第 8 章

函式

函式 (function) 就是將一些經常須要重複使用的程式碼從主函式 (main) 中分離出來自成一個程式區間，這些被分離出來的程式碼通常有其特殊功用，可以隨時等著被呼叫執行，因此函式可以用來避免相同的程式碼一再重複地出現，進而降低程式的複雜度。此外也由於函式的程式碼被獨立出來，也使得維護與修改程式碼的工作會變成較為輕鬆。

程式語言中的函式有如數學上的函數一般，可以根據某種演算方法計算出所傳入資料的對應值。舉例來說：y = sin(x)，由這個簡單的數學式子也透露一個函式應該包含 (1) 使用名稱 sin (2) 傳入函式的參數資料 x (3) 演算的程式碼，用來計算 sin(x) 的數值 (4) 回傳資料，用來將 sin(x) 的數值傳給 y。

一個 C++ 函式就是藉由函式傳遞的參數與回傳資料來跟呼叫的程式碼產生互動，本章將仔細說明 C++ 所定義的函式語法及其相關的應用。

8.1 函式基本型式

首先來看看函式的基本型式：

函式基本概念

早期使用函式的目的只不過是為了將一再重複的程式碼獨立出來，簡化程式，以方便程式碼的維護與修改，例如：以下為某一中型程式的程式片段，其中某幾行敘述須要計算浮點數的次方數值：

```
p = 1.0 ;                                    // 程式碼的第 27 行
for ( i = 0 ; i < 3 ; ++i ) p *= 2.0 ;
cout << "2.0 的 3 次方 = " << p << endl
...
p = 1.0 ;                                    // 程式碼的第 513 行
for ( i = 0 ; i < 4 ; ++i ) p *= 3.0 ;
cout << "3.0 的 4 次方 = " << p << endl
```

```
...
p = 1.0 ;                                    // 程式碼的第 783 行
for ( i = 0 ; i < 6 ; ++i ) p *= 2.0 ;
x = p * p - 3 * p + 2 ;
...
```

同樣計算次方程式片段若一再地重複出現於整個程式碼，則整個程式碼將非常零亂，難以維護。但如果將之取出成為一個獨立的程式區間，並讓其使用方式如同數學的函數一樣，譬如 power(a,n)，專門用來計算 a^n 的數值，a 與 n 為輸入函數的自變數 (independent variable)，而函數的計算結果由 power(a,n) 傳出，則以上零散的程式碼就可以變成：

```
cout << "2.0 的 3 次方 = "                   // 程式碼的第 27 行
     << power(2.,3) << endl ;
...
cout << "3.0 的 4 次方 = "                   // 程式碼的第 513 行
     << power(3.,4) << endl ;
...
p = power(2.,6) ;                            // 程式碼的第 783 行
x = p * p - 3 * p + 2 ;
...
```

如此整個程式碼變得簡潔一些，我們將獨立出來的程式碼稱為函式。讀者可以由此猜出，一個函式的基本架構至少包含了函式名稱，傳入於函式的資料，函式計算後的回傳資料，最後當然還須包含函式內部的程式碼。

函式基本架構

依照程式撰寫順序，一個基本的函式型式依次包含：回傳型別，函式名稱，參數列與函式的程式主體。例如：以上的 power(a,n) 若寫成函式，其型式如下：

```
// 函式計算浮點數 a 的 n 次方，這裡 n >= 0
double  power( double a , unsigned int n ) {
   double  p = 1.0 ;
   for ( int i = 0 ; i < n ; ++i ) p *= a ;
   return  p ;
}
```

這裡的第一個 double 為函式計算後回傳資料的型別，簡稱為回傳型別 (return type)，之後的 power 為函式的名稱，在小括號內的資料為傳入函式內使用的參數列 (parameter list)，可以包含由函式外部傳入函式內部使用的資料，或者是由函式內部傳出函式外部的資料，緊接其後由大括號所構成的區間為函式的程式區間，為函式用來計算的程式碼。為方便了解起見，我們將依照執行的先後依次介紹。

函式名稱

函式的名稱是認知函式的第一步，有時根據函式名稱就可以清楚地了解函式的作用。函式的命名方式與變數名稱的方式一致，一般來說，函式名稱要讓人可以馬上知道其功能與作用。若函式名稱包含許多個字，則之間可以使用底線連接，或者可以讓英文字的第一個字母以大寫方式書寫，以便於分辨。函式的名稱不可以與一般的變數或物件名稱相同，當然這也不是一個好主意。

參數列

函式的參數列控制著函式如何與外界程式碼溝通，為函式與外部程式碼的介面，資料可以透過參數列傳入函式內，經過更改變動後再予以傳出。一個參數列可能包含許多參數 (parameter)，各參數之間以逗點隔開，每一筆參數的語法包含了型別與名稱，例如：在以下的參數列中，第一個參數為浮點數 a，第二個參數為無號整數 n，

```
double  power( double a , unsigned int n ) {
    ...
}
```

參數列的參數是分別用來接收傳入的資料，例如：當使用了

```
double no = 2.0 ;
cout << power(no,4) << endl ;   // no 的 4 次方，no 為 2.0
```

則 power 函式會依據參數列中參數宣告的次序來執行個別參數的設定，也就是函式在開始執行時就會分別設定參數，以上為例，相當於在參數列中執行了：

```
double       a = no ;           // no 為 2.0
unsigned int n = 4  ;
```

由此也可以看出，no 與整數 4 是透過複製方式將其值傳入函式內使用。在函式的程式區間，是完全不知 no 與 4 的存在，而是其替身 a 與 n，也因此參數列被視為函式外部資料與函式內部資料的交換介面。

如果函式並不須要傳入或傳出任何資料，則參數列可以省略不寫或者是使用 void，即

```
int  foo()     ;   // 沒有參數列
int  foo(void) ;   // 沒有參數列
```

函式主體

撰寫一般函式的方式與主函式相同，都可以擁有函式自己所使用的變數與資料。函式開始執行時，函式所能使用的資料就是在參數列的所有參數與函式本身所定義的變數。當函式執行結束後，這些參數與變數就會全部消失。若程式再次執行函式時，則

函式內的參數與變數又會重新產生。有關函式內的參數，變數的產生與消失，若由變數的存在領域[43]來看，一切就很清楚。所有在函式內所宣告的變數在離開了函式的程式區間後，也就是其存在領域，會自動的消失，如此可以使得作業系統得以靈活地運用有限的記憶空間資源。

回傳型別

每個函式都可以在函式結束之前回傳一筆資料給呼叫的程式碼，這筆資料的型別稱為函式的回傳型別。使用者隨時可以在函式內的某敘述利用 return foo 的方式，將資料 foo 回傳出去，此時回傳資料的型別即為函式回傳的型別，例如在次方函式中：

```
double  power( double a , unsigned n ) {
    double p ;
    ...
    return p ;      // 回傳 p 的值
}
```

return 回傳敘述並不見得是要為函式中的最後一行，可以出現在函式內的任何一行，同時也可以根據須要出現若干次，例如以下為計算絕對值的函式：

```
double  abs( double a ) {
    if ( a >= 0 )
        return a ;
    else
        return -a ;
}
```

當然，函式的回傳型別都須要相同。如果函式不須要回傳任何資料，則可以將函式的回傳型別定義成空型別，用 void 來表示，這時函式的函式碼 return 之後就不須加上任何資料，例如：

```
// 依整數大小次序列印兩整數
void  print_max( int a , int b ) {
    if ( a >= b ) {
        cout << a << ' ' << b ;
        return ;
    } else {
        cout << b << ' ' << a ;
        return ;
    }
}
```

若函式不須回傳任何資料，也可以將 return 敘述省略，這時函式會執行到其最後一行敘述後結束。

8.2 函式的原型與特徵

對函式的使用者而言，了解函式介面與其回傳的資料遠比了解函式內部的程式碼如何運作來重要。就如對一般人而言，知道如何操作電視搖控器來觀看電視節目遠比了解電視機的顯像原理來的重要。

函式原型

在 C++ 編譯的過程中，若程式碼的某個敘述須要使用某函式，編譯器一定要確認函式已經存在。所謂存在，是指函式在之前的編譯過程中已經出現，也就是說，C++ 編譯器要先確認函式已經出現，則之後的程式碼才能呼叫使用。如此函式須要擺放在使用的程式碼之前。由於程式是由主函式起開始執行，因此所有的函式都須要擺放在主函式 main 之前，例如：

```
// 函式 abs 回傳整數的絕對值
int abs( int a ) {
    return ( a < 0 ? -a : a ) ;
}

// 呼叫函式 abs 的程式碼
int main() {
    ...
    cout << abs(-5) ;
    ...
}
```

通常一個程式碼會定義若干個函式，函式之間也可以相互呼叫使用，可能會有某 a 函式使用到 b 函式， b 函式使用到 c，d 兩函式，一層一層的使用下去，由於函式須要擺放在使用的程式區間之前，在越底層的函式就越須要定義在程式檔案的前端， 如此一來，程式中所有使用的函式通通會被擠到程式檔案的前面，而最後一個出現的函式才是整個程式碼的主函式 main，主函式於程式檔案的遙遠後端出現，很容易造成使用者對程式的整體執行流程失去焦點，增加維護程式的困難。為避免這種現象的產生，我們可以將函式的設計分成兩部份，一為函式的原型部份 (prototype)，一為函式的真正定義的部份，前者是用來告知編譯器有某函式的存在，後者則是此函式的內容。所謂「函式原型」是指去除程式碼的函式部份，也就是函式的標頭部份，以商品來比喻，前者如同商品的目錄，用來告知消費者商品的存在，作用相當於宣告函式的存在，而後者則為商品的實體，相當於定義函式。

程式設計員只要將函式的原型寫在使用的程式敘述之前即可，而將函式真正的內部程式碼通通移到程式檔案的後面。由於函式原型通常是短短的一行，如此主函式 main 相對的就會挪到程式檔案的前面，這樣的處理方式可以使得他人可以很快的了解整個程式架構，因此以上的例子可以改寫成：

```
// 函式 abs 原型
```

```
int abs( int ) ;

// 呼叫函式 abs 的程式碼
int main() {
    ...
    cout << abs(-5) ;
    ...
}

// 函式 abs 程式碼
int abs( int a ) {
   return ( a < 0 ? -a : a ) ;
}
```

去除程式碼的函式原型 (prototype) 包含了函式的回傳型別，函式名稱與函式的參數列，例如：

```
// (1) 函式回傳輸入兩整數的 a 與 b 的最大值
int     max( int a , int b )   ;

// (2) 函式回傳輸入的美金所能對換的臺幣金額
double  NT( double US_dollar ) ;

// (3) 函式回傳輸入浮點數的整數次方值
double  power( double , int )  ;
```

在撰寫函式的原型中，參數名稱並不是必要的，可以省略，但有時可以加以保留用以增加函式的可閱讀性，如第 (2) 例。

一個函式可能有上百行的程式碼，但函式的原型對多數而言通常是一行，頂多兩三行。使用者只要看到短短的函式原型，就可以知道程式某函式的存在，同時也可以知道如何使用函式，因為一個函式原型就包含了函式名稱，須要使用的參數型別，及回傳的資料型別，當然函式的更進一步細節可能仍須查閱註解才能了解。

對程式設計者而言，撰寫函式的原型並不是必要的，尤其對幾十行的小程式而言，可能是多此一舉。但對成千上萬行的大程式而言，在程式碼的前端標明程式中所有使用到的函式原型，可以讓使用者很快的得知程式碼有哪些函式存在，及各個函式所使用的參數列與回傳的資料型別。其用途如同在大型會議場的簽到簿一樣，一般人只要查閱簽到簿上的名字就可以知道所有出席會議的人士，而不須要到會場上費力地搜尋才能將出席會議的所有人士找出，如此一來可以省下許多時間。

函式原型如同名片一樣並不能代表函式本身，僅是用來告知某函式的存在，因此函式的原型可以依據須要重複地撰寫在多個不同地方。以在這方面來看，函式原型相當於函式的宣告 (declaration)，而函式內部真正的程式碼則相當於函式的定義 (definition)，這與變數的宣告與定義作用類似[23]。

函式特徵

C++ 辨別某一函式是否與另一函式相同，是藉由比對函式的特徵資料 (signature) 是否相同來作判斷。所謂的函式特徵是指函式的名稱與其所用的參數列，這裡的參數列資料包含所有的參數總數與其各自的資料型別。 C++ 禁止兩函式有相同的特徵。

由於函式名稱僅是其特徵的一部份，兩函式若名稱相同，但參數列不同，仍會被視為不一樣的函式，可以各自獨立存在，例如：

```
// (1) 找出兩整數的最大值
int     max( int a , int b ) ;

// (2) 找出三浮點數的最大值
double  max( double a , double b ) ;

// (3) 找出三整數的最大值
int     max( int a , int b , int c ) ;
```

以上三個函式的特徵都不相同，(1) 與 (2) 是因為參數型別不同，(1) 與 (3) 則是參數總數不同。因此三個函式都可以被定義使用。

由上，函式的特徵僅是其原型的一部份，若兩個函式的原型中只有回傳的型別不同，則 C++ 會因兩函式的特徵相同，在編譯過程就會產生錯誤訊息。例如：以下兩個絕對值函式的原型雖然不同，但函式特徵一樣，C++ 無法分辨兩者之間的差別，而造成編譯錯誤。

```
// 輸入一整數，以整數方式回傳其絕對值
int     abs( int a ) { return ( a > 0 ? a : -a ) ; }

// 輸入一整數，以浮點數方式回傳其絕對值
double  abs( int a ) {
    return ( a > 0 ? static_cast<double>(a)
                   : static_cast<double>(-a) ) ;
}
...
int main() {
    cout << abs(-3) << endl ;   // 錯誤 : C++ 無法選擇
    ...
}
```

8.3 參考資料型別

函式的參數列介面可以容許輸入的資料經過函式的處理後更改其內容輸出，也可以防止輸入的資料值遭受無意中更改，以確保輸入資料的安全性。C++ 特別定義了一些輸出入機制來處理資料的傳送。首先我們先來介紹一種新的資料型態：參考 (reference)。假設有一程式碼如下：

```
int  foo  = 3   ;
int  &bar = foo ;   // bar 為 foo 的參考
```

這裡在指定運算子的左邊，整數 bar 之前有個 & 符號用來表示變數 bar 是 foo 的一個參考。所謂 bar 是 foo 的一個參考是指著使用 bar 就跟使用 foo 一樣，兩者都代表同一筆資料，就好像一個信箱上貼著兩個名字一般。有時候，使用別名 (alias) 更能表達出其涵義，以口語來說，若原始變數為本尊，則參考變數即為分身，兩者都代表相同變數。如此一來，本尊 foo 的資料經過指定參考後，可以藉由分身 bar 來改變。例如：

```
bar = foo + 4 ;          // 相當於 foo = foo + 4
bar = bar * bar + foo ;  // 相當於 foo = foo * foo + foo
```

所有的參考出現的地方皆可以視為本尊的運算。由於分身是依附著本尊，在定義參考時一定要設定本尊，不能先定義分身，然後再去找本尊。所以以下的程式碼是錯誤的：

```
int   foo ;
int  &bar ;         // 錯誤，參考一定要設定參照對象
bar = foo ;         // 錯誤
```

若在參考之前加上 const 則表示程式中不能透過參考 (分身) 來更改原始變數 (即本尊) 的數值，但原始變數仍可自行更改其數值。例如：

```
int         foo = 3   ;
const int  &bar = foo ;   // 分身 bar 為 本尊 foo 的參考常數
bar++                 ;   // 錯誤 , 分身 bar 被視為常數
foo++                 ;   // 正確 , 本尊 foo 仍可修改。
```

在正常的情況下，於同一個程式區塊內使用參考修改變數值容易造成混淆，且多此一舉。但在函式參數列的參數資料傳遞卻是以參考型式為最常用的一種方式，這將會在下面加以說明。

參考與指標

參考與指標雖然在使用方式相差甚多，但它們都有類似的作用，即是兩者都可以用來間接地存取另一個變數資料值，也就是說，兩者的背後都隱藏了一個記憶空間可以用來儲存資料，因此都可以用來當成運算子的左值[25]使用，例如：

```
int  a = 10 , b = 20 ;

int *foo      ;   // 定義整數指標
foo  = &a     ;   // 指標 foo 指向 a
*foo = 30     ;   // 相當於 a = 30
foo  = &b     ;   // 指標 foo 指向 b
```

```
int &bar = a  ;   // bar 為 a 的參考
bar  = 30     ;   // 相當於 a = 30
&bar = b      ;   // 錯誤，bar 不能更改所參考的變數
```

很明顯的，在存取參照空間的資料時，參考在使用上較指標方便許多，這也是參考較受人歡迎的原因。

參考可以被看成一個不能更動指向位址的常數指標[96]，參考變數在定義時就須設定所要參考的變數，且之後不能更改，但一般的指標卻可以自由改變所指向的記憶空間位址，可以使用 +，-，= 等運算子自由地改變指標所指向的位址，但如果改變位址後的指標更改到其它程式所使用的記憶空間時，程式就會立即中斷執行，這是使用指標的危險性。然而參考卻無此顧慮，因為參考禁止在程式中更改所參考的對象，如此一來，程式會不會更動所參照的位址也就是選擇使用參考，或者是指標的一個簡單標準，例如，以下就只能使用指標方式來間接的更改陣列的元素值：

```
int  no[5] = { 2 , 4 , 6 , 1 , 7 } ;
int *p = no ;                          // 指標 p 指向陣列首位元素

// 藉著更改 p 的指向位址讓 no 陣列的每個元素加 3
for ( int i = 0 ; i < 5 ; ++i , ++p ) *p += 3 ;
```

在以上的迴圈判斷式中，如果讓 i < 5 改成 i < 1000，則程式可能由於更改到其它程式所使用的記憶空間而被迫中斷執行，這種問題是無法在程式的編譯過程中被偵測出來，總是等到實際執行時才會因錯中斷，這是使用指標的潛在危險。

指標的參考

參考除了用於基本資料型別外，也可以用在指標上，例如：

```
int   a = 3  ;
int*  p = &a ;      // p 指標指向 a 整數
int* &q = p  ;      // q 為指標 p 的參考
```

注意以上的撰寫型式，定義指標的參考時，須將 & 符號放置於 * 符號的右側。以上的定義除了設定 q 是一個整數指標外，同時也讓 p 指標多了一個新名稱 q。因此在程式碼中，使用 q 與使用 p 就沒有差別，也就是 q 與 p 代表同一個指標，例如：

```
int    a = 3 ;
int *  p = &a ;     // p 為一整數指標，指向整數 a
int * &q = p ;      // q 為 p 指標的參考
q = new int(5) ;    // q 指向一動態整數空間，內存 5
cout << *p ;        // 列印 5
```

以上當 q 指向新的動態空間時，p 也同時指向此新空間。如果去除以上 q 之前的 & 符號，則 p 與 q 兩者就為各自獨立的指標，例如：

```
int    a = 3 ;
int *  p = &a ;      // p 為一整數指標, 指向整數 a
int *  q = p ;       // q 也指向 p 指標所指向的位址
q = new int(5) ;     // q 指向一動態整數空間, 內存 5
cout << *p ;         // 列印 3
```

以上 int * q = p 敘述設定 q 為一新指標，只是剛好指向 p 所指向的位址。因此之後如果 q 指向其它位址時，其作用與 p 無關。

若要定義雙層指標[103]的參考，其語法也是類似，例如：

```
int **p ;            // p 為一雙層指標
int ** &q = p ;      // q 也為一雙層指標, 為 p 的另一個名稱

q = new int*[5] ;
for ( int i = 0 ; i < 5 ; ++i ) q[i] = new int[4] ;
```

以上末兩行讓雙層指標 q 指向一 5×4 的二維動態空間，其效果等同使用 p。同樣的，如果將第二行的 & 符號去除，則 p 與 q 兩者各為獨立的雙層指標，互不相干，如此末兩行的雙層指標 q 作用就與 p 無關。

在正常的情況下，程式碼中使用指標的參考似乎是多此一舉的。事實上，使用參考的主要目的是要方便外部資料可以透過函式參數傳入函式內使用與更改，這將在以下加以說明。

8.4 參數傳遞

函式透過參數列的參數與外界程式碼交換資料，基本上，函式在參數列定義參數的方式與一般程式碼定義變數的方式相當。一般來說，函式參數資料的傳遞模式，主要可以分為三種：

傳值方式

這是最簡單的傳遞資料方式，是將外部的數值資料「複製」到函式內部使用，所謂傳值方式 (passed by value)，可以由以下簡單的例子了解，例如：

(a) 呼叫的程式碼

```
int i = 3 , j = 5 ;
int k = max(i,j)  ;
cout << k << endl ;
```

(b) 函式碼 : 計算兩參數的最大值

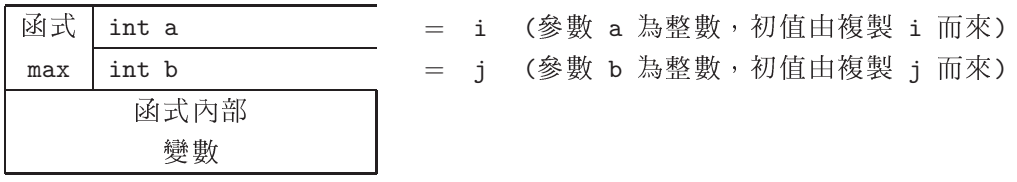
```
int max( int a , int b ) {
    return a > b ? a : b ;
}
```

在左邊呼叫的程式碼部份，輸入函式的資料為兩個整數，即 i 與 j，這兩個整數分別被當作右邊函式參數列的初值。若單獨將函式的參數列取出來看，當函式剛開始執行時，參數列會依序執行以下設定，

```
int a = i ;   // 將整數 i 的值複製給參數 a
int b = j ;   // 將整數 j 的值複製給參數 b
```

參數 a 與 b 是屬於函式在執行時所定義的內部變數，也就是說，a 與 b 可以在函式內使用，而傳入的整數 i 與 j，是屬於函式外部的變數，當其值被傳給參數後，i 與 j 就不能在函式中被使用了。由上面的程式碼看來，在函式中，a 的初值是由 i 複製而來，b 的初值是由 j 複製而來，如此，a 與 i 或者是 b 與 j 都佔有不同的記憶空間。如果在函式中，a 或是 b 的值被改變了，當然不會影響原來 i 與 j 的值，由於 a 與 b 的存在領域僅在函式之內，當函式執行結束後兩參數就自動消失。

以傳值來傳遞參數的重點在於資料是以複製的方式傳入函式，這類型的傳遞參數方式，對傳入的外部變數來說，由於是以資料複製，並不會被函式所更改，資料最為安全。但每次執行函式時，函式都要另外找尋新的記憶空間來儲存複製的參數資料，同時受到參數存在領域的限制，參數所佔用的記憶空間在函式執行結束後，也會自動消失。如此一來，執行速度當然會受到影響。我們可以用下圖來表示，在圖中的右上角，為函式的參數部份，能與輸入的變數產生互動，而函式內部所定義的變數則為函式所獨有，不會與函式外部資料有任何關聯。



指標傳遞方式

程式外部變數若是以資料複製的傳值方式傳入函式， 則原來的資料絕對不會受到執行函式的干擾或更改，因此保有資料的安全性。但有時我們希望函式可以將輸入的數值加以改變，在這種情況下，C++ 提供兩種方式可以改變輸入的資料值，第一種方式就是使用指標參數來傳送資料，由於指標是用來儲存其它同型別變數的位址，所以輸入的變數值應為變數的位址，如下例的對調函式 swap 可以將兩個輸入值對調。

(a) 呼叫的程式碼

```
int i = 3 , j = 5 ;
swap(&i,&j) ;

// 列印 : 5 3
cout << i << ' ' << j ;
```

(b) 函式碼 : 對調參數

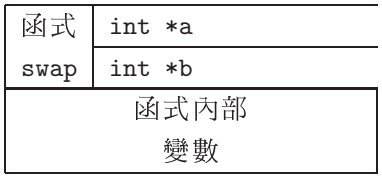
```
void swap( int *a , int *b ) {
    int tmp = *a ;
    *a = *b ;
    *b = tmp ;
}
```

以上的程式有兩個地方要特別注意。其一為呼叫函式時，傳入的資料為變數的位址，如左邊呼叫的程式碼 swap(&i,&j)，其二是在函式中，由於參數為一指標，如果是存取指標所指到的資料，則須要使用參照運算子 (dereference operator)，指標參數的設定相當於執行了

```
int *a = &i ;   // 指標參數 a 指向整數 i 的位址
int *b = &j ;   // 指標參數 b 指向整數 j 的位址
```

事實上，這兩個敘述也是利用複製的方式複製資料，只不過是複製兩個輸入的整數變數位址，並非其數值。因此使用指標參數傳遞的方式也被稱為傳址方式 (passed by

address)，如此一來，函式透過指標就可以間接的更改存在函式外部記憶空間的資料值，整個指標參數傳遞的方式可以用以下的圖形來表示：



須注意，在使用指標傳遞方式中，資料仍是使用複製的方式傳入，只是被複製的資料為傳入變數的位址，不是變數的內容資料。由於不管多複雜的變數資料，其位址所佔用的記憶空間大小都是固定的，等同一個整數，因此在執行效率上，以指標方式傳遞任何型別變數到函式內所須的時間相當於傳入一個整數的時間。與傳值方式比較，若傳入的變數資料相當複雜，則複製所要花費的時間自然就相當可觀，因此使用指標傳遞在執行效率上通常會較高。

使用指標來傳遞資料，很容易在呼叫函式時忘了寫上位址運算子，即 `&`，同時在函式內部做數值運算時，也須要時時記得將參照運算子加於指標之前，否則很容易發生執行上的錯誤。

這種使用指標傳入變數位址的方式，可以容許函式透過指標來更改變數的資料值，但也可以在參數之前加上 `const` 字樣禁止指標來改變指向的資料，例如：以下的 `length` 函式是單純地計算輸入字串的長度，並不想要利用函式來改變輸入的字串，

```
int  length( const char * str ) {
    int i = 0 ;
    while ( *(str+i) != '\0' ) ++i ;  // *(str+i) 相當於 str[i]
    return i ;
}

int main() {
    char p[] = "dog" ;
    cout << length( p ) << endl ;      // 列印 p 字串長度
    ...
}
```

以上參數列的設定是執行了：

```
const char *str = p ;   // 字元指標參數 str 指向 p 陣列首位位址
```

由於 `const` 是寫在參數型別之前，因此在函式內的字元指標參數 `str` 無法用來改變所指向的字元資料，如此可以用來確保傳入資料不會被函式的指標參數間接地變更，確保資料的安全性。但請注意，這裡的指標參數仍然可以自由地改變其所指向的位址。

有時候程式設計員須將常數字串輸入到某函式內使用，但函式的參數型別卻不是常數型別，例如：

```
int  length2( char * str ) {
    int i = 0 ;
    while ( *(str+i) != '\0' ) ++i ;  // *(str+i) 相當於 str[i]
    return i ;
}

int main() {
    const char captain[] = "Picard" ;
    cout << length2( captain )        // 錯誤, 參數不須為常數型別
         << endl ;
    ...
}
```

這時如果仍要堅持使用此函式，就須要暫時將 captain 的常數去除，為此 C++ 提供了 const_cast 的方式可以達到此目的，以上為例則須改成：

```
int main() {
    const char captain[] = "Picard" ;
    cout << length2( const_cast<char*>(captain) )
         << endl ;
    ...
}
```

當然這時程式設計員要自行負責常數 (const) 去除的後果。

參考型別傳遞

函式參數傳遞的第三種方式為使用參考型別來傳遞，也就是傳遞參考 (passed by reference)，這是最常被使用的參數傳遞方式，若以上面的對調函式為例：

(a) 呼叫的程式碼

```
int i = 3 , j = 5 ;
swap(i,j) ;

// 列印 : 5 3
cout << i << ' ' << j ;
```

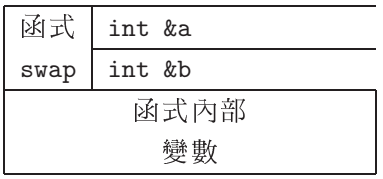
(b) 函式碼 : 對調參數

```
void swap( int &a , int &b ) {
    int tmp = a ;
    a = b ;
    b = tmp ;
}
```

這裡參數列的參數設定是執行了

```
int &a = i ;   // 參數 a 為變數 i 在函式內的參考
int &b = j ;   // 參數 b 為變數 j 在函式內的參考
```

呼叫的程式碼將變數 i 與 j 分別傳給其參考 a 與 b，相當於，變數 i 在函式內的分身為參數 a，變數 j 在函式內的分身為參數 b，當 a 與 b 的值在函式內更改了，則 i 與 j 的數值也會同時改變了。由此可知，資料以參考方式傳入函數內，函式可以間接的更改其值，這也是使用參考型別的主要目的，參考參數傳遞可以使用以下的圖形來表示：



由此函式可以透過參考參數來改變傳入變數的資料， 但相較於之前的指標方式，傳遞參考方式來得方便，程式不須像指標一樣，須要費力地在其前加上參照運算子才能變更或使用資料。若與第一類的傳值方式比較，由於參考參數並沒有複製變數資料，當然執行上較有效率。但參考參數卻可能會更動了不想被改變的值，故資料的安全性可能較差。然而這個缺點可以仿照指標常數的模式使用參考常數 (constant reference) 來克服，避免函式透過參考來變更資料數值，例如，以下 (A)，(B) 分別利用傳遞參考與指標的方式來計算兩數的最大值：

```
(A) max1 傳遞參考
int  i = 3 , j = 5 ;      int  max1( const int& a , const int& b ) {
int  k = max1(i,j) ;           return  a > b ? a : b ;
cout << k << endl ;       }
```

```
(B) max2 傳遞指標
int  s = 3 , t = 5 ;      int  max2( const int *a , const int *b ) {
int  r = max2(&s,&t) ;         return  *a > *b ? *a : *b ;
cout << r << endl ;       }
```

這兩種型式都可以用來避免函式意外地更動傳入資料的可能性，同時也可以避免複製傳入資料所造成的效率降低。也就是說，第一類型的參數傳值方式可以使用後兩種的常數型方式來替代。比較以上兩種參數的使用方式，很清楚地可以發現使用參考參數來得簡便，因其不像指標一樣，須要於指標變數之前加上參照運算子方式才能存取指向的資料。

參數傳遞方式的比較

以上所介紹的三種資料傳遞模式可以整理成表 8.1 來比較其間的優劣性， 由此表可

傳遞方式	空間使用	執行效率	資料安全	語法容易
數值	劣	劣	優	優
指標	優	優	優*	劣
參考	優	優	優*	優

表格 8.1: 函式參數傳遞的比較。優*為使用常數型的指標或者是參考

以看出使用參考方式傳遞資料的優異性，指標傳遞參數的模式幾乎都可以使用參考方

式來取代，但若傳入的資料型別本身就是指標時，則就須要使用指標參數來處理，例如：

```
// 將字元指標所指向的字串小寫字元轉成大寫並將其回傳
char *  to_upper ( char * str ) {
    for ( int i = 0 ; str[i] != '\0' ; ++i )
        if ( str[i] >= 'a' && str[i] <= 'z' ) str[i] += 'A' - 'a' ;
    return str ;
}
...
char  p[] = "dog" ;
cout << to_upper(p) << endl ;   // 將字串內的字元轉成大寫後列印 DOG
```

傳遞陣列

若須要將整個陣列的所有元素傳入函式中處理，這時可以透過指標的方式複製陣列的首位元素位址，然後利用指標運算來存取陣列元素。例如：

```
int  a[] = { 1 , 3 , 5 , 7 } ;
int   *p = a ;                    // 指標 p 指著 &a[0] 的位址
cout << *(p+2) << endl ;          // 列印 a[2] 的值
```

這裡要留意：以上的指標只記住陣列的首位元素的位址，無法得知陣列的長度。當陣列傳入函式中，也須將陣列的長度當作參數傳入函式中，例如以下的函式是用來計算陣列元素的總和。

(a) 呼叫的程式碼

```
int  a[3] = { 9 , 7 , 5 } ;

// 列印 a 陣列元素總和
cout << sum(a,3) << endl ;
```

(b) 函式碼 : 計算陣列元素總和

```
int sum( int *p , int n ) {
    int i , sum = 0 ;
    for ( i = 0 ; i < n ; ++i )
        sum += *(p+i) ;
    return sum ;
}
```

這裡函式 sum 的參數列設定相當於執行了：

```
int  *p = a ;   // 整數指標參數 p 指向 a 陣列的首位元素
int   n = 3 ;   // 整數參數 n 設為 3
```

如在第 5.4 節內說明，以上右式的指標運算方式可使用下標運算子較為方便，即 sum += p[i]。當然程式設計員一定要牢牢記得不能改變不屬於陣列元素的記憶空間。上式也可直接使用類似陣列的型式來當參數，例如：

```
int sum( int p[] , int n ) {
    int  i , sum = 0 ;
    for ( i = 0 ; i < n ; ++i ) sum += p[i] ;
    return  sum ;
```

```
}
```

這裡的 p[] 看起來似乎像陣列，但事實上，也是被當成指標看待，p 並無法預知輸入的陣列到底有多少個元素個數。陣列參數內也可以填上陣列大小，不過沒有什麼用處，C++ 也只是將它當成指標，指標並無法記得陣列大小，所以使用這種型式反而會造成誤導，例如：

```
int sum( int p[10] ) {
    int  i , sum = 0 ;
    for ( i = 0 ; i < 10 ; ++i ) sum += p[i] ;
    return  sum ;
}
```

事實上，以下三種函式的原型型式都被視為相同，

```
int  sum( int *p ) ;
int  sum( int p[] ) ;
int  sum( int p[10] ) ;
```

當傳遞陣列時，使用者通常須要將陣列的長度一起當做參數輸入，藉以防止函式更改到非陣列元素所佔用的記憶空間，而引起執行上的錯誤。由於傳入的陣列在函式中是以指標方式處理，透過指標，可以改變原始陣列的數值，這可能會影響資料的安全性，所以若要確保輸入的陣列不會在函式內被更改，這時可以使用指標常數，例如之前的陣列和函式可以改寫成：

```
int sum( const int * p , int n ) {
    int  i , sum = 0 ;
    for ( i = 0 ; i < n ; ++i ) sum += p[i] ;
    return  sum ;
}
```

如此可以確保傳入的陣列資料不會在函式內被意外的更改。如果要傳遞字串陣列到函式內，也須要將陣列的長度當成參數傳入。

```
void show_string( char * str[] , int no , char * sep ) {
    for ( int i = 0 ; i < no-1 ; ++i ) cout << str[i] << sep ;
    cout << str[no-1] << endl ;
}

...

char *  startrek[] = { "Space" , "The" , "Final" , "Frontier" } ;

// 印出 : Space--The--Final--Frontier
show_string( startrek , 4 , "--" ) ;
```

若要傳送二維以上的陣列到函式內時，在寫法上可以將多維陣列的第一維長度加以省略，例如：

```
// 計算一整數陣列內的所有元素之和
int sum( const int  data[][3] , int r , int c ) {
    int  i , j , sum = 0 ;
    for ( i = 0 ; i < r ; ++i )
        for ( j = 0 ; j < c ; ++j )
            sum += data[i][j] ;
    return  sum ;
}
...
int  foo[2][3] = { {1,2,3} , {4,5,6} } ;
cout << sum( foo , 2 , 3 ) << endl ;            // 印出 : 21
```

注意，這裡的 data[][3] 與以下型式相當：

```
int (*data)[3] ;        // 指向三個整數區塊的指標
```

為一個指向 3 個整數的指標，其與以下的定義方式所代表的涵義不同：

```
int * data[3] ;         // 可儲存三個指標元素的陣列
```

上式的 data 為有三個元素的陣列，而每個元素皆是指標。同樣的，指標並不記錄矩陣的行列數量，在運算上通常須要將矩陣的列數與行數當參數傳入。在以上的程式中，陣列參數之前加上 const 是用來表示函式並不會更動陣列內的資料。採用相同的方式，以下的程式碼透過函式來設定三維陣列的元素值：

```
// 將三維陣列的所有元素都設定為 val
void  setup_array( int no[][3][4] , int a , int b , int c ,
                   int val ) {
    int i , j , k ;
    for ( i = 0 ; i < a ; ++i )
        for ( j = 0 ; j < b ; ++j )
            for ( k = 0 ; k < c ; ++k )
                no[i][j][k] = val ;
}
...
int  data[2][3][4] ;
setup_array(data,2,3,4,1) ;  // 讓陣列元素皆設為 1
```

以上的陣列參數的第一維大小可以省略，同樣的，陣列參數也可寫成指標的方式：

```
void  setup_array( int (*no)[3][4] , int a , int b , int c ,
                   int val ) ;
```

有關多維陣列與指標間的關係可參考第 139 頁的內容

傳遞指標參考

在許多應用上，指標有時必須以參考的方式傳入函式內使用，也就是參數是以指標的參考[183]型式傳入函式。舉例來說，如果我們須要撰寫一個函式可用來動態配置適當

大小的記憶空間，並可設定元素初值，則相關的程式碼可寫成以下型式：

```
void  setup( int * & p , int size , int val ) {
    p = new int[size] ;
    for ( int i = 0 ; i < size ; ++i ) p[i] = val ;
}
...

int  * ptr ;

// 讓 ptr 指標指向一塊包含 5 個整數的動態空間, 元素初值皆為 1
setup( ptr , 5 , 1 ) ;
```

注意，函式的第一個參數型式為指標的參考，用來設定函式內的 p 指標與函式外的 ptr 指標是同一個指標，因此當函式內讓 p 指標指向一新配置的動態空間時，其作用等同讓 ptr 指標指向此記憶空間。仔細觀察參數的設定，setup 函式的參數列相當於執行了：

```
int * & p = ptr ;  // p 指標為 ptr 指標的參考, p 為 ptr 在函式內的代表
int  size = 5 ;
int  val  = 1 ;
```

因此當函式內的 p 指標最後指向何處，在函式外的 ptr 指標也會指向同一個地方。若讀者去除在參數列中指標 p 之前的參考符號 &，則 p 指標與 ptr 指標就代表兩個各自獨立的指標，互不相干，只不過是 p 指標起初會指向 ptr 指標所指向的位址，但當函式執行時，指標 p 改以指向新的動態空間，此動作並不會影響到在函式外的 ptr 指標，如此一來，就不是我們設計 setup 函式的目的。此外當函式執行完畢後，p 指標隨即消失，新配置的空間位址就沒有任何指標儲存，自然造成了記憶空間流失[101]的問題。

同樣的，若要使用函式配置一個二維動態空間，也須以雙層指標的參考當成參數型式傳入函式內，例如：

```
void  setup_2d( int ** & p , int row , int col , int val ) {
    int  i , j ;
    p = new int*[row] ;
    for ( i = 0 ; i < row ; ++i ) {
        p[i] = new int[col] ;
        for ( j = 0 ; j < col ; ++j ) p[i][j] = val ;
    }
}
...

int ** ptr ;

// 讓 ptr 雙層指標指向 3x5 的二維動態空間, 所有元素初值皆為 1
setup_2d( ptr , 3 , 5 , 1 ) ;
```

須留意，雙層指標 ptr 是透過函式 setup_2d 的內部執行 new 敘述而取得二維動態空間，因此也要記得在適當的時候使用 delete 釋放此動態空間回作業系統。同樣的，如果讀者將以上參數列中的 int ** & p 改成 int ** p，則 p 與 ptr 兩者就為各自獨立的雙層指標，當函式執行完畢後，程式內並無任何指標指向這些新配置來的動態空間，因此記憶空間流失現象就必然會發生。

參數預設值

函式內所使用的參數也可以設定預設值 (default value)，也就是說，使用者如果不輸入參數的數值，則函式會以預設值當作參數的傳入值使用，若是使用者想自行設定參數值，則參數預設值的作用就被關閉。使用者若要設定參數預設值只要將其值直接寫在參數列即可，為了讓編譯器可以辨認使用與不使用預設值的參數，所有使用預設值的參數都要放置在參數列的末尾，在使用這些設有預設值參數的函式時，可以將這些參數由參數列中省略，此時 C++ 就使用參數的預設值當做參數的輸入值，例如：

```
// 計算最多三數的平方和
int square_sum( int a , int b = 0 , int c = 0 ) {
    return  a * a + b * b + c * c  ;
}
```

以上的平方和函式 square_sum，在使用時最少可以只輸入一個參數值，其他未輸入的參數則是以預設值當作參數的輸入值，因此：

```
// a = 2 , b = 0 , c = 0 , 輸出 : 4
cout << square_sum( 2 ) << endl ;

// a = 2 , b = 1 , c = 0 , 輸出 : 5
cout << square_sum( 2 , 1 ) << endl ;

// a = 2 , b = 4 , c = 8 , 輸出 : 84
cout << square_sum( 2 , 4 , 8) << endl ;
```

由於編譯器「不夠聰明」，使用者並無法只設定第一個參數的預設值，此時須要將第一個參數擺放到末尾成為最後一個參數，例如：

```
// 計算兩到三個正數的最大值
unsigned  max( unsigned a = 0 , unsigned b , unsigned c ) ; // 錯誤
unsigned  max( unsigned b , unsigned c , unsigned a = 0 ) ; // 正確
```

當某函式使用了預設參數後，就要避免設計另一個只有參數數量不同的函式，以免造成混淆，例如以下兩個函式的參數列好像不一樣，但若我們使用了 square_sum(3) 程式碼，則編譯器無從判斷該使用何者

```
double  square_sum( double a , double b = 0 , double c = 0 ) ;
double  square_sum( double a ) ;
```

使用者可以根據需要，將所有的參數都設定預設值，C++ 並沒有加以禁止，例如：

```
int  square( int x = 0 ) {  return  x * x ; }
```

如果要利用先宣告後定義的方式來設計函式時，若函式使用參數預設值，則參數的預設值僅可以寫在函式的宣告階段，即其原型，不能寫在函式的定義，例如：

```
// 函式宣告 : 可以加入參數的預設值
double  square_sum( double a , double b , double c = 0. ) ;
...
// 函式定義 : 不可加入參數的預設值
double  square_sum( double a , double b , double c ) {
    return  a*a + b*b + c*c ;
}
```

這樣的規定可以防止函式在宣告與定義階段不小心使用了不同參數預設值的情況發生，因而造成編譯器無所適從。

8.5 函式主體

當程式進入函式執行時，函式所能使用的資料來源主要為參數，在函式內定義的局部變數[44]，與已定義的全域變數[44]，當函式執行結束後，參數與局部變數就會全部消失，當下一次程式再次執行此函式時，這些變數才會重新產生，如此一來，珍貴的記憶空間資源才能被作業系統有效地分配運用。

靜態變數

有時使用者也想保留函式內的某些變數資料，使得變數可以在函式執行結束後保存下來，等到之後再次執行函式時可以直接使用，不須重新設定。為此 C++ 利用 `static` 保留字來達到這樣的效果。舉例來說，以下為一個簡單計數函式，並不須要輸入任何參數，但每次執行時其輸出值都會進一位，函式內部主要是使用了一個靜態變數 (static variable)，並為其設定初值，靜態變數的初值僅在函式第一次被執行時設定，之後就使用已保留下來的數值，並不會再重新設定數值，

```
int counter() {
    static  int  i = 0 ;   // i 為靜態整數，初值為 0，只被執行一次
    ++i ;
    return  i ;
}
```

若 `static` 變數如果沒有設定初值，則 C++ 會自動將其設為 0。讀者可以發現使用靜態變數的函式有個特點，也就是函式可以在使用相同參數值，卻得到不一樣的計算結果。其原因就是在使用靜態變數，當然這裡我們假設了函式並沒有使用任何的全域變數。

隨機函式 rand() 也是在函式內使用了靜態變數來儲存資料，如此隨機函式才得以在不輸入參數的情況下，於每次的執行都能產生不同的隨機數字。如果隨機函式內的靜態變數在每次程式執行時都使用一樣的初值，則隨機函式所產生的隨機數將會一樣。為避免隨機函式的可預測性，在第一次使用隨機函式之前，通常我們須要執行 srand 函式來設定隨機函式內靜態資料的初始值，使得每次程式執行時會有不同的隨機數。使用的方式請參閱在第 45 頁上的程式碼說明。

主函式

C++ 的程式皆由主函式開始執行，主函式本身也有回傳與參數列，主函式的原型如下：

```
int main( int argc，char * argv[] ) ;
```

第一個參數所代表的為程式在執行時所輸入的字串個數，字串數包含可執行檔本身的名字，而第二個參數就是儲存參數的字串陣列。舉例來說，如果可執行檔的名稱為 string_addition，之後接了三的參數分別為 cat dog fish，當使用者在命令行中輸入

```
string_addition cat dog fish
```

則主函式 main 的第一個整數參數 argc 值為 4，第二個字串陣列 argv 的元素分別為 "string_addition"，"cat"，"dog"，"fish"。

以下程式在讀入命令行的 "數字" 後，會印出其中的最大數。由於主程式都是以字串方式讀取命令列的資料，為了比較這些數字的大小，我們使用了在標頭檔 cstdlib 內定義的 atoi 函式，將一個傳統的字串數字轉成一般的整數數字。

```
#include <iostream>
#include <cstdlib>           // 或者用 <stdlib.h>

using namespace std ;

int main( int argc , char * argv[] ) {
    // 讀入第一個字串整數並將之轉為整數數字
    int  max = atoi(argv[1]) ;
    int  no ;
    for ( int i = 2 ; i < argc ; ++i ) {
        no = atoi(argv[i]) ;
        if ( no > max ) max = no ;
    }
    cout << "> max value : " << max << endl ;
    return 0 ;
}
```

請注意，命令列的參數是由字串陣列 argv 的第二筆開始存起。

8.6 回傳型別

每個函式都可以在函式結束之前回傳一筆資料給呼叫的程式碼，這筆資料的型別就是函式的回傳型別 (return type)。我們可以在函式內使用 `return` 將其後的運算結果送回呼叫的程式碼，此時運算結果的型別即為函式回傳的型別，如下例 (a)。有時函式並不須要回傳任何資料，如下例 (b)，在這裡情況下，回傳的型別就定義成空的型別，用 `void` 來表示。一個函式可能有多個 `return` 敘述，但所有 `return` 之後的資料型別皆要一樣。

(a) 計算兩數的最大值

```
int max( int a , int b ) {
    if ( a > b )
        return  a ;
     else
        return  b ;
}
```

(b) 列印兩數的最大值

```
void print_max( int a , int b ) {
    if ( a > b )
        cout << a ;
    else
        cout << b ;
}
```

一般來說，每個函式僅能回傳一筆資料，若要一次送出許多筆資料，則可以將這些資料用結構型別[53]的方式包裝起來，例如：

```
// 定義一複變數結構型別
struct Complex {
    double  re ;
    double  im ;
} ;                          // 不要忘了分號

Complex  set_complex( double real , double imag = 0 ) {
    Complex  foo ;
    foo.re = real ;
    foo.im = imag ;
    return  foo ;                  // 回傳 foo
}
```

使用結構的時機是結構內的資料有相關性，如果硬是將無相關的資料變數放在一個結構內反而會造成誤解，因此這種方式不見得適合所有的情況，在這種情況下，使用者可能須要透過一些參數將資料回送給呼叫的程式敘述。

指標型別的回傳

函式也可以回傳指標型別的變數，也就是用來回傳某個資料的位址，例如：

(a) 呼叫的程式碼

```
int  a = 3 , b = 5  ;
int *c = max(&a,&b) ;
*c = *c + 10  ;
cout << b << endl ;
```

(b) 函式碼 : 回傳參數中較大數值的位址

```
int * max( int *p , int *q ) {
    return ( *p > *q ? p : q ) ;
}
```

這裡函式 max 的回傳型別為一個整數指標型別，用 int * 來表示，由於指標變數是用來儲存同型別變數的位址，因此回傳指標的用意當然是要將某個資料的位址回傳出去，而不是資料本身的數值。首先上式的呼叫程式碼部份先將兩個整數的位址傳入 max 函式，由兩個指標參數 p，q 接收，也就是 C++ 分別執行了

```
int *p = &a ;   // 指標參數 p 指向傳入變數 a 的位址
int *q = &b ;   // 指標參數 q 指向傳入變數 b 的位址
```

經過設定後，指標參數 p 會指向傳入變數 a 的位址，而指標 q 會指向 b 的位址。由於函式須要回傳一指標型別，當比較過兩指標所指向的數值大小後，函式會回傳指向較大數值的指標參數，也就是在 a 與 b 兩者之間較大的整數位址，以本例而言，由於 b 較 a 值為大，因此函式將回傳 q 指標，也就是 b 的位址。

當程式碼控制回到呼叫程式部份時，C++ 執行 int *c = q，使得整數指標 c 指向指標參數 q 所指到的整數，也就是 b 的位址，透過 c 指標的間接運算後，b 的數值最後也被改變成為 15。

以上呼叫部份的程式碼可以進一步簡化成以下方式，

```
(a) 呼叫程式
   int  a = 3 , b = 5 ;
   *max(&a,&b) += 10 ;   // 執行 *q += 10, 因為 q 指標指到 b 的位址
   cout << b << endl ;
```

這裡的 max(&a,&b) 回傳指標參數 q，因此 *max(&a,&b) += 10 也就是相當於 *q += 10，由於指標參數 q 是指向 b 的位址，因此最後 b 的值被改成 15。要切記 += 的左邊須有一個記憶空間用來儲存右側的數值，這觀念在第 2.6 節中有特別強調。

如果 max 的參數是使用參考方式將資料傳入函式使用，則呼叫的程式碼與函式須改寫為

```
(a) 呼叫的程式碼
   int  a = 3 , b = 5 ;
   int *c = max(a,b) ;
   *c = *c + 10 ;
   cout << b << endl ;
```

```
(b) 函式碼 : 回傳參數中較大數值的位址
   int * max( int &m , int &n ) {
       return ( m > n ? &m : &n ) ;
   }
```

程式也會印出 15。請留意，在函式參數列的參數 &m，&n 代表參考型別，分別為 a 與 b 的參考，而在回傳敘述中的 &m 與 &n 則是代表 m 與 n 的位址，也就是 a 與 b 的位址。最後指標 c 則會指向 a 與 b 兩整數的其中一個位址。

當函式回傳指標時，須特別留意指標不可指向函式內部所定義的變數或參數，因其在函式執行完畢後都會消失 (當然 static 靜態變數除外)，因此以下的程式碼是錯誤的：

```
(a) 呼叫程式碼
   char  *p ;
   p = char_str('x',10) ;
   cout << p << endl ;
```

```
(b) 函式碼 : 設定 s 字串
   int * char_str( char a , int n ) {
       char  s[100] = { '\0' } ;
       for ( int i = 0 ; i < n ; ++i )
```

```
        s[i] = a ;
    return  s ;
}
```

函式 char_str 的作用是要造出由 n 個 a 字元所組成的字串，但很不幸地，函式內的 s 字元陣列在函式執行完畢後隨即消失，陣列的記憶空間會被系統收回以供其它程式使用。因此函式若將這樣的記憶空間回傳給指標 p，而且程式若再透過 p 指標去存取修改已被系統收回的記憶空間內容，則會引發執行錯誤 (run-time error) 而中斷。

若函式須要回傳一個指標，一定要避免回傳暫時變數位址。以上的情況可以在函式內使用動態空間的方式[99]來解決，例如：

(a) 呼叫程式碼

```
char  *p ;
p = char_str('x',10) ;
cout << p << endl ;
...
delete p ;
```

(b) 函式碼

```
int * char_str( char a , int n ) {
    char * s = new char[n+1] ;
    for ( int i = 0 ; i < n ; ++i )
        s[i] = a ;
    s[n] = '\0' ;
    return s ;
}
```

由於使用 new 方式取得的動態空間，其佔用的記憶空間是永久存在的，此空間會持續保留直到遇到 delete 或者是程式執行結束後，才會將此記憶空間送回系統重新使用，使用這樣的方式就不會造成任何的執行上錯誤。

參考型別的回傳

指標與參考都可以用來間接地參照另一個變數資料，當函式回傳一個參考型別時，就代表著可以透過函式的回傳間接地讀取或者是更改回傳變數的資料。舉例來說，以下的左右兩邊都會將整數 c 的值複製給整數 a：

(a) 不使用函式

```
int a  = 3 , c = 5 ;
int &b = a ;
b = c      ;
```

(b) 使用函式

```
int & fn( int &d ) { return d ; }
...
int a = 3 , c = 5 ;
fn(a) = c ;
```

左邊是利用參考 b 間接地將 c 的值複製給 a，但右邊卻是透過一個函式以更間接地方式將 c 複製給 a，其主要的關鍵敘述為 fn(a) = c。由於指定運算子的基本作用是讀出等號右側變數的數值 (read value) 存入左側變數所在的記憶空間位址 (location value)。也就是說，指定運算子的左邊須要有一個記憶空間來儲存右邊的數值，由於函式 fn 的回傳型別是一個參考型別，間接地代表著回傳變數的記憶空間，由其回傳敘述 return d 可知代表著參數 d 所在記憶空間，然而參數 d 又是輸入的函式變數 a 的參考，因此繞了一個大圈後 fn(a) 所代表的仍是變數 a 的記憶空間。fn(a) = c 的作用就是讀出 c 的數值後存入 a 所在的記憶空間。但如果將函式 fn 改成 fn2：

```
int  fn2( int & d ) { return d ; }
```

則 fn2 函式所回傳的不是 d 的本身，而是 d 的複製，複製當然不能代表 a 本身，同時所回傳的複製資料在函式執行結束後會隨即消失，因此不能置放在指定運算子的左邊用來儲存資料

```
int a = 3 , c = 5 ;
fn2(a) = c ;                    // 錯誤
```

但不管是否回傳參考或者是複製都可以在指定運算子的右側使用，例如：

```
int a = 3 , b , c ;
b = 3 * fn(a)  ;                // b 等於 9
c = 2 * fn2(a) ;                // c 等於 6
```

由此可知，當函式回傳參考型別時，由於其可以擺放在 =，+=，-=，*=，/=，++，-- 等運算子的左邊，也因此增加函式使用的自由度，這也是回傳參考型別的主要理由，以下再以一個例子來說明：

(a) 呼叫程式碼

```
int  a = 4 , b = 6    ;
max(a,b)++            ;
min(a,b)--            ;
cout << " a = " << a  ;
cout << " b = " << b  ;
cout << endl          ;
```

(b) 函式碼

```
int  &max( int &p , int &q ) {
    return ( p > q ? p : q ) ;
}
int  &min( int &p , int &q ) {
    return ( p > q ? q : p ) ;
}
```

這裡的程式是要讓 a 與 b 兩數間的較大數增加 1，較小數扣除 1。 int &max 函式的作用是回傳兩個參數 p 或 q 的最大數的參考，int &min 函式則回傳參數中最小數的參考。當 C++ 執行 max(a,b)++ 時，首先先處理函式 max(a,b)，此時函式的參數 p 與 q 分別是變數 a 與 b 的參考，若 p 較 q 大時，則就回傳 p 的參考，否則回傳 q 的參考，因此 max 所代表的即是 a 與 b 兩者較大數的參考，以本例來說，max 為 b 的參考，接下來 C++ 執行了 b++，因此 b 的值變成 7。同理 min(a,b)-- 的執行結果為 a--，因此 a 最後為 3。以上為方便解釋起見，好像將 max(a,b) 或者 min(a,b) 當成變數看待，但事實上並不存在如此的函式變數。

如果將函式回傳型別的參考符號 (&) 去除，則函式就不能置於 ++， -- 的左側，函式的 return 敘述所回傳的記憶空間是一個馬上會被系統回收的暫時變數記憶空間，也就是函式並沒有回傳一個記憶空間可以用來儲存資料，因此以下的使用方式就有錯誤，

```
int  max( int &p , int &q ) {  return ( p > q ? p : q ) ;  }
int  a = 4 , b = 6    ;
max(a,b)++            ;      // 錯誤的使用
```

注意，此時 max 函式是回傳參數的數值，而非參數所佔用的記憶空間，因此不能當成運算子的左值[25]使用，同樣的情況也適用於 =，+=，-=，··· 等類似的運算子，例

如：

```
int  max1( int &p , int &q ) { return ( p > q ? p : q ) ; }
int &max2( int &p , int &q ) { return ( p > q ? p : q ) ; }
...
int  a = 4 , b = 6 ;
max1(a,b) = 10      ;      // 錯誤
max2(a,b) = 10      ;      // 正確，a = 4 , b = 10
```

同樣的，函式若要回傳參考型別的變數或物件時，其注意事項與回傳指標型別相同，也是切忌回傳暫時變數的參考。例如：

```
// 輸入成績分數 p，若小於 60，則調至 60。
int & adjust_score( int & p ) {
    int q = 60 ;
    return ( p > q ? p : q ) ;
}
...
int no ;
cin >> no ;              // 輸入成績 no
adjust_score(no)++ ;     // 調分後，再加 1 分
```

這裡若輸入的成績 no 小於 60 分時，則函式所回傳的參考是一個在函式執行完後隨即消失的暫時變數 p，此時再使其加 1 是沒有意義的，因此程式將會出錯。有關函式回傳參考型別的應用，將在以後的運算子覆載時會經常使用。

8.7 函式指標

指標除了可以用來指向一些記憶空間外，也可以用來指向函式，這類特殊的指標被稱為函式指標 (function pointer)。函式指標為了要與普通指標有些分別，須要將函式的回傳型別，與其所使用的參數型別在定義中就要寫出，例如：

```
// fn1 函式指標可以指向所有使用一個整數參數及回傳整數的函式
int  (*fn1)( int ) ;

// fn2 函式指標可以指向所有使用兩個浮點數參數及回傳浮點數的函式
double  (*fn2)( double , double ) ;
```

這裡包圍函式指標名稱的括號不可以省略，以免與函式回傳一指標的意義混淆。這樣定義的函式指標，就可以自由地指向任何參數與回傳值型別相同的函式了，因此如果定義了以下幾類函式：

```
int  abs    ( int a ) { return a > 0 ? a : -a ; }
int  square ( int a ) { return  a * a ; }
int  cubic  ( int a ) { return  a * a * a ; }

int  min ( int a , int b ) { return ( a > b ? b : a ) ; }
```

```
int  max ( int a , int b ) { return ( a > b ? a : b ) ; }
```

則我們可以使用

```
int (*foo)(int) = abs   ;   // foo 指到 abs  函式
cout << foo(-5) << endl ;   // 執行 foo(-5) 相當於 執行 abs(-5)
foo = square ;              // foo 指到 square 函式
cout << foo(-2) << endl ;   // 執行 foo(-2) 相當於 執行 square(-2)

int (*bar)(int,int) ;
bar = min ;                 // bar 指到 min 函式
cout << bar(3,5) << endl ;  // 執行 bar(3,5) 相當於 執行 min(3,5)
bar = max ;                 // bar 指到 max 函式
cout << bar(3,5) << endl ;  // 執行 bar(3,5) 相當於 執行 max(3,5)

bar = abs ;                 // 錯誤，bar 函式指標所指到的函式須要
                            // 兩個整數參數
```

讓函式可以當成參數傳入另一個函式中是使用函式指標的主要理由，舉例來說，以下定義了一個求和 sum_by 函式，可以接受一個函式參數與兩個整數所構成的範圍，用來計算在此整數範圍內的所有整數經過函式處理後的總和，

```
int sum_by ( int (*fp)(int) , int a , int b ) {
    int s = 0 ;
    for ( int i = a ; i <= b ; ++i ) s += fp(i) ;
    return  s ;
}
```

如此就可以使用

```
// 計算 abs(-2) + abs(-1) + ... + abs(2)
cout << sum_by( abs    , -2 , 2 ) << endl ;        // 印出 6

// 計算 square(-2) + square(-1) + ... + square(2)
cout << sum_by( square , -2 , 2 ) << endl ;        // 印出 10

// 計算 cubic(-2) + cubic(-1) + ... + cubic(2)
cout << sum_by( cubic  , -2 , 2 ) << endl ;        // 印出 0

// 錯誤，函式指標只能指到須要一個整數參數的函式
cout << sum_by( max  , -2 , 2 ) << endl ;
```

以上的程式碼如果不將 abs，square，cubic 等函式當成參數傳入 sum_by 函式內，若要以一般方式得到同樣的效果，則要撰寫三個類似的函式才能達成目的，例如：

```
int abs_sum ( int a , int b ) {
    int s = 0 ;
    for ( int i = a ; i <= b ; ++i ) s += abs(i) ;
    return  s ;
```

```
}

int square_sum ( int a , int b ) {
    int s = 0 ;
    for ( int i = a ; i <= b ; ++i ) s += square(i) ;
    return  s ;
}

...
```

這種寫法相當地不經濟而且增加程式的複雜度，應極力避免使用。

將函式當成參數傳入另一個函式，可以提昇函式運算的自由度，這在 C 程式是經常被使用，然而對 C++ 的程式設計員而言，還可以使用另一種方式來達到同樣的效果，這將留到第十七章介紹標準樣板函式庫時再加以介紹。

8.8 行內函式

當編譯器在編譯函式時，會將編譯完成的函式目標碼 (object code) 置放在整個可執行碼的某處，當程式執行須要用到此函式時，則程式碼的執行流程就會由現在正在執行的敘述轉往函式的目標碼地方執行，等到函式程序執行結束後，程式才會回到原來的敘述繼續執行。一般來說，每當程式碼要開始執行函式時，程式都須要花費時間額外處理一些動作，例如：處理參數資料的傳遞，或是將原有的資料由中央處理單元的暫存器搬到記憶空間儲存等等，舉例來說：

```
int i = 3 , j = 5 ;
int max1 , max2 ;
max1 = ( i > j ?  i :  j ) ;
max2 = max(i,j) ;
```

```
int max( int a , int b ) {
  return a > b ?  a :  b ;
}
```

以上的 max1 與 max2 都是儲存 i 與 j 兩整數的最大值，然而前者是直接使用簡單的條件敘述求得，而後者則透過函式處理。同樣是計算兩數的最大值，使用函式的方式就須要額外花費時間與記憶空間將 i 與 j 複製到函式的參數，因此就顯得較前者不經濟。然而使用函式的處理方式，可以增加程式碼的可讀性，使得未來的程式維護工作較為輕鬆。

為了增加函式的效率，C++ 提供了一個機制，即行內函式 (inline function)，可以同時保有使用函式的可讀性與不使用函式的經濟性。其基本的策略就是透過分析函式的程式碼，直接將其轉成一般的程式敘述放置在函式呼叫的地方。對使用者而言，感覺上仍是執行函式，但對 C++ 而言，並沒有執行函式，而是一般的程式碼。以上例而言，max2 = max(i,j) 是將函式 max(i,j) 執行的結果存入 max2，但若是將 max 函式定義為行內函式後，則聰明的編譯器就會分析程式碼，先將函式敘述轉成一般程式敘述，即 max2 = i > j ? i : j，然後再進行編譯，因此編譯結束後，並沒有產生 max 函式目標碼，如此自然沒有參數複製等額外花費的時間發生。因此使用行內函

式就可以節省相當的執行時間，同時這也是直接將函式編譯在程式碼行內 (inline) 的意思。

要將一函式定義成行內函式只須在函式的回傳型別之前加上 inline 即可。以上例為例：

```
inline  int  max ( int a , int b ) {
    return  a > b ? a : b ;
}
```

由於 C++ 會將行內函式的程式碼直接放置在呼叫函式的地方，如果程式碼在許多地方都使用行內函式，則最後產生的可執行碼的檔案大小會較大。此外，行內函式對編譯器而言，只是一種建議動作，並沒有強制性。如果行內函式的程式太複雜，超過編譯器所能處理的能力，則行內函式仍是以一般的函式方式編譯。一般來說，對於行數太多的函式而言，宣告其為行內函式並沒有什麼作用。

8.9 函式範例

早期的程式設計是以設計函式為程式設計的核心，起初程式設計員將問題先分解成一步一步可執行的步驟，每一個步驟再以一個函式來代替，之後定義函式聯繫外部程式碼的介面，若在某函式中須要使用到其它函式，則重複以上的方式，直到所須要的函式為最小單位為止。因此整個程式碼就是由許多的函式連接而成。接下來，程式設計員的主要任務就是撰寫之前定義的所有的函式程式碼，當所有函式的程式碼完成後，整個程式也就設計完成了。這樣的程式設計理念也是個別擊破演算法 (divide-and-conquer) 的技巧。

若以生產線上的輸送帶來比喻一個程式，各個函式如同輸送帶上的作業員，所要處理的資料就是在輸送帶上的物品，撰寫程式碼就如同在設計工廠的生產線一樣。這種程式設計方式有個特點，即是資料與處理資料的函式是分開設計的。當程式的程式碼很龐大時，其所定義的資料變數與函式也都會相當複雜，很容易發生輸入不恰當的資料到函式內而一無所知，或者是在函式內更改到不應改變的資料，這樣的情況都會引起程式執行錯誤。

若程式是由一群人協力撰寫，則很容易因彼此的溝通不良，造成各自撰寫功能近似卻又不同的函式而不知。這種情況會造成以後的程式維護人員無法分辨其間的差異而不知所措。因此這類以撰寫函式為主的設計理念已經漸漸被淘汰。雖然如此，初學者仍是要熟練本章的基本語法，但不要用來設計複雜的大程式，免得習慣了不良的程式設計方式，而造成往後學習不同程式設計理念的困難，以下我們用幾個例子來示範函式的使用。

字串數字相加

C++ 一般的整數多半使用四個位元組的記憶空間，其所能代表的最大的正整數為 $2^{32}-1$，將近 43 億左右，這樣的大小無法用來計算超過 13！的數字。如果須要處理

兩個很大的整數，假設超過 20 位數的整數間四則運算，則 C++ 所提供的整數型別就不夠用了。在這種情況下，一般的處理方式是使用字串來儲存整數，這裡我們要用字串方式來撰寫兩個大整數的相加。

一般加法的過程是由右往左一位一位相加，同時也要考慮到進位的問題。為了處理方便，首先將兩個字串長度調成一致，在較短的字串數字的前面加上字元 0，使得兩個字串長度等長。舉例來說，若有以下兩數字字串相加，則首先將短字串改成：

```
   "9876543210"                  "9876543210"
+      "123456"        ==>    +  "0000123456"
---------------               ---------------
```

接下來，就可以使用迴圈由字串的最後一個字元一一往前加，當然在加法過程中要考慮到進位的問題。在程式中，我們須要由數字字元計算其對應的數字整數，若已知一數字字元為 'n'，則其對應數字 n 就是字元 'n' 與字元 '0' 之間的距離，即 `n = 'n' - '0'`。若要由數字 n 求得字元 'n' 則可用 `'n' = static_cast<char>(n+'0')` 求得。兩個字元數字和是否進位可以計算其值除以 10 後的商數，如以下的 `c`：

```
char a = '9' , b = '7' ;
int  s = ( a - '0' ) + ( b - '0' ) ;
int  c = s / 10 ;
```

減法的處理稍微複雜些，我們將之留為作業題目。

數字字串相加　　bigint_addition.cc

```
01  #include <iostream>
02  #include <string>
03
04  using namespace std ;
05
06  // 字串整數相加
07  string  bigint_addition( string a , string b ) {
08
09      int  i , tmp ;
10
11      // 計算兩數字字串長度差值
12      int  size_difference = a.length() - b.length() ;
13
14      if ( size_difference < 0 ) size_difference *= -1 ;
15
16      // 將較短的數字字串之前補上字元 0 , 使得最後兩字串等長
17      if ( a.length() > b.length() )
18          for ( i = 0 ; i < size_difference ; ++i ) b = '0' + b ;
19      else
20          for ( i = 0 ; i < size_difference ; ++i ) a = '0' + a ;
21
22      string    sum = "" ;    // 字串數字和
23      int     carry = 0  ;    // 加法的進位數
24
25      // 由字串的最尾字元往前加, 將進位的數字存入 carry 變數中
26      for ( i = a.length()-1 ; i >= 0 ; --i ) {
27          tmp = ( b[i] - '0' ) + ( a[i] - '0' ) + carry ;
```

```
28            sum = static_cast<char>(( tmp % 10 + '0' )) + sum ;
29            carry =  tmp / 10 ;
30        }
31
32        // 處理最後剩下進位數字
33        if ( carry ) sum = static_cast<char>( carry + '0' ) + sum ;
34
35        return sum ;
36
37    }
38
39    int  main() {
40
41        string  a , b ;
42
43        cout << "> 輸入兩個正整數 : " ;
44        cin  >> a >> b ;
45
46        cout << '\n' << a << " + " << b << " = "
47             << bigint_addition(a,b) << endl ;
48
49        return 0 ;
50
51    }
52
```

執行結果

```
01    > 輸入兩個正整數 : 9876543210 1234567890123
02
03    9876543210 + 1234567890123 = 1244444433333
04
```

點矩陣數字

在第六章的點矩陣範例[143]中，我們介紹了如何使用數字來儲存字元的點矩陣資料，這些字元可以是標準的 ASCII[14] 字元，也可以是中文字。藉由變換字元的點矩陣資料，我們就可以造出種種不同的字型。這裡我們將利用函式的方式來建構數字字元的點矩陣，並將其印出。

在此程式中，我們將整個程式步驟分為四個主要部份，第一部份為讀入資料，第二個到第四個步驟分別是利用不同的函式來建構，列印與清除點矩陣字串陣列。如此一來，主程式部份就相當的乾淨與清楚。同時為了讓主程式位置提昇到程式的前段，我們特別使用函式的原型來宣告函式的存在。

點矩陣數字　　math_bitmap.cc

```
01  #include <iostream>
02  #include <string>
03
04  using namespace std ;
05
06  // 每個數字為 5 x 5 的點矩陣
07  const int ROW = 5 , COL = 5 ;
08
09  // 建構點矩陣
10  void  build_bitmap( const string& , string * ) ;
11
12  // 列印點矩陣
13  void  print_bitmap( const string * ) ;
14
15  // 清除點矩陣
16  void  clear_bitmap( string * ) ;
17
18  int main() {
19
20      // no     : 為輸入的數字字串
21      // bitmap : 陣列為每一列所構成的點矩陣字元資料
22      string  no , bitmap[ROW] ;
23
24      while( 1 ) {
25          cout << "> 輸入正整數字串 : " ;
26          cin >> no ;
27          if ( no == "0" ) break ;            // 跳離
28          build_bitmap( no , bitmap ) ;       // 建構點矩陣資料
29          print_bitmap( bitmap ) ;            // 列印數字的點矩陣
30          clear_bitmap( bitmap ) ;            // 清除點矩陣
31          cout << endl ;
32      }
33
34  }
35
36  // 建構輸入數字字串的點矩陣
37  void  build_bitmap( const string& no , string bitmap[] ) {
38
39      // 點矩陣數字 : 共十個數字 0 , 1 , 2 , ... , 9
40      static unsigned char no_bitmap[10][ROW] =
41          { {14,17,17,17,14} , {4,12,4,4,14} , {14,17,2,4,31}   ,
42            {30,1,14,1,30}   , {2,6,10,31,2} , {31,16,30,1,30}  ,
43            {15,16,30,17,14} , {31,1,2,4,8}  , {14,17,14,17,14} ,
44            {14,17,15,1,30}   };
45
46      int i , n , r , c ;
47      for ( r = 0 ; r < ROW ; ++r ) {
48          for ( i = 0 ; i < no.size() ; ++i ) {
49
50              // 計算每一列所對應的點矩陣資料
51              n = no_bitmap[ no[i] - '0' ][r] ;
52
53              // 列的點矩陣數字內若有數字  則以星號代替, 否則加上空白
54              for ( c = COL-1 ; c >= 0 ; --c ) {
55                  bitmap[r] += ( n & ( 1 << c ) ? '*' : ' ' ) ;
56              }
57
58              // 每個數字之間用兩個空白隔開
```

```
59              bitmap[r] += "  " ;
60          }
61      }
62  }
63
64  // 列印點矩陣
65  void  print_bitmap( const string * bitmap ) {
66      for ( int i = 0 ; i < ROW ; ++i ) cout << bitmap[i] << endl ;
67  }
68
69  // 清除點矩陣
70  void  clear_bitmap( string bitmap[] ) {
71      for ( int i = 0 ; i < ROW ; ++i ) bitmap[i] = "" ;
72  }
73
```

執行結果

```
01  > 輸入正整數字串 : 1234567890
02    *     ***   ****      *   *****   ****  *****   ***    ***    ***
03   **    *   *      *    **   *      *          *  *   *  *   *  *   *
04    *       *    ***    * *   ****   ****      *    ***    ****  *   *
05    *      *        *  *****      *  *   *    *    *   *      *  *   *
06   ***   *****  ****      *   ****    ***    *     ***   ****    ***
07
08  > 輸入正整數字串 : 0
09
```

在以上 build_bitmap 的函式中，每個數字使用了 5 列乘 5 行的點矩陣資料來表示，由於每一行僅須要 5 個位元 (bit) 的空間，為了減少浪費，我們特別使用 8 個位元的 unsigned char 空間來儲存，因此程式的 unsigned char 在使用上與字元無關，而只是用來單純地儲存 8 個位元的整數，可以儲存 0 到 255 之間的整數。此外讀者請留意，在 clear_bitmap 函式的 bitmap 參數為陣列模式，但其原型卻是為指標模式，但兩者在參數列的定義上並無差別。

在程式中，我們利用位元計算[48]來判斷某位元的資料，例如：1 << c，代表將數字 1 往左移動 c 個位元位置，之後再使用 & 運算子與數字 n 比較，也就是 n & (1 << c)，由其計算結果就可以知道整數 n 由右邊數來第 c+1 個位元是否有值。

讀者由此程式可以推知，一個 16 × 16 的點矩陣資料至少須要使用 32 個位元組 (byte) 的空間。 8000 個這樣大小的中文字就須使用 256 KB 的空間。因此 4 個 16 × 16 的不同中文字型就要佔用 1 MB 的記憶空間，若再考慮其它大小的字型，則就可以知道為何中文字型須要耗費許多空間來儲存的原因。

列印函式

在許多數學工具軟體內，都會設計一些方式讓使用者利用一些簡單的指令，就可以將

某函式的圖形畫在螢幕上，這類的指令通常類似以下的方式，例如：若要在螢幕上畫出 sin(x) 函式在 x 介於 [a,b] 之間的圖形

```
plot  sin(x) [a,b]
```

這裡的 plot 為這些工具軟體內一個專門負責繪圖的函式，而其後的 sin(x) [a,b] 則為此函式的輸入參數，如果要改畫另一個函式 cos(x)，則僅要將輸入的函式名稱換掉即可。如此一來，這些 sin(x) 函式也就變成另一個函式的輸入參數。為了達到這個目的，在這裡，我們可以使用函式指標來完成類似的功能，例如：以下的函式 print_function 可以將 sin(x) 與 cos(x) 函式在 x 介於 [a,b] 區間切割成 d 等份的函式值列印出來，且輸出時以 c 行的方式顯示：

```
print_function( sin , a , b , d , c ) ;
print_function( cos , a , b , d , c ) ;
```

這類程式的設計方法，就是利用函式指標來完成，函式指標的作用是將函式當成另一個函式的參數使用，這樣可以使得一些函式的使用更自由。

函式指標 — function_ptr.cc

```
01  #include <iostream>
02  #include <iomanip>
03
04  using  namespace  std ;
05
06  // fn(x) = x * x
07  double  square( double x ) { return  x * x ; }
08
09  // fn(x) = ( x - 1 ) ( x - 1 )
10  double  f     ( double x ) { return  x * x - 2. * x + 1. ; }
11
12  // 以 c 行列印 函式 fn 在區間 [a,b] 之值, 區間將切割成 division 等份
13  void  print_function( double (*fn)(double) , double a , double b ,
14                        int division = 20 , int c = 3 ) {
15
16      int     i , j ;
17
18      // 列印標頭
19      for ( i = 0 ; i < c ; ++i )
20          cout << setw(6) << " X  " << setw(7) << "F(X)"
21               << setw(4) << " " ;
22      cout << endl ;
23
24      for ( i = 0 ; i < c ; ++i )
25          cout << setw(6) << "====" << setw(7) << "===="
26               << setw(4) << " " ;
27      cout << endl ;
28
29      // n 為函式的點數
30      int     n = division + 1 ;
31
32      // x 為函式的自變數, dx 為每一等份的大小
33      double  x = a ;
```

```
34      double  dx = ( b - a ) / division ;
35
36      // 計算最少須要的顯示的列數 r (row)
37      int  r = n / c ;
38      if ( r * c < n ) r++ ;
39
40      // 最後一行所須要顯示的列數
41      int  s =  n - r * (c-1) ;
42
43      // 浮點數的小數位數以 2 位輸出, 且小數位數不足的部份補上 0
44      cout << fixed << setprecision(2) ;
45
46      // 列印 r 列 (row)
47      for ( j = 1 ; j <= r ; ++j , x += dx ) {
48
49          // 列印前 c-1 行 (column)
50          for ( i = 0 ; i < c-1 ; ++i )
51              cout << setw(6) << x+(r*i)*dx << setw(7) << fn(x+(r*i)*dx)
52                   << setw(4) << " " ;
53
54          // 列印最後一行
55          if ( j <= s ) cout << setw(6) << x+(r*(c-1))*dx
56                             << setw(7) << fn(x+(r*(c-1))*dx) ;
57
58          cout << endl ;
59
60      }
61  }
62
63  int main() {
64
65      // 列印  x*x 在 [0,1] 的數值, 以預定的 20 等份切割, 及 3 行列印
66      cout << "> X*X : " << endl ;
67      print_function( square , 0 , 1 ) ;
68      cout << endl ;
69
70      // 列印 (x-1)(x-1) 在 [0,3] 的數值, 以 30 等份切割, 且 4 行列印
71      cout << "> (X-1)(X-1) : " << endl ;
72      print_function( f , 0 , 3 , 30 , 4 ) ;
73
74      return 0 ;
75
76  }
77
```

執行結果

```
01  > X*X :
02     X     F(X)       X     F(X)       X     F(X)
03    ====   ====      ====   ====      ====   ====
04    0.00   0.00      0.35   0.12      0.70   0.49
05    0.05   0.00      0.40   0.16      0.75   0.56
06    0.10   0.01      0.45   0.20      0.80   0.64
07    0.15   0.02      0.50   0.25      0.85   0.72
08    0.20   0.04      0.55   0.30      0.90   0.81
```

```
09      0.25   0.06      0.60   0.36      0.95   0.90
10      0.30   0.09      0.65   0.42      1.00   1.00
11
12    > (X-1)(X-1) :
13       X     F(X)       X     F(X)       X     F(X)       X     F(X)
14      ====   ====      ====   ====      ====   ====      ====   ====
15      0.00   1.00      0.80   0.04      1.60   0.36      2.40   1.96
16      0.10   0.81      0.90   0.01      1.70   0.49      2.50   2.25
17      0.20   0.64      1.00   0.00      1.80   0.64      2.60   2.56
18      0.30   0.49      1.10   0.01      1.90   0.81      2.70   2.89
19      0.40   0.36      1.20   0.04      2.00   1.00      2.80   3.24
20      0.50   0.25      1.30   0.09      2.10   1.21      2.90   3.61
21      0.60   0.16      1.40   0.16      2.20   1.44      3.00   4.00
22      0.70   0.09      1.50   0.25      2.30   1.69
23
```

在此程式中，函式所須要列印的資料數量 (n) 與行數 (c) 已經確定，由於函式值是由上而下，由左而右的方式列印，假設前 c-1 行的列數為 r，最後一行的資料數量為 s，則我們有以下的關係式，

```
n = r * (c-1) + s
```

這樣的關係式造成前 c-1 行所需要印列的列數 r 並不是唯一的，如果要讓 r 越小越好，則經過簡單的運算可知，r 可以取成 $\lceil \frac{n}{c} \rceil$，也就是，如果 n 與 c 的比值如果有小數部份時，則無條件進位。相當於程式中的

```
int r = n / c ;
if ( r * c < n ) r++ ;
```

這也可以利用在 cmath 標頭檔內的進位函式，ceil， 改寫成 r = ceil(n/c)。當 r 得知後，最後一行的資料列數 s 就可以使用 n - r * (c-1) 求得。

此外在程式中，為了使得輸出的數字對齊，我們使用了在第十一章所介紹的控制輸出格式來設定浮點數的輸出格式[335]。由於不是本範例的重點，將不在此加以特別介紹其用法。

8.10 遞迴函式

每一個函式內的程式碼都可以依須要自由地呼叫其它的函式 (主函式除外)，如果函式呼叫自己本身，則這樣的函式稱為遞迴函式 (recursive function)。在數學上，有些函數也常常會以遞迴的方式來定義。例如，階乘函數通常被定義成：

$$n! = \begin{cases} 1 & n = 0 \text{ 或 } 1 \\ n \times (n-1)! & n > 1 \end{cases}$$

這類的遞迴函數若寫成函式也是相當近似，例如：

```
unsigned int  factorial( unsigned int n ) {
    if ( n == 0 || n == 1 )
        return  1 ;
    else
        return  n * factorial( n - 1 ) ;
}
```

依照上面的遞迴函式執行步驟，如果要計算 `factorial(4)`，則將會用以下的方式展開，

```
factorial(4) = 4 * factorial(3)
             = 4 * ( 3 * factorial(2) )
             = 4 * ( 3 * ( 2 * factorial(1) ) )
             = 4 * ( 3 * ( 2 * 1 ) )
             = 4 * ( 3 * 2 )
             = 4 * 6
             = 24
```

以上程式中的括號是刻意被加上的，用來辨別計算的順序。

由於遞迴函式會一直不斷的在呼叫自己，如果遞迴函式內的程式碼不作任何判斷，檢查是否遞迴步驟已經終止，則會造成無窮遞迴的狀況。一般來說，C++ 在執行函式時，系統會自動分配適當的空間給函式使用，如果函式一直遞迴重複沒有終結，則很快的系統的記憶空間就會消耗殆盡，造成程式執行中斷。因此撰寫遞迴函式的首要工作就是要確認遞迴函式的終止條件，在函式內要很明確地將終止條件寫出，這通常都是使用條件式來檢查，而且將之當成遞迴函式的第一個執行敘述，例如：

```
type  recursive_function (...) {
    if ( ... ) {                    // 判斷遞迴函式是否到達遞迴終止
        A                           // A : 終止遞迴
    } else {
        B                           // B : 繼續遞迴
    }
}
```

舉例來說，巴斯卡三角形 (Pascal's triangle) 也可以用遞迴公式來計算。

$$\mathbf{P} = \begin{bmatrix} 1 & & & & \\ 1 & 1 & & & \\ 1 & 2 & 1 & & \\ 1 & 3 & 3 & 1 & \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix} \qquad P_{i,j} = \begin{cases} 1 & i = j \text{ 或 } j = 0 \\ P_{i-1,j-1} + P_{i-1,j} & i > j \end{cases}$$

這裡的 `i` 代表列數 (row)，`j` 代表行數 (column)，其遞迴函式則為：

```
int  pascal( int i , int j ) {
    if ( i == j || j == 0 )
```

```
        return  1 ;
    else
        return  pascal(i-1,j-1) + pascal(i-1,j) ;
}
```

我們在一進入遞迴函式內，馬上就檢查遞迴是否終止，如果是，則直接回傳數值，如果不是才繼續遞迴，這種執行方式如同 for 迴圈一般。

如果使用以上的函式計算 pascal(4,2) 的數值，則須要經過以下的遞迴步驟：

```
pascal(4,2) = pascal(3,1) + pascal(3,2)
            = ( pascal(2,0) + pascal(2,1) ) + pascal(3,2)
            = ( 1 + pascal(2,1) ) + pascal(3,2)
            = ( 1 + ( psscal(1,0) + pascal(1,1) ) ) + pascal(3,2)
            = ( 1 + ( 1 + pascal(1,1) ) ) + pascal(3,2)
            = ( 1 + ( 1 + 1 ) ) + pascal(3,2)
            = ( 1 + 2 ) + pascal(3,2)
            = 3 + pascal(3,2)
            = 3 + ( pascal(2,1) + pascal(2,2) )
            = 3 + ( ( pascal(1,0) + pascal(1,1) ) + pascal(2,2) )
            = 3 + ( ( 1 + pascal(1,1) ) + pascal(2,2) )
            = 3 + ( ( 1 + 1 ) + pascal(2,2) )
            = 3 + ( 2 + pascal(2,2) )
            = 3 + ( 2 + 1 )
            = 3 + 3
            = 6
```

在以上運算過程中，有些相同的函式被重複的執行，例如：pascal(2,1)，這種情況對計算越大且越底層的巴斯卡三角形數值越嚴重，除了造成記憶空間的浪費外，也會嚴重拖累執行效率，讀者可以試試計算 pascal(50,25) 所要花費的時間。由此可知一些在數學上定義的公式在實際演算過程中可能都須要加以修正，以提高執行效率。

快速遞迴程式

遞迴演算常常會造成資料的重複計算，因而拖累整個計算時間，因此我們可以考慮將已經計算過的數值資料儲存下來，以後就可以重複使用，不須再浪費時間重新計算。例如：數學中的 Fibonacci 數列是由兩個整數 1 開始，兩兩相加得到下一數，也就是如下：

$$a_n = \begin{cases} 1 & n = 0 \text{ 或 } 1 \\ a_{n-2} + a_{n-1} & n > 1 \end{cases}$$

其數列如右 1，1，2，3，5，8，13，21，34，... 等。如果以遞迴的方式寫成函式時，則程式碼如下：

```
unsigned int  fib( unsigned int n ) {
    return ( n < 2 ? 1 : fib(n-2) + fib(n-1) ) ;
```

```
}
```

使用這樣的遞迴函式計算稍大的 n (例如大於 40 的整數) 時，則會因為大量地重複計算，嚴重地影響到計算速度，這時我們可以利用 static 變數將已經計算過的資料儲存起來供以後使用，也就是：

```
unsigned int  fib( unsigned int n ) {
    static unsigned int f[100] ;       // static 陣列初值自動設為 0
    if ( n < 2 )
       return  1 ;
    else if ( f[n] != 0 )              // 若已算過，則回傳儲存數值
       return  f[n] ;
    else
       return  f[n] = fib(n-2) + fib(n-1) ;
}
```

在以上我們使用一個靜態陣列 f 來儲存 Fibonacci 數列的值，如果所須要計算的資料已經被儲存，則直接回傳，如果沒有計算過，則利用遞迴方式計算，當結果存到陣列後才回傳。不過若要純粹考量執行效率，則就應當避免使用遞迴而是直接以迴圈的方式來計算，例如：

```
unsigned int  fib( unsigned int n ) {
    // 前兩數為 1，其它為 0
    static unsigned int  f[100] = { 1 , 1 } ;
    if ( n < 2 || f[n] != 0 )
       return  f[n] ;
    else {
       for ( int i = 2 ; i <= n ; ++i ) f[i] = f[i-2] + f[i-1] ;
       return  f[n] ;
    }
}
```

請留意，這類整數計算的問題都須要考慮到可能產生的數值溢位問題。例如：階乘函式就無法算出超過 13 階乘以上的函式值，因此如果要計算一個組合數 C_{13}^{15}，則不能直接使用其公式演算，例如：

$$C_{13}^{15} = \frac{15!}{2!13!}$$

而是要直接使用 7×15，這也是另一個數學的函數與程式語言的函式相異的部份。

遞迴函式的使用時機

這裡我們來看看為何要使用遞迴函式，假設有一個初學程式設計的學生要計算 1 到 10 的整數和，則其可以使用以下的建構式方法一條一條地寫下計算式子來加以演算：

```
int  sum = 0 ;
```

```
sum = sum + 1 ;  sum = sum + 2 ;  sum = sum + 3 ;  sum = sum + 4 ;
sum = sum + 5 ;  sum = sum + 6 ;  sum = sum + 7 ;  sum = sum + 8 ;
sum = sum + 9 ;  sum = sum + 10 ;
```

當然對學過迴圈的人而言，這是使用迴圈的最佳時機，

```
int  sum = 0 , max = 10 ;
for ( int i = 1 ; i <= max ; ++i ) sum = sum + i ;
```

這樣寫的好處在於如果臨時要計算 1 到 100 之間的所有整數和，則前者須要費力地再寫 90 個敘述，而後者只要將 max 的值改成 100 即可。

同理來看看以下的組合計算問題，例如：若要將 1 到 5 的 5 個相異數字隨意取出 3 個，並將其所有組合列印出來，由數學分析可知其共有 $C_3^5 = \frac{5!}{3!2!} = 10$ 種情況，這類問題可以利用 3 層迴圈方式來處理，例如：

```
const int m = 5 ;                          // 共有 5 個相異數
const int n = 3 ;                          // 取出 3 個數值
int   number[n] ;                          // 儲存取出的數字
int i , j , k , s ;

for ( i = 1 ; i <= m ; ++i ) {             // 第一層迴圈
    number[0] = i ;                        //     儲存第一個數字
    for ( j = i+1 ; j <= m ; ++j ) {       // 第二層迴圈
        number[1] = j ;                    //     儲存第二個數字
        for ( k = j+1 ; k <= m ; ++k ) {   // 第三層迴圈
            number[2] = k ;                //     儲存第三個數字

            cout << "[" ;                  //     列印取出的數字
            for ( s = 0 ; s < n-1 ; ++s )
                cout << number[s] << ' ' ;
            cout << number[n-1] << "] " ;

        }
    }
}
```

其輸出則為：

```
[1 2 3] [1 2 4] [1 2 5] [1 3 4] [1 3 5] [1 4 5] [2 3 4] [2 3 5]
[2 4 5] [3 4 5]
```

如果是要印出 C_4^5 的所有數字組合，則要多加一層迴圈，也就是使用四層迴圈，同理，如果要印出 C_{50}^{100} 的所有數字組合，則要使用 50 層迴圈。當取出的數量不一樣時，程式碼就須要隨之作更動，這樣的情況就有如之前要使用 100 個加法敘述來計算 1 到 100 的數字和一樣地不切實際。而解決後者的其中一種方式就是使用遞迴處理，以此為例，我們可以觀察到每個相鄰的兩層迴圈之間的演算關係是有相似的情況，如果將其轉化為呼叫函式的方式，則迴圈的層數剛好就是遞迴函式執行的深度

(depth)。如此一來，程式就可以簡化為以下的方式，這裡為方便起見，我們將 C_n^m 的 m 與 n 皆定義成全域變數[44]，仿照之前的程式碼，在遞迴函式內共使用了三個參數來傳遞資料，分別用來儲存取出的數字，遞迴深度，與每次進入迴圈的起始數值。

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

const int m = 5 ;     // 共有 1 到 5 , 五個相異數
int       n = 3 ;     // 取出 3 個數字

// 遞迴函式 : 列印所有組合數字  m = 5 , n = 3
//   number : 儲存取出的數字   depth : 遞迴深度
//   no : 每一次迴圈的起始數值
void print_combination( int number[] , int depth , int no ) {

    if ( depth == n )  {
        cout << "[" ;
        for ( int i = 0 ; i < n-1 ; ++i ) cout << number[i] << ' ' ;
        cout << number[n-1] << "] " ;
    } else {
        for ( int i = no ; i <= m ; ++i ) {
            number[depth] = i ;
            print_combination(number,depth+1,i+1) ;
        }
    }
}

int  main(void) {

    int  number[m] ;
    print_combination(number,0,1);
    return 0 ;

}
```

輸出與使用迴圈的方式一樣。在遞迴函式中，每次執行下一層遞迴函式時，遞迴深度與迴圈的起始數就增加一。遞迴函式的執行深度相當於之前程式的迴圈層數，也就是，設定遞迴深度，n，就相當於控制所要處理的迴圈層數，如此一來，遞迴函式就不會無止盡的遞迴下去。同時只要更改 n 的數值，執行的遞迴深度就會隨之變化，並不須要如之前的迴圈程式一樣，必須將程式做大幅度的修改，因此遞迴函式也避免了程式碼的變動性。

所謂的遞迴函式是指函式在執行的時候須要呼叫自己，也就是說，為了得到函式對大問題的結果須要使用到同樣的函式對較小問題的結果。由這個概念引申，我們可以推知使用遞迴函式的時機就是：某問題可以分解成若干個小問題，而每個小問題的解決方式又與原來的問題相似。

撰寫遞迴程式切忌追蹤敘述進入下一層遞迴函式內，如此常常會讓思緒陷入無窮

遞迴的陷阱中，無法跳出。基本上，撰寫一個成功的遞迴程式只要記住以下兩個原則即可：

(1) **尋找遞迴結構**　在原始問題的解決步驟中尋找同型式的小問題，構成基本遞迴架構

(2) **確認終結條件**　為避免程式陷入無止盡的遞迴，因此要確認終結條件是可以達到的

8.11 遞迴函式範例

以下我們用幾個常見的遞迴問題來解釋這兩個原則。

$\sqrt{2}$ 與連分數

在數學上，無理數根號 2 除了可以使用傳統開根號的方式[92]求得外，也可以將之改寫為分數型式來加以逼近。首先讓 $\sqrt{2} = 1 + x$，則

$$x = \sqrt{2} - 1 = \frac{1}{\sqrt{2}+1} = \frac{1}{1+\sqrt{2}} = \frac{1}{1+(1+x)} = \frac{1}{2+x}$$

等號右側的 x 重複利用 $x = \dfrac{1}{2+x}$ 取代，可得到：

$$x = \frac{1}{2+\dfrac{1}{2+x}} = \frac{1}{2+\dfrac{1}{2+\dfrac{1}{2+x}}} = \frac{1}{2+\dfrac{1}{2+\dfrac{1}{2+\dfrac{1}{2+\ddots}}}}$$

因此，$\sqrt{2}$ 可以使用以下的分數型式來表示：

$$\sqrt{2} = \mathbf{1} + \frac{1}{\mathbf{2}+\dfrac{1}{\mathbf{2}+\dfrac{1}{\mathbf{2}+\dfrac{1}{\mathbf{2}+\ddots}}}}$$

以上等號右側的分數型式被稱為連分數 (continued fraction)[註1]。請注意，這裡的連分

[註1] 更精確的說法為簡單連分數 (simple continued fraction)

數分子部份都為 1。為方便表示起見，一般習慣使用以下方式來代表連分數

$$[a_0, a_1, \cdots, a_n] = a_0 + \cfrac{1}{a_1 + \cfrac{1}{\ddots + \cfrac{\ddots}{a_{n-1} + \cfrac{1}{a_n}}}}$$

因此根號 2 若以連分數表示則為 $[\mathbf{1}, \mathbf{2}, \mathbf{2}, \cdots]$ 或是 $[\mathbf{1}, \overline{\mathbf{2}}]$。

根據數學理論，所有的無理數都可使用無窮項的連分數來表示，而有理數的連分數項數則為有限。若讓 c_n 代表連分數前 n+1 項所形成的分數，則當 n 越大時，c_n 會越逼近於此連分數所代表的數值，因此 c_n 被稱為此連分數的第 n+1 個漸近分數。若讓 a 陣列代表連分數 $[a_0, a_1, \cdots, a_n]$，則觀察以上的式子，漸近分數 c_n 可以使用以下的遞迴公式計算：

$$r_i = \begin{cases} a_n & i = 0 \\ a_{n-i} + \dfrac{1}{r_{i-1}} & i > 0 \end{cases}$$

以上所計算出來的 r_n 即為漸近分數 c_n 的數值。遞迴公式內的第一個條件為終結條件，另一個則為遞迴結構，因此直接將之寫成遞迴程式即為：

$\sqrt{2}$ 與連分數 — sqrt2.cc

```
01   #include <iostream>
02   #include <vector>
03   #include <cmath>              // 根號函式
04   #include <iomanip>
05
06   using namespace std ;
07
08   double  r( const vector<int>& a , int i ) {
09
10       int  n = a.size()-1 ;
11
12       if ( i == 0 )
13           return  a[n] ;
14       else
15           return  a[n-i] + 1./r(a,i-1) ;
16
17   }
18
19   int main() {
20
21       double  approx ;                      // 儲存估算值
22       double  sqrt2 = sqrt(2.) ;            // 計算根號 2
23
24       cout << "> 連分數估算 sqrt(2) : \n" ;
25
26       for ( int i = 0 ; i <= 15 ; ++i ) {
27
28           vector<int>  a(i+1,2) ;           // 設定連分數係數
```

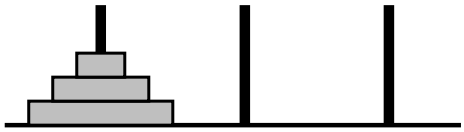
```
29          a[0] = 1 ;                          // 第一個元素改為 1
30
31          approx = r(a,i) ;                   // 連分數估算
32
33          cout << "c_" << left << setw(2) << i << " : "
34               << fixed << setw(10) << approx
35               << "  誤差 : " << right << scientific
36               << setw(14) << sqrt2-approx << endl ;
37
38      }
39
40      return 0 ;
41
42  }
43
```

執行結果

```
01  > 連分數估算 sqrt(2) :
02  c_0  : 1.000000    誤差 :   4.142136e-01
03  c_1  : 1.500000    誤差 :  -8.578644e-02
04  c_2  : 1.400000    誤差 :   1.421356e-02
05  c_3  : 1.416667    誤差 :  -2.453104e-03
06  c_4  : 1.413793    誤差 :   4.204589e-04
07  c_5  : 1.414286    誤差 :  -7.215191e-05
08  c_6  : 1.414201    誤差 :   1.237894e-05
09  c_7  : 1.414216    誤差 :  -2.123901e-06
10  c_8  : 1.414213    誤差 :   3.644036e-07
11  c_9  : 1.414214    誤差 :  -6.252177e-08
12  c_10 : 1.414214    誤差 :   1.072704e-08
13  c_11 : 1.414214    誤差 :  -1.840469e-09
14  c_12 : 1.414214    誤差 :   3.157747e-10
15  c_13 : 1.414214    誤差 :  -5.417822e-11
16  c_14 : 1.414214    誤差 :   9.295675e-12
17  c_15 : 1.414214    誤差 :  -1.594724e-12
18
```

為了讓輸出的數字對齊，在程式中使用到在第 335 頁中的輸入 / 輸出格式處理器。觀察程式的輸出數據，可以察覺到漸近分數是以上下振動的方式逐漸逼近連分數所代表的數值，這是漸近分數的一大特色。此外，另有遞迴公式可以用來算出漸近分數的分子與分母，但這裡將加以忽略。本題也可以直接以迴圈的方式撰寫，讀者可自行試試不用遞迴方式的寫法。連分數近來被人應用在曆法研究上，探討傳統中國曆法的精確度，非常有趣，有興趣者可利用 google 找尋相關文獻。

何內塔圓盤



何內塔問題為將左邊所有圓盤搬到右邊的步驟寫出，搬動的規則為每次圓盤只能搬動一個，且大圓盤一定要在小圓盤之下，圓盤只能由最上面的搬動。若要搬動 n 個圓盤，仔細分析其搬圓盤解決的步驟大致可以分為以下三大步：

1. 搬 n - 1 個圓盤由左邊到中間
2. 搬 1 個圓盤由左邊到右邊
3. 搬 n - 1 個圓盤由中間到右邊

原來的搬動 n 個圓盤的問題被巧妙地轉化成幾個較小的問題的組合，包含兩個搬動 n-1 個圓盤，與一個搬動一個圓盤的問題。當然對只要移動一個圓盤的問題，我們馬上就可以將步驟印出來，這同時也是函式的終結條件，所以整個程式可以寫成以下方式：

何內塔圓盤 hanoi.cc

```
01  #include <iostream>
02
03  using namespace std ;
04
05  // 搬圓盤程式
06
07  void move_disc ( int no , char from , char to , char through ) {
08      static int counter = 1 ;
09      if ( no == 1 )
10          cout << counter++ << " : "
11               << from << " --> " << to << endl ;
12      else {
13          move_disc( no-1 , from , through , to ) ;
14          move_disc( 1    , from , to , through ) ;
15          move_disc( no-1 , through , to , from ) ;
16      }
17  }
18
19  int  main() {
20
21      int  n = 3 ;
22
23      move_disc( n , 'L' , 'R' , 'M' ) ;
24
25      return 0 ;
26
27  }
28
```

執行結果

```
01   1 : L --> R
02   2 : L --> M
03   3 : R --> M
04   4 : L --> R
05   5 : M --> L
06   6 : M --> R
07   7 : L --> R
08
```

在上面的程式中，我們也刻意使用了一個靜態變數 counter 來當作步驟的計數器用，這是在遞迴程式中是常常被使用的方式。

遞迴包牌程式

在第 134 頁中，我們用簡單的迴圈來找出樂透包牌的所有組合，所謂的包牌問題與數學上的組合問題是一樣的。即是要找出由 m 個數字中取出 n 個數字的所有組合，其總數為

$$C_n^m = \frac{m!}{(m-n)!\,n!}$$

在之前的迴圈程式中，為了從 m 個數字中拿取 n 個數字，我們須要在程式寫入 n 層迴圈，如果 n 的數值改變了，則程式碼也須要將迴圈的數目改變，如此的程式碼會因資料的不同而須作人為的更動，在程式設計上是不切實際的。但這個問題可以使用遞迴函式來克服，在迴圈的程式中，可以發現到每一層迴圈所須處理的問題是類似的，因此我們可以將之轉成遞迴函式。設計的主軸為使用一個遞迴函式來替代每一層迴圈，原始的迴圈程式若須要 n 層迴圈，則遞迴的深度就有 n 層，限制遞迴的深度就可以當成遞迴程式的終結條件。

在程式中，我們使用了向量陣列 (vector) 來傳遞遞迴函式的參數，其使用的方法且一般的陣列型別的變數並無差別，但其與一般的陣列不同之處在於向量陣列並不須要將陣列長度當作參數傳入，而可以直接使用向量陣列的長度函式 (size()) 即可，這可以簡化函式參數列。為了方便計算答案的總數，在遞迴程式中，也使用靜態變數來記錄答案的數目。

遞迴式樂透包牌　　combination.cc

```
01   #include <iostream>
02   #include <vector>
03   #include <iomanip>
04
05   using namespace std ;
06
07   // 樂透包牌程式
```

```
08  void print_combination( const vector<int>& number ,
09                          vector<int>& chosen_no ,
10                          int depth , int start_index ) {
11
12      int i ;
13      static  int  count = 0 ;
14
15      // 當遞迴的深度與找到的號碼數相同
16      if ( depth == chosen_no.size() ) {
17
18          // 輸出數字組合
19          cout << setw(3) << ++count << " : " ;
20          for ( i = 0 ; i < chosen_no.size() ; ++i )
21              cout << setw(4) << chosen_no[i] ;
22          cout << endl ;
23
24      } else {
25
26          // 遞迴式迴圈, 深一層的遞迴初始下標值會自動加一
27          for ( i = start_index ; i < number.size() ; ++i ) {
28              chosen_no[depth] = number[i] ;
29              print_combination( number , chosen_no , depth+1 , i+1 ) ;
30          }
31
32      }
33
34  }
35
36  int main() {
37
38      int  no ;
39      cout << "> 輸入總號碼數量 : " ;
40      cin  >> no ;
41
42      // number  : 儲存所有的號碼
43      vector<int>  number(no)  ;
44
45      cout << "> 輸入 " << no << " 個號碼 : " ;
46      for ( int i = 0 ; i < no ; ++i ) cin >> number[i] ;
47
48      cout << "> 輸入包牌碼數量 : " ;
49      cin  >> no ;
50
51      // chosen_no  : 已找到的號碼
52      vector<int>  chosen_no(no) ;
53
54      // depth       : 遞迴深度
55      // start_index : 遞迴迴圈中的初始下標值
56      int  depth = 0 , start_index = 0 , count = 0 ;
57      print_combination( number , chosen_no , depth , start_index ) ;
58
59      return 0 ;
60
61  }
62
```

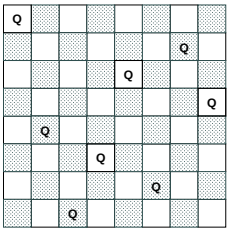
執行結果

```
> 輸入總號碼數量 : 7
> 輸入 7 個號碼 : 5 10 12 20 33 35 42
> 輸入包牌碼數量 : 6
  1 :    5  10  12  20  33  35
  2 :    5  10  12  20  33  42
  3 :    5  10  12  20  35  42
  4 :    5  10  12  33  35  42
  5 :    5  10  20  33  35  42
  6 :    5  12  20  33  35  42
  7 :   10  12  20  33  35  42

```

八個皇后問題

在西洋棋八乘八的棋盤上，皇后 (queen) 可以橫衝直撞又可斜行，是所有的棋子中最具威力的。八個皇后問題是指要在棋盤上擺上八個皇后而且彼此互相不干擾。如下圖：



由於皇后的特殊的走法，每一行與每一列剛好只能放置一個皇后。這個問題剛好可以使用八層迴圈來處理，若用遞迴方式來設計，可以將處理方式寫成以下的方式：

假設我們的策略為由第一行 (column) 起由左而右，由上而下尋找適當的列位置放置皇后，則放置皇后的遞迴步驟可以簡單地寫成以下的方式：

1. 在本行中由上而下選擇一個適當位置放置皇后，如果不能則回上一行
2. 進入下一行放置皇后

八個皇后：遞迴程式　　queen.cc

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

using namespace std ;

// 使用全域變數

// S : 棋盤大小 , queen : 皇后位置
```

```
09  const int  S = 8 ;
10  bool  queen[S][S] = { { false }, { false }, { false }, { false },
11                        { false }, { false }, { false }, { false } };
12
13  // 計算絕對值
14  int  abs( int x ) { return x > 0 ? x : -x ; }
15
16  // 檢查新的皇后若擺放在 (r,c) 位置是否適當
17  bool  valid_position(int r , int c ) {
18
19      int i , j , dx , dy ;
20
21      // 檢查是否與已有的皇后在同一行, 同一列或在對角線上
22      for ( j = 0 ; j < c ; ++j )
23          for ( i = 0 ; i < S ; ++i ) {
24              if ( queen[i][j] ) {
25                  dx = abs(r-i) ;
26                  dy = c - j ;
27                  if ( dx == 0 || dy == 0 || dx == dy ) return false ;
28              }
29          }
30      return true ;
31
32  }
33
34  // 列印所有的皇后位置
35  void  print_queen(int counter) {
36      int i , j ;
37      cout << "\n" << setw(7) << "[" << setw(3) << counter << "]\n" ;
38      for ( i = 0 ; i < S ; ++i ) {
39          for ( j = 0 ; j < S ; ++j )
40              cout << setw(2) << ( queen[i][j] ? "Q" : "+" ) ;
41          cout << endl ;
42      }
43  }
44
45  // 遞迴找尋皇后位置
46  void  locate_queen( int col ) {
47
48      int  row ;
49      static int  counter = 0 ;
50
51      if ( col == S ) {
52
53          print_queen(++counter) ;
54
55      } else {
56
57          for ( row = 0 ; row < S ; ++row ) {
58              if ( valid_position(row,col) ) {
59                  queen[row][col] = true ;
60                  locate_queen(col+1) ;
61                  queen[row][col] = false ;
62              }
63          }
64
65      }
66  }
67
```

```
68   // 主函式
69   int  main() {
70
71       int  col = 0 ;
72
73       locate_queen(col) ;
74
75       return 0 ;
76
77   }
78
```

執行結果：部份輸出

```
01          [  1]
02    Q + + + + + + +
03    + + + + + + Q +
04    + + + + Q + + +
05    + + + + + + + Q
06    + Q + + + + + +
07    + + + Q + + + +
08    + + + + + Q + +
09    + + Q + + + + +
10
11          [  2]
12    Q + + + + + + +
13    + + + + + + Q +
14    + + + Q + + + +
15    + + + + + Q + +
16    + + + + + + + Q
17    + Q + + + + + +
18    + + + + Q + + +
19    + + Q + + + + +
20
21          [  3]
22    Q + + + + + + +
23    + + + + + Q + +
24    + + + + + + + Q
25    + + Q + + + + +
26    + + + + + + Q +
27    + + + Q + + + +
28    + Q + + + + + +
29    + + + + Q + + +
30
31    .....
32
```

在遞迴程式中的主要迴圈中，首先程式先檢查是否新的位置為一個適合擺放皇后的位置，如果是，則記錄位置，繼續往右一行執行此遞迴函式，直到第八行時，則將結果印出，每當一遞迴函式執行完畢時 (可能是解答已經找到或者是找不到適當的新位置放置皇后)，為了繼續往下找解答，我們須要將之前所記錄皇后位置去除。這也

是一般所採用的方法。同樣類型的遞迴題目有許多，請參考本章後的習題。

在程式中，我們將放置皇后的陣列及其大小設為全域變數 (global variable)，藉以讓所有的函式都能自由存取，而不須要藉由參數來傳遞，避免造成複雜的參數列。但全域變數的使用要盡量減少，以免造成程式變數名稱的混淆。

如果程式改用迴圈方式處理，則會變成以下複雜樣式，相對而言，這提供了程式設計員使用遞迴方式來設計程式的很好的理由。

八個皇后：迴圈程式 queen_by_loop.cc

```
001   #include <iostream>
002   #include <iomanip>
003
004   using namespace std ;
005
006   // 使用全域變數
007
008   // S : 棋盤大小 , queen : 皇后位置
009   const int  S = 8 ;
010   bool  queen[S][S] = { { false }, { false }, { false }, { false },
011                         { false }, { false }, { false }, { false }
012                       } ;
013
014   // 計算絕對值
015   int  abs( int x ) { return x > 0 ? x : -x ; }
016
017   // 檢查皇后在(r,c)位置是否為適當位置
018   bool  valid_position(int r , int c ) {
019       int i , j , dx , dy ;
020       for ( j = 0 ; j < c ; ++j )
021           for ( i = 0 ; i < S ; ++i ) {
022               if ( queen[i][j] ) {
023                   dx = abs(r-i) ;
024                   dy = c - j ;
025                   if ( dx == 0 || dy == 0 || dx == dy )
026                       return false ;
027               }
028           }
029       return true ;
030   }
031
032   // 列印所有皇后的位置
033   void  print_queen(int counter) {
034       int i , j ;
035       cout << "\n" << setw(7) << "[" << setw(3) << counter << "]\n" ;
036       for ( i = 0 ; i < S ; ++i ) {
037           for ( j = 0 ; j < S ; ++j )
038               cout << setw(2) << ( queen[i][j] ? "Q" : "+" ) ;
039           cout << endl ;
040       }
041   }
042
043   // 主函式
044   int  main() {
045
046       int  counter = 0 ;
```

```
047       int  c1 , c2 , c3 , c4 , c5 , c6 , c7 , c8 ;
048       int  r1 , r2 , r3 , r4 , r5 , r6 , r7 , r8 ;
049
050       // 八層迴圈
051       for ( c1 = 0 , r1 = 0 ; r1 < S ; ++r1 ) {
052
053         if ( valid_position(r1,c1) ) {
054           queen[r1][c1] = true ;
055
056           for ( c2 = c1+1 , r2 = 0 ; r2 < S ; ++r2 ) {
057             if ( valid_position(r2,c2) ) {
058               queen[r2][c2] = true ;
059
060               for ( c3 = c2+1 , r3 = 0 ; r3 < S ; ++r3 ) {
061                 if ( valid_position(r3,c3) ) {
062                   queen[r3][c3] = true ;
063
064                   for ( c4 = c3+1 , r4 = 0 ; r4 < S ; ++r4 ) {
065                     if ( valid_position(r4,c4) ) {
066                       queen[r4][c4] = true ;
067
068                       for ( c5 = c4+1 , r5 = 0 ; r5 < S ; ++r5 ) {
069                         if ( valid_position(r5,c5) ) {
070                           queen[r5][c5] = true ;
071
072                           for ( c6 = c5+1 , r6 = 0 ; r6 < S ;
073                                 ++r6 ) {
074                               if ( valid_position(r6,c6) ) {
075                                   queen[r6][c6] = true ;
076
077                                   for ( c7 = c6+1 , r7 = 0 ; r7 < S ;
078                                         ++r7 ) {
079                                      if ( valid_position(r7,c7) ) {
080                                        queen[r7][c7] = true ;
081
082                                        for ( c8 = c7+1 , r8 = 0 ;
083                                              r8 < S ; ++r8 ) {
084                                          if ( valid_position(r8,c8) ) {
085                                              queen[r8][c8] = true ;
086                                              print_queen(++counter) ;
087                                              queen[r8][c8] = false ;
088                                          }
089                                        }
090                                        queen[r7][c7] = false ;
091
092                                      }
093                                   }
094                                   queen[r6][c6] = false ;
095
096                               }
097                           }
098                           queen[r5][c5] = false ;
099
100                         }
101                       }
102                       queen[r4][c4] = false ;
103
104                     }
105                   }
```

```
106                   queen[r3][c3] = false ;
107
108                 }
109               }
110               queen[r2][c2] = false ;
111
112             }
113           }
114           queen[r1][c1] = false ;
115
116         }
117       }
118
119       return 0 ;
120
121   }
122
```

輸出的結果同上例，這裡不再重複列印。

8.12 結語

早期的程式設計是以函式設計為主，程式設計員會將一個複雜的程式設計問題首先分解成一個個小問題，而每個小問題可以用一個簡單的函式來取代，如此一來，整個程式就是由許多個函式組合而成。程式設計就是函式設計。撰寫正確有效率的函式在早期的程式設計上是相當重要的。以函式為主的程式設計，其變數資料與處理資料的函式是分開設計的，當程式越來越龐大時，變數資料也會越來越多，在這種情況之下，資料常常會被不同的函式所使用或是更動，使得計算結果的正確性容易受到質疑。這與後來的程式設計的理念是有所不同的。雖然如此，嫻熟的利用函式來設計程式仍是程式設計中一個相當重要的基本功夫。

8.13 練習題

1. 下面的程式片段都有些錯誤，請找出錯誤的地方？

(a)

```
int foo(int& a){ ...  }
// 呼叫方式
int i = 1 , j = 2 ;
foo( i+j ) ;
```

(b)

```
int foo( int i ) {
    return i == 1 ?  "one" :  0 ;
}
```

(c)

```
int print( int a ) {
    cout << a ;
}
```

(d)

```
int abs( const int& a ) {
    if ( a < 0 ) a = -a ;
    return a ;
}
```

2. 以下的函式程式碼有錯嗎，原因為何？

```
string  username( const string& name ) {
    cout << "username = " << name ;
}
```

3. 某生寫了一個函式用來比較兩個的字串大小，假設兩字串的最長字串長為 S，請問以下的寫法有錯嗎？

```
bool  smaller( char a[] , char b[] , int S ) {
    for ( int i = 0 ; i < S ; ++i )  return  a[i] < b[i] ;
}
```

4. 某生寫了一個函式，但總覺得不對勁，他的函式標頭部份為 `int &foo(int a)`，請問是哪裡不對勁？

5. 若兩函式被定義成：

```
int &  max ( int& a , int& b ) { return a > b ? a : b ; }
int &  min ( int& a , int& b ) { return a < b ? a : b ; }
```

若 i，j 為兩整數，則 max(i,j) = min(i,j) + max(i,j) 是什麼意思？

6. 若定義一函式為 `int& max( int i , int j )`，可回傳兩輸入整數的最大值，假設 a = 3 與 b = 4，請問此函式可否執行右式 max(a,b) = 10 ？

7. 假設 min 函式被定義為：

```
int&  min ( int &a , int &b ) { return a < b ? a : b ; }
```

請問如果 a = 1，b = 3，c = 9，則連續執行兩次 min(a,b) += c 後，a，b，c 的值分別為何？

8. 尤拉 (Euler) 法是一種簡單數值方法，專門用來計算常微分方程式初始值問題。假設有一微分方程式為 $y'(x) = f(x, y(x))$，其初始值為 $y(0) = a$，此微分方程式若以幾何方式來解釋則表示解答的圖形在 (x, y) 點上的斜率等於函式 $f(x, y(x))$ 之值。假設我們在 X 軸上有若干等距的點，分別為 $x_0, x_1, x_2, \ldots, x_n$，其間距為

h，為方便起見讓 $x_0 = 0$，則在點 (x_n, y_n) 的斜率，就可以大概用 $\frac{y((n+1)h)-y(nh)}{h}$ 來估算，這裡的 y_n 為 $y(nh)$，代入微分方程式後可以得到

$$y_{n+1} = y_n + h\,f(x_n, y_n) \quad 且 \quad y_0 = a$$

請用此方法計算 $y' = x + y$ 與初始值 $y(0) = 0$，在 $y(10)$ 的數值解，假設 $h = 0.1$？

9. 不用遞迴方式，請寫一個程式用輾轉相除法求兩個正數的最小公倍數。

10. 請寫一個程式將輸入的十進位整數改以不同的進位方式輸出，例如：

十進位數字	二進位數字	五進位數字	八進位數字	十六進位數字
100	1100100	400	144	64
200	11001000	1300	310	C8

十進位以上的數字以英文字母取代，即數字 10 以 A，11 以 B 取代，其它依此類推，以數字 200 為例：

$$\begin{aligned} 200 &= 2 \times 10^2 + 0 \times 10^1 + 0 \times 10^0 \\ &= 1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + \\ &\quad 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \\ &= 1 \times 5^3 + 3 \times 5^2 + 0 \times 5^1 + 0 \times 5^0 \\ &= C \times 16^1 + 8 \times 16^0 \end{aligned}$$

11. 請寫一個程式將阿拉伯數字金額轉成中文方式表示，即：

```
7609802      ⇒    柒佰陸拾萬玖仟捌佰零貳元整
1009040300   ⇒    拾億玖佰零肆萬參佰元整
```

12. 請用字串方式，計算兩字串數字的減法，須注意有負數的存在？

13. 請用字串方式，將兩字串數字的乘法過程印出，即：

```
          9876543210
×              321
----------------------
          9876543210
         19753086420
        29629629630
----------------------
       3170370370410
```

14. 猜數字遊戲是中學生常玩的紙上遊戲 (規則請參閱第 149 頁中的比對數字習題)，請寫一個程式由程式來猜測數字。例如假設被猜測的數字為 1428，則程式執行過程為：

	電腦	結果			電腦	結果
(1)	8479	1A1B		(2)	1632	1A1B
(3)	8614	3B		(4)	1428	4A

(提示：可以使用蠻力法[79]來解決，將適合的四位數由小到大一一測試)

15. 請寫一個程式，由命令行輸入一任意長度的數字當作主函式的參數，然後由右至左每三個數字加一逗點的方式印出此數。例如：假設此可執行檔名稱為 `transform`，則在命令行輸入 `transform 915348130234`，則程式輸出 `915,348,130,234`。

16. 請參考連分數[216]的說明，撰寫一個程式找出輸入分數所代表的連分數型式，即為 $[a_0, a_1, \cdots, a_n]$。例如：

$$\frac{37}{8} = 4+\frac{5}{8} = 4+\frac{1}{\frac{8}{5}} = 4+\frac{1}{1+\frac{3}{5}} = 4+\frac{1}{1+\frac{1}{\frac{5}{3}}} = 4+\frac{1}{1+\frac{1}{1+\frac{2}{3}}}$$

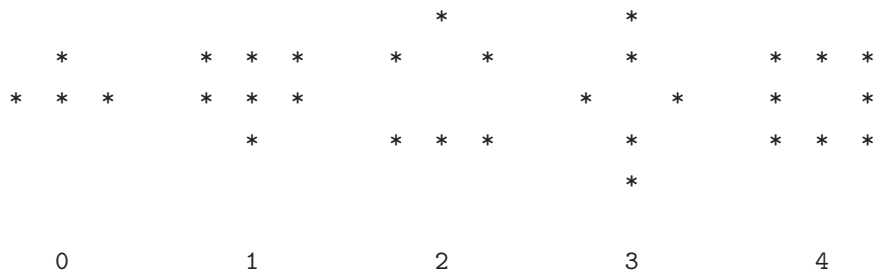
$$= 4+\frac{1}{1+\frac{1}{1+\frac{1}{\frac{3}{2}}}} = \mathbf{4}+\frac{1}{\mathbf{1}+\frac{1}{\mathbf{1}+\frac{1}{\mathbf{1}+\frac{1}{\mathbf{2}}}}}$$

因此 $\frac{37}{8} = [4, 1, 1, 1, 2]$

17. 請寫一個函式將一整數傳統字串轉成一個整數，例如：

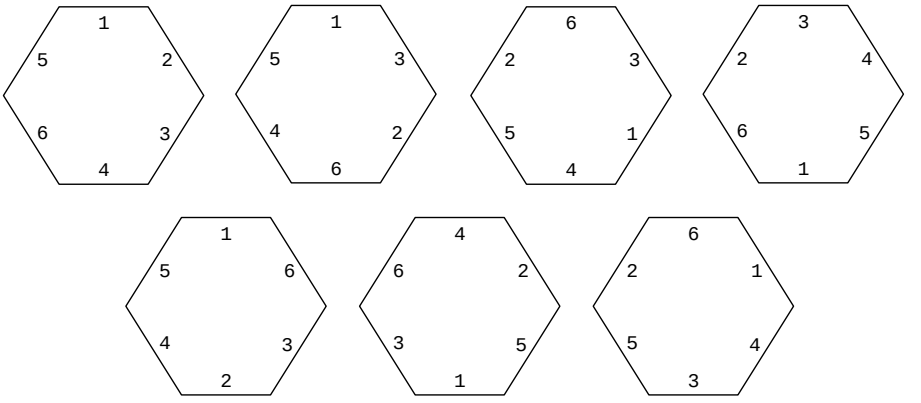
```
char*  num = "1234" ;
int    no  = ascii2int( num ) ;   // 轉成整數
cout <<  2 * no << endl  ;        // 印出 2468
```

18. 以下為 1970 年刊登在 scientific American 雜誌中的一個生物族群生態模擬遊戲，假設有一生物族群在某二維區域的格子生長，每個格子有八個緊鄰的格子，且每一個格子可以生長一生物體，此生物族群的生長與死亡規則如下：當某空格其緊鄰的格子若剛好共有 3 個生物體，則此空格會在下一代時產生出新的生命，若格子內的生物其鄰格共有 2 或 3 個生物體，則此生物體在下一代模擬時會繼續存活，若是其它種情況，則此生物體會因孤獨或者過度擁擠而死亡。舉例來說：

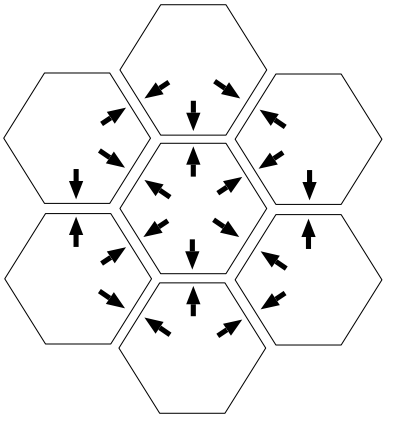


請設計一程式，輸入此生物體初始的族群分佈資料，模擬生物體之後族群的變化。

19. 有六角 IQ 拼圖如下：



請將此七張六角拼圖排成右式圖樣，使得所有兩兩相鄰的六角拼圖邊上的數字均相同，請寫一個程式將解答找出。



20. 某小學老師欲出 50 題兩個分數之間的加減乘除給學生練習，請為其撰寫一個程式產生此 50 道數學式子，請留意，小學生尚未有負數的觀念，且為避免數學式子太複雜，所有的分子與分母 (包含答案) 都要在 1 到 9 之間的正整數，此外分子不能等於分母，輸出時分子與分母須約分，且答案在約分後分子與分母也要在 1 到 9 之間，不能為負數，其輸出格式如下：

$$(1)\ \frac{3}{5}+\frac{1}{5} \qquad (2)\ \frac{1}{3}\div\frac{5}{9}$$

$$(3)\ \frac{7}{4}-\frac{1}{4} \qquad (4)\ \frac{5}{7}\times\frac{3}{5}$$

21. 請寫一個排列程式印出一個陣列元素所有的排列。如陣列為 { 1 , 2 , 3 }，則輸出為：

```
(1) 1  2  3          (2) 1  3  2
(3) 2  1  3          (4) 2  3  1
(5) 3  1  2          (6) 3  2  1
```

22. 古代羅馬人以字母來表示數字，例如：

```
十進位字母 :  I = 1  ,  X = 10  ,  C = 100  ,  M = 1000
其它       :  V = 5  ,  L = 50  ,  D = 500
```

超過 1000 以上的羅馬字母數字因無法以一般電腦的輸出字母顯示，在此忽略。羅馬數字的規則為：

(1) 字母書寫時由大到小排列，最終所代表的數字以加減法計算求得

```
MCLVI = 1000 + 100 + 50 + 5 + 1 = 1156
```

(2) 十進位字母數字最多可重複 3 次，其它則僅能使用 1 次

```
MMM = 3000  ,  CC = 200 , III = 3
```

(3) 當小的十進位字母數字寫在較大數之前，其數字和為大數減去小數

```
XL = 50 - 10 = 40
XC = 100 - 10 = 90
```

例如：

```
(1) 990  = 900 + 90 = ( 1000 - 100 ) + ( 100 - 10 )
         = CMXC

(2) 3848 = 3000 + 800 + 40 + 8
         = 3000 + ( 500 + 300 ) + ( 50 - 10 ) + ( 5 + 3 )
         = MMMDCCCXLVIII
```

請寫一個程式列印 3999 以下的羅馬數字。

23. 承上題說明，請寫一程式，在輸入一羅馬數字後，計算出其所對應的阿拉伯數字，例如：

```
(1)  XXXVIII          38
(2)  LXXXVIII         88
(3)  MMMCMLXXXIII     3983
```

24. 某城市盃動運比賽，主辦單位欲將各城市所得到的獎牌數分別依金，銀，銅牌個數，排列印出城市得獎順序，請寫一個程式來模擬。(註：排序程式可參考第 135 頁的插入排序法)，輸出：

名次	金牌	銀牌	銅牌	城市
1	12	13	12	台中
2	10	24	21	高雄
3	7	30	8	台北
...				

25. 請寫一個程式模擬虛擬的統一發票對獎方式，假設發票號碼共有八位數，每期共開出五個號碼，分別為一個特獎號碼與四個頭獎號碼。特獎為特獎號碼八位全對，頭獎為對中頭獎號碼的任一個，二獎為對中任一個頭獎號碼的末七碼，三獎為對中任一個頭獎號碼的末六碼，四獎，五獎及六獎分別為對中任一個頭獎號碼的末五，末四與末三碼。特獎彩金為兩佰萬元，頭獎為二十萬元，二獎為四萬元，三獎為一萬元，四獎為四千元，五獎為一千元，六獎為二佰元。請任意設定中獎號碼，及 100000 組發票號碼，印出中獎的種類及個別金額與總金額？

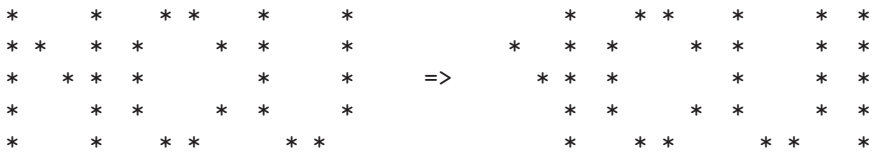
26. 以下是博奕論中著名的兩個囚犯困境案例：

    有兩個一起做壞事的歹徒被警方逮捕，分別被關在無法溝通訊息的兩間牢房接受審問。但警方一直找不到犯罪證據，這兩個囚犯都知道如果兩人都保持沉默，則最後兩人都可以無罪獲釋，警方也明白這點，因此給兩個囚犯一些刺激：如果兩人中有一人背叛，告發其同伙，則其同伙將被罰款且判重刑，而告發的人除可以無罪釋放外，又可以得到一筆獎金。但如果兩個人互相告發，則兩個都會被判重刑。

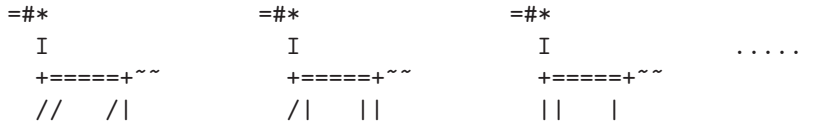
    現假設兩人分別為 A 與 B 兩囚犯，A 囚犯採取以合作開始，但以牙還牙方式為回報策略，請為 B 囚犯設計一套對應方式，試試是否可以在重複 1000 次的比較中獲得較好的成績。假設成績分數計算為無罪釋放得 1 分，獲取獎金得 1 分，被判刑扣 1 分，被罰款扣 1 分。也就是說，如果兩人中只有一個人告發，則告發的人得 2 分，被告發的人扣 2 分。

27. 在標頭檔 time.h (或者是 ctime) 中，有一個 time 函式可以回傳由 1970 年 1 月 1 日零時零刻起到執行時刻的總秒數，使用方式為 time()，請利用此函式設計一個計時用的二進位馬錶，此馬錶僅能顯示 4 個 0 或 1 的數字，若輸入一秒數 6，此馬錶可以依次顯示，0000、0001、0010、0011、0100、0101。請利用 7 x 5 的點矩陣方式來儲存數字 0 或 1。

28. 請利用第 83 頁中所介紹的 ANSI 跳離序列，撰寫文字跑馬燈的程式，使得文字點慢慢的往左移動，消失的點由右邊出現：



29. 請利用第 83 頁中所介紹的 ANSI 跳離序列控制螢幕的輸出，設計一匹馬各個跑動的動作，撰寫程式使得馬可以由螢幕的右邊跑到螢幕的左邊，如下圖：



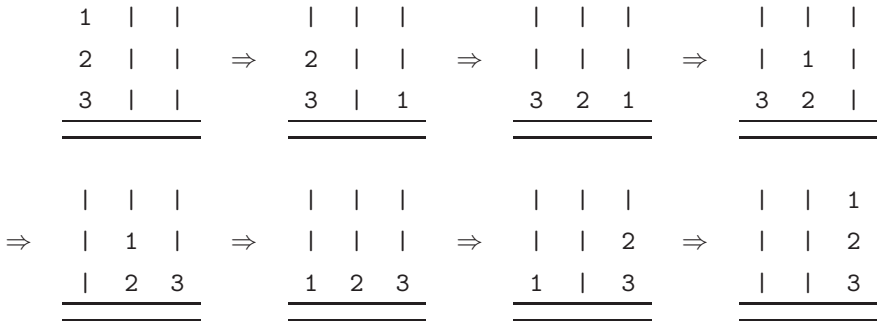
30. 請參考第 152 頁的畫函式圖形作業習題，設計一函式，畫出任意輸入的函式參數的圖形，請使用以下的形式：

```
plot_fn( sin , 0 , 6 ) ;   // 畫出 sin 函式於 [0,6] 之間的圖形
plot_fn( fn , -3 , 3 ) ;   // 畫出 fn 函式於 [3,3] 之間的圖形
```

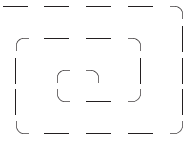
31. 輸入一正整數，請使用遞迴函式，以質因數乘積方式表達此數。

32. 請參考連分數[216]的說明，使用單純的迴圈方式撰寫函式計算根號 2 的逼近值。

33. 請寫一個程式將何內塔的解法步驟用圖型的方式印出來，例如：



34. 請用遞迴方式印出以下的圖型，假設圖型寬度為輸入值，例如下圖寬度為 5：



35. 請用遞迴方式分別印出以下的圖型，假設圖形最中間的數值為程式的輸入值：

(a)

```
1 1 1 1 1 1 1
1 2 2 2 2 2 1
1 2 3 3 3 2 1
1 2 3 4 3 2 1
1 2 3 3 3 2 1
1 2 2 2 2 2 1
1 1 1 1 1 1 1
```

(b)

```
    1
   1 1
  1 2 1
 1 2 2 1
1 2 3 2 1
1 2 2 2 2 1
1 1 1 1 1 1 1
```

36. 請用遞迴方式分別印出以下的數字螺旋圖形，假設高度 n 為輸入值。

(a)

```
1 2 3 4 5 6 7
4 5 6 7 8 9 8
3 0 1 2 3 0 9
2 9 8 9 4 1 0
1 8 7 6 5 2 1
0 7 6 5 4 3 2
9 8 7 6 5 4 3
```

(b)

```
    1
   8 2
  7 9 3
 6 7 0 4
5 6 8 1 5
4 5 4 3 2 6
3 2 1 0 9 8 7
```

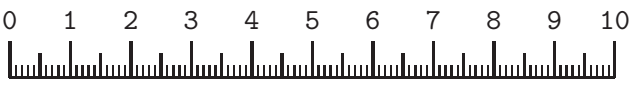
37. 請利用第 83 頁中所介紹的 ANSI 跳離序列控制螢幕的輸出，將上題的輸出依螺旋方式顯示出來？

38. 請寫一個遞迴程式在一個 5 × 5 的方格子內分別依次放入 1 到 25 的數字，任兩個數字之間是以日字形的對角線的方式擺放，例如：

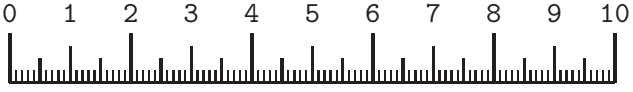
1	12	19	6	25
18	7	2	11	16
13	20	17	24	5
8	3	22	15	10
21	14	9	4	23

39. 一般中小學生用的直尺，其刻度線通常是以公分為最大單位，為利於辨視，常常以最長的刻度線來表示，而在每公分的中間刻度線則會以較短的刻度線表示，最後才在其間以每 1 公釐的間距顯示出最小刻度線，如下圖：

以一公分為最大刻度



以兩公分為最大刻度



請分別利用非遞迴與遞迴函式的方式來列印刻度長短有別的直尺。

40. 若有 r 種不同顏色的球放入袋中，今欲由袋中取出 n 個球，假設每種顏色的球都有 m 個，且其數量都至少 n 個，請寫一個程式印出所有可能的取法。

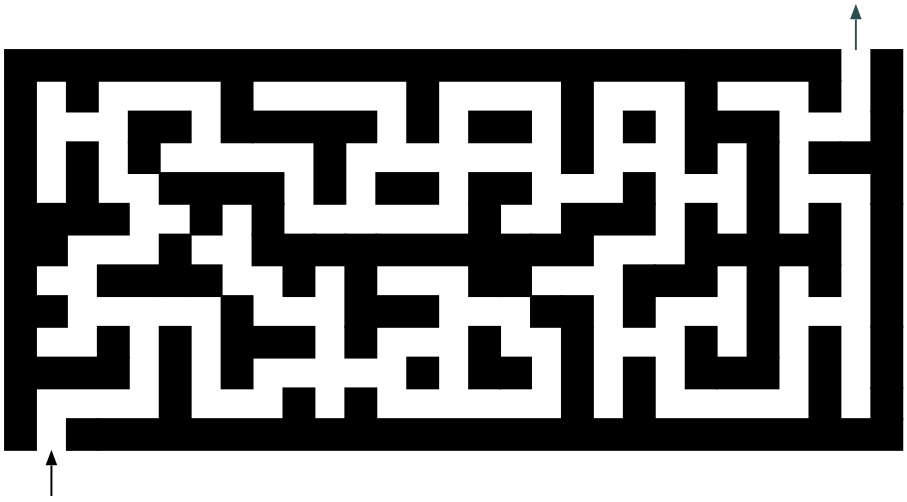
41. 請用遞迴方式將兩正數的最大公因數 (gcd) 求出，並將其轉成對應的程式碼，同時請寫另一函式計算兩此數的最小公倍數 (lcm)？

42. 在數學上常用輾轉相除法來求最大公因數，請寫一個程式將兩個正數的輾轉相除法過程，與最大公因數 (gcd) 印出，例如：

```
1 |       1 2 0 |         7 8 |  1
  |         7 8 |         4 2 |
  | ----------- | ----------- |
1 |         4 2 |         3 6 |  6
  |         3 6 |         3 6 |
  | ----------- | ----------- |
  |           6 |           0 |

gcd(120,78) = 6
```

43. 請利用矩陣來儲存以下迷宮資料，用遞迴方式撰寫一老鼠走迷宮程式，並將結果印出。



44. 請利用第 83 頁中所介紹的 ANSI 跳離序列，將上一題迷宮的行走路徑由起點至終點慢慢地顯示出來。

45. 若要對某序列排序可使用以下遞迴策略，請依此策略撰寫遞迴排序程式：

    1. 對前半段序列排序
    2. 對後半段序列排序
    3. 將前後段已排好次序的序列依元素大小重新組合

    此排序法被稱為合併排序法。(merge sort)

46. 請一個遞迴程式將某一正數字用正數數和展開，並將所有可能印出。例如：

```
4 = 4
4 = 3 + 1
4 = 2 + 2
4 = 2 + 1 + 1
4 = 1 + 3
4 = 1 + 2 + 1
4 = 1 + 1 + 2
4 = 1 + 1 + 1 + 1
```

47. 撰寫程式找出由 6 個數字 6 所構成的各種數字組合，此組合在經過基本的加減乘除依次運算後可以整除 100，請印出滿足的數學式子，例如：

```
(1) ( 666 - 66 ) / 6 = 100        (2) 6666 - 66 = 6600
```