國立屏東大學 資訊工程學系 C++程式設計入門教材

6. 輸入與輸出

幾乎所有的程式都必須取得外部的資料，並將運算後的結果輸出到外部，因此如何取得來自外部的輸入與輸出資料到外界，是程式設計非常重要的課題之一。C++語言使用「串流(Stream)」的概念，來做為程式與外部的溝通管道，例如我們在第3章所介紹的BMI(Body Mass Index，身體質量指數)計算程式，先透過cin輸入串流(Input Stream)來取得使用者從鍵盤輸入的身高與體重資訊，完成BMI數值的計算後，再使用對應到終端機的cout輸出串流(Output Stream)將結果加以輸出。本章將為讀者進一步說明「串流」的概念與C++語言所提供的四個標準串流：cin、cout、cerr與clog，其中我們將特別針對最常使用的cin與cout提供包含輸入/輸出格式設計、不同資料型態與數字系統的資料輸入與輸出問題等。最後，章末還要為讀者介紹兩個自早期Unix系統承襲至今的I/O重導向與Pipe管線功能。

6.1 串流(Streams)

在開始說明前，筆者要先指出的是：Stream一詞通常譯做「流」，但筆者偏好將其譯做「串流」，其原因將在本章後續小節裡為讀者們說明。

從上一世紀70年代的Unix系統開始，一直到現代的各式作業系統，當程式在執行時，系統都會為其建立三個與外界連接的渠道：

- stdin：標準輸入(Standard Input)：預設連接程式與鍵盤，讓程式可以透過stdin取得來自鍵盤的資料輸入。
- stdout：標準輸出(Standard Output)：預設連接程式與終端機，讓程式可以透過stdout將資料輸出到終端機。
- stderr：標準錯誤(Standard Error)：預設連接程式與終端機，讓程式可以透過stderr將錯誤訊息輸出到終端機。

由於有了這三個標準的渠道，程式設計師不需要知道如何與外界(例如電腦鍵盤與終端機)溝通，只要使用stdin就可以取得使用者所輸入的資料，並透過stdout與stderr就可以輸出資料或錯誤訊息給使用者知悉。C++語言進一步使用了「串流(Stream)」的概念，來統整輸入與輸出的相關操作。所謂的串流是指程式與外界(包含鍵盤、螢幕等外部裝置，以及其它的檔案)進行資料傳遞的方式。我們可以把串流想像成一種特別的「水管」，如果將這種水管架設在程式與鍵盤之間，使用者就可以透過鍵盤將所輸入的資料流動到程式內；同樣地，架設在程式與終端機間的「水管」，就可以讓程式將資料流動到螢幕終端機上加以顯示。依據資料流動的方向，還可以將「串流」再進一步區分為「輸入串流(Input Stream)」與「輸出串流(Output Stream)」：

- 輸入串流是可以將資料流動到程式內部的水管 — 在此情況下，程式是資料流動的「目的地(Destination)」。
- 輸出串流則是可以讓資料從程式中流動出去的水管 — 換言之，程式是資料流動的「來源地(Source)」。

C++語言在iostream這個標頭檔(此命名是取自Input Output Stream之意，也就是輸入/輸出串流的意思)裡，分別定義了輸入串流與輸出串流的“ 型態 ”：

- istream：代表Input Stream的“ 型態 ”

- ostrem代表Output Stream的“型態”

要知道型態本身並不能在程式中直接使用，我們必須先宣告變數才能夠拿來使用，就好比我們不會直接使用int型態，而是在程式中先宣告int型態的變數，例如先宣告 int x; 然後才能使用變數x來進行資料的操作。因此，在iostream裡，還使用了這兩個“型態”分別宣告了兩個“變數”：

```
istream cin;
ostream cout;
```

沒錯，此處的cin與cout就是在C++語言裡最常使用的字元輸入與輸出串流，本書已經在好幾個C++範例程式中，示範過如何使用cin與cout來取得使用者的輸入以及將資料輸出到終端機。由於在iostream裡包含有這些“型態”定義與“變數”宣告，所以大部份需要使用到輸入與輸出的程式，都必須在程式裡使用 #include <iostream>來將此標頭檔載入，否則將無法使用到這些輸入/輸出串流。

現在，讓我們複習一下目前所學到的知識istream與ostream是輸入與輸出串流的“型態”cin與cout則是這兩個“型態”的“變數”。所以在C++的程式裡，我們是使用cin與cout來進行資料的輸入與輸出，而不是直接使用istream與ostream

不知道細心的讀者們有沒有發現，筆者在討論到istream與ostream這兩個“型態”，以及cin與cout這兩個“變數”的時候，都在其前後加上了“雙引號”?! 這是因為其實它們並不是“型態”與“變數”，正確來說應該是“類別”與“物件”！但又是老話一句「在本書還未介紹物件導向的概念前，是無法解釋清楚什麼是類別與物件的！」，所以筆者選擇先以目前為止，讀者應該可以接受的“型態”與“變數”的概念來說明istream ostream與cin cout間的關係，等到談到物件導向的概念時，我們再回過頭來解釋吧！

C++語言在iostream裡總共定義了名為cin cout cerr與clog的輸入/輸出串流供程式設計師使用，它們分別是：

- cin用以取得字元資料的輸入串流，預設連接到標準輸入渠道stdin
- cout用以輸出字元資料的輸出串流，預設連接到標準輸出渠道stdout
- cerr用以輸出由字元所組成的錯誤訊息的輸出串流，預設連接到標準錯誤渠道stderr
- clog用以輸出由字元所組成的日誌訊息的輸出串流，與cerr相同，預設都是連接到標準錯誤渠道stderr

這四個輸入/輸出串流，被稱為是C++語言的「標準串流(Standard Stream)它們都是以c開頭命名，代表是用以輸入或輸出字元(Character)資料的串流，又被稱做「字元串流(Character Stream)其中cin是連接到系統所創建的stdin負責用以取得使用者從鍵盤所輸入的資料；至於cout則是連接到stdout用以輸出到終端機cerr與clog都是連接到stderr用來輸出錯誤訊息與日誌資訊到終端機。

std::cin std::cout std::cerr與std::clog

還要提醒讀者，這四個標準串流是屬於std的命名空間，所以其全名應為std::cin std::cout std::cerr與std::clog但因為我們幾乎都會在程式開頭處使用 using namespace std;所以在大部份的情況下都是直接以cin cout cerr與clog的名稱加以使用。

本書後續在介紹其它新登場的變數、物件、函式時，也將會提醒讀者其所屬的命名空間。

所謂的log(日誌)是用以記錄程式執行過程的細節，就像是船隻在航行時會記錄航行日誌一樣，程式設計師可以選擇將特定的執行資訊記錄於日誌裡，例如程式何時開始執行、何時結束、取得了什麼資料、做了什麼處理等。這些資訊可供我們查詢程式執行過程中發生了哪些事情，可做為除錯或改善程式效率的參考。

在更深入說明串流的用途前，讓我們先回顧一下在第3章裡所介紹的BMI計算程式碼片段：

```
cin >> weight;
cin >> height;
```

這就是透過在程式與鍵盤之間的cin字元輸入串流(程式與鍵盤分別是流動的目的與來源)，來將使用者從鍵盤所輸入的體重與身高資料，流動到程式內的變數weight與height裡的一個例子。讓我們再繼續回顧這個範例程式：

```
BMI = weight / ( height * height );
cout << BMI << endl;
```

取得使用者所輸入的weight與height之後，此程式接著進行BMI值的計算(也就是進行體重除以身高平方的計算)，然後將結果透過存在於程式與終端機間的cout字元輸出串流(程式此時為流動的來源，螢幕終端機則為目的地)加以輸出 — 這就是cout字元輸出串流的使用範例。我們在此將此BMI計算程式是如何使用輸入與輸出串流的過程，呈現在figure 1裡，希望能幫助讀者更容易理解整個過程。

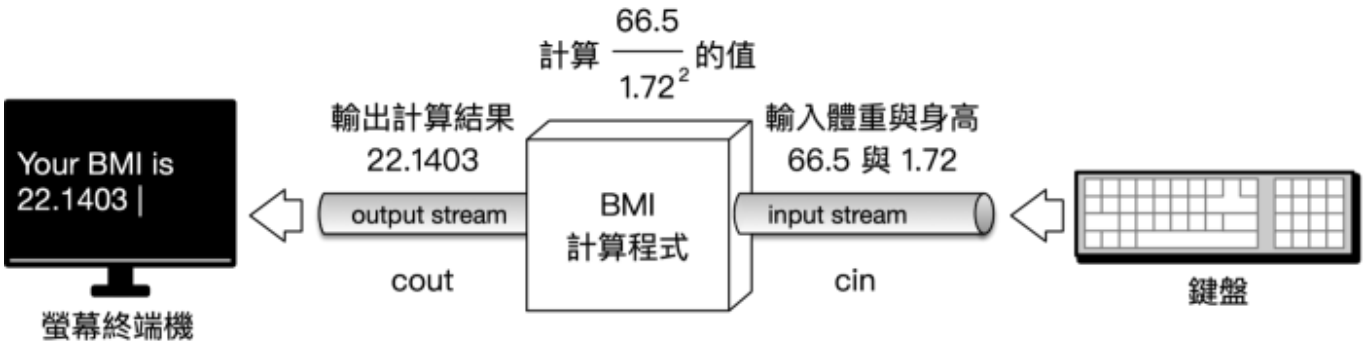

Fig. 1: BMI計算程式與cin輸入流及cout輸出流

讀者可能會指出「cin與cout不是也可以取得及輸出非字元型態的資料嗎？」，為何要把它們叫做“字元”輸入/輸出串流」呢？以3.2.2節的BMI計算程式為例，我們不就是

透過cin輸入串流取得了使用者所輸入的數值資料(身高與體重)，並且使用cout輸出串流將數值資料(經計算後得到的BMI數值)輸出到終端機嗎？

請再次參考figure1，沒錯！我們的確是透過cin這個“字元”輸入串流取得了數值資料「66.5」以及「1.72」，但是別忘了，這些所謂的數值資料都是使用者透過鍵盤所輸入的，以數值66.5為例，其實是使用者按下了兩次數字鍵6、1個小數點以及1個數字鍵5，然後按下Enter鍵後才提交給cin的。由於鍵盤上的每個按鍵都會對應到使用ASCII編碼的特定字元，因此66.5的輸入，其實是由使用者透過鍵盤連續輸入'6'、'6'、'.'、'5'這4個字元所組成的。當使用者按下Enter鍵時，這4個字元就會由cin進行後續處理—把'6'、'6'、'.'、'5'這4個字元變成數值66.5後，才存放到變數裡的！同樣的道理，終端機所能夠顯示的也只有字元，當我們把計算後的BMI數值22.1403透過cout輸出時，cout所做的其實是幫我們把22.1403轉換為連續的7個字元'2'、'2'、'.'、'1'、'4'、'0'及'3'，然後再將它們送交給終端機加以輸出。所以不論你所輸入輸出的是什麼樣的資料，從cin或cout的角度來看，其實就只是一堆連續的字元而已 — 這就是為什麼它們叫做“字元”串流的原因了！

6.2 cin輸入串流

我們在前一小節已經介紹過，cin是一個字元輸入串流，可以用來幫我們取得使用者所輸入的資料，本節將就其相關的使用情境加以說明。首先，請參考以下這種最簡單的使用情境 — 使用cin來將使用者的輸入放入到特定變數裡：

```
cin >> somewhere;
```

在過去的範例中，我們已經多次看過這種類似的做法 — 從cin輸入串流裡擷取資料放到變數somewhere裡。回想一下，除了在這種cin敘述裡，還有在哪裡看到過 >>？沒錯，» 是一個位元右移運算子，負責的是將二進制的數值進行右移(shift)的操作。咦，所以cin要往右移？移somewhere那麼多次？ 放心，當然不是的！前面已經先稍微提到「cin是iostream類別的物件」，做為物件導向的程式語言，類別有能力「改變遊戲規則」，它們可以「重新定義運算子」 — 讓運算子為物件服務，而不是讓物件位運算子服務！這個特性叫做「運算子重載(Operator Overloading)」等你“長大以後”再慢慢告訴你更多細節吧～ 現在，就先“享受”一下「cin所重載的位元右移運算子 » 吧～～ 呃，不對，它現在有個更貼切的名字 — 「串流擷取運算子(Stream Extraction Operator)」

6.2.1 串流擷取運算子

串流擷取運算子(Stream Extraction Operator)」其作用是從串流裡「擷取」資料[1]放到變數裡。哦，對了，它還有一個比較白話的名稱「Get From」運算子 — Get data from the stream」

做為一個運算子，>>是左關聯的二元運算子，左側的運算元必須是串流(例如cin)」右側的運算元(通常是變數)則是用來存放擷取回來的資料，當運算完成後(也就是完成資料的輸入或輸出後)，其位於左側的串流將做為其運算的結果。讓我們再以「cin >> weight」為例加以解析：由於在>>運算子的左右兩側分別是cin輸入串流與代表體重的變數weight」因此這個運算式的運算處理就是要從cin輸入串流裡「擷取」資料出來，並放入到變數weight裡。

cin >> weight ? cin << weight? 傻傻搞不清？

初學者有時會搞不清楚cin搭配的是 >> 還是 << ? 很簡單，為了幫助記憶，你可以把 cin >> weight想像成 cin → weight利用箭頭的方向→表明是把cin裡面的東西寫入到weight裡面。

本節接下來將說明cin輸入串流該如何和串流擷取運算子>>一同運作，以取得程式裡所需的資料。

6.2.2 從cin擷取資料

首先，讓我們從最簡單的使用情境開始 — 透過(連接到stdin渠道)的cin字元輸入串流取得一筆資料，並放入變數var裡面。在這種情境下，請使用cin >> var;運算敘述完成。請參考以下的程式片段：

```
int    a;
float  b;
double c;
char   d;

cin >> a;
cin >> b;
cin >> c;
cin >> d;
```

從上面的程式碼片段可以發現，儘管變數abcd與d的型態都不相同，但我們都是用同樣的方式從cin取得資料。這就是從cin串流裡擷取資料最棒的一點(當然也是最神奇的一點) — cin串流會視在 » 右側的變數之型態，「自動」幫我們將來自stdin裡的資料轉換為適當的型態！(如果你用過C語言的scanf()函式，你就會知道我在棒什麼～)

順便寫個C語言的版本給讀者進行比較：

```
int    a;
float  b;
double c;
char   d;

scanf("%d", &a);
scanf("%f", &b);
scanf("%lf", &c);
scanf("%c", &d);
```

如何？有沒有覺得C++好棒棒？！

現在讓我們看看在同一個運算式裡有兩個以上的串流擷取運算子的情況，由於>>是左關聯的運算子，且當運算完成後，其位於左側的串流將做為其運算的結果(因為很重要，所以又再講一遍)。請參考下面這個有兩個串流擷取運算子>>的例子：

```
cin >> a >> b;
```

此處兩個相同的串流擷取運算子>>，擁有相同的優先順序相同(廢言，都是同一個運算子當然相同)，依據左關聯的做法，此運算式將會從左至右進行，所以我們可以用括號將其執行的順序標明清楚：

```
(cin >> a) >> b;
```

當第一個>>運算子開始進行運算處理時，使用者就可以透過鍵盤先輸入一個數值資料，並將它放入到變數a裡面；要注意的是，當輸入完成以後(cin >> a)的運算結果將會是其原本左側的運算元 — 也就是此例中的cin，所以此運算式就變成了下面這樣：

```
cin >> b;
```

因此，第二個>>運算子就可以接著再讓使用者輸入第二個數值，並將它放入到變數b裡面。套用同樣的做法，我們可以將前面從cin擷取四筆資料的例子改寫為：

```
int    a;
float  b;
double c;
char   d;

cin >> a >> b >> c  >> d;
```

再順便寫個C語言的版本給讀者進行比較：

```
int    a;
float  b;
double c;
char   d;

scanf("%d %f %lf %c", &a, &b, &c, &d);
```

好的！我承認C++真的好棒棒！！

透過本節的說明，相信聰明的讀者們應該就可以理解為何筆者要將Stream譯做「串流」，而不單單是「流」而已了！

6.2.3 get()函式

我們已經先“ 預告 ”了很多次□cin輸入串流與cout輸出串流，都是以物件的方式實作，但是在還未正式為讀者說明物件導向的相關概念前，筆者還沒辦法為讀者深入介紹串流物件的細節。不過，為了讓讀者能夠更全面的使用cin輸入串流，所以我們打算在此先揭露一些些(是的，只能先講一些些… 不然後面就沒戲了…)...

再來一點物件導向吧！

假設有一個叫做integer的類別被設計用來代表整數，我們可以利用它來宣告一個“ 變數i” (好啦～ 沒騙到你，其實是物件啦):

```
integer i; // i is an object(物件) <-- 誰說i後面的be動詞一定要用am的?
```

這個“ 很像變數的物件i”□除了可以在其內部存放整數值以外，我們還可以讓它擁有一些操作的函式，例如我們可以事先將求絕對值、判斷是否為奇數、求i的j次方等操作整數的程式碼，事先撰寫在integer類別裡，並分別把它們命名為abs()□isOdd()與power(j)等函式，那麼作為integer類別的物件i□就可以使用這些函式。要使用這些函式的方法相當簡單，只要在物件名稱後面加上一個點以及該函式的名稱即可，例如：

```
i.abs();
i.isOdd();
```

我們把這種使用“ 操作方法 ”的方式稱為「呼叫(Calling)□□要使用什麼操作方法，就去“ 呼叫 ”它就可以了。

有時候，在呼叫操作方法的同時，還需要傳一些額外的資訊，例如呼叫power()方法來計算次方時，還要記得告訴它要計算的是幾次方：

```
i.power(j);
```

我們把這種額外的資訊，稱為「引數(Argument)□□所以上面這行呼叫可稱為「帶有

引數的函式呼叫」。

那我們就開始吧…

> 做為istream“型態”的“變數”cin其實…等等，先讓我們“轉譯”為正確的術語…
>
> > 做為istream“類別”的“物件”cin其實和一般的變數不一樣；一般的變數只能存放資料，但cin做為一個串流物件，它除了可以存放來自stdin的字元資料(讓»可以擷取並存放到變數裡)，還可以擁有一些函式供我們呼叫使用....

本節在此將先介紹cin輸入串流物件的一個相關的函式(當然，是定義在cin所屬的istream類別裡)，叫作get()它可以用來幫我們取得一個char型態的字元，它有兩種用法 — 帶引數及不帶引數：

- 帶引數呼叫：在呼叫get()時，把要用來存放所擷取回來的字元的變數做為其引數。例如cin.get(c);就會把擷取回來的字元放到變數c裡面。
- 不帶引數呼叫：在不帶引數的情況下，呼叫get()就可以取回一個字元，並將其所取回的字元視為是該呼叫的執行結果。如果你需要這個結果，你必須另外用別的變數加以保存，例如c=cin.get();使用變數c來存放cin所擷取回來的一個字元。讓我們來看看以下的範例：

`char c;`
`cin.get(c);`
`cout << "The character you inputted is " << c << endl;`
	`char c;`
	`c=cin.get();`
	`cout << "The character you inputted is " << c << endl;`

上面兩種方式的執行結果都是一樣的(取得一個字元、輸出一個字元)：

```
v⏎
The character you inputted is v
```

為了要在執行結果中，區分何者是使用者所輸入的內容？何者是程式的輸出？我們在使用者輸入的後面加上了⏎符號，以幫助讀者們區分輸入與輸出。

現在，讓我們把上面這兩個程式合併為一個：

```
char c;
cin.get(c);
cout << "The character you inputted is " << c << endl;
c=cin.get();
cout << "Another character you inputted is " << c << endl;
```

這程式的執行結果不是很容易預測嗎？不就是取得一個字元、輸出一個字元、再取得一個字元、再輸出一個字元！呃… 事情才沒聰明人想的那麼簡單～～ 讓我們來看看它(和你想得不一樣)的執行結果吧：

```
v⏎
The character you inputted is v
```

```
Another character you inputted is
```

它只取得了一個字元，然後就輸出一個字元、以及另一個字元，就這樣，然後沒了 Well 放心，不是你眼睛業障重、也不是你手指頭的問題，這個問題其實是緩衝區造成的，我們將在下一小節為你解答。

6.2.4 緩衝區

為了讓輸入與輸出更有效率 stdin採用了緩衝區(Buffer)的設計，讓所有經由鍵盤所輸入的內容，都先存放到緩衝區裡，直到緩衝區已滿、或是使用者明確地按下Enter鍵將輸入送出時，才會真正地將緩衝區裡的內容 “ 流動 ” 到stdin 進而再 “ 流動 ” 到其所預設連接的cin輸入串流裡。若是沒有緩衝區，那麼每當使用者按下任何鍵盤按鍵時，系統就必須將該輸入的字元送交給stdin 也就是要執行一次相對低速的I/O操作，系統整體的效能當然就會受到影響。

現在，讓我們解釋一下前面那個程式到底發生了什麼事？

```
char c;
cin.get(c);
cout << "The character you inputted is " << c << endl;
c=cin.get();
cout << "Another character you inputted is " << c << endl;
```

當我們執行到第二行的 cin.get(c); 時，使用者輸入了'v' 並且按下Enter鍵將它送出… — 此時的緩衝區裡存在以下的內 容：

'v','\n'

請注意 Enter鍵是一個不可視字元，儲存在電腦系統時是以ASCII的數值10表示，也就是C++語言裡的逸出字元'\n' 由於cin.get© 將其中的'v'擷取出來，並放到字元變數c裡面，所以緩衝區只剩下一個'\n'而已：

'\n'

接下來執行到下一行的 cout << “The character you inputted is ” << c << endl; 把字元變數c(其值為剛才擷取到的字元'v')加以輸出。然後，程式再繼續執行再下一行的 c=cin.get(); 試著再讀取使用者所輸入的下一個字元。然而在使用者還沒輸入下一個字元前，這行程式就已經直接從緩衝區裡擷取到了剛剛遺留在裡面的'\n' 所以字元變數c的內容就變成了'\n' — 根本不等使用者完成下一個字元的輸出，程式就已經繼續執行下去了。這就是這個程式所遇到的問題。

讓我們把程式修改一下，在第5行輸出第2個字元時，將字元變數c強制轉換型態為int整數(也就是其對應的ASCII編碼值):

```
char c;
cin.get(c);
cout << "The character you inputted is " << c << endl;
c=cin.get();
cout << "The ASCII value of another character you inputted is " << (int)c <<
endl;
```

此程式的執行結果就變成如下：

```
v⏎
The character you inputted is v
The ASCII value of another character you inputted is 10
```

看到了嗎？這裡所輸出的10就是'\n'的ASCII編碼值。既然已經理解問題的原因，那就可以想辦法來“對症下藥”了。

ignore()函式

cin物件可以使用定義在istream類別裡的ignore()函式，來將在緩衝區裡的內容加以清除，呼叫時需要兩個引數：size與delimiter[2]，用以指定清除在緩衝區裡面的前size個字元，或是是一直清除到遇到第一個delimiter字元為止。

讓我們將前面那個“有問題”的程式，在第2個get()前使用ignore()來清除在緩衝區裡造成問題的換行字元：

```
char c;
cin.get(c);
cout << "The character you inputted is " << c << endl;
cin.ignore(1,'\n');
c=cin.get();
cout << "Another character you inputted is " << c << endl;
```

此程式在第4行使用： cin.ignore(1,'\n'); ：來清除在緩衝區裡的第1個字元，或是清除到第1個換行字元為止 — 以本例來說，由於緩衝區裡只存在一個'\n'換行字元，所以兩者是完全相同的。其執行結果如下：

```
v⏎
The character you inputted is v
p⏎
Another character you inputted is p
```

好了，終於可以順利地取得第2個字元輸入了！

不過要注意的是，呼叫ignore()函式時，我們通常會把第1個引數設定為std::numeric_limits<streamsize>::max()：它是定義在limits標頭檔裡，屬於std命名空間，其值代表串流緩衝區大小的最大值；所以使用： cin.ignore(numeric_limits<streamsize>::max(), '\n');：就表示要清除掉在緩衝區裡的所有內容，或是遇到第1個'\n'為止 — 這是一個比較萬無一失的做法。只是千萬別忘了，必須要載入limits標頭檔案，才能正確的執行。請參考以下的程式：

```
char c;
cin.get(c);
cout << "The character you inputted is " << c << endl;
cin.ignore(numeric_limits<streamsize>::max(),'\n');
```

```
c=cin.get();
cout << "Another character you inputted is " << c << endl;
```

至此，關於緩衝區的討論暫時告一段落；不過，請相信我，在不久的將來，我們還會再遇到它～～

6.3 cout輸出串流

cout輸出串流預設連接作業系統所提供的stdout標準輸出串流，也就是在預設的情況下，使用cout輸出串流就可以把資料呈現的stdout預設連接的終端機裡。本節後續將從其所支援的運算子開始說明，並舉例示範如何用以輸出資料。

6.3.1 串流插入運算子

就如同前一小節的cin重新定義了>>一樣，cout輸出串流也重載了<<運算子 — 稱之為「串流插入運算子(Stream Insertion Operator)」，讓它往連接到stdout的cout輸出串流裡「插入」資料。它同樣也有個好記的別名「Put To」— Put data to the stream。

和>>相同，<<也是左關聯的二元運算子，左側的運算元必須是串流(例如cin)，右側的運算元(通常是變數、常數、字元及字元組成的文字串資料)則是用來指定所要輸出的資料，當運算完成後(也就是完成資料的輸出後)，其位於左側的串流將做為其運算的結果。

6.3.2 插入資料到cout串流

<<串流插入運算子的使用方式同樣也很簡單，只要將資料插入到cout串流裡就可以了，例如：

```
cout << something;
```

與cin的自動轉換型態一樣，cout也會自己想辦法(其實是寫在ostream類別的程式碼幫你完成的啦～)把你所“餵”給它的變數內容，轉換為stdout所需要的“字元”型態，例如：

```
int weight=66;
double height=172.5;
cout << weight; // 自動將66轉換為'6'、'6'再交給stdout輸出到終端機
cout << height; // 自動將172.5轉換為'1'、'7'、'2'、'.'、'5'再交給stdout輸出到終端機
```

同樣的事情換到C語言，是這樣寫的：

```
int weight=66;
```

```
double height=172.5;
printf("%d", weight);
printf("%f", height);
```

好的□C++棒多了！

現在讓我們示範使用多個串流插入運算子，將多筆資料一個串一個地輸出：

```
cout << "The value of a and b are " << a << "and" << b << endl;
```

讓我們依據左關聯的做法，使用括號來標明其執行的順序如下：

(((((cout << “The value of a and b are ”) << a) << “and”) << b) << endl);

現在，你應該“ 更 ”能理解為何把Stream叫做“ 串流 ”，而不只是“ 流 ”了。

6.3.3 put()函式

cout同樣也提供一些定義在ostream類別裡的函式供我們使用，在此我們選擇介紹和cin用來取得一個字元的get()函式相反的put()函式，其作用是讓cout輸出一個字元—嗯，一個get()□一個put()□拿出來、放進去，還蠻好記的。以下的程式碼簡單示範其使用方式：

```
char c='A';
cout.put(c);   //以字元變數c做為引數
cout.put('B'); //以字元值'B'做為引數
cout.put(67);  //以字元的ASCII數值做為引數，此處的67是英文大寫字母C的ASCII數值
```

上述的程式呼叫cout物件的put()函式，並將所要輸出字元做為引數 — 不論是變數或直接給定數值皆可(包含字元或是其對應的ASCII編碼皆可)，其輸出結果為□ABC□□

6.4 輸入與輸出格式設定

本節將介紹如何控制cin與cout物件，讓它們可以依特定的格式來進行資料的輸入與輸出。我們將會使用到定義在istream與ostream類別裡的函式，以及一些定義在ios□iostream□iomanip等標頭檔案裡的一些專門被設計用來設定cin與cout輸入/輸出格式的函式。除此之外，我們也會介紹如何讓cin□cout配合取得或輸出各種資料型態與數字系統的資料。

6.4.1 輸出寬度與對齊設定

cout預設的輸出是採用「靠右對齊(Right Align)□的格式，但也可以透過設定更改為「靠左對齊(Left Align)□□等等…到目前為止所有的程式輸出好像都是靠左邊，沒有看到靠右邊的呀～ 為什麼說預設是靠右呢？ 沒錯，你說的對，目前為止所看到的“ 好像 ”都是靠左對齊的，例如：

```
cout << "Hello" << endl;
```

它的輸出結果為：

```
Hello
```

沒錯，的確靠左邊，但這只是“ 假象 ”，預設的確是靠右對齊，但你有設定右邊界在哪裡嗎？如果有適當地設的輸出資料的右邊界在哪，你就能看到預設的靠右對齊的效果了。

width()函式

做為ostream類別的物件cout可以使用定義在ostream類別的函式，其中名為width()的函式就可以幫我們設定使用cout輸出資料時的右邊界，或者更正確的說width()就是寬度的意思，它所設定的是cout輸出的“ 範圍 ” — 包含左邊界與右邊界。再換種方式來說，所謂的寬度是指cout串流所輸出的資料最多能夠呈現的字元個數，例如使用cout.width(5);將寬度設定為5，就表示最多只能顯示5個子元，若所輸出的字元數少於5個，那麼會往右側對齊在第5個字元處，且其所缺的位數會以空白字元填補。請參考以下的例子：

```
int a=1024;
cout << "123456789" << endl;
cout.width(9);
cout << "Hello" << endl;
cout << "C++" << endl;
cout.width(2);
cout << a;
```

上面的程式片段先輸出一行「123456789」做為“ 尺標 ”的功能，讓我們可以更容易看出輸出格式控制的結果。接著在第3行及第6號呼叫cout物件的width()函式，並分別使用9與2做為其引數 — 將待會使用cout串流的輸出設定為切齊由寬度所定義的右邊界，也就是會將輸出的資料往右側切齊在第9個與第2個字元處。程式還使用了3個cout敘述，分別將“Hello”、“C++“以及變數a的數值加以輸出，其執行結果如下：

```
123456789
    Hello
C++
1024
```

正如你所看到的，受到cout.width(9)的影響，所以其下一行的cout輸出”Hello”時就會被限制在9個字元的寬度，而且是採用靠右對齊的方式編排(你可以很容易地從上一行的數字尺標來比對其輸出結果)。但是，要特別提醒讀者注意的是，cin.width()是一次性的設定(換句話說，它只規範了下一次的cout輸出)，因此在其下一行的cout « “C++”;並不會受到此寬度的影響，它回歸到沒有設定寬度的情況(也就是cout.width(0)的意思)，由於沒有寬度、就沒有所謂的右邊界，自然不會有靠右對齊的效果產生。後續第6行又進行了一次設定將寬度設為2，這時就發生了第7行所要輸出的變數a(因為其數值為1024，需要4個字元)所需的空間超過了所設定的寬度2 — 這些情況都是所要輸出的資料超出了wdith()所設定的寬度cout串流的做法是把資料完整輸出而不管寬度限制了。所以最後兩行的輸出看起來“ 好像 ”靠左對齊 — 其實，它們還是預設的靠右對齊，只不過超出了右邊界而已。

fill()函式

其實從結果來看，所謂的靠右對齊只不過是在輸出資料的前面，加上適當個數的空白字元，好讓輸出的結果看起來有靠右對齊的感覺而已 — 在前面的例中，就是在“Hello”的前面，補上4個空白字元。如果你不喜歡空白字元，也可以使用fill()函式來改為你偏好的字元。請參考以下的程式碼片段：

```
int a=1024;
cout.width(9);
cout.fill('#');
cout << "Hello" << endl;
cout.width(9);
cout << "C++" << endl;
cout.width(9);
cout << a << endl;
```

其執行結果如下：

```
####Hello
######C++
#####1024
```

由於使用了fill()函式來設定用來 “ 填補 ” 的字元為'#'，因此在“Hello”前面就會補上4個#號。在上面這段程式片段裡，我們還在輸出完“Hello”後，再次使用兩次的width(9)將稍後2次的cout串流輸出再次設定為靠右對齊到第9個字元處，其執行結果正如你所看到的，全部都切齊到第9個字元處了。對了，這個例子還有一個作用，它展示了對比width()函式只有一次性的作用□fill()函式的設定則是具有持續性的效果。

cout除了可以使用定義自ostream類別裡的函式以外，還有一些定義在ios□iostream□iomanip或其它標頭檔案裡的函式也可以搭配cout一起使用。本節後續所要介紹的函式是專門設計用來操控串流的輸入與輸出格式，又被稱為「串流操控子(Stream Manipulator)□□

std::setw()

首先我們要介紹的是名為setw()的串流操控子，它是定義在iomanip標頭檔案[3)]裡面的函式，而且屬於std命名空間，所以你必須使用#include <iomanip>載入其標頭檔，並且要記得使用std命名空間□setw()串流操控子的命名來自set width之意，其功能也和width()函式一樣，都是一次性的設定接下來串流輸出資料的寬度，並且將輸出靠右對齊□setw()的使用法方，請參考以下的程式碼：

使用setw()串流操控子	使用cout的width()函式
`cout << setw(9) << "Hello";`	`cout.width(9);` `cout << "Hello";`

```
    Hello
```

好啦～ 我知道看不清楚在“Hello”的前面到底有幾個空白，讓我們修改一下上面的程式碼，把fill()函式加進去：

使用setw()串流操控子	使用cout的width()函式

`cout.fill('#');`
`cout << setw(9) << "Hello";`
	`cout.fill('#');`
	`cout.width(9);`
	`cout << "Hello";`

它們的執行結果如下：

```
####Hello
```

還有人看不清楚嗎？@_@

std::left, std::right 要靠左還是靠右？

其實不論是使用width()函式或setw()操控子都只是單純的設定了輸出的「範圍」而已 — 這個範圍不但包含右邊界的設定，同時也包含了左邊界；有些人會誤以為wdith()與setw()所設定的是靠右對齊，只不過是因為cout預設的對齊方式是靠右而已。

我們可以使用left與right這兩個串流操控子，它們定義在ios與iostream標頭檔裡，只要使用#include將它們其中之一載入即可；還有，它們屬於std命名空間，要記得using namesapce std或是使用std::left與std::right去使用它們。讓我們看看它們的使用方式：

使用setw()串流操控子	使用cout的width()函式

`cout.fill('#');`
`cout << left << setw(9) << "Hello" << endl;`
`cout << setw(9) << "C++" << endl;`
	`cout.fill('#');`
	`cout.width(9);`
	`cout << left;`
	`cout << "Hello" << endl;`
	`cout.width(9);`
	`cout << "C++" << endl;`

它們的執行結果如下：

```
Hello####
C++######
```

從上面的例子可以觀察到□left的設定是“ 持續性 ”的，不像是setw()與width()只有一次性的作用。如果你想要改回預設的靠右對齊，那麼只要使用right即可：

使用setw()串流操控子	使用cout的width()函式

`cout.fill('#');`
`cout << left << setw(9) << "Hello" << endl;`
`cout << setw(9) << right << "C++" << endl;`
	`cout.fill('#');`
	`cout.width(9);`
	`cout << left;`
	`cout << "Hello" << endl;`
	`cout << right;`
	`cout.width(9);`
	`cout << "C++" << endl;`

請注意在上述程式碼中，我們也“故意”示範了寬度與對齊的設定是可以依任意順序使用的。它們的執行結果如下：

```
Hello####
######C++
```

現在，再讓我們看看下一個例子：

```
cout.fill('.');
cout << setw(9)  << "Name"     << setw(10) << "Score" << endl;
cout << setw(14) << "Jun Wu"   << setw(5)  << 100     << endl;
cout << setw(14) << "Alex Liu" << setw(5)  << 90      << endl;
```

這段程式碼適當地利用setw()將輸出加以對齊成“貌似”表格一樣，請參考下面的執行結果：

```
.....Name.....Score
........Jun Wu..100
......Alex Liu...99
```

想法很好，但看起來有點不漂亮～～ 如果能夠將名字的部份靠左對齊，成績的部份維持靠右，看起來可能會好一點… 也就是像下面這樣：

```
Name.........Score
Jun Wu..........100
Alex Liu.........99
```

如果要改成這樣，同樣是使用setw()操控子來設定寬度，但我們可以利用left與right分別設定它們對齊的方式：

```
cout << left << setw(9)  << "Name"     << right << setw(10) << "Score" <<
endl;
cout << left << setw(14) << "Jun Wu"   << right << setw(5)  << 100     <<
endl;
cout << left << setw(14) << "Alex Liu" << right << setw(5)  << 90      <<
endl;
```

使用cin也可以設定輸入資料的寬度與對齊方式嗎？ 相信讀者對於這個問題應該很感到興趣，其實答案當然是可以的，可以對於本節目前為止所示範的內容來說，寬度與對齊的設定還不能套用到cin輸入串流。相關的輸入“寬度”與“對齊”設定，我們將留待第X章 字串再進行討論，敬請期待!

6.4.2 浮點數的精確度

在預設的情況下，cout輸出浮點數數值預設的精確度(Precision)是6，意即在「數字」部份將可以顯示到6個數字，請先觀察以下的範例：

```
cout << 12.345 << endl;
cout << 12.3456 << endl;
cout << 12.34567 << endl;
cout << 76.54321 << endl;
cout << 123456.789 << endl;
cout << 1234567.89 << endl;
```

其執行結果如下：

```
12.345
12.3456
12.3457
76.5432
123457
1.23457e+06
```

從上述的執行結果可以觀察到，cout在輸出浮點數值時，所謂的6位數的精確度，指得是不包含小數點在內的6個位數，只要數值扣除掉小數點後的位數不超過6位，全部都可以精確呈現(不論是小數點前的整數部份，或是小數點後的小數部份)，例如:

- 12.345 → 輸出 12.345
- 12.3456 → 輸出 12.3456

當扣除掉小數點後的位數超過6位時，則會採用四捨五入的方式到第6個位數，例如：

- 12.34567 → 超出的部份進位 → 輸出12.3457
- 76.5432 → 超出的部份捨棄 → 輸出76.5432
- 123456.789 → 超出的部份進位 → 輸出123457

如果所要輸出的浮點數數值的整數部份超出了6位，那麼cout會自動改以「科學記號表示法(Scientific Notation)」將數值輸出，例如：

- 1234567.89 → 整數位數超出6位 → 輸出1.23457e+06 也就是1.23457×10^6之意。

cout串流輸出浮點數值的預設6位精確度是不包含 “ 負號 ” 的。我們將上面的範例修改，將輸出的數值都改為負數：

```
cout << -12.345 << endl;
cout << -12.3456 << endl;
cout << -12.34567 << endl;
cout << -76.54321 << endl;
cout << -123456.789 << endl;
cout << -1234567.89 << endl;
```

其執行結果如下：

```
-12.345
-12.3456
-12.3457
-76.5432
-123457
-1.23457e+06
```

從結果來看，數字部份保持著6位數的輸出，沒有受到負號的影響。

接下來還要提醒讀者注意的是，浮點數的輸出精確度與“ 寬度 ”無關，請參考以下的例子：

```
cout.fill('*');
cout << setw(10) << 12.345 << endl;
cout << setw(10) << 12.3456 << endl;
cout << setw(10) << 12.34567 << endl;
cout << setw(10) << 76.54321 << endl;
cout << setw(10) << 123456.789 << endl;
cout << setw(10) << 1234567.89 << endl;
```

其執行結果如下：

```
****12.345
***12.3456
***12.3457
***76.5432
****123457
1.23457e+06
```

從其執行結果可發現輸出的數值並沒有因為寬度設定為10，就能夠顯示更多的位數 — 輸出的寬度與精確度是兩個不同的設定，彼此並不相關。

precision()函式與std::setprecision()操控子

要改變cout輸出浮點數的精確位數，可以使用定義在iomanip標頭檔裡的std::setprecision()串流操控子，或是使用cout的precision()函式，請參考以下的例子：

使用setprecision()串流操控子	使用cout的precision()函式

`cout << setprecision(8);`
`cout << 12.345 << endl;`
`cout << 12.3456 << endl;`
`cout << 12.34567 << endl;`
`cout << 76.54321 << endl;`
`cout << 123456.789 << endl;`
`cout << 1234567.89 << endl;`
	`cout.precision(8);`
	`cout << 12.345 << endl;`
	`cout << 12.3456 << endl;`
	`cout << 12.34567 << endl;`
	`cout << 76.54321 << endl;`
	`cout << 123456.789 << endl;`
	`cout << 1234567.89 << endl;`

其執行結果如下：

```
12.345
12.3456
12.34567
76.54321
123456.79
1234567.9
```

std::fixed操控子

上一小節已經介紹過□setprecision()操控子與precision()函式都可以用來設定cout在輸出浮點數時的精確度—除小數點以外要輸出的位數(包含小數點前與小數點後的部份)。如果我們只想要設定小數點後的部份，那麼就可以使用定義在ios與iostream標頭檔(兩個標頭檔載入其中一個即可)裡的std::fixed操控子來達成：

```
cout << setprecision(4);
cout << fixed;
cout << 3.123    << endl;
cout << 5.132223 << endl;
cout << 79.29228 << endl;
```

由於我們在第2行使用了fixed操控子，所以就將第1行所設定的4位精確度“ 限縮 ”到只規範小數的部份 — 也就是說設定為小數點後顯示4位的輸出。請參考以下的執行結果：

```
3.1230
5.1322
79.2923
```

從上述的執行結果可以發現□fixed的設定是「持續性」的；此外，如果小數點後少於精確度要求的位數會補0，但超出的部份則會四捨五入到所設定的精確位數。

std::defaultfloat操控子

defaultfloat操控子從GCC 5.1後開始支援，系計中的ws工作站目前暫不支援。

由於fixed的設定是「持續性」的，如果要改回預設的包含小數點以外所有的位數，那麼可以使用另一個同樣定義在ios與iostream標頭檔裡的std::defaultfloat操控子：

```
cout << setprecision(4);
cout << fixed;
cout << 3.123    << endl;
```

```
cout << defaultfloat;
cout << 5.132223 << endl;
cout << 79.29228 << endl;
```

執行結果如下：

```
3.1240
5.132
79.29
```

由於我們在第4行使用defaultfloat還原回預設設定，所以後兩個cout所輸出的浮點數，其整數與小數部份合計都不能超過4個位數。

std::scientific操控子

前面已經提到過，當浮點數的整數部份位數大於所設定的精確度時，cout會自動改以「科學記號表示法(Scientific Notation)」將數值輸出。如果要強制將浮點數改為科學記號表示法，那麼就可以使用又是同樣定義在ios與iostream標頭檔裡的std::scientific操控子：

```
cout << scientific;
cout << 31.24  <<  endl;
cout << setprecision(4);
cout  << 5.132223 << endl;
cout << std::defaultfloat;
cout << 79.29228 << endl;
```

其執行結果如下:

```
3.124000e+01
5.1322e+00
79.29
```

從執行結果可以發現，scientific是「持續性」的設定，且精確度的設定會套用在有效數(significand)的小數部份，例如上例中第1個及第2個輸出的有效數的小數分別經由精確度設定為6位與4位；另外，如果要回復到原先的預設浮點數輸出方式，則同樣可以使用defaultfloat操控子完成。

std::showpoint與std::noshowpoint操控子

請參考以下的程式碼：

```
double a=3.0;
cout << a << endl;
```

此程式的執行結果可能和你想的並不一樣：

```
3
```

由於此例中的浮點數變數值為3.0，並沒有小數的部份，所以cout預設在輸出時連小數點都不呈現。如果你不滿意這樣的做法，可以使用showpoint操控子強制cout輸出浮點數時一定要包含小數點 — 在含有小數點的情況下，小數的部份也會強制顯示出來(儘管它們都是0啦)。showpoint操控子的設定是持續性的，但你可以使用noshowpoint操控子將它關閉。請參考以下的程式：

```
double a=3.0;
cout << showpoint;
cout << a << endl;
cout << a << endl;
cout << noshowpoint << a << endl;
```

此程式的第2行設定要強制顯示小數點，並在接下來的兩行將a的數值輸出兩次，好讓你檢查看看showpoint的效果是否真的是持續性的。後續在第5行則使用noshowpoint將原先的設定關閉，並將不帶小數點的3加以輸出，其執行結果如下：

```
3.00000
3.00000
3
```

6.4.3 數字系統

數字系統指的是數值所使用的基底(Base)。除了一般日常生活慣用的十進制(Decimal)數字系統以外，還有資訊界慣用的二進制(Binary)、八進制(Octal)與十六進制(Hexdecimal)。分述如下：

- 二進制(Binary)：以2為基底，每個位數由0到1，共2種可能，超出後則進到下一位數。
- 八進制(Octa)：以8為基底，每個位數由0、1、2、3、4、5、6到7，共8種可能，超出後則進到下一位數。
- 十進制(Decimal)：以10為基底，每個位數由0、1、2、3、4、5、6、7、8到9，共10種可能，，超出後則進到下一位數。
- 十六進制(Hexdecimal)：以16為基底，每個位數由0、1、2、3、4、5、6、7、8、9、A、B、C、D、E到F，共16種可能，超出後則進到下一位數。

本節將分別就如何透過cin取得不同數字系統的數值，以及如何使用cout輸出加以說明。

6.4.3.1 設定cin輸入的數字系統

cin除了可以使用定義自istream類別裡的函式(例如前面所介紹的cin.get()函式)以外，還有一些定義在iostream或其它標頭檔案裡的函式也可以搭配cin一起使用。本節所要介紹的函式都是專門設計用來操控串流的輸入與輸出格式，又被稱為「串流操控子(Stream Manipulator)」— 此處的主角當然是可以設定讓cin取得不同數字系統的操控子：

- std::dec命名取自Decimal的縮寫，意即採用十進制的數字系統。
- std::hex命名取自Hexdecimal的縮寫，意即採用十六進制的數字系統。
- std::oct命名取自Octal的縮寫，意即採用八進制的數字系統。

我們可以在使用cin取得數值資料時，使用以上述的操控子來規範所要使用的數字系統為何？要注意的是，儘管在電腦系統裡，相對比較重要的數字系統是二進制，但C++並沒有支援二進制的輸出與輸入。

以下的範例要求使用者輸入十進制、十六進制與八進制的數值：

```
int a, b, c;
cin >> dec >> a;
cin >> hex >> b;
cin >> oct >> c;
cout << a << endl;
cout << b << endl;
cout << c << endl;
```

在下面的執行結果裡，我們先輸入了三個100，但它們分別是十進制、十六進制與八進制的數值，後續再使用cout將它們都輸出為10進制(cout預設就是以十進制來輸出數值)：

```
100⏎
100⏎
100⏎
100
256
64
```

幫設定要取得特定數字系統的數值時，若使用者的輸入了超出該數字系統該有的內容時，cin只會擷取符合的部份，請考以下的程式：

```
cin >> hex >> a;
cout << hex << a;
```

使用者在此應該要輸入一個十六進制的數字FF20，但卻不小心打錯為FF2O(零打成歐):

```
FF2O⏎
FF2
```

從執行結果可看出，儘管使用的輸入了不正確的十六進制數值，但cin仍然還是幫我們將正確的部份取回。

6.4.3.2 設定cout輸出的數字系統

cout也可以使用std::dec、std::hex與std::oct來將數值輸出為十進制、十六進制與八進制。請參考以下的程式範例：

```
int a=100;
cout << dec << a << endl;
cout << hex << a << endl;
cout << oct << a << endl;
```

其執行結果如下：

```
100
64
144
```

除此之外，還有一些操控子可以設定輸出格式：

- std::setbase() 設定所要使用的數字系統，可用的引數包含8、10與16，若給定其它數值則一律視為10進制。當引數為8、10或16時，其作用等同於oct、dec與hex。
- std::showbase 設定要輸出各數字系統置於數值前的前綴，例如八進制及十六進制數值前分別冠以0及0x。
- std::noshowbase 關閉showbase設定。
- std::uppercase 設定在輸出數值時，將其中包含的英文字母以大寫方式輸出。主要用於十六進制的數值，包含其0X前綴以及數值A、B、C、D、E與F。
- std::nouppercse 關閉uppercase設定。

要注意的是，以上的操控子皆具持續性。

請參考以下的範例：

```
int a;
cin >> dec >> a;
cout << setbase(10) << a << endl; //設定為十進位
cout << showbase;                   // 設定要輸出數字系統前綴
cout << setbase(8) << a << endl;   // 設定為八進制
cout << setbase(16) << uppercase << a << endl; //設定為十六進制，並將英文字母部份設定為大寫
cout << nouppercase << a << endl;  // 取消大寫設定
cout << setbase(16) << uppercase << a << endl; //設定為十六進制，並將英文字母部份設定為大寫
cout << oct << noshowbase << a << endl;  // 取消輸出數字系統前綴
```

其執行結果如下：

```
100⏎
100
0144
0X64
0x64
144
```

6.4.4 布林型態的數值

C++語言的布林型態bool可以有兩種數值□true與false□分別用以表示某種情況、情境或是狀態、條件的「正確」與「錯誤」、「成立」與「不成立」、「真」與「偽」等「正面的」或「負面的」兩種可能，讀者可以回顧本書4.3.4 布林型態。要在提醒讀者注意的是，布林型態的數值true與false亦可以整數表示，其中所有非0的整數值皆視為true(但預設值為1)□false則使用0表示。

boolalpha

讓我們看看以下的範例：

```
bool b1=true;
bool b2=false;
cout << b1 << endl;
cout << b2 << endl;
```

其執行結果如下：

```
1
0
```

從此執行結果可看出，在預設的情況下□cout會將bool型態的變數值輸出為整數，其中以1代表true□以0代表false□如果你不喜歡“ 看到 ”這種用整數代表布林值的結果，可以使用定義在ios與iosteam標頭檔裡的boolalpha串流操控子，設定cout將bool型態輸出為true與false□請參考以下的例子：

```
bool b1=true;
bool b2=false;
cout << boolalpha;
cout << b1 << endl;
cout << b2 << endl;
```

由於使用了cout « boolalpha的設定，所以cout將會把bool型態的數值以true□false輸出，其執行結果如下：

```
true
false
```

noboolalpha

請注意□boolalpha操控子是持續性的設定，如果要取消可以使用另一個操控子noboolalpha□這一組boolalpha與noboolalpha操控子除了可以設定cout的輸出以外，也可以用來設定cin的輸入。請參考以下的程式：

```

```

```
bool b1, b2;
cin >> boolalpha >> b1;
cin >> b2;
cout << boolalpha << b1 << endl;
cout << noboolalpha;
cout << b2 << endl;
```

此程式的執行結果如下：

執行結果：

1.
```
true⏎
true⏎
true
1
```

2.
```
false⏎
false⏎
false
0
```

3.
```
typo⏎
false⏎
false
0
```

此程式的執行結果依據使用者輸入的不同而有所差異，因此我們將其多次的執行畫面都加以呈現，以涵蓋各種可能的使用者輸入；例如我們針對這個程式，分別考慮了使用者輸入true□false與typo(輸入了true與false之外的內容，也就是錯誤的輸入情況)。

寫給不同作業系統的用戶

上面這個範例程式依據使用者輸入的不同、所使用的作業系統不同，其執行結果的畫面將會有些差異。因此我們將此程式在Linux/MacOS與Windows上的多次執行畫面都分別加以呈現，以幫助使用不同作業系統的讀者：

Linux/Mac用戶	Windows用戶
[user@urlinux examples]$ ./a.out true⏎ true 1 [user@urlinux examples]$ ./a.out false⏎ false 0 [user@urlinux examples]$ ./a.out typo⏎ false 0 [user@urlinux examples]$	C:\example> a true⏎ true 1 C:\example> a false⏎ false 0 C:\example> a typo⏎ false 0 C:\example>

細心的讀者應該已經發現，上述兩種輸出結果，其實只有在“ 下達執行程式 ”的指令上略有差異，整體而言兩者“ 執行結果 ”內容是相同的。因此，本書後續在這種需要顯示多次不同輸入的執行結果時，將不再顯示不同作業系統的執行結果，同時我們將省略“ 下達執行程式 ”的指令，僅提供“ 執行結果 ”的部份供讀者參考。

6.4.5 再談緩衝區

請先閱讀以下程式，想想看，它的執行結果為何？

```
bool b1, b2;
cin >> boolalpha;
cin >> b1;
cin >> b2;
cout << boolalpha;
cout << b1 << endl;
cout << b2 << endl;
```

嗯⋯ 想好了，這個程式執行兩次的「接收使用者輸入的true或false的布林值，然後加以輸出」。請比對以下的執行結果：

1. ```
   true⏎
   false⏎
   true
   false
   ```

2. ```
   truth⏎
   false
   false
   ```

其中第1組執行結果和你想的一樣「接收使用者輸入的true或false的布林值，然後加以輸出□x 2□但第2組的執行結果，只有「接收使用者輸入的true或false的布林值」一次，然後就連續輸出兩個布林值了！那請你再想想看，為什麼會這樣～～

答案和之前介紹cin.get()時一樣，又是緩衝區的問題。請仔細看上面的第2組執行結果，使用者在輸入第1個布林值時發生了錯誤，把true打成了truth□所以在程式中第3行的□cin >> b1;□沒能擷取到正確的布林值，因此會將b1視為false□然而，正因為發生了這個錯誤，所以對於下一個第4行的□cin >> b2;□造成了影響，導致b2也沒能擷取到正確的布林值，所以b2也被視為是false□

好的，沒問題，遇到問題就來解決問題，既然知道原因，那麼又可以來「對症下藥」了！我們將程式修改如下：

```
bool b1, b2;
cin >> boolalpha;
cin >> b1;
cin.ignore(numeric_limits<streamsize>::max(), '\n');
cin >> b2;
cout << boolalpha;
cout << b1 << endl;
cout << b2 << endl;
```

其實我們在前面6.2.4 緩衝區已經遇過類似的情況，所以這次直接在第4行處增加清空緩衝區的程式碼(要記得#include <limits>□將numeric_limits<streamsize>::max()所需的標頭檔載入)。好了，搞定收工，看看它的執行結果吧：

```
truth⏎
false
false
```

等等，結果怎麼還是不正確？！其實此處所遇到的問題和6.2.4 緩衝區節的問題並不相同，所以不能一概而論。此處所遇到的問題，其實是因為在前一個□cin >> b1;□使用者輸入了錯誤的內容(既非true亦非flase)所導致的，因此□cin被註記在擷取資料時發生錯誤；至於6.2.4 緩衝區節的問題只是有遺留的換行字元，我們並不能確定那是不是一種錯誤(說不定是有意為之)。

針對此種cin發生擷取資料錯誤的情形，除了將緩衝區清空外，更重要的是記得使用clear()函式，來將cin被註記的錯誤「解除」，請參考以下的程式：

```
bool b1, b2;
cin >> boolalpha;
cin >> b1;
cin.clear();
cin.ignore(numeric_limits<streamsize>::max(), '\n');
cin >> b2;
cout << boolalpha;
cout << b1 << endl;
cout << b2 << endl;
```

這次我們在第4行使用cin.clear();將錯誤狀況解除，並在第5行清除了緩衝區，程式的執行結果終於可以在第一個布林值輸入錯誤的情況下，讓我們繼續擷取下一個布林值了！請參考以下的結果：

```
truth⏎
true⏎
false
true
```

搞定 

6.5 cerr與clog輸出串流

cerr與clog預設都是連接到stderr用來輸出錯誤訊息與日誌資訊到終端機，而stderr在許多系統的實作上與stdout一樣，都是連接到終端機。它們的使用方式和cout完全相同，本節針對cout所介紹的各種使用方式，都能套用在cerr與clog裡，在此不予贅述。

當然，它們還是有些不一樣啦～ 其最主要的差別是cerr是無緩衝的，而clog是有緩衝的。由於cerr設計的目的是要顯示程式的錯誤訊息，這有一定的時效性，所以採用無緩衝的設計好讓所有經由cerr輸出的資料，可以直接快速地(相對於有緩衝的設計)輸出到stdout相對的clog的輸出是做為程式執行時的工作日誌用途，所以並沒有時效性的問題，所以和cout一樣被設計為有緩衝的。

所謂的緩衝(Buffer)可以想像為一塊記憶體空間，有緩衝的輸出串流會將所有的輸出都先放在緩衝區裡，等到遇到以下的輸入或情況，才會將緩衝區內的資料送交給其所連接到的裝置(例如終端機)加以顯示：

- flush()函式 — 強制刷新緩衝區(也就是將緩衝區內所有的內容都交由stdout輸出)
- endl — 其實它也是串流操控子，其作用是送出一個換行字元'\n'然後使用flush()強制清空緩衝區
- 緩衝區已滿

由於此部份受到不同作業系統實作的差異，以及串流所連接的實體裝置的不同，其實並沒有一致性的做法。在終端機操作時，一般而言，由於採用行緩衝(Line Buffered)所以每當遇到使用者按下Enter時(也就是產生一個換行字元'\n'時)就會將緩衝區清空；但若是連接到檔案時，遇到換行字元也不會將緩衝區清空，除非遇到緩衝區已滿或使用flush()強制清空時，才會發生作用。有時候，甚至不是作業系統的問題，有一些編譯器(包含許多人使用的GNU編譯器)，連flush()都沒有實作出該有的功能。

有緩衝的設計，讓輸出串流不需要每一次得到一個輸入就把它立刻輸出到實體裝置上，因為這會耗用掉高成本的I/O操作；採用緩衝的設計，能在匯集較多的輸入資料以後，才將其加以輸出，可以有效減少系統發生I/O操作的次數，進而提升系統效能。

6.6 I/O重導向與Pipe管線

正如本章開頭處所說的，作業系統為每個程式準備了stdin、stdout與stderr三個標準的輸入/輸出渠道，其中stdin預設連接到鍵盤輸入裝置、stdout與stderr則預設連接到終端機。當程式在執行的時候，大部份的作業系統因為或多或少都承襲了早期Unix系統的特性，所以也都有提供從早期就有的I/O重導向(I/O Redirection)與Pipe管線的操作方法。簡單來說I/O重導向就是讓我們可以把stdin、stdout與stderr重新連接到系統內的其它資源，包含特定的檔案與硬體裝置Pipe管線則更進一步讓我們可以把不同程式的標準的輸入/輸出渠道互相連接，因此一個程式可以從另外的程式取得輸入的資料，而且也可以將輸出的資料做

為其它程式的輸入。如此一來，具有不同功能的程式，就可以串接起來進而提供更強大的功能。由於C++語言所提供的cin、cout、cerr與clog四個標準串流，也是對應連接到標準的輸入/輸出渠道(其中cin連接到stdin、cout連接到stdout、cerr與clog則都連接到stderr)。因此上述的I/O重導向與Pipe管線也能實現在使用C++語言所撰寫的程式裡。

I/O重導向(I/O Redirection)具體的做法是使用 > 、 » 與 < 符號，指定所要轉向的來源或目的。請先參考以下的程式：

```
#include <iostream>
using namespace std;

int main()
{
   int a, b;
   cin >> a;
   cin >> b;
   cout << (a+b) << endl;
}
```

接著開啟任何你偏好的文字檔案編輯軟體，建立一個檔案名為data.txt，其內容如下：

```
121
72
```

現在請在將addTwoNumbers.cpp編譯成檔名為addTwoNumbers的可執行檔(Windows系統的讀者請編譯為addTwoNumber.exe)。

Linux/MacOS的讀者可以使用 -o 參數指定要產生的可執行檔檔名：

```
[user@urlinux examples]$ C++ addTwoNumbers.cpp -o
addTwoNumbers
```

至於使用Windows的讀者，則可以在Dev-C++裡進行相關的設定。

現在請打開終端機(Windows的讀者請打開命令提示字元)試著執行該檔：

Linux/Mac用戶	Windows用戶
[user@urlinux examples]$ ./addTwoNumbers 5 3 8 [user@urlinux examples]$	C:\example> addTwoNumbers 5 3 8 C:\example>

現在，讓我們試著將addTwoNumbers這個程式所使用的stdin進行I/O重導向，將原本要從鍵盤取得的輸入，改為從data.txt檔案讀取，也就像是把data.txt「餵」給addTwoNumbers一樣： addTwoNumbers ← data.txt請參考下面的做法：

Linux/Mac用戶	Windows用戶

`[user@urlinux examples]$ ./addTwoNumbers < data.txt`
`193`
`[user@urlinux examples]$`
	`C:\example> addTwoNumbers < data.txt`
	`193`
	`C:\example>`

由於data.txt內的兩個數字分別為121與72，所以上述的執行結果將會是其相加後的193。注意到了嗎？addTwoNumbers < data.txt 這個指令就長得和 addTwoNumbers ← data.txt一樣，所以應該很容易記得。透過這個用來進行輸入重導向的 < 符號，在程式裡原本透過cin從stdin取得使用者從鍵盤所輸入的資料，就變成是從data.txt檔案取得其內容做為輸入。

除了輸入可以重導向以外，我們也可以使用 > 符號進行輸出的重導向：

Linux/Mac用戶	Windows用戶

`[user@urlinux examples]$ ./addTwoNumbers < data.txt > output.txt`
`[user@urlinux examples]$ cat output.txt`
`193`
`[user@urlinux examples]$`
	`C:\example> addTwoNumbers < data.txt > output.txt`
	`C:\example> type output.txt`
	`193`
	`C:\example>`

看懂了嗎？透過 > 這個輸出重導向的符號，我們讓原本連接到終端機的stdout改為連接到一個檔案output.txt — 如此一來，在程式裡原本透過cout輸出給stdout的資料，其目的地就從終端機改成了output.txt檔案。讀者應該也已經注意到了addTwoNumbers > output.txt 這個指令就長得和 addTwoNumbers → output.txt一樣，好用又好記。

不過要特別注意的是，在使用 > 輸出重導向的符號時，作業系統會幫我們建立新的檔案，若是檔案原本已經存在則會被覆寫。如果不要覆寫，而是要接續既有的檔案內容，那麼就要使用另一個輸出重導向的符號 »，它會讓我們把新的輸出附加到檔案原有的內容後面(當然，若是檔案並不存在，» 還是會幫我們建立新的檔案)。請參考下面的例子：

Linux/Mac用戶	Windows用戶

`[user@urlinux examples]$ cat output.txt`
`193`
`[user@urlinux examples]$ ./addTwoNumbers < data.txt >> output.txt`
`[user@urlinux examples]$ cat output.txt`
`193`
`193`
`[user@urlinux examples]$`
	`C:\example> type output.txt`
	`193`
	`C:\example> addTwoNumbers < data.txt >> output.txt`
	`C:\example> type output.txt`
	`193`
	`193`
	`C:\example>`

由於在執行前output.txt檔案已經存在且保有上一次寫入的內容，所以這次的輸出就會附加在既有的內容之後，所以你會看到兩行的193。

經過上面的討論之後，相信讀者已經能夠理解I/O重導向是什麼意思，同時也已經學會如何使用I/O重導向的功能… 等等，現在才講完用<進行stdin的重導向，以及使用 > 與 » 進行stdout的重導向，不是還有一個stderr嗎？ 嗯，是的，讀者們果然都很細心，其實stderr的重導向也十分簡單，只要使用 2> 就可以了，這個出現在 > 輸出重導向符號前的數字2，就表示要進行第2個標準輸出渠道的重導向stdout當然是第1個標準輸出渠道，第2個當然就是各位懸在心上的stderr了。請先參考以下的程式，我們在程式裡利用cerr及clog輸出了一些訊息到stderr

```
#include <iostream>
using namespace std;

int main()
{
   cerr << "This is an error message" << endl;
   clog << "This is a log message" << endl;
}
```

請將這個程式編譯為errorAndLog可執行檔，並使用下列方法測試：

Linux/Mac用戶	Windows用戶
[user@urlinux examples]$ ./errorAndLog 2> errlog.txt [user@urlinux examples]$ cat errlog.txt This is an error message This is a log message [user@urlinux examples]$	C:\example> errorAndLog 2> errlog.txt C:\example> type errlog.txt This is an error message This is a log message C:\example>

好了，打完收工… 等等… 不是還有一個叫做Pipe的東西還沒講嗎？ 對哦，差點就忘了。

Pipe管線的意思，就是可以在兩個程式之間，將其輸入與輸出進行串接(當然也可以串接更多程式)，其使用方式非常簡單，讓我們再多寫一個程式來做示範：

```
#include <iostream>
using namespace std;

int main()
{
   int x;
   cin >> x;
   cout << (x*2) << endl;
}
```

這個叫做doubleIt.cpp的程式，先取得一個整數的輸入，再將它變成2倍以後加以輸出。請自行完成它的編譯，並把可執行檔命名為doubleIt接下來，讓我們使用 | 符號，將原先的addTwoNumbers程式與這個doubleIt程式串接起來：

Linux/Mac用戶	Windows用戶
[user@urlinux examples]$ ./addTwoNumbers | ./doubleIt 386 [user@urlinux examples]$	C:\example> errorAndLog 2> addTwoNumbers | doubleIt 386 C:\example>

上面的做法，使用 | 符號，將addTwoNumbers的輸出串接到doubleIt做為其輸入，所以addTwoNumbers的輸出193，就變成了doubleIt的輸入，所以它把193乘以2後輸出386；也就是說，我們實現了將某個程式的輸出視為是另個程式的輸入，從此以後，我們所開發的一個一個小程式，就能夠串接組合出更多變化、實

現更複雜、但具有分工合作特性的應用功能。

好了，真的打完收工了，下回見。

---

[1] Stream Extraction Operator除譯做串流擷取運算子之外，亦有譯做串流「提取」運算子。
[2] Delimiter是分隔符的意思，是常見的資訊術語之一
[3] iomanip的命名是取自IO Manipulator