lvalue e rvalue

Categorie di espressioni: lvalue e rvalue

Le espressioni del C++ possono essere classificate in a) **lvalue** (*left* value) b) **xvalue** (*expiring* lvalue) c) **prvalue** *primitive* rvalue)

L'unione di lvalue e xvalue forma i glvalue (generalized lvalue). L'unione di xvalue e prvalue forma gli rvalue (right value).

Intuitivamente, un glvalue è una espressione che permette di stabilire l'identità di un oggetto in memoria. Il nome "left value", in origine, indicava che tali espressioni potevano comparire a sinistra dell'operatore di assegnamento.

Esempio:

```
int i;
int ai[10];
i = 7;     // l'espressione i è un lvalue (e quindi un glvalue)
ai[5] = 7; // l'espressione ai[5] è un lvalue (e quindi un glvalue)
```

![[lvalue_rvalue.png]]

m: movable *i*: has identity

Torna all'indice

---

xvalue

Un xvalue è un glvalue che denota un oggetto le cui risorse possono essere riutilizzate, tipicamente perché sta terminando il suo lifetime (expiring lvalue). Un lvalue è un glvalue che non sia un xvalue.

Esempio:

```
Matrix foo1() {
 Matrix m;
 // ... codice
 return m; // l'espressione m è un xvalue
}
```

`m` verrà distrutto automaticamente in uscita dal blocco nel quale è stato creato; il valore ritornato dalla funzione non è `m`, ma una sua "copia".

```
void foo2() {
 Matrix m1;
 m1 = foo1();    // l'espressione foo1(), cioè il risultato ottenuto
      // dalla chiamata di funzione, è un xvalue
}
```

La soluzione del C++11 prevede non di copiare gli xvalue, ma di spostarli in una nuova locazione, dato che questi non sono più richiesti dalla funzione chiamata. In particolare, non adottando questo approccio, le copie sarebbero due:

- `m` viene copiato nella locazione di ritorno della funzione `foo1()`
- l'oggetto temporaneo restituito da `foo1()` viene copiato in `m1`

Torna all'indice

---

prvalue

Un prvalue è una espressione che denota un valore "primitivo", ovvero un valore costante o il risultato di una computazione. Il nome "right value", in origine, indicava che tali espressioni potevano comparire *solo* a destra dell'operatore di assegnamento (ovvero, espressioni che darebbero errore se comparissero a sinistra). Intuitivamente, un prvalue *NON* identifica un oggetto in memoria e quindi non è lecito assegnarvi un valore e non ha nemmeno senso prenderne l'indirizzo.

Esempio:

```
int i;
i = 5;     // l'espressione 5 è un prvalue (e quindi un rvalue)
i = 4 + 1; // l'espressione 4 + 1 è un prvalue (e quindi un rvalue)
i = i + 1; // l'espressione i + 1 è un prvalue (e quindi un rvalue)
```

> In alcuni casi, un prvalue può essere "materializzato", creando un oggetto temporaneo (un lvalue) che viene inizializzato con il valore del prvalue. Questo è quello che succede, per esempio, quando ad una funzione che ha un argomento di tipo riferimento a costante viene passato un prvalue.

Esempio:

```
void foo(const double& d);
void bar() {
 foo(0.5);
}
```

Qui sopra `0.5` è un rvalue; viene materializzato in un oggetto temporaneo (un lvalue) con cui viene inizializzato il riferimento a lvalue `d`.

La classificazione delle espressioni in lvalue, xvalue e prvalue è rilevante per capire la differenza tra riferimenti a lvalue (`T&`) e riferimenti a rvalue (`T&&`). Questi ultimi sono stati introdotti nel C++11 per risolvere problemi tecnici del linguaggio che impedivano di fornire implementazioni efficienti per alcuni costrutti.

Torna all'indice

---

lvalue vs rvalue

Nel C++ 2003, ogni classe era fornita (se non veniva fatto qualcosa per disabilitarle) di 4 funzioni speciali:

- costruttore di default
- costruttore di copia
- assegnamento per copia
- distruttore

```
struct Matrix {
 Matrix();  // costruttore di default
 ~Matrix(); // distruttore

 Matrix(const Matrix&);            // costr. di copia (copy ctor)
 Matrix& operator=(const Matrix&); // assegn. per copia (copy assign.)

 // ... altro
};
```

Una funzione che avesse voluto prendere in input un oggetto `Matrix` e produrre in output una sua variante modificata (senza modificare l'oggetto fornito in input), doveva tipicamente ricevere l'argomento per riferimento a costante e produrre il risultato per valore:

```
Matrix bar(const Matrix& arg) {
 Matrix res = arg; // copia (1)
 // modifica di res
 return res; // ritorna una copia (2)
}
```

Questo era fonte di inefficienze, perché:

1. Non c'era un modo semplice per il chiamante di comunicare il fatto che, in alcuni casi (non tutti), la risorsa passata in input non era più di suo interesse e quindi poteva essere modificata in loco, invece di effettuare la prima copia.
2. Non c'era modo semplice per la funzione per ritornare l'oggetto `res` senza fare la seconda copia (si noti che non è possibile ritornare per riferimento, perché si creerebbe un dangling reference).

Nel C++11, alle 4 funzioni speciali delle classi ne sono state aggiunti altre due:

- costruttore per spostamento (move constructor), e
- assegnamento per spostamento (move assignement)

che lavorano su riferimenti a rvalue:

```
struct Matrix {
 Matrix();  // costruttore di default
 ~Matrix(); // distruttore

 Matrix(const Matrix&);            // costr. di copia (copy ctor)
 Matrix& operator=(const Matrix&); // assegn. per copia (copy assign.)

 Matrix(Matrix&&);            // costr. per spostamento (move ctor)
 Matrix& operator=(Matrix&&); // assegnam. per spostamento (move assign.)

 // ... altro
};
```

Intuitivamente, una funzione che riceve come argomento un riferimento a rvalue (`Matrix&&`) sa che l'oggetto riferito può essere solo un prvalue o un xvalue. In entrambi i casi, le risorse contenute nell'oggetto non possono essere utilizzate da altri e quindi possono essere spostate dall'oggetto (si potrebbe dire "rubate" all'oggetto, che ne era il proprietario) invece che copiate, guadagnando in efficienza.

Riconsideriamo l'esempio precedente, assumendo che sia disponibile il costruttore per spostamento per `Matrix`:

```
Matrix bar(const Matrix& arg) {
 Matrix res = arg; // copia (1)
 // modifica di res
 return res; // sposta (non copia)
}
```

Il compilatore si accorge che, nella `return res`, l'espressione `res` è un xvalue e quindi utilizza il costruttore di spostamento (invece della copia) per restituirlo al chiamante.

Volendo, è possibile ottimizzare anche la prima copia, fornendo una versione alternativa (in overloading) della funzione `bar`:

```
Matrix bar(Matrix&& arg) {
 // modifica in loco di arg
 return arg; // sposta (non copia)
}
```

Questa seconda versione verrà invocata quando alla funzione viene passato un rvalue (che potrà essere modificato direttamente), mentre la prima versione verrà usata per gli lvalue.

E' anche possibile fondere le due versioni in una sola, usando il passaggio dell'argomento per valore:

```
Matrix bar(Matrix arg) {
 // modifica in loco di arg
 return arg; // sposta (non copia)
}
```

In questo terzo caso, all'atto di effettuare il passaggio dell'argomento alla funzione bar, vi sono due possibilità:

1. il chiamante fornisce un lvalue: verrà utilizzato il costruttore di copia sull'argomento, comportandosi come nel caso di `Matrix bar(const Matrix& arg);`
2. il chiamante fornisce un rvalue: verrà utilizzato il costruttore per spostamento, senza copie, come nel caso di `Matrix bar(Matrix&& arg).`

Torna all'indice

La funzione `std::move`

Supponimo che il chiamante si trovi a dovere invocare la funzione `bar` discussa sopra con un lvalue `m` di tipo `Matrix`, ma non è interessato a preservare il valore di `m` e quindi lo vorrebbe "spostare" nella funzione `bar`, evitando la copia (costosa e inutile).

Se si usa la chiamata:

```
bar(m);
```

siccome `m` è un lvalue verrebbe comunque invocato (almeno una volta) il costruttore per copia. Per evitarla, occorre un modo per convertire il tipo di `m` da riferimento a lvalue (`Matrix&`) a riferimento a rvalue (`Matrix&&`): questo è esattamente l'effetto ottenuto usando la funzione di libreria `std::move`.

```
bar(std::move(m));
```

> Si noti che la `std::move(m)` NON "muove" nulla: piuttosto, trasformando un lvalue in rvalue, lo rende "movable" (spostabile); lo spostamento vero e proprio viene effettuato durante il passaggio del parametro.

Torna all'indice