如何學寫程式

子　由

2007-03-30

一：初學者的學習困境

對許多程式初學者而言，撰寫程式是很痛苦的一件事。初學者往往在面對著一個簡單的程式題目時，若直接使用一般的筆紙，只要耐心地花費一些時間仔細計算，程式所要顯示的答案總是可以算出來。但若是要改用程式碼來控制計算機依照心裡想要的方式執行，則往往就不是那麼一回事。似乎腦筋所想的步驟無法那麼自然地以程式碼的方式表現出來，兩者之間總有一段落差，無法接連在一起。初學者總是在每一次程式執行之前充滿著希望，然而很不幸地，希望總是在按下 Enter 鍵後瞬間破滅，接下來所要面對的就是無窮無盡的除蟲步驟 (debugging)，眼睛不斷地緊盯著螢幕視窗畫面，一列一列的思考，想盡辦法找出可能錯誤的程式碼，加以改寫程式，期許程式在重新編譯後執行，正確的答案就會顯示在螢幕之上。

二：傳統程設建議的盲點

對一個想學好程式設計的初學者而言，傳統上的建議就是練習，模仿，再練習。這種學習方式就是希望透過不斷地練習已有的程式，模仿他人的程式，與練習新的程式題目，以達到會自行撰寫程式的能力。這樣的學習過程如同禪宗的頓悟一般，初學者須要在不斷地挫折中，仍然稟持著終究會成功的信念，最後才能有成。然而很不幸地，實際上多數初學者的信心往往在學校的第一次考試之後就完全消失，淪為程式設計界的失敗者，自此消聲匿跡，不再學習程式。仔細探討其原因，可以察覺到傳統的建議，雖然大原則是正確的，但並沒有提供一套循序漸進的有效學習方法，僅是要求初學者透過不斷地練習來增加程式設計的經驗，而程式設計的方法則須由初學者自行建構揣模，如此適應不良的學生自然就會被淘汰出局，這種沒有一套循序漸進的學習方式來引領初學者進入程式設計的世界，自然是很殘酷，正是以下所要解決的。

三：迴圈與邏輯

在正式介紹學習方法之前，首先須要了解兩個程式設計的基本用法。對初學者而言，初期程式設計的主要障礙是在不知如何將應用問題轉以程式的迴圈與邏輯式子代表。當一個初學者學會了如何在程式中活用迴圈與邏輯式子，則就不知不覺地踏進程式設計的大門，其他未學到的程式語法只是飯後點心而矣，通常難不倒他。

一般來說，程式中的迴圈即代表著**重複執行**，大多數的程式通常會用迴圈來重複執行某些式子，這也是電腦最擅長的部份。至於邏輯式子則是代表**執行的分叉**，表示程式將由一堆式子中選擇其中之一執行。大部份的程式設計都是由這兩種基本程式用法以不同的方式組合而成。

迴圈在 C++ 程式語言中共有 3 種，分別為 `for`，`while` 與 `do {...} while`。以使用度來說，第一種大概佔了九成。前兩種功能大致相等，例如：若 `A`， `B`，`C` 與 `D` 代表一些敘述，其中 `B` 為邏輯判斷式，則

```
for ( A ; B ; D ) {
    C ;
}
```

相當於，

```
A ;
while ( B ) {
    C ;
    D ;
}
```

這兩種迴圈可能因為初始的 `B` 邏輯判斷敘述為假，使得程式直接跳離迴圈，造成 `D` 可能一次都未被執行到。但這種現象並不會發生在最後一種 `do { ...} while` 迴圈上，例如：

```
do {
  A ;
} while ( B ) ;
```

由於邏輯判斷式 `B` 在 `A` 式子之後，因此 `A` 至少會被執行一次才可能跳離迴圈。此外，若程式須在迴圈的敘述中提早跳離迴圈，則可以使用 `break` 敘述。

至於 C++ 的邏輯式子主要是由 `if` 與其衍生的 `if ... else ...` 等等組成，例如：

```
if ( A ) B ;
```

```
if ( A )
    B1 ;
else
    B2 ;
```

```
if ( A1 )
    B1 ;
else if ( A2 )
    B2 ;
else
    B3 ;
```

其他尚有較少用的 `switch` 與 `()?():()` 運算式。後者的用法與單純的 `if ... else ...` 相當，例如：

```
if ( A )
    B1 ;
else
    B2 ;
```

```
( A ) ? ( B1 ) : ( B2 ) ;
```

B1 與 B2 最好為單一敘述。()?():() 最常與輸出式子一起使用，例如以下是用來輸出 i 的數值，但若 i 小於 60 則輸出 60：

```
cout << ( i < 60 ? 60 : i ) << endl ;
```

以上所用的 C++ 語法都是最基本的內容，這在所有的程設書本都有詳細的說明。

四：建構式程式設計

所謂的建構式程式設計即是一種以最簡單的式子型式，一步一步的將程式碼建構出來。在這種建構式程式中，除非顯而易見，否則並不用迴圈與邏輯式子。舉例來說，以下的式子是用來計算 1 到 10 的奇數和。

```
int sum = 0 ;
sum = sum + 1 ;   sum = sum + 3 ;   sum = sum + 5 ;
sum = sum + 7 ;   sum = sum + 9 ;
cout << sum << endl ;
```

若要列印 1 到 9 之間的數字，且在每 3 數字之後跳一行從頭列印，則可寫成

```
cout << 1 << " " ;  cout << 2 << " " ;  cout << 3 << " " ;
cout << endl ;
cout << 4 << " " ;  cout << 5 << " " ;  cout << 6 << " " ;
cout << endl ;
cout << 7 << " " ;  cout << 8 << " " ;  cout << 9 << " " ;
cout << endl ;
```

列印一座山

```
cout << "*" << endl ;
cout << "*" << "*" << "*" << endl ;
cout << "*" << "*" << "*" << "*" << "*" << endl ;
```

初學者最好在計算紙上撰寫建構式程式，不必講究文字對齊，且文字間不要太擠，以免引起錯看。在撰寫建構式程式時，只要輸出的資料可以分辨，可以暫時不管輸出資料間的對齊，以避免增加程式的複雜度。 同時如果在撰寫過程中，若已經可以看出程式敘述間的規律性，則可使用 ... 藉以節省時間。

這樣的程式大概沒有人不會，只要有耐心，每個人都可以將程式碼寫出來。但這不像程式設計，程式設計須要找出程式碼的規則性，想辦法將其用迴圈的方式改寫，使得程式較為簡潔。

五：由建構式程式出發

建構式的程式雖然簡單，卻不能登大雅之堂，這樣的程式無法稱為程式設計。但建構式程式倒是為初學者提供了一個程式草圖，初學者可根據此程式草圖，分析敘述之間

的相關性，將一些資料以適當的變數取代，找出變數的變化方式，以下提供幾個例子加以說明。

■ 九九乘法表

這大概是每個初學者都會寫到的程式，也是經常被問的程式題目，我們將以建構式方法，將此程式碼寫出來。

建構式九九乘法表

```
// 第一列 : 1x1=1 1x2=2 1x3=3 ... 9x1=9
cout << 1 << "x" << 1 << "=" << 1 << "  " ;
cout << 2 << "x" << 1 << "=" << 2 << "  " ;
...
cout << 9 << "x" << 1 << "=" << 9 << "  " ;
cout << "\n" ;

// 第二列 : 1x2=2 2x2=4 3x2=6 ... 9x2=18
cout << 1 << "x" << 2 << "=" << 2  << "  " ;
cout << 2 << "x" << 2 << "=" << 4  << "  " ;
...
cout << 9 << "x" << 2 << "=" << 18 << "  " ;
cout << "\n" ;

...

// 第九列 : 1x9=9 2x9=18 3x9=27 ... 9x9=81
cout << 1 << "x" << 9 << "=" << 9  << "  " ;
cout << 2 << "x" << 9 << "=" << 18 << "  " ;
...
cout << 9 << "x" << 9 << "=" << 81 << "  " ;
cout << "\n" ;
```

以上的程式由於輸出有九個換行字元，因此共輸出九列 (row)。在每一列中，分別有九個計算式。有了此基本建構程式碼後，接下來須將之轉成較像一般C++ 程式的程式碼。

分析建構程式

在分析建構式程式碼中，首先要做的是設定適當的變數來替代，變數的個數要越少越好，以此為例，緊接在每個 `cout` 後的整數似乎有規則性，且都是由 1 到 9 依次變化，因此可用定義一個整數變數 `i` 來暫代，其值都是在每一個輸出列 (row) 中由 1 到 9 方式遞增，

此外在輸出式等號之前的整數，也是由 1 變化到 9，不過是以每一列增加 1 的方式改變，也可以用一個整數變數 `j` 取代， `j` 是以列的方式由 1 到 9 遞增。在每個 `j`

中，i 會由 1 遞增到 9。同時等號後的數字就只是 i 與 j 的乘積。因此若由變數 i 的角度寫成迴圈，輸出的式子可寫成

```
for ( i = 1 ; i <= 9 ; ++i ) {
    cout << j << "x" << i << "=" << i*j << "  " ;
}
cout << "\n" ;
```

由於變數 j 遞增的速度較 i 為慢，因此 j 的迴圈須擺在 i 迴圈的外層，且為 1 遞增到 9，所以可寫成

```
for ( j = 1 ; j <= 9 ; ++j ) {
    for ( i = 1 ; i <= 9 ; ++i ) {
        cout << j << "x" << i << "=" << i*j << "  " ;
    }
    cout << "\n" ;
}
```

如此就完成了九九乘法表主要的程式碼，剩下的工作就只是將一些瑣碎的敘述補齊，例如：加入標頭檔，定義變數等。

```
#inclue <iostream>

using namespace std ;

int main() {

    int i , j ;
    for ( j = 1 ; j <= 9 ; ++j ) {
        for ( i = 1 ; i <= 9 ; ++i ) {
            cout << j << "x" << i << "=" << i*j << "  " ;
        }
        cout << "\n" ;
    }
    return 0 ;
}
```

■ 質數問題

檢查某正整數是否為質數也是初學者經常會遇到的問題，以數學的觀點來說，所謂的質數是指大於 1 的正整數，且此數僅能被 1 與本身整除。如此若要檢查某個整數 n 是否為質數，若使用筆紙，最老實的方式就是讓 n 由整數 2 開始相除，若除不盡，則與整數 3 相除，若再除不盡，則再進一位，如此持續下去，直到 n-1 為止。在相除的步驟當中，若兩數可以整除，則 n 就不是質數，不必再繼續除下去。如果所有的數都不能整除，則 n 就是質數。以下我們試著利用建構式方法來撰寫程式。

建構式質數檢查

根據質數的定義，可將程式寫成以下的式子

```
int  n ;
cin >> n ;

if ( n == 2 )         { cout << n << " 是質數\n" ; return 0 ; }
if ( n % 2 == 0 )     { cout << n << " 不是質數\n" ; return 0 ; }
if ( n % 3 == 0 )     { cout << n << " 不是質數\n" ; return 0 ; }
...
if ( n % (n-1) == 0 ) { cout << n << " 不是質數\n" ; return 0 ; }

cout << n << " 是質數\n" ;
```

以上的 return 0 代表著程式將跳離主函式 main 結束執行。

分析建構程式

在以上的程式中，除了第一個 if 條件式外，其它的條件式的除數有規則性，即由 2 開始一一遞增到 n-1，所以自然可以使用一個變數來替代，若定義其為 i，則從第二個條件式開始的條件式就可通通置放在一個迴圈之內，程式可以改寫成，

```
for ( int i = 2 ; i <= n-1 ; ++i ) {
    if ( n % i == 0 ) {
        cout << n << " 不是質數\n" ;
        return 0 ;
    }
}
```

加上頭尾的敘述，整個程式就變成，

```
#include <iostream>

using namespace std ;

int main() {
    int n ;
    cin >> n ;

    if ( n == 2 ) {  cout << n << " 是質數\n" ;  return 0 ;   }

    for ( int i = 2 ; i <= n-1 ; ++i ) {
        if ( n % i == 0 ) {
            cout << n << " 不是質數\n" ;
            return 0 ;
        }
    }
    cout << n << " 是質數\n" ;
```

```
    return 0 ;
}
```

以上的程式碼在執行上雖然是正確的，但卻沒有效率。例如：在數學上，要判斷 n 是否為質數，並不須要由 2 開始依次除到 n-1。事實上，僅要除到小於等於 $\sqrt{n}$ 的整數即可，此外除了數字 2 以外，並不須要一再的除以其它的偶數，然而這些細節在此將予以忽略。

■ 列印數字圖形

這個範例是要寫一個程式印出以下的數字圖形，數字是由 1 開始，由上而下，由左而右的方式遞增。且數字是列印成下三角形 (lower triangle) 樣式。本程式的困難處是在數字增加方式與在程式語言中，由左而右，由上而下的列印方式剛好相反：

```
列數 : 4                    列數 : 5

1                           1
2    5                      2    6
3    6    8                 3    7    10
4    7    9    10           4    8    11    12
                            5    9    12    14    15
```

建構式程式

假設程式須印出的列數為 n，可由輸入決定。根據以上的圖形，可以將建構式程式碼編寫成以下方式：

```
cin >> n ;  // 總列數

cout << 1 ;                              // 第 1 列
cout << "\n" ;

cout << 2     << " "                     // 第 2 列
cout << (n+1) << " " ;
cout << "\n" ;

...

cout << n           << " " ;             // 第 n 列
cout << (n+1)+(n-2) << " " ;
...
cout << (n+1)+(n-1)+(n-3) + ... + 1 << " " ;
cout << "\n" ;
```

以上真正要輸出的數字須要仔細地用數學方式推導，因此在此可先寫少數幾個數字以

做為代表。

分析建構程式

很明顯的，以上每一輸出列 (row) 的列數可以使用一個變數 r 來替代，r 是由 1 遞增到 n，每一輸出列上，輸出的數字總量與列數相同，也就是第一列，印一筆數字，第二列印兩筆數字， ...， 第 r 列印 r 筆數字，在每一列中，共要印出 r 個數字。在程式設計上，就可使用迴圈來處理。整理以上的說明，程式就可改寫成：

```
cin >> n ;
for ( r = 1 ; r <= n ; ++r ) {
    for ( c = 1 ; c <= r ; ++c ) {
        f = ... ;                    // f 表要輸出的變數，
                                     // 須要另外導出數值
        cout << f << " " ;
    }
    cout << "\n" ;
}
```

以上的變數 f 會因 r 與 c 的不同而有所差異，若用數學的方式表示，則可寫成 f = f(r,c) 代表在第 r 列第 c 行中要輸出的數字，數字的大小只與 r 與 c 有關，且 r 永遠大於等於 c。 接下來只要用數學的方法導出 f(r,c) 函數，寫成程式碼，如此程式就完成了。

由於數字是由上而下，由左而右，若導出每一行 (column) 的第一個數字大小，則同一行的其它數字就能很快地計算出來。也就是：

$$\mathtt{f}(\mathtt{r},\mathtt{c}) = \mathtt{f}(\mathtt{c},\mathtt{c}) + (\mathtt{r}-\mathtt{c})$$

很明顯地，數字 f(c,c) 相當於第 c-1 行之前所有數字的總數再加上 1。由於第一行有 n 個數字，第二行有 n-1 個數字，第三行有 n-2 個數字，因此第 c-1 行共有 $\mathtt{n}+(\mathtt{n}-1)+\ldots+(\mathtt{n}-(\mathtt{c}-2))$ 個數字，寫成數學模式則為 $\sum_{\mathtt{k}=\mathtt{n}-(\mathtt{c}-2)}^{\mathtt{n}} \mathtt{k}$，因此

$$\begin{aligned}
\mathtt{f}(\mathtt{c},\mathtt{c}) &= \sum_{\mathtt{k}=\mathtt{n}-(\mathtt{c}-2)}^{\mathtt{n}} \mathtt{k} \;+\; 1 \\
&= \sum_{\mathtt{k}=1}^{\mathtt{n}} \mathtt{k} \;-\; \sum_{\mathtt{k}=1}^{\mathtt{n}-(\mathtt{c}-2)-1} \mathtt{k} \;+\; 1 \\
&= \frac{\mathtt{n}(\mathtt{n}+1)}{2} - \frac{(\mathtt{n}-\mathtt{c}+1)(\mathtt{n}-\mathtt{c}+2)}{2} \;+\; 1
\end{aligned}$$

前兩個數學式子結合在一起，

$$\mathtt{f}(\mathtt{r},\mathtt{c}) \;=\; \frac{\mathtt{n}(\mathtt{n}+1)}{2} - \frac{(\mathtt{n}-\mathtt{c}+1)(\mathtt{n}-\mathtt{c}+2)}{2} + (\mathtt{r}-\mathtt{c}) + 1$$

將以上導出 f(r, c) 的公式加入程式碼中，再調整程式的輸出格式，最後的程式碼就可以寫成以下樣式：

```
#include <iostream>
#include <iomanip>

using namespace std ;

int main() {

    int  r , c , n , f ;

    cout << "> " ;
    cin >> n ;
    cout << "\n\n" ;

    for ( r = 1 ; r <= n ; ++r ) {
        for ( c = 1 ; c <= r ; ++c ) {
            f = n*(n+1)/2 - (n-c+1)*(n-c+2)/2 + (r-c) + 1 ;
            cout << setw(5) << f << "   " ;
        }
        cout << "\n" ;
    }

    return 0 ;
}
```

由以上程式的推導方式可以得知，對某些程式問題，程式設計員本身的數學能力有時會扮演相當大的角色。

■ 估算 e^x 函數

在微積分中，e^x 函數可以使用多項式來估算，其公式若以泰勒展開方法表示則為

$$e^x \approx 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!}$$

當然 n 越大，計算值就越逼近真正的函數值，這裡我們要寫個程式來估算 e^x 函數。

建構式程式

以上公式包含了許多次加法，在建構程式中我們將其分解成一項一項的加法，由於每項的分子與分母都有相關性，因此將其分別計算，如此建構式程式碼就可寫成：

```
cin >> x ;

num = 1. ;      // 各項的分子
den = 1. ;      // 各項的分母
```

```
exp = 1.              ;       // 第一項

num = num * x ;
den = den * 1 ;
exp = exp + num / den ;       // 前二項之和

num = num * x ;
den = den * 2 ;
exp = exp + num / den ;       // 前三項之和

...

num = num * x ;
den = den * n ;
exp = exp + num / den ;       // 前 n+1 項之和
```

與之前三題的程式碼不同之處，這裡的建構程式碼須要使用暫時的變數來儲存中間的計算值，若非如此，則建構程式將更為複雜。

分析建構程式

由以上的程式碼中可知，自第二項起，所有計算的步驟都相當類似，其中唯一的差別是計算分母的部份，其數字是由 1 遞增到 n，因此可以將其寫入一個簡單的迴圈中，

```
for ( int i = 1 ; i <= n ; ++i ) {
    num = num * x ;
    den = den * i ;
    exp = exp + num / den ;
}
```

加上前後的式子，整個程式就可寫成

```
#include <iostream>

using namespace std ;

int main() {

    int  n = 10 ;
    double  x , den , num , exp ;
    cin >> x ;

    den = num = exp = 1. ;

    for ( int i = 1 ; i <= n ; ++i ) {
        num = num * x ;
        den = den * i ;
        exp = exp + num / den ;
```

```
    }
    cout << "exp(" << x << ") = " << exp << endl ;

    return 0 ;
}
```

在程式中，我們直接設定 n 為整數 10。初學者須注意一點，如果將程式中所有的變數都定義成整數型別，則計算的結果將與真正的數值產生相當大的差異，主要原因在於同型別之間的運算結果也是相同型別，因此 3 除以 4 的結果就為 0，並非 0.75，若干個計算上的小誤差結合在一起就成為大誤差，因此不可輕忽資料型別的計算差異。

■ 估算 π

π 也可以使用級數方式來估算，最簡單的方式是使用 arctan(x) 的泰勒展開式：

$$\arctan(x) \approx x - \frac{x^3}{3} + \frac{x^5}{5} + \cdots + (-1)^n \frac{x^{2n+1}}{2n+1}$$

由於 $\arctan(1) = \frac{\pi}{4}$，因此 $\pi = 4\arctan(1)$。當然 n 越大，計算值就越逼近真正的函數值，雖然此公式的逼近速度很慢，不過這並不是這裡的重點，這裡我們要寫個程式來估算 π 的值。

建構式程式

觀察 arctan(x) 的泰勒展開式，可以推知 arctan(x) 的計算方式基本上就是一項一項的輪流作加減的動作，仿照前例，可將其步驟寫成以下的程式碼：

```
x = 1 ;
atan = 0 ;

// i = 1
num  = x ;
den  = 1 ;
atan = atan + num/den ;

// i = 2
num  = num * x * x ;            // num = x^3
den  = 3 ;
atan = atan - num/den ;

// i = 3
num  = num * x * x ;            // num = x^5
den  = 5 ;
atan = atan + num/den ;

...
```

```
// i = n
num  = num * x * x ;              // num = x^(2n-1)
den  = 2*n-1 ;
atan = atan - num/den             // 如果 i 為偶數
atan = atan + num/den             // 如果 i 為奇數
```

最後兩個式子須根據 i 的奇偶數擇一執行。

分析建構程式

分析以上重複的式子，除了初始的分子部份與眾不同外，其它都有相似性，因此寫成程式則為：

```
for ( i = 1 ; i <= n ; ++i ) {

    if ( i == 1 )
        num = x ;
    else
        num = num * x * x ;

    den = 2*i-1 ;

    if ( i % 2 == 1 )
        atan = atan + num/den ;
    else
        atan = atan - num/den ;

}
```

以上兩個 if 的條件式可分別簡化為：

```
num = ( i == 1 ? x : num * x * x ) ;
atan = atan + ( i % 2 == 1 ? num/den : -num/den ) ;
```

由於 i % 2 == 1 與 i % 2 在邏輯式內的作用相當，同時 atan = atan + ... 也可再簡化為 atan += ...，因此上式可再進一步簡化為：

```
atan += ( i % 2  ? num/den : -num/den ) ;
```

加上前後必須的式子，整個程式就可寫成：

```
#include <iostream>

using namespace std ;

int main() {

    double  pi , atan , num , den , x ;
```

```
    x = 1 ;
    atan = 0 ;

    for ( int i = 1 ; i <= 1000000 ; ++i ) {
        num = ( i == 1 ? x : num * x * x ) ;
        den = 2*i-1 ;
        atan += ( i % 2 ? num/den : -num/den ) ;
    }

    pi = 4 * atan ;
    cout << "PI = " << pi << endl ;

    return 0 ;

}
```

這個級數是以很緩慢的方式收斂到 π，在數學分析上還有其它級數可以以較快的速度收斂的 π，有興趣的讀者可以上網找尋相關資料。

六：結語

初學者學習建構式程式設計主要的目的即是試圖讓藏在腦筋內的想法，首先透過簡單不修飾的程式碼呈現出來，接下來再分析程式碼，設定適當的變數，找出程式碼中的規則性，利用迴圈與邏輯式子來替代，以達到簡化程式的目的。

建構式程式設計的使用對像主要是針對初學程式設計的人士，初學者在利用建構程式之前，必須將程式語言中一些最基礎的邏輯式子與迴圈的用法弄清楚，否則仍然會動彈不得。當初學者可以在程式中自由地運用迴圈與邏輯式子後，就可以放下建構式程式設計，自行發展適合本身思考習慣的程式設計技巧，正式走入有趣的程式設計世界。