4. Gestione Risorse e Exception Safety

Una problematica riguardante lo sviluppo del software é quella della gestione delle risorse.
Bisogna infatti mirare ad un'ottima gestione delle risorse per ottenere dei vantaggi in termini di "peso" del software (memoria occupata, costi computazionali, ecc...).

In generale, si puó ottenere una gestione corretta quando il programma segue un flusso di esecuzione previsto dallo sviluppatore.

In questo capitolo vedremo dunque come ottimizare la gestione ma soprattutto come comportarci nel caso in cui si verifichino **eccezioni**, ovvero quando il software NON segue il flusso previsto.

Prerequisiti

Eccezioni in C++

In C++ esiste un tipo di errori detti **errori di runtime** che non possono essere rilevati in fase di compilazione poiché si verificano solo durante la fase di esecuzioni in particolari circostanze.
Sono errori pericolosi, se non opportunamente gestiti possono anche causare il crash dell'applicazione.
Per questo C++ mette a disposizione alcuni meccanismi per gestire queste eccezioni.

In questo linguaggio é previsto l'uso del costrutto **try-catch** come meccanismo.
Nel blocco `try` si inseriscono le istruzioni cosiddette "a rischio". Invece `catch` introduce il blocco che contiene le istruzioni per un determinato tipo di errore.
Ad esempio.

```
try {

     // Istruzioni a rischio di errori di runtime.
}
catch(tipoErrore1) {

     // Istruzioni che gestiscono l'errore 1.
}
catch(tipoErrore2) {

     // Istruzioni che gestiscono l'errore 2.
```

```
}
catch(...)  {

     // Istruzioni che gestiscono tutte le altre eccezioni.
}
```

Gestione Risorse

L'interazione del software con le risorse deve seguire uno schema predefinito composto da 3 passi:

1. Acquisizione della risorsa.
2. Utilizzo.
3. Rilascio.

In particolare, al termine del suo utilizzo la risorsa deve essere rilasciata. Se questo non si verifica si parla di **memory leak**, che in alcuni casi potrebbe portare all'esaurimento della memoria.

Inoltre non é lecito utilizzare una risorsa dopo averla rilasciata (bisogna tornare alla fase di acquisizione), quando si tenterá di fare ció si verificherá il cosiddeto **dangling pointer**.

Vediamo un esempio.

```
void foo() {
     int* pi = new int(42); // Acquisizione.
     do_the_job(pi); // Utilizzo (tramite funzione ausiliaria).
     delete pi; // Rilascio.
}
```

Exception Safety

Come detto all'inizio, la gestione corretta delle risorse é semplice da ottenere quando il software segue il flusso di esecuzione previsto.
Diventa dunque importante imparare a gestire tutte quelle situazioni in cui il software esce da questo flusso.

In C++ vengono lanciate delle eccezioni che vengono catturate da un blocco `catch` che penserá a gestire poi la situazione specifica.
Ovviamente é essenziale imparare a capire nel dettaglio cosa succede, in modo da poter sapere come e quali eccezioni gestire; questo é sostanzialmente il compito della exception safety.

Ad esempio, il codice scritto nel paragrafo precedente, NON é exception safety. Infatti nel caso in cui la funzione `do_the_job` (o qualunque funzione invocata da questa) dovesse lanciare un'eccezione (e questa non venisse gestita internamente), il flusso di esecuzione NON seguire piú il percorso previsto e di conseguenza NON andrebbe ad eseguire l'istruzione `delete pi`.
Quindi si uscirebbe da `foo` senza aver rilasciato la risorsa ottenendo il fenomeno di memory leak.

La domanda da porsi ora é: quali tecniche si possono utilizzare affinché il codice diventi exception safe? E quali, tra le varie tecniche, sono migliori e quindi piú raccomandabili?

Gestione Risorse con Try-Catch

É possibile gestire le tre fasi delle risorse utilizzando il costrutto C++ `try-catch`. Bisogna seguire alcune regole:

1. Si crea un blocco `try-catch` per ogni risorsa acquisita (non per ogni suo utilizzo).
2. Il blocco si apre subito dopo l'acquisizione.
   Infatti se l'acquisizione dovesse fallire non avremmo nulla da rilasciare.
3. La responsabilitá `try-catch` é di proteggere quella singola risorsa per cui é stato progettato, senza preoccuparsi delle altre.
4. Al termine del blocco `try` (appena prima del `catch`) bisogna effettuare la normale restituzione della risorsa.
   Questo rappresenta infatti il caso NON eccezionale.
5. La clausola `catch` deve usare la forma `...` ovvero:

   ```
   catch(...)  { // Codice del blocco catch.  }
   ```

   Questo perché non interessa nello specifico che errore si é verificato, bisogna solamente rilasciare la risorsa.

6. Nella clausola `catch` bisogna effettuare due operazioni: la prima é, appunto, il rilascio della risorsa. La seconda é rilanciare l'eccezione catturata usando l'istruzione `throw`.

Vediamo l'esempio di codice dal punto di vista dell'utente.

```
void codice_utente() {

      Risorsa* r1 = acquisisci_risorsa();

      // Blocco che protegge la risorsa 1.
      try {

            uso_risorsa(r1);
            Risorsa* r2 = acquisisci_risorsa();

            // Blocco che protegge la risorsa 2.
            try {

                  uso_risorse(r1,r2);
                  restituisci_risorsa(r2);
            }
            catch(...)  {

                  restituisci_risorsa(r2);
                  throw;
            }

            restituisci_risorsa(r1);
      }
      catch(...)  {

            restituisci_risorsa(r1);
            throw;
      }
}
```

Idioma RAII

Un'altra tecnica per la gestione delle risorse é rappresentata dall'idioma RAII anche detto **RAII-RRID**.

L'acronimo sta per Resource Acquisition Is Initializaton - Resource Release Is Destruction.

Infatti segue alla lettera il suo significato RAII; quando l'oggetto é inizializzato automaticamente acquisice la risorsa.
L'acquisizione viene dunque inserita nel costruttore nel seguente modo:

```
Gestore_Risorsa() :  r1(acquisici_risorsa()) {}
```

Quando viene invocato il costruttore la risorsa 1 viene inizializzata con il risultato della funzione `acquisisci_risorsa`, se l'invocazione della funzione ha successo viene restituito un puntatore che viene salvato in `r1`, se invece l'invocazione non va a buon fine non viene acquisita nessuna risorsa e si uscirá dal costruttore in modalitá eccezzionale.

Analogamente la seconda parte dell'acronimo RRID ci suggerisce che il rilascio delle risorse viene effettuato in fase di distruzione in questo modo:

```
~Gestore_Risorsa() { restituisci_risorsa(r1); }
```

Questo idioma viene dunque usato per legare le risorse alla durata (vita) dell'oggetto.

Livelli di Exception Safety

Abbiamo detto che una porzione di codice si dice **exception safe** quando si comporta in maniera adeguata anche in presenza di comportamenti anomali segnalati tramite le eccezioni senzaa peró compromettere lo stato del programma.
Esistono tre livelli di exception safety:

- **Base**
  Se nel momento in cui si verificano eccezioni si mantengono queste proprietá:

  1. Non si ha perdita di risorse.
  2. Il codice rimane neutrale rispetto alle eccezioni. Ovvero cattura l'eccezione e la rilancia senza modificarla, in questo modo anche altre parti di codice "a rischio" possono verificare la medesima eccezione cosí come é stata pensata.

3. Anche in caso di uscita in modalitá eccezionale gli oggetti (risorse) sui quali si stava lavorando rimangono quanto meno distruggibili, senza causare quindi comportamenti indefiniti.

- **Forte**
  Detta anche **strong**, garantisce tutte le proprietá del livello base aggiungendo una sorta di atomicitá delle operazioni.
  Quindi se nella parte di codice a rischio una delle operazioni non va a buon fine tutti gli oggetti (risorse) utilizzati all'interno del blocco venogno riportati allo stato precedente la chiamata.

- **Nothrow**
  Quando l'esecuzione del codice garantisce di NON terminare in modalitá eccezionale.

  Si verifica in casi specifici come operazioni molto semplici che non hanno la possibilitá di generare eccezioni, oppurre se la funzione é in grado di gestire completamente al suo interno le eccezioni.

  Esistono anche un tipo di funzioni dette **noexcept** che, nell'impossibilitá di risolvere l'eccezione e proseguire in modo sicuro, determinano la terminazione di tutto il programma (come i distruttori).

Libreria Standard e Exception Safety

I contenitori forniti dalla libreria standard (vector, list, deque, ecc...) garantiscono exception safety.
Ovviamente, dato che si parla di contenitori templatici (istanziati a partire da un qualunque tipo di dato `T`), é necessario che `T` a sua volta fornisca garanzie analoghe di esception safety.

Di base le operazioni su questi contenitori garantiscono un livello strong, tranne nei casi in cui esse operino su molti elementi contemporaneamente.
In questo caso infatti risulterebbe troppo costoso il livello strong e si usa quindi il livello base.

Smart Pointer

Questo tipo di puntatori forniscono un supporto diretto per la gestione delle risorse.
Abbiamo visto come l'idioma RAII si prestasse bene come aiuto in questo compito, ma diventa impegnativo (e comunque soggetto a possibili errori) scrivere una classe RAII per ogni tipo `T` ogni volta che si deve usare un `T*`.
Per questo la libreria standard ci fornisce classi templatiche che implementano diverse tipologie di **smart pointer** che sono:

- **Unique** - `std::unique_ptr`
  Puntatore smart ad un oggetto di tipo `T`.
  In particolare implementa il concetto di **owning**, ovvero considera solo l'unico proprietario della risorsa dandogli la responsabilitá di gestione di questa.
  Non sono copiabili (violerebbe il requisito di unicitá) ma sono invece spostabili (spostano la proprietá della risorsa al nuovo gestore).
  Esempio.

```
void foo(std::unique_ptr<int> pj);

void bar() {

      std::unique_ptr<int> pi(new int(42));
      foo(pi); // Errore di compilazione:  copia non ammessa.
      foo(std::move(pi)); // Spostamento ammesso.
      // Da qui in poi pi non gestisce piú nessuna risorsa.
}
```

- **Shared** - `std::shared_ptr`
  Puntatore smart ad un oggetto di tipo `T`.
  Ogni volta che il puntatore viene copiato l'originale e la copia condividono la responsabilitá della (stessa) risorsa.
  A livello di implementazione, la copia causa l'incremento al contatore di riferimenti alla risorsa (reference counter), quando il puntatore viene distrutto decrementa il contatore e, qualora si accorgesse di essere l'ultimo, ne effettua il rilascio.

  Esempio.

```
void foo() {

      std::shared_pointer<int> pi; {

            // RC = Reference Counter.
```

```
            std::shared_ptr<int> pj(new int(42)); // RC=1.
            pi = pj; // Condivisione risorsa, RC=2.
            ++(*pi); // Uso risorsa condivisa, valore = 43.
            ++(*pj); // Uso risorsa condivisa, valore = 44.
      } // Distruzione di pj, RC=1.

      ++(*pi); // Uso risorsa, valore = 45.
} // Distruzione di pi, RC=0 e rilascio risorsa.
```

- **Weak** - `std::weak_ptr`
  Un problema degli shared pointer é la possibilitá di ritrovarsi strutture cicliche di risorse che si puntano l'uno sull'altra.
  In questo caso le risorse comprese nel ciclo mantengono dei reference counter positivi anche se non sono piú raggiungibili, causando memory leak.
  Per risolvere questo problema si usano i **weak pointer** che sono puntatori ad una risorsa condivisa che peró non partecipano attivamente alla gestione della risorsa.

  Sostanzialmente un weak pointer non accede direttamente alla risorsa, prima controlla che sia ancora disponibile invocando il metodo `lock()` che da come risultato uno shared pointer. Se la risorsa non é piú disponibile lo shared pointer ottenuto sará nullo.

  Esempio.

```
void maybe_print(std::weak_ptr<int> wp) {

      if (auto sp2 == wp.lock())
            std::cout << *sp2;
      else
            std::cout << "Non piú disponibile";
}

void foo() {

      std::weak_ptr<int> wp; {
            auto sp = std::make_shared<int>(42);
            wp = sp; // wp non incrementa il RC della risorsa.
            *sp = 55;
            maybe_print(wp); // Stampa 55.
      } // sp viene distrutto insieme alla risorsa.

      maybe_print(wp); // Stampa "Non piú disponibile".
}
```