Gestione delle eccezioni

Anna Corazza

aa 2023/24

Dove studiare

- Str'13, cap. 13
- Str'14, sezioni 5.6, 19.4, 19.5

Str'13 Bjarne Stroustrup, The C++ Programming Language (4th edition), 2013 `https://www.stroustrup.com/4th.html`

Str'14 Bjarne Stroustrup, Programming: Principles and Practice using C++ (2nd edition), 2014 `https://www.stroustrup.com/programming.html`

Gestione degli errori

- Ci concentreremo sugli errori che non possono venir gestiti localmente.
- Richiedono di separare la loro gestione in diverse parti del programma.
- Parti diverse che spesso sono state sviluppate separatamente.
- Le indicheremo col termine di **librerie**: al momento della loro implementazione, lo sviluppatore può non conoscere nemmeno il tipo di programmi in cui verranno utilizzate.
  - Un errore può venir individuato all'interno di una libreria, ma lì non si sa come trattarlo.
  - L'utente della libreria sa cosa fare dell'errore, ma non come individuarlo (altrimenti lo avrebbe già fatto fuori della libreria.
- Ovviamente questo discorso è valito di programmi grossi e strutturati la cui esecuzione si prolunga nel tempo.

Eccezioni

- Lo scopo delle **eccezioni** è quello di portare l'informazione dal punto in cui un errore è stato individuato a dove può essere gestito.
- Una funzione che non è in grado di gestire un problema **throws** un'eccezione, sperando che qualcuno la raccolga (**catch**)
- Un componente che chiama una funzionalità indica quali eccezioni vuole gestire per mezzo di un meccanismo **try-catch**.
- Il componente chiamato, se non riesce a completare il task assegnatogli, lancia un'eccezione mediante un'espressine **throw**

Meccanismo `try-catch`

```
void taskmaster(){
  try{
    auto result=do_task();
    // usa i risultati
  }
  catch(Some error){
    // gestisci il problema
  }
}

int do_task(){
  // ...
  if(/* in grado di eseguire il task */)
     return result;
  else
     throw Some_error();
  }
```

Eccezioni

- Un'eccezione è un oggetto che viene "lanciato" come rappresentazione dell'occorrenza di un errore.
- Può essere di qualsiasi tipo che ammetta la copia, ma è raccomandabile usare un tipo definito apposta, in modo da minimizzare il clash tra errori lanciati da librerie diverse.
- La libreria `std` definisce una piccola gerarchia di eccezioni.
- Un'eccezione può trasportare informazioni sull'errore che rappresenta.

Gerarchia degli errori della libreria standard

- logic_error — possono essere individuati o prima che cominci l'esecuzione o attraverso dei test sugli argomenti di funzioni e costruttori.
  - length_error
  - domain_error
  - out_of_range
  - invalid_argument
  - future_error
- runtime_error — tutti gli altri
  - range_error
  - overflow_error
  - underflow_error
  - system_error
- bad_exception
- bad_alloc
- bad_typeid
- bad_cast

Gestione degli errori tradizionali

Senza le eccezioni

Terminare il programma

Restituire un valore di errore non sempre possibile (nessun valore disponibile)

Stato di errore Restituire un valore legale, ma lasciare il programma in uno stato di errore (occorre controllare esplicitamente)

Funzione che gestisce l'errore (si rimanda a come la funzione gestisce l'errore)

RAII – Resource Acquisition is Initialization

- Una tecnica del tutto generale basata sulle proprietà dei costruttori e dei distruttori e sulla loro interazione con la gestione degli errori.
- In generale, l'idea è di allocare le risorse nel costruttore e di rilasciarle nel distruttore.
- Si noti che le risorse possono essere anche diverse dalla memoria: thread, file aperti, socket aperti, spazio disco, connessioni alla base dati.
- In generale, tutto ciò di cui abbiamo a disposizione una quantità finita.
- Ogni risorsa viene incapsulata in una classe, in cui
  - il costruttore acquisisce la risorsa, e lancia un'eccezione se non ci riesce
  - il distruttore rilascia la risorsa senza mai lanciare eccezioni

RAII – esempio

```
std::mutex m;

void bad()
{
    m.lock();                    // acquire the mutex
    f();                         // if f() throws an exception, the mutex is never released
    if (!everything_ok())
        return;                  // early return, the mutex is never released
    m.unlock();                  // if bad() reaches this statement, the mutex is released
}

void good()
{
  std::lock_guard<std::mutex> lk(m); // RAII class: mutex acquisition is
                                     //   initialization
    f();                             // if f() throws an exception, the mutex
                                     //   is released
    if (!everything_ok())
        return;                      // early return, the mutex is released
}                                    // if good() returns normally, the mutex
                                     //   is released
```

Esempio: commenti

- **mutex** procedimento di sincronizzazione fra processi o thread concorrenti con cui si impedisce che più task paralleli accedano contemporaneamente ai dati in memoria o ad altre risorse soggette a corsa critica.
- Il RAII è seguito dalle classi della libreria standard che gestiscono le proprie risorse: `std::string`, `std::vector`, e molte altre.
- RAII non può essere utilizzato per risorse che non vengono acquisite prima dell'utilizzo: si pensi al tempo CPU.