

Chapter 8

函式

函式基本成員

```
y = sin(x) ;
```

- 使用名稱：`sin`
- 傳入參數：`x`
- 演算的程式碼：用來計算 `sin(x)` 的數值
- 回傳資料：`y`

函式基本概念 (一)

■ 某程式片段

```
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 ;
...
```

函式基本概念 (二)

- 若函式 `power(a,n)` 專門用來計算 a^n 的數值

```
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 ;
...
```

❖ 適當的使用函式可用來簡化程式碼

函式基本架構（一）：簡介

■ `power(a,n)` 函式：

// 函式計算浮點數 `a` 的 `n` 次方，這裡 `n >= 0`

```
④      ①      ②
double power( double a , unsigned int n ) {
    double p = 1.0 ;
    ③    for( int i = 0 ; i < n ; ++i ) p *= a ;
    return p ;
}
```

① 函式名稱

③ 函式主體

② 參數列

④ 回傳型別

函式基本架構（二）：參數列

- 參數列：為函式與外部程式碼的介面

若定義

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

呼叫函式

```
double no = 2.0 ;  
cout << power(no,4) << endl ;
```

則參數列執行

```
double a = no ;  
int n = 4 ;
```

- 函式也可以不輸入任何參數，但仍須保留小括號

```
int fn() ;
```

或寫成

```
int fn(void) ;
```

函式基本架構（三）：回傳型別

- 回傳型別：函式在結束前回傳給呼叫程式的資料型別
- 回傳函式內資料：使用 **return**

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

- **return** 敘述可在函式任意一行，也可多次出現

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

函式基本架構（四）：回傳型別

- 若函式不須回傳資料，則回傳型別為 `void`，同時 `return` 之後不須接任何變數資料

```
// 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，也可省略 `return` 敘述

函式原型 (一)

■ 函式原型： 去除函式執行程式碼的函式部份

//(1) 函式回傳所輸入兩整數 a 與 b 最大值

```
int max( int a , int b ) ;
```

//(2) 函式回傳所輸入的美金所能兌換的臺幣金額

```
double NT( double US_dollar ) ;
```

//(3) 函式回傳所輸入浮點數的整數次方值

```
double power( double , int ) ;
```

❖ 函式原型的參數名稱也可以省略

function prototype

函式原型 (二)

- 函式原型是用來宣告函式的存在，可以在程式碼中重複出現

```
int abs( int ) ;                // 函式 abs 原型

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

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

- 函式原型相當於函式的宣告，函式內部程式碼則相當於函式的定義

函式特徵 (一)

- 函式特徵：去除回傳型別的**函式原型**部分
- 作用：C++藉由比對函式的特徵資料，來判斷兩個函式是否相同

// (1) 找出兩整數的最大值

```
int      max( int a , int b ) ;
```

// (2) 找出兩浮點數的最大值

```
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 (1), (2) 參數型別不同
(1), (3) 參數總數不同

function signature

函式特徵 (二)

- 若兩函式的函式特徵相同，則函式會被視為重複，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
    ...
}
```

參考資料型別 (一)

- 參考：一筆資料有多個使用名稱

```
int foo = 3 ;
```

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

如此使用 `foo` 與 `bar` 是相同的效果

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

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

❖ 參考變數使用原變數(被參考變數)的記憶空間儲存資料

reference

參考資料型別 (二)

- 參考變數在產生之際就須依附著被參考變數

```
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**，則程式不能透過參考變數來改變原始變數的數值

```
int          foo = 3    ;  
const int &bar = foo ;  // 分身 bar 為本尊 foo 的常數參考  
++bar        ;        // 錯誤，分身 bar 被視為常數  
++foo        ;        // 正確，本尊 foo 仍可照常修改
```

參考與指標（一）

- 兩者背後都隱藏了一個記憶空間，可用來儲存資料

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

指標

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

參考

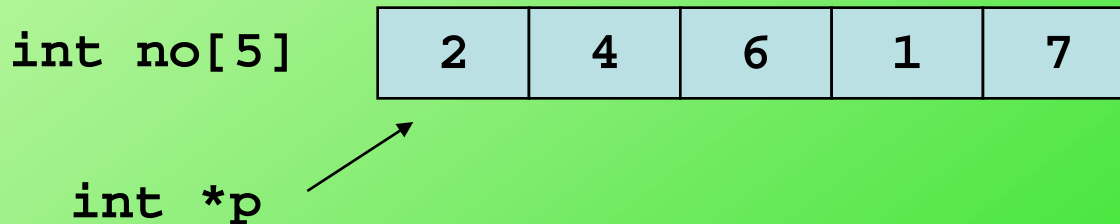
```
int &foo = a ;           // foo 為 a 的參考
foo = 30 ;               // 相當於 a = 30
&foo = b ;               // 錯誤， foo 不能更改所參照的變數
```

參考與指標 (二)

- 參考變數在定義時，就須設定所要參照的變數，且之後不可改變，但指標卻可以

```
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 ;
```



指標的參考 (一)

■ 指標的參考：參考的型別為指標

```
int a = 3 ;
```

```
int * p = &a ;           // 指標 p 指向 a 元素
```

```
int * &q = p ;           // 指標 q 為 p 指標的參考
```

■ 指標的參考也是指標，與被參照的指標為同一個指標

```
int a = 3 ;
```

```
int * p = &a ;
```

```
int * &q = p ;           // 指標 q 為 p 指標的參考
```

```
q = new int(5) ;         // 指標 q 指向一動態空間內存 5
```

```
cout << *p << endl ;    // 列印 5
```

❖ 指標的參考的 & 符號須寫在變數之前

指標的參考 (二)

■ 雙層指標的參考：參考的型別為雙層指標

```
int ** p ;
```

```
int ** &q = p ;           // 雙層指標 q 為 p 指標的參考
```

```
q = new int*[5] ;         // 雙層指標 q 指向一二維動態空間
```

```
for ( int i = 0 ; i < 5 ; ++i )
```

```
    q[i] = new int[3] ;
```

❖ 以上的雙層指標 **q** 與 **p** 實為同一個雙層指標，所以當 **q** 指向二維動態空間，也代表 **p** 也指向同一動態空間

參數傳遞（一）：傳值

- 傳值方式：參數是以複製的方式，將外部數值資料傳入函式內部使用

(a) 呼叫程式碼

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

(b) 函式碼：回傳兩參數的最大值

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

以上函式的參數列執行了

```
int a = i ;
int b = j ;
```

// 將整數 i 的值複製給參數 a
// 將整數 j 的值複製給參數 b

函式 max	int a
	int b
函式內部 變數	

= i (參數 a 為整數，初值由複製 i 而來)
= j (參數 b 為整數，初值由複製 j 而來)

passed by value

參數傳遞（二）：傳址

- 指標傳遞方式：參數透過指標複製外部變數的位址到函式內部使用

(a) 呼叫程式碼

```
int i = 3 , j = 5 ;
swap( &i , &j ) ;
// 列印 5 , 3
cout << i << ' ' << j ;
```

以上函式的參數列執行了

```
int *a = &i ;
int *b = &j ;
```

函式	int *a
swap	int *b
函式內部 變數	

(b) 函式碼：對調兩整數參數的數值

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

// 指標參數 a 指向整數 i 的位址

// 指標參數 b 指向整數 j 的位址

= &i (參數 a 為整數指標，指向 i 的位址)

= &j (參數 b 為整數指標，指向 j 的位址)

passed by address

參數傳遞（三）：傳址

- 不管傳入資料的複雜度，指標傳遞只是複製變數的位址
- 指標傳遞可以在函式內以間接方式更動指向參數的數值
- 如果要避免函式使用指標參數更動傳入的參數值，則可在指標變數前加上 **const**

```
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
    ...
}
```

以上參數列執行：

```
const char *str = p ;    // 字元指標參數 str 指向指標 p 所指向的位址
```

參數傳遞（四）：參考型別傳遞

■ 參考傳遞方式：使用參考型別方式傳遞參數

(a) 呼叫程式碼

```
int i = 3 , j = 5 ;
swap( i , j );
// 列印 5 , 3
cout << i << ' ' << j ;
```

(b) 函式碼：對調兩參數值

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

以上函式的參數列執行了

```
int &a = i ;
int &b = j ;
```

// 參數 a 為變數 i 在函式內的參考
// 參數 b 為變數 j 在函式內的參考

函式	int &a
swap	int &b
函式內部 變數	

= i (參數 a 為 i 在函式內的參考)
= j (參數 b 為 j 在函式內的參考)

passed by reference

參數傳遞（五）：參考型別傳遞

- 參考常數傳遞：若要避免參考參數意外地更動不想被改變的值則可以使用參考常數

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 ;    }  

```

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 << k << endl ;    }  

```

參數傳遞方式的比較

■ 函式參數傳遞的比較：

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

❖ 這裡的 優* 為使用常數型的指標或者是參考

傳遞陣列 (一)

- 透過指標傳遞方式複製陣列首位元素位址，使得程式可在函式內利用指標間接地存取陣列元素

(a) 呼叫程式碼

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

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

(b) 函式碼：計算陣列元素總合

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

以上函式的參數列執行了

```
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
```

❖ 以上 **p** 指標只記住陣列的首位元素位址，並無法得知陣列長度

傳遞陣列（二）

■ 三種傳遞陣列方式：

`sum` 函式也可以使用以下作用相同的方式設定

```
int sum( int *p      , int n ) ;
```

```
int sum( int p[]     , int n ) ;
```

```
int sum( int p[10]   , int n ) ;    // 避免使用
```

- ❖ 後兩種的 `p` 仍是指標，並非為存有元素的陣列。
同時最後一式的寫法易造成誤解，應避免使用

傳遞陣列 (三)

- 若要傳遞二維以上的陣列到函式內，在寫法上可將多維陣列的第一維長度加以省略

```
// 計算一整數陣列內所有元素的和
int sum( const int data[][3] , int r , int c ){
    int i , j , s = 0 ;
    for( i = 0 ; i < r ; ++i )
        for( j = 0 ; j < c ; ++j )
            s += data[i][j] ;
    return s ;
}

...
int foo[2][3] = { {1,2,3} , {4,5,6} } ;
cout << sum( foo , 2 , 3 ) << endl ;    // 印出 21
```

傳遞陣列（四）

- 以上的函式參數 `data[][3]` 與以下形式相當

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

但不是

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

傳遞陣列（五）

■ 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 set_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] ;
set_array( data , 2 , 3 , 4 , 1 ) ;    // 讓陣列元素皆設為 1
```

同樣的，陣列參數也可以寫成指標的形式

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

傳遞指標參考（一）

■ 函式的參數為指標的參考

// 設定 `p` 指向 `s` 個整數動態空間，且元素初值設為 `val`

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

```
int *ptr ;
```

```
setup( ptr , 5 , 1 ) ; // ptr 指向 5 個動態空間，元素值皆 1
```

- ❖ 若 `ptr` 指標不以指標參考方式傳入函式，而是使用傳址方式，例如 `int* p`，則 `p` 與 `ptr` 即為兩個獨立的指標，同時函式所產生的動態空間在函式執行完畢後，隨即遺失，`ptr` 並不會儲存其位址

傳遞指標參考 (二)

■ 雙層指標的參考

```
// 設定 p 指向 rxc 二維動態空間，且元素初值設為 val
void setup_2d( int ** & p , int r , int c , int val ){
    int i , j ;
    p = new int*[r] ;
    for ( i = 0 ; i < r ; ++i ) {
        p[i] = new int[c] ;
        for ( j = 0 ; j < c ; ++j ) p[i][j] = val ;
    }
}
...
int ** ptr ;

// ptr 指向新產生的 5x3 二維動態空間，元素初值皆為 1
setup_2d( ptr , 5 , 3 , 1 ) ;
```

參數預設值（一）

■ 參數可以使用預設值

// 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 ;  
}
```

在使用上

```
// 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 ;
```


參數預設值（二）

- 所有參數預設值都須由參數列的末尾依次往前置放

```
int max( int a = 0 , int b , int c ) ;    // 錯誤
```

```
int max( int a , int b , int c = 0 ) ;    // 正確
```

- 使用參數預設值的函式與其它函式的特徵資料亦不可相同
- 如果利用先宣告後定義的方式設計函式，則參數預設值僅可寫在函式宣告階段

```
// 函式宣告：可以加入參數的預設值
```

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

```
// 函式定義：不可加入參數的預設值
```

```
int square_sum( int a , int b , int c ) {  
    return a*a + b*b + c*c ;  
}
```

靜態變數

- 當函式內的變數被定義為靜態變數時，則此變數將在函式執行結束後繼續存在

```
int counter() {  
    // i 為靜態變數，初值為 0，此定義式只被執行一次  
    static int i = 0 ;  
    ++i ;  
    return i ;  
}
```

- 若 **static** 變數沒有設定初值，則會被設定為 0

static variable

主函式

■ C++的主函式被定義為

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

參數

`int argc` : 程式在執行時所輸入的字串個數
`char* argv[]` : 字串陣列

■ 若某一可執行檔的檔名為 `animal` , 當使用者在命令列中輸入

```
animal cat dog fish
```

則

```
int argc = 4  
char* argv[] = {"animal", "cat", "dog", "fish"}
```

main function

回傳型別

■ 回傳單一資料與無回傳資料

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

```
void out_max( int a , int b ) {
    cout << ( a > b ? a : b ) ;
}
```

■ 回傳包裝資料

// 定義複變數結構型別

```
struct Complex {
    float re , im ;
};
```

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

// 回傳 foo

return type

指標型別回傳 (一)

- 函式也可以回傳指標，用以回傳某筆資料的位址

(a) 呼叫程式碼

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

// 印出 15
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 ) ;
}
```

以上函式的參數列執行了

```
int *p = &a ;
int *q = &b ;
```

```
// 指標參數 p 指向傳入變數 a
// 指標參數 q 指向傳入變數 b
```

指標型別回傳 (二)

- 回傳指標的函式可以直接置放在指定運算子的左側

(a) 呼叫程式碼

```
int a = 3 , b = 5 ;
*max(&a, &b) += 10 ;
// 印出 15
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 ) ;
}
```

- 以上的 `max(&a, &b)` 回傳指標參數 `q`

```
*max(&a, &b) += 10
<=>    *q += 10           // 因 return q
<=>    b += 10           // 因 int *q = &b
```

指標型別回傳 (三)

- 若以參考方式將資料傳入函式，則程式碼可改成

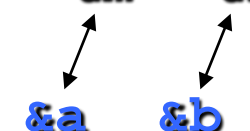
(a) 呼叫程式碼

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

// 印出 15
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 ) ;
}
```



以上函式的參數列執行了

```
int &m = a ;
int &n = b ;
```

```
// 參數 m 為變數 a 在函式內的參考
// 參數 n 為變數 b 在函式內的參考
```

指標型別回傳 (四)

- 指標不可指向函式內部定義的變數，因其在函式執行完畢後會全部消失

(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

```
char * char_str( char a , int n ){  
    char s[100] = { '\0' } ;  
    for( int i = 0 ; i < n ; ++i )  
        s[i] = a ;  
    return s ;  
}
```

- ❖ 右側的 **s** 字串在函式執行完畢後隨即消失，指標 p 指向已消失的字串易引發執行錯誤

指標型別回傳 (五)

- 若函式必須回傳指標，則應避免回傳暫時變數的位址，此時可用動態空間處理

(a) 呼叫程式碼

```
char *p ;
p = char_str('x',10);
// 印出 10 個 'x'
cout << p << endl ;
...
delete [] p ;
```

(b) 函式碼：設定 s 字串

```
char * 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 ;
}
```

- ❖ 右側的 s 字串為使用 new 向系統取來的動態空間，此空間會一直存在直到使用 delete 將此空間歸還，因此在函式執行完畢後動態空間仍舊存在

參考型別回傳 (一)

■ 函式可回傳另一變數的參考

(a) 不使用函式

```
int a = 3 , c = 5 ;  
int &b = a ;  
  
b = c ;  
cout << a << endl ;
```

(b) 使用函式

```
int & fn( int &d ){ return d ; }  
...  
int a = 3 , c = 5 ;  
fn(a) = c ;  
cout << a << endl ; // 印出 5
```

- ❖ 右側的 `fn(a)` 回傳 `d` 的參考，`d` 參數又是輸入整數 `a` 的參考，因此 `fn(a) = c` 即是取出整數 `c` 所在空間的數值，存入 `a` 整數所在的記憶空間

參考型別回傳 (二)

- 函式若沒有回傳參考或位址則不能置放在指定運算子的左側

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

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

以上 **fn2** 函式是以複製方式將 **d** 回傳出函式。複製的資料在函式執行後隨即消失，不能用來儲存資料

- 函式不管是回傳參考或是複製資料都可以放在指定運算子的右側使用

```
int a = 3 , b ;  
b = 3 * fn(a) + fn2(a) ;
```

參考型別回傳 (三)

■ 使用回傳參考的函式

(a) 呼叫程式碼

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

以上執行後 **a = 3 , b = 7**

參考型別回傳（四）

- 函式所回傳的參考不能參照在函式內的局部變數

```
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
```

- ❖ 以上當 **no** 小於 **q** 時將會引發執行錯誤，原因在於函式回傳內部暫時變數的參考，而暫時變數在函式執行結束後隨即消失，無法用來儲存資料

函式指標（一）

■ 函式指標：指向函式的指標

```
// fn1 函式指標可以指向所有須要  
// 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須要  
// 兩個浮點數參數及回傳浮點數的函式  
double (*fn2)( double , double ) ;
```

function pointer

函式指標 (二)

■ 若定義以下幾個函式

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

```
int (*f)(int) = abs ;           // f 指到 abs 函式
cout << f(-5) << endl ;        // f(-5) → abs(-5)
f = square ;                   // f 指到 square 函式
cout << f(-2) << endl ;        // f(-2) → square(-2)
```

```
int (*g)(int,int) = min ;      // g 指到 min 函式
cout << g(3,5) << endl ;       // g(3,5) → min(3,5)
g = max ;                     // g 指到 max 函式
cout << g(3,5) << endl ;       // g(3,5) → max(3,5)
g = abs ;                     // 錯誤
```

函式指標 (三)

■ 函式指標可用來簡化程式碼

```
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 ;  
}
```

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

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

```
// 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 ;
```


行內函式（一）

- 執行函式可以增加程式的可讀性，但可能會降低程式執行效率

```
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 ;  
}
```

- **max1** 是在程式碼內直接比較 **i** , **j** 大小
- **max2** 透過函式 **max** 間接處理，在執行上較花費時間

inline function

行內函式（二）

- 為了增加函式效率，可以定義 **max** 函式為行內，可以同時保有使用函式的可讀性與不使用函式的經濟性

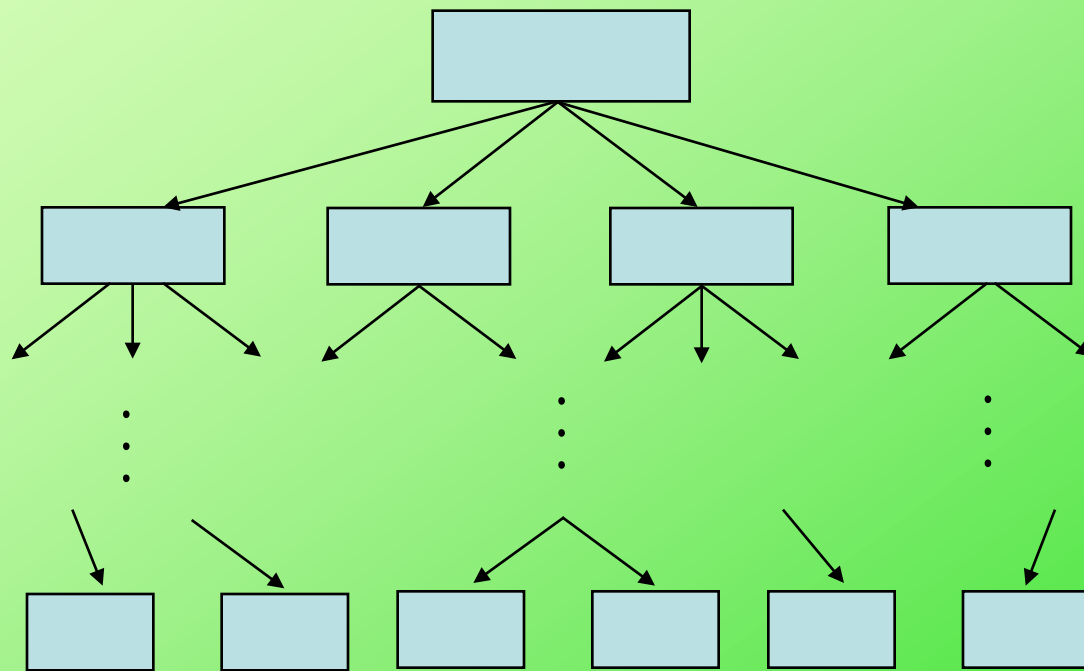
```
inline int max( int a , int b ){  
    return a > b ? a : b ;  
}
```

編譯器會分析**行內函式**的程式碼，在將其轉成一般的程式敘述後，放置在函式呼叫的地方，對使用者而言感覺仍是在執行函式，但對C++而言，並非執行函式

- ❖ 對編譯器而言，行內函式僅是一種建議動作，若函式太複雜，編譯器仍會將行內函式看成一般函式編譯

個別擊破法

- 個別擊破法：將複雜的問題切割成若干個簡單的小問題，然後個別解決每個小問題



divide and conquer

字串數字相加

- 四個位元組整數的儲存上限： $2^{32}-1$

若要處理更大的整數，通常須以字元陣列來儲存數字

- 兩字串數字相加，可以先將短字串數字之前補上 0

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

- 字元數字相加：由右往左將兩兩字元數字相加

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

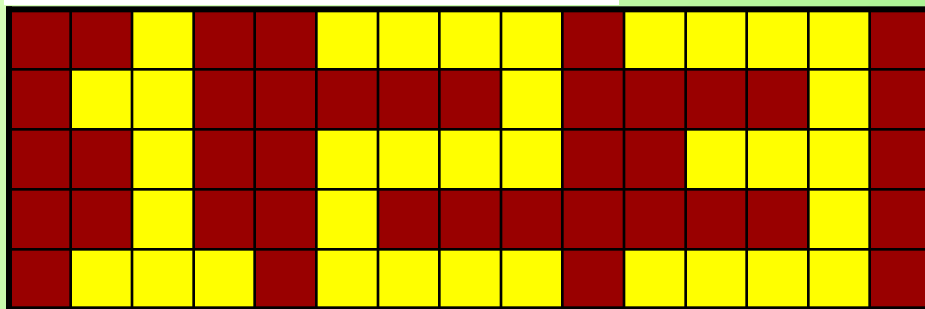
程式

輸出

點矩陣數字

- 跑馬燈可使用若干個等長的字串陣列模擬

```
string bitmap[5] ;
```



=> bitmap[0]

=> bitmap[1]

=> bitmap[2]

=> bitmap[3]

=> bitmap[4]

- 若將每個 5x5 點矩陣字元存入此字串陣列，就可以模擬跑馬燈的顯示

- $1 \ll c$: 將 1 往左移動 c 個位元位置
 $n \& (1 \ll c)$: 檢查整數 n 由右邊數來第 $c+1$ 個位元是否有值

程式

輸出

列印函式

- 函式可以將另一函式當成參數輸入函式內處理，如此可增加函式使用的自由度
- `plot_function(fn,a,b)` 可以將函式 `fn` 在 `[a,b]` 之間的數值印出。當 `fn` 改變了，則列印的函式也會隨之改變，這裡的 `fn` 參數是以函式指標方式傳入 `plot_function` 內



程式

輸出

遞迴函式 (一)

- 遞迴函式：函式執行時呼叫自己本身

階乘函數

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

階乘函式

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

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

recursive function

遞迴函式 (二)

■ 基本樣式

```
T recursive_function(...) {  
    // 判斷遞迴函式是否已到達遞迴終止條件  
    if (...) {  
        A                // A：終止遞迴  
    } else {  
        B                // B：繼續遞迴  
    }  
}
```

❖ B 包含繼續執行 recursive_function 函式

遞迴函式 (三)

■ 巴斯卡三角形

$$\begin{bmatrix} p \end{bmatrix} = \begin{bmatrix} 1 \\ 1 & 1 \\ 1 & 2 & 1 \\ 1 & 3 & 3 & 1 \end{bmatrix}$$

數學
函數

$$P_{i,j} = \begin{cases} 1 & i = j \text{ 或 } j = 0 \\ P_{i-1,j-1} + P_{i-1,j} & i > j \end{cases}$$

遞迴
函式

```
int pascal( int i , int j ){
    if( i == j || j == 0 )
        return 1 ;
    else
        return pascal(i-1,j-1) + pascal(i-1,j) ;
}
```

遞迴函式 (四)

■ 遞迴迭代過程

```
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 + pascal(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
```

快速遞迴程式 (一)

■ Fibonacci 數列：1 1 2 3 5 8 13 21 ...

數學函數

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

傳統遞迴

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

❖ 當 n 變大時(>40)，計算速度變得相當緩慢

Fibonacci sequence

快速遞迴程式 (二)

■ 快速遞迴

```
unsigned int fib( unsigned int n ){  
  
    static unsigned int f[100] = {0} ;  
  
    if( n < 2 )  
        return 1 ;  
    else if( f[n] != 0 )  
        return f[n] ;  
    else  
        return f[n] = fib(n-2) + fib(n-1) ;  
  
}
```

❖ 利用靜態陣列儲存已計算過的數值藉以避免重複計算

快速遞迴程式 (三)

■ 迴圈

```
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] ;  
    }  
}
```

❖ 若程式可使用迴圈處理則應避免使用遞迴藉以提高執行效能

遞迴的使用時機 (一)

■ 1 到 10 的整數和

```
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 ;
```

可改成

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

遞迴的使用時機 (二)

■ 列出由五個數字中任選三個的所有組合

```

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] << "]" " ;

        }
    }
}

```

共有 10 種

[123]	[124]
[125]	[134]
[135]	[145]
[234]	[235]
[245]	[345]

遞迴的使用時機 (三)

■ 列出五個數字中任選三個的所有組合(遞迴)

```
#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( 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 ;
}
```


遞迴的使用時機（四）

- 某問題可以分解成若干個小問題，而個別小問題的解決方式又與原來的問題相似

（1）尋找遞迴結構：

在原始問題的解決步驟中尋找同形式的小問題，構成基本遞迴架構

（2）確認終結條件：

為避免程式陷入無窮的遞迴，因此要確認終結條件是可以到達的

根號 2 與連分數 (一)

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

$$\sqrt{2} = 1 + x = 1 + \frac{1}{2 + x} = 1 + \frac{1}{2 + \frac{1}{2 + x}}$$

$$= 1 + \frac{1}{2 + \frac{1}{2 + \frac{1}{2 + x}}} = 1 + \frac{1}{2 + \frac{1}{2 + \frac{1}{2 + \dots}}}$$

$$= [1, 2, 2, 2, \dots] = [1, \overline{2}]$$

continued fraction

根號 2 與連分數 (二)

■ 連分數定義

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

■ 基本性質

- 無理數可用無窮項的連分數表示
- 有理數的連分數項數為有限個
- 若 c_i 為連分數前 $i+1$ 項所代表的數字，則當 i 越趨近 n ， c_i 越逼近連分數所代表的數

根號 2 與連分數 (三)

■ 連分數 a

$$a = [a_0, a_1, \dots, a_n]$$

■ 漸近分數 c_i

$$c_0 = [a_0]$$

$$c_1 = [a_0, a_1]$$

$$c_2 = [a_0, a_1, a_2]$$

...

$$c_n = [a_0, a_1, \dots, a_n] = a$$

❖ 漸近分數是以上下振盪方式趨近真正的數字

根號 2 與連分數 (四)

■ 根號 2 與連分數

$$\sqrt{2} = [1, 2, 2, 2, \dots] = [1, \overline{2}]$$

■ 根號 2 與漸近分數

$\sqrt{2}$	>	[1]	=	1.0
	<	[1, 2]	=	1.5
	>	[1, 2, 2]	=	1.4
	<	[1, 2, 2, 2]	=	1.416667
	>	[1, 2, 2, 2, 2]	=	1.413793
	<	[1, 2, 2, 2, 2, 2]	=	1.414286

根號 2 與連分數 (五)

■ 連分數 a

$$a = [a_0, a_1, \dots, a_n]$$

■ 第 n 項漸近分數 c_n 遞迴公式

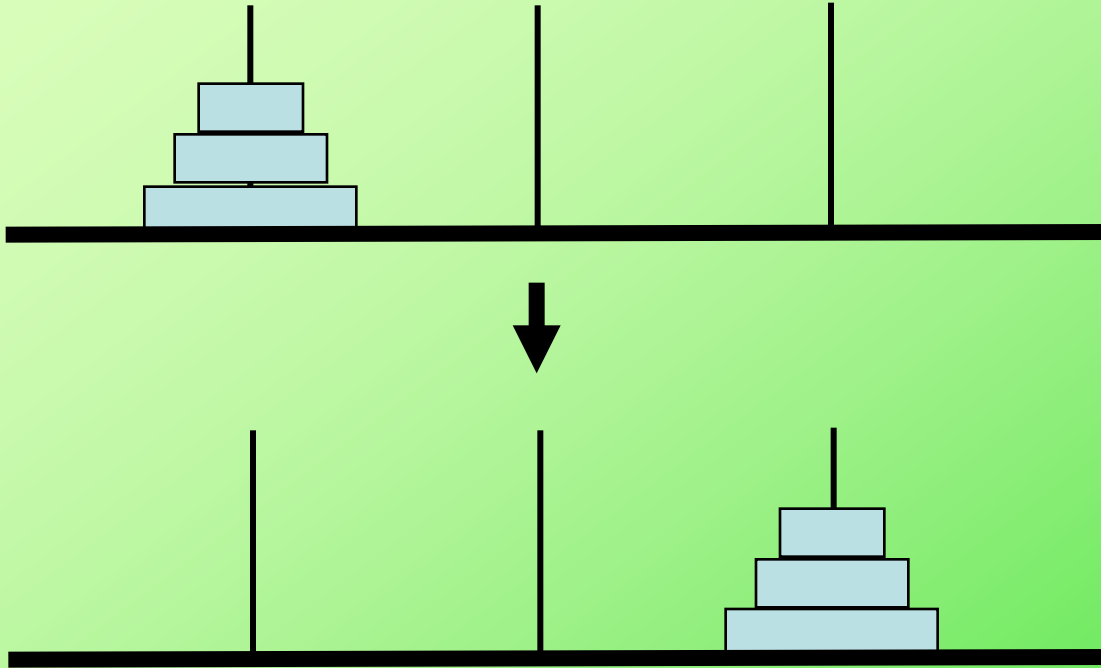
$$r_i = \begin{cases} a_n & , i = 0 \\ a_{n-i} + \frac{1}{r_{i-1}} & , i > 0 \end{cases}$$

以上的 r_n 即為漸近分數 c_n

程式

輸出

何內塔圓盤（一）



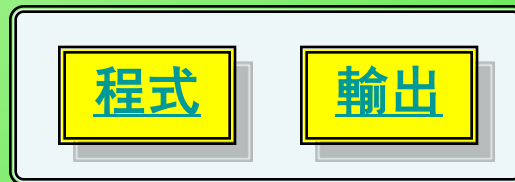
■ 搬動規則

1. 每次移動一個圓盤
2. 大圓盤在小圓盤之下

何內塔圓盤（二）

■ 搬動步驟

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



遞迴包牌程式

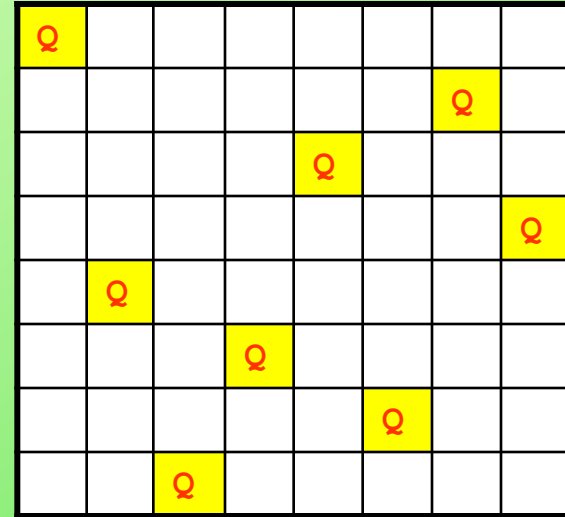
- 找出由 m 個數字中取出 n 個數字的所有組合，總數為

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

[程式](#)[輸出](#)

八個皇后（一）

- 將八個皇后放置於西洋棋上，使之不會互相影響



- 策略：由第一行 (column) 起由左而右，由上而下尋找適當的列 (row) 放置皇后
 1. 在本行中由上而下選擇一個適當位置放至皇后，如果不能則回上一行
 2. 進入下一行放置皇后

八個皇后 (二)

■ 輸出結果

[1]

```
Q + + + + + + +
+ + + + + + Q +
+ + + + Q + + +
+ + + + + + + Q
+ Q + + + + + +
+ + + Q + + + +
+ + + + + Q + +
+ + Q + + + + +
```

[2]

```
Q + + + + + + +
+ + + + + + Q +
+ + + Q + + + +
+ + + + + Q + +
+ + + + + + + Q
+ Q + + + + + +
+ + + + Q + + +
+ + Q + + + + +
```

...

■ 遞迴程式



■ 迴圈程式

