9. Operator Overloading in C++

本章將介紹如何讓 C++的運算子適用於自訂類別之物件，此過程稱為運算子覆載(Operator Overloading)。

運算子提供程式設計師簡潔的表示法，內建的型別能配合 C++大量的運算子一起使用。雖然 C++不允許創造新的運算子，不過卻允許程式設計師為其定義的新類別覆載(重新定義)大部分現有的運算子，以便能更適切地配合新型別的物件，並使的物件更易於被使用，程式更易於被理解。

我們之前介紹了拷貝指定運算子的覆載，其即是使用 C++運算子覆載之功能，為我們自訂的類別，重新定義 「=」算符的意義，我們於 operator=函式內所定義的程式命令將取代 C++類別對「=」算符所預設的逐員拷貝動作，而在我們對此類別之物件間使用等號時被呼叫使用。

對於某些種類的類別，特別是數學方面之類別，當然是可以以一般函式呼叫的型式進行物件間的所有運算，但不如以運算子來表示要來的淺顯易讀，例如有 a、b、c 三個矩陣物件，a = b + c 之表示要比 a=MaddM(b, c)要來的明白易懂。

這些類別最好能使用 C++豐富的內建運算子，並以運算子覆載賦予其對此類別的意義，並用其來進行物件的操作。有些運算子經常被覆載，例如拷貝指定運算子與某些算數運算子，如「+」、「-」等。

8.1 Operator Overloading Basic

C++的運算子，原本即已經對內建之各種不同型別，覆載了不同的意義，例如「+」對於 int 與 double，其所做的運算是完全不同的，但我們已很習慣均是使用「+」，來對各種型別作相加的計算。運算子覆載的目的即是讓我們也能對自訂的類別，也能如內建型別使用運算子作運算。

運算子覆載的方法是為欲覆載的運算子，如一般函式定義的方式，撰寫其定義，運算子本身之原型其實是一個函式，函式名稱即是保留字 operator 加上欲覆載的運算符號，例如，為某類別 classX 定義名為 operator+ 可以用來覆載加法運算子「+」，如下。

```
classX& operator+(const classX& rightObj)
{
    classX* ans = new class();
    ans->data1 = data1 + rightObj.data1;
    …….
    return *ans;
}
```

我們之前對於物件間的運算，都是使用一般函式的格式，例如 vec2 = MmultiV(mat1, vec1); ，使用運算子覆載，不但可達到同樣的功能，並使程式更容易了解，例如使用 vec2 = mat1 * vec1; 來取代之前的函式，只不過我們將 MmultiV 相同的函式內容，定義於 operator*。

運算子覆載無法自動作到，程式設計者必須定義運算子覆載函式來執行想要的運算。

運算子函數可以是類別的函式成員或是獨立的非成員函式，當他們是非成員函式時，一般都是相關類別的朋友函式，以提升執行的效率，我們先來看函式成員的例子。

程式範例: oop_ex83.cpp

注意要點:

1. 每一個運算子均需覆載後才能運用在類別物件上，不過有兩個運算子例外，即「=」與「&」，其不需覆載即可直接使用。未覆載前。「=」的預設意義是對物件作 memberwise copy(此對有指標成員的類別十分危險)，「&」的預設意義是傳回物件的位址值。
2. 覆載的運算子通常會將物件本身以傳參考的方式傳回(將運算子函式宣告為傳回值的參考，並於函式結束後以 return *this 傳回物件本身)，如此一來，我們可將數個覆載的運算子，放在同一個程式命令內使用(連續呼叫)，例如： ans = ++objA * objB + objC--;
3. 運算子覆載最適用於數學方面的類別，例如 matrix、vector。
4. 運算子覆載可允許為類別定義資料成員之存取運算子，如( )、[ ]。
5. 程式設計者做運算子覆載時，需了解預覆載運算子的意義與用法，以製作出合理的覆載內容。

8.2 Restriction on Operator Overloading Basic

C++運算子覆載功能有以下限制：

1. 以下運算子不可以被覆載：「.」、「.*」、「::」、「? :」、「sizeof」。
2. 運算子執行之優先順序不會因覆載而改變(例：always 先乘除後加減)。
3. 運算子的結合性也不會因覆載而改變。
4. 無法改變運算指所需的運算員個數(unary operator always take one operand, binary operator always takes two operands)。
5. 運算子只有經過明確的宣告定義才會被覆載，並不存在隱函式的覆載。例如定義了 + 與 = 後，並不表示也定義了 +=。
6. 只有已存在的運算子才可以被覆載，創造新的運算符號為不可。
7. 對於內建型別(int, double…)的運算子定義，無法以覆載改變之。運算子覆載只能用於自訂型別之物件(class type)，或自訂型別與內建型別的混和狀態。
8. 除了 operator( )外，運算子覆載函式之參數不可預設初值(default arguments is illegal)。

程式範例: oop_ex83.cpp

8.3 Overloading Unary and Binary Operators

運算子函數可以是類別的函式成員或是獨立的非成員函式，當他們是非成員函式時，一般都會宣告為相關類別的朋友(friend)函式，以提升效率。如為函式成員，函式內可使用 this 指標得到 unary 運算子的唯一或 binary 運算子左邊的運算原物件；如為非函式成員，則其作用之運算元均需明確地宣告為函式的輸入參數。

當覆載( )、[ ]、->或任何指定運算子(有「=」者)時，運算子覆載函式必須是類別的函式成員，而其他的運算子覆載函式則可以是函式成員或非成員函式。

如運算子覆載函式為類別的函式成員，則 unary 運算子唯一的或 binary 運算子左邊的運算元必須是此運算子覆載函式所屬類別之物件(因為如此才能呼叫其本身之函式成員)。如果 binary 運算子左邊的運算元必須是不同類別或內建的物件，則此運算子覆載函式必須為非成員函式。

例；

```
matrix m;
vector v;
……
v = m * v;
```

則 operator*覆載函式可能是：(1) matrix 的函式成員。(2)非函式成員。
不可能是：vector 的函式成員。

如同時需能做 v = v * m?

8.3.1 Overloading Unary Operators

類別的一元運算子覆載函式可以是無參數的 non-static 函式成員；或是具有一個參數的非函式成員，其參數必須為此類別之物件或參照到此類別物件之參考(不可為指標，Why?)。其格式如下；

格式：

```
傳回值   operator 運算子( ){  覆載內容; }
friend  傳回值   operator 運算子(作用類別名& ){  覆載內容; }
```

例：

```
Matrix& operator-();
friend Matrix& operator-(Matrix& m);
```

程式範例: oop_ex84.cpp oop_ex85.cpp

注意要點:

1. 「+」、「-」、「*」、「&」等四個運算子同時為 unary 與 binary，可擇一或二者同時覆載。
2. 運算子覆載的成員函式必須是 non-static 函式，如此才能存取 non-static 的資料。
3. 覆載的運算子通常會將物件本身以傳參考的方式傳回，以支援連續的呼叫。
4. 一元運算子覆載函式可以是無參數的函式成員(oop_ex84.cpp)，或是具有一個參數的非函式成員(oop_ex85.cpp)。

8.3.2 Overloading Binary Operators

類別的二元運算子覆載函式可以是具有一個參數的 non-static 函式成員；或是具有二個參數的非函式成員，其中一個參數必須為此類別之物件或參照到此類別物件之參考。其格式如下；

格式：

傳回值　operator 運算子(作用類別名&){ 覆載內容; }
friend 傳回值　operator 運算子(作用類別名&, 類別名&){ 覆載內容; }

例：

```
Matrix& operator*(Matrix& m);
friend Matrix& operator*(Matrix& m1, Matrix& m2);

Matrix& operator*(Vector& v);
friend Matrix& operator*(Matrix& m, Vector& v);
```

呼叫時，如二元運算子覆載函式定義為函式成員，則運算子右邊之物件為函式之呼叫參數；如二元運算子覆載函式定義為非函式成員，則運算子左邊之物件為函式之第一個呼叫參數，右邊之物件為函式之第二個呼叫參數。

例：

(1) Matrix& operator*(Vector& v);
m * v;　之二元運算子覆載函式呼叫為：　m.operator*(v);

(2) friend Matrix& operator*(Matrix& m, Vector& v);
m * v;　之二元運算子覆載函式呼叫為：　operator*(m, v);

程式範例: oop_ex85.cpp　oop_ex86.cpp　oop_ex87.cpp

注意要點:

1. 二元運算子覆載函式可以是具有一個參數的 non-static 函式成員(oop_ex85.cpp)；或是具有二個參數的非函式成員(oop_ex86.cpp)，其中一個參數必須為此類別之物件或參照到此類別物件之參考。

2. 覆載( )、[ ]、->或任何指定運算子(有「=」者)時，運算子覆載函式必須是類別的函式成員，不可為非函式成員。
3. 運算子覆載函式如為非函式成員，不一定要是類別的 friend 函式(oop_ex87.cpp)，如為此情況，類別需提供適當的私有成員存取函式。

8.4 Overloading << and >> Operators

C++中，<<與>>適用於處理內建型別資料的輸入與輸出，對於我們自訂的類別，我們也可以將此二個運算子覆載，用來處理自訂型別的輸入輸出，例如：

```
Matrix m(10,10);
……..
cin >> m;
cout << m;
```

我們可注意到<<與>>為 binary operators，其左邊的運算元必定為內建之 istream(cin)或 ostrean(cout)物件，而我們欲做運算子覆載的類別，例如 Matrix，必定為右邊的運算元。

基於我們先前所介紹運算子覆載的機制，以上的指令，呼叫的運算子覆載方式，可以是左運算物件之函式成員，或接受左右兩運算元之非函式成員，故可能覆載型式有：

(1) istream 或 ostream 的運算子覆載函式成員：

```
istream& operator>>(Matrix& );          cin.operator>>(m);
ostream& operator<<(Matrix& );          cout.operator<<(m);
```

由於 istream 與 ostream 均是 C++內建的型別，根據規則，我們不能覆載其運算子，故以上方法不可被接受。

(2) 自訂類別之非函式成員(朋友函式)：

```
friend istream& operator>>(istream&, Matrix& );        operator>>(cin, m);
friend ostream& operator<<(ostream&, Matrix& );        operator<<(cout, m);
```

由以上可知，<<與>>的覆載函式，必須為自訂類別之非函式成員。

程式範例: oop_ex88.cpp

注意要點:

1. 將<<與>>的覆載函式宣告為自訂類別之函式成員為錯誤的做法，因為運算子覆載函式成員必須是由左運算元呼叫，而非右運算元，對於<<與>>，我們自訂的類別之物件必定是右運算元，故您所定義於自訂類別內之覆載函式永遠不會被呼叫，系統呼叫的是 cin 或 cout 的<<或>>之內建定義，故產生錯誤。(cin>>m; 呼叫的型式為 cin.operator>>(m) 或 operator>>(cin, m); 而非 m.operator>>(cin);
2. 必須要以傳參考方式傳回所傳入之 istream 或 ostream 物件，以支援串接式的呼叫，如 cout<<m1<<m2<<m3……;。
3. cout<<m1<<m2;之執行方式：
   首先，cout<<m1 會以以下呼叫方式執行：
   operator<<(cout, m1);
   此呼叫會以輸入之 cout 的參考作為傳回值，亦即 cout<<m1 執行後變成 cout 的參考，故 cout<<m1<<m2;於 cout<<m1 執行後變成 cout<<m2;故可接續執行以下呼叫：
   operator<<(cout, m2);

8.5 Overloading ++ and -- Operators

「++」與「--」運算子，無論為前置或後置，都可以加以覆載，我們知道，前置與後置的執行行為上並不完全相同，故需分開各別覆載，且前置與後置的覆載函式的型式需不一樣，編譯器才能分辨出應使用哪一個覆載函式。

對於前置運算子的覆載，與其他一元運算子相同，例如：

++m;

會視其覆載函式為函式成員或非函式成員，做以下的呼叫之一：

m.operator++( ); 其函式原型為： Matrix& operator++( );

或 operator++(m); 其函式原型為： friend Matrix& operator++(Matrix&);

至於後置運算子的覆載，則比較特殊，C++為了與前置運算子之使用區別，呼叫時會多傳入一個 0，作為呼叫上的區別，例如：

m++;

會視其覆載函式為函式成員或非函式成員，做以下的呼叫之一：

m.operator++(0);　故其函式原型應為：　Matrix operator++(int);

或 operator++(m, 0);　故其函式原型應為：friend Matrix operator++(Matrix&, int);

0 這個輸入參數，只是使前置與後置能有不同的函式呼叫方式，以讓編譯器用來分辨所呼叫的是您所定義的哪一個覆載函式，您製作的覆載函式則須配合此呼叫規則來製作(後置多一個整數參數)，此規則同樣適用於任何同時有前置與後置用法的運算子。

程式範例: oop_ex89.cpp　oop_ex90.cpp

注意要點:

1. 前置與後置運算子覆載函式之製作須符合其應有之執行行為，故二者的製作內容應不相同。
2. 前置++之覆載較單純，其做法是將物件之值加一後，以傳參考的方式傳回物件本身。
3. 後置++之覆載為了要製作出後置遞增的效果，其技巧是將物件原有的值先拷貝給一暫時物件，之後對物件之值加一後，以傳執法傳回暫時物件之值(物件加一前之值)，而非物件本身。
4. 前置與後置覆載函式可為類別的函式成員或非函式成員。

8.6 User-Defined Conversions

對於內建的型別，我們都知道對不同型別的資料做運算時，系統必要時會做自動型別轉換。例如以下程式的執行結果應為 8.14159。

```
int a = 5;
double b = 3.14259;
cout<<a+b<<endl;
```

對於我們自訂的型別，則不具有此自動轉換的功能，例如：oop_ex91.cpp，欲使自訂的 myInt 與內建的 int 型別資料作加減運算，需提供「+」「-」運算子覆載函式。

程式範例: **oop_ex91.cpp**

注意要點:

1. 我們將 myInt 之物件與浮點數做加減運算，其結果為整數，Why?
2. 對此，有一種解決方案，是對所有可能的與浮點數之運算形式，均提供運算子覆載函式，使其正常運作，但此法非常不經濟。

C++對自訂的類別，提供了可自訂型別轉換的機制，當某類別之物件與其他型別物件間以運算子做運算時，此類別如無定義適當的運算子覆載函式，則編譯器會嘗試去呼叫其自訂轉換函式，如有轉換函式其轉換後的型別可與運算子另一邊的物件做運算，則執行之以將此類別的物件轉換為指定型態之資料，轉換之後此運算即可順利執行，不需有對型別對應之運算子覆載函式。

例如，以下程式，如 classX 沒有定義負責處理與 int 相加的「+」運算子覆載函式(int operator+(int&);，但有定義轉換成 int 或 double 等等可與 int 相加之型別的轉換函式(不一定需轉換成 int)，亦可順利執行。

```
classX a(10);
cout<< a + 10 <<endl;
```

自訂型別轉換函式的格式如下：

```
operator type( );
```

格式上的要求為：

1. 其中 type 必須為系統已知之內建或自訂型別的名稱，被呼叫執行後則需傳回 type 型別之物件，以使執行結果表現出某類別之物件已被此函式轉換成為 type 之資料。
2. 其必須為某類別之函式成員。
3. 其不可指定傳回值(type 為傳回值)，也不可以有輸入參數。

程式範例: **oop_ex92.cpp**

注意要點:

1. 為一類別提供多個自訂型別轉換函式，需避免編譯器可有多個選擇的情況，造成 ambiguous call。例如對一類別同時提供轉換為 int 與 double 之轉換函式，選擇任何一個均可以與浮點數(或整數)相加，編譯器無法決定選擇何者。
2. 對物件做強制型別轉換，亦是嘗試呼叫適當的自訂型別轉換函式，其不一定需與指定轉換之目標型別相同，只要轉換後之型別可進一步轉換至目標型別即可。
3. 自訂型別轉換函式執行次序上地位對等，故亦會與運算子覆載函式相衝突，需避免。例：將 oop_ex92.cpp 中之自定轉換函式成員，加到 oop_ex91.cpp 中，會造成 ambiguous。