiati, definendo una curva di apprendimento molto più ripida rispetto a linguaggi più permissivi come Python o anche C. A questo va aggiunto il fatto che Rust si limita a prevenire gli errori di gestione della memoria, ma non quelli logici: la correttezza di un algoritmo o di un'implementazione rimane ancora pienamente responsabilità del programmatore.

È quindi opportuno, o comunque consigliato, evitare un impiego indiscriminato del linguaggio. Da un lato, utilizzare Rust in contesti dove i suoi vantaggi non si traducono in benefici concreti (e.g. un'applicazione che non lavora con una quantità significativa di memoria dinamica) rischia di introdurre soltanto complessità; dall'altro, utilizzare Rust come se fosse un'altro linguaggio, aggirando o utilizzando in maniera scorretta i costrutti disponibili (per esempio scrivendo ampie porzioni di codice *unsafe*) vanifica gran parte dei principi alla base del linguaggio stesso.

La prospettiva più equilibrata è dunque quella di impiegare Rust nei contesti critici, in cui sia le prestazioni che la sicurezza della memoria sono fondamentali e in cui i vantaggi offerti si traducono in maniera concreta, evitando un uso eccessivo o improprio che ne comprometterebbe i punti di forza.

BIBLIOGRAFIA

- [1] Dark Bears. Why c continues to be the preferred systems programming language, 2018.
- [2] Mohamed Amin Bouali. Bit manipulation in c, 2023. Capacità del linguaggio C rispetto alle operazioni bitwise.
- [3] Asahi contributors. Asahi linux - linux on apple silicon, 2020. Usato semplicemente come riferimento per i lettori interessati ai dettagli del driver per GPU AGX.
- [4] Community contributors. Rust for linux, 2025. Raccolte informazioni sui progetti sia dentro che fuori la mainline.
- [5] Redox Developers. Redox os, 2025. Utilizzata principalmente la sezione FAQ per ricavare informazioni sul progetto.
- [6] GNU foundation. Gcc, the gnu compiler collection - online documentation, 2025. Informazioni sulle macro C.
- [7] Steve Klabnik and Caron Nichols. *The Rust Programming Language*. No Starch Press, San Francisco, 2nd edition, 2023. Sezioni di riferimento: Ownership (4.1), Borrowing (4.2), Lifetime (10.3), Unsafe Rust (20.1).
- [8] Dario Meoli. Linus torvalds odia il c++. e' un linguaggio orribile, parola di linus., 2007. L'articolo esplora a fondo alcuni motivi per cui Torvalds non apprezza C++, nella trattazione è stato citato in maniera molto riassuntiva.
- [9] Gary Olsen. Does windows kernel use rust and what does it mean for it?, 2025. Integrazione di Rust nel kernel di Windows 11 e l'evoluzione del kernel Windows.
- [10] Rust Embedded Working Group. The embedded rust book, n.d.
- [11] Ada Computer Science. Bitwise manipulation, n.d. Esempi sull'importanza delle operazioni bitwise.
- [12] Joey Sneddon. Ubuntu 25.10 switches to rust-based sudo, 2025.

- [13] Git Summary. History log dei commit relativi null_blk, 2020. Usato come riferimento per mostrare la mole di errori relativi al blocco nullo.
- [14] The Computer Language Benchmarks Game. The computer language benchmarks game, 2025. Raccolti estratti di benchmark: (n-body e fasta). Data di accesso: 2025-07-18.
- [15] Clive Thompson. How rust went from a side project to the world's most-loved programming language, 2023. MIT Technology Review.
- [16] Francesco 'whocaresleft' Biribò. C-data-structures, 2025.