Smart pointers

Come accennato quando si è introdotto il discorso della gestione delle risorse e dell'exception safety, uno dei casi più frequenti che si verificano è quello della corretta gestione dell'allocazione dinamica della memoria.

L'uso dei semplici puntatori forniti dal linguaggio (detti anche puntatori "raw" o "naked" o addirittura "dumb", in contrapposizione a quelli "smart", ovvero intelligenti) si presta infatti a tutta una serie di possibili errori di programmazione nei quali può incappare anche un programmatore esperto (se cala il livello di attenzione).

L'idioma RAII-RRID si presta bene a neutralizzare la maggior parte di questi errori, rendendoli molto meno probabili. D'altra parte, scrivere una classe RAII per ogni tipo `T` ogni volta che si vuole usare un `T*` è operazione noiosa, ripetitiva e comunque soggetta a errori.

La libreria standard viene però in aiuto, fornendo delle classi templatiche che forniscono diverse tipologie di puntatori "smart": `unique_ptr`, `shared_ptr` e `weak_ptr`. Le tre classi templatiche sono definite nell'header file `<memory>`.

> NOTA BENE: i puntatori smart forniti dalla libreria standard sono concepiti per memorizzare puntatori a memoria allocata dinamicamente sotto il controllo del programmatore; non si possono utilizzare per la memoria ad allocazione statica o per la memoria ad allocazione automatica (sullo stack di sistema).

Torna all'indice

---

unique_ptr

Uno `std::unique_ptr<T>` è un puntatore smart ad un oggetto di tipo `T`. In particolare, `unique_ptr` implementa il concetto di puntatore "*owning*", ovvero un puntatore che si considera l'unico proprietario della risorsa.

Intuitivamente, allo smart pointer spetta l'onere di fornire una corretta gestione della risorsa (nello specifico, rilasciarla a lavoro finito).

Esempio:

```
#include

void foo() {
 std::unique_ptr pi(new int(42));
 std::unique_ptr pd(new double(3.1415));
 *pd *= *pd; // si dereferenzia come un puntatore
 // altri usi ...
} // qui termina il tempo di vita di pi e pd e viene rilasciata la memoria
```

Una caratteristica degli unique_ptr è il fatto di NON essere copiabili, ma di essere (solo) spostabili. La copia è impedita in quanto violerebbe il requisito di unicità del gestore della risorsa; lo spostamento è invece consentito, in quanto si trasferisce la proprietà della risorsa al nuovo gestore.

Esempio:

```
void foo(std::unique_ptr pi);

void bar() {
  std::unique_ptr pj(new int(42));
  // foo(pj);         // errore di compilazione: copia non ammessa
  foo(std::move(pj)); // ok: spostamento ammesso
  // dopo lo spostamento, pj non gestisce nessuna risorsa
}
```

La classe fornisce poi metodi per potere interagire con i puntatori "raw", da usarsi nel caso in cui ci si debba interfacciare con codice che, per esempio, era stato sviluppato prima dell'adozione dello standard C++11.

Esempio:

```
std::unique_ptr pi;    // pi non gestisce (ancora) una risorsa
int* raw_pi = new int(42);

pi.reset(raw_pi);           // NON devo invocare la delete su raw_pi

int* raw_pj = pi.get();     // NON devo invocare la delete su raw_pj
int* raw_pk = pi.release(); // devo invocare la delete su raw_pk
```

Con il metodo `reset()` il puntatore prende in gestione una nuova risorsa (diventandone il proprietario), rilasciando la risorsa che aveva in gestione precedentemente, se presente.

Il metodo `get()` fornisce il puntatore raw alla risorsa gestita, che però rimane sotto la responsabilità dello `unique_ptr`; il metodo `release()`, invece, restituisce il puntatore raw e ne cede anche la responsabilità di corretta gestione.

Torna all'indice

---

shared_ptr

Uno `std::shared_ptr<T>` è un puntatore smart ad un oggetto di tipo `T`. Lo `shared_ptr` implementa il concetto di puntatore per il quale la responsabilità della corretta gestione della risorsa è "condivisa": intuitivamente, ogni volta che uno `shared_ptr` viene *copiato*, l'originale e la copia condividono la responsabilità della gestione della (stessa) risorsa.

A livello di implementazione, la copia causa l'incrementato di un contatore del numero di riferimenti alla risorsa (`reference counter`).

Quando uno `shared_ptr` viene distrutto, decrementa il `reference counter` associato alla risorsa e, se si accorge di essere rimasto l'unico `shared_ptr` ad avervi ancora accesso, ne effettua il rilascio (informalmente, si dice che "*l'ultimo chiude la porta*").

Esempio:

```
#include

void foo() {
 std::shared_ptr pi;

 {
  std::shared_ptr pj(new int(42)); // ref counter = 1
  pi = pj;  // condivisione risorsa, ref counter = 2
  ++(*pi);  // uso risorsa condivisa: nuovo valore 43
  ++(*pj);  // uso risorsa condivisa: nuovo valore 44
 } // distruzione pj, ref counter = 1

 ++(*pi);    // uso risorsa condivisa: nuovo valore 45

} // distruzione pj, ref counter = 0, rilascio risorsa
```

Come detto, gli `shared_ptr` sono *copiabili* (e spostabili). La classe fornisce i metodi `reset()` e `get()`, con la semantica intuitiva.

Esempio:

```
void foo(std::shared_ptr pi);

void bar() {
 std::shared_ptr pj(new int(42));

 foo(pj);            // ok: copia ammessa, risorsa condivisa
 std::cout << *pj;   // stampa la risorsa come modificata da foo

 foo(std::move(pj)); // ok: spostamento ammesso
 // dopo lo spostamento, pj non gestisce nessuna risorsa
}
```

Torna all'indice

Template di funzione (make_shared e make_unique)

Un puntatore shared deve interagire con due componenti: la risorsa e il "blocco di controllo" della risorsa (una porzione di memoria nella quale viene salvato anche il reference counter). Per motivi di efficienza, sarebbe bene che queste due componenti fossero allocate con una singola operazione: questa è la garanzia offerta dalla `std::make_shared`.

Esempio:

```
void bar() {
 auto pi = std::make_shared(42);
 auto pj = std::make_shared(3.1415);
}
```

Oltre all'efficienza, l'uso di `std::make_shared` consente di evitare alcuni errori subdoli che potrebbero compromettere la corretta gestione delle risorse in presenza di comportamenti eccezionali.

Esempio:

```
void bar(std::shared_ptr pi,
         std::shared_ptr pj);

void foo() {
  // codice NON exception safe
  bar(std::shared_ptr(new int(42)),
      std::shared_ptr(new int(42)));

  // codice exception safe
  bar(std::make_shared(42),
      std::make_shared(42));
}
```

Siccome l'ordine di esecuzione delle sottoespressioni è non specificato, nella prima chiamata della funzione `bar()` una implementazione potrebbe decidere di valutare per prime le due espressioni `new` passate come argomenti ai costruttori degli `shared_ptr` e solo dopo invocare i costruttori.

Se la prima allocazione tramite `new` andasse a buon fine ma la seconda invece fallisse con una eccezione, si otterrebbe un memory leak (per la prima risorsa allocata), in quanto il distruttore dello `shared_ptr` NON verrebbe invocato (perché l'oggetto non è stato costruito).

Il problema non si presenta nella seconda chiamata a `bar()`, perché le allocazioni sono effettuate (implicitamente) dalla `make_shared`.

> NOTA: questo esempio NON dovrebbe causare un problema di exception safety nel caso di una implementazione conforme allo standard C++17: in questo standard, infatti, è stata modificata la regola relativa all'ordine di valutazione degli argomenti in una chiamata di funzione.

A partire dallo standard C++14 è stata resa disponibile anche la `std::make_unique`. L'uso degli smart pointer e di queste funzioni per la loro creazione dovrebbe consentire al programmatore di limitare al massimo la necessità di utilizzare (esplicitamente) le espressioni `new` e le corrispondenti invocazioni di `delete`: in effetti, nelle più recenti linee guida alla programmazione in C++, l'uso diretto (naked) di `new` e `delete` è considerato "cattivo stile", quasi quanto l'uso dell'istruzione `goto`.

http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

Torna all'indice

---

weak_ptr

Un problema che si potrebbe presentare quando si usano gli `shared_ptr` (più in generale, quando si usa qualunque meccanismo di condivisione di risorse basato sui reference counter) è dato dalla possibilità di creare insiemi di risorse che, puntandosi reciprocamente tramite `shared_ptr`, formano una o più strutture cicliche.

In questo caso, le risorse comprese in un ciclo mantengono dei reference count positivi anche se non sono più raggiungibili a partire dagli shared_ptr ancora accessibili da parte del programma, causando dei memory leak. L'uso dei `std::weak_ptr` è pensato per risolvere questi problemi.

Un `weak_ptr` è un puntatore ad una risorsa condivisa che però non partecipa attivamente alla gestione della risorsa stessa: la risorsa viene quindi rilasciata quando si distrugge l'ultimo `shared_ptr`, anche se esistono dei `weak_ptr` che la indirizzano. Ciò significa che un `weak_ptr` non può accedere direttamente alla risorsa: prima di farlo, deve controllare se la risorsa è ancora disponibile. Il modo migliore per farlo è mediante l'invocazione del metodo `lock()`, che produce uno `shared_ptr` a partire dal `weak_ptr`: se la risorsa non è più disponibile, lo `shared_ptr` ottenuto conterrà il puntatore nullo.

Esempio:

```
void maybe_print(std::weak_ptr wp) {
 if (auto sp2 = wp.lock())
  std::cout << *sp2;
 else
  std::cout << "non più disponibile";
}

void foo() {
 std::weak_ptr wp;
 {
  auto sp = std::make_shared(42);
  wp = sp; // wp non incrementa il reference count della risorsa
  *sp = 55;
  maybe_print(wp); // stampa 55
 } // sp viene distrutto, insieme alla risorsa

 maybe_print(wp); // stampa "non più disponibile"
}
```

Torna all'indice