C++ Tricks 3.1 左值右值與常量性 (lvalue，rvalue & constant)

從 farseerfc.wordpress.com 導入

3.1 左值右值與常量性(lvalue，rvalue & constant)

首先要搞清楚的是，什麼是左值，什麼是右值。這裏給出左值右值的定義：

1、左值是可以出現在等號(=)左邊的值，右值是隻能出現在等號右邊的值。

2、左值是可讀可寫的值，右值是隻讀的值。

3、左值有地址，右值沒有地址。

根據左值右值的第二定義，值的左右性就是值的常量性——常量是右值，非常量是左值。比如：

1=1;//Error

這個複製操作在C++中是語法錯誤，MSVC給出的錯誤提示爲“error C2106: '=' : left operand must be l-value”，就是說’=’的左操作數必須是一個左值，而字面常數1是一個右值。可見，嚴格的區分左值右值可以從語法分析的角度找出程序的邏輯錯誤。

根據第二定義，一個左值也是一個右值，因爲左值也可讀，而一個右值不是一個左值，因爲右值不可寫。

通常情況下，聲明的變量是一個左值，除非你指定const將它變成一個右值：

```
int lv=1;

const int rv=lv;
```

由於右值的值在程序執行期間不能改變，所以必須用另一個右值初始化它。

一個普通變量只能用右值初始化，如果你想傳遞左值，必須聲明一個引用或一個指針：

```
int & ref=lv;//用引用傳遞左值

int * plv=&lv;//傳遞指針以間接傳遞左值
```

必須用左值初始化引用，然而，可以用右值初始化常量引用：

```
int & r1=1; //Error!

const int & r2=1; //OK
```

這實際上相當於：

```
int _r2=1;

const int & r2=_r2;
```

這樣的寫法在函數體內沒什麼作用，但是在傳遞函數參數時，它可以避免潛在的(傳遞左值時的)複製操作，同時又可以接受右值。

通常情況下，函數的參數和返回值都只傳回右值，除非你明確的通過引用傳遞左值。

明確了左值與右值的區別，有助於我們寫函數時確定什麼時候應該有const，什麼時候不該有。比如，我們寫了一個代表數學中複數的類Complex：

class Complex;

然後，我們寫針對Complex的運算符重載：operator+和operator=。問題在於，參數和返回值應該是什麼類型，可選類型有四種：Complex、const Complex、Complex&、const Complex&。

對於operator+，我們不會改變參數的值，所以可以通過const Complex&傳遞參數。至於返回值類型，由於int類型的加法返回右值，所以根據Do as the ints do的原則，返回值類型爲const Complex：

const Complex operator+(const Complex&,const Complex&);

對於operator=，同樣要思考這些問題。我們寫入第一個參數，所以第一個參數爲Complex&，我們只讀取第二個參數，所以第二個參數爲const Complex&。至於返回值，還是Do as the ints do。int的賦值返回左值，不信你可以試一試：

int i;

(i=1)=2;

雖然比較傻，先將i賦爲1，再將其改爲2，但是這是被C++語法支持的做法，我們就理應遵守。所以返回第一個參數的左值：

Complex& operator=(Complex&,const Complex&);

const是C++引入的語言特性，也被ANSI C99借鑑，在經典版本的C語言中是沒有的。關於const的歷史，有幾點值得玩味。最初Bjarne Stroustrup引入const時，可寫性是和可讀性分開的。那時使用關鍵字readonly和writeonly。這個特點被首先提交到C的ANSI標準化委員會(當時還沒有C++標準化的計劃)，但是ANSI C標準只接受了readonly的概念，並將其命名爲const。隨後，有人發現在多線程同步的環境下，有些變量的值會在編譯器的預料之外改變，爲了防止過度優化破壞這些變量，C++又引入關鍵字violate。從語義特點來看，violate是const的反義詞，因爲const表示不會變的量，而violate表示會不按照預期自行變化的量。從語法特點而言，violate與const是極爲相似的，適用於const的一切語法規則同樣適用於violate。

值的常量性可以被劃分爲兩種：編譯期常量和運行期常量。C++語法並沒有嚴格區分這兩種常量，導致了少許混亂：

const int i=5;const int * pi=&i;

const_cast<int&>i=1;//對於運行期常量，在需要時可以去除它的常量性

int a[i];//對於編譯期常量，可以用它來指定數組大小

cout<<i<<sizeof(a)/sizeof(a[0])<<*pi;

這種將編譯期與運行期常量的特性混用的方法，勢必導致語義的混亂。數組a的大小最終是5，因爲採用了i的編譯期值，而不管i在運行期是否被改變了值。最後一句代碼將（有可能）輸出551，第一個i的值作爲一種優化在編譯期綁定，第二個值標明瞭a的大小，第三個值通過指針顯示地輸出i的運行期真實值。

在C++的近親C#的語法中，這兩種常量被嚴格地區分開：編譯期常量由const指定，只能是內建類型變量；運行期常量由readonly指定，可以是任何類型。永遠不會改變的常量，如圓周率pi的值，應該用const聲明；而其它有可能改變的常量，皆由readonly聲明。

C++中的const的特點更傾向於C#中的readonly，雖然語法上允許使用const的編譯期常量性，但正如上文所展示的，這容易造成混亂。爲了得到C#中const的語義，在C++中，我們不必迴歸惡魔#define的懷抱，可以使用所謂“匿名enum技巧”。當匿名聲明一個enum類型時，其中的枚舉值就是一個int類型的編譯期常量，比如：

```
enum{Size=5;};

int a[Size];
```

這種使用匿名enum來聲明編譯期常量的做法，被廣泛應用於STL、boost等模板庫的實現代碼中。