Gestione degli errori

Tutte le computazioni possono fallire. **Fallimento causa**:

- Non ricevo dato atteso
- Se computazione produceva effetti collaterali (scrivere file su disco), non so dire di per se cosa succeda
    - Es: db correggono questa situazione con il concetto di transazione → butta via modifiche fatte da inziio transazione a punto attuale → nei sistemi operativi questo concetto non esiste.
- **Cause di fallimenti**:
    - **Prevedibili**
        - Argomenti illeciti
        - Conversione testo numero
    - Possibili ma **non** proprio prevedibili
        - Memoria/spazio disco esauriti
        - Malfunzionamento della rete o di periferiche
- Fallimenti possono essere di **2 tipi principali**:
    - Malfunzionamenti **recuperabili**
        - Non caompromettono lo stato del programma → si può fare ripristino
        - RIPRISTINO
            - Ritentare operazione
                - Potrebbe funzionare aggiungendo una pausa, potrebbe esser stato causato da problema momentano di rete
            - Richiedere intervento utente/admin
            - Utilizzare strategia alternativa
                - In alcuni casi sono presenti dei metodi veloci per fare le cose e altri noiosi ma più safe
    - Malfunzionamenti **non** recuperabili
        - Causano alterazione impredicibile dello stato o che indicano l'impossibilità di procedere con ulteriore computazione
        - TERMINO PROCESSO

Nota: non è detto che il punto in cui si verifica il fallimento sia a conoscenza di informazioni tali da comprendere come comportarsi

- Occorre fare in modo che, in caso di fallimento della computazione all'interno di una funzione, il **controllo torni al suo chiamante**, corredato di una opportuna descrizione di quanto successo
    - Segnalo che si è rotto qualcosa e specifico cosa si è rotto
    - Eccezioni

Rust offre tipi algebrici → **Result<T, E>** e **Option<T>** per esprimere gli esiti delle computazioni

- **Result**
    - Nota per E: meglio che sia un tipo che implementa il tratto Error
- **Option**, ritorna
    - Risultato
    - None → non è stato possibile calcolare il risultato ma non ti dico il perché
- Offre inoltre la macro **panic!(..)** per forzare l'interruzione del thread corrente producendo una descrizione testuale di quanto successo.

NOTA SU ALTRI LINGUAGGI

In C++ l'eccezione può essere un qualunque dato; una funzione al suo interno può contenere l'istruzione throw, che, quando viene eseguita prende un dato qualunque e forza il ritorno dalla funzione stessa.

Nota su c++: return mi fa tornare alla riga successiva. Throw mi fa arrivare in quella linea e mi guardo interno per vedere se il blocco è in un blocco try. Se è in un blocco try, salto alla sua fine e vedo se il mio errore è presente in uno dei suoi catch. Catch ha il compito di sistemare tutto.

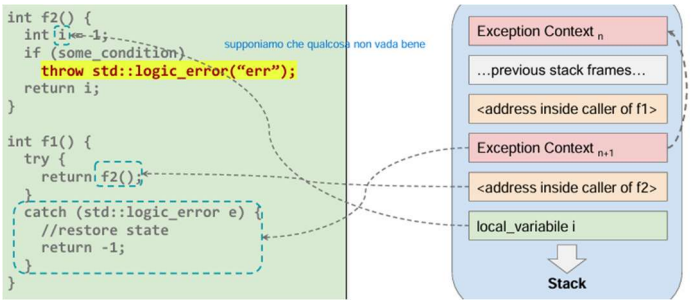
```
try {
  //codice che può fallire direttamente o indirettamente
} catch (ExceptionType1 e1) {
   //…istruzioni di ricupero
} catch (ExceptionType2 e2) { … }
```

Nel caso in cui non sia racchiuso in un try, vado al chiamante del chiamante e così via… Se nessuno lo ha, torno all'inziio del thread e, in C++, il programma si arresta con codice di errore !=0.

- Se sono thread principale → exit
- Se sono thread secondario → terminate

Nota su java: su java c'è una tipologia di exception, runtimeException che è esentata. Ha come sottoclasse nullPointerException. Inoltre java permetteva Finally → comunque sia andata sta roba, bene o male → esegui il blocco finally.

Ritorno a C++: processore deve salvarsi nello stack l'Exception Context. Il più recente Exception Context è salvato in un registro dedicato –immagini slide 11



RAII: Resourse Acquisition Is Initialization → si possono usare i distruttori deglinogegtti per mestter apoosto delle cose. Le azioni del distruttore devono potersi fare in ogni caso.

Nota: il compilatore non è in grado di accertare dove avvengono eccezioni

Nota: in alcuni casi un'eccezione può a sua volta causare eccezione

- Es: file non creato
  - Se si scende nei controlli si scopre che era stata causata da altra eccezione che comunicava disco pieno.

RUST

Usa **Result<T,E>**

- **T**, tipo che descrive cosa ritorni se va bene
- **E**, tipo che descrive cosa ritorni se va maleù

```
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),
  }
}
```

Metodi per Result:

- I metodi **`is_ok(&self)`** e **`is_err(&self)`** permettono, rispettivamente, di determinare se l'esito di un'operazione ha avuto successo o meno
- I metodi **`ok(self)`** e **`err(self)`** consumano il risultato trasformandolo in un oggetto di tipo **`Option<T>`** piuttosto che **`Option<E>`**
- Il metodo **`map(self, op: F)->Result<U,E>`** applica la funzione al valore contenuto nel risultato, se questo è ok, altrimenti lascia l'errore invariato
- Il metodo **`contains(&self, x: &U)`** restituisce vero se il risultato è valido e contiene un valore che equivale all'argomento
- Il metodo **`unwrap(self)->T`** restituisce il valore contenuto, se è valido, ma invoca la macro **`panic!(...)`** se il risultato contiene un errore

Accetta messaggio per indicare errore

Se c'è un male unrecoverable → uso **panic**

- Se ci sono dei distruttori pendenti che avevano seganto qualcosa da fare, vengono eseguiti
- Se il thread che ha invocato panic è principale → cessa tutto
- Se il thread che ha invocato panic è secondario → solo lui temrina, gli altri continuano a fare quello che dovevano

Il tipo Result<T, E, ha dellle funzioni aggiuntive:

- **unwrap** → sono certo che è andato bene quindi prendilo (se va male panica)
- **expect** → prendilo ma se non riesci a prenderlo stampa questa cosa (se va male stampa cosa)

nota: singole righe che possono fallire, falle segire dal **punto interrogativo ?** → verifica internamente se Ok o Err e ritorna quello che ottieni.

In questo caso va bene perché file:open e red_to_string restituiscono lo stesso tipo di errore (io::Error)

```
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)
}
```

Errori eterogenei

In alcuni casi è importante che l'errore ritornato dal? sia lo stesso ritornato dalla funzione dove si scaturisce.

- Rust offre diversi modi per propagare errori eterogenei, la scelta spetta al programmatore sulla base delle sue esigenze

Per questo libreria stadandard ha **implementazione generica del tratto From** → posso ritornare un Boc<dyn error> → **trasferisco error su heap e do il puntatore**.

```
impl<E: error::Error> From<E> for Box<dyn error::Error>;
```

- Gli oggetti-tratto richiedono l'utilizzo di fat pointer e vtable con il conseguente costo in termini di memoria
- Durante la conversione vengono perse le informazione sul tipo dell'errore
- Si può risalire allo specifico errore tramite l'utilizzo del downcast_ref() a patto che si conosca l'implementazione della funzione che genera gli errori
- La conversione può avvenire in maniera implicita attraverso l'utilizzo dell'operatore ?

```
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) ? ; // io::Error -> Box<dyn error::Error>
  let mut sum = 0;
  for line in contents.lines() {
    sum += line.parse::<i32>() ? ;       // 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::<io::Error>() {...}  //tratto io::Error
      else if let Some(e) = err.downcast_ref::<ParseIntError>() {...}  //tratto ParseIntError
      else { unreachable!(); }  //non può capitare
    }
```

Downcast ref si chiede il tipo dell'errore verificato e in base a quello do il messaggio opportuneo.

Propagare errori eterogenei: è possibile implementare degli **errori custom** in modo tale da propgare errori eteorgenie senza forzare il sistema dei tipi:

- Errori custom devono implementare il tratto **Error** e i tratti **Debug** e **Display**
- Utilizzo di enum permette di racchiudere i diversi tipi di errore da gestire con **match**
- Necessario implementare tratto **From** per convertire i diversi errori nel tipo custom da propagare

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

```
#[derive(Error, Debug)]
enum SumFileError {

  #[error("IO error {0}")]
  Io(#[from] io::Error),

  #[error("Parse error {0}")]
  Parse(#[from] ParseIntError),
}
```
MAIN

```
#[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::<i32>()?;
  }
  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)) => {...},
  }
}
```

Le librerie aiutano:

- **Thiserror** → si riesce a genarare in automatico il **tratto errorr** tramite la macro derive
  - Etc
- **Anyhow** → permette di gestire in modo semplice la **gestione degli errori**
  - Informa che la funzione genera un anyhow error
  - Mi da la possibilità di aggiungere dei contesti

- Il crate **anyhow** definisce l'oggetto-tratto **anyhow::Error** che semplifica la gestione idiomatica degli errori
  - Si può usare il tipo **anyhow::Result<T>** per incapsulare il valore di ritorno di una funzione che può fallire
- Questo tipo offre un'implementazione automatica del tratto **From<T: Error>**, il che permette di utilizzare la notazione basata sull'operatore **?** per propagare l'errore ritornato
  - Quando viene generato un errore è possibile aggiungere una descrizione che contestualizza ciò che è successo tramite i metodi **context(...)** e **with_context(...)**
  - Il messaggio di errore associato verrà sostituito dalla stringa indicata seguita dalla causa originale

```
fn sum_file(path: &Path) -> anyhow::Result<i32> {
  let mut file = File::open(path).with_context(|| format!("Missing path {}", path)) ? ;

  let mut contents = String::new();
  file.read_to_string(&mut contents).context("File read error") ? ;
  let mut sum = 0;
  for line in contents.lines() {
    sum += line.parse::<i32>().with_context(|| format!("Not a number: {}", line)) ? ;
  }
  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::<io::Error>() {...}  //tratto io::Error
      else if let Some(e) = err.downcast_ref::<ParseIntError>() {...}  //tratto ParseIntError
      else { unreachable!(); }  //non può capitare
```

Codice lez24/04 → ok