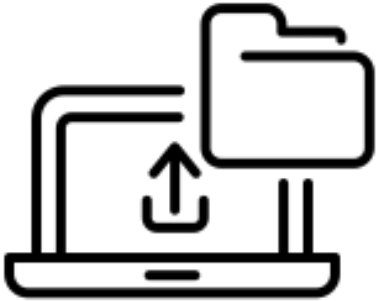


File I/O

Gestione della persistenza

File e file system

- Un file consiste in un'astrazione offerta dal sistema operativo che lega un blocco di byte di dimensione arbitraria ad un nome
  - I nomi sono organizzati in una struttura gerarchica fatta di cartelle o directory che permette di identificare uno specifico file attraverso una concatenazione dei nomi di cartelle e nome del file che ne esprime il cammino (*path*) a partire da una radice nota
- Ad ogni file sono associati vincoli di sicurezza
  - Il sistema operativo garantisce che solo chi dispone delle necessarie autorizzazione possa leggere, scrivere o eseguire il file
- Le librerie dei linguaggi di programmazione offrono meccanismi indipendenti dal sistema operativo per accedere al contenuto di un file
  - Nel caso di Rust, l'astrazione principale è offerta dalla struct **`std::fs::File`**, che modella un file aperto in lettura e/o scrittura.
  - Il C++ dalla versione 17 ha std::filesystem (in versione antecedenti, si possono usare le versioni experimental o boost)

Percorsi

- Ogni sistema operativo ha regole proprie per la definizione di cosa sia un percorso lecito per indicare un file, quali siano le radici note dei percorsi, come combinare segmenti parziali in un percorso complessivo, …
  - Le struct **`std::path::Path`** e **`std::path::PathBuf`** nascondono tali differenze offrendo un meccanismo portabile per comporre e scomporre un cammino e ricavare indicazioni sul file eventualmente referenziato
  - **`Path`**, analogamente a **`str`**, è ***`unsized`*** e accessibile in sola lettura
  - **`PathBuf`**, analogamente a **`String`**, possiede il proprio contenuto e può essere modificato
- Attraverso i metodi offerti da questi tipi, è possibile ricavare informazioni sulla esistenza del file, sulla sua natura (file semplice, cartella, collegamento simbolico, …), sui metadati associati (dimensione, data di creazione e di ultima modifica, permessi, …)

Navigare il file system

- La funzione **`std::fs::read_dir(dir: &Path) -> Result<ReadDir>`** restituisce, se ha successo, un iteratore al contenuto della cartella dir
  - Le singole voci ritornate sono di tipo **`std::fs::DirEntry`** e descrivono gli elementi contenuti nella cartella in termini di nome, tipo (file, cartella, collegamento simbolico), metadati e cammino
- La funzione **`std::fs::create_dir(dir: &Path) -> Result<()>`** crea una nuova cartella
  - Fallisce se non si dispone delle necessarie autorizzazioni, se la cartella esiste già o se la cartella genitrice del cammino indicato non esiste
- La funzione **`std::fs::remove_dir(dir: &Path) -> Result<()>`** rimuove una cartella
  - A condizione che esista, si disponga dei necessari permessi e che sia vuota

Manipolare i file nel file system

- La funzione `std::fs::copy(from: &Path, to: &Path) -> Result<i64>` copia il contenuto di un file in un secondo file
  - Restituisce in caso di successo il numero di byte copiati
- La funzione `std::fs::rename(from: &Path, to: &Path) -> Result<()>` rinomina (sposta) un file in un secondo file
  - Sostituendo il contenuto del file destinazione con quello sorgente
  - Il comportamento di questa funzione dipende dal sistema operativo
- La funzione `std::fs::remove_file(path: &Path) -> Result<()>` elimina un file
  - Se il file è in uso, la sua eliminazione può essere rimandata dal sistema operativo

Operazioni con i file

- L’accesso al blocco di byte legato ad un file è totalmente mediato dal sistema operativo
  - Per poter leggere o scrivere tale blocco occorre “aprire” il file
  - Il sistema operativo offre apposite funzioni che restituiscono un riferimento opaco al file sotto forma di *handle* o *file descriptor* (di fatto un numero intero)
- La struct **`File`** offre due metodi di base per aprire un file
  - **`open(path: P) -> Result<File> where P: AsRef<Path>`** - apre il file in lettura, a condizione che esista
  - **`create(path: P) -> Result<File> where P: AsRef<Path>`** - tronca il file a 0 byte, se esiste, o lo crea, se non esiste ancora, dopodiché lo apre in scrittura
- Maggiori opportunità sono offerte dalla struct **`std::fs::OpenOption`**
  - Essa permette di impostare come un file debba essere aperto e quali operazioni sono consentite su di esso

Leggere e scrivere file

- Le funzioni **`std::fs::read_to_string(path: &Path)`** e **`std::fs::write(path: &Path, contents: &[u8])`** offrono un meccanismo compatto per leggere e scrivere il contenuto di un file di moderate dimensioni
  - Poiché un file può avere dimensioni molto maggiori della massimo blocco di memoria allocabile, occorre utilizzare tali funzioni quando si è certi che il contenuto può essere ospitato nella memoria del processo

```
use std::fs;
let contents = fs::read_to_string(filename)
   .expect("Something went wrong reading the file");
println!("Text is:\n{}", contents);
```

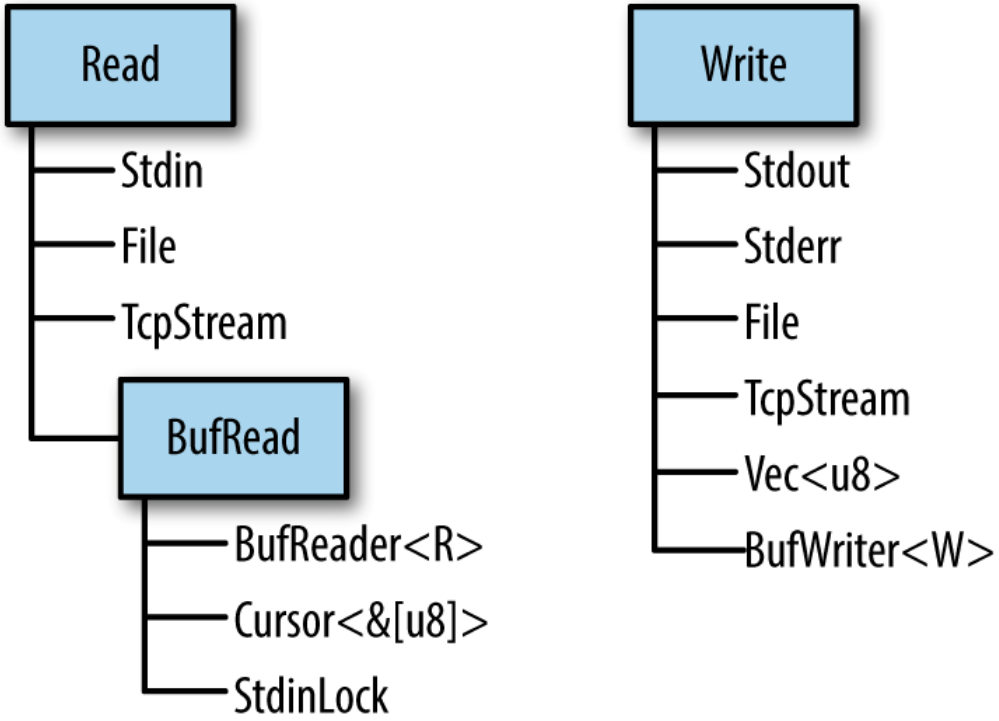
Aprire un file

```
use std::fs::File;
use std::io::{Write, BufReader, BufRead, Error};

let path = "lines.txt";

let mut output = File::create(path)?;
write!(output, "Rust\n💖\nFun")?;

let input = File::open(path)?;
let buffered = BufReader::new(input);

for line in buffered.lines() {
    println!("{}", line?);
}
```

I tratti relativi a I/O

- Rust gestisce le operazioni di I/O attraverso l'utilizzo di alcuni tratti che implementano i metodi di base per le attività di lettura e scrittura
  - L'utilizzo di tratti favorisce la scrittura di codice generico, indipendente dal tipo specifico su cui viene eseguito
- I tratti principali offerti da rust sono: **`Read`**, **`BufRead`**, **`Write`** e **`Seek`**
  - Tutti e quattro i tratti possono essere importati in maniera concisa attraverso il costrutto `use std::io::prelude::*;`
- In caso di errore durante le operazioni di I/O viene ritornata una delle varianti disponibili nell' enum **`ErrorKind`**
  - La variante **`ErrorKind::Interrupted`** indica un errore non fatale, che generalmente può essere gestito riprovando l'operazione

I tratti relativi ad I/O



std::io::Read

- Tratto che indica la capacità di leggere un flusso di byte
  - **`File`**, **`Stdin`** e **`TcpStream`** sono alcuni dei tipi che implementano questo tratto
- Per implementare il tratto **`Read`** è sufficiente fornire l'implementazione del metodo **`read(buf: &mut [u8]) -> Result<usize>`**
  - Rust genererà tutti gli altri metodi sulla base dell'implementazione di **`read(buf: &mut [u8])`** fornita dal programmatore
- In caso di successo, il metodo **`read(...)`** deve ritornare **`Ok(n)`**
  - L'implementazione deve garantire che **`n`** sia compreso tra 0 e **`buf.len()`**
  - Il valore **`Ok(0)`** può indicare che il flusso è terminato oppure che il buffer passato ha lunghezza 0
- Ogni chiamata al metodo **`read(...)`** può causare l'invocazione di una chiamata di sistema
  - Con il conseguente costo legato al cambio di contesto

Metodi del tratto Read

- L'implementazione del tratto **Read** mette a disposizione diversi metodi per gestire le operazioni più comuni di I/O
  - `read_to_end(buf: &mut Vec<u8>) -> Result<usize>` continua a leggere fino all' EOF: si limita a richiamare il metodo `read()` fino a quando quest'ultimo non ritorna un `Ok(0)` o un errore fatale
  - `read_to_string(buf: &mut String) -> Result<usize>` continua a leggere fino all' EOF e riceve come parametro una `&mut String`
  - `read_exact(buf: &mut [u8]) -> Result<()>` prova a leggere l'esatto numero di byte necessario a riempire completamente `buf`, se non riesce ritorna `ErrorKind::UnexpectedEof`
  - `bytes() -> Bytes<Self>` ritorna un iteratore sui bytes, gli elementi dell'iteratore sono dei `Result<u8, io::Error>`
  - `chain<R: Read>(next: R) -> Chain<Self, R>` permette di concatenare due reader
  - `take(limit: u64) -> Take<Self>` limita il numero massimo di byte che sarà possibile leggere

std::io::BufRead

- Il tratto **`BufRead`** offre una serie di metodi che permettono di migliorare le prestazioni dell'I/O appoggiandosi ad un buffer in memoria
  - Ogni chiamata a **`read()`** può dare origine ad una system call, l'utilizzo di un buffer permette di effettuare meno chiamate
  - Risulta particolarmente efficace se si eseguono molte letture di piccole dimensioni
  - Non è utile quando si legge da elementi già presenti in memoria
- L'implementazione del tratto richiede i metodi **`fill_buf()`** e **`consume(amt: usize)`**
  - I due metodi devono sempre essere utilizzati insieme: **`fill_buf()`** si limita a ritornare il contenuto del buffer in memoria, successivamente è necessario chiamare il metodo **`consume(...)`** per garantire che i byte non siano ritornati nuovamente
- Offre i metodi **`read_line(&mut self, buf: &mut String)`** e **`lines(self)`** per accedere al contenuto testuale di un flusso

std::io::BufRead

```
use std::io;
use std::io::prelude::*;

let stdin = io::stdin();                                                    // stdin non
implementa il tratto BufRead
let mut handle = stdin.lock();                                          // lock permette di ottenere
uno StdinLock

          // che implementa BufRead

let buffer = handle.fill_buf().unwrap();

println!("{:?}", buffer);

let length = buffer.len();

stdin.consume(length);                                                      // garantisce che i
byte letti non vengano
          // ritornati nuovamente alle prossime letture
```

std::io::Write

- Tratto che indica la capacità di scrivere un flusso di dati
  - Questo tratto è implementato, tra gli altri, dalle struct **`File`**, **`Stdout`**, **`StdErr`** e **`TcpStream`**
- Il tratto **`Write`** richiede l'implementazione dei metodi **`write`** e **`flush`**
  - **`write(buf: &[u8]) -> Result<usize>`** prova a scrivere l'intero contenuto del buffer ricevuto come argomento e ritorna il numero di byte scritti
  - **`flush() -> Result<()>`** finalizza l'output garantendo che tutti gli eventuali buffer transitori siano correttamente svuotati
- Il metodo **`write_all(buf: &[u8])`** si limita a chiamare ricorsivamente il metodo **`write`** fino a quando i dati sono stati tutti scritti o viene restituito un errore fatale

std::io::Seek

- Tratto che permette di ri-posizionare il cursore di lettura/scrittura in un flusso di byte
  - Quando il flusso attinge ad un dato di dimensione nota, è possibile posizionare il cursore in modo relativo rispetto all'inizio del flusso (`SeekFrom::Start(n: u64)`), alla sua fine (`SeekFrom::End(n: i64)`) o alla posizione corrente (`SeekFrom::Current(n: i64)`)
- Il tratto offre i seguenti metodi
  - `fn seek(&mut self, pos: SeekFrom) -> Result<u64>`: posiziona il cursore alla posizione (in byte) indicata dal parametro pos
  - `fn rewind(&mut self) -> Result<()>`: posiziona il cursore all'inizio del flusso
  - `fn stream_position(&mut self) -> Result<u64>`: restituisce la posizione corrente del cursore rispetto all'inizio del flusso

Esempio: lettura di un file contenente dati binari

```
use std::fs::File;
use std::io::Read;

fn main() -> std::io::Result<()> {
    // "/dev/urandom": sorgente di byte casuali provenienti da una sorgente sicura
    let mut f = File::open("/dev/urandom")?;

    loop {
        let mut buff = [0;4];                          // buffer in cui depositare i byte
        let r = f.read_exact(&mut buff);               // lettura del file
        if r.is_err() { return r; }
        if buff.iter().any(|b| *b==0) { return Ok(()); } // se trovo un byte nullo, termino
        let i = i32::from_be_bytes(buff);              // conversione del buffer in i32
        println!("{:x}",i);                            // uso il valore letto
    }
}
```

Lettura e scrittura di contenuti strutturati

- Sebbene sia possibile operare a livello di singoli byte/caratteri e ricostruire un dato strutturato a partire dai suoi componenti elementari, spesso è poco conveniente farlo
  - Rust mette a disposizione il framework **`Serde`** il cui scopo è generare implementazioni efficienti di funzioni di serializzazione e deserializzazione di strutture dati arbitrarie verso formati standard quali **`JSON`**, **`CSV`**, **`Avro`**, **`BSON`**, **`YML`** e altri
  - Tale framework definisce i tratti **`serde::Serialize`** e **`serde::Deserialize`** che possono essere implementati dai tipi definite all'interno di un programma
- Serde offre un'implementazione predefinita per la serializzazione dei tipi elementari e di alcuni tipi della libreria standard
  - **`String`**, **`&str`**, **`Vec<T>`**, **`HashMap<K,V>`**, …
  - Supporta inoltra la macro **`#[derive(Serialize, Deserialize)]`** per generare, in fase di compilazione, l'implementazione dei tratti corrispondenti, a condizione che la struttura dati cui tale macro è applicata non contenga elementi "patologici"

Uso del framework Serde

- Si inseriscono, nel file cargo.toml, le dipendenze
  - `serde = { version = "1.0", features = ["derive"] }`
  - `serde_json = "1.0"`
- La seconda indica il tipo di formato da generare/leggere
  - In alternativa, è possibile indicare un altro sotto-progetto compatibile come `csv = "1.3"` o `bson = "2.9"`
- Si decora la struttura dati da leggere con la marco **#[derive(...)]**

```
#[derive(Serialize, Deserialize, Debug)]
struct Data {
    name: String,
    data: Vec<u8>,
    attributes: HashMap<String, String>,
}
```

Uso del framework Serde

```
fn save(data: &Data, path: &str) ->
                                Result<()> {
    let mut f = File::options()
        .write(true)
        .create(true)
        .truncate(true)
        .open(path)?;

    f.write(serde_json::to_string(data)?
         .as_bytes())?;

    Ok(())
}
```

```
fn load(path: &str) -> Result<Data> {
    let mut f = File::open(path)?;

    let mut s = String::new();

    f.read_to_string(&mut s)?;

    return Ok(serde_json::from_str(&s)?);

}
```

Per saperne di più

- Reading and Writing Files in Rust
  - https://rustjobs.dev/blog/reading-and-writing-files-in-rust/
  - Breve riassunto con esempi basilari per la lettura e la scrittura di file
- How to Read Files in Rust
  - https://dev.to/oliverjumpertz/how-to-read-files-in-rust-525d
  - Strategie a confronto per la lettura di file
- Reading Large Files and Perf
  - https://sensepost.com/blog/2023/reading-large-files-and-perf/
  - Analisi delle prestazioni dei diversi approcci nella lettura di file di grosse dimensioni
- Detail Guide to Serialization and deserialization with Serde in Rust
  - https://medium.com/@2018.itsuki/detail-guide-to-serialization-and-deserialization-with-serde-in-rust-4fa70a6a8c4b
  - Guida al framework Serde