Cast

Conversioni esplicite di tipo in C++

Il C++ fornisce varie sintassi per effettuare il cast (conversione esplicita di tipo) di una espressione, allo scopo di ottenere un valore di un tipo (potenzialmente) diverso:

1. `static_cast`
2. `dynamic_cast`
3. `const_cast`
4. `reinterpret_cast`
5. cast "funzionale"
6. cast stile C

Prima di considerare nel dettaglio le varie sintassi dei cast, vale la pena ragionare sui motivi (validi) per il loro uso.

Torna all'indice

Classificazione delle motivazioni per l'uso di cast espliciti

Essendo conversioni esplicite di tipo, i cast dovrebbero essere utilizzati solo quando necessario. E' possibile classificare gli usi dei cast in base alla motivazione, che frequentemente ricade in una di queste categorie:

1. Il cast implementa una conversione di tipo che NON è consentita dalle regole del linguaggio come conversione implicita, in quanto considerata una frequente fonte di errori di programmazione. Il programmatore, richiedendo esplicitamente la conversione con il cast, si assume la responsabilità della sua correttezza. Esempio:

```
struct B { /* ... */ };
struct D : public B { /* ... */ };
D d;
B* b_ptr = &d;
// b_ptr è (staticamente, cioè a tempo di compilazione) un puntatore a B,
// ma (dinamicamente, cioè a tempo di esecuzione) sta puntando ad un
// oggetto di tipo D.
/* ... altro codice ... */
// Il programmatore forza il down-cast, prendendosi la responsabilità
// di eventuali errori: se qualcuno nel frattempo avesse modificato b_ptr
// e questo non puntasse più ad un oggetto di tipo D, si ottiene un
// Undefined Behavior.
D* d_ptr = static_cast(b_ptr);
```

2. Come nel caso precedente, ma il programmatore usa (in modo appropriato) un `dynamic_cast` allo scopo di controllare, a tempo di esecuzione, se la conversione richiesta è effettivamente consentita. Esempio:

```
struct B { /* ... */ };
struct D : public B { /* ... */ };

void foo(B* b_ptr) {
 if (D* d_ptr = dynamic_cast(b_ptr)) {
  // posso usare d_ptr, che punta ad un oggetto di tipo D
 } else {
  // qui so che b_ptr NON sta puntando ad un oggetto di tipo D
 }
}
```

3. Il cast NON è strettamente necessario (in quanto la corrispondente conversione implicita è consentita dal linguaggio), ma il programmatore preferisce comunque la forma esplicita a scopo di documentazione, per tenere una traccia esplicita della conversione di tipo effettuata e migliorare la leggibilità del codice. In altre parole, il programmatore ritiene che il cast sia necessario dal punto di vista metodologico (anche se non lo è dal punto di vista tecnico). Esempio:

```
double d = /* .... */;
// La conversione implicita double->int è ammessa, ma usando il cast
// il programmatore vuole probabilmente attirare l'attenzione sul
// fatto che passando da un tipo floating point ad un tipo intero
// tipicamente si perde informazione.
int approx = static_cast(d);
```

4. Un caso speciale di uso (qualcuno potrebbe pure dire "abuso") di un cast esplicito riguarda la conversione di una espressione al tipo void (che non ha valori), che intuitivamente corrisponde ad una richiesta di "scartare" o "ignorare" il valore dell'espressione. Per convenzione, il cast a void si può usare per silenziare alcune segnalazioni di warning fornite dal compilatore (questa convenzione è rispettata sia da g++ che da clang++). Esempio:

```
// Il parametro size lo si usa solo nella assert e quindi, quando le
// asserzioni NON sono attivate, il compilatore mi segnalerebbe
// il suo mancato uso mediante un warning;
// il cast esplicito serve a silenziare questo warning.
void foo(int pos, int size) {
 assert(0 <= pos && pos < size);
 static_cast(size);
 /* ... codice che non usa size ... */
}
```

   In questo caso spesso si usa, per convenzione, un cast stile C: `(void) size`. In realtà, questo è l'unico caso in cui l'uso di un cast stile C (in C++) è tollerato: ogni altro uso è considerato (giustamente) cattivo stile.

Torna all'indice

Tipologie di cast

Descriviamo ora brevemente le diverse tipologie di cast:

- [[#static_cast]]
- [[#dynamic_cast]]
- [[#const_cast]]
- [[#reinterpret_cast]]
- [[#cast funzionale]]
- [[#cast stile C]]

Torna all'indice

---

static_cast

Probabilmente, è il cast utilizzato più frequentemente. La sintassi

```
static_cast(expr)
```

calcola un nuovo valore ottenuto dalla conversione del valore dell'espressione `expr` al tipo `T`. Il cast è legittimo in uno dei casi seguenti (elenco parziale, non esaustivo):

- è legittima la corrispondente conversione implicita (caso banale);

  ```
  double d = 3.14;
  int approx = static_cast(d);
  ```

- è legittima la costruzione diretta di un oggetto di tipo T passando expr come argomento;

  ```
  Razionale r = static_cast(5);
  ```

- si effettua la conversione inversa rispetto ad un sequenza di conversione implicita ammissibile (con alcune restrizioni, per esempio non si possono invertire le trasformazioni di lvalue);

```
int i = 42;
void* v_ptr = &i;
int* i_ptr = static_cast(v_ptr);
```

- il cast implementa un downcast in una gerarchia di classi;
- il cast implementa un cast da un tipo numerico ad un tipo enumerazione;
- il tipo destinazione è void.

Torna all'indice

---

dynamic_cast

Il dynamic_cast è uno degli operatori che forniscono il supporto per la cosiddetta *RTTI* (Run-Time Type Identification, cioè identificazione del tipo a tempo di esecuzione). I dynamic cast possono essere usati per effettuare conversioni all'interno di una gerarchia di classi legate da ereditarietà (singola o multipla). In particolare, si possono effettuare:

- up-cast: conversione da classe derivata a classe base; effettuata raramente mediante dynamic_cast, in quanto è una conversione consentita anche implicitamente e quindi non necessita della RTTI.
- down-cast: conversione da classe base a classe derivata; è il caso più frequente di utilizzo del dynamic cast, in quanto si sfrutta la RTTI per verificare che la conversione sia legittima.
- mixed-cast: caso particolare che si verifica quando si utilizza l'ereditarietà multipla; consiste in uno spostamento nella gerarchia di ereditarietà ottenibile combinando up-cast e down-cast (da cui il nome di cast "misto"); siccome prevede comunque la presenza di down-cast, anche in questo caso si ha un uso non banale della RTTI.

Il dynamic cast si può applicare ai tipi puntatore (caso tipico) e anche ai tipi riferimento (caso raro), con una importante differenza semantica.

Supponiamo di avere la seguente gerarchia:

```
struct B { /* ... */ };
struct D1 : public B { /* ... */ };
struct D2 : public B { /* ... */ };
```

e di avere la funzione

```
void foo(B* b_ptr) { /* ... */ }
```

Se la classe B è dinamica (ovvero, se contiene almeno un metodo virtuale) allora è dotata delle informazioni per la *RTTI* e possiamo applicare cast dinamici ai puntatori per sapere se sono di un determinato tipo. Per esempio:

```
D1* d1_ptr = dynamic_cast(b_ptr)
```

Dopo l'esecuzione di questo cast, se il puntatore `b_ptr` punta ad un oggetto di tipo `D1` (incluso eventualmente un oggetto di una classe derivata, anche indirettamente, da `D1`), allora `d1_ptr` avrà assegnato un valore NON nullo. Se invece `b_ptr` non punta ad un oggetto di tipo `D1` (per esempio, punta ad un oggetto di tipo `D2`), allora a `d1_ptr` viene assegnato il puntatore nullo. Di conseguenza, il programmatore può sapere se il cast è andato a buon fine controllando se il puntatore è non nullo:

```
if (d1_ptr != nullptr) {
/* d1_ptr è valido */
}

if (d1_ptr) {
  /* equivalente: ho sfruttato la conversione a bool */
}

if (D1* d1_ptr = dynamic_cast(b_ptr)) {
  /* equivalente: ho compattato cast e test */
}
```

Il caso di conversione per un riferimento è diverso (e raro), perché non esiste il concetto di riferimento nullo e quindi NON possiamo usare facilmente cast dinamici su riferimento per fare dei test RTTI. Se proviamo ad eseguire questo:

```
D1& d1_ref = dynamic_cast(*b_ptr)
```

se `b_ptr` punta ad un `D1`, il cast va a buon fine e `d1_ref` è inizializzato correttamente; se invece NON punta a `D1`, il cast dinamico fallisce e, non potendo segnalare la cosa con il riferiemento nullo, genera una eccezione (di tipo `std::bad_cast`).

Torna all'indice

---

const_cast

Il const_cast viene usato per rimuovere la qualificazione const. Tipicamente, si applica ad un riferimento o puntatore ad un oggetto qualificato `const` (cioè non modificabile) per ottenere un riferimento o puntatore ad un oggetto non qualificato (e quindi modificabile).

```
void promessa_da_marinaio(const int& ci) {
 int& i = const_cast(ci);
 ++i;
}
```

La funzione ha promesso al chiamante che NON modificherà l'argomento, ma si rimangia la promessa, elimina la qualificazione `const` e poi modifica l'argomento (proprio quello passato dal chiamante, non una copia).

Usando il `const_cast`, quindi, potremmo "rompere" il contratto stipulato con l'utente. Tra i pochi casi in cui può essere legittimo usare questo tipo di cast possiamo elencare i metodi di una classe che devono modificare la rappresentazione interna di un oggetto, senza però alterarne davvero il significato. Si tratta quindi di metodi che mantengono la "constness" a livello logico, pur violandola a livello fisico.

Esempio: Una classe mantiene un collezione di elementi ed è fornita di un metodo (etichettato const) che stampa gli elementi secondo un dato ordinamento. L'ordinamento è costoso da calcolare e quindi la collezione è mantenuta internamente NON ordinata. Quando però mi viene richiesta una stampa ordinata, potrei decidere di modificare la rappresentazione interna allo scopo di memorizzare la sequenza ordinata (di modo che successive chiamate della routine di stampa siano più efficienti). In questo caso, la routine di stampa potrebbe usare un `const_cast` per modificare la rappresentazione interna (senza però modificare dal punto di vista semantico la collezione).

> Nota: alcuni usi di `const_cast` si potrebbero eliminare mediante l'utilizzo del modificatore mutable su alcuni dati membro di una classe.

Torna all'indice

---

reinterpret_cast

Un reinterpret cast può essere usato per effettuare le seguenti conversioni:

- da un tipo puntatore ad un tipo intero (sufficientemente grande da poter rappresentare il valore del puntatore);
- da un tipo intero/enumerazione ad un tipo puntatore;
- da un tipo puntatore (oppure riferimento) ad un altro tipo puntatore (oppure riferimento).

> Non è possibile usare un `reinterpret_cast` per rimuovere la qualificazione `const` (occorre usare il `const_cast`).
>
> Nel caso del `reinterpret_cast` (diversamente dallo `static_cast`) le conversioni tra puntatori sono consentite anche quando i due tipi puntati NON sono in alcuna relazione tra di loro (in particolare, anche quando non fanno parte di una gerarchia di classi derivate). Quindi i `reinterpret_cast` sono una tra le forme di conversione più pericolose, in quanto i controlli di correttezza sono lasciati quasi completamente nelle mani del programmatore.

Torna all'indice

---

cast funzionale

La sintassi `T(expr)` oppure `T()`, dove `T` è il nome di un tipo, viene spesso indicata come "cast funzionale". Intuitivamente, corrisponde alla costruzione diretta di un oggetto di tipo `T`, usando un costruttore (nel secondo caso, il costruttore di default). Si parla di cast funzionale in quanto la sintassi si può applicare anche al caso dei tipi built-in (che in senso tecnico non sono dotati di costruttori). Nel caso di un tipo built-in, la forma `T()` produce la cosiddetta zero-initialization.

Esempio:

```
template
void foo(T t, U u) {
 if (t == T(u)) // cast funzionale
   // ...
}
```

Se foo viene istanziata con `T = int` e `U = double`, il test condizionale diventa

```
if (t == int(u))
```

nel quale abbiamo il cast funzionale `int(u)`.

Torna all'indice

---

cast stile C

Hanno la sintassi

```
(T) expr
```

Il loro uso è considerato cattivo stile (tranne il caso nominato sopra del cast a void per sopprimere warning del compilatore), perché:

- sono difficili da individuare nel codice mediante ricerca testuale;
- non differenziano le diverse tipologie di cast.

Con i cast stile C si possono simulare `static_cast`, `const_cast` e `reinterpret_cast`, ma NON si possono effettuare i `dynamic_cast` (in particolare, non hanno accesso a informazioni *RTTI* e quindi non effettuano nessun controllo a run-time).

Torna all'indice