Exception Safety

Una porzione di codice si dice exception safe quando si comporta in maniera "adeguata" anche in presenza di comportamenti anomali segnalati tramite il lancio di eccezioni.

In particolare, occorre valutare se la porzione di codice, in seguito al comportamento eccezionale, non abbia compromesso lo stato del programma in maniera irreparabile: esempi di compromissione sono il mancato rilascio (cioè la perdita) di risorse oppure la corruzione dello stato interno di una risorsa (ad esempio, l'invariante di classe non è più verificata), con la conseguenza che qualunque ulteriore tentativo di interagire con la risorsa si risolve in un comportamento non definito (*undefined behavior*).

Livelli di exception safety

Esistono tre diversi livelli di exception safety:

- base
- forte
- nothrow

Livello base

Una porzione di codice (una funzione o una classe) si dice exception safe a livello base se, anche nel caso in cui si verifichino delle eccezioni durante la sua esecuzione:

1. Non si hanno perdite di risorse (resource leak).
2. Si è neutrali rispetto alle eccezioni quando, ogni qual volta viene ricevuta un'eccezione questa viene catturata momentaneamente, gestita in modo "locale", e successivamente viene rilasciata al chiamante (permettendo così la sua propagazione così che possa prenderne atto ed eseguire a sua volta eventuali azioni correttive necessarie).
3. Anche in caso di uscita in modalità eccezionale, gli oggetti sui quali si stava lavorando sono distruggibili senza causare comportamenti non definiti. Quindi lo stato interno di un oggetto, anche se parzialmente inconsistente, deve comunque consentirne la corretta distruzione (o riassegnamento).

Questo è il livello minimo che deve essere garantito per poter parlare di exception safety.

> Gli altri livelli, che forniscono garanzie maggiori, sono spesso considerati opzionali (perché più costosi da ottenere).

Torna all'indice

Livello forte

Il livello forte (strong) di exception safety si ottiene quando, oltre a tutte le garanzie fornite dal livello base, si aggiunge come ulteriore garanzia una sorta di atomicità delle operazioni (tutto o niente). Intuitivamente, l'invocazione di una funzione exception safe forte, in caso di eccezione, garantisce che lo stato degli oggetti manipolati è rimasto inalterato, identico allo stato precedente la chiamata.

> es: *Rollback* dei DBMS.

Esempio

Supponiamo di avere una classe che implementa una collezione ordinata di oggetti e di avere un metodo `insert` che inserisce un nuovo oggetto nella collezione esistente.

Se il metodo in questione garantisce l'exception safety forte, allora in seguito ad una eccezione durante una operazione di insert la collezione si troverà esattamente nello stesso stato precedente all'operazione di `insert` (cioè, conterrà

esattamente gli stessi elementi che conteneva prima della chiamata alla insert).

Se invece fosse garantita solo l'exception safety a livello base, non avendo la proprietà di atomicità, in caso di uscita con eccezione la collezione si troverebbe in uno stato consistente, ma potenzialmente in nessun rapporto con lo stato precedente alla chiamata (per esempio, potrebbe essere vuota o contenere elementi diversi rispetto a quelli contenuti precedentemente).

Torna all'indice

Livello nothrow

È il livello massimo: una funzione è nothrow se la sua esecuzione è garantita, non terminare in modalità eccezionale.

Questo livello lo si raggiunge in un numero limitato di casi:

- Quando l'operazione è così semplice che non c'è alcuna possibilità di generare eccezioni (esempio, assegnamento di tipi built-in).
- Quando la funzione è in grado di gestire completamente al suo interno eventuali eccezioni, risolvendo eventuali problemi e portando comunque a termine con successo l'operazione richiesta.
- Quando la funzione, di fronte a eventuali eccezioni interne, nell'impossibilità di attuare azioni correttive, determina la terminazione di tutto il programma. Questo è il caso delle funzioni che sono dichiarate (implicitamente o esplicitamente) `noexcept`, come i *distruttori*: in caso di eccezione non catturata, viene automaticamente invocata la terminazione del programma.

  > Intuitivamente, devono garantire il livello nothow i distruttori e le funzioni che implementano il rilascio delle risorse (non è ipotizzabile ottenere l'exception safety se l'operazione di rilascio delle risorse può non avere successo).

Si noti che il livello nothrow, per definizione, *NON* è neutrale rispetto alle eccezioni.

Torna all'indice

Libreria standard e exception safety

I contenitori (vector, deque, list, set, map, ...) forniti dalla libreria standard sono exception safe.

Tale affermazione vale sotto determinate condizioni. Dato che si parla di contenitori templatici, quindi possono essere istanziati a partire da un qualunque tipo di dato `T`, le garanzie di exception safety del contenitore sono valide a condizione che il tipo di dato `T` degli elementi contenuti fornisca analoghe garanzie.

> Molte operazioni su questi contenitori forniscono la garanzia forte (strong exception safety). Alcune però forniscono solo una garanzia base, tipicamente quando si opera su molti elementi contemporaneamente, perché quella strong sarebbe troppo costosa.

Esempio strong safety

Se viene invocato il metodo `void push_back(const T& t)` su di un oggetto di tipo `std::vector<T>` e il tentativo di copiare l'oggetto `t` all'interno del *vector* dovesse fallire lanciando una eccezione (per esempio, perché il costruttore di copia di `T` ha esaurito le risorse a disposizione e non può effettuare la copia), si può essere sicuri che il vector *NON* è stato modificato. Se prima della chiamata conteneva gli n elementi `[t1, ..., tn]`, in uscita dalla chiamata contiene ancora gli stessi elementi (nello stesso ordine).

Esempio base safety

Il metodo `void assign(size_type n, const T& val)` sostituisce il contenuto del vector con n copie del valore `val`, siccome un'eccezione potrebbe essere lanciata da una qualunque delle n operazioni di costruzione, il *vector*, in caso di eccezione, rimane in uno stato valido, ma il suo contenuto non è predicibile (in particolare, molto probabilmente il contenuto precedente è irrecuperabile).

Torna all'indice

Approcci alternativi per gestire le risorse

Un esempio su tre approcci possibili che l'utente può adottare per ottenere un uso corretto di una risorsa anche in presenza di segnalazioni di errore:

Errori "tradizionali" (no eccezioni)

risorsa_no_exc.hh

```
#ifndef GUARDIA_risorsa_no_exc_hh
#define GUARDIA_risorsa_no_exc_hh 1

// Tipo dichiarato ma non definito (per puntatori "opachi")
struct Risorsa;

// Restituisce un puntatore nullo se l'acquisizione fallisce.
Risorsa* acquisisci_risorsa();

// Restituisce true se si è verificato un problema.
bool usa_risorsa(Risorsa* r);

// Restituisce true se si è verificato un problema.
bool usa_risorse(Risorsa* r1, Risorsa* r2);

void restituisci_risorsa(Risorsa* r);

#endif // GUARDIA_risorsa_no_exc_hh
```

user_no_exc.cc

```
#include "risorsa_no_exc.hh"

bool codice_utente() {
 Risorsa* r1 = acquisisci_risorsa();

 if (r1 == nullptr) {
  // errore durante acquisizione di r1: non devo rilasciare nulla
  return true;
 }

 // acquisita r1: devo ricordarmi di rilasciarla

 if (usa_risorsa(r1)) {
  // errore durante l'uso: rilascio r1
  restituisci_risorsa(r1);
  return true;
 }

 Risorsa* r2 = acquisisci_risorsa();

 if (r2 == nullptr) {
  // errore durante acquisizione di r2: rilascio di r1
  restituisci_risorsa(r1);
  return true;
 }

 // acquisita r2: devo ricordarmi di rilasciare r2 e r1
```

```
 if (usa_risorse(r1, r2)) {
  // errore durante l'uso: rilascio r2 e r1
  restituisci_risorsa(r2);
  restituisci_risorsa(r1);
  return true;
 }

 // fine uso di r2: la rilascio
 restituisci_risorsa(r2);
 // ho ancora r1: devo ricordarmi di rilasciarla

 Risorsa* r3 = acquisisci_risorsa();

 if (r3 == nullptr) {
  // errore durante acquisizione di r3: rilascio di r1
  restituisci_risorsa(r1);
  return true;
 }

 // acquisita r3: devo ricordarmi di rilasciare r3 e r1

 if (usa_risorse(r1, r3)) {
  // errore durante l'uso: rilascio r3 e r1
  restituisci_risorsa(r3);
  restituisci_risorsa(r1);
  return true;
 }

 // fine uso di r3 e r1: le rilascio
 restituisci_risorsa(r3);
 restituisci_risorsa(r1);

 // Tutto ok: lo segnalo ritornando false
 return false;
}
```

Torna all'indice

Uso di blocchi try/catch

risorsa_exc.hh

```
#ifndef GUARDIA_risorsa_exc_hh
#define GUARDIA_risorsa_exc_hh 1

#include "risorsa_no_exc.hh"

struct exception_acq_risorsa {};
struct exception_uso_risorsa {};

// Lancia una eccezione se non riesce ad acquisire la risorsa.
inline Risorsa*
acquisisci_risorsa_exc() {
 Risorsa* r = acquisisci_risorsa();
 if (r == nullptr)
  throw exception_acq_risorsa();
 return r;
}

// Lancia una eccezione se si è verificato un problema.
inline void
usa_risorsa_exc(Risorsa* r) {
 if (usa_risorsa(r))
  throw exception_uso_risorsa();
}

// Lancia una eccezione se si è verificato un problema.
inline void
usa_risorse_exc(Risorsa* r1, Risorsa* r2) {
 if (usa_risorse(r1, r2))
  throw exception_uso_risorsa();
}

#endif // GUARDIA_risorsa_exc_hh
```

user_try_catch.cc

```
#include "risorsa_exc.hh"

void codice_utente() {
 Risorsa* r1 = acquisisci_risorsa_exc();

 try { // blocco try che protegge la risorsa r1
  usa_risorsa_exc(r1);

  Risorsa* r2 = acquisisci_risorsa_exc();

  try { // blocco try che protegge la risorsa r2
   usa_risorse_exc(r1, r2);
   restituisci_risorsa(r2);
  } // fine try che protegge r2
  catch (...) {
   restituisci_risorsa(r2);
   throw;
  }

  Risorsa* r3 = acquisisci_risorsa_exc();
  try { // blocco try che protegge la risorsa r3
   usa_risorse_exc(r1, r3);
   restituisci_risorsa(r3);
  } // fine try che protegge r3
  catch (...) {
   restituisci_risorsa(r3);
   throw;
  }
  restituisci_risorsa(r1);

 } // fine try che protegge r1
 catch (...) {
  restituisci_risorsa(r1);
  throw;
 }
}
```

Osservazioni:

1. Si crea un blocco try/catch per ogni singola risorsa acquisita.
2. Il blocco si apre subito *dopo* l'acquisizione della risorsa (se l'acquisizione fallisce, non c'è nulla da rilasciare).
3. La responsabilità del blocco try/catch è di proteggere *quella* singola risorsa (ignorando le altre).
4. Al termine del blocco try (prima del catch) va effettuata la "normale" restituzione della risorsa (caso NON eccezionale).
5. La clausola catch usa $\cdots$ per catturare qualunque eccezione: non ci interessa sapere che errore si è verificato (non è nostro compito), dobbiamo solo rilasciare la risorsa protetta.
6. Nella clausola catch, dobbiamo fare due operazioni:
    - rilasciare la risorsa protetta;
    - rilanciare l'eccezione catturata (senza modificarla) usando l'istruzione `throw;`.

Il rilancio dell'eccezione catturata garantisce la "neutralità rispetto alle eccezioni": i blocchi catch catturano le eccezioni solo temporaneamente, lasciandole poi proseguire. In questo modo anche gli altri blocchi catch potranno fare i loro rilasci di risorse e l'utente otterrà comunque l'eccezione, con le informazioni annesse, potendo quindi decidere come "gestirla".

Torna all'indice

Uso dell'idioma RAII-RRID

- RAII: Resource Acquisition Is Initialization
- RRID: Resource Release Is Destruction

risorsa_raii.hh

```
#ifndef GUARDIA_risorsa_raii_hh
#define GUARDIA_risorsa_raii_hh 1
```

```
#include "risorsa_exc.hh"

// classe RAII-RRID (spesso detta solo RAII, per brevità)
// RAII: Resource Acquisition Is Initialization
// RRID: Resource Release Is Destruction

class Gestore_Risorsa {
private:
 Risorsa* res_ptr;
public:
 // Costruttore: acquisisce la risorsa (RAII)
 Gestore_Risorsa() : res_ptr(acquisisci_risorsa_exc()) { }

 // Distruttore: rilascia la risorsa (RRID)
 ~Gestore_Risorsa() {
  // Nota: si assume che restituisci_risorsa si comporti correttamente
  // quando l'argimento è il puntatore nullo; se questo non è il caso,
  // è sufficiente aggiungere un test prima dell'invocazione.
  restituisci_risorsa(res_ptr);
 }

 // Disabilitazione delle copie
 Gestore_Risorsa(const Gestore_Risorsa&) = delete;
 Gestore_Risorsa& operator=(const Gestore_Risorsa&) = delete;

 // Costruzione per spostamento (C++11)
 Gestore_Risorsa(Gestore_Risorsa&& y)
  : res_ptr(y.res_ptr) {
  y.res_ptr = nullptr;
 }

 // Assegnamento per spostamento (C++11)
 Gestore_Risorsa& operator=(Gestore_Risorsa&& y) {
  restituisci_risorsa(res_ptr);
  res_ptr = y.res_ptr;
  y.res_ptr = nullptr;
  return *this;
 }

 // Accessori per l'uso (const e non-const)
 const Risorsa* get() const { return res_ptr; }
 Risorsa* get() { return res_ptr; }

 // Alternativa agli accessori: operatori di conversione implicita
 // operator Risorsa*() { return res_ptr; }
 // operator const Risorsa*() const { return res_ptr; }

}; // class Gestore_Risorsa

#endif // GUARDIA_risorsa_raii_hh
```

user_raii.cc

```
#include "risorsa_raii.hh"

void codice_utente() {
 Gestore_Risorsa r1;
 usa_risorsa_exc(r1.get());
{
    Gestore_Risorsa r2;
    usa_risorse_exc(r1.get(), r2.get());
}     // L'inserimento di questo blocco serve per fare in modo che
   // lo scope di r2 finisca proprio in questo punto,
   // prima di inizializzare r3
 Gestore_Risorsa r3;
 usa_risorse_exc(r1.get(), r3.get());
}
```

Torna all'indice

Esercizio

!

Soluzione:

```
void job(){
 Res* r1 = new Res("res1");
 try {
  Res* r2 = new Res("res2");
  try {
   do_task(r1, r2);
   delete res1;
   delete res2;
  } catch (...) {
   delete res2;
   throw;
  }
 } catch (...) {
  delete res1;
  throw;
 }
}
```

Torna all'indice

Altro materiale

- Articolo di Stroustroup Exception Safety: Concepts and Techniques http://www.stroustrup.com/except.pdf
- Video e lucidi della presentazione di Jon Kalb al CppCon 2014 (con bonus per i fan di Star Wars) Video parte 1: https://www.youtube.com/watch?v=W7fIy_54y-w Video parte 2: https://www.youtube.com/watch?v=b9xMIKb1jMk Video parte 3: https://www.youtube.com/watch?v=MiKxfdkMJW8 Ludici: http://exceptionsafecode.com/slides/esc.pdf

Torna all'indice