# I04\_3 InputOutput

### **File**

Un file è un’astrazione che i sistemi operativi mettono a disposizione → **pacco di byte associato ad un nome** che:

* Posso accedere
* Posso manipolare

I file sono strutturati per poter contenere una quanità grande di informazione. **L’acceso** a file avviene in una modalità di **streaming**: leggiamo un pezzo alla volta e piano piano eleaboraiamo il contenuto.

I sistemi operativi permettono di **raggruppare i file in cartelle** e permettono di fare accesso alle varie cartelle

*(. → cartella corrente; .. → cartella genitrice).*

Ad ogni file sono associati dei **vincoli di sicurezza** che limitano l’accesso ai file ai soli utenti che ne hanno il diritto. Per accedere ad un file su Rust:

* **std∷fs∷File** : astrazione che modella il concetto di file su disco aperto in lettura/scrittura.

Per sapere i **file presenti in una cartella**:

* C → non mi aiuta
* C++ → da qualcosa
* Java → modella l’entità file con il suo nome e offre metodi per vedere se esiste un file, cancellarlo, creare una cartella
* **Rust** → **std∷fs** e introduce due classi sotto std∷path
  + **Path** → cammino all’interno del file system accessibile in sola lettura (in prestito)
    - Unsized
    - Sola lettura
  + **Path\_buf** → cammino di cui abbiamo il possesso e quindi possiamo modificare
    - Posseduto, modifcabile

Offrono **metodi funzionali** per navigare nelle cartelle

* Andare a cartella superiore
* Avere una serie di chunk in cui muoverci
* Metodi per controllare il contenuto
* Per capire natura del dile (file semplice, cartella, collegamento)

Offrono **metadati associati**

* Dimensione
* data creazione
* ultima modifica
* permessi

## RUST

**Read\_dir:**   
Restituisce un IO-Result di iteratore per accedere ai dati contenuti in una cartella

* 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

**Create\_dir:** Crea una nuova cartella se dispone delle autorizzazioni necessarie

* Se non ha autorizzazioni/ cartella già esistente/ gartella genitrice non esistente → Fallisce

**Remove\_dir**:

Elimina cartella. Per poter eliminare la cartella, questa deve essere vuota e bisogna possedere i diritti

**Copy**:

Permette di copiare un file in una cartella

* Se non riesce → error
* Se riesce→ informa su quanti byte ha copiato

**Rename**:

Sposta il file da una posizione ad un’altra

* Solitamente, se nello stesso fylesystem → eliminato fle in cartella partenza e aggiunto file in cartella destinazione
* Se lo si rinomina nella stessa cartella, si cambia la entry della chiamata da *vecchio* a *nuovo*
* Se si vuole passare da un volume ad un altro disgiunto (diverso fyilesystem) → bisogna effettura una copia e una cancellazione
  + Problematico
    - Potrebbe non esserci più spazio necessario
      * Operazione fallisce, originale invariato, non effettua copia
    - Molto lento

**Remove file**: rimuove il file

### Operazioni con i file

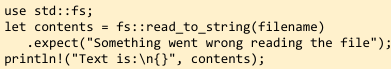
Struct **File** offre:

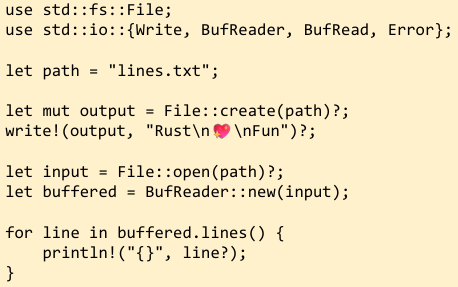
* **open**: apre file in lettura, se esiste
* **create**: assume che il file non ci sia e prova a crearne uno di dimensione 0
  + Se il file già esiste, svuta il file esistente e sovrascrivi

*NOTA: Esiste la struct* ***std∷fs::OpenOption*** *che offre ulteriori funzionalità:*

* *si può scegliere .open o .create per ottenere possibilità di modificare il file stesso*

Si possono avere **diverse esigenze in base alla dimensione del file**:

* File grande → magari voglio acedere a pezzi
* File **piccolo** (abbastanza per processo ~100MB)
  + posso prendere il file intero usando **read\_to\_string**
  + posso scrivere sul file usando la **write**

Nel caso in cui voglio agire sul file **poco alla volta**, posso popolare il mio file in vari modi.

* Scrivere con **write!**

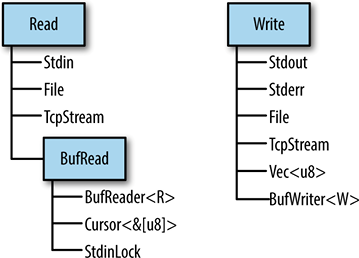
Per leggere a pezzi:

* **BufReader** → leggo il file riga per riga, offrendomi meccanismi per andarci all’interno
  + Iteratore

**Nota**: Tutte le operazioni di IO possono in ogni punto fallire  
**Nota**: in Rust non c’ bisogno di specificare la chiusura del file perché si applica il **RAII** → risorsa rilasciata in modo automatico quando fuori scope.  
**Nota**: se non si chiede il lock, sistema operativo ti lascia darlo in scrittura ad uno e in lettura ad un altro;  
 nonostante ciò, la macro write esegue 2 operazioni:

* Affida **un buffer al SO** dicendo di aggiungere al file
  + SO può decidere quando vuole scriverlo (es: ogni 4kB faccio)
* **Flash**: obbliga il S.O. a prendere quello che ha ricevuto e caricarlo sul disco subito
  + Senza questa garanzia, rischio che non siano effettuate velocemente.

### Tratti relativi ad IO -slide 9

Esistono alcuni tratti che facilitano l’operazione e permettono di trattare operazioni di IO, in maniera non diversa da operazioni normali:

* **Read**
  + **Stdin**: flusso di ingresso di un processo (es: tastiera→ cosa digita il processo)
    - Leggo un byte alla volta (Se siamo in due a leggere, rischio che io leggo CI e lal’tro legge AO e nessuno capisce nulla)
  + **File** : leggere un blocco binario, caratteri, stringhe da gfile
  + **TcpStream**: connessione di tipo TCP con un host
    - Rappresenta uno dei due end-Point
    - Connessione messa in piedi dalla quale posso leggere
    - Fa system call interpellando il S.O. ogni volts che c’è qualcosa
      * Costosa perché rompi le scatole al S.O.
      * Per diminuire il carico al S.O.
        + Dico di leggere 500 byte, mettere in un buffer e ne prendo un po’ alla volta senza rompere al S.O. ogni volta → bufRead
* **BufRead**
  + BufReader<R> → leggere da file in chunk grossi, dandoli poi poco alla volta
    - Offre iteratore lines
  + Cursor<&[u8]>
    - Oggetto che permette di muoversi all’interno di uno slice di byte
    - Usato quando ottengo accesso memroy mapped ad un file
  + StdinLock
    - Leggo prendendo accesso in modo esclusivo ai blocchi (evito la situazione di stdin)
* **Write**
  + Stdout: flusso di uscita standard di un processo
  + StdErr: flusso di errore di un file
  + File: mandare un file o un chunk di byte
  + TcpStream: oltrw a leggerci, posso anche scriverci
  + Vex<U8> : posso trattare un vettor edi byte come fosse un file
  + BufWriter: tengo in pancia i vari byte e quando ne ho abbastanza li scrivo
    - In mdood tale da non attraversare per ogni byte la barriera user - kernel
* **Seek**

Fanno parte del prelude → vanno importati importando il prelude

Restituiscono dei **Result** che:

* Positivi
  + Numero
  + Void per dire che non c’era nulla
* Errore di **ErrorKind**
  + Ingestibili
  + Interrupted → situazione transiztoria
    - Potrebbe valere la pena riprovarci dopo un po’

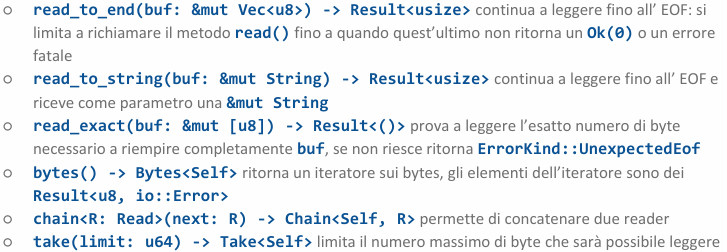
### **Tratto Read:**

**read**: mi da possibilità di leggere all’interno.

All’interno del tratto read, molti metodi con implementazione di default su tratto read.

Il metodo read fornisce un **risultato**

* **Positivo**: intero tra 0 e lunghezzaBuffer buf.len()
  + Nel caso in cui sia Ok(0) → raggiunto EndOfFile (Se file pieno) oppure: file vuoto
* **Negativo** → Errore

****Ogni volta che chiamo read → cambio contesto con system call → costosa → 500 cicli macchina

**read\_to\_end**→ legge dimensione, prende un buffer e inserisce dentro tutto; fare attenzione che ci sia abbastanza spazio  
**read\_to\_string** → legge e mette in stringa verificando che quei byte mettano su stringa utf8 ben formata

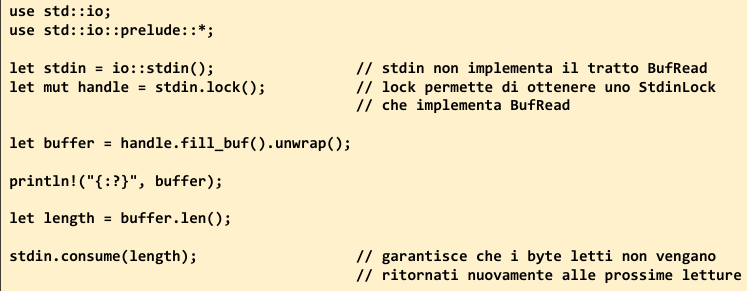
**read\_exact**→ prova a leggere esattamente il nr di byte che gli ho richiesto, se non riesce, errore

**bytes()** →restituisce un iteratore che permette di tornare i byte uno alla volta

**chain**()→ permette di attaccare un read ad un altro read, ad esempio se leggo un file dividendolo su 3 pezzi, poi faccio 2 .chain e li concateno, anche tra filesystem diversi  
**take**()→ fornisce un iteratore dino ad un max nr di byte

### **Tratto BufRead:**

Migliora le prestazioni dell’I/O appoggiandosi ad un buffer in memoria

**Si basa** internamente **su** due metodi principali:

* **fill\_buf()**: ritorna il contenuto dal buffer in memoria
* **consume(..)**: garantisce che i byte non siano tornati nuovamente

Offre i metodi:

* **read-Line** → fino al /n
* **lines** →

**Tratto write:**

Contiene due metodi:

* **write**: affida a sistema operativo un blocco
* **flush**: garantisce che il S.O. trasferisce tutto fino in fondo.
* Altri:
  + **write\_all**: prende uno slice e garantisce che lo scrive per intero
    - richiama ricorsivamente write finchè tutto scritto o errore fatale

Nota:

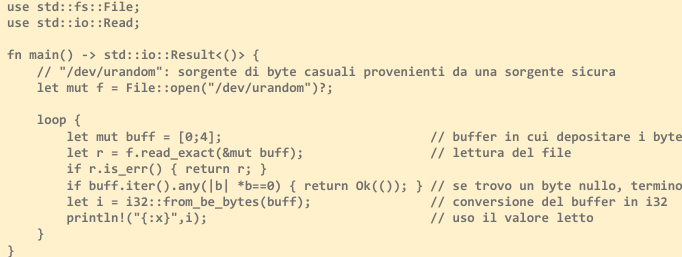
* di default: read, legge da inizio a fine
* write: scrivo partendo dalla fine;

### **Tratto seek:**

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamentetratto che permette di riposizionare il cursore di lettura/scrittura in un flusso di byte

Esempio di lettura di un file contenente i binari - ore 12:49



**Urandom** → finto file che quando lo leggo, genera byte a caso appoggiandosi ad una periferica che genera dei numeri che siano robustamente casuali.

* So che mi da 32 bit → prepari un buffer di 4 byte
* Chiamo read\_exact → riempi il contenuto con file urandom
* Verifico errore
* Se vuoto → Ok()
* Altrimenti converto in buffer e stampo

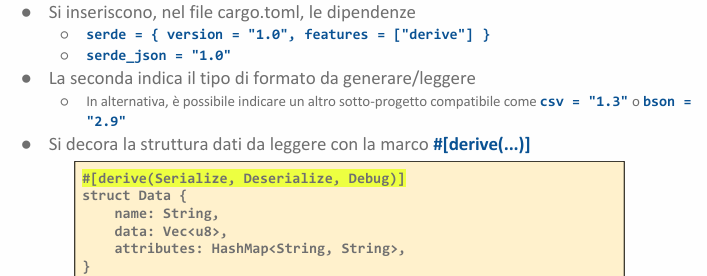
#### Lettura dati strutturati:

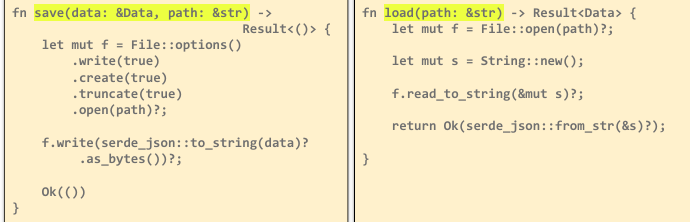
Possono esserci dati strutturati: JSON, CSV, etc

**Serde**:

* **Serialization**→ trasformo in formato interno di rust
* **Deserialization**→ trasformo in formato esterno

Possiamo includere con la macro Derive, la macro Serialize e Deserialize





**--lezione SerializeDeserialize**