10 - Lvalue e Rvalue

In C++, le espressioni possono essere classificate in tre categorie principali, che aiutano a definire il *comportamento degli oggetti*, la *gestione della memoria* e l'*ottimizzazione*. Queste categorie sono:

lvalue (left value):

un'espressione che si riferisce a un oggetto con un'identità in memoria, quindi qualcosa che può essere "a sinistra" dell'operatore di assegnamento.

```
int i;
i = 5; // 'i' è un lvalue perché identifica un oggetto in memoria
```

xvalue (expiring value):

una forma speciale di lvalue che denota un oggetto "in scadenza", cioè che sta per terminare il proprio ciclo di vita. Ad esempio, il risultato di una chiamata a funzione che restituisce un oggetto temporaneo.

```
Matrix foo();
Matrix m = foo(); // 'foo()' è un xvalue: il valore temporaneo può essere spostato
```

prvalue (pure value):

un'espressione che rappresenta un valore puro, come una costante o il risultato di un'operazione, che non identifica alcun oggetto in memoria. Non è possibile prendere l'indirizzo di un prvalue o assegnargli un valore.

```
int x = 5; // '5' è un prvalue
int y = x + 1 // 'x + 1' è un prvalue
```

materializzazione dei prvalue:
in alcuni contesti, un prvalue può essere "materializzato", cioè il compilatore crea un oggetto temporaneo (un lvalue) che rappresenta il valore puro.

```
void foo(const int& x); // Accetta un riferimento costante a un lvalue.
foo(42);                // '42' (prvalue) viene materializzato come temporaneo.
```

Relazioni tra le categorie

- **glvalue (generalized value)**: l'unione tra lvalue e xvalue. Identifica sempre un oggetto in memoria.

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

- un xvalue è un glvalue che denota un oggetto le cui risorse possono essere riutilizzate, tipicamente perchè sta terminando il suo lifetime;
- un lvalue è un gvalue che non sia un xvalue;

```
Matrix foo1() {
    Matrix m;
    //codice
    m.transpose(); //qui m è un lvalue (quindi glvalue)
    return m; //qui m è un xvalue (quindi glvalue)
}
/* '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 */
```

- **rvalue (right value)**: l'unione tra xvalue e prvalue. Indica un valore temporaneo o calcolato che non ha un'identità stabile in memoria.
  nota: gli xvalue sono sia glvalue, sia rvalue.

Riferimenti a lvalue e rvalue

C++ supporta due tipi principali di riferimenti, ciascuno associato a diverse categorie di espressioni:

1. **riferimenti a lvalue (`T&`)**:
   Questi riferimenti sono utilizzati per accedere a oggetti esistenti e non temporanei.

   ```
   int x = 10;
   int& ref = x; //riferimento a lvalue
   ```

2. **riferimenti a rvalue (`T&&`)**:
   questi riferimenti permettono di lavorare con oggetti temporanei (prvalue e xvalue). Sono fondamentali per implementare il "move semantics".

   ```
   Matrix foo();
   Matrix&& temp = foo(); // 'foo()' è un rvalue, catturato con un riferimento a
   rvalue
   ```

C++ 03: Costrutti e Assegnazioni

Ogni classe dispone di quattro funzioni speciali generate automaticamente:

1. **costruttore di default**: inizializza l'oggetto;
2. **costruttore di copia**: crea una copia di un altro oggetto della stessa classe;
3. **assegnazione per copia**: copia i valori da un altro oggetto;
4. **distruttore**: libera le risorse dell'oggetto.

```
struct Matrix {
    Matrix();                   // Costruttore di default
    ~Matrix();                  // Distruttore
    Matrix(const Matrix&);      // Costruttore di copia
```

```
    Matrix& operator=(const Matrix&); // Assegnazione per copia
};
```

Problemi:
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)
}
```

1. parametro passato per riferimento costante:
   `arg` è passato per riferimento a costante, il che significa che il chiamante non crea una copia completa dell'oggetto. Questo è utile per evitare il costo di copia quando il parametro è un oggetto grande; tuttavia non può essere modificato perché è `const`.
2. creazione di `res`:
   la variabile `res` viene inizializzata copiando l'oggetto `arg`, dunque avviene la *prima copia*, necessaria in quanto vogliamo modificare `res` senza influenzare `arg`.
3. ritorno di `res`:
   quando la funzione restituisce `res`, viene effettuata un'ulteriore copia per restituire l'oggetto al chiamante, dunque avviene la *seconda copia*.
   Inefficienze:
4. l'oggetto `arg` viene copiato per creare `res`. Questo è inefficiente quando il chiamante non ha più bisogno di `arg` e sarebbe disposto a lasciarlo modificare direttamente.

   ```
   Matrix m1;
   // Supponiamo che il chiamante non abbia più bisogno di m1
   Matrix m2 = bar(m1); // Il chiamante vuole che m1 venga usato per produrre m2
   ```

   anche se il chiamante non ha più bisogno di `m1`, la funzione `bar` non lo sa; quindi, per sicurezza, deve copiare `m1` (usando il costruttore di copia) per creare `res`. Questo è uno spreco di risorse perché:
   - la copia potrebbe essere costosa;
   - se il chiamante avesse un modo per segnalare che `m1` non è più necessario, la funzione potrebbe riutilizzare direttamente le risorse di `m1` invece di copiarle.
5. quando la funzione ritorna, il compilatore deve creare un'altra copia di `res` per trasferire il valore al chiamante.

   ```
   Matrix m2 = bar(m1);
   ```

   la funzione `bar` deve restituire un nuovo oggetto `Matrix`, questo significare che il valore di `res` deve essere copiato nel nuovo oggetto `m2`.
   Questo è un altro spreco di risorse perché:
   - l'oggetto `res` non è più necessario dopo il ritorno.

- sarebbe più efficiente "spostare" le risorse interne di res direttamente nel risultato, evitando la copia.

Soluzione introdotta nel C++ 11: move semantic

Vengono aggiunte alle 4 funzioni speciali delle classi altre due che lavorano su riferimenti a rvalue:

- **costruttore per spostamento** (move constructor);
- **assegnamento per spostamento** (move assignment).

```
struct Matrix {
Matrix (); // default constructor
Matrix (const Matrix&); // copy constructor
Matrix& operator=(const Matrix&); // copy assignment
Matrix (Matrix&&); // move constructor
Matrix& operator=( Matrix&&); // move assignment
~Matrix(); // destructor
// altro
};
```

Con l'introduzione delle semantiche di spostamento (move semantics), i problemi di inefficienza sono stati risolti:

1. **costruttore di spostamento**:
   - se la classe `Matrix` implementa un costruttore per spostamento, il compilatore può evitare di copiare `res` quando la funzione ritorna; invece di fare una copia di `res`, il compilatore sposta le risorse interne di `res` nel risultato della funzione;
2. **nuova implementazione di bar**:
   con il *move constructor* la seconda copia non avviene più, quando la funzione ritorna, il compilatore riconosce che `res` è un xvalue e utilizza il costruttore per spostamento invece del costruttore di copia.

Per evitare la prima copia

Supponiamo che il chiamante si trovi a dover invocare la funzione bar con un lvalue `m`, ma non è interessato a preservarne il valore: quindi lo vorrebbe "spostare" nella funzione bar evitando la copia. Se si usa la chiamata `bar(m);` dato che `m` è un lvalue viene comunque invocata, almeno una volta, la copia. Per evitare questa copia inutile, occorre un modo per convertire il tipo di `m` da riferimento a lvalue (`Matrix&`) a riferimento a rvalue (`Matrix&&`):

```
Matrix m = bar(std::move(m1)); // Sposta m1 invece di copiarlo
```

la `std::move` non "muove" nulla, ma trasformando un lvalue in rvalue, lo rende "movable", lo spostamento avviene durante il passaggio del parametro.

Versione generale

Una versione ancora più generale combina entrambe le versioni di `bar`:

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

- Se il chiamante passa un **lvalue**, il costruttore di copia sarà usato per inizializzare `arg`.
- Se il chiamante passa un **rvalue**, il costruttore di spostamento sarà usato per inizializzare `arg`.
  Vantaggio: Una sola funzione gestisce sia gli lvalue sia gli rvalue in modo efficiente.

13 - Progettazione di un tipo di dato concreto