Gestione degli errori

Computazioni invalide

Errori ed eccezioni

- Tutte le computazioni possono fallire prematuramente
  - Impedendo che possa essere restituito il valore atteso o che vengano eseguiti tutti gli effetti collaterali richiesti
- Questi fallimenti possono essere dovuti a cause differenti
  - Alcune facilmente prevedibili (argomenti illeciti, conversione da testo a numero, …)
  - Altre dovute a limiti del sistema di elaborazione (memoria/spazio su disco esauriti, malfunzionamento della rete o di altre periferiche, …)
- Indipendentemente dalle cause che li hanno prodotti, i fallimenti possono essere catalogati in due gruppi principali
  - Malfunzionamenti **recuperabili** - quelli che non hanno compromesso lo stato del programma, per cui è possibile attivare una strategia di ripristino
  - Malfunzionamenti **non recuperabili** - quelli che causano un'alterazione impredicibile dello stato o che indicano l'impossibilità di procedere ulteriormente nella computazione

Errori ed eccezioni

- Nei casi irrecuperabili, occorre terminare il processo
  - Eventualmente eseguendo qualche operazione di pulizia sull'ambiente esterno
- Negli altri, occorre mettere in atto una strategia di ripristino dello stato
  - Ritentare l'operazione, richiedere l'intervento dell'utente/amministratore, utilizzare una strategia alternativa, …
- Non è detto che il punto in cui si verifica il fallimento possegga abbastanza contesto per discriminare 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
- Questo requisito tende ad aumentare la complessità del codice, introducendo rami diversi (if/else, switch/case, match…) lungo i quali la computazione può procedere, favorendo l'introduzione di errori logici legati alla quantità di dettagli cui occorre badare
  - Per questo motivo, i linguaggi "moderni" includono particolari costrutti a supporto della gestione delle eccezioni

Supporto sintattico alla gestione degli errori

- Il linguaggio C non fornisce alcun supporto sintattico alla modellazione degli errori
  - Lasciando completamente al programmatore la responsabilità di gestire la situazione
- Il linguaggio C++ (come molti altri linguaggi, come Java, C#, JavaScript, …) introduce il concetto di eccezione
  - Offre le parole-chiave **`try/catch/throw`** per esprimere la logica di notifica e gestione
  - Impone una particolare struttura del contesto di esecuzione per supportare tale genere di astrazione
- Rust non utilizza il concetto di eccezione, ma offre i tipi algebrici **`Result<T,E>`** e **`Option<T>`** per esprimere gli esiti delle computazioni
  - Offre inoltre la macro **`panic!(...)`** per forzare l'interruzione del thread corrente producendo una descrizione testuale di quanto successo

Eccezioni in C++

- In C++, qualsiasi computazione può arrestarsi invocando l'istruzione **`throw`** seguita da un valore di qualsiasi tipo che descrive il tipo di malfunzionamento verificatosi
  - L'esecuzione di tale istruzione comporta il ritorno forzato della funzione corrente al suo chiamante e, eventualmente, al chiamante del chiamante, …, arretrando progressivamente nella storia della computazione, fino a raggiungere un'invocazione racchiusa all'interno di un blocco **`try { … }`**
  - Oppure fino alla totale contrazione dello stack
- Un blocco try ha la seguente sintassi

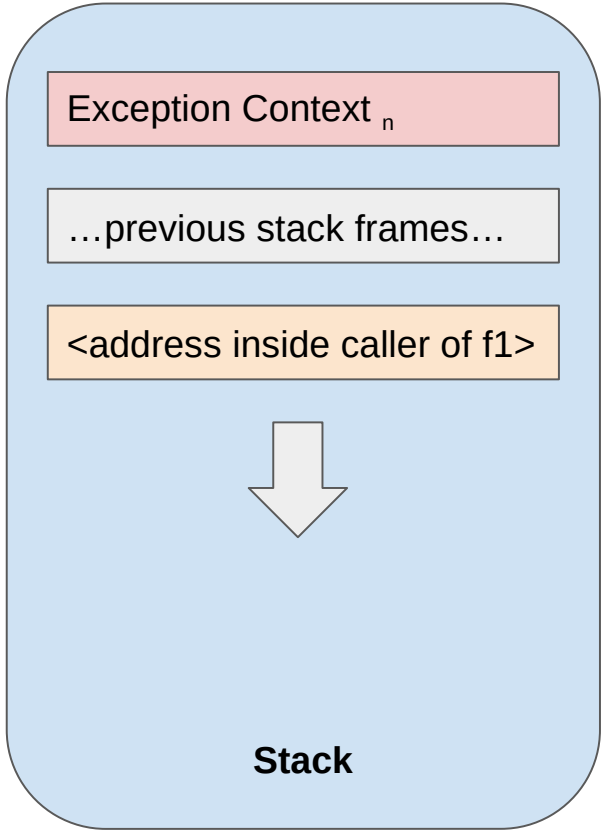
```
try {
  //codice che può fallire direttamente o
indirettamente
} catch (ExceptionType1 e1) {
   //…istruzioni di ricupero
} catch (ExceptionType2 e2) { … }
```

Eccezioni in C++

- Se nella storia delle chiamate è stato localizzato un blocco try, il tipo del valore che è stato lanciato viene confrontato, in sequenza, con i tipi indicati nei successivi blocchi **`catch`**
  - In caso di corrispondenza, viene eseguito il blocco di codice che segue il costrutto catch appropriato allo scopo di ricuperare lo stato del sistema, dopodiché la computazione riprende dall'istruzione successiva all'ultimo blocco catch
  - Se nessun blocco è adatto a gestire il tipo di errore verificatosi, il processo si ripete tornando ad un blocco try più esterno, se esistente
- Altrimenti, il processo si arresta e viene indicato un codice di errore diverso da 0

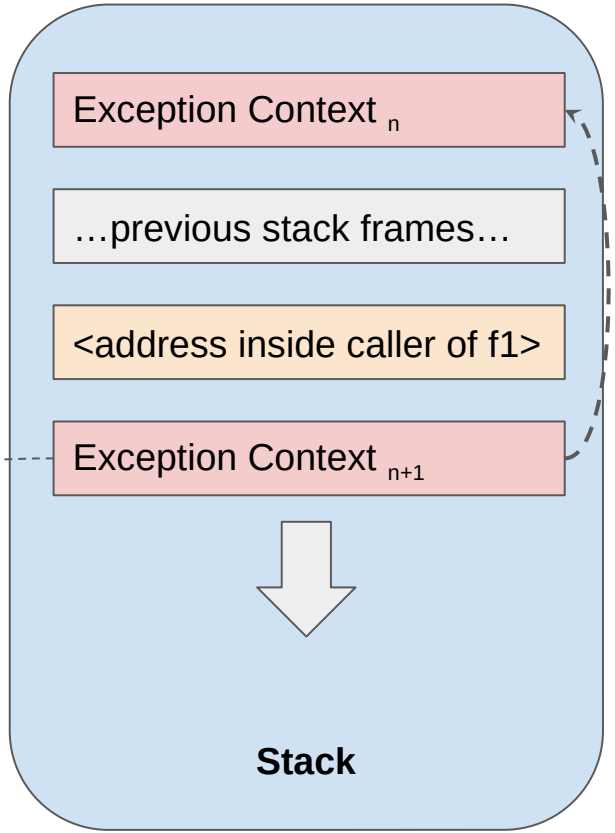
Eccezioni in C++

```
int f2() {
  int i = 1;
  if (some_condition)
    throw std::logic_error("err");
  return i;
}

int f1() {
  try {
    return f2();
  }
  catch (std::logic_error e) {
    //restore state
    return -1;
  }
}
```



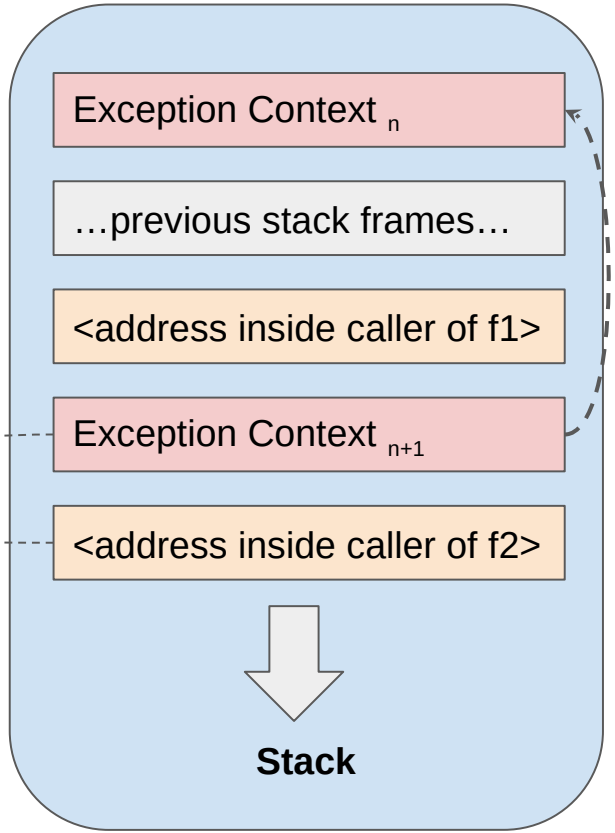
Eccezioni in C++

```
int f2() {
  int i = 1;
  if (some_condition)
    throw std::logic_error("err");
  return i;
}

int f1() {
  try {
    return f2();
  }
  catch (std::logic_error e) {
    //restore state
    return -1;
  }
}
```



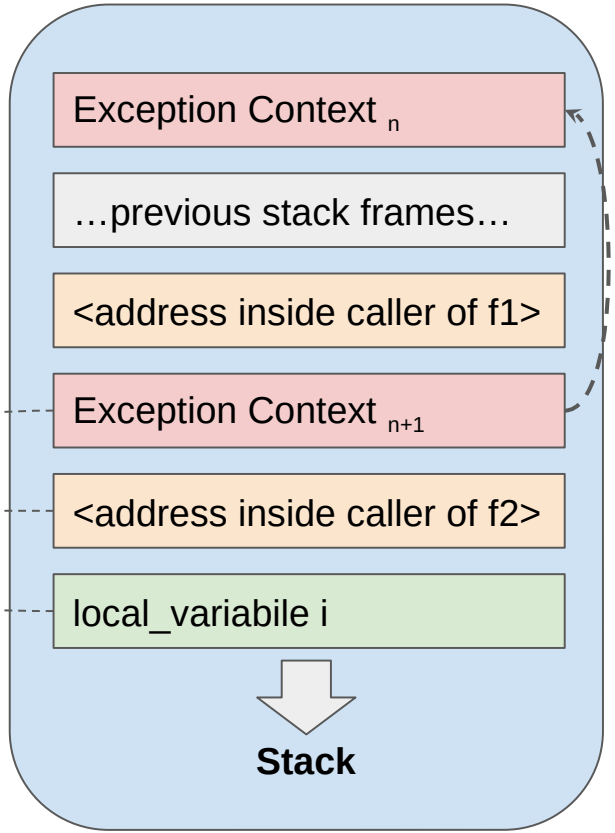
Eccezioni in C++

```
int f2() {
  int i = 1;
  if (some_condition)
    throw std::logic_error("err");
  return i;
}

int f1() {
  try {
    return f2();
  }
  catch (std::logic_error e) {
    //restore state
    return -1;
  }
}
```



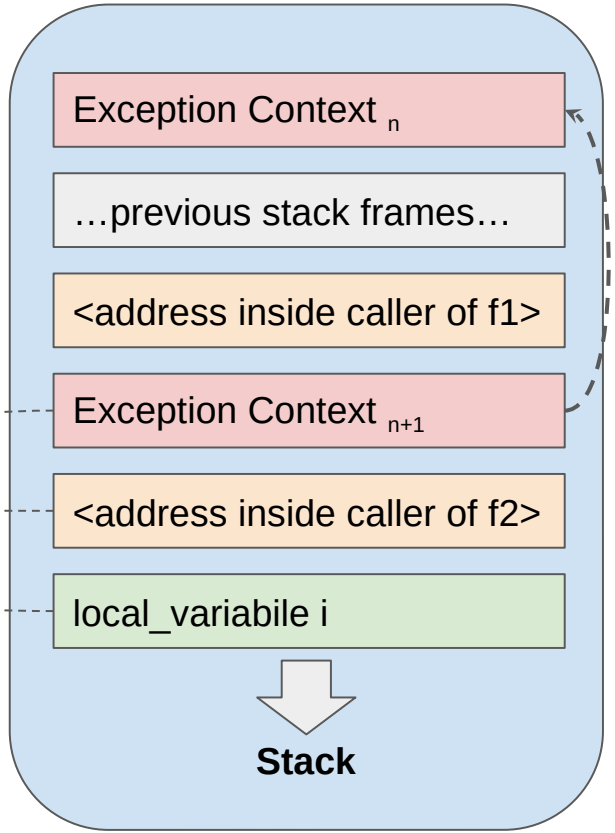
Eccezioni in C++

```
int f2() {
  int i = 1;
  if (some_condition)
    throw std::logic_error("err");
  return i;
}

int f1() {
  try {
    return f2();
  }
  catch (std::logic_error e) {
    //restore state
    return -1;
  }
}
```



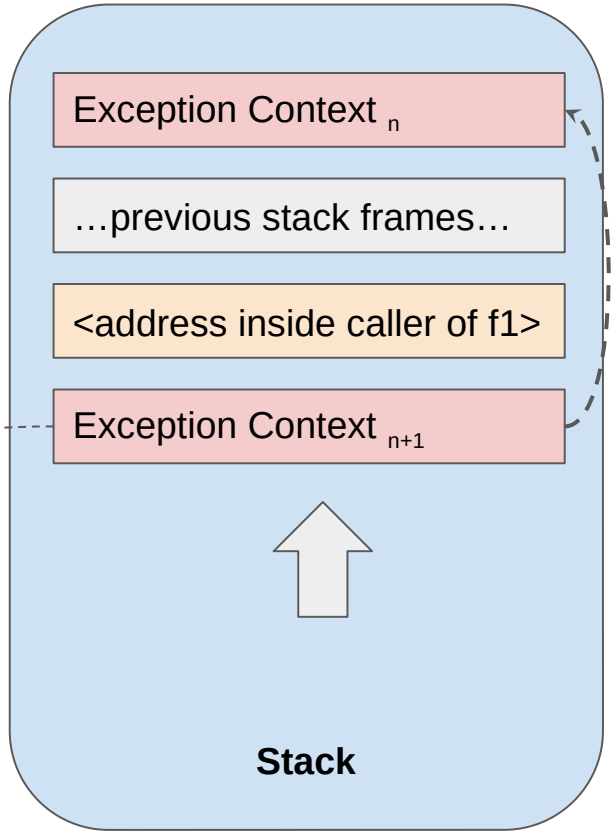
Eccezioni in C++

```
int f2() {
  int i = 1;
  if (some_condition)
    throw std::logic_error("err");
  return i;
}

int f1() {
  try {
    return f2();
  }
  catch (std::logic_error e) {
    //restore state
    return -1;
  }
}
```



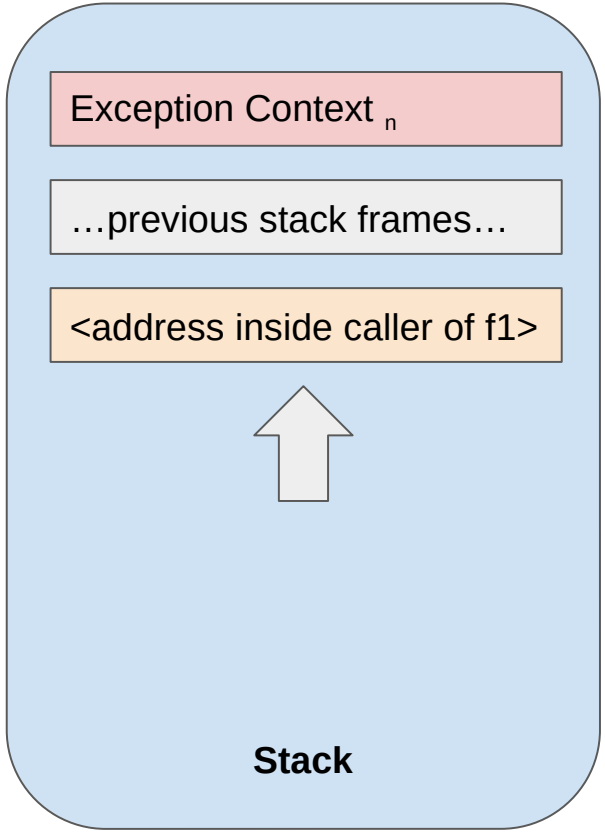
Eccezioni in C++

```
int f2() {
  int i = 1;
  if (some_condition)
    throw std::logic_error("err");
  return i;
}

int f1() {
  try {
    return f2();
  }
  catch (std::logic_error e) {
    //restore state
    return -1;
  }
}
```



Eccezioni in C++

```
int f2() {
  int i = 1;
  if (some_condition)
    throw std::logic_error("err");
  return i;
}

int f1() {
  try {
    return f2();
  }
  catch (std::logic_error e) {
    //restore state
    return -1;
  }
}
```



Eccezioni in C++

- Sebbene il C++ non ponga né vincoli sintattici né semantici sul tipo di dato utilizzato per descrivere un'eccezione, è uso comune usare classi derivate da **`std::exception`**
  - Volte ad introdurre una serie di tipi distinti, così da abilitare il supporto che il compilatore è in grado di fornire al programmatore
  - E a contenere, nelle proprie variabili istanza, maggiori dettagli sul malfunzionamento verificatosi
- Poiché il lancio di un'eccezione comporta l'immediata interruzione della funzione corrente (causando un ritorno anticipato, senza peraltro inizializzare il valore di ritorno), è anche comune gestire una parte del problema legato al comportamento inatteso con il pattern RAII
  - **Resource Acquisition Is Initialization**
  - Il ritorno comporta infatti la contrazione dello stack, con il conseguente rilascio di tutte la variabili locali e l'esecuzione dei relativi distruttori
  - Questi ultimi possono essere usati per liberare risorse acquisite o disfare effetti collaterali avviati dal costruttore, contando sul fatto che verranno eseguiti sempre e comunque, anche in caso di eccezioni
- Rust riprende questa idea, appoggiandosi su strutture che implementano il tratto **`Drop`**

I limiti della gestione delle eccezioni in C++

- Il compilatore non è in grado di identificare dove vengano restituite eccezioni
  - Conseguentemente, non forza l'utilizzo di costrutti in grado di gestirle
- La generazione di un'eccezione blocca l'esecuzione del codice seguente e riporta tipicamente un singolo tipo di errore
  - Rendendo, ad esempio, onerosa la validazione contemporanea di più criteri
- La possibilità che un'eccezione dello stesso tipo possa essere generata in parti diverse della computazione, ma gestita in un unico punto a monte, rende complessa la scelta delle contromisure da applicare
  - Richiedendo l'ispezione (manuale) della catena delle chiamate fallite per identificare il punto di rottura
- Per permettere la corretta contrazione dello stack e il ritorno al blocco `try {...}` più recente occorre imporre una certa sovrastruttura allo stack ed al contesto di esecuzione
  - Tale sovrastruttura non si adatta particolarmente alle assunzioni che vengono fatte nel kernel di Linux e questo è uno dei principali motivi per cui non è possibile scrivere moduli kernel in C++ per quel sistema operativo

Gestioni delle eccezioni in Rust

- Rust offre una risposta funzionale al problema della modellazione degli errori basata sul tipo algebrico generico **`Result<T,E>`**
  - Modella l'unione di tutti i possibili risultati con successo **`T`** e tutti gli errori **`E`** che possono verificarsi nell'esecuzione di una funzione

```
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),
  }
}
```

Elaborare i risultati

- Result<T,E> mette a disposizione svariati metodi che consentono di accedere ai dati contenuti al suo interno
  - 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

Gestire gli errori

- Se l'invocazione di una funzione restituisce un valore di tipo Result contenente un errore, occorre mettere in atto una strategia di gestione
  - Terminare, in modo ordinato, il programma
  - Ritentare l'esecuzione, evitando di entrare in un loop infinito
  - Registrare un messaggio nel log e propagare l'errore al chiamante
  - Propagare *tout court* l'errore al chiamante, delegando ad esso la corrispondente responsabilità
  - Altro…
- Sebbene terminare il programma possa apparire facile (è sufficiente invocare la funzione **`std::process::exit(code: i32) -> !`**), farlo in modo pulito può essere complesso
  - Occorre infatti garantire che non vengano lasciati oggetti persistenti (come file o altre risorse di sistema) in stati non coerenti
  - Per questo motivo, Rust offre la macro **`panic!(...)`**

Panic

- In alcune situazioni, lo stato del programma risulta così compromesso che non è pensabile eseguire azioni di ripristino: questo spesso è dovuto alla presenza di errori logici nel programma stesso
  - Accesso ad array al di fuori dei relativi limiti
  - Divisioni di interi per zero
  - Fallimenti di asserzioni
- In queste situazioni, Rust offre la macro **`panic!(...)`** che accetta argomenti simili a quelli offerti dalla macro **`println!(...)`** per formulare un messaggio di errore
  - L'effetto dell'invocazione di questa macro è la contrazione dello stack (come nel caso delle eccezioni C++), la distruzione (via **`.drop()`**) delle variabili che implementano il tratto **`Drop`** fino alla terminazione del thread corrente
  - Se il thread in cui è stato invocato **`panic!(...)`** è il thread principale dell'applicazione, il processo termina con un codice di errore non nullo, altrimenti **continua**

Ignorare gli errori

- In alcune situazioni, può capitare che l'errore che potenzialmente viene restituito da una funzione non possa di fatto succedere, a conseguenza di quanto si è appena verificato nel corso dell'esecuzione
  - Oppure che il programmatore assuma che non sia necessario mettere in atto una strategia di contenimento e gestione dell'errore, lasciando semplicemente terminare il processo
- Il tipo **`Result<T,E>`** mette a disposizione i metodi **`unwrap()`** e **`expect(...)`** che restituiscono il valore di tipo **`T`**, se presente
  - Ed invocano la macro **`panic!(...)`** in caso di errore
  - **`expect(...)`** permette di indicare una stringa che sarà usata al posto del messaggio standard generato da **`unwrap()`**, nel caso si sia verificato un errore

Propagare gli errori

- Nella maggior parte delle funzioni in cui si verifica un errore, non si sa quale strategia mettere in atto per ripristinare lo stato del programma
  - Si può facilmente ovviare a questo problema restituendo, a propria volta, un oggetto di tipo **`Result<T,E>`**
  - Questo porta, tuttavia, a costrutti alquanto complessi, con espressioni di controllo difficili da decodificare
- Per semplificare la sintassi, Rust offre l'operatore **`?`** che può essere applicato a qualunque espressione che produce un valore di tipo **`Result<T,E>`**
  - Se il valore risulta essere **`Ok(v)`**, l'operatore restituisce il valore **`v`** racchiuso nell'enum
  - Se, invece, risulta essere **`Err(e)`**, la funzione corrente termina, ritornando il valore che incapsula l'errore
- Perché questo comportamento possa funzionare, occorre che la funzione in cui è usato l'operatore **`?`** ritorni un oggetto di tipo **`Result<U,E>`**
  - E che Il tipo dell'errore ritornato sia compatibile con l'errore **`E`** che proviene dalla funzione chiamata

Propagare gli errori

- L'utilizzo dell'operatore **?** permette di ottenere una sintassi molto più compatta, evidenziando il comportamento della funzione lungo il cammino principale
  - E demandando al compilatore la scrittura delle clausole if / match necessarie a valutare il comportamento da adottare

```
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)
}
```

Altri modi di esprimere il fallimento

- In alcune situazioni, può essere sufficiente distinguere se la computazione ha prodotto il proprio risultato, oppure indicare che non è stato possibile completarla
  - Per questi casi esiste il tipo **`Option<T>`** che racchiude due alternative:
  - **`Some<T>`**, usata per descrivere il risultato atteso
  - **`None`**, che indica l'assenza di risultato, senza descriverne le ragioni
- L'operatore **`?`** può essere applicato anche al tipo **`Option<T>`**
  - A condizione che la funzione in cui viene adottato abbia tipo di ritorno **`Option<U>`**, con **`U`** qualsiasi
- **`Result<T,E>`** e **`Option<U>`** sono correlati
  - Result offre il metodo **ok(self)** che restituisce Option<T>, valorizzato con il dato in caso di successo (il dato risulta mosso da Result, che quindi diventa inaccessibile)
  - Offre anche il metodo **err(self)** che restituisce Option<E>, valorizzato con l'errore in caso di fallimento (anche in questo caso, con movimento)

Propagare errori eterogenei

- Quando una funzione produce diverse tipologie di errore, è necessario propagare i diversi errori così da poter specializzare le contromisure
  - Rust offre diversi modi per propagare errori eterogenei, la scelta spetta al programmatore sulla base delle sue esigenze
- La libreria standard mette a disposizione l'implementazione generica del tratto **`From`**, che converte qualsiasi valore che implementi il tratto **`Error`** in **`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 **`?`**

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

Propagare 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) ? ; // 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
    }
  }
}
```

Propagare errori eterogenei

- Per propagare errori eterogenei senza forzare il sistema dei tipi è possibile implementare degli errori custom
  - Tutti gli errori custom devono implementare il tratto **`Error`** e conseguentemente anche i tratti **`Debug`** e **`Display`**
  - L'utilizzo di un **`enum`** permette di racchiudere i diversi tipi di errore da gestire successivamente tramite l'utilizzo del costrutto **`match`**
  - E' necessario implementare il tratto **`From`** per convertire i diversi errori nel tipo custom da propagare

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

Propagare errori eterogenei

```
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,
    })
  }
}
```

Propagare errori eterogenei

```
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)) => {...},
  }
}
```

Propagare errori eterogenei

- Un grosso aiuto all'implementazione di tipi che implementano il tratto Error viene dal crate **`thiserror`**
  - Esso offre una implementazione della macro **`derive`** specializzata per il tratto **`Error`**
- Definendo un tipo (enum, struct, unit) preceduto dall'attributo **`#[derive(Error, Debug)]`** si ottiene l'implementazione automatica di questi due tratti
  - L'attributo **`#[error("Messaggio con formato")]`** posta di fronte alle singole voci enumerative o all'intera struct genera l'implementazione del tratto **`Display`**
  - L'attributo **`#[from]`** posto di fronte ad un campo il cui tipo implementa il tratto **`Error`** genera l'implementazione del tratto **`From`** a partire dal tipo del campo
- Questo approccio è particolarmente adatto nella creazione di librerie, dove si vogliono rendere i possibili errori parte della definizione del contratto (API)

Propagare errori eterogenei

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

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

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

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

Propagare errori eterogenei

- 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
- Questo crate interopera correttamente con **`thiserror`** e risulta adatto nella scrittura di codice applicativo, piuttosto che di librerie
  - In questo caso, la semplicità di scrittura del codice domina rispetto al controllo dell'informazione contenuta nell'errore generato, tenuto conto che sarà interpretato da persone

Propagare errori eterogenei

```
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
    }
  }
}
```