附 錄 C

輸入 / 輸出格式處理器設計範例

在第 340 頁中我們簡單地介紹如何設計輸入 / 輸出格式處理器，在此我們提供幾個較完整的範例以供讀者參考。

C.1 輸出格式處理器：攝氏溫度轉華氏溫度

假設我們須要設計一個攝氏溫度轉華氏溫度的格式處理器，若其使用型式如下：

```
for ( int deg = 0 ; deg <= 100 ; deg+=10 )
    cout << deg << " C <==> " << fahrenheit << deg <<  " F\n" ;
```

deg 代表攝氏溫度，fahrenheit 為溫度轉換格式處理器。當 fahrenheit 出現在 deg 的前面時，則其後的 deg 要自動由攝氏溫度轉為華氏溫度輸出。

在設計格式轉換器之前，首先要認知所謂的「格式轉換器」也是一種隸屬於某種資料型別的物件。以此例來說，fahrenheit 為某一型別的物件，在此我們先將其所歸屬的型別稱為 Fahrenheit。為簡化問題起見，考慮以下輸出式子：

```
int deg = 30 ;
cout << fahrenheit << deg << "\n" ;
```

觀察以上的輸出式子，很明顯的，三個輸出運算子 (operator<<) 皆不相同，除最後一個使用系統所預設的輸出運算子外，前兩者因都用到了 Fahrenheit 型別，須要使用者自行設計。參考前兩個輸出運算子各別所須要的參數型別，我們暫時先將其介面設計為以下型式：

```
// 第一個輸出運算子
Fahrenheit   operator<< ( ostream& , Fahrenheit ) ;

// 第二個輸出運算子
ostream&     opreator<< ( const Fahrenheit& , int ) ;
```

這裡第一個輸出運算子的回傳型別為 Fahrenheit 物件，此物件隨即被當成第二個輸出運算子的第一個參數使用。由於第二個輸出運算子的回傳型別為 ostream 類別，

並不在運算子輸入的參數列中，因此須要想辦法在 Fahrenheit 物件內留住 ostream 的資料。一個簡單的方式即是在 Fahrenheit 內使用指標讓其指向 ostream 物件，如此 Fahrenheit 型別，fahrenheit 格式處理器與第一個輸出運算子就可以設計為：

```
class Fahrenheit {
    private :
        ostream  *ptr ;
    public :
        friend  Fahrenheit   operator<< ( ostream& , Fahrenheit ) ;
        friend  ostream&     operator<< ( const Fahrenheit& , int ) ;
} ;

// 設定 fahrenheit 格式處理器為 Fahrenheit 結構的物件
Fahrenheit   fahrenheit ;

// 第一個輸出運算子
Fahrenheit  operator<< ( ostream& out , Fahrenheit foo ) {
    foo.ptr = &out ;
    return  foo ;
}
```

由於接在格式處理器後的攝氏溫度須要以華氏溫度方式輸出，這須要在第二個輸出運算子內計算，因此可設計成以下方式：

```
// 第二個輸出運算子
ostream&     operator<< ( const Fahrenheit& foo , int deg ) {
    *(foo.ptr) << ( 9*deg/5 + 32 ) ;
    return  *(foo.ptr) ;
}
```

如此我們就可執行：

```
for ( int deg = 0 ; deg <= 100 ; deg+=10 )
    cout << setw(3) << deg << " C <==> "
         << setw(3) << fahrenheit << deg <<  " F\n" ;
```

而得到以下的輸出：

```
  0 C <==>  32 F
 10 C <==>  50 F
 20 C <==>  68 F
 ...
 90 C <==> 194 F
100 C <==> 212 F
```

同樣地，類似的方式也可以用來設計輸入格式處理器，例如：

```
int  deg ;
cin >> fahrenheit >> deg ;   // 自動將輸入的攝氏溫度轉為華氏溫度
```

這部份就留待讀者自行設計練習。

C.2 輸出格式處理器：小數位數數值截去

現行貨幣的最小單位為分，因此貨幣若以元為單位，則分就為小數點後兩位，在小數點後兩位以後的數字則通常須要將之截去或是四捨五入。在數學上，截去的處理方式通常是將此數乘上 100 後，去除小數部份，再除以 100，若寫成程式則為：

```
double  x = 0.1267 ;
cout << floor(x*100)/100 << endl ;    // 輸出 : 0.12
```

這裡使用到 cmath 標頭檔內的 floor 函式，用來去除浮點數的小數部份。讀者可察覺到這種截去數值的處理方式甚為醜陋，大有改善的空間。若改為以下方式輸出，則程式就較為乾淨清爽：

```
double  x = 0.1267 ;
cout << truncate(2) << x << endl ;     // 輸出 : 0.12
```

這裡的 truncate(n) 為尚待定義的輸出格式處理器，用來去除緊接在其後的浮點數小數點後 n 位以下的數字。

仔細分析以上的式子，我們可知，由於 truncate 輸出格式處理器為事前定義的物件，因此 truncate(n) 就不可能為一普通函式或是使用參數建立的物件，而是執行函式運算子[548]的格式處理器，相當於 truncate.operator()(n)，而輸入的參數 n 則須要存入格式處理器內，因此 truncate 所屬的型別除了如之前須有指向 ostream 物件的指標外，另須保留一整數儲存 n。若讓處理器 truncate 為 Truncate 型別的物件，依照前一範例模式，相關的程式碼可以定義為：

```
class Truncate {
    private :
        int       n ;       // 截去小數點第 n 位之後的數值
        ostream   *ptr ;

    public :
        // 定義函式運算子
        Truncate   operator() ( int s ) {
            n = s ;
            return  *this ;
        }
        friend  Truncate   operator<< ( ostream& , Truncate ) ;
        friend  ostream&   operator<< ( const Truncate& , double ) ;
} ;

// 定義 truncate 格式處理器
Truncate  truncate ;

// 覆載輸出運算子
```

```
Truncate  operator<< ( ostream& out , Truncate foo ) {
    foo.ptr = &out ;
    return  foo ;
}
```

接下來，須覆載 operator<< 來處理 Truncate 格式處理器與其後浮點數間的關係，程式如下：

```
ostream&  operator<< ( const Truncate& foo , double num ) {
    // pow(a,b) 計算 a 的 b 次方, 定義於 cmath 標頭檔
    double  x = pow(10.,foo.n) ;
    *(foo.ptr) << floor(num*x)/x ;
    return  *(foo.ptr) ;
}
```

這裡須提一點，以上截去數值的作法並未考慮到計算機本身截去誤差[18]所造成的影響，因此偶而會造成奇怪的運算結果，例如以下的程式碼：

```
for ( double x = 1 ; x <= 2 ; x += 0.2 ) {
    cout << truncate(1) << x << " " ;
}
```

執行後輸出為：

```
1 1.1 1.3 1.5 1.7 1.9
```

這是很明顯的錯誤，正確的數值應該為 1 1.2 1.4 1.6 ...。一般來說，補救的作法是讓其加上一個微小數，處理方式如下：

```
ostream&  operator<< ( const Truncate& foo , double num ) {
    // 定義一個微小數
    const double  TINY = 1./pow(10.,foo.n+7) ;
    double  x = pow(10.,foo.n) ;
    *(foo.ptr) << floor(num*x+TINY)/x ;
    return  *(foo.ptr) ;
}
```

然而這種處理方式並不是萬靈丹，仍會有一些數字會成為漏網之魚，不過機會大概會少許多。

C.3 輸入格式處理器：每行讀取指定數量的資料

假設有一資料檔，我們要設計一輸入格式處理器使得程式可以在每讀取一行後，只將此行的前 n 筆資料存入之後的向量陣列內，多餘的資料則不加以理會，若行的資料數不足 n 筆，則須全部存入陣列內。以下為我們所要的操作方式：

```
ifstream  infile1("int_data") ;    // 整數資料檔
string        line
```

```
vector<int>   a ;
while ( getline(infile1,line) ) {
    line >> readn(5) >> a ;       // 每行最多讀取 5 筆整數存入向量陣列
    ...
}

ifstream  infile2("double_data") ;
vector<double>  b ;
while ( getline(infile2,line) ) {
    line >> readn(3) >> b ;       // 每行最多讀取 3 筆浮點數存入向量陣列
    ...
}
```

這裡須留意，當資料由檔案中讀入後被存入一 C++ 字串，此字串隨即被當成輸入運算子的資料來源，而不再是傳統的 istream 物件。參考前兩個範例的設計方式，分析以上的輸入敘述部份，我們可以設計出以下對應的程式碼，

```
class  Readn {
    private :
        int  n ;            // 最多讀入的資料數量
        string  *ptr ;      // 指向資料所在的字串

    public :

        // 函式運算子 : 設定每行最多讀入的資料數量
        Readn  operator() ( int s ) {
            n = s ;
            return *this ;
        }

        // 將字串資料儲入物件內
        friend  Readn  operator>> ( string& , Readn ) ;

        // 執行讀取資料的步驟
        template <class T>
        friend  void  operator>> ( Readn , vector<T>& ) ;

} ;
```

在此輸入運算子的資料來源不再是 istream 物件，而是 C++ 字串，因此類別內須改用字串指標指向資料行。同時也因應題目的特性，特別讓第二個泛型輸出運算子不再回傳任何資料。接下來，透過輸入型字串資料串流的輔助，我們可將資料由 C++ 字串一筆一筆的讀入存入容器內，因此這兩個 Readn 的夥伴程式可被設計成：

```
Readn  operator>> ( string& str , Readn rdn ) {
    rdn.ptr = &str ;
    return  rdn ;
}
```

```
template <class T>
void  operator>> ( Readn rdn , vector<T>& foo ) {
    istringstream  istr(*(rdn.ptr)) ;
    int  i = 0 ;
    T  data ;
    while ( istr >> data ) {
        foo.push_back(data) ;
        if ( ++i == rdn.n ) break ;
    }
}
```

最後再定義輸出格式處理器即可使用，

```
Readn  readn ;
```

C.4 輸入 / 輸出格式處理器：列舉型別資料的輸入 / 輸出

C++ 所設計的列舉資料型別[??]可以讓使用者以文字的方式來代表整數資料，例如：

```
enum  Season  { spring=1 , summer , fall , winter } ;
enum  Weekday { Sun , Mon , Tue , Wed , Thu , Fri , Sat } ;
```

以上的 spring 代表整數 1，summer 代表整數 2，其餘依次遞增。若起始的文字不設定整數值，則以 0 為起始值，因此 Sun 代表整數 0， Mon 為 1，...。當程式碼使用文字來代替數字資料，會使得程式碼變得較容易理解而有意義：

```
Season  foo ;
foo = spring ;     // foo 值設定為 spring
```

但很不幸的，列舉型別的方便性到此為止，它並沒有進一步延伸到列舉型別變數的輸入 / 輸出上，例如：

```
Weekday  foo ;
cin >> foo ;            // 錯誤：無法如此使用
cout << foo << endl ;   // 正確：但輸出 foo 所對應的整數數值
```

如果希望程式在執行時，可直接以輸入文字方式存入以上的列舉型別變數 foo，例如：Sun，或者是在輸出時，可以列印出 foo 所代表的文字，而非其對應的數字，則我們只有自求多福，C++ 並不提供這些功能。為達到此目的，我們可以針對所使用的列舉型別設計一輸入 / 輸出格式處理器，若以 Weekday 列舉型別為例，我們可以為其專門設計一個格式處理器，其使用方式如下：

```
Weekday  foo ;
cin >> alphaweekday >> foo ;   // 輸入 : Fri
cout << alphaweekday << foo ;  // 輸出 : Fri
```

以上 alphaweekday 為格式處理器，其作用就是在輸入時，可以將文字資料經過轉換存入之後的列舉型別變數。若用在輸出，則可以列印其後的列舉型別變數所對應的文字。這裡的 alphaweekday 格式處理器的用法與之前的範例類似，因此在設計上也是大同小異，只不過現在我們須在格式處理類別內定義兩個指標，根據須要分別指向負責輸入或輸出的物件。以下為針對 Weekday 列舉型別的輸入 / 輸出所設計的格式處理類別，即 Alpha_Weekday：

```
// 定義 Weekday 列舉型別
enum  Weekday { Sun , Mon , Tue , Wed , Thu , Fri , Sat };

// 針對 Weekday 的輸入/輸出格式處理類別
class  Alpha_Weekday {
  private :

    static const string  weekday[7] ;  // 儲存列舉型別對應的字串
    istream  *inptr ;                  // 指標指向 istream 物件
    ostream  *outptr ;                 // 指標指向 ostream 物件

  public :

    // 處理資料的輸入
    friend Alpha_Weekday operator>> ( istream& , Alpha_Weekday ) ;
    friend istream&      operator>> ( Alpha_Weekday , Weekday& ) ;

    // 處理資料的輸出
    friend Alpha_Weekday operator<< ( ostream& , Alpha_Weekday ) ;
    friend ostream&      operator<< ( Alpha_Weekday , Weekday ) ;

} ;

const string Alpha_Weekday::weekday[7] = { "Sun" , "Mon" , "Tue" ,
                                           "Wed" , "Thu" , "Fri" ,
                                           "Sat" } ;

// 定義格式處理器
Alpha_Weekday  alphaweekday ;
```

以下為整個 Alpha_Weekday 格式處理類別的程式碼與其執行結果，

Weekday 輸入 / 輸出格式處理器　　程式：weekday.cc

```
01   #include <iostream>
02   #include <string>
03
04   using namespace std ;
05
06   // 定義 Weekday 列舉型別
07   enum  Weekday { Sun , Mon , Tue , Wed , Thu , Fri , Sat } ;
08
```

```
09  // 格式處理類別
10  class  Alpha_Weekday {
11    private :
12
13      static const string  weekday[7] ;   // 儲存列舉型別對應的字串
14      istream  *inptr ;                   // 指標指向 istream 物件
15      ostream  *outptr ;                  // 指標指向 ostream 物件
16
17    public :
18
19      // 處理資料的輸入
20      friend Alpha_Weekday operator>> ( istream& , Alpha_Weekday ) ;
21      friend istream&    operator>> ( Alpha_Weekday , Weekday& ) ;
22
23      // 處理資料的輸出
24      friend Alpha_Weekday operator<< ( ostream& , Alpha_Weekday ) ;
25      friend ostream&    operator<< ( Alpha_Weekday , Weekday ) ;
26
27  } ;
28
29  const string Alpha_Weekday::weekday[7] = { "Sun" , "Mon" , "Tue" ,
30                                             "Wed" , "Thu" , "Fri" ,
31                                             "Sat" } ;
32
33  Alpha_Weekday  operator >> ( istream& in , Alpha_Weekday foo ) {
34      foo.inptr = &in ;
35      return  foo ;
36  }
37
38  istream&  operator >> ( Alpha_Weekday foo , Weekday& weekday ) {
39
40      string  str ;
41      *(foo.inptr) >> str ;
42
43      for ( int i = 0 ; i < 7 ; ++i ) {
44          if ( str == foo.weekday[i] ) {
45              weekday = static_cast<Weekday>(i) ;
46              break ;
47          }
48      }
49
50      return  *(foo.inptr) ;
51  }
52
53  Alpha_Weekday  operator << ( ostream& out , Alpha_Weekday foo ) {
54      foo.outptr = &out ;
55      return  foo ;
56  }
57
58  ostream&  operator<< ( Alpha_Weekday foo , Weekday  weekday ) {
59      *(foo.outptr) << foo.weekday[static_cast<int>(weekday)] ;
60      return  *(foo.outptr) ;
61  }
62
63  // 定義格式處理器
64  Alpha_Weekday  alphaweekday ;
65
66  int main() {
```

```
67
68        Weekday  weekday ;
69
70        while ( 1 ) {
71            cout << "輸入：" ;
72            cin >> alphaweekday >> weekday ;
73            cout << "輸出：" << weekday << endl ;
74            cout << "輸出：" << alphaweekday << weekday << endl ;
75        }
76
77        return 0 ;
78
79    }
80
```

執行結果

```
01    輸入：Fri
02    輸出：5
03    輸出：Fri
04    輸入：Mon
05    輸出：1
06    輸出：Mon
07
```

結語

由以上的幾個範例，讀者應可觀察到，格式處理器的設計也只是運算子覆載的一種型式的應用。須留意，這裡所設計的格式處理器後皆須緊接著所要處理的資料，但如果在資料前另有其它型式的格式處理器，例如：

```
cout << fahrenheit  << setw(4) << deg << endl ;
cout << truncate(2) << fixed   << 12.237 << endl ;
```

等等型式，則還須分別為其設計相關類型的輸出運算子，這都只是運算子覆載的應用而矣，只不過稍微煩瑣一些。但不管如何，使用自行設計的格式處理器可以大大地增加程式輸入 / 輸出的自由度，且較能設計出符合程式須求的輸入 / 輸出功能。

練習題

以下題目須以樣板函式[461]方式設計輸出運算子，讀者若不熟悉，可先針對個別資料型別獨立設計。

1. 設計一輸出格式處理器 center 使得其後的資料可以靠中對齊，例如：

```
// 以 10 格空間列印數值 234 , 但 234 須靠中對齊
cout << center(10) << 234 << endl ;

// 以 20 格空間列印 abcd 字串 , 但字串須靠中列印
cout << center(20) << "abcd" << endl ;
```

2. 撰寫一輸出格式處理類別 Split，若 split 格式處理器為其物件，則以下的程式碼可以將緊接其後的資料以指定的格子數分開列印，此輸出格式處理器也可讓使用者自行指定隔離字元，處理器預設的隔離字元為空白字元，例如：

```
double  a = 3.14159 ;
int     b = 2000 ;
char   *c = "math" ;
string  d = "hello" ;

cout << split(1)     << a << endl ;       // 輸出 : 3 . 1 4 1 5 9
cout << split(3,'=') << b << endl ;       // 輸出 : 2===0===0===0
cout << split(3,'-') << c << endl ;       // 輸出 : m---a---t---h
cout << split(2,'*') << d << endl ;       // 輸出 : h**e**l**l**o
```

3. 請針對序列容器設計輸出格式處理器 splita，使得資料之間的輸出以指定的字串分開，末筆資料之後不須加入隔離字串。此外若不指定隔離字串，則自動以一個空格分開，使用方式如下：

```
int    no[5] = { 3 , 2 , 1 , 4 , 0 } ;

list<int>     a(no,no+4) ;
// 輸出 : 3 2 1 4
cout << splita() << a << endl ;

// 輸出 : 3 -- 2 -- 1 -- 4
cout << splita(" -- ") << a << endl ;


string       str[3] = { "數學" , "物理" , "化學" } ;
vector<string>  b(str,str+3) ;

// 輸出 : 數學 <==> 物理 <==> 化學
cout << splita(" <==> ") << b << endl ;
```

4. 許多程式設計員若用慣了 C 程式語言的輸出格式，可能對其輸出格式的簡潔性有很深刻的印象，例如：

```
// 以十格靠右列印 123，左側多出來的空格補 '0'
printf("%010d",123) ;

// 以十格靠左列印 123
```

```
printf("%-10d",123) ;
```

同樣效果的輸出，若改成使用 C++ 的輸出格式處理器，則變成：

```
cout << setfill('0') << setw(10) << 123 << setfill(' ') ;
cout << left << setw(10) << 123 << right ;
```

這裡需留意，C++ 在使用過 `setfill`[338] 或者是 `left`[338] 後，都要記得改回預設值，否則其作用會持續影響到之後的輸出。由此讀者可察覺，以使用的方便性而言， C++ 實在是不如 C 語言。不過即使如此，我們倒是可以設計一個擁有 C 程式語言輸出風格的 C++ 輸出格式處理器，例如：

```
cout << cint("%010d") << 123 ;    // 同 print(%010d",123)
cout << cint("%-10d") << 123 ;    // 同 print(%-10d",123)
```

請針對整數的輸出，設計一個帶有 C 風格的簡易型輸出格式處理器，由此例也可以看出自行設計 C++ 格式處理器的實用性。