Oggetti C++ e copie

G. Lettieri

24 Marzo 2021

1 Oggetti temporanei

Tutte le espressioni C++ hanno un tipo e, normalmente, un valore di quel tipo[1] Il valore è, in generale, un oggetto, che deve dunque essere memorizzato da qualche parte. Nei casi più semplici il compilatore può semplicemente memorizzare il valore dell'espressione in uno o più registri, ma nel caso generale può essere costretto a creare un *oggetto temporaneo* che possa contenere il valore dell'espressione. Gli oggetti temporanei sono oggetti senza nome, creati e distrutti automaticamente dal compilatore e non esplicitamente dal programmatore.

La creazione di oggetti temporanei può essere necessaria quando il tipo dell'espressione è struttura, classe o unione e l'oggetto è più grande di 16 byte, oppure il programmatore ha ridefinito alcune delle funzioni speciali di quel tipo (come il costruttore di copia o il distruttore). In questi casi diremo che l'oggetto ha un tipo "non banale".

Vediamo alcuni esempi di espressioni, assumendo che `C` sia una classe o struttura non banale (gli esempi non esauriscono in alcun modo tutti i tipi possibili di espressioni C++, ma sono sufficienti per i nostri scopi).

1.1 Singolo oggetto

Consideriamo il seguente frammento di codice:

```
1  C c, x;
2  // ...
3  x = c;
```

La parte dopo l'operatore di assegnamento, alla riga 3, è una espressione di tipo "classe `C`". In questo caso l'espressione consiste solo dell'oggetto `c` e non c'è bisogno di allocare oggetti temporanei: l'oggetto `c` stesso funge da valore dell'espressione. L'assegnamento a `x` verrà eseguito invocando sull'oggetto `x` l'operatore di assegnamento della classe `C` con un riferimento a `c`. Ovviamente l'operatore di assegnamento potrebbe essere quello di default, che si limiterà a copiare i membri di `c` nei corrispondenti membri di `x`.

[1] Una eccezione sono le espressioni di tipo **`void`**, che non hanno un valore.

1.2 Chiamata di funzione

Consideriamo ora una funzione che restituisca un oggetto di classe `C` per valore.

```
1  C g();
2
3  void f() {
4     C x;
5     // ...
6     x = g();
7  }
```

La parte dopo l'operatore di assegnamento alla riga 6 è, di nuovo, una espressione di tipo "classe `C`". Questa volta, però, è necessario allocare un oggetto temporaneo per contenere il valore dell'espressione restituito dalla funzione `g()`, che chiamiamo *oggetto risultato*. L'operatore di assegnamento su `x` riceverà poi un riferimento a questo oggetto risultato.

Come tutti gli oggetti, anche l'oggetto risultato deve essere: (i) allocato e (ii) inizializzato. L'allocazione (i) è svolta dalla funzione in cui compare l'espressione (`f()`, nel nostro caso), che tipicamente l'allocherà sulla sua pila, come se fosse una ulteriore variabile locale. L'inizializzazione (ii), invece, è svolta dalla funzione che restituisce il valore (`g()`, nel nostro caso). A questo scopo, la funzione `f()` passa un puntatore all'oggetto risultato come primo argomento nascosto della funzione `g()`. Nell'ABI System V tale puntatore sarà dunque passato tramite il registro **`%rdi`**; eventuali altri parametri di `g()` saranno passati usando i registri da **`%rsi`** in poi.

All'interno della funzione `g()` ci saranno una o più istruzioni **`return`** con una espressione compatibile con il tipo `C`. La traduzione di queste istruzioni comporta i seguenti passi:

- la funzione `g()` deve prima calcolare il valore dell'espressione contenuta nel **`return`**, ottenendo un altro oggetto di tipo `C`, sia y (si noti che, in base alla forma dell'espressione, y potrebbe essere un *altro* oggetto temporaneo, questa volta creato da `g()` e locale ad essa);
- la funzione `g()` dovrà invocare il costruttore di copia della classe `C` sull'oggetto risultato creato da `f()`, passandogli un riferimento all'oggetto y che contiene il risultato dell'espressione da restituire. Come caso particolare, se il programmatore della classe `C` non ha ridefinito il costruttore di copia, la funzione `g()` dovrà eseguire lei stessa la copia membro a membro tra y e l'oggetto risultato creato da `f()`. Per rispettare le regole dei costruttori previste dall'ABI System V, la funzione dovrà anche, in questo caso, lasciare l'indirizzo dell'oggetto risultato in **`%rax`**.

Al ritorno da `g()`, la funzione `f()` dovrà completare l'assegnamento tra l'oggetto risultato e `x` e poi distruggere l'oggetto risultato (invocare il distruttore, se definito).

In generale, gli oggetti temporanei creati durante la valutazione di una espressione sono tutti distrutti al termine dell'istruzione in cui si trovava l'espressione (il loro tempo di vita si estende fino al punto e virgola che si trova alla fine dell'istruzione).

1.3 Costruttore

Una espressione può consistere anche in una chiamata esplicita ad uno dei costruttori della classe. Per esempio, se `C` ha un costruttore con un parametro **`int`** e `x` è di tipo `C`, l'assegnamento

```
1    x = C(100); // o meglio C{100} con sintassi C++11
```

crea un oggetto temporaneo (inizializzato invocando il costruttore `C(int)` con parametro `100`) e poi lo assegna a `x`.

Allo stesso modo, all'interno di una funzione che restituisce `C` per valore (come la funzione `g()` dell'esempio precedente), si può trovare una istruzione come

```
1    return C(100); // C{100}
```

Anche qui l'espressione `C(100)` crea un oggetto temporaneo, che poi deve essere usato per inizializzare (tramite costruttore di copia) l'oggetto temporaneo allocato dal chiamante della funzione.

Si noti, incidentalmente, che anche "**`return`** `100;`" avrebbe lo stesso effetto, in quanto il compilatore inserirebbe automaticamente la chiamata al costruttore `C(int)`, a meno che questo non sia dichiarato **`explicit`**.

2 Costruttori di copia

Il costruttore di copia di una classe `C` deve essere invocato ogni volta che un oggetto di classe `C`, sia esso temporaneo o non temporaneo, deve essere inizializzato a partire da un altro oggetto di classe `C`, temporaneo o no che sia. Assumiamo che `e` sia una *espressione* di tipo `C` (come, per esempio, quelle viste nella sezione precedente). Il costruttore di copia va invocato tre occasioni:

1. durante una inizializzazione del tipo "`C c = e;`" o "`C c(e);`" (la differenza è solo sintattica);
2. in una espressione del tipo "`f(e)`", se `f()` accetta un argomento di tipo `C` per valore;
3. in una istruzione del tipo "**`return`** `e;`" in una funzione che restituisce un tipo "`C`" per valore.

In ogni caso il costruttore di copia riceve come argomento un riferimento all'oggetto che contiene il risultato di `e` (che, si ricordi, potrebbe essere un oggetto temporaneo). Per quanto riguarda invece l'oggetto da inizializzare (puntatore

this all'interno del costruttore), nel caso 1 il costruttore inizializza `c`, nel caso 2 inizializza *il parametro di* `f` e nel caso 3, come abbiamo già visto, inizializza l'oggetto risultato passato dal chiamante della funzione.

Si noti che il parametro di `f` (caso 2) è un altro oggetto di classe `C` che deve essere allocato e inizializzato. Le regole su come ciò avviene sono complicate e noi eviteremo di trattare questo caso. Negli esercizi in cui una funzione riceve un parametro di tipo classe per valore, la classe sarà sempre "banale" (dimensione massima di 16 byte e senza ridefinizione del costruttore di copia e/o del distruttore).

2.1 Elisione dei costruttori di copia

Le regole che abbiamo visto, se applicate alla lettera, comportano la creazione di un gran numero di oggetti temporanei e di invocazioni dei costruttori di copia e dei distruttori. Si consideri, ad esempio, il seguente codice:

```
1  C g() {
2    return C(100);
3  }
4
5  void f() {
6     C x = g();
7  }
```

Per inizializzare `x`, la funzione `f()` deve prima valutare l'espressione "`g()`". Per farlo alloca un risultato temporaneo t_1 sul suo stack e invoca `g()`, passandole l'indirizzo di t_1 come primo parametro nascosto. A questo punto la funzione `g()` deve eseguire l'istruzione "**return** `C(100)`" e, per farlo, deve prima valutare l'espressione "`C(100)`". Questo comporta la creazione di un oggetto temporaneo t_2, su cui invoca il costruttore `C(`**int**`)` con argomento `100`. Quindi, `g()` invoca il costruttore di copia su t_1 con argomento (un riferimento a) t_2, distrugge t_2 e ritorna. Ora la funzione `f()` invoca il costruttore di copia su `x` con argomento (un riferimento a) t_1 e infine distrugge t_1.

Lo standard C++, però, permette e, a partire dal C++17, *rende obbligatorie* alcune ottimizzazioni che riescono ad evitare la creazione di molti oggetti temporanei e, dunque, anche delle corrispondenti invocazioni dei costruttori di copia e dei distruttori. In particolare, alla riga 2 il compilatore può (o deve) invocare il costruttore *direttamente* sull'oggetto risultato (cioè, si ricordi, quello all'indirizzo ricevuto come primo parametro nascosto), senza creare e poi distruggere l'oggetto temporaneo che avevamo chiamato t_2. Inoltre, alla riga 6, il compilatore può (o deve) evitare di costruire l'oggetto risultato che deve contenere il risultato dell'espressione e può (o deve) usare direttamente `x`.

Nell'esempio qui sopra, l'effetto combinato delle due ottimizzazioni è il seguente: alla riga 6 la funzione `f()` passa a `g()`, come oggetto risultato, direttamente l'indirizzo di `x`; alla riga 2 la funzione `g()` invoca il costruttore `C(`**int**`)` direttamente su `x`. Nessun oggetto temporaneo è stato dunque creato, e tantomeno copiato e distrutto.

Si noti che le normali ottimizzazioni non possono cambiare gli effetti visibili di un programma, ma queste due sono autorizzate a farlo. Infatti, se i costruttori di copia o i distruttori avessero effetti collaterali (per esempio, se stampassero qualcosa), il comportamento visibile del programma cambierebbe nel caso ottimizzato rispetto al caso non ottimizzato (si noti però che il caso non ottimizzato non è lecito in C++17). Questa eccezione è ammessa solo per i costruttori di copia e i distruttori. Lo standard permette sì di elidere anche le operazioni di assegnamento per copia, ma solo se il compilatore è sicuro che ciò non causi effetti visibili.

Queste due ottimizzazioni prendono il nome di *copy elision*. L'ottimizzazione effettuata alla riga 2 è anche chiamata *Return Value Optimizazion* (RVO). Lo standard ammette (ma non rende obbligatoria) anche un'altra ottimizzazione, ancora più sofisticata, detta *Named* Return Value Optimizazion (NRVO), che possiamo illustrare con un esempio:

```
1  C g() {
2    C c(100);
3    c.do_something();
4    return c;
5  }
```

La funzione, alla riga 4, sta restituendo per nome una variabile locale (`c`) che ha lo stesso tipo del valore da restituire (classe `C`, per valore). Lo standard autorizza il compilatore a non creare `c` e ad usare, al suo posto, direttamente l'oggetto risultato ricevuto dal chiamante di `g()` (tramite il puntatore contenuto nel registro `%rdi`). Grazie a questa ottimizzazione, la riga 2 invocherà il costruttore `C(int)` direttamente sull'oggetto risultato, la riga 3 invocherà la funzione membro `do_something()` sullo stesso oggetto e la riga 4 non farà niente, in quanto l'oggetto risultato conterrà già, a quel punto, quello che sarebbe stato il contenuto di `c`.