16 - Gestione delle risorse

La gestione delle risorse garantisce che le risorse limitate siano utilizzate correttamente e rilasciate al termine del loro utilizzo.

Definizione di "risorse"

Con "risorse" indichiamo genericamente entità che sono disponibili in quantità limitata, se tali risorse vengono esaurite o mal gestite, possono compromettere il funzionamento del software.
Esempi:

- **memoria dinamica (heap)**: acquisita con `new` e rilasciata con `delete`;
- **file**: descrittori aperti tramite operazioni come `fopen` o `std::ifstream` e chiusi tramite `fclose` o la terminazione dello stream;
- **lock**: utilizzati per sincronizzare l'accesso a risorse condivise da più thread o processi;
- **connessioni di rete**: ad esempio connessioni verso server o database (DBMS);
- altri esempi includono **socket**, **handler di dispositivi** e perfino **risorse di sistema** come semafori o thread.

Tre fasi della gestione

L'interazione con una risorsa segue uno schema predefinito composto da tre fasi ordinate temporalmente:

1. **acquisizione**: la risorsa viene richiesta (es. allocazione di memoria o apertura di un file);
2. uso: la risorsa viene utilizzata per svolgere un compito specifico (es. scrittura su file o accesso a memoria);
3. rilascio: la risorsa viene restituita o liberata (es. chiusura del file o deallocazione della memoria).

Regole fondamentali

- acquisizione prima dell'uso: non si può utilizzare una risorsa prima di averla acquisita;
- rilascio dopo l'uso: una risorsa deve essere rilasciata al termine del suo utilizzo;
- divieto di uso dopo il rilascio: una risorsa non deve mai essere usata dopo essere stata rilasciata (es. accesso a puntatore "dangling");
- divieto di rilascio multiplo: non si deve rilasciare una risorsa più volte (es. doppia chiamata `delete`).

Esempi cattiva gestione delle risorse

Memory leak

Si verifica quando una risorsa (es. memoria dinamica) non viene rilasciata.

```
void memory_leak_ex() {
    int* ptr = new int(42);
    //nessuna delete -> memory leak
}
```

Dangling pointer

Si verifica quando si accede a memoria già rilasciata.

```
void dangling_pointer_ex() {
    int* ptr = new int(42);
    delete ptr;
    *ptr = 10; //errore: accesso a memoria rilasciata
}
```

Double free

Si verifica quando si tenta di rilasciare una risorsa già rilasciata.

```
void double_free_ex() {
    int* ptr = new int(42);
    delete ptr;
    delete ptr; //errore: rilascio multiplo
}
```

Exception safety

Un codice si dice *exception safe* quando si comporta in maniera appropriata anche in caso di eccezioni. Questo significa che, qualora si verifichino errori gestiti tramite eccezioni, il programma:

1. non perde risorse;
2. mantiene uno stato consistente (o, nel peggiore dei casi, lascia lo stato degli oggetti manipolati distruggibile senza causare comportamenti non definiti);
3. propaga correttamente l'eccezione, permettendo al chiamante di gestirla.
   Es.

```
void foo() {
    int* pi = new int(42); //acquisizione
    do_the_job(pi); //uso
    delete pi; //rilascio
}
```

questo codice non è exception safe perché se la funzione `do_the_job` lancia un'eccezione, l'istruzione `delete pi` non verrà mai eseguita causando un memory leak.

Tre livelli di exception safety

1 - Livello base

Una funzione è exception safe a livello base se:

- non ci sono *perdita di risorse* (esempio: la memoria allocata dinamicamente viene sempre rilasciata anche in caso di eccezioni);
- lo stato del programma rimane *consistente* e l'oggetto su cui si lavora può essere distrutto senza causare problemi;

- le eccezioni generate vengono propagate al chiamante.

```
void example() {
    std::unique_ptr<int> ptr(new int(42)); // Risorsa gestita
    // Operazione che potrebbe lanciare un'eccezione
    risky_operation(*ptr);
    // Non serve esplicitare il rilascio: la memoria sarà gestita automaticamente
}
```

anche se `risky_operation` lancia un'eccezione, la memoria allocata tramite `std::unique_ptr` viene rilasciata automaticamente grazie alla gestione RAII (Resource Acquisition Is Inizialization) e lo stato del programma rimane consistente.

2 - Livello forte (strong exception safety)

Una funzione garantisce exception safet forte se assicura la proprietà di atomicità: o l'operazione ha successo e lo stato cambia, oppure in caso di eccezione lo stato rimane invariato.

```
void safe_insert(std::vector<int>& vec, int value) {
    std::vector<int> copy(vec); // Crea una copia del contenitore originale
    copy.push_back(value);      // Modifica la copia
    std::sort(copy.begin(), copy.end()); // Ordina la copia
    vec = std::move(copy);      // Sostituisce il contenuto originale solo se tutto
va a buon fine
}
```

Se qualcosa va storto il contenitore originale rimane intatto; lo stato di `vec` cambia solo se tutte le operazioni intermedie sono state completate con successo.
Questa garanzia è più costosa da implementare rispetto al livello base, ma è spesso desiderabile per operazioni critiche.

3 - Livello nothrow

Il livello massimo di sicurezza si raggiunge quando una funzione è dichiarata **nothrow** (ovvero garantisce che non lancerà mai eccezioni). Questo livello è fondamentale per alcune operazioni, come i distruttori e le funzioni che rilasciano risorse.

```
class SafeResource {
public:
    ~SafeResource() noexcept {
        // Garanzia che il distruttore non lanci eccezioni
        release_resource();
    }
};
```

Quando si utilizza `nothrow`:

- operazioni che non possono fallire per definizione (es. assegnamenti tra tipi built-in come `int`);
- funzioni che gestiscono completamente le eccezioni al loro interno, senza propagare errori all'esterno;

- funzioni di rilascio di risorse o distruttori: non ha senso rischiare di fallire mentre si tenta di recuperare uno stato consistente.
  N.B. le funzioni dichiarate `noexcept` non possono propagare eccezioni. Se una funzione `noexcept` lancia un'eccezione, il programma termina con un errore irreversibile.

Comportamento della libreria standard

La libreria standard garantisce che i suoi contenitori siano exception safe, ma le garanzie variano a seconda delle operazioni eseguite:

- **garanzia forte**: alcune operazioni forniscono la proprietà di atomicità, ad esempio se la `push_back()` fallisce, il contenuto del vettore rimane invariato;
- **garanzia base**: operazioni più complesse (come `assign` o `resize`) possono modificare lo stato del contenitore in caso di eccezione, ma lo stato rimane valido e consistente.
- **garanzia nothrow**: operazioni come la distruzione degli elementi o il rilascio della memoria non generano mai eccezioni.

Tecniche per raggiungere exception safety:

Vi sono tre approcci che possono essere combinati tra loro:

- evitare le eccezioni;
- uso dei blocchi `try` / `catch`;
- uso dell'idioma RAII-RRID.

Esempi:

user.cc

Codice utente che vorrebbe lavorare su alcune risorse garantendo la corretta interazione con le risorse (acquisizione, uso e rilascio) anche in presenza di errori. Intuitivamente, si vorrebbe eseguire questa sequenza di operazioni:
acquisisci risorsa r1
usa risorsa r1

acquisisci risorsa r2
usa risorse r1 e r2
restituisci risorsa r2

acquisisci risorsa r3
usa risorse r1 e r3
restituisci risorsa r3
restituisci risorsa r1

```
/* Una codifica che NON è corretta in presenza di errori */

#include "risorsa_no_exc.hh"

void codice_utente() {
  Risorsa* r1 = acquisisci_risorsa();
  usa_risorsa(r1);
```

```
  Risorsa* r2 = acquisisci_risorsa();
  usa_risorse(r1, r2);
  restituisci_risorsa(r2);
  Risorsa* r3 = acquisisci_risorsa();
  usa_risorse(r1, r3);
  restituisci_risorsa(r3);
  restituisci_risorsa(r1);
}
```

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) { //controllo se sono riuscito ad acquisire la risorsa
    // 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;
}
```

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'argomento 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());
  }
  Gestore_Risorsa r3;
```

```
  usa_risorse_exc(r1.get(), r3.get());
}
```

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.cpp

```
#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 "..." 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 (seconda parte del punto 6)
  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".
*/
```