# 06-函數

# 6-1 函數

隨著寫程式經驗愈來愈多，你會發現有些程式碼會不斷重複出現，就像例行性工作一樣，例如：求平方根、將資料排序、驗證帳號密碼.....等等。一次又一次的輸入這些程式碼會讓人很不耐煩。對於這些經常出現的程式碼片段，我們可以使用函數來把它們包裝起來。C/C++裡面的函數就像數學裡面的函數，例如：

$f(x)=2x^2+3x+4$

它有一個輸入：x，有一個輸出：f(x)。你給它一個輸入 3，它在運算後會給你一個輸出31；你給它另一個輸入 2，它會給你另一個相應的輸出18。不管你給的輸入是什麼，它都會很忠實的去完成該做的事 $2x^2 + 3x + 4$ ，並把結果輸出給你。

## 定義函數

以上面那個 f(x) 為例，我們可以這樣在 C++ 裡定義它。

```
int f(int x)
{
  int result = 2*x*x +3*x + 4;
  return result;
}
```

其架構如下：

```
回傳值型別 函數名稱(參數1型別 參數1名稱, 參數2型別 參數2名稱, ...)
{
    // 實作程式碼
    return 回傳值;
}
```

接下來我們就可以使用這個函數了。

```
#include <iostream>

using namespace std;

int f(int x)
{
  int result = 2*x*x +3*x + 4;
  return result;
}

int main()
{
    int n;
    int ans;

    n = 2;
    ans = f(n);
    cout << ans << endl;   // 18

    ans = f(3);
    cout << ans << endl;   // 31

    n = 5;
    cout << f(n) << endl;  // 69

    return 0;
}
```

請注意我們把 函數f 定義在 main() 的前面。如同變數在使用前要先宣告，函數也是一樣。

我們在第 17 行首次使用到 函數f，所以在這之前必須先知道 函數f 長什麼樣子。

如果把 函數f 定義在後面，在編譯時就會發生錯誤。

```
#include <iostream>

using namespace std;

int main()
{
    int n;
    int ans;

    n = 2;
    ans = f(n);  // f( ) 是什麼？往前看不到有人告訴我 f( ) 是什麼。
    cout << ans << endl;   // 18

    ans = f(3);
    cout << ans << endl;   // 31

    n = 5;
    cout << f(n) << endl;  // 69

    return 0;
}

// 定義在後面
int f(int x)
{
  int result = 2*x*x +3*x + 4;
  return result;
}
```

```
main.cpp: In function 'int main()':
main.cpp:11:11: error: 'f' was not declared in this scope
    ans = f(n);
          ^
```

# 宣告函數

有沒有注意到，前面我們一直說定義函數，而不是宣告函數(declare)。

我們以同一個 函數f 為例，宣告這個函數的作法為：

```
int f(int x);
```

或

```
int f(x);
```

宣告函數只要講清楚這幾個重點即可：

1. 函數名稱
2. 參數列 (每個參數的型別，可以沒有名字)
3. 回傳值型別

我們把上面的範例程式改成只有宣告試試。

```
#include <iostream>

using namespace std;

int f(int x);  // 宣告在這裡

int main()
{
```

```
    int n;
    int ans;

    n = 2;
    ans = f(n);    // 使用到 函數f
    cout << ans << endl;   // 18

    ans = f(3);    // 使用到 函數f
    cout << ans << endl;   // 31

    n = 5;
    cout << f(n) << endl;  // 69,  使用到 函數f

    return 0;
}
```

建置(build)這個程式時，出現了沒看過的錯誤。

```
main.cpp:13: undefined reference to `f(int)'
main.cpp:16: undefined reference to `f(int)'
main.cpp:20: undefined referenc
```

這個 `undefined reference to 'f(int)'` 是什麼意思呢？

我們的程式碼要經過「編譯(compile)」、「連結(link)」兩個步驟，才能生成最終的可執行檔。

在編譯階段，編譯器看到叫用(call)函數時，只會確認之前宣告過的函數

1. 名稱是否相符
2. 參數列的數量和型別是否相符
3. 回傳值型別是否相符

如果都符合，會在叫用函數的地方留個「空位」，然後編譯將會成功完成，進入連結階段。

在連結階段必須真的有一個函數被定義過，才能把這個函數「身體」所在的位置填入之前編譯階段留下的「空格」。

我們修改程式，在末端補上 函數f 的定義，即可順利建置。

```
#include <iostream>

using namespace std;

int f(int x);  // 宣告在這裡

int main()
{
    int n;
    int ans;

    n = 2;
    ans = f(n);    // 使用到 函數f
    cout << ans << endl;   // 18

    ans = f(3);    // 使用到 函數f
    cout << ans << endl;   // 31

    n = 5;
    cout << f(n) << endl;  // 69,  使用到 函數f

    return 0;
}

// 定義在後面
int f(int x)
{
  int result = 2*x*x +3*x + 4;
  return result;
```

```
}
```

或許有同學會覺得把它拆成兩段一個放前面、一個放後面，不是多此一舉嗎？

這個設計的考量是，我們在開發大專案時，不會把所有程式碼寫在同一個檔案裡，而是會分散在多個檔案裡。

如果有 10,000 行程式碼放在同一檔案裡，只要有一行修改，這 10,000 行都要重新編譯、連結、產出執行檔。

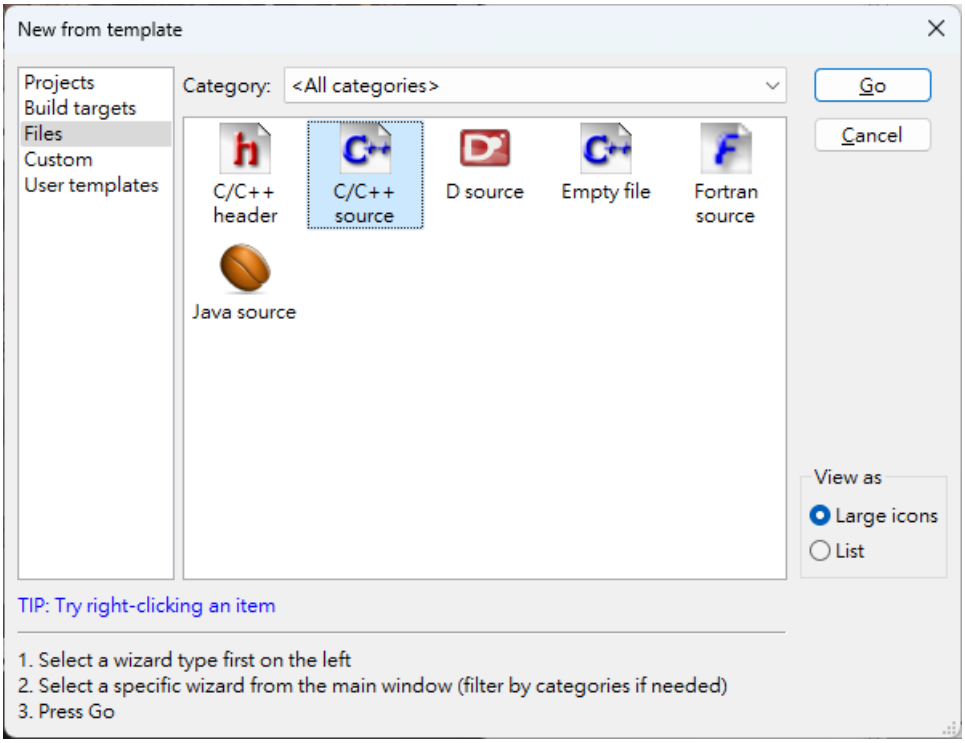
但若是把它拆成 10 個 1,000 行的檔案，當其中一行修改時，只有包含那行檔案 的 1,000 行需要重新編譯，然後把 10 個編譯後的檔案連結產出執行檔即可。

# 多檔案專案

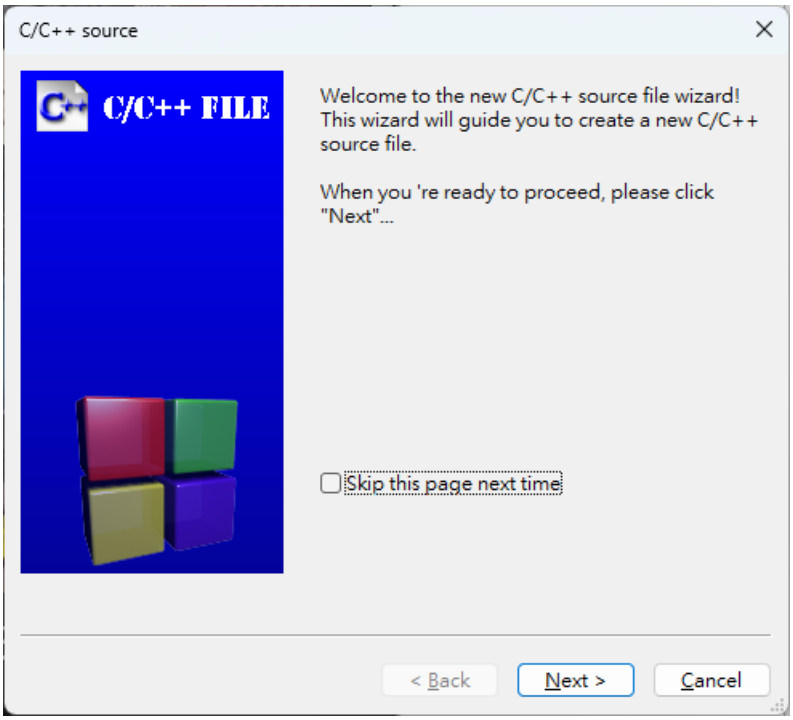
我們來實作一下把範例程式拆成兩個 .cpp 檔案。

目前我們有一個 main.cpp，接下來新增一個 myfuntion.cpp。

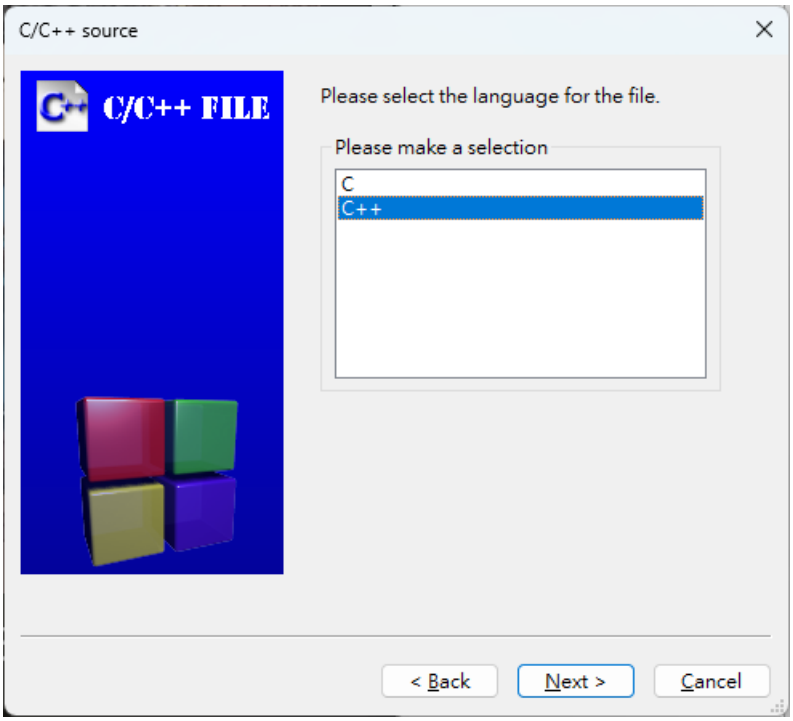
1. 首先依序點選 Code::Blocks 選單 [File]->[New]->[file...]
2. 選擇 [C/C++ source]->[Go]



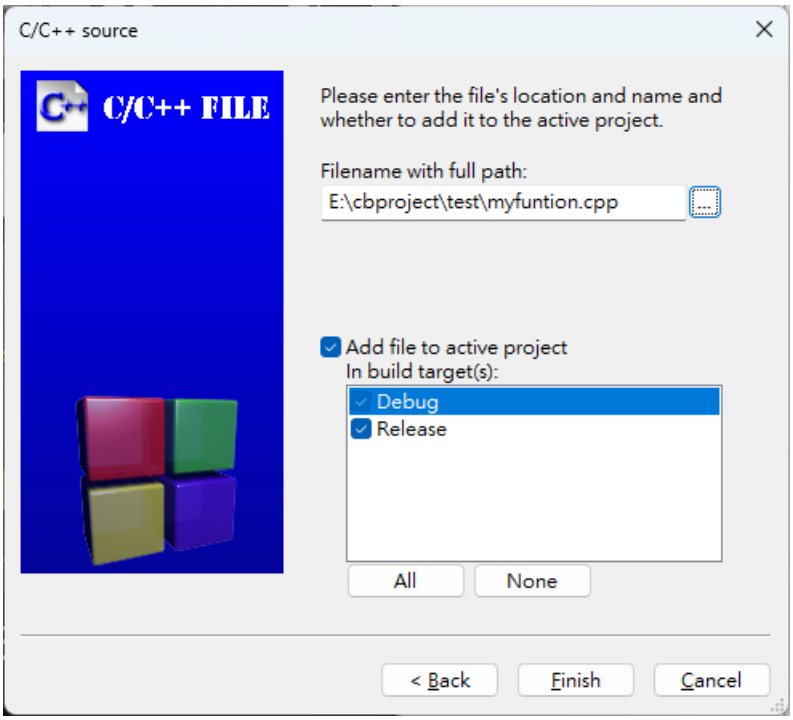

[Next]



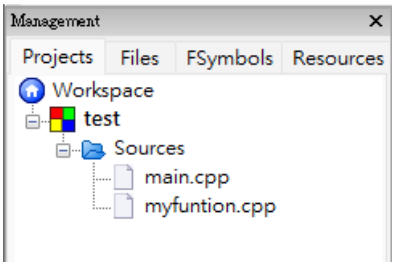

[Next]




點選 [...] 檔名輸入 "myfunction.cpp"，接著點選 [All]->[Finish]




3. 現在專案裡就可以多一個 function.cpp 檔了。




在 [function.cpp] 裡定義好函數f。

```
int f(int x)
{
  int result = 2*x*x +3*x + 4;
  return result;
}
```

在 [main.cpp] 裡宣告函數f 並使用它。

```
#include <iostream>

using namespace std;

int f(int x);  // 宣告在這裡

int main()
{
    int n;
    int ans;

    n = 2;
    ans = f(n);
    cout << ans << endl;   // 18

    ans = f(3);
    cout << ans << endl;   // 31

    n = 5;
    cout << f(n) << endl;  // 69
```

```
    return 0;
}
```

試著建置並執行，應該可以順利完成。

### [練習] 增加一個 $g(x) = x(x-1)$

在 [myfunction.cpp] 裡定義 g(x) 函數

```
int f(int x)
{
  int result = 2*x*x +3*x + 4;
  return result;
}

int g(int x)
{
  return x*(x-1);
}
```

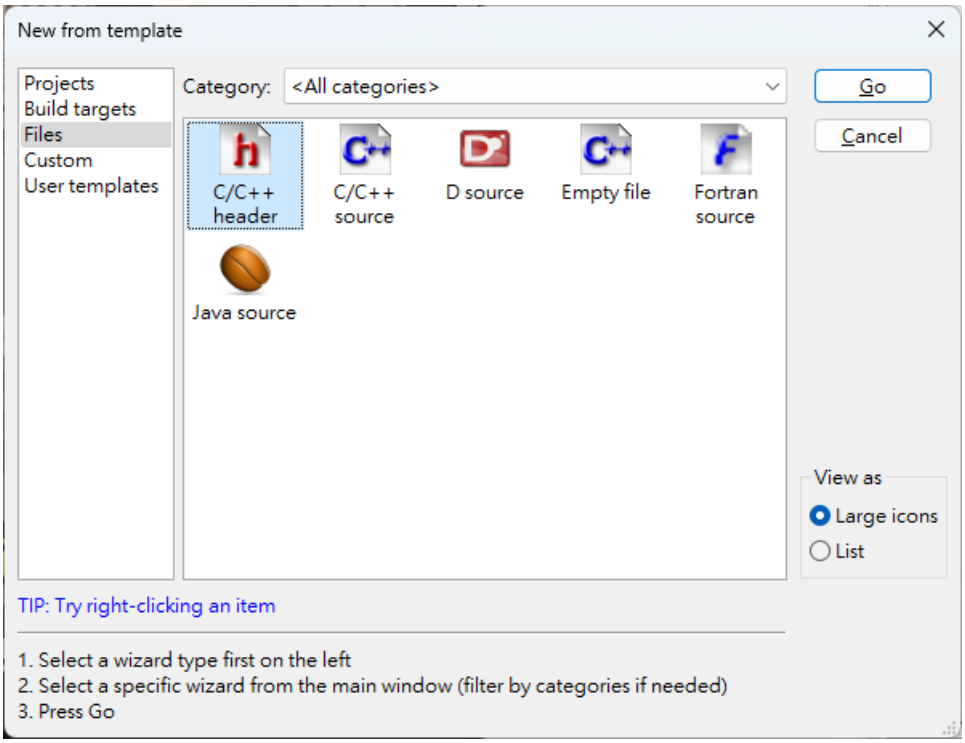
在 [main.cpp] 裡宣告並使用 g(x) 函數

```
#include <iostream>

using namespace std;

int f(int x);
int g(int x); // 宣告 g(x)

int main()
{
    int n;
    int ans;

    n = 2;
    ans = f(n);
    cout << ans << endl;

    cout << g(3) << endl;  // 6, 使用 g(x)

    return 0;
}
```

## 標頭檔(header file)

隨著自己定義的函數愈來愈多，[main.cpp] 前面的宣告會愈來愈多行。我們可以把這些宣告移到另一個檔案裡。

類似之前我們新增 [C/C++ source]檔 的方式，這次我們新增一個 **[C/C++ header]** 檔，並命名為 "myfunction.h"。




把 [main.cpp] 裡的宣告移到 [myfunction.h] 裡。

```
int f(int x);
int g(int x);
```

在 [main.cpp] 裡引入(include)標頭檔 [myfunction.h] 。在編譯時，編譯器會把 myfunction.h 檔案的內容抄到這個引入的地方。

```
#include <iostream>
#include "myfunction.h"  // 引入標頭檔 myfunction.h

using namespace std;

int main()
{
    int n;
    int ans;

    n = 2;
    ans = f(n);
    cout << ans << endl;

    cout << g(3) << endl;

    return 0;
}
```

建置並執行後，程式應該可以順利運行。

我們從一開始學 C++ 就在程式的開頭有一行 `#include <iostream>`。現在你應該可以了解它的作用了，它裡面放的就是和輸入、輸出相關的宣告。

至於為什麼它用角括號 `< >`，我們自己寫的用雙引號 `" "` 呢？

這跟標頭檔所在的位置有關，用角括號 `< >` 編譯器會去內建函式庫的資料夾找標頭檔，用雙引號 `" "` 編譯器會去目前這個專案的資料夾去找標頭檔。

# 6-2 在函數中使用函數

## 相同名稱的函數

原則上函數的名稱不能重覆，但是只要其參數列不同，就可以使用相同的名稱。

以下面的程式為例，我們可以觀察到叫用函數時，編譯器會檢查函數名稱和參數列數量和型別。

```
#include <iostream>

using namespace std;

// 回傳 2 整數中的最小值
int MIN(int a, int b)
{
    cout << "回傳 2 整數中的最小值" << endl;
    if(a<=b)
        return a;
    else
        return b;
}

// 回傳 2 浮點數中的最小值
double MIN(double a, double b)
{
    cout << "回傳 2 浮點數中的最小值" << endl;
    if(a<=b)
        return a;
    else
        return b;
}

// 回傳 3 整數中的最小值
int MIN(int a, int b, int c)
{
    cout << "回傳 3 整數中的最小值" << endl;
    if(a<=b && a<=c)
        return a;
    else if(b<=a && b<=c)
        return b;
    else
        return c;
}

int main()
{
    int x=2, y=5, z=3;
    double i=5.3, j=2.1, k=4.3;

    cout << MIN(x, z) << endl;       // 回傳 2 整數中的最小值
    cout << MIN(i, j) << endl;       // 回傳 2 浮點數中的最小值
    cout << MIN(x, y, z) << endl;    // 回傳 3 整數中的最小值

    return 0;
}
```

```
回傳 2 整數中的最小值
2
回傳 2 浮點數中的最小值
2.1
回傳 3 整數中的最小值
2
```

# 在函數中叫用函數

在前例中我們為了求 3 整數中的最小數，又另外寫了一個 3 參數的 MIN 函數，其內容也是整個重寫。

我們的另一種選擇是利用已寫好的 2 參數 MIN 函數，來實作出 3 參數的 MIN 函數。

```
#include <iostream>

using namespace std;

// 回傳 2 整數中的最小值
int MIN(int a, int b)
{
    cout << "回傳 2 整數中的最小值" << endl;
    if(a<=b)
        return a;
    else
        return b;
}

// 回傳 3 整數中的最小值
int MIN(int a, int b, int c)
{
    cout << "回傳 3 整數中的最小值" << endl;
    return MIN(MIN(a, b), c);  // 利用 MIN(int , int)
}


int main()
{
    int x=2, y=5, z=3;

    cout << "Step 1:" << endl;
    cout << MIN(x, z) << endl;          // 回傳 2 整數中的最小值

    cout << "Step 2:" << endl;
    cout << MIN(x, y, z) << endl;       // 回傳 3 整數中的最小值

    cout << "Step 3:" << endl;
    cout << MIN(x, MIN(y, z)) << endl; // 回傳 3 整數中的最小值

    return 0;
}
```

由輸出結果我們可以看到，叫用 MIN(int , int, int) 時，MIN(int, int) 被叫用了 2 次。

```
Step 1:
回傳 2 整數中的最小值
2
Step 2:
回傳 3 整數中的最小值
回傳 2 整數中的最小值
回傳 2 整數中的最小值
2
Step 3:
回傳 2 整數中的最小值
回傳 2 整數中的最小值
2
```

## 練習：求 a, b 兩正整數的最大公因數(GCD)

設計一個 GCD 函數，求 2 正整數的最大公因數。

---

**1. 用迴圈慢慢找**

```
int GCD(int a, int b)
{
```

```
  if(a>b)
    swap(a, b);
  int ans = 1;
  for(int i=1; i<=a; i++) {
    if(a%i==0 && b%i==0) {
      ans = i;
    }
  }
  return ans;
}
```

**2. 超級快的「輾轉相除法」**

```
int GCD(int a, int b)
{
  int r;
  while(a%b>0) {
    r = a%b;
    a = b;
    b = r;
  }
  return b;
}
```

## 練習：求 a, b 兩正整數的最小公倍數(LCM)

設計一個 LCM 函數，求 2 正整數的最小公倍數。

---

**利用之前的 GCD 函數**

```
int LCM(int a, int b)
{
  return a/GCD(a, b)*b;
}
```

我們不使用 `a*b/GCD(a,b)` 的原因是，若先把 a*b ，其相乘後數值溢位的可能性更大，先把 a 除以兩數的公因數，再乘上 b，可以減低溢位的風險。

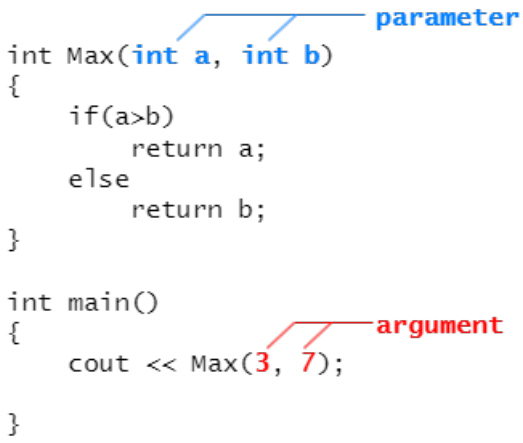
# 6-3 傳值呼叫 與 傳參考呼叫

## 參數與引數

在提到函數與呼叫使用函數時，我們會用到 **參數(parameter)** 和 **引數(argument)** 這兩個名詞。

我們可以簡單的用這張圖來區分他們。

- **參數(parameter)** 是在定義函數時，用來承接傳入資料的變數。
- **引數(argument)** 是在呼叫使用函數時，傳入的資料。




然而在大多數的情況下，大家並不會區分的那麼清楚，很多時候我們都會用 **參數** 來意指兩者。在後續的內容裡除非特別需要指出其不同，否則我們會使用 **參數** 這個詞。

## 傳值呼叫(call by value)

在叫用函數時，我們通常都會傳入數個參數給該函數，例如底下這個求等差數列第 n 項的函數 An( )。

```
int An(int a, int d, int n)
{
  return a+(n-1)*d;
}

int main()
{
  cout << An(1, 2, 10) << endl;  // 19
  cout << An(2, 3, 5) << endl;   // 14

  return 0;
}
```

你可以這樣想像，在第 9 行叫用 An(1, 2, 10) 的時候

1. An 函數產生了 a, d, n 這三個變數，用來承接傳入的引數
2. a 接收到了 1, d 接收到了2, n 接收到了 10
3. 回傳 a+(n-1)*d 的運算結果
4. An 函數之前產生的 a, d, n 三個變數消滅不再存在
5. 返回叫用函數的地方(第9行)，繼續執行下去。

當第 10 行叫用 An(2, 3, 5) 的時候，以上流程會再發生一次。請注意 2 個重點：

1. a, d, n 都是區域變數，當 An() 被叫用時會產生一份區域變數，返回時這些區域變數就會消滅。
2. 叫用 An( ) 時，參數的「值」被複製一份給 a, d, n。所以我們叫它 **傳「值」呼叫 (call by value)**。

接下來這個 exchange 函數會讓你把這個機制的第2個重點看得更清楚。

```
void exchange(int a, int b)
{
  int t = a;
  a = b;
  b = t;
}

int main()
{
  int a = 3;
  int b = 5;

  exchange(a, b);

  cout << "a = " << a << endl;  // a = 3
  cout << "b = " << b << endl;  // b = 5

  return 0;
}
```

第 13 行叫用 exchange(a, b)時，在 main( ) 裡的 a, b 和 exchange( ) 裡的 a, b 是互不相關的。

**外面的**(main的) a, b 只是把它當下的值複製一份傳給**裡面的**(exchange的) a, b。

在函數裡的 a, b 在 t 的協助下互相交換其值，並且在離開函數回到 main 裡繼續執行前，函數裡的 a, b, t 都消滅了。

函數結束回到 main 裡，接著用 cout 輸出 a, b，這個被輸出的是 main 的 a, b。由於剛才互相交換值的是 exchange 函數內的 a, b，和現在 main 的 a, b 一點關係都沒有，所以輸出的 a 還是 3，b 還是 5。

# 傳參考呼叫(call by referance)

如果我們真的需要一個函數，能夠幫我們把外面的兩個變數值交換，必須使用 **「傳參考呼叫(call by reference)」**。

唯一不同的地方是在函數的參數列裡，把要被傳入的變數前面加上 `&`。

```
void exchange(int &a, int &b)
{
  int t = a;
  a = b;
  b = t;
}

int main()
{
  int a = 3;
  int b = 5;

  exchange(a, b);

  cout << "a = " << a << endl;  // a = 5
  cout << "b = " << b << endl;  // b = 3

  return 0;
}
```

使用傳參考時，你可以想像外面的變數真的被傳進去了，你在函數裡對它做什麼，實際上真的會作用在外面的變數上。

你也會看到有人會這麼描述傳參考呼叫 **「參考就是別名(alias)」**。用下面這個例子比較容易理解這個別名的概念。

我們把傳入的`引數 a` 取個`別名叫 c`，把傳入的 `引數b` 取個 `別名叫 d`。於是在函數裡提到的 c 實際上就是外面的 a，在函數裡提到的 d 實際上就是外面的 b。

```
void exchange(int &c, int &d)
{
```

```
  int t = c;
  c = d;
  d = t;
}

int main()
{
  int a = 3;
  int b = 5;

  exchange(a, b);

  cout << "a = " << a << endl;  // a = 5
  cout << "b = " << b << endl;  // b = 3

  return 0;
}
```

# 6-4 將陣列傳入函數

## 傳址呼叫(call by address)

除了「傳值呼叫」、「傳參考呼叫」外，還有一種參數傳遞方式叫「傳址呼叫」。

為什麼叫「傳址」呢？因為這種方式是直接把變數在記憶體中的 **「位址(address)」** 傳進去給函數，在函數裡我們直接到記憶體中的相應位置去操作這個變數的值。所以傳址呼叫和傳參考呼叫一樣可以動到外面變數的值。

關於傳址呼叫，因為會接涉到記憶體位置和指標(pointer)，比較複雜，我們會稍後再來看這個主題。

不過由於大家可能會有需要把一個陣列傳入函數裡，所以我們先來看要如何做到。

一個陣列裡面的元素可能會有非常多個，把它的值全部複製一份再傳給函數未免太浪費時間。由於陣列裡的每個元素都是相同型別，所佔記憶體大小相同，又在記憶體中連續緊密排列，所以 C/C++ 裡採取的是把陣列開頭的位址傳進去即可。

但是只有開頭，不知道陣列結束在哪裡，所以我們還得把陣列的長度也一併做為引數傳入。

### 範例：將陣列傳入函數

```
int showArray(int A[], int n)
{
  for(int i=0; i<n; i++)
  {
    cout << A[i] << " ";
  }
  cout << endl;
}

int main()
{
  int data[5] = {1, 3, 5, 7, 9};

  showArray(data, n);  // 1 3 5 7 9

  return 0;
}
```

```
1 3 5 7 9
```

# 6-5 全域變數與靜態變數

## 全域變數(Global variable)

一般來說，我們使用函數時會將操作到的變數限制在函數裡，也就是以區域變數的方式使用。如有需要操作到函數外面的變數，我們會用傳參考或傳址的方式來處理。

我們以一個抽號碼牌的程式來示範。

### 練習：抽號碼牌(1)

```
#include <iostream>

using namespace std;

int getTicket(int &num)  // 以傳參考方式遞增外面的 num 變數值
{
    num++;
    return num;
}

int main()
{
    int num = 0;  // 記錄目前發到幾號

    cout << "I have ticket No." << getTicket(num) << endl;
    cout << "I have ticket No." << getTicket(num) << endl;
    cout << "I have ticket No." << getTicket(num) << endl;

    return 0;
}
```

```
I have ticket No.1
I have ticket No.2
I have ticket No.3
```

---

使用這種方式沒什麼問題，但是每次都要傳遞變數 num。如果想避免這個麻煩，可以使用全域變數，也就是把 num 宣告在所有函數(包含 main)的外面。

### 練習：抽號碼牌(2)

```
#include <iostream>

using namespace std;

int num = 0;  // 記錄目前發到幾號。宣告在這裡是全域變數

int getTicket()  // 沒有參數
{
    num++;  // 因為 num 是全域變數，所以到處都可以存取它
    return num;
}

int main()
{
    cout << "I have ticket No." << getTicket() << endl;  // 沒有引數
    cout << "I have ticket No." << getTicket() << endl;  // 沒有引數
    cout << "I have ticket No." << getTicket() << endl;  // 沒有引數

    return 0;
}
```

```
I have ticket No.1
```

```
I have ticket No.2
I have ticket No.3
```

---

使用全域變數雖然很方便，但是它有一個極大的缺點，就是大家都可以動到它。

有時候你會很納悶，明明我沒動它，它的值怎麼變了。找了半天才發現在某個不起眼角落或函數裡的程式碼動到它的值。

# 函數裡的靜態變數(static variable)

一般來說宣告在函數裡的變數都是區域變數(local variable)，一但離開函數後就會消滅，下次被呼叫時才會重新產生出來。

但是如果在宣告時，在前面加上 `static` 修飾詞，它就會是個靜態變數，在離開函數時變數會記得當下的值，不會消滅。下次函數被呼叫時，它依然活著不會被重新產生和給定初值。

## 練習：抽號碼牌(3)

```
#include <iostream>

using namespace std;

int getTicket()  // 沒有參數
{
 static int num = 0;  // 靜態變數，只在程式開始時指定一次初值
    num++;
    return num;
}

int main()
{
    cout << "I have ticket No." << getTicket() << endl;  // 沒有引數
    cout << "I have ticket No." << getTicket() << endl;  // 沒有引數
    cout << "I have ticket No." << getTicket() << endl;  // 沒有引數

    return 0;
}
```

```
I have ticket No.1
I have ticket No.2
I have ticket No.3
```

---

在某些情況下，靜態變數是很好用的！