

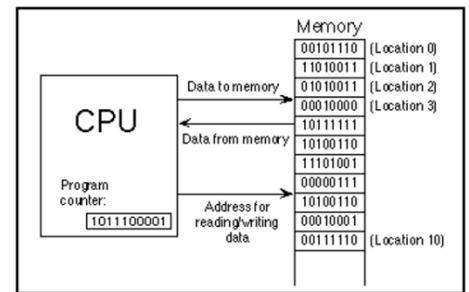
API - PROGRAMMAZIONE DI SISTEMA

0) ALLOCAZIONE MEMORIA (C, C++)

Programma eseguibile = insieme di istruzioni macchina (il cui significato è cablato nel processore), dati, valori di configurazione e controllo rappresentati come sequenze binarie (il cui significato è contenuto nelle istruzioni macchina e dipende da processore e OS). Per essere eseguito, deve essere caricato nella memoria accessibile al processore (nei sistemi grossi se ne occupa il **loader** [trasferisce da disco a RAM il contenuto dell'exe]). Quando si trova in memoria (**RAM**), può essere eseguito, ovvero il processore prende una alla volta le istruzioni macchina del programma e le esegue. Il modello base di esecuzione è:

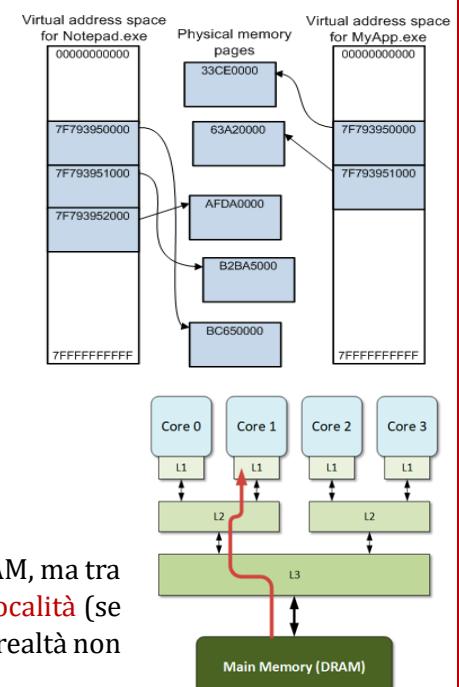
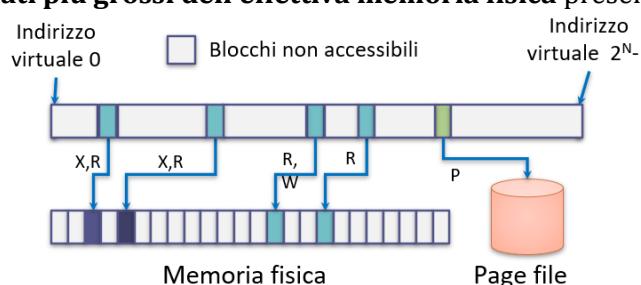
- **Fetch** = istruzione trasferita dalla memoria alla CPU
- **Decode** = istruzione trasformata in comandi da eseguire
- **Execute** = comandi eseguiti

Il processore fa riferimento ad una cella di memoria tramite l'**indirizzo** (IP = registro della CPU che ricorda l'indirizzo della prossima istruzione da eseguire). L'esecuzione di un programma avviene nel suo **spazio di indirizzamento** (una sorta di array di Byte [non per forza contiguo] gestito dall'OS) che memorizza dati, istruzioni, info di controllo; il numero di celle “**potenzialmente**” presenti nello spazio di indirizzamento dipende dal processore, mentre il numero di celle “**effettivamente**” presenti è limitato dalla RAM disponibile ed è controllato dall'OS



Gli indirizzi che un programma usa (**virtuali [logici]**) non corrispondono alla effettiva posizione in memoria (**fisici**). Un blocco hardware nel processore (**MMU [Memory Management Unit]**) traduce indirizzo logico in fisico; la traduzione consente:

- programmi diversi insieme in esecuzione **non interferiscano** tra loro
- controllare quali **operazioni** siano **possibili**
- manipolare **dati più grossi dell'effettiva memoria fisica** presente



In sistemi non elementari il processore non interagisce direttamente con la RAM, ma tra i 2 è presente una **CACHE** (> velocità di accesso) che si basa sul principio di **località** (se accedo all'indirizzo **I**, è molto probabile che l'accesso subito dopo sia a **I+Δ** [in realtà non è 1 cache, ma una gerarchia di cache su più livelli sempre > capaci ma > lenti])

I linguaggi *ad alto livello* tolgono la visibilità di dettagli legati all'esecuzione e introducono delle sovrastrutture che gestiscono l'interazione tra codice ed effettiva esecuzione che impattano sul codice scritto, introducendo vincoli e restrizioni. Ogni linguaggio di programmazione ha uno specifico **modello di esecuzione** (comportamenti dell'elaboratore rispetto ai costrutti di alto livello del linguaggio), ma questo non corrisponde a quello di un dispositivo reale: è il **compilatore** che pone uno strato di adattamento composto dalle **traduzioni** costrutto-istruzioni e da **run-time libraries** (**librerie di esecuzione**).

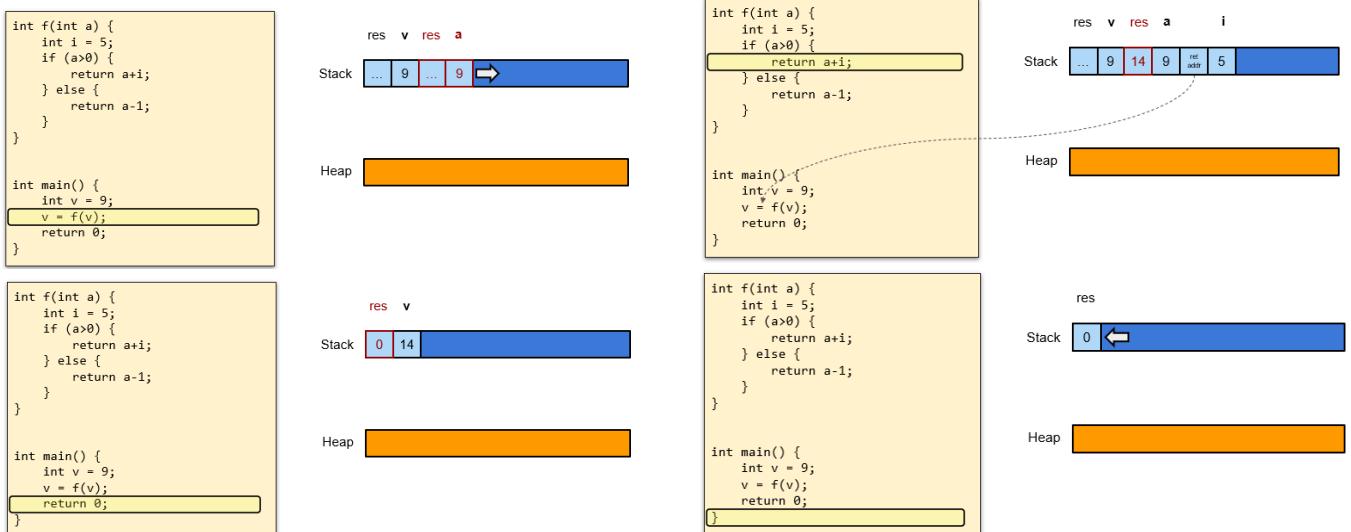


Queste librerie offrono agli applicativi meccanismi di base per il loro funzionamento (es. interfaccia uniforme tra i diversi OS) e sono costituite da funzioni invisibili al programmatore e funzioni standard

Nel **modello di esecuzione di C e C++**, i programmi sono pensati come gli unici utilizzatori di un elaboratore a loro completamente dedicato (isolamento) e assumono di poter accedere a qualsiasi indirizzo di memoria nello spazio di indirizzamento. Un programma C/C++ è formato da un insieme di istruzioni eseguite nell'ordine indicato dal programmatore e all'interno del flusso principale di esecuzione si possono attivare flussi secondari (**thread**). A differenza di Java, il C++ ha anche il *distruttore* oltre che il costruttore; il punto di partenza è definito dal **main()** in C e dai *costruttori* delle variabili globali seguiti dal **main()** in C++

In C/C++ ci sono anche:

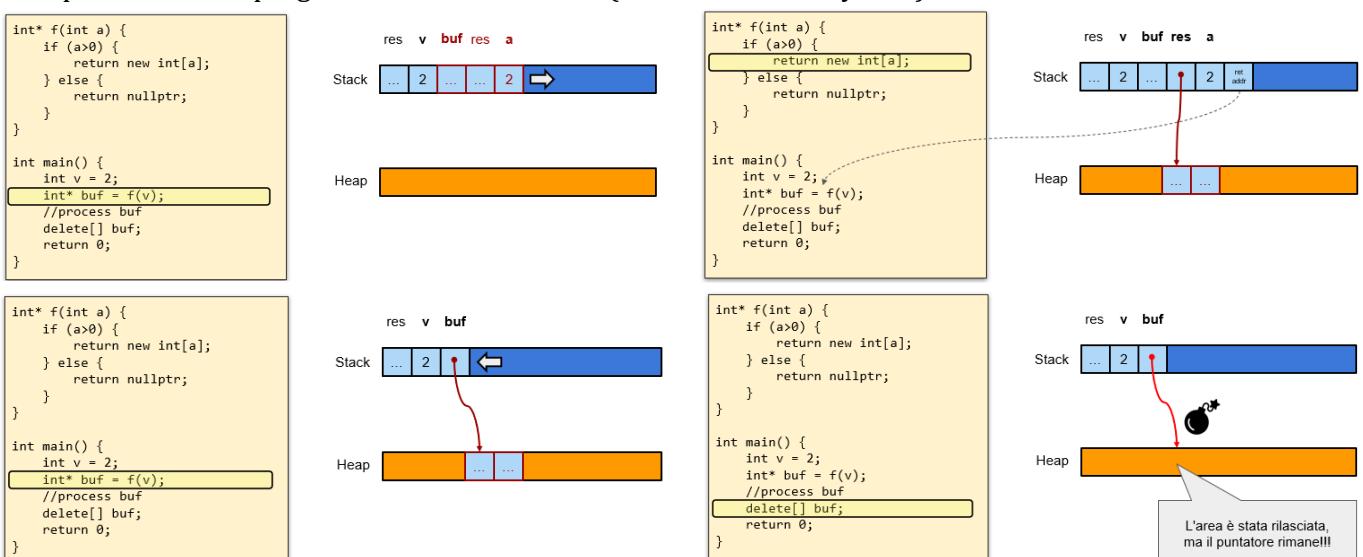
- **Stack** = gestione **chiamate annidate** tra funzioni [supporta **ricorsione** e **variabili locali**, in C++ anche gestione strutturata delle eccezioni]; allocato automaticamente all'avvio del programma. Dato che ha dimensione finita, c'è limite alla profondità max di ricorsione e alla dimensione totale delle variabili locali usabili (se **Stack overflow** → l'OS termina il programma)



⚠ Ogni volta che faccio una syscall (ovvero passo da user a kernel), lo stack viene spostato per motivi di sicurezza (es. come chi ci spia guardando nella nostra spazzatura, così lo stack potrebbe essere usato). Per questo passare da user a kernel mode è molto oneroso ogni volta

⚠ L'allocazione ed il rilascio sullo stack sono efficienti grazie alla politica di espansione/contrazione. Non si può memorizzare nello stack un dato più a lungo della funzione in cui è stato allocato e non si può salvare un dato con dimensione > dello stack

- **Heap (Free Store)** = gestione **dinamica della memoria** (allocare e rilasciare blocchi di dimensione arbitraria [eventualmente non nota in fase di compilazione, ma solo run-time]). A differenza dello stack, le aree allocate sull'heap non sono associate ad un nome di variabile, ma si accede al loro contenuto solo tramite **puntatori**. È responsabilità del programmatore fare le **free** (altrimenti memory leak)

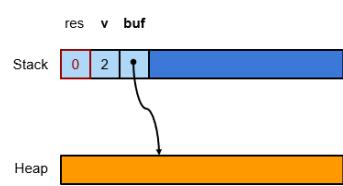


```

int* f(int a) {
    if (a>0) {
        return new int[a];
    } else {
        return nullptr;
    }
}

int main() {
    int v = 2;
    int* buf = f(v);
    //process buf
    delete[] buf;
    return 0;
}

```

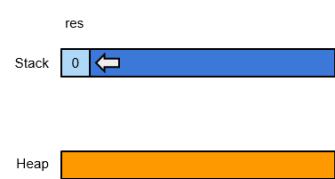


```

int* f(int a) {
    if (a>0) {
        return new int[a];
    } else {
        return nullptr;
    }
}

int main() {
    int v = 2;
    int* buf = f(v);
    //process buf
    delete[] buf;
    return 0;
}

```



⚠ Come viene organizzato lo spazio di indirizzamento? Codice eseguibile (istruzioni in codice macchina; lettura + esecuzione **r-x**), costanti (solo lettura **r--**), variabili globali (lettura + scrittura **rw-**), stack (lettura + scrittura **rw-**; 1 per thread), heap (1 solo; **rw-**) sono le sole zone accessibili (quelle non accessibili vengono non segnate o segnate con **---**) [⚠ se si vede lo spazio di indirizzamento con **cat /proc/<PID>/maps**, dopo **gcc -o programma programma.c**, vedo che heap e stack sono inizialmente vuoti, ovvero dimensione 0, poi verranno gestiti dalle librerie di esecuzione]

Distinguiamo **VARIABILI**:

- **GLOBALI** = indirizzo fisso determinato da compilatore e linker
- **LOCALI** = indirizzo relativo alla cima dello stack (vita della funzione di appartenenza), valore iniziale casuale
- **DINAMICHE** = indirizzo assoluto determinato in esecuzione (accessibili solo con **puntatori**) (vita controllata dal programmatore); in:
 - **C**
 - **void *malloc(size_t s)** [se fallisce, **NULL**]
 - **void *calloc(int n, size_t s)** [se fallisce, **NULL**]
 - **void *realloc(void* p, size_t s)** [se fallisce, blocco non modificato]
 - **C++**
 - **new NomeTipo{argomenti...}** [se fallisce, eccezione]
 - **new (std::nothrow) NomeTipo{args...}** [se fallisce, **nullptr**]
 - **new NomeClasse[numero_elementi]**

⚠ I rilasci devono essere **duali** alla allocazione (**malloc** → **free()**, **new** → **delete**, **new[]** → **delete[]**)

I **PUNTATORI** in C/C++ permettono accesso diretto a blocco di memoria (es. **int A = 10** → **int* pA = &A**) e possono essere allocati per lo scopo (es. **int* pB = new int{24}**). Possono essere esplicitamente invalidi (**0**, **NULL** (**(void *)0**) o **nullptr**).

I puntatori si possono **usare per** accedere ad un'informazione contenuta in un'altra struttura dati senza doverla ricopiare o per indicare ad una funzione dove depositare parte dei dati che deve calcolare.

```

bool read_data1(int* result) {
    //Se il puntatore sembra valido
    //e ci sono dati...
    if(result!=nullptr && some_data_available()) {

        //accedi in scrittura all'indirizzo indicato dal chiamante
        *result = get_some_data();

        //indica operazione eseguita correttamente
        return true;
    } else
        //operazione fallita
        return false;
}

```

```

int* read_data() {
    unsigned int size = count_available_data();
    if (size > 0 ) {
        int* ptr= new int[size];
        consume_data(ptr, size);
        //indica operazione eseguita
        //correttamente
        return ptr;
    } else
        //nessun dato disponibile
        return nullptr;
}

int* result = read_data();
...
if (result) delete[] result;

```

```

{
    const char* ptr = "Quel ramo del lago di Como...";

    //conta gli spazi
    int n=0;

    //usa l'aritmetica dei puntatori
    for (int i=0; *(ptr+i)!=0; i++) {

        //usa il puntatore come fosse un array
        if (isSpace(ptr[i]) ) n++;
    }
    //altro...
}

```

```

struct simple_list {
    int data;
    struct simple_list *next;
};

struct simple_list *head;
// head è responsabile di tutte le proprie parti
// quando si rilascia la lista, occorre liberarne
// tutti gli elementi

```

⚠ In **C** la gestione dei puntatori è tutta affidata al programmatore (limitare gli accessi, non assegnare a puntatori indirizzi non mappati, rilasciare tutta la memoria dinamica allocata). In **C++** sono stati introdotti riferimenti e smart pointer per aiutare il programmatore (ma va comunque saputo il problema)

RISCHI:

- accedere ad un indirizzo quando il suo ciclo di vita è terminato ha effetti imprevedibili (“**Dangling pointer**”)
- non rilasciare la memoria spreca risorse (“**Memory leakage**”), ma rilasciarla più volte corrompe le strutture dell’heap (“**Double free**”)
- se assegno ad un puntatore un indirizzo non mappato nello spazio di indirizzamento e lo uso, si ha una **interruzione del processore**
- se non inizializzo un puntatore e lo uso, potrebbe puntare ovunque (“**Wild Pointer**”)

Memory leakage

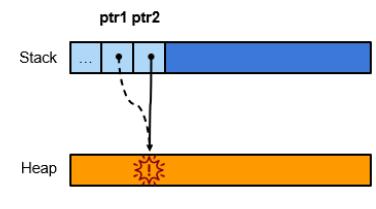
Dangling Pointer

```
{  
    char* ptr = nullptr;  
  
    { // inizio di un nuovo blocco  
        char ch='!';  
        ptr = &ch;  
  
    } // fine blocco: lo stack si contrae  
    // le variabili qui definite cessano di  
    // esistere  
  
    printf("%c", *ptr);  
    //contenuto impredicibile  
}
```

```
{  
    char* ptr = NULL;  
  
    ptr = (char*) malloc(10); // Alloco un blocco  
  
    strcpy(ptr, "Leakage!"); // Lo uso  
  
    printf("%s\n", ptr);  
}  
tracce  
// Ne perdo le
```

Double free

```
{  
    char* ptr1 = malloc(10);  
  
    char* ptr2 = ptr1;  
  
    // use ptr1 and/or ptr2  
  
    free(ptr1);  
}  
free(ptr2);
```



⚠ Il vincolo di rilascio è problematico: dato un indirizzo non nullo, non è possibile dire se sia **valido** o meno, né a quale **area** appartenga, né se occorra o meno **rilasciarlo**. Ogni volta che alloco un blocco dinamico e ne salvo l’indirizzo in una variabile, questa diventa **proprietaria** del blocco (*owner*) ed è lei che dovrà liberarlo; se invece ad un puntatore viene assegnato l’indirizzo di un’altra variabile, la proprietà è della libreria di esecuzione

Con **strutture dati complesse**, queste devono appoggiarsi a blocchi di memoria ausiliari in cui memorizzare le informazioni che gestiscono, che dovranno essere rilasciati quando l’oggetto viene distrutto (es. con oggetto di tipo `std::vector<T>`); le strutture dati possono anche mantenere riferimenti ad oggetti dell’OS. Collettivamente, questi dati si chiamano **dipendenze** (il C non ci aiuta, mentre il C++ offre per le **class**, **struct** e **union** i costruttori e distruttori)

1) INTRODUZIONE A RUST

RUST è un linguaggio compilato focalizzato su **correttezza**, **velocità** e supporto alla **programmazione concorrente**; è adatto alla programmazione di sistema in quanto è **staticamente** e **fortemente tipizzato** (tutti i tipi sono noti in fase di compilazione). È un linguaggio **memory safe** per la programmazione di sistema **privo di comportamenti non definiti**, concorrente e pratico + offrire **astrazioni a costo nullo** (generazione del miglior codice assembler). Non ha garbage collection, ma ha **ecosistema** di compilazione/gestione dipendenze/test integrato (**Cargo**). Rust si appoggia ad un sistema di validazione dei tipi in fase di compilazione che **impedisce** (per i programmi che non fanno uso delle estensioni “**unsafe**”):

- **Dangling pointer**
- **Double free** (esiste 1 solo possessore del dato, l’unico che può rilasciare [introdotto il **possesso di un valore**])
- **Corse critiche** (accesso a dati il cui contenuto è indeterminato a seguito di eventi fuori dal programma)
- **Buffer overflow** (accessi ad aree di memoria contigue a quelle di una variabile, ma non di sua appartenenza)
- **Iteratori invalidi e Overflow aritmetici** (solo in debug)

Il **compilatore ottimizza aggressivamente** dimensioni e velocità del codice generato: le strutture dati base del linguaggio favoriscono l’uso della **cache** (preferendo array, rispetto a strutture con puntatori), l’invocazione standard è basata su **indirizzi statici** e la **gestione integrata delle dipendenze** facilita la condivisione del codice.

Tipi generici:

```
struct MyVec<T> {
    // ...
}

impl<T> MyVec<T> {
    pub fn find<P>(&self, predicate: P) -> Option<&T>
    where P: Fn(&T) -> bool {
        for v in self {
            if predicate(v) { return Some(v); }
        }
        None
    }
}
```

Tipi algebrici e pattern:

```
enum HttpRequest {
    Get, Post(String), Put(String), Unknown
}

fn process(req: HttpRequest) {
    match req {
        HttpRequest::Get => { /* handle get request */ },
        HttpRequest::Post(data) | HttpRequest::Put(data)
        if !data.is_empty() => {
            // process data
        } else => { /* manage error */ }
    }
}
```

```
// Option<T> è un enum che può essere Some(T) o None
if let Some(f) = my_vec.find(|t| t >= 42) { /* found */ }

enum DecompressionResult {
    Finished { size: u32 },
    InputError(std::io::Error),
    OutputError(std::io::Error),
}
// errore in fase di compilazione: manca caso generico
match decompress() {
    Finished { size } => { /* analizzato con successo */ }
    InputError(e) if e.is_eof() => { /* gestisco EOF */ }
    OutputError(e) => { /* output fallisce con l'errore e */ }
}
```

Test e documentazione integrati:

```
#[test]
fn it_works() { assert_eq!(1 + 1, 2); }

/// Returns one more than its argument.
///
/// ``
/// assert_eq!(one_more(42), 43);
/// ``
pub fn one_more(n: i32) -> i32 { n + 1 }
```

I **puntatori** sono **controllati in fase di compilazione** (proprietà/possesso del dato è **esteso** ai dati cui si accede indirettamente [sia per riferimento sia con puntatore]), la **sicurezza dei thread** è incorporata nel sistema dei tipi e non c'è **nessuno stato nascosto** (gli errori e le opzionalità richiedono il controllo esplicito del programmatore)

Possesso di valore:

```
// possiede il valore in lettura e scrittura
let mut v = Vec::new();
// Il valore di v può essere modificato
v.push(1);

// movimento: ora v1 possiede il valore che prima era in v
let mut v1 = v;

// Il compilatore impedisce l'accesso tramite v
v.push(2);

// Ora v1 contiene [1, 3]
v1.push(3);
```

```
// possiede il valore in lettura e scrittura
let mut i = 12;

// r è un riferimento ad i: ha in PRESTITO il suo valore
let r = &i;

// mentre r esiste, non è lecito modificare i
i = 23;

// accedo al valore a cui r fa riferimento
println!("{}: {}", *r);
```

⚠ In JavaScript quando uso `const` rendo la variabile non riassegnabile (ovvero non cambio il valore assegnato), ma se facessi `variabile.push(8)` dentro `variabile` finisce 8 (cambia il contenuto). In Rust, con `let mut variabile` faccio in modo che il valore contenuto in quella variabile può essere modificato, ma solo da quella variabile (guarda esempio sopra immagine sinistra)

```
let v = Vec::new();
// la compilazione avviene correttamente:
println!("len: {}", v.len());

// la compilazione non avviene correttamente:
// richiede un accesso mutabile
v.push(42);
```

Puntatori controllati in compilazione:

```
// Ogni valore ha un proprietario, responsabile del rilascio (RAII)
// il compilatore verifica che ci sia sempre un solo possessore
// e che non avvengano doppi rilasci
let x: Vec<i32> = Vec::new();
// y ora possiede il valore contenuto in x
let y = x;
drop(x); // illegale, y è ora il proprietario

// Nessun puntatore vive dopo le modifiche o il rilascio
// Nessun dangling pointer/use-after-free
let mut x = vec![1, 2, 3];
let first = &x[0];
let y = x;
println!("{}: {}", *first); // illegale, first diventa invalido
// quando x è stata mossa
```

```
let v = Vec::new();
accidentally_modify(&v);

fn accidentally_modify(v: &Vec<i32>) {
    println!("len: {}", v.len());
    // la compilazione non avviene correttamente;
    // è necessario un &mut Vec<i32>
    push(v);
}

// dichiarazione esplicita di un puntatore mutabile
fn push(v: &mut Vec<i32>) {
    v.push(42);
}
// anche questo non verrà compilato: v non è mutabile
push(&mut v);
```

Sicurezza dei thread incorporata nei tipi:

```
use std::cell::Rc; // reference-counted, non atomico
use std::sync::Arc; // reference-counted, atomico
// non compila:
let rc = Rc::new("not thread safe");
std::thread::spawn(move || {
    println!("I have an rc with: {}", rc);
});
// compila correttamente:
let arc = Arc::new("thread safe");
std::thread::spawn(move || {
    println!("I have an arc with: {}", arc);
});
// non compila:
let mut v = Vec::new();
std::thread::spawn(|| {
    v.push(42);
});
let _ = v.pop();
```

Nessuno stato nascosto:

```
enum Option<T> { Some(T), None, }
enum Result<T,E> { Ok(T), Err(E), }

// r è Option<&T>, non &T -
// non può essere utilizzato senza verificare la presenza di None
let r = my_vec.find(|t| t >= 42);

// n è di tipo Result: per accedere al valore occorre verificare
// che non sia un Err(_)
let n = "42".parse();
// ? Suffixo che forza un ritorno in caso di errore
let n = "42".parse()?
```

⚠ La memoria è rilasciata non appena una variabile esce di visibilità (no interruzioni per il garbage collection), nessuno spreco di memoria, si possono invocare syscall, si può eseguire su device senza OS

Controllo allocazione e chiamate dinamiche:

```
// Si può allocare nello heap
let heap_x = Box::new(x);
let heap_z = vec![0; 1024];

// Si può cambiare l'allocatore
#[global_allocator]
static A: MyAllocator = MyAllocator;

// Si può attivare la chiamata dinamica(vtable):
// solo una copia di find per T
impl<T> MyVec<T> {
    pub fn find(&self, f: &dyn Fn(&T) -> bool) -> Option<&T> {
        // ...
    }
}
```

Si **crea un nuovo progetto** Rust con i comandi:

`cargo new project_name`
`cargo new -lib library_name`

Si **compila/esegue un progetto** con i comandi:

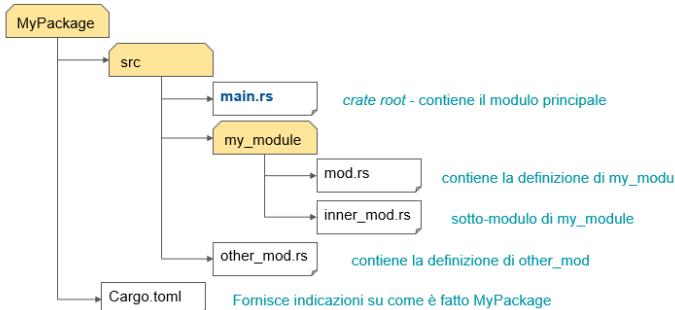
`cargo build`
`cargo run`

Si può **importare una libreria pubblica** con `cargo add <libname>` e si può scaricare librerie su <https://crates.io>

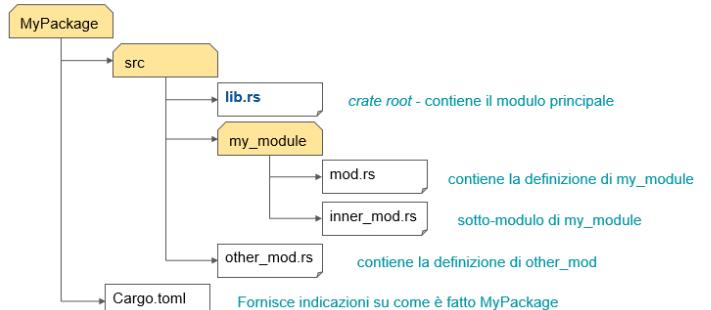
Definizioni:

- **Crate** = unità di compilazione che può generare un eseguibile o una libreria
- **Create root** = file sorgente da cui parte il compilatore Rust per creare il modulo principale del crate (un crate eseguibile [binario] deriva da `src/main.rs`, mentre una libreria deriva da `src/lib.rs`)

Struttura di un progetto eseguibile Rust



Struttura di un progetto di libreria Rust



- **Module** = suddividere gerarchicamente il codice sorgente in unità logiche differenti (un modulo può contenere funzioni, tipi e altri moduli)
- **Package** = insieme di 1 o più crates per fornire funzionalità (progetto); è ospitato in una cartella che contiene il `Cargo.toml` (descrive come costruire i crates di cui è composto il package)

Il punto di ingresso di un programma è `fn main() {...}`; per stampare su `stdout` si usa `print!` o `println!` (es. la sintassi `println!("Hello, {}!", "ciao")` stamperà `Hello, ciao!`). Le variabili si dichiarano con `let variabile : tipo = valore` (e se voglio poi poterle modificare dovrò fare `let mut`) [*]

Vediamo degli esempi utili:

```
//Read a line from the console

use std::io;
fn main() {
    println!("Type a name:");
    let mut name = String::new();
    io::stdin()
        .read_line(&mut name)
        .expect("Error reading line");
    println!("Hello, {}!", name);
}
```

```
//Read a file line by line

use std::fs::File;
use std::io::{BufRead, BufReader};
fn main() {
    let file = File
        ::open("hello.txt")
        .expect("File error");
    let reader = BufReader
        ::new(file);
    for line in reader.lines() {
        println!("{}",
            line.expect("Line error"));
    }
}
```

```
//Write a file

use std::fs::File;
use std::io::Write;
fn main() {
    let mut file = File
        ::create("output.txt")
        .expect("Creation error");
    file
        .write_all("Hello!".as_bytes())
        .expect("Write error");
}
```

```
//List directory contents

use std::fs;
fn main() {
    let entries = fs
        ::read_dir(".").expect("Failed to read dir");
    for entry in entries {
        let path = entry
            .path().expect("Failed to read")
            .path();
        println!("{}: {}", path.display());
    }
}
```

⚠ Se ho `for line in lines()` e poi ho `match line` per verificare se Ok/Err, dopo il match non esisterà più line perché è stata aperta da match; quindi noi dovremmo usare `match &line` in modo che con un **puntatore solo in lettura**, line non venga modificata

2) LINGUAGGIO RUST

Una **VARIABILE** lega un valore/espressione (non più modificabile) ad un nome e viene introdotta con `let` (o `let mut` come abbiamo visto prima per poter modificare il valore); ad ogni variabile è assegnato staticamente (per tutto il programma) un tipo, che può essere esplicitamente definito oppure no (come visto prima [*]):

```
let v: i32 = 123; // v è immutabile e ha tipo i32 (intero a 32 bit con segno)
// v = -5;           // ERRORE: Non è possibile riassegnare il valore

let mut w = v;     // w può essere riassegnata, ha lo stesso tipo di v (i32)
w = -5;           // OK. Ora w vale -5

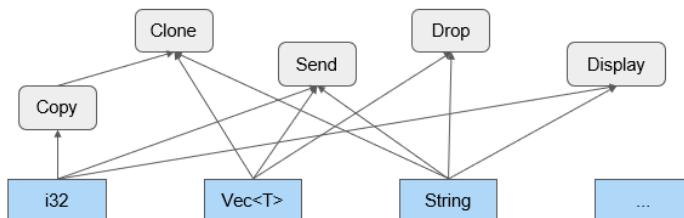
let x = 1.3278;   // x è immutabile di tipo f64 (floating point a 64 bit)

let y = 1.3278f32; // y è immutabile di tipo f32 (floating point a 32 bit)

let one_million = 1_000_000 // si possono usare '_' per separare le cifre
```

La maggior parte delle espressioni (es. `4 + (3*2)`) produce un valore, ma ci sono anche costrutti che non producono un valore a differenza del C/C++ (es. le **assegnazioni** in Rust sono tipo `()`- che corrisponde al void).

Rust offre **tipi predefiniti**. I tipi non sono organizzati in una gerarchia di eredità, ovvero le proprietà di cui il tipo gode sono definite con un meccanismo dichiarativo basato su un insieme di **tratti** (**tratto** = insieme di metodi [comportamenti] che un tipo implementa) [i tratti sono come le interfacce di altri linguaggi]. Dato che tipi diversi possono avere **tratti comuni**, si crea una sorta di parentela articolata tra tipi:



TIPI ELEMENTARI sono:

- **Interi signed** → `i8, i16, i32, i64, i128, isize`
- **Interi unsigned** → `u8, u16, u32, u64, u128, usize`
- **Floating point** → `f32, f64`
- **Logici** → `bool`
- **Caratteri (32 bit)** → `char`
- **Unit** → `()` → rappresenta una tupla di 0 elementi (il simbolo `()` indica sia il tipo sia il suo unico valore possibile) [corrisponde al void di C/C++]

Le **TUPLE** sono collezioni **ordinate** di valori **eterogenei** (`elem1, elem2, ...`). Si accede ad un singolo campo con la notazione `.indice`:

```

let t: (i32, bool) = (123, false); // t è una tupla formata da un intero
                                    // e da un booleano

let mut u = (3.14, 2.71);          // u è una tupla riassegnabile formata
                                    // da due double

let i = t.0;                      // i contiene il
valore 123

u.1 = 0.0;                       // adesso u contiene
(3.14, 0.0)

```

Ci sono diversi modi per rappresentare **INDIRIZZI DI MEMORIA**:

- **RIFERIMENTI** (condivisi e mutabili; non possono essere nulli) [gestiti dal *borrow checker*]:
 - **let r1 = &v;** → definisce ed inizializza il riferimento **r1**, ovvero la variabile **r1** prende in prestito il valore (o espressione) **v** e potrà accedervi in sola lettura con ***r1** [non modificabile] (**MULTIPLE READERS**)
 - **let r2 = &mut v;** → definisce ed inizializza il riferimento **r2**, ovvero la variabile **r2** prende in modo esclusivo il valore (o espressione) **v** e potrà modificarlo con ***r2 = valore;** [**se esiste un mut su quel valore, non posso crearne altri perché mutualmente esclusivo; mentre esiste un mut sul valore v, non potrà nemmeno leggere e modificare v in quanto preso in esclusiva da r2**] [se assegno un nuovo valore ad **r2**, il vecchio viene liberato] (**SINGLE WRITER**)

<code>fn main() { let mut i = 32; let r = &i; println!("{}", *r); i = i+1; // Problematico! println!("{}", *r); }</code>	<code>error[E0506]: cannot assign to `i` because it is borrowed --> src/main.rs:11:3 8 let r = &i; -- borrow of `i` occurs here ... 11 i = i+1; ^^^^^^ assignment to borrowed `i` occurs here 12 println!("{}", *r); -- borrow later used here</code>	<code>fn main() { let mut i = 32; let r = &mut i; println!("{}", i); // Problema! *r = *r+1; println!("{}", *r); }</code>	<code>error[E0502]: cannot borrow `i` as immutable because it is also borrowed as mutable --> src/main.rs:9:18 8 let r = &mut i; ----- mutable borrow occurs here 9 println!("{}", i); ^ immutable borrow occurs here 10 11 *r = *r+1; -- mutable borrow later used here</code>
--	---	---	---

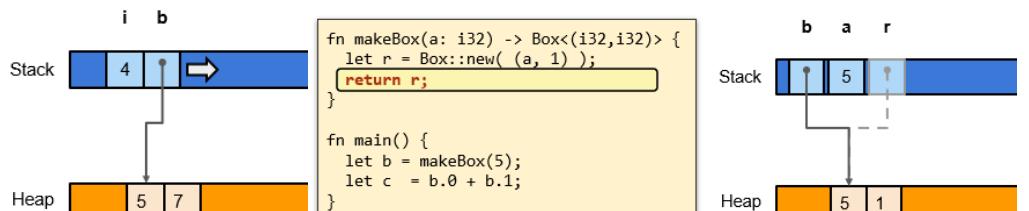
- **BOX** = in alcuni casi non si sa la dimensione del valore da allocare oppure va prolungato il tempo di vita della variabile oltre lo scope in cui è definita: in questi casi si può allocare un oggetto sull'heap usando il tipo generico **Box<T>** con sintassi **let b = Box::new(v);** (quindi la variabile **b** conterrà un puntatore al blocco allocato sull'heap contenente **v**) [**allocazione dinamica**]. A **v** si accede con ***b**.

⚠ A fine scope della variabile **b**, il blocco sarà rilasciato, ma solo se il contenuto di **b** non è stato messo in un'altra variabile con un altro scope (in questo caso cambia lo scope del blocco)

```

fn useBox() {  
    let i = 4;  
    let mut b = Box::new( (5, 2) );  
  
    (*b).1 = 7;  
  
    println!("{}:{}, {}", *b); // (5,7)  
    println!("{}:{}, {}", b); // (5,7)  
}

```



⚠ Nel 2°, quando faccio **let b = makeBox(5)**, viene allocato sullo stack lo spazio per il valore ritornato dalla funzione (**r**)

- **PUNTATORI NATIVI** (costanti e mutabili) [equivalenti ai puntatori in C/C++] = posso accedere al loro contenuto in lettura/scrittura solo in blocchi **unsafe{...}**:
 - ***const T** = puntatore costante
 - ***mut T** = puntatore mutabile

⚠ L'uso dei puntatori in Rust è semplificato grazie alle **garanzie del compilatore** (verifica il possesso e il tempo di vita delle variabili e garantisce solo accessi realizzabili) [se non voglio questo devo usare **unsafe{...}**]

ARRAY = sequenza di oggetti **omogenei** disposti **consecutivamente** nello stack, con **dimensione** finita alla sua creazione e poi immutabile:

```

let a: [i32; 5] = [1, 2, 3, 4, 5]; // a è un array di 5 interi

let b = [0; 5];                  // b è un array di 5 interi inizializzati a
@                                         // NOTARE il ; per distinguere le notazioni

let l = b.len();                 // l vale 5
let e = a[3];                     // e vale 4

```

Rust permette di fare riferimento ad una sequenza di valori la cui lunghezza è nota a runtime, con le **SLICE** ("fat pointer"), che sono **porzioni di array**. Sono immutabili, se le voglio mutabili devo usare la notazione `&mut a[...]`

```
let a = [ 1, 2, 3, 4 ];
let s1: &[i32] = &a; //s1 contiene i valori 1, 2, 3, 4
let s2 = &a[0..2]; // s2 contiene i valori 1, 2
let s3 = &a[2..]; // s3 contiene i valori 3, 4
```

⚠ Come negli array, accedo ad un singolo elemento di una slice con `s[i]`

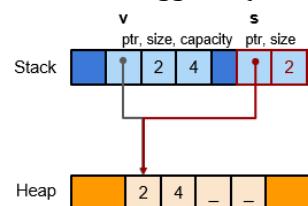
Il tipo `Vec<T>` rappresenta una sequenza ridimensionabile di elementi di tipo `T` allocati sull'heap; una variabile `Vec<T>` è una **tupla di 3 valori privati** (accedibili con `vet.ptr()`, `vet.size()`, `vet.capacity()`):

- `ptr` → puntatore a buffer allocato sull'heap dove sono memorizzati gli elementi
- `size` → intero unsigned che indica la dimensione del buffer
- `capacity` → intero unsigned che indica il n° di elementi nel buffer

Quando inserisco un elemento in `Vec<T>`, questo viene messo nella **1^ posizione libera**; se buffer già pieno, viene allocato un **nuovo buffer** di dimensione maggiore (es. se aveva `capacity 5`, verrà messo `capacity 10`, poi `20` etc...):

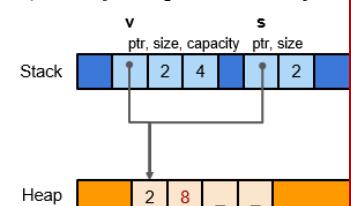
```
fn useVec() {
    let mut v: Vec<i32> = Vec::new();
    v.push(2);
    v.push(4);

    let mut s = &mut v[..];
    s[1] = 8;
}
```



```
fn useVec() {
    let mut v: Vec<i32> = Vec::new();
    v.push(2);
    v.push(4);

    let mut s = &mut v[..];
    s[1] = 8;
}
```



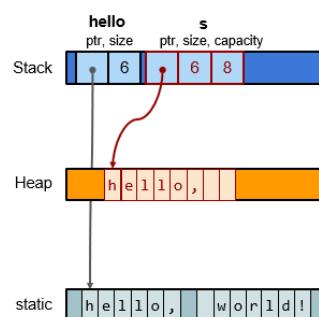
STRINGHE = 2 modi principali per rappresentare le stringhe:

- ✓ **str** → tipo primitivo (array di caratteri immutabili in area statica); non è manipolabile e si accede ad esso solo tramite slice (`&str`; contiene l'indirizzo del primo char e la length) [`&str` può referenziare anche oggetti `String`, permettendo **interoperabilità** tra i 2 formati (tutti i metodi su `&str` sono anche su `&String`)]
- ✓ **String** → oggetti **allocati dinamicamente** (stessi campi di `Vec<T>`)

I valori stringa sono tra doppi apici ("")

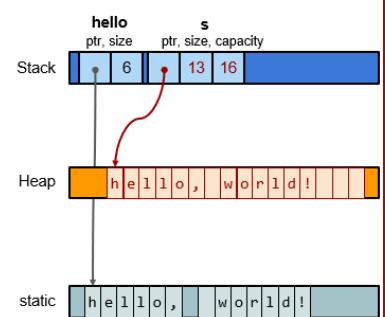
```
fn main() {
    let hello: &str = "hello,";

    let mut s = String::new();
    s.push_str(hello);
    s.push_str(" world!");
}
```



```
fn main() {
    let hello: &str = "hello,";

    let mut s = String::new();
    s.push_str(hello);
    s.push_str(" world!");
}
```



⚠ Alcuni metodi su `String`:

- Creare stringa vuota → `let s0 = String::new();`
- Creare stringa inizializzata → `let s1 = String::from("text");` o `let s2 = "text".to_string();`
- Ricavare `&str` da `String` → `s2.as_str();`
- Aggiunge al fondo → `s3.push_str("This goes to the end");`
- Inserisce alla posizione indicata → `s3.insert_str(0, "This goes to the front");`
- Elimina carattere alla posizione indicata → `s3.remove(4);`
- Svuota stringa → `s3.clear();`
- Crea nuovo String tutto maiuscolo → `let s4 = s1.to_uppercase();`
- Crea nuovo String sostituendo un blocco → `let s5 = s1.replace("some", " more ");`
- Crea nuovo String eliminando spaziature iniziali e finali → `let s6 = s1.trim();`

⚠ Le **ISTRUZIONI** tornano il tipo () [es. `let` o `let mut`], le **ESPRESSIONI** tornano tipo arbitrario [es. `{...}`, `if...else`, `loop...` con `break` e `continue`]

Le **FUNZIONI** sono il centro del programma. Hanno sintassi `fn nome_funzione(argomento1: tipo, argomento2: tipo) {...}` se tornano il tipo (), mentre se tornano tipi diversi la sintassi è `fn nome_funzione(argomento1: tipo, argomento2: tipo) -> tipo Ritorno {...}`. Il valore tornato può essere dopo il `return` oppure può essere l'ultima riga della funzione se non metto ";" alla fine di essa

```
fn find_number(n: i32) -> i32 {
    let mut count = 0;
    let mut sum = 0;
    loop {
        count += 1;
        if count % 5 == 0 { continue; }
        sum += if count % 3 == 0 { 1 } else { 0 };
        if sum == n { break; }

    }
    count
}

fn main() {
    println!("{}", find_number(5));
}
```

```
fn main() {
    'outer: loop {
        println!("Entrato nel ciclo esterno");

        'inner: loop {
            println!("Entrato nel ciclo interno");

            // La prossima istruzione interromperebbe il ciclo interno
            // /break;

            // Così si interrompe il ciclo esterno
            break 'outer;
        }
        // Il programma non raggiunge mai questa posizione
    }
    println!("Terminato il ciclo esterno");
}
```

Le notazioni **a..b** e **c..=d** indicano tutti gli elementi da **a** a **b** (escluso) e tutti gli elementi da **c** a **d** (incluso):

```
fn main() {
    for n in 1..10 {                                // Stampa i numeri da
        1 a 9
        println!("{}", n);
    }

    let names = ["Bob", "Frank", "Ferris"];
    for name in names.iter() {                      // Stampa i tre nomi
        println!("{}", name);
    }

    for name in &names[ ..=1 ] {                  // Stampa i primi due nomi
        println!("{}", name);
    }

    for (i,n) in names.iter().enumerate() { //stampa indici e nomi
        println!("names[{}]: {}", i, n);
    }
}
```

⚠ Da notare che esiste `while` e `for` anche in Rust, ma il `for` esiste solo con `for ... in` (e non esiste la notazione con i 3 campi del C)

Esiste anche l'espressione `match` (che sarebbe una sorta di switch), ma deve essere completo (ovvero non compila se non metto tutti i possibili casi nel `match`)

```
let s = match item {
    0 => "zero",
    10 ..= 20 => "tra dieci e venti",
    40 | 80 => "quaranta o ottanta",
    _ => "altro",
}
```

⚠ Il comando `use std::env::args;` va messo ad inizio programma per consentire di passare dei **parametri da riga di comando**. La libreria `clap` gestisce i parametri passati da linea di comando; inoltre da notare che i commenti per la documentazione inline si mettono con `///`:

```
use clap::Parser;

/// Simple program to greet a person
#[derive(Parser, Debug)]
#[command(version, long_about = None)]
struct Args {
    /// Name of the person to greet
    #[arg(short, long)]
    name: String,
    /// Number of times to greet
    #[arg(short, long, default_value_t = 1)]
    count: u8,
}

fn main() {
    let args = Args::parse();
    for _ in 0..args.count {
        println!("Hello {}!", args.name)
    }
}
```

```
$ demo --help
Simple program to greet a person

Usage: demo[EXE] [OPTIONS] --name <NAME>

Options:
    -n, --name <NAME>      Name of the
                           person to greet
    -c, --count <COUNT>    Number of times
                           to greet [default: 1]
    -h, --help              Print help
    -V, --version           Print version

$ demo --name Me
Hello Me!
```

```
use std::io;

fn main() {
    let mut s = String::new();

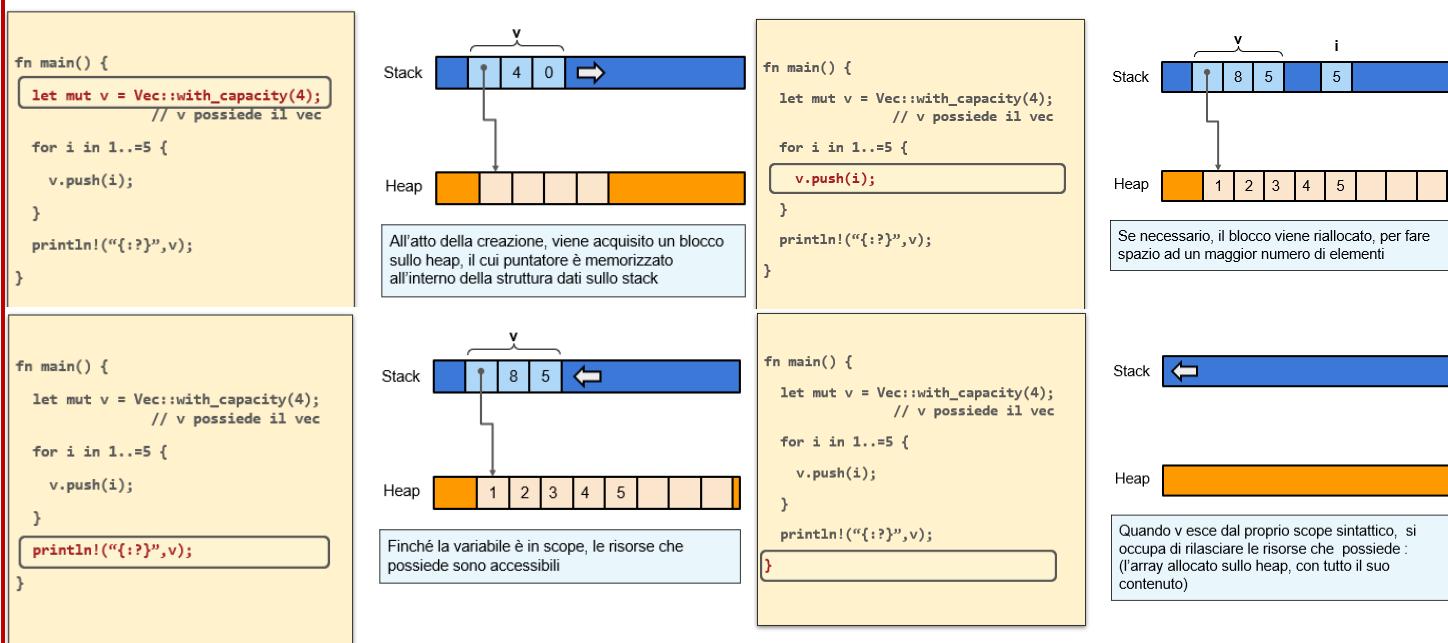
    if io::stdin().read_line(&mut s).is_ok() {
        println!("Got {}", s.trim());
    } else {
        println!("Failed to read line!");
    }

    //alternativamente
    io::stdin().read_line(&mut s).unwrap();
    println!("Got {}", s.trim());
}
```

⚠ L'I/O da console è fatto così →

2.1) POSSESSO (OWNERSHIP)

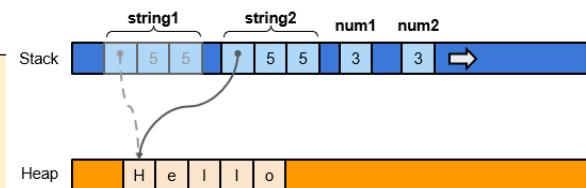
In RUST ogni valore introdotto nel programma è posseduto da 1 sola variabile (come detto prima, controllato dal *borrow checker* e se violazione, no compilazione) [POSSEDERE un valore = essere responsabili del suo rilascio, il quale avviene quando la variabile che lo possiede esce dallo scope o quando le viene assegnato un nuovo valore]



Se una variabile viene assegnata ad un'altra variabile o passata come argomento ad una funzione, il suo contenuto (valore) viene **MOSSO nella destinazione** (e seppur la variabile originale resti allocata fino a fine scope, non si può più accedere al suo valore **in lettura** [errore compilazione], mentre **in scrittura**, se scrivo dentro la variabile originale, questa viene riabilitata e viene riabilitata anche in lettura) [variabile destinazione avrà una **copia** del valore originale]

Alcuni tipi (come quelli numerici) sono “**copiabili**”, ovvero implementano il tratto **Copy**: quando il valore viene assegnato ad un'altra variabile o usato come argomento in una chiamata a funzione, il **valore originale rimane accessibile in lettura**. Vediamo un confronto:

```
let string1 = "Hello".to_string();
let string2 = string1; //da qui in poi, string1 è inaccessibile in lettura
                     //a meno che non venga riassegnato
let num1: i32 = 3;
let num2 = num1;    //nessun vincolo su num1!
```



Alcuni tipi sono “**duplicabili**” con il metodo `clone()`, ovvero implementano il tratto **Clone** (per avere il tratto Copy, un tipo deve avere anche il tratto Clone): viene creato un oggetto duplicato dall'oggetto originale e le loro modifiche non sono più sincronizzate dopo la duplicazione

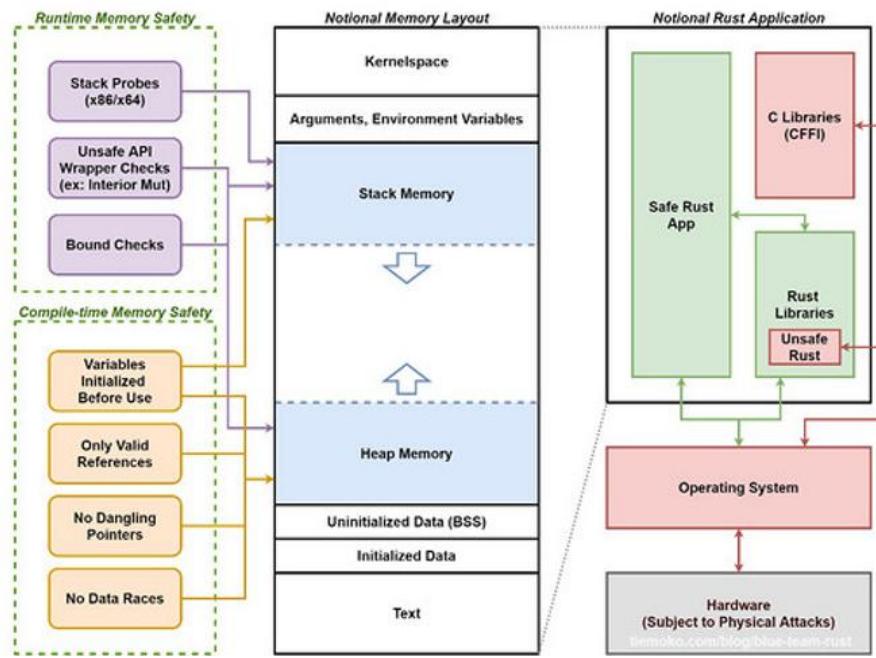
⚠ Riprendendo i **RIFERIMENTI**, questi sono implementati in modo diverso in base al tipo puntato:

- **Puntatore semplice** (se si conosce la dimensione del dato puntato) [classico riferimento]
- **Fat pointer** = puntatore + dimensione (se la dimensione del dato è nota solo a runtime)
- **Puntatore doppio** (se il tipo di dato puntato è noto solo per l'insieme di tratti che implementa)

Il *borrow checker* controlla anche che accessi al riferimento avvengano fino a che il dato contenuto in esso esiste (**no dangling pointer**); il **lifetime** del riferimento è l'insieme delle righe in cui si accede ad esso (può anche essere espresso nella firma del tipo con la sintassi `&'a NomeTipo`, dove il `'` serve a quantificarlo come lifetime) [se un riferimento è valido per tutto il programma si usa la notazione `&'static NomeTipo`]. Per essere lecito ovviamente il **lifetime del riferimento deve essere incluso nel lifetime del valore puntato**

⚠ Una slice non possiede i dati cui fa riferimento (se l'array venisse via, lo slice non esisterebbe più)

Sotto la **disposizione in memoria** di un programma Rust:



3) TIPI COMPOSTI

In C/C++ la **struct** permette di creare un nuovo **TIPO COMPOSTO** con un gruppo di campi accessibili a tutti (public) ed in C++ si poteva usare la **class** (simile alla struct ma con accessibilità customizzabile [public/private/protected]). In entrambi i linguaggi si possono definire anche **enum** (insieme di valori costanti assegnabili) e **union** (permette di usare lo stesso blocco di memoria per rappresentare dati di tipo diverso, alternativamente l'uno all'altro).

In RUST per unire dati eterogenei ci sono le **tuple**, ma se vogliamo associare una semantica/significato particolare ai campi della struttura, esiste la **struct**, cioè un costrutto che permette di rappresentare un blocco di memoria in cui sono disposti consecutivamente una serie di campi il cui nome e tipo sono indicati dal programmatore:

```
struct Player {
    name: String, // nickname
    health: i32,   // stato di salute (in punti vita)
    level: u8,     // livello corrente
}
```

Per convenzione il **nome della struct** comincia con lettera maiuscola e usa **camelCase**. Dopo la dichiarazione (img sopra), la struct va istanziata con la notazione `let [mut] s = Player {name: "Mario".to_string(), health: 25, level: 1};` se i nomi delle variabili contenenti i valori coincidono con i campi della struct (es. name, health, level), posso direttamente usare `let s = Player {name, health, level}.`

Si può istanziare una **nuova struct partendo da un'altra dello stesso tipo** e se ometto dei campi nella nuova struct, verranno presi dalla struct originale (`let s1 = Player {name: "Paolo".to_string(), .. s}.`).

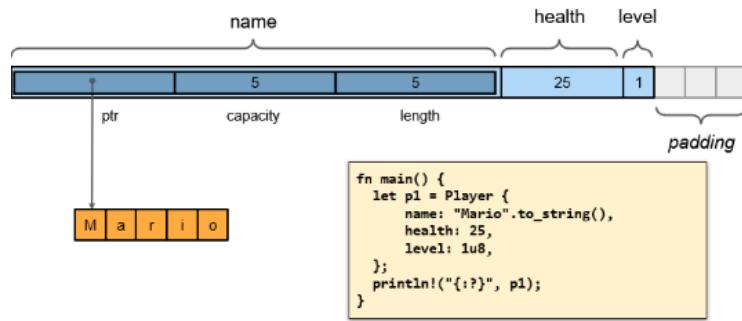
Anche qui si **accede ai campi** della struct con **notazione puntata**, ma a differenza delle tuple (`t.0, t.1...`) si usa il **nome del campo** della struct (e non un indice), ovvero `s.health`; se voglio fare una **scrittura in un campo** della struttura, questa deve essere mutabile (es. `s.level += 1` → funziona solo se `let mut s`)

⚠ Quindi **con ogni nuova struct si definisce un nuovo tipo** (con nome coincidente al nome della struct)

⚠ Si possono anche definire le struct con "**notazione alla tupla**", ovvero con `struct Player(String, i32, u8)` e si possono definire anche struct vuote (analoghe al tipo `()`) con `struct Player` (si possono poi anche quindi dichiarare delle variabili struct con la "notazione alla tupla" → `let mut f = Player("Andrea".to_string(), 90, 3)`)

La disposizione in memoria dei singoli campi è dovuta ottimizzazione (es. allineamento su 2^i); **allineamento e disposizione** di struct sono controllati con `#[repr(...)]` anteposto alla dichiarazione della struct e possiamo usare:

- `std::mem::align_of_val(...)` → per conoscere l'allineamento richiesto da un valore
- `std::mem::size_of_val(...)` → per conoscere la dimensione del valore



⚠ Le struct **non** sono organizzate in una gerarchia di ereditarietà.

Sia la struct nel suo complesso sia i singoli campi che la formano possono essere preceduti da un **modificatore di visibilità**: di base i campi della struct sono **privati** (accessibili solo al codice del modulo corrente e ai suoi sottomoduli), ma li posso rendere pubblici antemponendo loro **pub**. Ciò permette di implementare l'**incapsulamento** ("information hiding").

A differenza di quanto avviene in altri linguaggi, in Rust i **metodi** associati alla struct sono definiti separatamente in un blocco **impl nomeTipo**, dove le funzioni il cui 1° parametro è **self**, **&self** o **&mut self** sono **METODI**, mentre le funzioni senza questo 1° parametro sono dette **funzioni associate** (come costruttori e metodi statici):

Altri linguaggi
(C++, Java, Javascript ES6+, ...)

```

class Something {
    int i;           Dati
    String s;

    void process() {...} Metodi
    int increment() {...}

}
  
```

Rust

```

struct Something {
    i: i32;           Dati
    s: String;
}

impl Something {
    fn process(&self) {...} Metodi
    fn increment(&mut self) {...}
}
  
```

Questi metodi si richiamano con la notazione puntata:

```

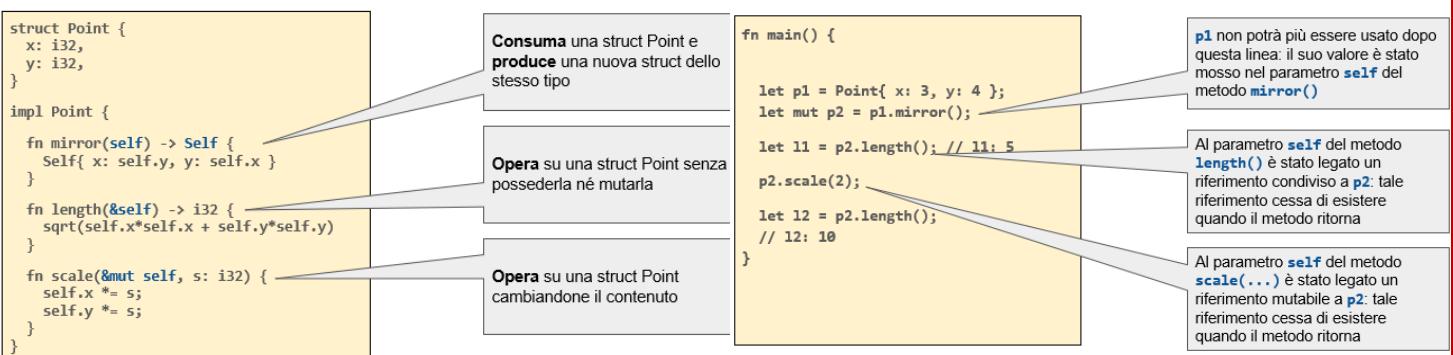
impl str {
    pub const fn len(&self) -> usize //...
}
  
```

```

let str1: &str = "abc";
println!("{}", str1.len());          // 3
println!("{}", str1:len(str1));     // 3
  
```

Il **1° parametro del metodo** definisce il **livello di accesso** che il codice del metodo ha sulla variabile chiamante:

- **self** → ricevitore passato per **movimento** (consumando il contenuto della variabile) [**self: Self**]
- **&self** → ricevitore passato per **riferimento condiviso** (solo lettura) [**self: &Self**]
- **&mut self** → ricevitore passato per **riferimento esclusivo** [**self: &mut Self**]



⚠ In C++ abbiamo i **costruttori**, in **Rust NO**; per favorire l'incapsulamento però, le implementazioni dei metodi spesso includono metodi per inizializzare delle istanze ("fake costruttori") con **pb fn new() -> Self { ... }**. A differenza dei costruttori nei vari linguaggi (dove posso usare lo stesso nome con altri parametri per un altro costruttore in quanto la firma dei metodi è nome + parametri), in Rust **non si possono usare 2 new con diversi parametri**, ma, se mi serve un metodo istanziatore con parametri diversi la convenzione è usare **pb fn with_details(...)** -> Self { ... }

⚠ In C++ abbiamo i **distruttori** (~nomeClasse), i quali permettono l'approccio **RAII** (Resource Acquisition Is Initialization), ovvero le risorse sono incapsulate in una classe in cui il **costruttore** acquisisce le risorse e può

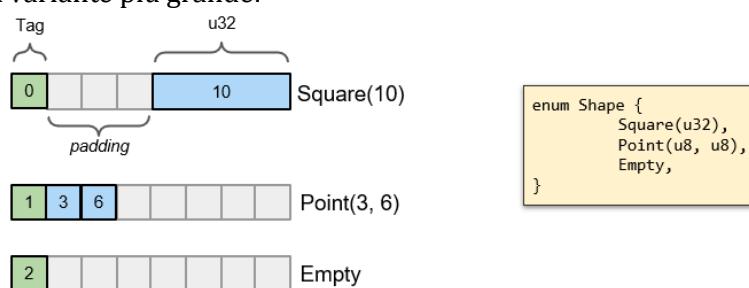
lanciare eccezioni, mentre il **distruttore** rilascia le risorse e non lancia mai eccezioni; ha una gestione automatica della durata delle risorse o il lifetime è connesso al lifetime di un altro oggetto → in questo contesto, la semantica del “**movimento**” garantisce il corretto trasferimento delle risorse, mantenendo la sicurezza del rilascio.

Rust gestisce il rilascio delle risorse di un’istanza sempre usando l’approccio RAII, ma con il tratto **Drop** (contenente la sola funzione `drop(&mut self) -> ()`; possiamo anche farlo manualmente chiamando questa funzione) [il tratto **Drop** è **mutualmente esclusivo** con il tratto **Copy**, ovvero se un tipo implementa uno non può implementare l’altro]

⚠ In C++ si possono usare i metodi **static** (metodi che operano su tutte le istanze della classe [come ad ASE]); l’equivalente di ciò in **Rust** è usare **metodi non indicando come 1° parametro self** (o i suoi derivati) [es. il metodo `new` visto prima]

In Rust si possono usare gli **enum** come in C/C++ e si possono anche legare metodi a tali enum come fossero struct (con la sintassi `impl`); **enum** è definito come tipo **somma** (l’insieme dei valori che può contenere è l’unione dei valori delle singole alternative), mentre **struct** è tipo **prodotto** (l’insieme dei valori è il prodotto cartesiano degli insiemi legati ai singoli campi). In memoria gli enum occupano lo spazio di un int (1Byte) + spazio necessario a contenere la variante più grande:

```
enum HttpResponse {
    Ok,
    NotFound(String),
    InternalError {
        desc: String,
        data: Vec<u8>,
    }
}
```



⚠ L’uso combinato di **enum** e **match** è molto utile per iterare tra i possibili comportamenti dei valori di enum; ma si può anche usare il costrutto `if let <pattern> = <value>` oppure `while let <pattern> = <value>`:

```
enum Shape {
    Square { s: f64 },
    Circle { r: f64 },
    Rectangle { w: f64, h: f64 }
}
```

```
fn compute_area(shape: Shape) -> f64 {
    match shape {
        Square { s } => s*s,
        Circle { r } => r*r*3.1415926,
        Rectangle {w, h} => w*h,
    }
}
```

```
fn process(shape: Shape) {
    // stampa solo se shape è Square...
    if let Square { s } = shape {
        println!("Square side {}", s);
    }
}
```

La tecnica appena vista (**DESTRUZZURAZIONE**) è anche utile per ottenere i singoli campi di una struct e lavorarli come variabili:

```
pub struct Point {
    x: f32,
    y: f32
}

...

let p = Point { x: 5., y: 10. };
...

// la destrutturazione deve rispettare i nomi dei campi
let Point { x, y } = p;

println!("The original point was: ({},{}), x, y);
```

⚠ Se il valore originale non è posseduto (es. è un riferimento) e non è copiabile, occorre far precedere al nome della variabile da assegnare la parola chiave **ref** (eventualmente seguita da **mut**)

Vedremo tra poco che anche in Rust si possono definire **TIPI GENERICI** (es. `Vec<T>` = vettore generico di valori omogenei di tipo T) e ci sono **2 importanti enumerazioni generiche**:

- **Option<T>** → valore di tipo T **opzionale**; ha 2 valori possibili:
 - **Some(T)** = indica valore presente e contiene il valore
 - **None** = indica valore assente
- **Result<T,E>** → rappresenta **alternativamente** un valore di tipo T o un errore di tipo E; si usa per l’esito di una computazione e può valere:
 - **Ok(T)** = successo, valore restituito ha tipo T
 - **Err(E)** = failure, il tipo E è usato per descrivere il fallimento

```

fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        None => None,
        Some(i) => Some(i + 1),
    }
}

```

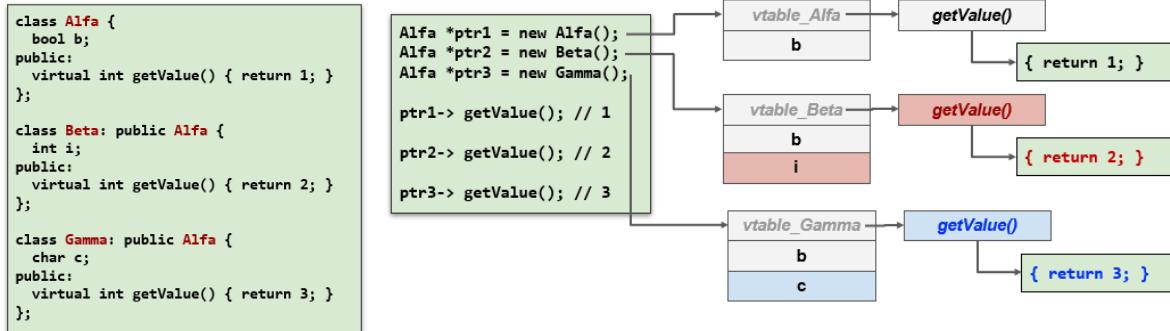
```

fn open_file(n: &str) -> File {
    match File::open(n) {
        Ok(file) => file,
        Err(_) => panic!("error"),
    }
}

```

4) TRATTI & GENERICS (polimorfismo)

POLIFORMISMO = capacità di associare comportamenti comuni ad un insieme di tipi diversi (in Java lo avevamo visto con l'ereditarietà, le **interfacce** e i **generics**). Il **C** non lo supporta di base, mentre il **C++** supporta **ereditarietà** e **metodi virtuali** (metodi chiamati indirettamente, passando attraverso una **VTABLE** [contenente un array con l'indirizzo effettivo dei metodi virtuali che la classe implementa] [quindi costose in termini di memoria e tempo di accesso]):



In C++ si può omettere il corpo di un metodo virtuale dichiarandolo “=0”; questo crea una **funzione virtuale astratta**. Se una classe contiene almeno 1 funzione virtuale astratta è una classe astratta, mentre se contiene solo funzioni virtuali astratte è una **classe astratta pura** (l'equivalente di interfaccia)

I **TRATTI** in Rust sono l'**equivalente delle classi astratte pure/interfacce**: un tratto definisce un insieme di metodi (eventualmente con un'implementazione di default); inoltre, qui in Rust, se si invoca su un valore una funzione relativa ad un tratto [sempre con la sintassi `valore.funzione`], non si ha costo aggiuntivo (eccetto se si crea esplicitamente un riferimento dinamico con `&dyn NomeTratto`).

Un **tratto si definisce** con `trait NomeTratto { fn nomeOperazione(&mut self) -> risultato; ... }`. Una struttura dati (come struct/enum) può dichiarare di **implementare un tratto** con `impl NomeTratto for NomeTipo {...}`

⚠ Alcuni tratti (come Clone e Iter) non vanno importati esplicitamente perché sono parte della libreria standard

```

trait T1 {
    fn returns_num() -> i32; // ritorna un numero
    fn returns_self() -> Self; // restituisce un'istanza del tipo che lo implementa
}

```

Self (come in figura in `returns-self()`) si riferisce al **tipo che implemenerà il tratto**

```

struct SomeType;
impl T1 for SomeType {
    fn returns_num() -> i32 { 1 }
    fn returns_self() -> Self { SomeType }
}

```

```

struct OtherType;
impl T1 for OtherType {
    fn returns_num() -> i32 { 2 }
    fn returns_self() -> Self { OtherType }
}

```

Se una funzione tra quelle definite da un tratto non usa, come 1° parametro, né `self` (né `&self`, `&mut self`, ...) non è legata all'istanza del tipo che la implementa e si invoca con `let var : tipo = NomeTratto::funzione()` (oppure `let var = tipo::funzione()`). Infatti noi definiamo **METODO** una funzione che usa, come 1° parametro, `self (&self, &mut self, ...)` e, a differenza delle funzioni appena viste, si invoca con `valore.metodo()`

Un tratto può avere **1 o + tipi associati** (come ovvio che sia) e si può anche specificare un'implementazione di **default** per una data funzione [img dx]:

```

trait T3 {
    type AssociatedType;
    fn f(arg: Self::AssociatedType);
}

```

```

struct SomeType;
impl T3 for SomeType {
    type AssociatedType = i32;
    fn f(arg: Self::AssociatedType) {}
}

```

```

trait T4 {
    fn f() { println!("default"); }
}

```

```

struct SomeType;
impl T4 for SomeType { }
// uso dell'implementazione di default

```

```

struct OtherType;
impl T3 for OtherType {
    type AssociatedType = &str;
    fn f(arg: Self::AssociatedType) {}
}

```

```

fn main() {
    SomeType::f(1234);
    OtherType::f("Hello, Rust!");
}

```

```

struct OtherType;
impl T4 for OtherType {
    fn f() { println!("Other"); }
}

```

```

fn main() {
    SomeType::f(); // default
    OtherType::f(); // Other
}

```

La notazione `trait Subtrait: Supertrait { ... }` indica che i tipi che implementano **Subtrait** devono implementare anche **Supertrait** (ma le 2 implementazioni sono comunque *indipendenti*):

```
trait Supertrait {
    fn f(&self) {println!("In super");}
    fn g(&self) {}
}

trait Subtrait: Supertrait {
    fn f(&self) {println!("In sub");}
    fn h(&self) {}
}

struct SomeType;
impl Supertrait for SomeType {}
impl Subtrait for SomeType {}

fn main() {
    let s = SomeType;
    s.f(); // Errore: chiamata ambigua
    <SomeType as Supertrait>::f(&s);
    <SomeType as Subtrait>::f(&s);
}
```

Come abbiamo citato prima, l'invocazione dei metodi di un tratto può avvenire con **INVOCAZIONE**:

- **Statica** → se il tipo del valore è noto, il compilatore può identificare l'indirizzo della funzione da chiamare e generare il codice corrispondente senza alcuna penalità
- **Dinamica** → se si ha un puntatore ad un valore di cui il compilatore sa solo che implementa un dato tratto, si può fare una chiamata indiretta passando per una VTABLE (variabili destinate a contenere puntatori ad un valore che implementa un tratto sono annotate con `dyn`) [vedi sopra]

```
trait Print {
    fn print(&self);
}
struct S { i: i32 }
impl Print for S {
    fn print(&self){
        println!("S {}", self.i);
    }
}

fn process(v: &dyn Print){
    v.print();
}

fn main() {
    process(&S{i: 0});
}
```

Questi riferimenti/puntatori (`&S as &dyn Print`) ai **tipi-tratto** vengono detti **oggetti-tratto** e sono implementati con *fat pointer*

Vediamo quindi **TRATTI DELLA LIBRERIA STANDARD**:

- **Eq o PartialEq** → `==` e `!=` tra 2 istanze di un dato tipo; metodi:
 - `eq(&self, other: &RHS)` → `bool` → `==`
 - `ne(&self, other: &RHS)` → `bool` → `!=` (implementazione di default di solito)
- **Ord o PartialOrd** → `<`, `>`, `<=` e `=>` tra 2 istanze di un dato tipo

```
enum Ordering {
    Less,
    Equal,
    Greater,
}
trait PartialOrd<Rhs = Self>: PartialEq<Rhs>
where Rhs: ?Sized, {
    fn partial_cmp(&self, other: &Rhs) -> Option<Ordering>;
    // metodi con implementazione di default
    fn lt(&self, other: &Rhs) -> bool;
    fn le(&self, other: &Rhs) -> bool;
    fn gt(&self, other: &Rhs) -> bool;
    fn ge(&self, other: &Rhs) -> bool;
}

trait Ord: Eq + PartialOrd<Self> {
    fn cmp(&self, other: &Self) -> Ordering;
    // implementazione di default
    fn max(self, other: Self) -> Self;
    fn min(self, other: Self) -> Self;
    fn clamp(self, min: Self, max: Self) -> Self;
}
```

- **Add, Sub, Mul, Div, Rem, BitAnd, BitXor, Shl, Shr, Neg, Not** → `+, -, *, /, %, &, ^, <<, >>, -` (unario), !
- **Display** → stampa a schermo (con metodi come `println!` e `format!`) usando `{}` per variabili
- **Debug** → stampa di debug a schermo usando `{:?}`, per rappresentare il contenuto di strutture complesse
- **Clone** → creare un duplicato di un valore dato (per **trasformare** un riferimento `&T` in un valore posseduto `T`)
- **Copy** → **Subtrait** di **Clone**; non trasforma, ma la copia ottenuta è solo l'esatto **duplicato** (può essere implementato solo da strutture dati i cui campi implementino il tratto stesso) [la sua presenza cambia la semantica delle assegnazioni: quello che normalmente sarebbe un **"movimento"** (ovvero che rende inaccessibile il valore originale) diventa in realtà una **"copiaatura"** a tutti gli effetti)]
- **Drop** → il suo metodo `drop(&mut self)` è come il distruttore in C++; è mutualmente esclusivo con **Copy**
- **Index o IndexMut** → permette di usare una struttura dati **come un array**, abilitando la notazione `t[i]` (verrà trasformata poi dal compilatore in `*t.index(i)` in **lettura** e `*t.index_mut(i)` in **scrittura**) [es. `Vec<T>`]

```
trait Index<Idx> {
    type Output: ?Sized;
    fn index(&self, index: Idx) -> &Self::Output;
}

trait IndexMut<Idx>: Index<Idx> {
    fn index_mut(&mut self, index: Idx) -> &mut Self::Output;
}
```

- **Deref** o **DerefMut** → permette di trattare una struttura dati **come un puntatore**, abilitando la sintassi `*t` (verrà trasformata poi dal compilatore in `*(t.deref())`) se vogliamo un valore **immutable** e `*(t.deref_mut())` se vogliamo un valore **mutable** [entrambi `Self::Target`]:

```
trait Deref {
    type Target: ?Sized;
    fn deref(&self) -> &Self::Target;
}

trait DerefMut: Deref {
    fn deref_mut(&mut self) -> &mut Self::Target;
}
```

⚠ Questi 2 tratti permettono la conversione automatica da `&Self` a `&Self::Target` (e da `&mut Self` a `&mut Self::Target`) → questa proprietà è detta **deref coercion** e consente l'interoperabilità tra `&String` e `&str`, per cui è lecito invocare su un valore di tipo `String` metodi definiti per `str`

- **RangeBounds<T>** → permette di definire **intervalli** di valori attraverso l'operatore `..` (è lo stesso che consente di definire intervalli in tipi base di Rust con `a..b` o `a..=b`)

```
pub trait RangeBounds<T>{
    fn start_bound(&self) -> Bound<&T>;
    fn end_bound(&self) -> Bound<&T>;
    fn contains<U>(&self, item: &U) -> bool { ... }
}

pub enum Bound<V>{
    Included(V),
    Excluded(V),
    Unbounded,
}
```

- **From** e **Into** → permettono **conversione** tra tipi (`T: From<i32>` equivale a `i32: Into<T>`) [speculari] [From però non è sempre simmetrico: `T→U` non garantisce `U→T`]

⚠ Ci sono anche i tratti **TryFrom** e **TryInto** se c'è possibilità di fallire la conversione (ritornano `Result<T, E>`); c'è anche il tratto **FromStr** per gestire la conversione da stringa con eventuale fallimento:

```
trait From<T>: Sized {
    fn from(other: T) -> Self;
}

trait Into<T>: Sized {
    fn into(self) -> T;
}

pub trait TryFrom<T>: Sized {
    type Error;
    fn try_from(value: T) -> Result<Self, Self::Error>;
}

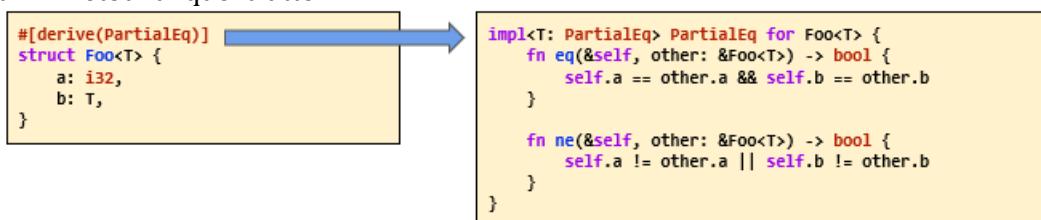
pub trait TryInto<T>: Sized {
    type Error;
    fn try_into(self) -> Result<T, Self::Error>;
}
```

```
pub trait FromStr {
    type Err;
    fn from_str(s: &str) -> Result<Self, Self::Err>;
}
```

- **Error** → in Rust gli errori non vengono lanciati, ma tornati in `Result<T, E>`; dato che il tipo generico `E` può assumere qualsiasi valore, è preferibile usare solo tipi che implementano il tratto **Error**

```
pub trait Error: Debug + Display {
    fn source(&self) -> Option<&(dyn Error + 'static)> { ... }
    fn backtrace(&self) -> Option<&Backtrace> { ... }
    fn description(&self) -> &str { ... }
    fn cause(&self) -> Option<&dyn Error> { ... }
}
```

Nonostante si possano implementare a mano tutti i metodi richiesti da un dato tratto, a volta conviene **affidarsi al compilatore** → se scriviamo davanti ad una struct/enum `#[derive(NomeTratto)]`, il compilatore aggiunge alla struct/enum i metodi di quel tratto:



Sebbene una forte tipizzazione sia giusta per garantire robustezza al codice, Rust (come C++) fornisce la possibilità di usare i **GENERICs** (scrivere funzioni e tipi senza sapere su che tipo di dato operino):

```
fn max<T>(
    t1: T,
    t2: T) -> T where T: Ord {
    return
        if t1 < t2 { t2 }
        else { t1 }
}
```

Rust

Come vediamo qui sotto, se una **struct generica** implementa dei metodi, i nomi delle **meta-variabili** ed i **vincoli** cui sono soggette vengono ripetuti nel blocco `impl`:

```
struct MyStruct<T> where T: SomeTrait {
    foo: T,
    //...
}
```

```
impl<T> struct MyStruct<T>
where T: SomeTrait {
    fn process(&self) { ... }
}
```

Ci sono 2 sintassi possibili per **specificare i vincoli** sui tipi generici:

- ❖ versione **compatta** → `<T: NomeTratto>`
- ❖ versione **estesa** → `<T> ... where T: NomeTratto`

⚠ Si può anche definire un “**tratto generico**”, ovvero i cui metodi ricevono/restituiscono valori generici (eventualmente vincolati da altri tratti); inoltre, il codice usando struct generiche al posto dei tratti (e della sintassi `&dyn NomeTratto`) è solitamente più efficiente (non passo dalla VTABLE perché il compilatore conosce il tipo concreto in fase di monomorfizzazione)

5) LIFETIME (dei riferimenti) e CHIUSURE (funzioni lambda)

Se una funzione riceve un parametro riferimento (`mut` o non `mut`), il **LIFETIME DEL RIFERIMENTO** diventa parte della firma della funzione (`fn f(p: &i32) { ... } ⇒ fn f<'a>(p: &'a i32) { ... }`) [è il compilatore che cambia la nostra scrittura con questa più complessa]; se invece riceve **più riferimenti**, potrebbe essere necessario indicare se i loro lifetime siano:

- **vincolati al più breve** → `fn f<'a>(p1: &'a i32, p2:&'a i32) { ... }`
- **disgiunti** → `fn f<'a, 'b>(p1: &'a i32, p2:& 'b i32) { ... }`

⚠ Alcuni problemi sono dati da funzioni che restituiscono un riferimento, preso da una delle strutture dati in input: se la funzione memorizza il riferimento ricevuto in input in una struttura dati, il compilatore pensa che il **lifetime della struttura deve essere incluso** (o coincidente) con il **lifetime del riferimento**.

```
fn f(s: &str, v: &mut Vec<&str>) {  
    v.push(s);  
}
```

```
fn f<'a>(s: &'a str, v: &'a mut Vec<&str>) {  
    v.push(s);  
}
```

```
error: lifetime mismatch  
1 | fn f(s: &str, v: &mut Vec<&str>) {  
|   ----  
|   these two types are declared with different lifetimes...  
2 |     v.push(s);  
|     ^ ...but data from `s` flows into `v` here
```

```
error: explicit lifetime required in the type of `v`  
1 | fn f<'a>(s: &'a str, v: &'a mut Vec<&str>) {  
|   ----- help: add explicit  
|   lifetime ``a` to the type of `v`: ``&'a mut Vec<&'a str>`  
2 |     v.push(s);  
|     ^ lifetime ``a` required
```

⚠ Se una funzione ha **2 o + riferimenti** in input e **1 riferimento** in output, il compilatore capisce da quale prende il riferimento in output (qui l'inserimento degli identificatori nella firma della funzione risolve)

Gli scopi degli **identificatori di lifetime** sono **2**:

- per il **codice chiamante** la funzione → indicare su quale, tra gli indirizzi in input, è basato l'output
- per il **codice chiamato** (nella funzione) → garantire il return di soli indirizzi a cui si può accedere per (almeno) il lifetime indicato

Infatti spesso (soprattutto in strutture nested), dobbiamo **specificare il lifetime della struttura rispetto ai lifetime dei riferimenti** al suo interno (Rust di default assegna ad ogni riferimento presente in una struttura dati o tra i parametri di una funzione un tempo di vita distinto)

⚠ Se funzione **restituisce un riferimento o un tipo che contiene** (direttamente/indirettamente) **un riferimento**, dobbiamo **disambiguare il lifetime del valore in return** (a meno che non ho solo 1 parametro in input dotato di lifetime, in quel caso sarà lo stesso lifetime)

⚠ Se ho un metodo con **&self**, Rust assume che il lifetime dell'output sia quello del riferimento a self

Parliamo ora delle **CHIUSURE (funzioni lambda)**: la programmazione funzionale introduce le **funzioni di ordine superiore** (funzioni in cui input/output sono a loro volta funzioni; quindi posso trattare le funzioni come normali dati [posso anche memorizzarle in variabili]) →

```
fn f1(i: i32, d: f64) -> f64 {  
    return i as f64 * d;  
}  
  
let ptr: fn(i32, f64) -> f64;  
ptr = f1; //assegno il puntatore  
ptr(2, 3.14); // chiamo la funzione
```

⚠ In C++ esiste anche l'**oggetto funzionale** (ovvero un oggetto con dentro la funzione **operator()** chiamabile come una funzione e che fa le cose scritte dentro **operator()**; si possono anche avere più operator() cambiando i tipi nei parametri)

La notazione degli oggetti funzionali però è verbosa, quindi noi usiamo le **funzioni lambda** in Rust (**closure**) che hanno sintassi **let f = |v| {v+1}** (che equivale in JavaScript a **const f = (v) => v+1**). Si può anche passare una funzione lambda come **argomento** di una funzione da invocare o usare una funzione lambda come **return**.

```
Rust
fn ret_fun() -> fn(i32) -> i32 {
    return |x|{ x+1 };
}
```

Il compilatore Rust trasforma le lambda (closure) in **tuple** con tanti campi quante sono le **variabili libere** (quelle dentro **|...|**). Questa tupla implementa 1 dei tratti funzionali **FnOnce**, **FnMut**, **Fn**.

Tutte le variabili libere nel corpo di una closure sono catturate per riferimento (il compilatore crea in automatico un **prestito in lettura** **&**) (se le vogliamo con **&mut** dobbiamo dichiarare la variabile a cui assegno la lambda con **let mut variabile = |...|{...}**) (se vogliamo che la closure acquisisca il **possesso dei valori** contenuti nelle variabili libere devo fare **let variabile = move |...| {...}**)

Rust definisce quei **3 tratti sopra citati per le closures**:

```
trait FnOnce<Args> {
    type Output;
    fn call_once(self, args: Args) -> Self::Output;
}

trait FnMut<Args>: FnOnce<Args> {
    fn call_mut(&mut self, args: Args) -> Self::Output;
}

trait Fn<Args>: FnMut<Args> {
    fn call(&self, args: Args) -> Self::Output;
}
```

- **FnOnce<Args>** se consuma uno o più valori come parte della propria esecuzione:

```
let range = 1..10;
let f = || range.count();
let n1 = f(); // 10
let n2 = f(); // Errore di compilazione: l'intervallo è stato consumato
```

- **FnMut<Args>** può essere invocata più volte, ma ha catturato una o più variabili in modo esclusivo (**&mut**):

```
let mut sum = 0;
let mut f = |x| sum += x;
f(5); // ok, sum: 5
f(7); // ok, sum: 12
```

- **Fn<Args>** si limita ad accedere in sola lettura alle variabili libere (**&**)

```
let s = "hello";
let f = |v| v < s;
f("world"); // false
f("bye"); // true
```

Si può anche implementare una **funzione che accetta come parametro una closure** tramite i **Generics**:

```
fn higher_order_function<F, T, U>(f: F) where F: Fn(T) -> U {
    // ... codice che usa f...
}
```

Oppure una **funzione che ritorna una closure**:

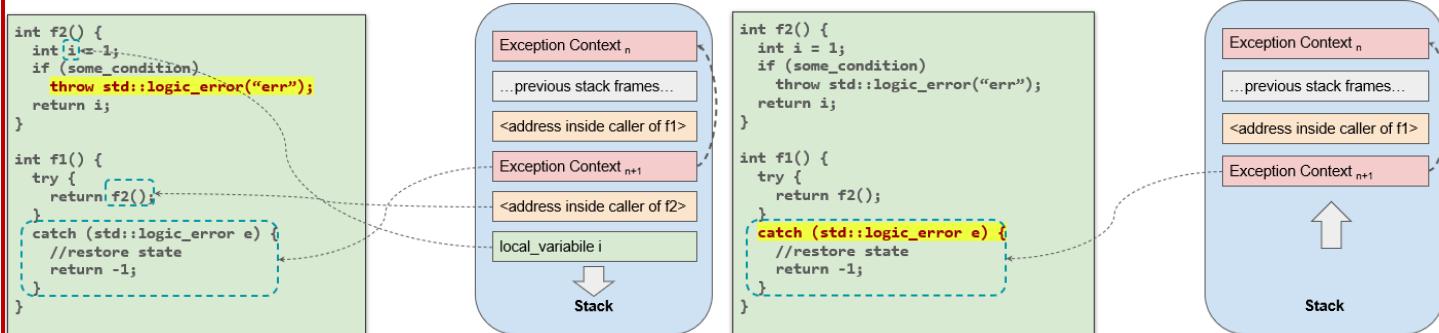
```
fn function_generator<T>(v: T) -> impl Fn() -> T where T: Clone {
    return move || { v.clone() };
}
```

6) ERRORI

Indipendentemente dalle cause che li hanno prodotti, i fallimenti (**FAILURE**) possono essere classificati in:

- **Recuperabili** = non hanno compromesso lo stato del programma (posso attivare **ripristino**)
- **Non recuperabili** = causano alterazione impredicibile dello stato oppure non permettono di procedere ulteriormente (va **terminato** il processo/programma con **panic!(...)** → contrae lo stack, distrugge con **drop()** tutte le variabili che implementano il tratto **Drop** fino alla terminazione del thread corrente [**se thread == programma principale**, allora processo **termina con codice d'errore**, altrimenti **continua**])

Questi possibili errori rendono il **flusso di esecuzione del programma complesso**; per questo i linguaggi moderni hanno costrutti per la **GESTIONE DELLE ECCEZIONI** (il C no, ma il **C++ ha introdotto try/catch/throw**); **Rust** non usa il concetto di eccezione, ma come abbiamo già visto, offre tipi algebrici come **Result<T,E>** e **Option<T>** per esprimere gli esiti delle computazioni (oltre a **panic!(...)** che interrompe il thread corrente e offre una descrizione di cosa è successo al suo interno). Vediamo un esempio di **flusso di eccezioni nello stack di C++:**



⚠ In C++ ci sono limiti della gestione delle eccezioni: il compilatore non sa dove vengano date eccezioni; le eccezioni bloccano l'esecuzione del codice e riportano tipicamente 1 solo tipo di errore; per permettere la corretta contrazione dello stack e il ritorno al blocco try più recente, va imposta una certa sovrastruttura allo stack e al contesto di esecuzione.

In Rust, come dicevamo, posso usare:

- **Option<T>** → si può usare con l'operatore `?` (a condizione che la funzione in cui viene adottato abbia return type `Option<U>` con `U` qualsiasi); racchiude 2 alternative:
 - **Some<T>** = descrive il **return atteso**
 - **None** = **assenza** di risultato (non descrive le cause)
- **Result<T,E>** → `T` = **successo** (tipo di ritorno), `E` = **errore**; è correlato a **Option<T>**

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

```
fn read_file(name: &str) -> Result<String, io::Error> {
    let r1 = File::open(name);
    let mut file = match r1 {
        Err(why) => return Err(why),
        Ok(file) => file,
    };
    let mut s = String::new();
    let r2 = file.read_to_string(&mut s);
    match (r2) {
        Err(why) => Err(why),
        Ok(_) => Ok(s),
    }
}
```

Result offre i metodi:

- **ok(self)** = restituisce `Option<T>`, che ha quindi valore in caso di **successo** (il dato di return viene quindi mosso da `Result`, ovvero diventa inaccessibile da lì)
- **err(self)** = restituisce `Option<E>`, che ha quindi valore in caso di **errore** (anche qui movimento)
- **is_ok(self)** = dice se successo
- **is_err(self)** = dice se errore
- **map(self, op: F) -> Result<U,E>** = **applica la funzione al valore contenuto nel return se è ok**, altrimenti lascia l'errore invariato
- **contains(&self, x: &U)** = ritorna **true se risultato valido** e contiene un **valore = all'argomento**

- `unwrap(self) -> T` = ritorna il valore contenuto se valido (tipo `T`), ma **invoca panic!(...)** se errore; se usato con `expect(...)`, questo metodo indica una `String` che sarà usata al posto del messaggio standard generato da `unwrap` in caso di errore

Anche `Result` permette di usare l'**operatore ?** (ritorna il valore `v` racchiuso in `Ok(v)` nell'enum se successo, mentre se errore avrà `Err(e)` e la funzione corrente termina ritornando il valore `e`):

```
fn read_file(name: &str) -> Result<String, io::Error> {
    let mut file = File::open(name)?;
    let mut s = String::new();
    file.read_to_string(&mut s)?;
    Ok(s)
}
```

Se una funzione produce **errori eterogenei**, bisogna propagare i diversi errori così da poter specializzare le contromisure; la libreria standard di Rust implementa il tratto generico `From` che converte qualsiasi valore che implementi il tratto `Error` in `Box<dyn error::Error>` (con `sintassi impl<E: error::Error> From<E> for Box<dyn error::Error>`). Si può poi risalire allo **specifico errore** con il `downcast_ref()` (a patto che si conosca l'implementazione della funzione che genera gli errori). **Esempio** (gestione errori eterogenei):

```
fn sum_file(path: &Path) -> Result<i32, Box<dyn error::Error>> {
    let mut file = File::open(path)?; // io::Error -> Box<dyn error::Error>
    let mut contents = String::new();
    file.read_to_string(&mut contents)?;
    let mut sum = 0;
    for line in contents.lines() {
        sum += line.parse()?; // ParseIntError -> Box<dyn error::Error>
    }
    Ok(sum)
}
fn handle_sum_file_errors(path: &Path) {
    match sum_file(path) {
        Ok(sum) => println!("sum is {}", sum),
        Err(err) => {
            if let Some(e) = err.downcast_ref::() {...} // tratto io::Error
            else if let Some(e) = err.downcast_ref::() {...} // tratto ParseIntError
            else { unreachable!(); } // non può capitare
        }
    }
}
```

Per **propagare errori eterogenei senza forzare il sistema dei tipi**, si posso usare **errori custom**: tutti questi errori devono implementare il tratto `Error` (e conseguentemente anche `Display` e `Debug`); usando un `enum`, posso racchiudere i diversi tipi di errore (che gestirò con il `match`); va anche implementato il tratto `From` per convertire i diversi errori nel tipo custom da propagare. Vediamo lo stesso esempio di prima, ma con questo approccio:

```
#[derive(Debug)]
enum SumFileError {
    Io(io::Error),
    Parse(ParseIntError),
}
```

```
impl From<io::Error> for SumFileError {
    fn from(err: io::Error) -> Self { SumFileError::Io(err) }
}
impl From<ParseIntError> for SumFileError {
    fn from(err: ParseIntError) -> Self { SumFileError::Parse(err) }
}
impl fmt::Display for SumFileError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match self {
            SumFileError::Io(err) => write!(f, "IO error: {}", err),
            SumFileError::Parse(err) => write!(f, "Parse error: {}", err),
        }
    }
}
impl error::Error for SumFileError {
    fn source(&self) -> Option<&(dyn error::Error + 'static)> {
        Some(match self {
            SumFileError::Io(err) => err,
            SumFileError::Parse(err) => err,
        })
    }
}
```

```
fn sum_file(path: &Path) -> Result<i32, SumFileError> {
    let mut file = File::open(path)?;
    let mut contents = String::new();
    file.read_to_string(&mut contents)?;
    let mut sum = 0;
    for line in contents.lines() {
        sum += line.parse()?;
    }
    Ok(sum)
}
fn handle_sum_file_errors(path: &Path) {
    match sum_file(path) {
        Ok(sum) => println!("the sum is {}", sum),
        Err(SumFileError::Io(err)) => {...},
        Err(SumFileError::Parse(err)) => {...},
    }
}
```

⚠️ Un aiuto con i tipi che implementano il tratto `Error` viene dal crate `thiserror`: offre un'implementazione della derive specializzata per il tratto `Error`; inoltre, definendo un enum/struct/unit preceduto da `#[derive(Error, Debug)]` si ottiene **implementazione automatica dei tratti**:

- **Display**, se metto `#[error("Messaggio con formato")]` davanti alla dichiarazione dell'enum/struct/unit
- **From**, se metto `#[from]` davanti ad un campo che implementa il tratto `Error`

```
[dependencies]
thiserror = "1.0"
```

```
#[derive(Error, Debug)]
enum SumFileError {
    #[error("IO error {0}")]
    Io(#[from] io::Error),
    #[error("Parse error {0}")]
    Parse(#[from] ParseIntError),
}
```

⚠️ Un altro aiuto viene dal crate `anyhow`, il quale definisce l'oggetto-tratto `anyhow::Error` che semplifica la gestione idiomatica degli errori; infatti tramite i metodi `context(...)` e `with_context(...)` permette di aggiungere una descrizione per contestualizzare l'errore.

7) ITERATORI e COLLEZIONI

Un **ITERATORE** è una struttura dati con `stato`, in grado di **generare una sequenza di valori**; in Rust `next() -> Option<E>` combina il verificare se ci sono altri valori da generare e accedere al valore successivo. Gli iteratori si sovrappongono concettualmente ai **cicli for/while**, ma permettono **compattezza**, **leggibilità** e **efficienza** (sono però *"pigri"*, ovvero generano/prelevano il valore successivo solo quando questo viene richiesto) [inoltre **si può prelevare un iteratore da un altro iteratore**]:

```
let v1 = vec![1,2,3,4,5,6];
let mut v2 = Vec::<String>::new();

for i in 0..v1.len() {
    if v1[i] % 2 != 0 { continue; }
    v2.push(format!("a{}", v1[i]))
}
println!("{}: {:?}", v2); // ["a2", "a4", "a6"]
```

```
let v1 = vec![1,2,3,4,5,6];
let mut v2: Vec<String>;
```

- v2 = v1.**iter()**
- .**filter(|val|{ *val % 2 == 0})**
- .**map(|val| format!("a{}", val))**
- .**collect();**

```
println!("{}: {:?}", v2); // ["a2", "a4", "a6"]
```

⚠️ In C++ gli iteratori sono una versione generalizzata dei *puntatori* (sfruttano il concetto di “visita” di un array); inoltre, un iteratore è definito in modo *Implicito* (senza assegnargli un tipo specifico): quindi ogni contenitore mette a disposizione il proprio tipo specifico di iteratore. La std lib offre varie classi contenitore (`std::array<T>`, `std::list<T>`, `std::vector<T>`), ciascuna con una diversa strategia di implementazione.

In Rust, un **ITERATORE** è una qualsiasi struttura dati che implementa il tratto `std::iter::Iterator` (sx); un tipo può dire di poter essere **esplorato tramite un iteratore** implementando il tratto `std::iter::IntoIterator` (dx):

```
impl<const FROM:isize, const TO:isize> Iterator for MyRangeIterator<FROM,TO> {
    type Item = isize;
    fn next(&mut self) -> Option<Self::Item> {
        if FROM < TO {
            if self.val < TO {
                let ret = self.val;
                self.val += 1;
                Some(ret)
            } else {
                if self.val > TO {
                    let ret = self.val;
                    self.val -= 1;
                    Some(ret)
                } else {
                    None
                }
            }
        } else {
            None
        }
    }
}

struct MyRange<const FROM: isize, const TO: isize> {
    val: isize
}

impl<const FROM: isize, const TO: isize> IntoIterator for MyRange<FROM, TO> {
    type Item = isize;
    type IntoIter = MyRangeIterator<FROM, TO>;
    fn into_iter(self) -> Self::IntoIter {
        MyRangeIterator::<FROM, TO>::new()
    }
}

struct MyRangeIterator<const FROM: isize, const TO: isize> {
    val: isize
}

impl<const FROM: isize, const TO: isize> MyRangeIterator<FROM, TO> {
    fn new() -> Self {
        MyRangeIterator{ val: FROM }
    }
}
```

Proprio per i vantaggi spiegati prima, **ogni ciclo for viene trasformato dal compilatore Rust in codice basato sugli iteratori**. I **contenitori della libreria standard** mettono a disposizione **3 metodi** per ricavare un iteratore ai dati contenuti al loro interno:

- **`iter()`** → restituisce oggetti di tipo `&Item` e non consuma il contenuto del contenitore
- **`iter_mut()`** → restituisce oggetti di tipo `&mut Item` e permette modifica di elementi nel contenitore

- **into_iter()** → restituisce oggetti di tipo **Item** estraendoli dal contenitore (prende possesso del contenitore)

E' poi comune per tali contenitori dichiarare **3 implementazioni distinte** del tratto **IntoIterator**:

- 1 per il tipo **Container** (quello effettivo), che richiama il metodo **into_iter()**
- 1 per il tipo **&Container**, che richiama il metodo **iter()**
- 1 per il tipo **&mut Container**, che richiama il metodo **iter_mut()** [in alcuni casi non presente]

```
let mut v = vec![String::from("a"), String::from("b"), String::from("c")];

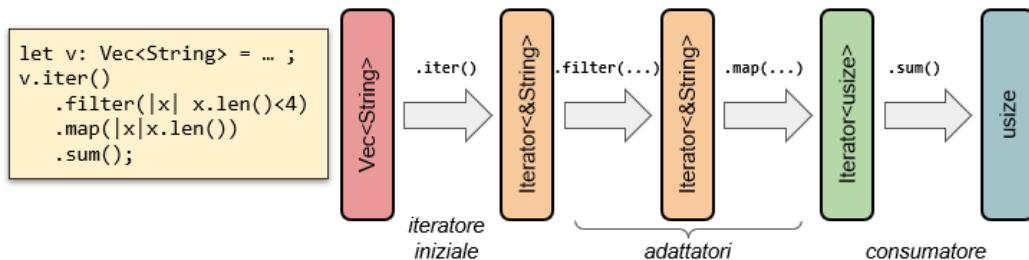
for s in &v {
    println!("{}", s); // s: &String
}

for s in &mut v {
    s.push_str("1"); // s: &mut String - Modifico il contenuto del vettore
}

for s in v {
    println!("{}", s); // s: String - invalido il contenuto del vettore
}
```

⚠ **into_iter()** appartiene anche al tratto **Iterator** (oltre che IntoIterator) e qui restituisce l'iteratore stesso

Il tratto **Iterator** definisce molti metodi che consumano un iteratore e ne derivano uno differente, in grado di offrire funzionalità ulteriori:



ADATTATORI:

- **map<B, F>(self, f: F) -> Map<Self, F>**
 - applica la closure ricevuta come argomento su ogni elemento dell'iteratore in return
- **filter<P>(self, predicate: P) -> Filter<Self, P>**
 - ritorna un iteratore con solo gli elementi per i quali la closure ricevuta come argomento è true
- **filter_map<B, F>(self, f: F) -> FilterMap<Self, F>**
 - fa filter+map (l'iteratore risultante conterrà solo elementi per i quali la closure ritorna Some(B))
- **flatten(self) -> Flatten<Self>**
 - ritorna un iteratore dal quale sono state rimosse le strutture annidate
- **flat_map<U, F>(self, f: F) -> FlatMap<Self, U, F>**
 - fa map+flatten (esegue la closure ricevuta e rimuove le strutture annidate)
- **take(self, n: usize) -> Take<Self>**
 - ritorna un iteratore con al massimo i primi n elementi dell'iteratore originale (meno, se l'iteratore originale non contiene abbastanza elementi)
- **take_while<P>(self, predicate: P) -> TakeWhile<Self, P>**
 - conserva tutti gli elementi fino a quando la closure ritorna true
- **skip(self, n: usize) -> Skip<Self>**
 - ritorna un iteratore senza i primi n elementi dell'iteratore originale (se si raggiunge la fine torna un iteratore vuoto)
- **skip_while<P>(self, predicate: P) -> SkipWhile<Self, P>**
 - esclude tutti gli elementi fino a quando la closure ritorna false
- **peekable(self) -> Peekable<Self>**
 - ritorna un iteratore su cui si possono chiamare i metodi **peek()** e **peek_mut()** per accedere al valore successivo senza consumarlo
- **fuse(self) -> Fuse<Self>**
 - ritorna un iteratore che termina dopo il primo None
- **rev(self) -> Rev<Self>**
 - ritorna iteratore originale in reverse

- **inspect<F>(self, f: F) -> Inspect<Self, F>**
 - o ogni volta che riceve una richiesta, **preleva un elemento dall'iteratore a monte** e la passa sia alla **funzione** (che **lo ispeziona**) che al consumatore a valle
- **chain<U>(self, other: U) -> Chain<Self, <U as IntoIterator>::IntoIter>**
 - o prende come argomento un iteratore e lo **concatena all'originale**, ritorna uno nuovo iteratore
- **enumerate(self) -> Enumerate<Self>**
 - o ritorna un iteratore che restituisce una tupla formata dall'indice dell'iterazione e dal valore (**i, val**)
- **zip<U>(self, other: U) -> Zip<Self, <U as IntoIterator>::IntoIter>**
 - o **combina 2 iteratori** in un nuovo **iteratore** con elementi le coppie dei valori dei primi 2 iteratori attaccate
- **by_ref(&mut self) -> &mut Self**
 - o prende in prestito un iteratore **senza consumarlo** (lascia intatto il possesso dell'originale)
- **copied<'a, T>(self) -> Copied<Self>**
 - o ritorna un nuovo iteratore, tutti gli **elementi dell'iteratore originale vengono copiati**
- **cloned<'a, T>(self) -> Cloned<Self>**
 - o ritorna un nuovo iteratore, tutti gli **elementi dell'iteratore originale vengono clonati**
- **cycle(self) -> Cycle<Self>**
 - o raggiunta la fine di un iteratore riparte dall'inizio (**cicla all'infinito**)

CONSUMATORI:

- **for_each<F>(self, f: F)**
 - o applica la **closure ricevuta su tutti gli elementi** dell'iteratore
- **try_for_each<F, R>(&mut self, f: F) -> R**
 - o applica una closure che può fallire su tutti gli elementi dell'iteratore, **si ferma dopo il 1° fallimento**
- **collect(self) -> B**
 - o **trasforma un iteratore in una collezione**
- **nth(&mut self, n: usize) -> Option<Self::Item>**
 - o ritorna **n_i elemento** dell'iteratore
- **all<F>(&mut self, f: F) -> bool**
 - o **verifica che la closure ricevuta restituisca true per tutti gli elementi** restituiti dall'iteratore
- **any<F>(&mut self, f: F) -> bool**
 - o **verifica che la closure ricevuta restituisca true per almeno 1 elemento** restituito dall'iteratore
- **find<P>(&mut self, predicate: P) -> Option<Self::Item>**
 - o **cerca un elemento sulla base della closure** ricevuta come argomento e lo ritorna
- **count(self) -> usize**
 - o ritorna il **n° di elementi** dell'iteratore
- **sum<S>(self) -> S**
 - o **somma** tutti gli elementi di un iteratore e ritorna il valore ottenuto
- **product<P>(self) -> P**
 - o **moltiplica** tutti gli elementi di un iteratore e ritorna il valore ottenuto
- **max(self) -> Option<Self::Item>**
 - o ritorna il **max** tra gli elementi dell'iteratore (per 2 massimi =, torna l'**ultimo**; se l'iteratore è vuoto viene ritornato **None**)
- **max_by<F>(self, compare: F) -> Option<Self::Item>**
 - o ritorna il **max** sulla base della **closure di confronto** ricevuta come argomento
- **max_by_key<B, F>(self, f: F) -> Option<Self::Item>**
 - o **applica la closure** ricevuta su tutti gli elementi e **ritorna quello che produce il risultato max**
- **min(self) -> Option<Self::Item>**
 - o // (come **max**)
- **min_by<F>(self, compare: F) -> Option<Self::Item>**
 - o // (come **max_by**)
- **min_by_key<B, F>(self, f: F) -> Option<Self::Item>**
 - o // (come **max_by_key**)
- **position<P>(&mut self, predicate: P) -> Option<usize>**
 - o **cerca un elemento in base alla closure** ricevuta e **ritorna la posizione**
- **rposition<P>(&mut self, predicate: P) -> Option<usize>**
 - o uguale a **position**, ma parte da **dx**
- **fold<B, F>(self, init: B, f: F) -> B**

- applica la closure ricevuta accumulando i risultati sul 1° argomento ricevuto
- **try_fold**`<B, F, R>(&mut self, init: B, f: F) -> R`
 - uguale a **fold**, ma fino a che non torna con **true**
- **last**`(self) -> Option<Self::Item>`
 - ritorna l'**ultimo elemento** dell'iteratore
- **find_map**`<B, F>(&mut self, f: F) -> Option`
 - applica la closure ricevuta su tutti gli elementi e **ritorna il 1° risultato valido**
- **partition**`<B, F>(self, f: F) -> (B, B)`
 - consuma un iteratore e **ritorna 2 collezioni sulla base del predicato ricevuto**
- **reduce**`<F>(self, f: F) -> Option<Self::Item>`
 - riduce l'iteratore ad un **singolo elemento** eseguendo la closure ricevuta
- **cmp**`<I>(self, other: I) -> Ordering`
 - confronta gli elementi di 2 iteratori
- **eq/ne/lt/le/gt/ge/...<I>(self, other: I) -> bool**
 - verifica se gli elementi di 2 iteratori sono **==, !=, <, <=, >, >=, ...**

Ora vediamo invece le **COLLEZIONI** (un elenco e la loro complessità temporale [importante scegliere bene la struttura a seconda del nostro scopo]):

Descrizione	Rust	C++	Java	Python
Array dinamico	<code>std::Vec<T></code>	<code>std::vector<T></code>	<code>java.util.ArrayList<T></code>	<code>list</code>
Coda a doppia entrata	<code>std::VecDeque<T></code>	<code>std::deque<T></code>	<code>java.util.ArrayDeque<T></code>	<code>collections.deque</code>
Lista doppiamente collegata	<code>std::LinkedList<T></code>	<code>std::list<T>*</code> *esiste anche collegata solo in avanti (<code>forward_list</code>)	<code>java.util.LinkedList<T></code>	—
Coda a priorità	<code>std::BinaryHeap<T></code>	<code>std::priority_queue<T></code>	<code>java.util.PriorityQueue<T></code>	<code>heapq</code>
Tabella hash	<code>std::HashMap<K,V></code>	<code>std::unordered_map<K,V></code>	<code>java.util.HashMap<K,V></code>	<code>dict</code>
Mappa ordinata	<code>std::BTreeMap<K,V></code>	<code>std::map<K,V></code>	<code>java.util.TreeMap<K,V></code>	—
Insieme Hash	<code>std::HashSet<T></code>	<code>std::unordered_set<T></code>	<code>java.util.HashSet<T></code>	<code>set</code>
Insieme ordinato	<code>std::BTreeSet<T></code>	<code>std::set<T></code>	<code>java.util.TreeSet<T></code>	—

Descrizione	Accesso	Ricerca	Inserimento	Cancellazione
Array dinamico	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Coda a doppia entrata	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Lista doppiamente collegata	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Coda a priorità	$O(1)$	—	$O(\log(n))$	$O(\log(n))$
Tabella hash	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Mappa ordinata	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$
Insieme Hash	—	$O(1)$	$O(1)$	$O(1)$
Insieme ordinato	—	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$

Ci sono dei **metodi comuni a tutte le collezioni**:

- **new()** → alloca nuova collezione
- **len()** → dà dimensione collezione
- **clear()** → svuota collezione

- `is_empty()` → true se è vuota
- `iter()` → itera sui valori della collezione

Oltre a questi, tutte le collezioni implementano i tratti **IntoIterator** e **FromIterator** e hanno quindi i metodi:

- `into_iter()` → converte qualsiasi collezione in iteratore
- `collect()` → ottiene collezione partendo da iteratore

Vediamo quindi una **panoramica veloce sulle collezioni** (prima citate in tabella):

- **`Vec<T>`** → **sequenza ridimensionabile** di elementi di tipo `T` allocati nell'heap; creato con `Vec::new()` o con `vec![val1, val2, ...]`. Una variabile di tipo `Vec<T>` è una tupla formata da 3 valori privati: **puntatore** al buffer nell'heap, **dimensione** complessiva del buffer (unsigned int) e **n° elementi correnti** nel buffer (unsigned int). Si può inserire un elemento al fondo del buffer con `push(...)`; se il buffer è pieno, viene **sostituito** da uno più grande (buffer precedente viene deallocated).

Si ottiene un riferimento al contenuto del vettore con `&v[indice]` (in questo caso errore di indice genera `panic!`) o con `get(...)` e `get_mut(...)` (in questo caso: successo = `Option::Some(ref)`; errore = `Option::None`).

Ci sono diversi metodi utili:

- `Vec::with_capacity(n)` alloca un vettore con capacità `n`
- `capacity()` ritorna la len del vettore
- `push(value)` aggiunge un elemento al fondo
- `pop()` rimuove e ritorna un `std::Option` con l'elemento al fondo del vettore (se esiste)
- `insert(index, value)` aggiunge un elemento all'indice scelto
- `remove(index)` rimuove e ritorna l'elemento all'indice scelto
- `first()` e `first_mut()` ritornano un riferimento (mutabile) al 1° elemento
- `last()` e `last_mut()` ritornano un riferimento (mutabile) all'ultimo elemento
- `get(index)` e `get_mut(index)` ritornano un `std::Option` con il riferimento (mutabile) all'elemento all'indice scelto (se esiste)
- `get(range)` e `get_mut(range)` ritornano un `std::Option` con lo slice nell'intervallo di indici (se esiste)

I dati però devono essere **omogenei**; se li voglio **eterogenei**, posso usare un **enum** come "busta" di elementi:

```
enum SpreadsheetCell {
    Int(i32),
    Float(f64),
    Text(String),
}

let row = vec![
    SpreadsheetCell::Int(3),
    SpreadsheetCell::Text(String::from("blue")),
    SpreadsheetCell::Float(10.12),
];
```

- **`VecDeque<T>`** → **coda a doppia entrata**: quindi anch'esso con `Vec<T>` è nell'heap, però permette `push_back`, `push_front`, `pop_back`, `pop_front` (quindi da testa e da coda). Viene implementato come un **buffer circolare** e non garantisce che gli elementi siano contigui in memoria (va usato `make_contiguous()`)
 - **`LinkedList<T>`** → **lista doppia linkata** con **tempo di accesso costante**; è meno efficiente di `Vec` e `VecDeque` e ha meno metodi. Si può creare con `LinkedList::From([val1, val2, ...])`
 - **`HashMap<K, V>`** → **chiave (K) – valore (V)** (i valori sono salvati nell'heap come 1 **hash table**). Si usano quando le **chiavi non hanno un ordine**; l'inserimento di una nuova entry può causare riallocazione e movimento dei dati. La chiave deve essere univoca e il suo tipo (`K`) deve implementare i tratti **Eq** e **Hash**
 - **`BTreeMap<K, V>`** → **chiave (K) – valore (V)** (i valori sono salvati nell'heap come 1 **albero** dove ogni entry è un nodo). Si usano quando le **chiavi hanno un ordine**; l'inserimento di una nuova entry può causare riallocazione e movimento dei dati. La chiave deve essere univoca e il suo tipo (`K`) deve implementare il tratto **Ord**
- ⚠ Dato che le coppie chiave-valore appartengono alla mappa, i metodi di accesso (`get`, `get_key_value`, `keys...`) usano `&K` e restituiscono `&V` (ovvero **riferimenti**)

- ⚠ Rust ottimizza l'uso delle mappe: il metodo `entry(&mut self, key: K) -> Entry<'a, K, V>` permette di cercare una chiave nella mappa e ritorna un `enum (Entry)` in base al risultato della ricerca; l'enum `Entry<'a, K, V>` mette a disposizione metodi per gestire il risultato (riducendo spostamenti in memoria):
- `and_modify<F>(self, f: F)` in caso di successo permette di eseguire delle azioni aggiuntive sul risultato ottenuto
 - `or_insert(self, default: V)` in caso di fallimento è possibile inserire una nuova entry senza costi aggiuntivi poiché il puntatore sarà già indirizzato verso una zona di memoria libera

```
let mut animals: HashMap<&str, u32> = HashMap::new();

animals.entry("dog")
    .and_modify(|v| { *v += 1 })
    .or_insert(1);
```

- **HashSet<T>** → set di elementi univoci di tipo T (valori salvati nell'heap come 1 **hash table**). L'inserimento di una nuova entry può causare riallocazione e movimento dei dati; è implementato come un **wrapper** attorno al tipo `HashMap<T, ()>`
- **BTreeSet<T>** → set di elementi univoci di tipo T (valori salvati nell'heap come 1 **albero** dove ogni elemento è un nodo). L'inserimento di una nuova entry può causare riallocazione e movimento dei dati; è implementato come un **wrapper** attorno al tipo `BTreeMap<T, ()>`
- **BinaryHeap<T>** → collezione di elementi tipo T (valori salvati nell'heap e l'elemento **max** è sempre nella 1^a posizione) [T deve implementare il trait **Ord**]; `peek()/peek_mut()` ritornano l'elemento max (quello nella 1^a posizione) [se mut, modificabile]

8) FILE I/O

Un **FILE** è un'astrazione che lega un blocco di byte ad un nome (parte di Cabodi); c'è la struct `std::fs::File` (che modella un file aperto in lettura/scrittura). Ogni OS ha la sua gerarchia/organizzazione, ma con Rust si possono usare le struct `std::path::Path` (come `str`, è unsized ed è sola lettura) e `std::path::PathBuf` (come `String`, ha contenuto ed è modificabile) che nascondono tali differenze (**interoperabilità**).

Navigare il File System:

- `std::fs::read_dir(dir: &Path) -> Result<ReadDir>` restituisce, se successo, iteratore al contenuto della cartella dir (le singole entry ritornate sono `std::fs::DirEntry`)
- `std::fs::create_dir(dir: &Path) -> Result<()>` crea nuova cartella (fallisce se non si ha autorizzazione, se la cartella esiste già o se la cartella genitrice del path indicato dir non esiste)
- `std::fs::remove_dir(dir: &Path) -> Result<()>` rimuove cartella dir (se esiste, se si ha autorizzazione e se è vuota)
- `std::fs::copy(from: &Path, to: &Path) -> Result<i64>` copia contenuto di file in un altro file (in caso di successo ritorna n° byte copiati)
- `std::fs::rename(from: &Path, to: &Path) -> Result<()>` rinomina (sposta) un file in un altro file
- `std::fs::remove_file(path: &Path) -> Result<()>` elimina file (se in uso, verrà fatto in seguito dall'OS)

La struct **File** offre 2 metodi per aprire un file:

- `open(path: P) -> Result<File>` where `P: AsRef<Path>` apre il file in lettura (se esiste)
- `create(path: P) -> Result<File>` where `P: AsRef<Path>` tronca il file a 0 byte (se esiste) o lo crea (se non esiste ancora), poi lo apre in scrittura

⚠ Maggiori opzioni vengono dalla `std::fs::OpenOption` (per impostare come un file debba essere aperto e quali permessi si hanno su di esso [vedi laboratori])

Operazioni su file:

- `std::fs::read_to_string(path: &Path)` legge il file in 1 stringa

➤ `std::fs::write(path: &Path, contents: &[u8])` scrive sul file il contenuto del buffer

Ci sono dei **tratti per gestire le operazioni di I/O** (importabili con `std::io::prelude::*`) [i principali sono `Read`, `BufRead`, `Write` e `Seek`] e se si verificano errori, viene ritornata 1 delle varianti nell'enum `ErrorKind`; vediamo questi tratti:

- **std::io::Read** → capacità di leggere un flusso di byte (es. in `File`, `Stdin` e `TcpStream`); la sua implementazione **richiede** il metodo `read(buf: &mut [u8]) -> Result<u8>`; se successo, torna `Ok(n)` con `0 < n < buf.len()` [infatti `Ok(0)` indica che il flusso è terminato o che il buffer passato ha `len = 0`] [ogni chiamata a `read` può generare una syscall]
Offre diversi metodi:
 - `read_to_end(buf: &mut Vec<u8>) -> Result<u8>` continua a leggere fino all' EOF
 - `read_to_string(buf: &mut String) -> Result<u8>` continua a leggere fino all' EOF e mette ciò che legge dentro `&mut String`
 - `read_exact(buf: &mut [u8]) -> Result<u8>` 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>` concatena 2 reader
 - `take(limit: u64) -> Take<Self>` limita il numero massimo di byte che sarà possibile leggere
- **std::io::BufRead** → migliora le prestazioni di I/O appoggiandosi ad un buffer in memoria (meno syscalls, meno context change). La sua implementazione **richiede** i metodi `fill_buf()` [ritorna il contenuto del buffer in memoria] e `consume(amt: u8)` [garantisce che i byte non siano tornati nuovamente; va sempre chiamato dopo `fill_buf`]; **offre** i metodi `read_line(&mut self, buf: &mut String)` e `lines(self)` per accedere al contenuto testuale di un flusso
- **std::io::Write** → capacità di scrivere un flusso di dati (es. in `File`, `Stdout`, `Stderr` e `TcpStream`); la sua implementazione **richiede** i metodi `write(buf: &[u8]) -> Result<u8>` [prova a scrivere l'intero contenuto del buffer e ritorna n° byte scritti] e `flush() -> Result<u8>` [finalizza output garantendo che tutti gli eventuali buffer tmp siano svuotati]. Il metodo `write_all(buf: &[u8])` chiama ricorsivamente `write` fino a che non ho scritto tutti i dati o errore fatale
- **std::io::Seek** → riposiziona il cursore di lettura/scrittura in un flusso di byte (`SeekFrom::Start`, `SeekFrom::End`, `SeekFrom::Current`). Offre i 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<u64>` 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

⚠ Lettura e scrittura di **contenuti strutturati**: framework **Serde** (genera implementazioni efficienti di funzioni di **serializzazione e deserializzazione di strutture dati arbitrarie verso formati standard** come JSON, CSV...). Lo si importa (inserendo nel file `cargo.toml` le dipendenze) e lo si usa con:

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

```
fn save(data: &Data, path: &str) -> Result<u8> {
    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)?);
}
```

9) SMART POINTER

Ogni valore manipolato da un programma è memorizzato nello spazio di indirizzamento del processo. L'operatore `&` e `&mut` permette di ottenere l'**indirizzo del 1° byte** in cui è memorizzato (in Rust si attiva il già citato **borrow checker**). L'operazione duale detta **dereferenza** (o risoluzione del riferimento) trasforma un indirizzo nel corrispondente **valore puntato** (con `*` e `.`).

L'uso dei puntatori nativi però è spesso causa di errore ed inoltre le **regole restrittive del borrow checker** di Rust impediscono (con l'uso di riferimenti nativi) la creazione di strutture cicliche (ogni valore è parte di 1 solo albero con radice in una variabile). Tramite gli **SMART POINTER** possiamo avere **più possessori** di 1 valore [con `Rc<T>` e `Arc<T>`] e possiamo avere **strutture cicliche** [`std::rc::Weak` e `std::sync::Weak`]

⚠ In C++ abbiamo gli smart pointer con `std::unique_ptr<T>` (rilasciato automaticamente fuori scope o quando viene distrutto/rilasciato; non può essere copiato, ma solo mosso in un'altra variabile) e `std::shared_ptr<T>` (ha associata una struttura di controllo che mantiene il **n° di riferimenti esistenti** in quanto può essere copiato [la copia indica lo stesso blocco dell'originale, ma viene incrementato il n° di riferimenti]; ha contatore dei riferimenti forti [copie] e dei deboli e quando il contatore dei forti arriva a 0 viene rilasciato)
In alcuni casi, se si forma un ciclo di dipendenze ($A \rightarrow B$ e $B \rightarrow A$) il contatore degli shared_ptr non potrà mai annullarsi (anche se gli oggetti A e B non sono più noti): `std::weak_ptr<T>` permette di **creare dipendenze cicliche senza incrementare il n° di riferimenti esistenti**; lui lavora infatti sui **riferimenti deboli**

In **Rust** abbiamo **varietà maggiore di smart pointer**, ma in generale sono realizzati mediante struct con dentro le info necessarie e che implementano i tratti **Deref** e **DerefMut** (quando il compilatore incontra l'espressione `*ptr` la trasforma in `(*ptr.deref())` oppure `(*ptr.deref_mut())`):

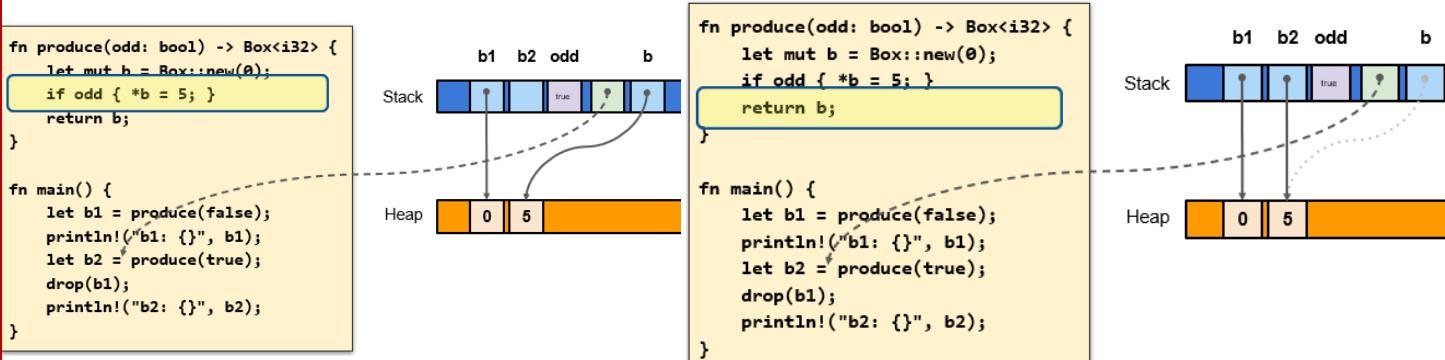
```
trait Deref {  
    type Target: ?Sized;  
    fn deref(&self) -> &Self::Target;  
}  
  
trait DerefMut: Deref {  
    fn deref_mut(&mut self) -> &mut Self::Target;  
}
```

Vediamo questi **smart pointer di Rust**:

- **std::Box<T>** → incapsula un puntatore ad un blocco allocato dinamicamente sullo heap all'atto della sua costruzione (tramite `Box::new(t)`). Il dato è **posseduto** da Box; quando la struttura esce dal suo scope, il blocco sull'heap viene rilasciato automaticamente grazie all'implementazione del tratto **Drop** (posso anche rilasciarlo prima invocando `drop(blocco)`). Se la struttura viene mossa in un'altra variabile (o in return da funzione), il possesso del puntatore passa alla **destinazione** che diventa responsabile del suo rilascio (otteniamo quindi **life cycle che si estende oltre lo scope della funzione in cui il dato è stato creato**)

⚠ Il tipo **T** può avere **dimensione non nota** in fase di compilazione (ovvero può **non** implementare il tratto **Sized**): in questo caso il `Box<T>` diventa un **fat pointer** con un puntatore seguito da un **usize** contenente la lunghezza del dato puntato

⚠ Analogamente, se al posto del tipo concreto **T** si indica un **oggetto-tratto (dyn Trait)**, si ha un **fat pointer** composto da **2 puntatori**: quello al dato sullo heap e quello a vtable del tratto



- **`std::rc::Rc<T>`** → se serve disporre di **più possessori di uno stesso dato immutabile**: internamente ha **copia del dato e 2 contatori** (il 1° indica quante copie/riferimenti forti del puntatore esistono [ogni volta che viene clonato incremento], il 2° quanti **riferimenti deboli** sono presenti); quando il puntatore esce dal suo scope, il contatore viene **decrementato** (il 1°) e se == 0, il blocco viene **rilasciato** (come lo `std::shared_ptr<T>` del C++). Quindi `Rc<T>` si presta a realizzare *alberi e grafi aciclici*
 - ⚠ Per evitare problemi di omonimia con i metodi contenuti nel dato encapsulato, i metodi di `Rc` sono dichiarati usando `this` al posto di `self` (es. `pub fn strong_count (this: &Rc<T>) -> usize`); in questo modo **non possiamo usare la notazione puntata** per invocare i metodi, ma vanno chiamati nella forma estesa
 - ⚠ Per motivi di efficienza, l'operazione di incremento e decremento sui campi privati `strong_count` e `weak_count` **non è thread-safe** (per questo **non può essere usato da più di 1 thread**); per superare questo limite sulla concorrenza, si usa la classe `std::sync::Arc<T>`
- **`std::rc::Weak<T>`** → stessa cosa di prima vista in C++ sulle **dipendenze circolari**; si crea un valore `Weak<T>` a partire da un valore di tipo `Rc<T>` con il metodo `Rc::downgrade(&rc)` [viceversa torno a `Rc<T>` con il metodo `upgrade()`]
- **`std::cell::Cell<T>`** → funziona solo in contesti **non concorrenti** (1 thread solo); implementa “**interior mutability**” (**implementa mutabilità del dato** contenuto al suo interno **attraverso metodi che non richiedono la mutabilità del contenitore Cell**)


```
use std::cell::Cell;
struct SomeStruct {
    a: u8,
    b: Cell<u8>,
}

let my_struct = SomeStruct {
    a: 0,
    b: Cell::new(1),
};
my_struct.a = 100; // ERRORE: `my_struct` è immutabile

my_struct.b.set(100);
// OK: anche se `my_struct` è immutabile, `b` è una Cell e può essere modificata

assert_eq!(my_struct.b.get(), 100);
```
- I suoi **metodi** sono:
 - `get(&self)` -> `T` = restituisce il dato contenuto al suo interno (a condizione che `T` implementi `Copy`)
 - `take(&self)` -> `T` = restituisce il valore contenuto, sostituendolo con il risultato dell'invocazione di `Default::default()` (a condizione che `T` implementi `Default`)
 - `replace(&self, val:T)` -> `T` = sostituisce il valore contenuto nella `Cell` con quello passato come parametro e lo restituisce
 - `into_inner(&self)` -> `T` = consuma la `Cell` e restituisce il valore contenuto
- **`std::cell::RefCell<T>`** → mentre `Cell<T>` non consente di creare riferimenti al dato contenuto al suo interno ma permette solo di inserire, estrarre o sostituire il valore, a `RefCell<T>` posso accedere attraverso particolari smart pointer che **simulano il comportamento di riferimenti condivisi e mutabili** (la **compatibilità con le regole del borrow checker** avviene a **runtime** [se non vengono rispettate, `panic!`]). I **metodi**:
 - `borrow(&self)` -> `Ref<'_, T>` = restituisce smart pointer che implementa `Deref<T>` oppure `panic!` se è già presente un riferimento mut
 - `borrow_mut(&self)` -> `RefMut<'_, T>` = restituisce smart pointer che implementa `DerefMut<T>` oppure `panic!` se è già presente un riferimento semplice
- **`std::borrow::Cow<'a, B>`** → implementa il meccanismo di **clone on write (COW)**:
 - se cerco di modificare il dato contenuto ed è **condiviso**, il dato viene **clonato** (si prende possesso della copia e la si modifica, lasciando l'originale invariato)

- se cerco di modificare il dato contenuto ed è **posseduto**, non avviene clonazione e si **modifica diretto**

⚠ Si implementa con **enum** e si istanzia con il metodo **Cow::from(...)** → il compilatore sceglie in base al dato fornito se collocare il valore nella variante **Owned** o **Borrowed**

Esempio con albero doppiamente collegato:

```
#[derive(Debug)]
struct Node {
    name: String,
    parent: RefCell<Weak<Node>>,
    children: RefCell<Vec<Rc<Node>>>,
}

impl Node {
    fn new(name: &str) -> Rc<Self> {
        Rc::new(Node {
            name: name.to_string(),
            parent: RefCell::new(Weak::new()),
            children: RefCell::new(Vec::new()),
        })
    }

    fn add_child(parent: Rc<Node>, child: Rc<Node>) {
        *child.parent.borrow_mut() = Rc::downgrade(&parent);
        parent.children.borrow_mut().push(child);
    }

    fn print_tree(node: &Rc<Node>, depth: usize) {
        println!("{}{}", " ".repeat(depth * 2), node.name);
        for child in node.children.borrow().iter() {
            Node::print_tree(child, depth + 1);
        }
    }

    fn get_root(node: &Rc<Node>)-> Rc<Node> {
        let mut n = node.clone();
        loop {
            let parent = n.parent.borrow().upgrade();
            match parent {
                Some(p) => n = p.clone(),
                None => break,
            }
        }
        n
    }
}

fn main() {
    let root = Node::new("root");
    let child1 = Node::new("child1");
    let child2 = Node::new("child2");
    Node::add_child(root.clone(), child1.clone());
    Node::add_child(root.clone(), child2);
    Node::add_child(child1.clone(), Node::new("subchild"));
    Node::print_tree(&Node::get_root(&child1), 0);
}
```

```
pub enum Cow<'a, B>
where B: 'a + ToOwned + ?Sized,
{
    Borrowed(&'a B),
    Owned(<B as ToOwned>::Owned),
}
```

10) MODULARITÀ e TEST

La **compilazione** di un progetto Rust è basato sul concetto di **crate** (unità di compilazione; insieme di file che costituiscono il progetto). La **compilazione di ciascun file appartenente al crate origina un file oggetto** che potrà essere **linkato con altri file oggetto** per diventare un **eseguibile** oppure una **libreria**.

La sezione **[lib]** del file **Cargo.toml** (file che descrive il progetto) contiene i dettagli relativi al tipo di libreria che si intende creare, infatti il **crate-type** può essere:

- **rlib** → libreria **statica** Rust (valore di default)
- **dylib** → libreria **dinamica** Rust; trasformata in un file **.dll** in windows, **.so** in linux, **.dylib** in macOS; può essere utilizzata solo da codice Rust, ma sarà caricata in fase di esecuzione
- **cdylib** → libreria **dinamica** conforme alle specifiche **C** (usabile da eseguibili scritti in altri linguaggi); se usata da programma Rust deve essere inclusa come funzioni esterne
- **staticlib** → libreria **statica** conforme alle specifiche **C**; trasformata in un file **.lib** in windows e **.a** in linux e macOS (usabile da eseguibili scritti in altri linguaggi); se usata da programma Rust deve essere inclusa come funzioni esterne

Il codice contenuto in un crate è costituito da **albero di moduli**; il codice non appartenente ad un blocco di tipo **mod nome_modulo {...}** fa parte del **MODULO root** (ovvero l'unità di compilazione corrente, rappresentata dalla parola chiave **crate**). Se un elemento definito in un modulo ha la parola **pub** davanti, è accessibile in altri moduli

⚠ Se voglio usare delle funzioni in un altro modulo, devo usare il percorso completo del tipo **path::modulo::nome_funzione**; oppure per semplicità posso mettere **use path::modulo::*** all'inizio del file e in questo modo posso **usare direttamente i nomi senza ogni volta specificare tutto il path**

⚠ Crate esterni importati con cargo, saranno elencati nella sezione **[dependencies]** del file **cargo.toml**

Alcuni **simboli frequenti** (come **Vec**, **String**, ...) non dovranno essere importati esplicitamente perché sono elencati nel modulo **std::prelude** importato automaticamente nella compilazione di un file sorgente.

TEST in **Rust** (già visti nei LAIB, ma non qui in teoria): abbiamo i **test di unità** (testano singolo componente software; costo modifiche evidenziate da questo test è normalmente basso) [in Rust vengono scritti direttamente nel modulo da testare con **#[cfg(test)]**], i **test di integrazione** (testano il comportamento all'interfaccia dei moduli software; costo modifiche più alto) [in Rust sono contenuti in una **cartella separata /tests** a lato della cartella **/src**] e i **test di sistema** (testano prodotto finito, basandosi su modelli di comportamento tratti da casi d'uso reali; **E2E**).

```
// funzione da testare
fn sum(a: i32, b: i32) -> i32 {
    a + b
}

#[cfg(test)]
mod tests {
    fn sum_inputs_outputs() -> Vec<((i32, i32), i32)> {
        vec![(1, 1), 2, ((0, 0), 0), ((2, -2), 0)]
    }

    #[test]
    fn test_sums() {
        for (input, output) in sum_inputs_outputs() {
            assert_eq!(crate::sum(input.0, input.1), output);
        }
    }
}
```

La struttura di una **funzione di test** (**#[test]**) è **arrange** (preparazione dei valori su cui testare), **act** (esecuzione del codice da testare) e **assert** (verifica che i risultati siano quelli attesi). Per eseguire i test uso **cargo test** (tutti i test) oppure **cargo test nome** (esegue tutti i test i cui nomi contengono la parola "nome").

⚠ Posso mettere davanti alcuni test **#[ignore]** per toglierli dall'esecuzione; si può usare la libreria **rstest** per creare **test parametrici** (i parametri vengono forniti con funzioni **#[fixture]** oppure con elenco di tuple **#[case(...)]**); tali funzioni di test devono essere precedute da **#[rstest]**

11) CONCORRENZA

Un programma concorrente dispone di **più flussi di esecuzione contemporanei**; alla **creazione** un processo ha 1 flusso di esecuzione (**thread principale**), ma questo può chiedere allo scheduler la **creazione di altri thread**.

THREAD = **computazione indipendente**, basata su un **proprio stack** (pre-allocato alla creazione del thread) collocato nello **stesso address space degli altri thread del processo**. La gestione dei thread può essere chiesta all'OS (**THREAD NATIVI**; supporto offerto da *Rust*) o gestita a livello utente con supporto parziale dell'OS (**GREEN THREAD** o **FIBRE**; supporto offerto da *library di terze parti*).

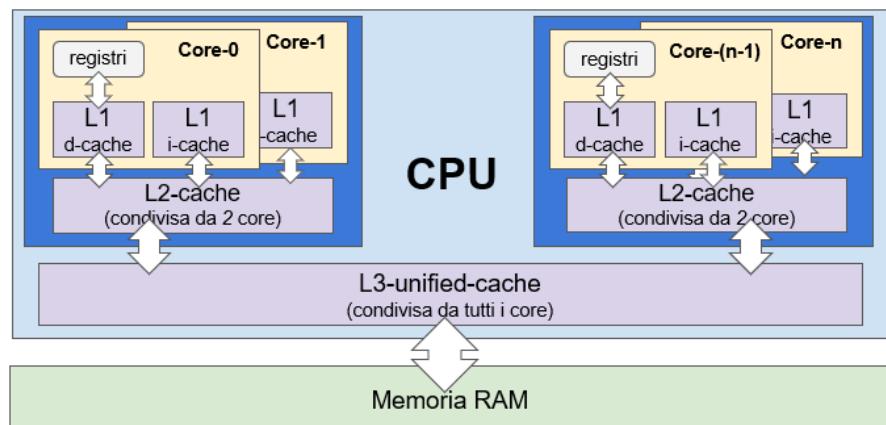
Riguardo i **thread nativi**, l'OS può **creare** thread (indicando la funzione che devono svolgere e la dimensione dello stack richiesto; ritornano un handle opaco con cui fare riferimento al thread), **identificare** il thread corrente (TID) e **attendere la terminazione** di un thread. **Non può cancellare un thread** (questa funzione può essere **solo implementata in cooperazione** con il thread stesso)

⚠ In un sistema **single-core**, il thread è un'**astrazione offerta dall'OS** (il processore modifica la sua esecuzione basata su operazioni suddivise in fetch/decode/execute: lo scheduler permette l'**out-of-order execution**)

Cosa implica la concorrenza? Possibilità di **sovrapporre** temporalmente **computazione e operazioni I/O** (solitamente I/O bloccante, ma con più thread posso suddividere l'algoritmo in più thread evitando tempi di wait lunghi) e **riduzione del sovraccarico** dovuto alla **comunicazione tra processi** (**i thread a differenza dei processi condividono l'address space quindi meno problemi** nella parallelizzazione).

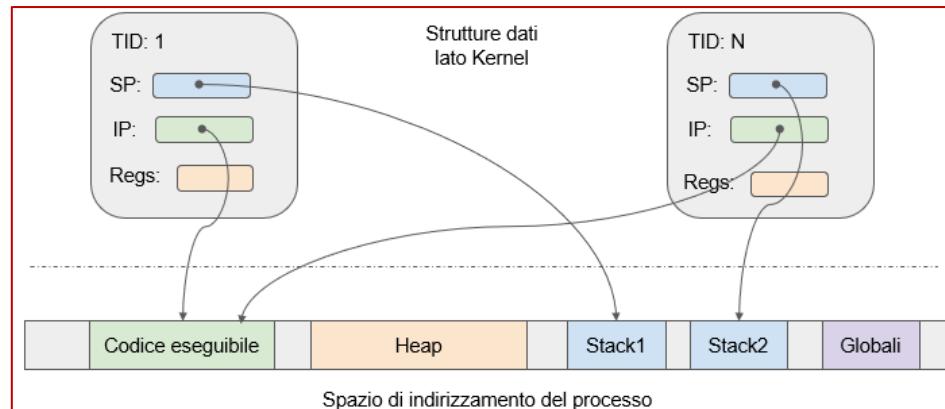
Inoltre, si ottiene **vero parallelismo con il multi-threading** (sfrutto appieno l'**elaborazione multicore**), ma aumenta anche la **complessità** del programma (**non-determinismo dell'esecuzione**): la memoria non può essere più vista come un "deposito statico" e i **thread devono coordinare l'accesso alla memoria**.

Quando un thread legge il contenuto di una locazione di memoria, può trovare il valore iniziale contenuto nell'exe, il valore **modificato dallo stesso thread** o il valore **modificato da un altro thread**: il 3° caso è **problematico**! Vediamo il modello di **memoria "packed cores"** (da notare che le **Level1** cache sono più veloci delle **L2** cache...):



⚠ L'uso **superficiale** dei costrutti di sincronizzazione porta problemi, mentre la loro **assenza** porta a risultati imprevedibili. Inoltre, si possono verificare malfunzionamenti **casuali** e gli **errori** possono manifestarsi al **cambio della piattaforma** di esecuzione

L'OS mantiene una **rappresentazione dei thread** presenti nei vari processi (ognuno con identificativo univoco all'interno del processo [TID], stato di esecuzione e informazioni per il **context switching**). **Tutti i threads di un processo condividono** le **variabili globali**, le **costanti**, l'area eseguibile in cui c'è il **codice** e l'**heap**, ma **ognuno ha il suo stack**.



⚠ Ricorda che in Rust una **variabile globale mutabile** non è permessa, se non con l'uso di **unsafe**

La **SINCRONIZZAZIONE** dei thread può riguardare il raggiungimento di un certo stato da parte di un thread (**abilitando gli altri threads a procedere**) oppure l'esigenza di un thread di eseguire azioni su aree condivise (**impedire ad altri threads di accedere alle stesse aree**). Se non gestiamo l'ordinamento dell'esecuzione dei threads, il non determinismo dà origine a **comportamenti inattesi** (in un contesto di elaborazione sequenziale possiamo avere molti output possibili in base all'ordine dei threads) [es. l'operazione **a++** sembra un semplice incremento, ma in realtà nasconde 2 operazioni: **int tmp = a; a = tmp+1;** → in questo caso con 2 thread che operano entrambi con **a++** su **a** variabile globale, si genera **INTERFERENZA** causando malfunzionamenti casuali]

Se 2 thread cercano di **accedere in lettura/scrittura ad una stessa variabile** si verifica una **CORSA CRITICA** (**race condition**, ovvero il suo valore dipende dall'ordine dell'esecuzione); quindi l'accesso a questa variabile deve essere "protetto": alcune di queste protezioni sono date da **istruzioni macchina** (dipendono dal processore), altre richiedono la garanzia di essere **invarianti a livello di sistema** (fornita dall'OS) [quindi la sincronizzazione dipende da **ABI** (Application Binary Interface), ovvero **processore + OS**; ma le librerie standard dei linguaggi (es. C++11 e Rust) standardizzano tali comportamenti].

⚠ Non deve mai capitare che un **thread modifichi un dato se un altro thread sta già operando sullo stesso** oggetto e **non** devono essere visibili **stati transitori** dell'oggetto; oggetti condivisi mutabili devono avere queste proprietà

In Rust le limitazioni imposte dal **borrow checker** sull'esclusività dell'accesso in scrittura + l'uso di **tratti** che modellano il comportamento di un tipo quando viene passato **da un thread ad un altro** sono garanti della **correttezza degli accessi** (*fearless concurrency*).

Quindi i **problemi della concorrenza** (risolti dalle famiglie di processori in modo diverso) sono:

- **Atomicità** delle operazioni
- **Visibilità** delle modifiche fatte da un thread da parte di un altro thread
- **Ordinamento** delle operazioni se ho più thread attivi

Le **soluzioni** sono:

- parola chiave **atomic** davanti alle variabili per fare in modo che le operazioni su di essa siano atomiche
- i **mutex** (**mutua esclusione**) posseduti da un thread per volta (se mutex già occupato, il thread deve aspettare che venga liberato)
- **condition variable** → in alcuni casi per il proseguimento del thread bisogna aspettare che si verifichi una condizione più complessa del semplice rilascio di un mutex (può essere usata solo in coppia con un mutex)

⚠ In C++ la classe **std::thread** offre supporto per i thread nativi (possiamo crearlo, attendere la terminazione con **join()** bloccante, disgiungere l'oggetto dal thread nativo con **detach()**) [se finisce male, **std::terminate()**]

In **Rust** si **crea un thread nativo** con la funzione **std::thread::spawn(lambda)** [**lambda** è la funzione lambda che rappresenta **ciò che deve fare il thread**], che ritorna una struct **std::thread::JoinHandle<T>** con **T** tipo in return dalla computazione del thread. Per sapere quando il thread ha **finito la sua computazione** si usa il metodo **join()** offerto dall'handler: questo ritorna un **std::thread::Result** che contiene se **Ok** → valore finale, se **Err** → valore di errore (passato eventualmente alla **panic!** se chiamata dal thread stesso):

```
use std::thread;

let thread_join_handle = thread::spawn(move || { //move trasferisce alla funzione
    //il possesso di quanto catturato
    //computazione da eseguire
});

match thread_join_handle.join() {
    Ok(res) => { ... },
    Err(err) => { ... },
}
```

Possiamo anche **configurare un thread** prima di lanciare la sua esecuzione con la struct **std::thread::Builder**, la quale permette di assegnargli il nome e la dimensione dello stack (metodo **spawn** prima visto consuma **Builder**, crea il thread e ritorna un enum di tipo **io::Result<JoinHandle>**).

```

use std::thread;

let builder = thread::Builder::new()
    .name("t1".into())
    .stack_size(100_000);

let handler = builder.spawn(|| { /* codice */ }).unwrap();

handler.join().unwrap();

```

Per garantire correttezza di accessi e assenza di comportamenti indefiniti, ci sono **2 tratti marcatori** (no metodi):

- **std::marker::Send** → applicato automaticamente a tutti i tipi **trasferibili in sicurezza** da un thread ad un altro, ovvero che **non è possibile avere accessi al contenuto contemporaneamente**. Con pub unsafe auto trait `Send{}` significa che il tipo può essere passato “**per valore**” ad altri thread; l’uso del movimento (o della copia dove possibile) garantisce la non-contemporaneità degli accessi

⚠ Puntatori e riferimenti non hanno il tratto **Send**

⚠ I tipi composti (struct, tuple, enum, array) godono del tratto **Send** se tutti i loro campi lo posseggono

- **std::marker::Sync** → applicato automaticamente a tutti i tipi `T` tale che `&T` risulta avere il tratto **Send**, ovvero che **non possono essere condivisi in sicurezza** tra thread. Con pub unsafe auto trait `Sync{}` significa che il tipo può essere passato “**come riferimento non mutabile**” ad altri thread

⚠ I tipi con mutabilità interna (`Cell` e `RefCell`) non hanno il tratto **Sync**

⚠ Si possono creare thread con `spawn` solo se i **dati catturati dalla funzione lambda** (che descrive ciò che deve fare il thread) e il suo **return type** hanno il tratto **Send** (altrimenti borrow checker dà errore)

I modelli di concorrenza offerti da Rust sono:

- condivisione di dati basata su sincronizzazione degli accessi a **struttura dati condivisa**
- condivisione di dati basata sullo **scambio di messaggi** (con 1 solo destinatario)

⚠ In C++ per condividere dati mutabili tra thread bisogna usare un **MUTEX** (mutua esclusione) con i metodi di `lock()` [ottenere possesso di un mutex o mettersi in attesa di ottenerlo] e `unlock()` [rilasciare il mutex]; non c’è però corrispondenza sintattica tra un mutex e la struttura dati che questo protegge (può proteggere diversi tipi di strutture). Se però un thread in possesso di un mutex termina senza rilasciarlo? In C++ si ricorre al paradigma *RAII*, ovvero si usa `std::lock_guard` che incapsula il mutex e lo **rilascia se lock_guard cessa di esistere**.

In **Rust**, invece, l’accesso ad uno **stato condiviso** (dato mutabile) richiede l’uso di **2 blocchi in cascata**:

1. **std::sync::Arc<T>** → permette il **possesso multiplo** di una struttura dati **in sola lettura** da più threads
 - a. permette di condividere il possesso di un dato, allocandolo nell’heap e mantenendo un conteggio dei riferimenti esistenti di tipo thread-safe; ha la struttura di uno smart-pointer
 - b. si può clonare con `clone()`

Analogamente a `Rc<T>`, essendo `Arc<T>` come uno **smart-pointer**, si può usare `std::sync::Weak<T>` per creare **dipendenze circolari senza problemi**: un oggetto `Weak<T>` si crea anche qui con `downgrade()` su `Arc<T>` (con `upgrade()` si fa l’opposto)

2. **std::sync::Mutex<T>** oppure **std::sync::RwLock<T>** oppure **tipi atomici** → consente l’**acquisizione** in lettura/ scrittura della struttura dati
 - a. incapsula un dato di tipo `T` oltre al riferimento ad un **mutex nativo** dell’OS
 - b. si accede al dato con il metodo `lock()`, il quale restituisce un oggetto `LockResult<MutexGuard<T>>` e si blocca fino a che non è stato possibile acquisire il mutex nativo
 - i. se `lock()` ha avuto **successo** la risposta contiene `MutexGuard<T>`; questo oggetto:
 1. implementa il tratto `Deref<T>` e si comporta come **smart-pointer** (deferenziandolo si ottiene `&mut T`)
 2. se `MutexGuard<T>` esce dallo scope, il mutex nativo viene **rilasciato**
 - c. se ultimo thread che ha acquisito il mutex termina prima di rilasciarlo, il mutex si trova in stato **avvelenato** (**poison**) e la risposta dà errore

Se gli accessi in lettura e scrittura sono sbilanciati usiamo **RwLock**<T> al posto di **Mutex**<T> (questa offre il metodo **read()** per accedere in lettura condivisa [restituisce **LockResult**<**RwLockReadGuard**>] e **write()** per accedere in scrittura esclusiva [restituisce **LockResult**<**RwLockWriteGuard**>]).

Invece, possiamo usare **std::sync::atomic** che mette a disposizione strutture dati che costituiscono primitive di comunicazione tra thread basate su memoria condivisa (**tipi atomici**); accanto alle operazioni di **read** e **write** con associata **barriera di memoria**, questi tipi offrono funzionalità **Read-Modify-Write** come **swap**, **compare_exchange**, **fetch_add**, **fetch_update** (tutte queste operazioni hanno come parametro il tipo di barriera di memoria da applicare per **read** e per **write**).

I tipi atomici sono **thread-safe** (implementano il tratto **Sync**) [quindi richiedono **solo &self** e non **&mut self** in quanto già **thread-safe**], ma non offrono meccanismi di condivisione esplicita (per permettere accesso condiviso al contenuto, incapsulati in **Arc**<T>); analogamente a **Cell**<T>, implementano **interior mutability**.

```
use std::sync::Arc;
use std::sync::atomic::{AtomicUsize, Ordering};
use std::{hint, thread};

fn main() {
    let spinlock = Arc::new(AtomicUsize::new(1));

    let spinlock_clone = Arc::clone(&spinlock);
    let thread = thread::spawn(move|| {
        spinlock_clone.store(0, Ordering::Release);
    });

    // Attendi
    while spinlock.load(Ordering::Acquire) != 0 {
        hint::spin_loop();
    }
    thread.join().unwrap();
}
```

Questa **combinazione** (ispirata all'approccio *RAII*) risolve il problema di **rendere esplicito nel codice cosa sia condiviso** (impedendo l'accesso senza il corretto possesso del relativo lock):

```
let shared_data = Arc::new(Mutex::new(Vec::new()));

let mut threads = vec![];
for (i in 1..10) {
    let mut data = shared_data.clone(); //duplicazione del possesso
    threads.push(thread::spawn( move || { //data è ceduto al thread
        let mut v = data.lock().unwrap(); //v è di tipo MutexGuard<T>
        v.push(i); //quando v esce dall scope, il lock
    }) );
}
for t in threads { t.join().unwrap(); } //v contiene i numeri da 1 a 9
```

Creare un thread con **spawn** non permette di **specificare la sua durata**: possiamo usare **std::thread::scope(|s: std::thread::Scope| {lambda})** dove **lambda** racchiude l'intero life cycle dei threads creati al suo interno; il parametro **s** invece offre il metodo **spawn** per creare nuovi threads. Terminata l'esecuzione di **lambda**, la funzione **scope** non ritorna finché tutti i thread creati al suo interno non sono terminati [ciò permette al borrow checker di considerare corretto l'uso di **riferimenti a variabili locali**]:

```
let mut v = vec![1, 2, 3];
let mut x = 0;
thread::scope(|s| {
    s.spawn(|| { // è lecito creare un riferimento a v
        println!("length: {}", v.len());
    });
    s.spawn(|| { // anche qui viene catturato &v
        for n in &v {println!("{}"), }
        x += v[0]+v[2]; // x è catturata come &mut
    });
})
// Solo quando entrambi i thread saranno terminati si proseguirà
v.push(4); // non ci sono più riferimenti, si può modificare
assert_eq!(x, v.len());
```

Spesso un thread deve essere bloccato nell'**attesa di specifica condizione** (es. un risultato intermedio): per farlo si usano le **CONDITION VARIABLES**; le variabili per la valutazione della condizione devono essere incapsulate in un **mutex** e i thread che devono completare la condizione per fare proseguire il thread devono poterlo **notificare** (ogni CV deve essere usata in coppia con 1 singolo mutex). In Rust c'è la struct **std::sync::Condvar** e la struttura dei suoi metodi facilita il collegamento con il dato protetto dal mutex:

- **pub fn new() -> Condvar** = crea nuova CV
- **pub fn wait<'a, T> (&self, guard: MutexGuard<'a, T>) -> LockResult<MutexGuard<'a, T>>** = sospende thread corrente fino alla ricezione di una notifica: durante la sospensione **unlock**, al ricevere della notifica riacquisisce lock e restituisce una **nuova MutexGuard**

⚠ Può capitare che un thread in attesa su una CV si svegli in assenza di un'esplicita notifica ("**notifiche spurious**"); c'è una versione della wait che aggiunge la **funzione condition come parametro** in modo che, al risveglio, il thread riacquisisce il lock ma prima di procedere valuta la **condition** ed esce dall'attesa solo se **condition** ritorna **true**

⚠ Può capitare anche che una notifica di risveglio vada persa ("**notifiche perse**"); si può usare **wait_while** che protegge da questa situazione

⚠ Ci sono poi anche delle versioni di wait che impostano un **timeout temporale** oltre cui il thread si sveglia indipendentemente dalla condizione (la 2^ versione gestisce il caso delle notifiche spurious)

```
pub fn wait_timeout<'a, T>(
    &self,
    guard: MutexGuard<'a, T>,
    dur: Duration
) -> LockResult<(MutexGuard<'a, T>, WaitTimeoutResult)>
```

```
pub fn wait_timeout_while<'a, T, F>(
    &self,
    guard: MutexGuard<'a, T>,
    dur: Duration,
    condition: F
) -> LockResult<(MutexGuard<'a, T>, WaitTimeoutResult)>
where F: FnMut(&mut T) -> bool
```

- **pub fn notify_one(&self)** = sveglia **1 thread casuale** in attesa della CV
- **pub fn notify_all(&self)** = sveglia **tutti i thread** in attesa della CV, che usciranno 1 alla volta dal metodo wait possedendo il **lock** (risveglio progressivo di tutti i thread, ma 1 alla volta in quanto ho 1 singolo lock)

```
let pair = Arc::new((Mutex::new(false), Condvar::new()));
let pair2 = Arc::clone(&pair);

// Inside of our lock, spawn a new thread, and then wait for it to start.
thread::spawn(move|| {
    let (lock, cvar) = &*pair2;
    let mut started = lock.lock().unwrap();
    *started = true;
    // We notify the condvar that the value has changed.
    cvar.notify_one();
});

// Wait for the thread to start up.
let (lock, cvar) = &*pair;
let mut started = lock.lock().unwrap();
while !*started {
    started = cvar.wait(started).unwrap();
}
```

Oltre alla condivisione dello stato, Rust offre un meccanismo di sincronizzazione/comunicazione tra thread con la **CONDIVISIONE DI MESSAGGI**: **std::sync::mpsc::channel<T>()** restituisce una coppia ordinata formata da una **struct Sender<T>** ed una **struct Receiver<T>**; tutti i dati inviati tramite **send** [che ha capacità infinita di memorizzazione temporanea dei messaggi] di **Sender** possono essere consumati tramite **recv** [che si blocca senza consumare cicli macchina in attesa di un messaggio] di **Receiver**. Ha approccio **N producer - 1 consumer**

⚠ Se il **Receiver** viene deallocato, eventuali tentativi di invio falliscono con **SendError<T>**; se **tutti i Sender** vengono deallocati, tentativi di lettura sul ricevitore falliscono con la generazione di **RecvError**

La condivisione dei messaggi ha anche una **versione sincrona** con **std::sync::mpsc::sync_channel<T>(bound: usize)** che restituisce una coppia di (**SyncSender<T>**, **Receiver<T>**); a differenza di un canale semplice, questo ha dei **limiti** sul numero di messaggi [se si costruisce un canale sincrono di dim = 0, diventa un "rendezvous": ogni operazione di lettura deve sovrapporsi temporalmente ad una di scrittura].

Ultime cose che vediamo sono le **LIBRERIE ESTERNE**: **CROSSBEAM** → supporta elaborazione concorrente:

- **Costrutti atomici** (`crossbeam::atomic::AtomicCell<T>`)
- **Strutture dati concorrenti** (`crossbeam::deque` (**Injector**, **Stealer** e **Worker**) = schedulatori basati sul furto di attività da eseguire; `crossbeam::queue`::{**ArrayQueue**, **SegQueue**} = implementano code di messaggi basate su multiple producer – multiple consumer [**MPMC**])
- **Canali MPMC** (`crossbeam::channel`::{**bounded**, **unbounded**} = creano canali unidirezionali; `crossbeam::channel`::{**after**, **tick**} = creano il solo estremo di ricezione che consegnerà un messaggio dopo il tempo indicato o periodicamente). **Pattern concorrenti** in Rust:
 - **Fan-out/Fan-in** = distribuire attività a più thread indipendenti e raccogliere i risultati prodotti in un 1 punto; usa coppia di canali

```
fn worker(id: usize, rx: Receiver<i32>, tx: Sender<String>) {
    while let Ok(value) = rx.recv() {
        tx.send(format!("W{} ({})", id, value)).unwrap();
    }
}

fn main() {
    let (tx_input, rx_input) = bounded::<i32>(10);
    let (tx_output, rx_output) = bounded::<String>(10);
    let mut worker_handles = Vec::new();
    for i in 0..3 {
        let rx = rx_input.clone();
        let tx = tx_output.clone();
        worker_handles.push(thread::spawn(move || worker(i, rx, tx)));
    }
    for i in 1..=10 { tx_input.send(i).unwrap(); }
    drop(tx_input);
    while let Ok(result) = rx_output.recv() {
        println!("Received result: {}", result);
    }
    for handle in worker_handles { handle.join().unwrap(); }
}
```

- **Pipeline** = serie di operazioni, ciascuna delle quali è eseguita da 1 thread e utilizza 1 canale per inoltrare i semi-lavorati tra 2 fasi successive

```
fn stage_one(rx: Receiver<i32>, tx: Sender<String>) {
    while let Ok(value) = rx.recv() {
        tx.send(format!("S1({})", value)).unwrap();
    }
}
fn stage_two(rx: Receiver<String>, tx: Sender<String>) {
    while let Ok(value) = rx.recv() {
        tx.send(format!("S2({})", value)).unwrap();
    }
}
fn main() {
    let (tx_input, rx_input) = bounded::<i32>(10);
    let (tx_stage_one, rx_stage_one) = bounded::<String>(10);
    let (tx_output, rx_output) = bounded::<String>(10);

    let stage_one_handle = thread::spawn(move || stage_one(rx_input, tx_stage_one));
    let stage_two_handle = thread::spawn(move || stage_two(rx_stage_one, tx_output));

    for i in 1..=10 { tx_input.send(i).unwrap(); }
    drop(tx_input);

    while let Ok(result) = rx_output.recv() { println!("Received result: {}", result); }

    stage_one_handle.join().unwrap();
    stage_two_handle.join().unwrap();
}
```

- **Producer/consumer** = consente ad 1 o più thread producer di generare valori che saranno elaborati dal 1° thread consumatore disponibile; usa 1 canale per la comunicazione

```
fn producer(id: usize, tx: Sender<(usize, i32)>) {
    for i in 1..=5 { tx.send((id, i)).unwrap(); }
}

fn consumer(id: usize, rx: Receiver<(usize, i32)>) {
    while let Ok((sender_id, val)) = rx.recv() {
        println!("Consumer {} received {} from {}", id, val, sender_id);
    }
}

fn main() {
    let (tx, rx) = bounded::<(usize, i32)>(10);

    let mut handles = Vec::new();
    for i in 0..3 {
        let tx = tx.clone();
        handles.push(thread::spawn(move || producer(i, tx)));
    }
    for i in 0..2 {
        let rx = rx.clone();
        handles.push(thread::spawn(move || consumer(i, rx)));
    }
    drop(tx);
    for handle in handles { handle.join().unwrap(); }
}
```

12) PROCESSI

PROCESSO = unità base di esecuzione di un applicativo nel contesto di un OS, identificato univocamente a livello di sistema da un **PID**; definisce un **proprio address space isolato** dove possono operare 1 o più thread (flussi di esecuzione schedulabili in modo indipendente) [⚠ i processi **possono comunque interferire** tra loro con accesso alle periferiche, alla rete e alle risorse centralizzate; per un controllo maggiore, gli OS mettono a disposizione meccanismi **IPC** (Inter-Process Communication)].

Ad ogni processo è associato almeno 1 thread (se 1, *primary thread*).

In **Windows**, un processo si crea con la funzione **CreateProcess(...)** la quale crea un nuovo address space, lo inizializza con l'immagine di un exe e attiva il primary thread al suo interno. Il **processo figlio** **può condividere** variabili d'ambiente e handle a file, semafori, pipe, ma **non può condividere** handle a thread, processi, librerie dinamiche e regioni di memoria.

In **Linux**, si crea un **processo figlio** con la syscall **fork()** la quale crea un nuovo **address space** "identico" a quello del padre e condivide i **riferimenti** alle stesse pagine di memoria fisica. Il figlio si trova **stack** popolato con la storia delle chiamate effettuate nel padre, **heap** con della memoria allocata, **codice** e **spazio globale** nello stesso stato in cui erano nel padre; inoltre, dopo una **fork**, tutte le pagine sono marcate dal flag **CopyOnWrite** (cioè duplica se modifica, così padre e figlio avranno 2 versioni diverse se 1 dei 2 modifica).

Le funzioni **exec*()** sostituiscono l'attuale immagine di memoria dell'address space ri-inizializzandola a quella descritta dall'eseguibile indicato come parametro. Vediamo un esempio di C++:

```
int main ( const int argc, const char* const argv[] ) {
    pid_t childPid = fork();
    switch (childPid) {
        case -1:
            puts( "parent: error: fork failed!" );break;
        case 0:
            puts( "child: here (before execl)" );
            if (execl( "./ch.exe", "./ch.exe", 0 )==-1)
                perror( "child: execl failed:" );
            puts( "child: here (after execl)!" );
            //non si dovrebbe arrivare qui
            break;
        default:
            printf( "par: child pid=%d \n", ret );
            break;
    }
    return 0;
}
```

Ma con **programmi concorrenti** (**multi-thread**) la **fork** genera problemi in quanto il processo figlio conterrà 1 solo thread e la sincronizzazione è sballata rispetto al padre. Posso usare quindi la **int pthread_atfork(void (*prepare)(void),void (*parent)(void),void (*child)(void))** che registra un **gruppo di funzioni da chiamare** se viene invocata la **fork**.

Terminare un processo? Un processo continua fino a che non termina o viene terminato dall'esterno; in Windows **ExitProcess(int status)**, in Linux **_exit(int status)**, entrambe causano l'immediata terminazione di tutti i thread associati al processo e la chiusura di tutti i file aperti. Le librerie **C/C++** hanno **exit(int status)** e **std::exit(int status)**; con **std::atexit(void (*callback)())** si associa callback alla terminazione del processo. Un programma però termina anche quando la funzione principale (**main**) ritorna (chiamando internamente la **exit(status)** dove status è il valore tornato dal **main**); il valore restituito da **exit(...)** è arbitrario.

Dopo questa parte introduttiva sui processi, vediamo ora la **GESTIONE DEI PROCESSI IN RUST (std::process)**. La struct **Command** permette la **creazione di un nuovo processo**: i metodi **arg()** e **args()** possono essere usati per passare al processo figlio 1 o più argomenti; il metodo **output()** genera il processo e attende la sua terminazione ritornando un valore **Result<Output>** dove struct **Output** contiene **status** (tipo **ExitStatus**) e **stdout** e **stderr** (2 **Vec<u8>**).

```
use std::process::Command;

fn main() {
    let output = if cfg!(target_os = "windows") {
        Command::new("cmd")
            .args(["/C", "echo hello"])
            .output()
            .expect("failed to execute process")
```

Possiamo anche aggiungere/rimuovere/vedere le **variabili d'ambiente** del processo prima di avviarlo con `env<K, V>(&mut self, key:K, val:V)`, `envs<I,K,V>(&mut self, vars:I)`, `env_remove<K>(&mut self, key:K)`, `env_clear(&mut self)` e `get_envs(&self)`.

Possiamo ridigere i flussi standard con i metodi `stdin(...)`, `stdout(...)` e `stderr(...)` passando loro 1 delle opzioni:

- `inherit()` → processo figlio eredita descrittore in uso nel padre
- `piped()` → creata una pipe monodirezionale con da 1 estremità il processo figlio, dall'altra sarà memorizzata una struttura restituita dal comando di avvio
- `null()` → flusso ignorato

Con `current_dir<P: AsRef<Path>>(&mut self, dir:P)` possiamo modificare la directory dove si avvia il figlio

Abbiamo poi i metodi:

- `status(&mut self)` → avvia il processo, ne attende la terminazione; restituisce un valore di tipo `Result<ExitStatus>` con `ExitStatus` che permette di avere info su codice di uscita e motivo di terminazione
- `spawn(&mut self)` → avvia il processo senza attendere la terminazione; restituisce un valore di tipo `Result<Child>` con `Child` struttura che consente di rappresentare e interagire con il processo figlio, attraverso:
 - `campi stdin, stdout, stderr`
 - metodi:
 - `id(&self)` = ritorna pid
 - `wait(&mut self)` = attende la terminazione, restituendo il codice di uscita
 - `wait_with_output(&mut self)` = chiude stdin del figlio, ne attende la terminazione e ritorna quanto non ancora letto da stdout e stderr in una struct `Output`
 - `kill()` = forza terminazione

```
use std::io::Write;
use std::process::{Command, Stdio};

let mut child = Command::new("rev")
    .stdin(Stdio::piped())
    .stdout(Stdio::piped())
    .spawn()
    .expect("Failed to spawn child process");

let mut stdin = child.stdin.take().expect("Failed to open stdin");
std::thread::spawn(move || {
    stdin.write_all("Hello, world!".as_bytes())
        .expect("Failed to write to stdin");
});

let output = child.wait_with_output().expect("Failed to read stdout");
assert_eq!(String::from_utf8_lossy(&output.stdout), "!dlrow ,olleH");
```

⚠ Se in Linux un processo fa `fork()`, tutto lo stato della sua memoria sarà duplicato, ma se nei **buffer di I/O** (`stdin, stdout e stderr`) c'è del contenuto **pending**, questo verrà poi mandato al `flush()` 2 volte, ovvero 1 dal padre e 1 dal figlio; quindi conviene sempre fare una `flush()` di tutti i file aperti (anche `std::cout` e `std::cerr`) **prima della `fork()`** proprio per evitare questo invio duplicato.

Terminare un processo in Rust?

- `std::process::exit(code:i32) -> ! = termina` immediatamente il processo corrente con tutti i suoi threads
- `std::process::abort(code:i32) -> ! = termina` immediatamente il processo corrente con tutti i suoi threads, ma segnala che è un'interruzione anomala
- `panic!` = causa la contrazione dello stack corrente, con l'esecuzione di tutti i distruttori posti al suo interno, senza determinare la **terminazione** del processo (a meno che la chiamata avvenga dal thread principale)

Nella **gestione di altri processi**, ciascun OS offre funzioni diverse:

- `WaitForSingleObject(...), WaitForMultipleObjects(...)` e `GetExitCodeProcess(...)` in Windows
- `wait(...), waitpid(...)` e `waitid(...)` in Linux:

- `pid_t wait(int *status)`
 - se processo non ha figli → return -1
 - se nessuno dei figli dell'attuale processo è terminato, attendiamo che 1 termini

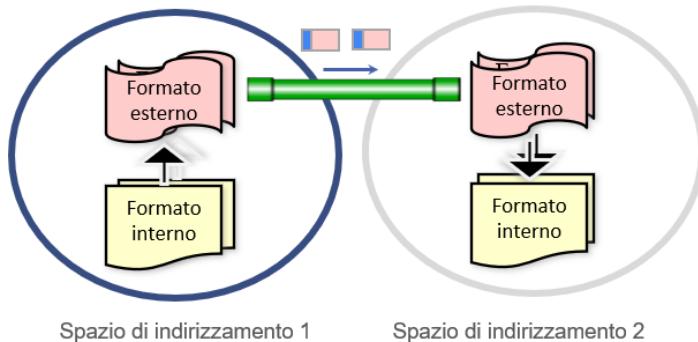
- quando termina figlio:
 - se status != null → all'interno troviamo il motivo della terminazione
 - sistema aggiorna contatori di uso del sistema inserendo quelli del processo figlio
 - ritorna pid del figlio terminato
- **pid_t waitpid(pid_t pid, int *status, int options)** → permette di fare una wait non bloccante su uno specifico figlio e non 1 a caso

⚠ Se processo padre termina prima che figlio sia terminato, figlio diventa **ORFANO** (il padre di questo diventerà il processo con PID = 1, ovvero l'**init**); se processo figlio termina prima che il padre abbia fatto la **wait**, figlio diventa **ZOMBIE**

Come rappresento le informazioni scambiate dai processi? Spesso la rappresentazione interna dei dati non è adatta ad essere esportata (es. i puntatori non hanno senso fuori dall'address space): si usa **rappresentazione esterna** (formati testuali [JSON, CSV...], binari [XDR, ...] o riferimenti indipendenti), ovvero **trattabile come blocchi compatti di byte senza perdere significato (SERIALIZZAZIONE)**. La sorgente esporta le proprie info in formato esterno (*marshalling*) e il destinatario ricostruisce (*unmarshalling*).

La **coda di messaggi** è un esempio di ciò: è una struttura dati mantenuta dall'OS per permettere a molti processi sorgente di inviare messaggi ad un destinatario (comunicazione **asincrona** e **mono-direzionale**); ogni coda è caratterizzata da un id univoco nell'OS (in Windows oggetti **mailslot**, in Linux oggetti **fifo**). Ovviamente la sequenza è: P1 → formato interno → formato esterno → coda di msg → formato esterno → formato interno → P2

Se voglio invece comunicazione **sincrona** posso usare le **PIPE** (tubi che permettono il trasferimento di sequenze di byte di dimensioni arbitrarie) [implementati come buffer nella memoria kernel]:



Si creano in Linux con **int pipe(int fd[2])** e le operazioni di lettura/scrittura che eseguo su di esse sono le stesse con cui interagisco con l'OS (quindi le **normali** **read** e **write**); si chiudono con **close()**. Pipe in **RUST**:

```
let echo_child = Command::new("echo")
    .arg("Oh no, a typo!")
    .stdout(Stdio::piped())
    .spawn()
    .expect("Failed to start echo process");

let echo_out = echo_child.stdout.expect("Failed to open echo stdout");

let mut sed_child = Command::new("sed")
    .arg("s/typo/typo/")
    .stdin(Stdio::from(echo_out))
    .stdout(Stdio::piped())
    .spawn()
    .expect("Failed to start sed process");

let output = sed_child.wait_with_output().expect("Failed to wait on sed");
assert_eq!(b"Oh no, a typo!\n", output.stdout.as_slice());
```

Per quanto riguarda lo **scambio di messaggi con strutture tra processi in Rust**, posso usare il crate **serde**: questa libreria estende la macro **#[derive(Serialize, Deserialize)]** per aggiungere in modo automatico il supporto alle conversioni a tipi definiti dall'utente. Vediamo alcune regole di serializzazione JSON →

```
struct W {
    a: i32,
    b: i32,
}
let w = W { a: 0, b: 0 }; // Rappresentato come l'oggetto `{"a":0,"b":0}`

struct X(i32, i32);
let x = X(0, 0); // Rappresentato come l'array `[0,0]`

struct Y(i32);
let y = Y(0); // Rappresentato come il solo valore `0`

struct Z;
let z = Z; // Rappresentato come `null`
```

E ora vediamo un esempio di serializzazione con serde:

```
use serde::Serialize, Deserialize;

#[derive(Serialize, Deserialize, Debug)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let point = Point { x: 1, y: 2 };

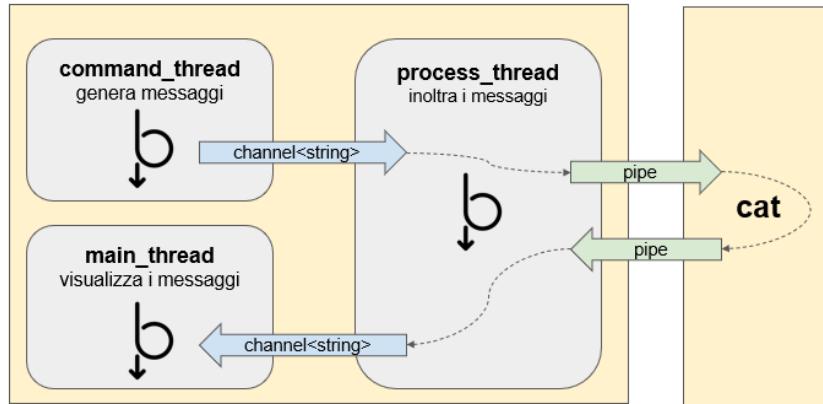
    let serialized = serde_json::to_string(&point).unwrap();

    println!("{}: {}", "serialized", serialized); // {"x":1,"y":2}

    let deserialized: Point = serde_json::from_str(&serialized).unwrap();

    println!("{}: {}", "deserialized", deserialized); // Point { x: 1, y: 2 }
}
```

Il crate [interprocess](#) offre la possibilità di gestire la comunicazione tra processi tramite un'interfaccia univoca multipiattaforma:



Esempio di comunicazione tra processi:

```
use std::io::{BufRead, BufReader, Write};
use std::process::{Command, Stdio};
use std::sync::mpsc::{channel, Receiver, Sender};
use std::thread;
use std::thread::sleep;
use std::time::Duration;

fn start_process(sender: Sender<String>, receiver: Receiver<String>) {

    let child = Command::new("cat")
        .stdin(Stdio::piped())
        .stdout(Stdio::piped())
        .spawn()
        .expect("Failed to start process");

    thread::spawn(move || {
        let mut pipe_in = BufReader::new(child.stdout.unwrap());
        let mut pipe_out = child.stdin.unwrap();
        for line in receiver {
            pipe_out.write_all(line.as_bytes()).unwrap();
            let mut buf = String::new();
            match pipe_in.read_line(&mut buf) {
                Ok(_) => {
                    // inoltra quanto ricevuto dalla pipe sul canale di uscita
                    sender.send(buf).unwrap();
                    continue;
                }
                Err(e) => {
                    println!("an error!: {:?}", e);
                    break;
                }
            }
        }
    })
}
```

```

    }

fn start_command_thread(sender: Sender<String>) {
    thread::spawn(move || {
        for i in 1..10 {
            sleep(Duration::from_secs(3));
            sender.send(String::from(format!("Message {} from command thread\n", i)))
                .unwrap();
        }
    });
}

fn main() {
    let (tx1, rx1) = channel();
    let (tx2, rx2) = channel();

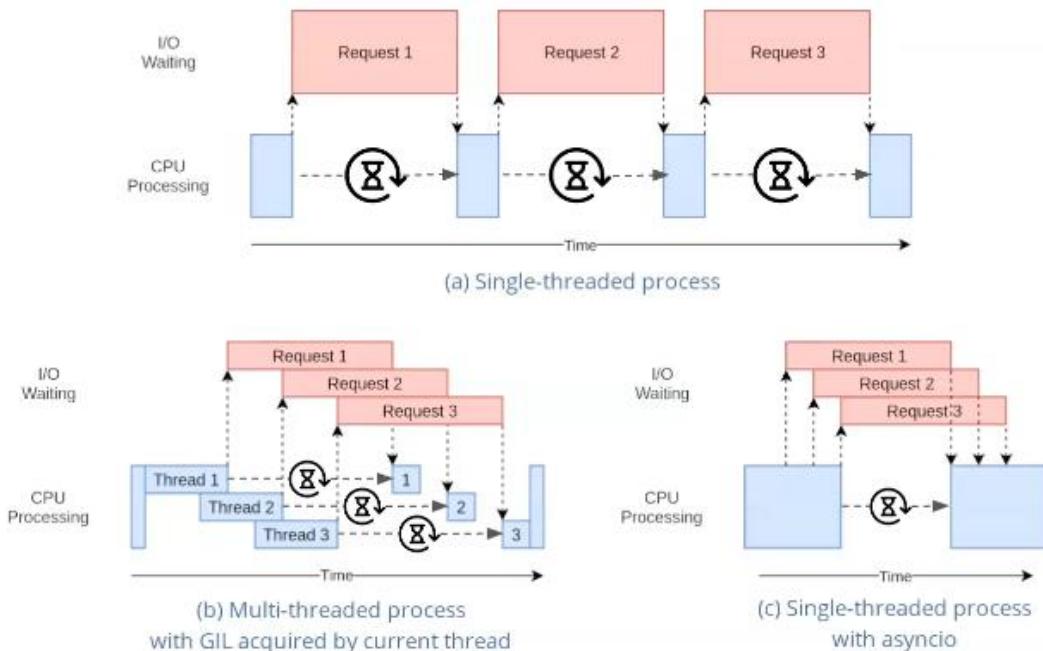
    start_process(tx1, rx2);
    start_command_thread(tx2);

    rx1.iter().for_each(|line| println!("Echo process response: {}", line))
}

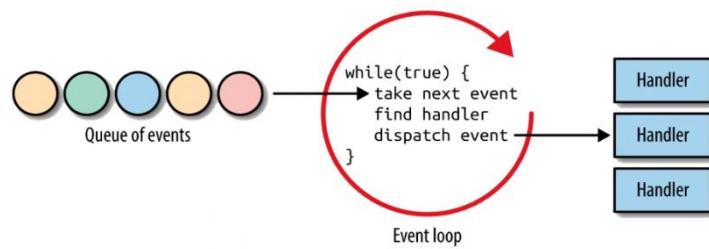
```

13) PROGRAMMAZIONE ASINCRONA

Eseguire più operazioni in parallelo mediante multi-threading (programmi concorrenti) è efficiente in hardware multicore, ma al tempo stesso ha alti **costi e complessità** legati ai meccanismi di sincronizzazione. Inoltre, è inefficiente nel caso di **OPERAZIONI BLOCCANTI**, ovvero quando devo **comunicare con un sottosistema separato** e devo **attendere** delle informazioni da questo (come abbiamo detto si possono anche far partire altri thread mentre il thread è in attesa delle informazioni, ma devono comunque essere presenti altre operazioni da svolgere oltre all'attesa). **Strategie di esecuzione:**



Usiamo **multi-thread** quando occorre **elaborare in parallelo**, usiamo **esecuzione asincrona** (con 1 thread, guarda immagine sopra) quando occorre **attendere in parallelo**: per farlo mettiamo le azioni conseguenti ad un'operazione bloccante in una **callback** (la callback riceve come parametro il **risultato dell'operazione bloccante**, ma come viene fornito alla callback e in quale thread avverrà la chiamata? → in JavaScript per esempio c'è una coda di messaggi in cui il driver del sottosistema interrogato mette il risultato, mentre thread principale attende):



Un modo per implementare le operazioni asincrone è basarsi su **API ad eventi (event handlers)** [come in JS]:

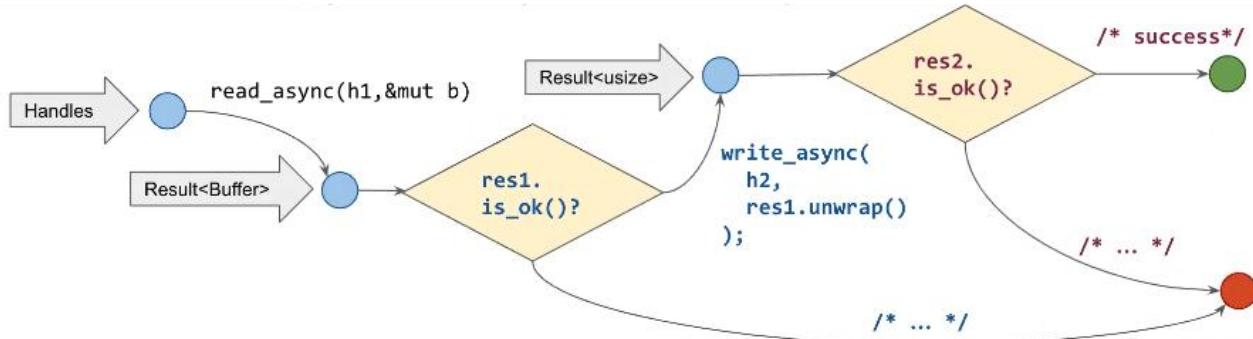
```
val f1: AsyncRead = ...
val f2: AsyncRead = ...
read_async(f1, vec![], |buffer: &[u8]|{
    // process buffer from file1...
});
read_async(f2, vec![], |buffer: &[u8]|{
    // process buffer from file2
});
```

⚠ **async** trasforma la nostra funzione in una **FSM** (macchina a stati), ovvero tanti chunk avanzando alla 1^a **await**

Se abbiamo **più operazioni asincrone in cascata**, viene a generarsi “l’**inferno delle callback**” in quanto ci sono callback che chiamano callback e ci possono essere errori in punti diversi del programma. Una 1^a soluzione è riscrivere le callback da forma **annidata** a forma **lineare** (appoggiandosi ad una struttura **future** che mantenga le informazioni di stato) [rappresentabile come una **FSM** che mostra un’**esecuzione parziale a step perché lineare**]:

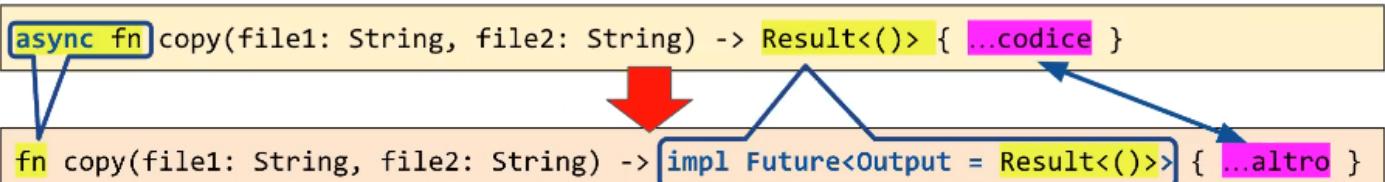
```
read_async(h1, &mut buffer, |res1|{
    if (buffer.is_ok()) {
        write_async(h2, res1.unwrap(), |res2| {
            if (res2.is_ok()) { /* success */ }
            else { /* ... */ }
        });
    } else {
        // lettura fallita
    }
})?;
```

```
read_async(h1, &mut buffer) ← Future<Result<u8>>
    .and_then(|res1| {
        if (res1.is_ok()) {
            write_async(h2, res1.unwrap());
        } else { /* ... */ }
    }) → Future<Result<u8>>
    .and_then(|res2| {
        if (res2.is_ok()) { /* success */ }
        else { /* ... */ }
    }) → Future<Result<()>>
    .map_error(|err| { /* ... */ });
```



Con questa visione, la FSM può essere implementata da una **closure (chiusura)** che racchiude il suo stato e tutte le variabili locali di cui l’esecuzione ha bisogno; quando viene eseguita torna un valore che indica se ha raggiunto **stato finale** o se ancora in **stato intermedio** → questo permette di **non bloccare il thread corrente** e procedere con l’esecuzione di altri task. I **punti di blocco** sono gli **stati intermedi**: questi sono preceduti dall’invocazione di una funzione asincrona; quando uno stato intermedio viene raggiunto, la closure ritorna. Se si verificano tutte le condizioni richieste, arriverò ad 1 degli stati finali.

Rust supporta la programmazione asincrona con le parole chiave **async** e **await** e con il tipo **Future**; se funzione/ blocco di codice è preceduto da **async**, questo viene **trasformato dal compilatore Rust in una FSM**.



Il tipo restituito viene trasformato da `T` in un **tipo anomino** che implementa il tratto **Future** al cui interno è implementata la FSM precedentemente sintetizzata. Se all'interno del codice della funzione preceduta da `async` è presente una chiamata ad un'altra funzione asincrona, per accedere al suo risultato occorre fare `.await` (questo introduce un nuovo stato nella FSM associata alla funzione):

```
async fn copy(h1: FileHandle, h2: FileHandle) -> std::io::Result<()> {
    let mut buffer = vec![];
    h1.read_async(&mut buffer).await?;
    h2.write_async(&buffer).await
}
```

Il tratto **Future** è fatto così:

```
use std::pin::Pin;
use std::task::{Context, Poll};

pub trait Future {
    type Output;

    fn poll(self: Pin<&mut Self>, cx: &mut Context)
        -> Poll<Self::Output>;
}
```

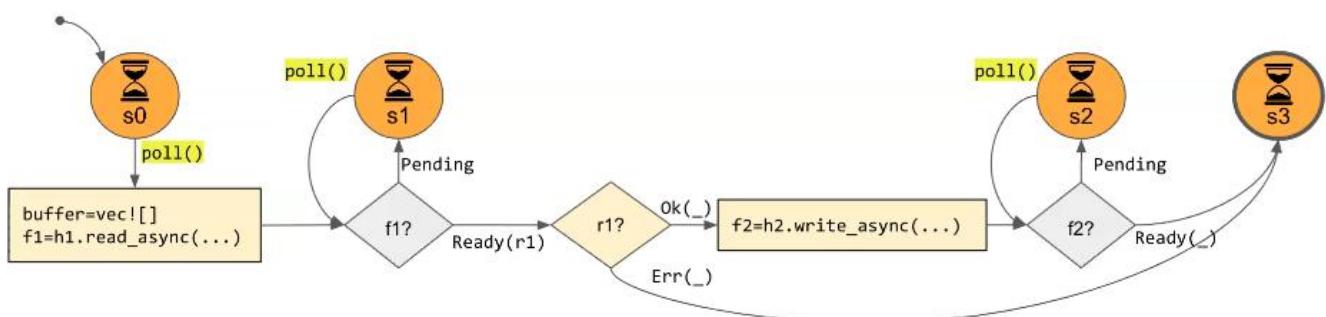
```
pub enum Poll<T> {
    Ready(T),
    Pending,
}
```

dove `Pin<T>` è uno smart-pointer che impedisce che struttura venga mossa (permette uso di riferimenti relativi) e `Context` incapsula un oggetto di tipo `Waker` (mediante cui posso notificare all'esecutore che il metodo `poll(...)` può essere richiamato) [`poll` restituisce un enum `Poll<T>` che indica: se `Pending`, computazione ancora in corso; se `Ready(val)`, computazione terminata con risultato `val`; `poll` implementa la logica della FSM associata alla func]

⚠ Questo tratto è direttamente il compilatore ad implementarlo in presenza di un blocco `async`; un oggetto che implementa questo tratto è “**inerte**” (affinchè la computazione proceda, qualcuno deve invocare il metodo `poll`)

Generare la FSM:

```
async fn copy(h1: FileHandle, h2: FileHandle) -> std::io::Result<()> {
    let mut buffer = vec![];
    h1.read_async(&mut buffer).await?;
    h2.write_async(&buffer).await
}
```



Per ogni stato individuato, il compilatore crea una **struct** con tutte le variabili locali per andare avanti:

```
struct S0 {
    h1: FileHandle,
    h2: FileHandle,
}
```

```
struct S1 {
    h2: FileHandle,
    buffer: Vec<u8>,
    f1: impl Future<Output=Result<usize>>,
}
```

```
struct S2 {
    f2: impl Future<Output=Result<usize>>,
}
```

```
struct S3 {}
```

Viene però anche generato un **enum** (che implementa **Future**) che **racchiude tutti i possibili stati** (in modo da capire in che stato sono attualmente):

```
enum CopySM {
    s0(S0),
    s1(S1),
    s2(S2),
    s3(S3)
}
```

```
impl Future for CopySM {
    type Output = std::io::Result<()>
    fn poll(self: Pin<&mut Self>, cx: &mut Context) -> Poll<Self::Output> {
        loop {match self {
            CopySM::s0(state) => { ... },
            CopySM::s1(state) => { ... },
            CopySM::s2(state) => { ... },
            CopySM::s3(state) => { ... },
        } }
    }
}
```

Quindi da qui genero **tutti i passaggi tra i vari stati della mia FSM**:

```
CopySM::s0(state) => {
    let mut buffer = vec![];
    let f1 = state.h1.read_async(&mut buffer);
    let state = S1 {h2: state.h2, buffer, f1};
    *self = CopySM::S1(state);
}
```

```
CopySM::s1(state) => {
    match (state.f1.poll(cx)) {
        Poll::Pending => return Poll::Pending,
        Poll::Ready(r1) =>
            if r1.is_ok() {
                let f2 = state.h2.write_async(&mut state.buffer);
                let state = S2{ f2 };
                *self = CopySM::s2(state);
            } else {
                *self = CopySM::s3(S3);
                return Poll::Ready(r1);
            }
    }
}
```

```
CopySM::s2(state) => {
    match (state.f2.poll(cx)) {
        Poll::Pending => return Poll::Pending,
        Poll::Ready(r2) =>
            *self = CopySM::s3(S3);
            return Poll::Ready(r2);
    }
}
```

```
CopySM::s3(_) => {
    panic!("poll() was invoked again after Poll::Ready has been returned");
}
```

E il compilatore mi avrà generato dalla funzione l'**entry point per la FSM**, ovvero:

```
fn copy(h1: FileHandle, h2: FileHandle) ->
    impl Future<Output = std::io::Result<()>> {
        CopySM::s0(
            S0 { h1, h2 }
        )
    }
}
```

⚠ Se la funzione asincrona viene chiamata all'interno di un'altra funzione asincrona, diventa automaticamente **parte della FSM del chiamante**

⚠ Se si invoca una funzione asincrona all'interno di una funzione normale, bisogna gestire esplicitamente il risultato di tipo **Future** mediante un **Executor**

Il compilatore di Rust genera automaticamente i tipi che implementano la FSM associata a funzione asincrona, ma **non c'è supporto invece per la gestione dell'esecuzione**: dobbiamo scegliere noi quale libreria adottare nel nostro progetto:

- **Tokio** (ambiente **più diffuso**; supporto per connessioni di rete e DB)
- **smol** (ambiente semplificato; usato in embedded)
- **async-std** (ambiente che offre la controparte asincrona delle librerie standard bloccanti)

Tokio è **multi-thread** (**delega l'esecuzione di più Future in parallelo**; quindi tutti i valori usati in più stati della funzione asincrona devono implementare il tratto **Send**, mentre i riferimenti devono implementare il tratto **Sync**) [usa N thread se ho N core sulla mia macchina, ma questo N è modificabile da noi]

Come sempre, **per usare Tokio** devo scaricarlo e metterlo nel **Cargo.toml** e poi devo creare **tokio::main**:

```
#[tokio::main(flavor = "multi_thread", worker_threads = 4)] //o altro...
async fn main() {
    //... creazione di future e attesa relativa
}
```

Dopo aver creato **tokio::main**, posso **definire un task** eseguito in modo concorrente con altri task, con esecuzione che inizia subito (a differenza di Future dove va invocato `.await`) tramite **tokio::task::spawn(f: T)**:

```
#[tokio::main]
async fn main() {
    let task = tokio::spawn(async { println!("Hello, Tokio!"); });
    task.await.unwrap()
}
```

⚠ Quindi usiamo **spawn** esattamente come lo usavamo per creare dei thread, ma ora creiamo delle task che verranno passati ai thread con l'obiettivo di saturare l'uso della CPU.

Si può usare **join!(f1: impl Future, ..., fn: impl Future)** all'interno di un blocco **async** per forzare l'attesa fino a che tutti i suoi parametri sono completati (**aspetto la terminazione di più Future**). Una versione specializzata è **try_join!(...)** usabile quando le espressioni passate come parametro hanno **Result<T,E>** come return (questo **Result** contiene una tupla con i risultati solo se tutti i Future hanno avuto successo, altrimenti dà errore al 1° fallimento):

```
async fn do_stuff_async() { ... }
async fn more_async_work() { ... }

#[tokio::main]
async fn main() {
    let (first, second) = tokio::join!(
        do_stuff_async(),
        more_async_work()
    );
    // do something with the values
}
```

```
async fn do_stuff_async() { ... }
async fn more_async_work() { ... }
```

```
#[tokio::main]
async fn main() {
    tokio::select! {
        _ = do_stuff_async() => {
            println!("do_stuff_async() completed first")
        }
        _ = more_async_work() => {
            println!("more_async_work() completed first")
        }
    };
}
```

C'è poi **select!(...)** che permette di attendere su più rami asincroni (eseguiti nell'ambito dello stesso thread) **quello che termina per 1°** →

⚠ Il singolo ramo nella select ha condizioni nella forma: <pattern> = <async expression> (,if <pre-condition>)? => <handler> else => <expression>

Tokio offre anche **gestione del tempo**:

- `tokio::time::sleep(d: Duration).await` → sospende l'esecuzione del task corrente per $t = d$
- `tokio::time::timeout(d: Duration, f: F).await` → attende $t_{max} = d$ che il **Future** passato come 2° parametro si completi e restituisca un valore **Result<T, Elapsed>**

Se in una funzione asincrona occorre eseguire, oltre a tutte le computazioni, una **computazione intensa e lunga**, si può richiedere che venga **eseguita da un thread apposito** per non bloccare tutto il sistema e lo si fa con `tokio::task::spawn_blocking(f: FnOnce() -> R)` [non ne vanno lanciati troppi così]; questo thread vive in un altro thread-pool rispetto ai thread normali

Se 2 task devono **condividere una struttura dati**, questa va protetta: ci sono primitive asincrone in `tokio::sync` (alcune basate sulla **condivisione dello stato** [**Barrier**, **Mutex**, **Notify**, **RwLock**, **Semaphore**], altre basate sulla **comunicazione di messaggi** [**oneshot**, **mpsc**, **broadcast**, **watch**]). Vediamoli nel dettaglio.

Come uso **Arc/Mutex**? (condivisione dello stato):

```
use tokio::sync::Mutex;
use std::sync::Arc;
#[tokio::main]
async fn main() {
    let data = Arc::new(Mutex::new(0));
    let mut v = vec![];
    for _ in 0..4 {
        let data = Arc::clone(&data);
        v.push(tokio::spawn(async move {
            let mut lock = data.lock().await;
            *lock += 1;
        }));
    }
    for h in v { let _ = join!(h); }
    assert_eq!(*data.lock().await, 4);
}
```

Come uso **canali oneshot**? (comunicazione di messaggi, ma **1 solo messaggio**):

```
async fn some_computation() -> String { "Some result".to_string() }

#[tokio::main]
async fn main() {
    let (tx, rx) = oneshot::channel();

    tokio::spawn(async move {
        let res = some_computation().await;
        tx.send(res).unwrap();
    });

    // Do other work while the computation is happening in the background

    // Wait for the computation result
    let res = rx.await.unwrap();
}
```

Come uso **canali mpsc**? (condivisione di messaggi, **multiple-producer single-consumer**):

```
async fn some_computation(i: u32) -> String { format!("Value {}", i) }

#[tokio::main]
async fn main() {
    let (tx, mut rx) = mpsc::channel(100);

    tokio::spawn(async move {
        for i in 0..10 {
            let res = some_computation(i).await;
            tx.send(res).await.unwrap();
        }
    });

    while let Some(res) = rx.await.unwrap() { println!("{}: {}", i, res); }
}
```

Come uso canali **broadcast**? (comunicazione *N-N*) [in questo esempio canale broadcast può avere max 16 elementi pending]:

```
#[tokio::main]
async fn main() {
    let (tx, mut rx1) = broadcast::channel(16);
    let mut rx2 = tx.subscribe();

    tokio::spawn(async move {
        assert_eq!(rx1.recv().await.unwrap(), 10);
        assert_eq!(rx1.recv().await.unwrap(), 20);
    });
    tokio::spawn(async move {
        assert_eq!(rx2.recv().await.unwrap(), 10);
        assert_eq!(rx2.recv().await.unwrap(), 20);
    });
    tx.send(10).unwrap();
    tx.send(20).unwrap();
}
```

Come uso canali **watch**? (pattern **Observer**):

```
#[tokio::main]
async fn main() {
    let (tx, mut rx) = watch::channel("value 0");

    for i in 0..2 {
        let mut rx = rx.clone();
        tokio::spawn(async move {
            while rx.changed().await.is_ok() {
                println!("received: {:?}", *rx.borrow());
            }
        });
    }
    let d = Duration::from_secs(1);
    tx.send("value 1").unwrap(); tokio::time::sleep(d).await;
    tx.send("value 2").unwrap(); tokio::time::sleep(d).await;
}
```

Esempio di semplice implementazione di **server HTTP** tramite Tokio:

```
use tokio::io::AsyncWriteExt;
use tokio::net::{TcpListener, TcpStream};
use tokio::task;

#[tokio::main]
async fn main() {
    let listener = TcpListener::bind("127.0.0.1:8181").await.unwrap();

    loop {
        let (stream, _) = listener.accept().await.unwrap();
        tokio::spawn(handle_connection(stream));
    }
}

async fn handle_connection(mut stream: TcpStream) {
    let contents = "{\"message\": \"Hello, Tokio!\"}";

    let response = format!(
        "HTTP/1.1 200 OK\r\nContent-Type: application/json\r\nContent-Length: {}",
        contents.len(),
        contents
    );

    stream.write(response.as_bytes()).await.unwrap();
    stream.flush().await.unwrap();
}
```

⚠ Creare un **task** (entità user-space), il context switching di un task e l'uso di memoria di un task sono tutti molto minori di quanto richiesto da un thread (quindi **task meno costosi a livello di spazio e tempo dei thread**)

FINE