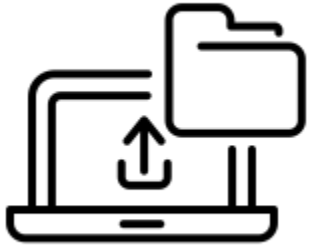


File I/O

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

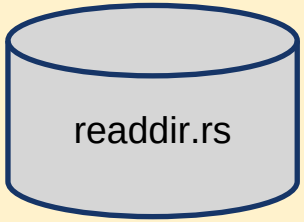
- Ogni sistema operativo ha regole proprie per la definizione di cosa sia un percorso lecito per indicare un file, quali siano le radici note dei percorsi, come combinare segmenti parziali in un percorso complessivo, …
  - Le struct **`std::path::Path`** e **`std::path::PathBuf`** nascondono tali differenze 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

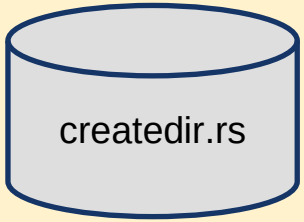
read_dir()

```rust
fn main() -> std::io::Result<()> {
    // Ottieni il percorso della directory
    let directory_path = ".";

    // Leggi il contenuto della directory
    let entries = fs::read_dir(directory_path)?;

    // Itera sugli elementi nella directory
    for entry in entries {
        // Gestisci eventuali errori nell'accesso ai file/directory
        let entry = entry?;

        // Ottieni il nome dell'elemento
        let file_name = entry.file_name();

        // Stampa il nome dell'elemento
        println!("{:?}", file_name);
    }

    Ok(())
}
```



create_dir()

```rust
use std::fs;

fn main() -> std::io::Result<()> {
    // Definisci il percorso della nuova directory da creare
    let new_directory_path = "./mynewdir";

    // Crea la nuova directory
    fs::create_dir(new_directory_path)?;

    println!("Directory creata con successo!");

    Ok(())
}
```



remove_dir()

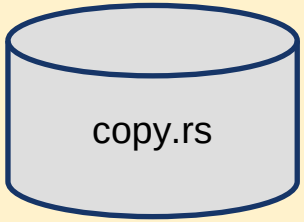
```
use std::fs;

fn main() -> std::io::Result<()> {
    // Definisci il percorso della directory da rimuovere
    let directory_to_remove = "./ciao";

    // Rimuovi la directory
    fs::remove_dir(directory_to_remove)?;

    println!("Directory rimossa con successo!");

    Ok(())
}
```

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

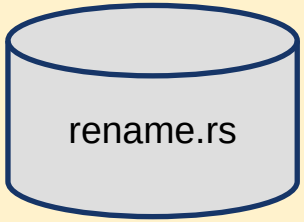
copy()

```
use std::fs;
use std::io;

fn main() -> io::Result<()> {
    // Percorso del file di origine
    let source_path = "./prova.txt";

    // Percorso di destinazione per il file copiato
    let destination_path = "./file.txt";

    // Copia il file
    fs::copy(source_path, destination_path)?;

    println!("File copiato con successo!");

    Ok(())
}
```



rename()

```
use std::fs;

fn main() -> std::io::Result<()> {
    // Definisci il percorso del file o della directory da rinominare
    let old_path = "./file.txt";

    // Definisci il nuovo nome o percorso del file o della directory
    let new_path = "./new.txt";

    // Rinomina il file o la directory
    fs::rename(old_path, new_path)?;

    println!("Rinominato con successo!");

    Ok(())
}
```



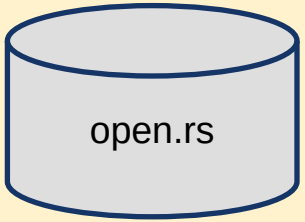
remove_file()

```
use std::fs;

fn main() -> std::io::Result<()> {
    // Definisci il percorso del file da rimuovere
    let file_to_remove = "./new.txt";

    // Rimuovi il file
    fs::remove_file(file_to_remove)?;

    println!("File rimosso con successo!");

    Ok(())
}
```

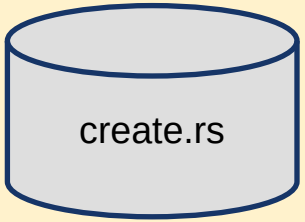
Operazioni con i file

- 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

```rust
use std::fs::File;
use std::io::prelude::*;
use std::io::Error;

fn main() -> Result<(), Error> {
    // Definisci il percorso del file da aprire
    let file_path = "./myfile";

    // Apri il file in modalità di lettura
    let mut file = File::open(file_path)?;

    // Leggi il contenuto del file in una stringa
    let mut contents = String::new();
    file.read_to_string(&mut contents)?;

    // Stampa il contenuto del file
    println!("Contenuto del file:");
    println!("{}", contents);

    Ok(())
}
```



```rust
use std::fs::File;
use std::io::prelude::*;
use std::io::Error;

fn main() -> Result<(), Error> {
    // Definisci il percorso del nuovo file da creare
    let file_path = "./new_file.txt";

    // Crea un nuovo file
    let mut file = File::create(file_path)?;

    // Scrivi una stringa direttamente nel file
    let text = "Questo è un nuovo file creato in Rust!";
    file.write(text.as_bytes())?;

    println!("File creato e scritto con successo!");

    Ok(())
}
```



Operazioni con i file (II)

- Maggiori opportunità sono offerte dalla struct **`std::fs::OpenOption`**
  - viene utilizzata per configurare le opzioni di apertura di un file prima di aprirlo effettivamente tramite la funzione **`open`**
- **`OpenOptions`** fornisce diversi metodi per configurare queste opzioni, tra cui:
  - **`.read(bool)`**: imposta la modalità di apertura per la lettura del file
  - **`.write(bool)`**: imposta la modalità di apertura per la scrittura del file
  - **`.create(bool)`**: specifica se creare il file se non esiste
  - **`.truncate(bool)`**: specifica se troncare il file se esiste già
  - **`.append(bool)`**: specifica se scrivere alla fine del file invece di sovrascrivere il contenuto esistente.

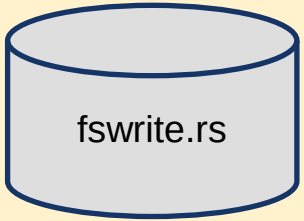
```rust
use std::fs::OpenOptions;
use std::io::prelude::*;

fn main() -> std::io::Result<()> {
    let file_path = "./new_file.txt";

    // Aprire il file in modalità di scrittura, troncandolo se esiste già
    let mut file = OpenOptions::new()
        .write(true)
        .truncate(true)
        .open(file_path)?;

    let text = "Questo è un nuovo file creato in Rust!";
    file.write(text.as_bytes())?;


    Ok(())
}
```

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

```rust
use std::fs;
use std::io::Error;

fn main() -> Result<(), Error> {
    let file_path = "./file.txt";

    // Leggi il contenuto del file in una stringa
    let contents = fs::read_to_string(file_path)
        .expect("Something went wrong reading the file");

    // Stampare il contenuto del file
    println!("Contenuto del file:\n{}", contents);

    Ok(())
}
```

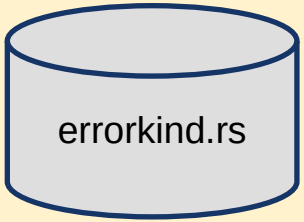
```
use std::fs;
use std::io::Error;

fn main() -> Result<(), Error> {
    let file_path = "./file.txt";
    let text = "Questo è un testo scritto con fs::write!";
    fs::write(file_path, text)?;

    Ok(())
}
```



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`**
- In caso di errore durante le operazioni di I/O viene ritornata una delle varianti disponibili nell' enum **`ErrorKind`** che rappresenta le diverse categorie di errori che possono verificarsi durante le operazioni di input/output (I/O). Questi errori sono suddivisi in categorie per facilitare la loro gestione e consentire una maggiore precisione nell'individuazione dei problemi. Ecco un elenco di alcune varianti di ErrorKind:
  - **`NotFound`**: Indica che il file o la directory specificati non sono stati trovati.
  - **`PermissionDenied`**: Indica che non si dispone delle autorizzazioni necessarie per eseguire l'operazione.
  - **`AlreadyExists`**: Indica che il file o la directory che si sta cercando di creare esiste già.
  - **`InvalidInput`**: Indica che l'input fornito è invalido o non valido per l'operazione.
  - **`TimedOut`**: Indica che l'operazione ha raggiunto il timeout specificato.
  - **`Interrupted`**: Indica che l'operazione è stata interrotta (ad esempio, da un segnale di interruzione).

```rust
fn main() {
    // Tentativo di apertura di un file
    match File::open("testo.txt") {
        Ok(mut file) => {
            let mut contenuto = String::new();
            // Tentativo di lettura del contenuto del file
            match file.read_to_string(&mut contenuto) {
                Ok(_) => println!("Contenuto del file: {}", contenuto),
                Err(e) => match e.kind() {
                    // Gestione specifica per diversi tipi di errori
                    ErrorKind::NotFound => println!("Il file non è stato trovato."),
                    ErrorKind::PermissionDenied => println!("Permesso negato."),
                    _ => println!("Si è verificato un errore durante la lettura del file: {}", e),
                },
            }
        },
        Err(e) => match e.kind() {
            ErrorKind::NotFound => println!("Il file non è stato trovato."),
            ErrorKind::PermissionDenied => println!("Permesso negato."),
            _ => println!("Si è verificato un errore durante l'apertura del file: {}", e),
        },
    }
}
```



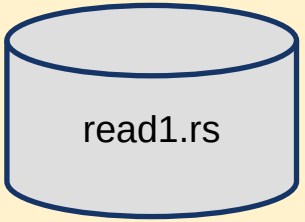
std::io::Read

- 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

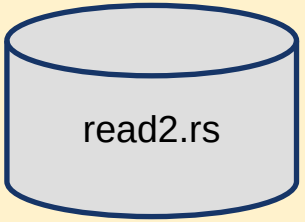
Metodi del tratto Read

- 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

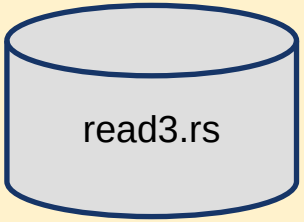
```rust
fn main() {
    // Apro il file in modalità lettura
    let mut file = match File::open("test.txt") {
        Ok(file) => file,
        Err(e) => {
            println!("Errore durante l'apertura del file: {}", e);
            return;
        }
    };

    // Creo un buffer vuoto per contenere i dati letti
    let mut buffer = Vec::new();

    // Leggo il contenuto del file nel buffer
    match file.read_to_end(&mut buffer) {
        Ok(_) => {
            // Converto il buffer in una stringa UTF-8 e stampo il contenuto
            match String::from_utf8(buffer) {
                Ok(content) => println!("{}", content),
                Err(_) => println!("Errore nella decodifica del contenuto del file"),
            }
        }
        Err(e) => println!("Errore durante la lettura del file: {}", e),
    }
}
```



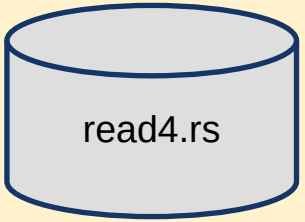
```rust
fn main() {
    // Apro il file in modalità lettura
    let mut file = match File::open("test.txt") {
        Ok(file) => file,
        Err(e) => {
            println!("Errore durante l'apertura del file: {}", e);
            return;
        }
    };

    // Dichiaro una stringa vuota per contenere il contenuto del file
    let mut content = String::new();

    // Leggo il contenuto del file nella stringa
    match file.read_to_string(&mut content) {
        Ok(_) => {
            // Stampo il contenuto del file
            println!("{}", content);
        }
        Err(e) => println!("Errore durante la lettura del file: {}", e),
    }
}
```



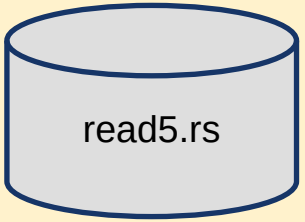
```rust
use std::fs::File;
use std::io::{self, Read};

fn main() -> io::Result<()> {
    // Apro il file in modalità lettura
    let mut file = File::open("test.txt")?;

    // Dichiaro un array vuoto per contenere i byte letti
    let mut buffer = [0; 5]; // Leggo 5 byte

    // Leggo esattamente 5 byte dal file
    file.read_exact(&mut buffer)?;

    // Stampo i byte letti
    println!("I byte letti sono: {:?}", buffer);

    Ok(())
}
```



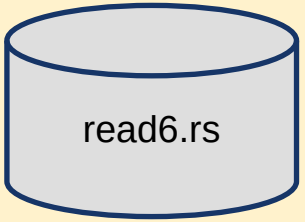
```rust
use std::fs::File;
use std::io::{self, Read};

fn main() -> io::Result<()> {
    // Apro il file in modalità lettura
    let mut file = File::open("test.txt")?;

    // Ottengo un iteratore sui byte del file
    let bytes_iter = file.bytes();

    // Itero sui byte e stampo il valore di ciascun byte
    for byte in bytes_iter {
        match byte {
            Ok(b) => println!("Byte: {}", b),
            Err(e) => println!("Errore durante la lettura del byte: {}", e),
        }
    }

    Ok(())
}
```



```rust
use std::fs::File;
use std::io::{self, Read};

fn main() -> io::Result<()> {
    // Apro due file in modalità lettura
    let file1 = File::open("file1.txt")?;
    let file2 = File::open("file2.txt")?;

    // Concateno i due lettori
    let mut chained_reader = file1.chain(file2);

    // Dichiaro un buffer per contenere i dati letti
    let mut buffer = Vec::new();

    // Leggo i dati concatenati nei buffer
    chained_reader.read_to_end(&mut buffer)?;

    // Converto il buffer in una stringa UTF-8 e la stampo
    match String::from_utf8(buffer) {
        Ok(content) => println!("{}", content),
        Err(_) => println!("Errore nella decodifica del contenuto"),
    }

    Ok(())
}
```



```rust
use std::fs::File;
use std::io::{self, Read};

fn main() -> io::Result<()> {
    // Apro il file in modalità lettura
    let file = File::open("test.txt")?;

    // Creo un nuovo lettore che legge solo i primi 10 byte dal file originale
    let mut limited_reader = file.take(8);

    // Dichiaro un buffer per contenere i dati letti
    let mut buffer = Vec::new();

    // Leggo i primi 10 byte dal file e li memorizzo nel buffer
    limited_reader.read_to_end(&mut buffer)?;

    // Converto il buffer in una stringa UTF-8 e la stampo
    match String::from_utf8(buffer) {
        Ok(content) => println!("{}", content),
        Err(_) => println!("Errore nella decodifica del contenuto"),
    }

    Ok(())
}
```

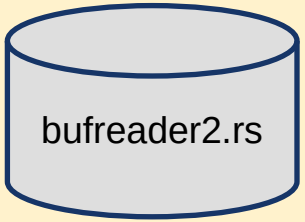


std::io::BufRead

- 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)`**
  - **`fill_buf()`** ritorna il contenuto del buffer in memoria
  - **`consume(...)`** consuma il numero specificato di byte
- Offre i metodi **`read_line(&mut self, buf: &mut String)`** e **`lines(self)`** per accedere al contenuto testuale di un flusso

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


fn main() -> io::Result<()> {
    let path = "myfile";

    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?);
    }

    Ok(())
}
```

```rust
fn main() -> Result<(), Error> {
    // Aprire il file in modalità di lettura
    let file = File::open("./prova.txt")?;

    // Creare un BufReader per il file
    let mut buf_reader = BufReader::new(file);

    // Buffer per memorizzare il contenuto letto
    let mut buffer = String::new();

    // Riempire il buffer interno di BufReader
    buf_reader.fill_buf()?;

    // Leggere il contenuto del buffer fino a quando non si raggiunge la fine del file
    while buf_reader.buffer().len() > 0 {
        // Leggere una riga dal buffer
        let bytes_read = buf_reader.read_line(&mut buffer)?;

        // Stampa la riga letta
        print!("{}", buffer);

        // Pulisci il buffer per prepararlo per la prossima lettura
        buffer.clear();
        // Continua a riempire il buffer interno di BufReader
        buf_reader.fill_buf()?;
    }
    Ok(())
}
```



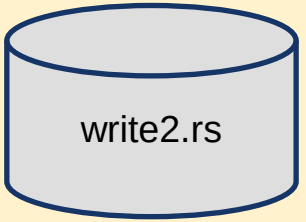
```rust
fn main() -> Result<(), Error> {
    let file = File::open("./prova.txt")?;

    // Creare un BufReader per il file
    let mut buf_reader = BufReader::new(file);

    // Buffer per memorizzare il contenuto letto
    let mut buffer = String::new();
    loop {
        // Riempire il buffer interno di BufReader
        buf_reader.fill_buf()?;

        // Se il buffer è vuoto, siamo alla fine del file, quindi usciamo dal ciclo
        if buf_reader.buffer().is_empty() { break; }

        // Consuma i primi 5 byte dei dati letti dal buffer
        buf_reader.consume(5);

        // Leggi il prossimo blocco di dati dal buffer
        buf_reader.read_line(&mut buffer)?;

        // Stampa il blocco di dati letti
        println!("{}", buffer);
        // Pulisci il buffer per prepararlo per la prossima lettura
        buffer.clear();
    }
    Ok(())
}
```

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

```rust
fn main() -> io::Result<()> {
    let mut file = File::create("output.txt")?;

    // Dati da scrivere nel file
    let data = b"Hello, world!\n";

    let num_writes = 5; // Numero di volte che voglio scrivere i dati nel file
    let mut total_bytes_written = 0;

    // Ciclo per scrivere più volte i dati nel file
    for _ in 0..num_writes {
        // Scrivo i dati nel file e controllo il valore di ritorno
        match file.write(data) {
            Ok(bytes_written) => {
             total_bytes_written += bytes_written; // Aggiorno il conteggio totale dei byte scritti
            }
            Err(err) => {
                // Se si verifica un errore durante la scrittura
                // stampo un messaggio di errore e termino il programma
                eprintln!("Errore durante la scrittura nel file: {}", err);
                return Err(err);
            }
        }
    }
    // Stampo il numero totale di byte scritti con successo nel file
    println!("Totale byte scritti nel file: {}", total_bytes_written);
    Ok(())
}
```

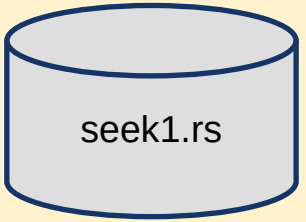
```rust
use std::fs::File;
use std::io::{self, Write};

fn main() -> io::Result<()> {
    // Apro il file in modalità di scrittura
    let mut file = File::create("output.txt")?;

    // Dati da scrivere nel file
    let data = b"Hello, world!\n";

    // Scrivo i dati nel buffer del file
    file.write_all(data)?;

    // Eseguo il flush e gestisco il risultato
    match file.flush() {
        Ok(()) => println!("Dati scritti con successo nel file."),
        Err(err) => {
            eprintln!("Errore durante il flushing dei dati nel file: {}", err);
            return Err(err);
        }
    }

    Ok(())
}
```



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 stream_position(&mut self) -> Result<u64>`: restituisce la posizione corrente del cursore rispetto all'inizio del flusso

```rust
fn main() -> io::Result<()> {
    // Apro il file in modalità di scrittura e lettura
    let mut file = OpenOptions::new()
        .read(true)
        .write(true)
        .create(true)
        .open("example.txt")?;
    file.write_all(b"Hello, world!")?;
    // Sposto il cursore di lettura/scrittura alla fine del file
    file.seek(SeekFrom::End(0))?;

    // Scrivo dei dati aggiuntivi alla fine del file
    file.write_all(b" Additional data")?;

    // Sposto il cursore di lettura/scrittura alla posizione 7 nel file
    file.seek(SeekFrom::Start(7))?;

    // Scrivo dei dati in una posizione specifica nel file
    file.write_all(b"Rust ")?;

    // Sposto il cursore di lettura/scrittura all'inizio del file
    file.seek(SeekFrom::Start(0))?;
    let mut buffer = String::new();
    file.read_to_string(&mut buffer)?;
    println!("Contenuto del file: {}", buffer);
    Ok(())
}
```




```
fn main() -> io::Result<()> {
        let mut file = OpenOptions::new()
        .read(true)
        .write(true)
        .create(true)
        .open("example.txt")?;
    file.write_all(b"Prova di Testo del File")?;
    file.seek(SeekFrom::Start(0))?;         // Sposto il cursore all'inizio del file utilizzando seek_from_start
    println!("Posizione corrente del cursore: {}", file.stream_position()?);
    let mut buffer = [0; 10];
    file.read_exact(&mut buffer)?;         // Leggo i primi 10 byte dal file
    println!("I primi 10 byte del file: {:?}", buffer);
    println!("Posizione corrente del cursore: {}", file.stream_position()?);
    file.seek(SeekFrom::End(0))?;         // Sposto il cursore alla fine del file utilizzando seek_from_end
    println!("Alla Fine: posizione corrente del cursore: {}", file.stream_position()?);
    file.seek(SeekFrom::Current(-5))?;    // Sposto il cursore 5 byte indietro dalla fine del file
    println!("Indietro di 5: posizione corrente del cursore: {}", file.stream_position()?);
    file.write_all(b" Additional data")?; // Scrivo ulteriori dati alla fine del file
    file.seek(SeekFrom::Start(10))?;   // Riporto il cursore alla posizione 10 all'interno del file
    println!("Vado in posizione 10: posizione corrente del cursore: {}", file.stream_position()?);
    let mut buffer = [0; 5];
    file.read_exact(&mut buffer)?;       // Leggo 5 byte dalla posizione corrente
    println!("Dati letti dalla posizione corrente: {:?}", buffer);
    file.seek(SeekFrom::Start(0))?;
    let mut buffer = Vec::new();
    file.read_to_end(&mut buffer)?;
    println!("Tutto il file: {:?}", buffer);
    Ok(())
}
```