第 6 章
函式

本章提要

- 6-1 認識函式
- 6-2 傳遞參數
- 6-3 函式傳回值
- 6-4 行內函式
- 6-5 巨集
- 6-6 C++ 標準函式庫
- 6-7 函式多載 (Overloading)
- 6-8 綜合演練

6-1 認識函式

- 隨著程式越寫越大, main() 函式的內容也越來越複雜, 當我們在寫較長的程式時, 可能會發現程式中有許多段功能相同的敘述會重複出現, 而且由於程式的需求, 這些程式碼無法被省略, 只好重複一寫再寫。本章所要介紹的函式, 讓我們能以更簡便的方式, 取代會重複用到的一段程式碼, 提昇寫作程式的效率, 也讓程式的功能模組化。

使用函式的好處

- 所謂**函式** (function) 就是**一組敘述的集合**, 並且以一個函式名稱來代表此敘述集合。如果這一組敘述代表一項經常用到的功能, 我們每次要用到這組敘述時, 只要寫下函式的名稱, 就是告訴編譯器我們要執行這項功能。
- 舉例來說, 如果程式前前後後要計算階乘多次, 且不是連續計算, 因此不適合用迴圈, 這時就變成要讓計算階乘的敘述在 main() 函式中出現多次：

使用函式的好處

```
int  main()
{
   for()...  //  計算階乘的迴圈
   ...       //  其它敘述
   for()...  //  計算階乘的迴圈
   ...       //  其它敘述
   for()...  //  計算階乘的迴圈
   ...       //  其它敘述
}
```

使用函式的好處

- 但如果我們將**計算階乘**的敘述獨立出來寫成一個函式, 以後程式中計算階乘時, 只要**呼叫**這個函式來進行計算即可, 不必每次都要重寫計算階乘的一整段迴圈敘述：(如下頁)
- 如此一來, 計算階乘的動作在 main() 函式中只需以一行簡單的敘述代替即可, 大幅省下撰寫重複程式的時間。如果函式的內容愈複雜, 也代表節省下來的時間更多。

使用函式的好處

```
int fact() {...}                    // 將計算階乘的敘述放在一個函式中

int main()
{
   fact();    // 呼叫函式計算階乘
   ...        // 其它敘述
   fact();    // 呼叫函式計算階乘
   ...        // 其它敘述
   fact();    // 呼叫函式計算階乘
   ...        // 其它敘述
}
```

使用函式的好處

- 簡單的說, 使用函式的好處包括：
  - 將具有特定功能的敘述獨立成函式, 可提高程式的可讀性。
  - 將程式模組化, 讓程式碼可重複使用, 提升寫程式的效率。
  - 將程式分解成函式, 發生錯誤時, 可以很容易找出問題在哪一個函式, 提高除錯的效率。

函式的定義

- 函式和變數一樣, 在使用前必須先定義。定義函式就是定義函式的資料型別、呼叫 (call) 函式 (程式中**使用**函式的動作就稱為**呼叫**函式) 時所需的參數、以及函式所要執行的動作 (敘述) :

```
型別   函式名稱   (型別 參數1,   型別 參數2, ...)
{
   敘述的集合...      // 函式內容
}
```

函式的定義

- **型別**：函式和變數一樣都有型別, 不過函式的型別並不是函式儲存的資料類型, 而是函式**傳回值**的資料類型。函式處理完工作後, 可以將處理的結果以**傳回值**的方式傳回給呼叫它的敘述, 讓程式可據以做進一步的處理。要將資料傳回, 需使用 return 敘述, 例如 “return 0;” 表示傳回 0, 若在 return 敘述後放變數或運算式, 就表示傳回變數值或運算式的結果。函式也可以沒有傳回值, 此時需將函式的型別宣告為 **void**。
- **函式名稱**：函式的命名規則和變數相同, 且不可與變數名稱重複。

函式的定義

- **參數**：參數就像數學公式中的變數一樣, 同一個公式以不同變數值代入計算, 即可得到不同的結果。參數就是函式的變數, 每次以不同的參數值呼叫函式, 即可得到不同的結果。當然我們也可設計沒有參數的函式。此處所列的參數名稱, 是在函式中用來代表參數的變數名稱, 呼叫函式時, 可使用變數、運算式、常數來呼叫。
- **函式本體**：大括號的部份就稱為函式本體 (function body), 在大括號中可放入任何要執行的敘述。

函式本體

- 在函式本體中我們可放入希望該函式執行的任何動作，在此先舉個簡單的例子：若程式每處理一段事情，就要用嗶聲提醒使用者，我們可將發出嗶聲的動作獨立成一個函式：

```
void beep()                  //  此函式不需有傳回值，故型別爲 void
{                            //  函式也沒有參數
   cout << "工作完成\a\n";      //  函式本體只有一行
}
```

函式本體

1. 這個函式只是單純發出嗶聲, 所以不需有傳回值, 因此函式的型別為 void。
2. 函式沒有任何參數, 在函式名稱後的括號保持空白。
3. 函式本體只有一行敘述, 就是由 cout 輸出**工作完成**的訊息, 並以字元 '\a' 使電腦發出嗶聲。

- 將這個函式加到程式中, 我們就可在 main() 函式中每處理某件工作告一段落時, 呼叫 beep() 函式使電腦發出嗶聲, 呼叫的方式就是**函式名稱();**, 請見以下的範例。

函式本體

程式 Ch06-01.cpp 呼叫函式以顯示訊息及發出嗶聲

```
01 #include <iostream>
02 using namespace std;
03
04 void beep()          // 定義一個 beep() 函式
05 {
06    cout << "工作完成\a\n";     // 讓電腦發出嗶聲
07 }
08
09 int main()
10 {
11    cout << "現在開始處理工作" << endl;
12
13    for (int i=0; i < 50000000; i++)
```

函式本體

```
14          ; // 用執行五千萬次加法及比較的迴圈模擬電腦在做一件事
15      beep();                 // 處理完畢, 呼叫 beep() 函式
16
17      for (int i=0; i < 80000000; i++)
18          ; // 用執行八千萬次加法及比較的迴圈模擬電腦在做另一件事
19      beep();                 // 處理完畢, 呼叫 beep() 函式
20  }
```

執行結果

```
現在開始處理工作
工作完成
工作完成
```

依電腦速度快慢, 這兩行訊息會『稍後』才顯示, 同時電腦也會發出嗶聲

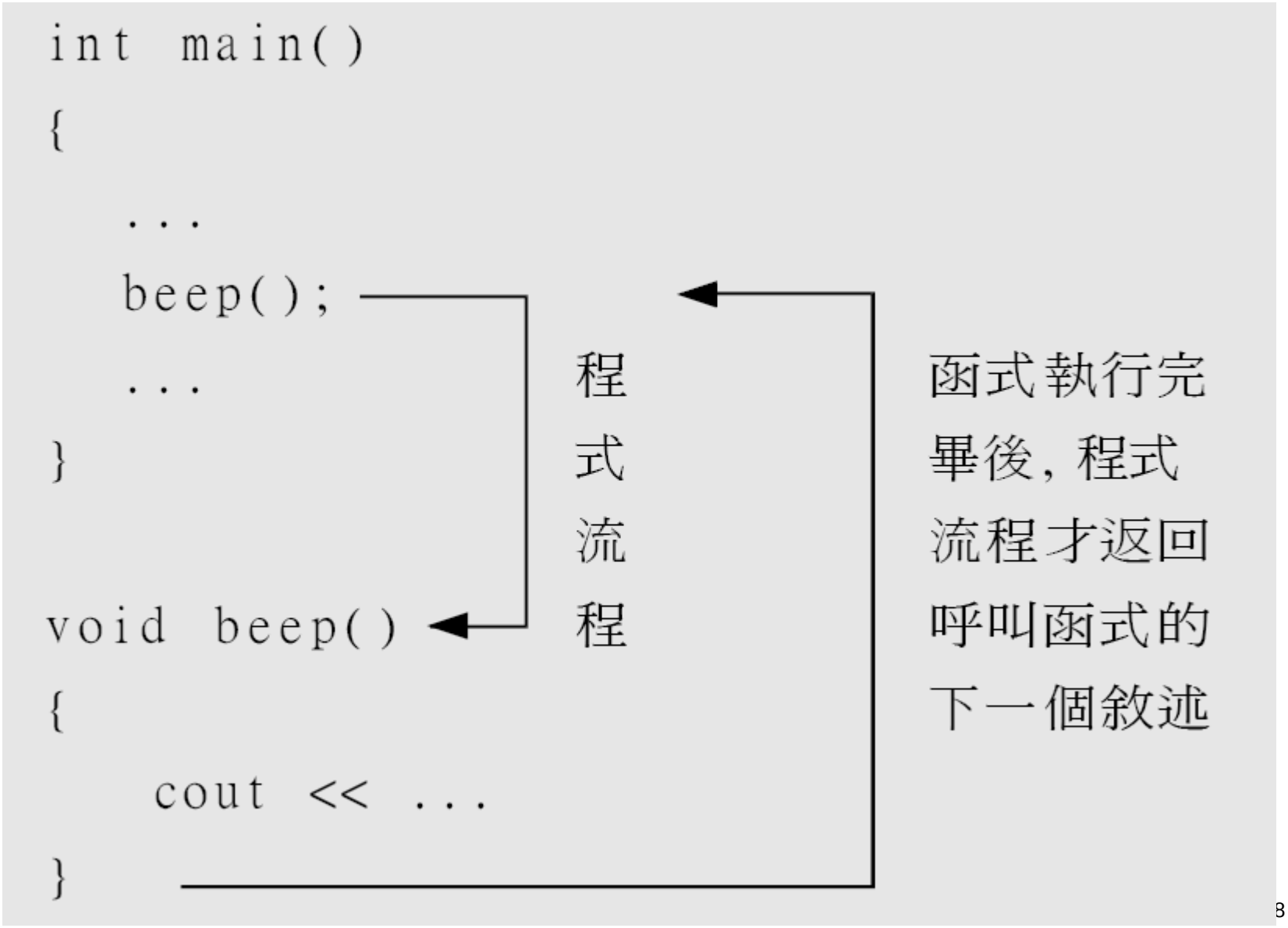
函式本體

1. 第 4～7 行就是先定義自訂函式的內容, 以免稍後在 main() 函式中呼叫函式的敘述, 會讓編譯器發出找不到識別字的錯誤訊息。
2. 第13、17 行的 for 迴圈都緊跟著一個沒有任何運算式的空敘述 (也就是只有一個分號的敘述), 但 for 迴圈本身的條件運算式及控制運算式仍要執行, 所以這兩個迴圈等於讓電腦做了數千萬次的加法運算 (i++) 及比較運算 (i<...), 我們用此方式來模擬程式正在執行某項工作。

函式本體

3. 第 15、19 行的敘述就是呼叫 beep() 函式。此時執行流程會先跳進 beep() 函式中, 待 beep() 函式的工作執行完畢, 才返回 main() 函式中的下一行敘述繼續執行。

函式本體


int main()
{
...
beep();
...
}
void beep()
{
cout << ...
}
程式流程
函式執行完畢後，程式流程才返回呼叫函式的下一個敘述

函式本體

- 程式每次執行完指定的工作後, 就會呼叫 beep() 函式, 所以我們會在螢幕上看到**工作完成**的訊息, 同時也會聽到電腦發出的嗶聲。透過函式呼叫的方式, 我們可用較簡單的 "beep()" 取代一串 "cout <<..." 敘述, 而且 main() 函式的內容也更簡捷, 閱讀起來更容易瞭解其意思。

函式的宣告

- 如果不想將函式的定義寫 在 main() 函式之前, 或是函式是定義在其它的程式檔等狀況, 則在呼叫函式之前必須先宣告函式。函式宣告又稱為函式的原型 (prototype), 函式宣告通常會放在整個程式的最前面、所有前置處理指令之後。宣告函式時只需列出函式的傳回值型別、函式名稱、及各參數的型別即可：

```
型別    函式名稱    (參數 1 的型別,    參數 2 的型別,  . . . );
```

函式的宣告

- 例如前述的範例程式可改成：

程式 Ch06-02.cpp 呼叫比較整數大小的函式

```
01 #include <iostream>
02 using namespace std;
03
04 void beep();                    // 宣告函式
05                                  // 沒有函式本體
06 int main()
07 {
08    cout << "現在開始處理工作" << endl;
```

函式的宣告

```
09
10      for (int i=0; i < 50000000; i++)
11          ; // 用執行五千萬次加法及比較的迴圈模擬電腦在做一件事
12      beep();                 // 處理完畢, 呼叫 beep() 函式
13
14      for (int i=0; i < 80000000; i++)
15          ; // 用執行八千萬次加法及比較的迴圈模擬電腦在做另一件事
16      beep();                 // 處理完畢, 呼叫 beep() 函式
17  }
18
19  void beep()                 // 函式定義在 main() 之後
20  {
21      cout << "工作完成\a\n";      // 讓電腦發出嗶聲
22  }
```

6-2 傳遞參數

- 參數型別
- 參數的預設值
- 變數的作用範圍與生命週期

參數型別

- 沒有參數的函式功能有限, 因為它的形式固定, 能做的事情就少有變化。而透過參數的設計, 每次呼叫函式時, 就能將不同的變數傳遞給函式, 讓函式能針對不同的數值、狀況進行處理, 如此函式的執行結果就能有所變化。
- 舉例來說, 我們要計算華氏溫度轉換成攝氏溫度, 若 f 為攝氏溫度, 則轉換公式可寫成：

```
攝氏溫度 = (f - 32) * 5 / 9
```

參數型別

- 若將它設計成一個函式, 我們可讓 f 為函式參數, 這樣每次用不同數值呼叫函式, 就能計算不同華氏溫度轉換成攝氏溫度的情形：

```
void FtoC (double f)   // 參數 f 爲 double 型別
{
   cout << "換算成攝氏溫度爲 "
        << ((f - 32) * 5 / 9) << " 度";
}                  // 每次呼叫函式用的 f 值不同, 函式計算結果就不一樣
```

- 以下就是這個溫度換算函式的應用方式：

參數型別

程式 Ch06-03.cpp 將華氏溫度轉成攝氏溫度

```
01 #include <iostream>
02 using namespace std;
03
04 void FtoC (double f)          // 將華氏溫度轉成攝氏的函式
05 {
06    cout << "換算成攝氏溫度爲 "
07         << ((f - 32) * 5 / 9) << " 度";   // 溫度轉換公式的算式
08 }
09
10 int main()
11 {
12    double x;
13    cout << "請輸入華氏的溫度：";
```

參數型別

```
14      cin >> x;
15      FtoC(x);                    // 用 x 爲參數呼叫 FtoC()
16  }
```

執行結果

```
請輸入華氏的溫度：59
換算成攝氏溫度爲  15  度
```

參數型別

- 第 15 行用 x 呼叫 FtoC() 函式時, 編譯器會將 x 的數值傳遞給 FtoC() 函式的參數 f, 以此處為例, 就是將 59 傳遞給 FtoC(), 而此時 FtoC() 中的 f 參數值即為 59。(如下頁)
- 因此 f=59 代入第 7 行的運算式時, 即算出攝氏溫度為 15, 所以執行結果會輸出 15 度。若在執行程式時輸入不同的數值, main() 也會將該數值傳遞給 FtoC() 函式, 讓它計算出不同的結果。

參數型別

```
void FtoC (double f)
{
  ...
}

int main()
{
   ...
   FtoC(x);
}
```

呼叫函式時，x 的值會『傳遞給』f，

若 x 爲 59，f 就會是 59；

若 x 爲 68，f 就會是 68

參數型別

- 原則上函式宣告的參數型別為何, 我們呼叫函式時也需使用相同型別的變數或常數。不過函式參數也適用第 4 章介紹過的自動轉型功能, 例如在上例中用 int 變數呼叫 FtoC() 函式, 則編譯器會將自動將變數轉型為 double 型別。若是無法轉型成功, 則編譯程式時會出現錯誤。

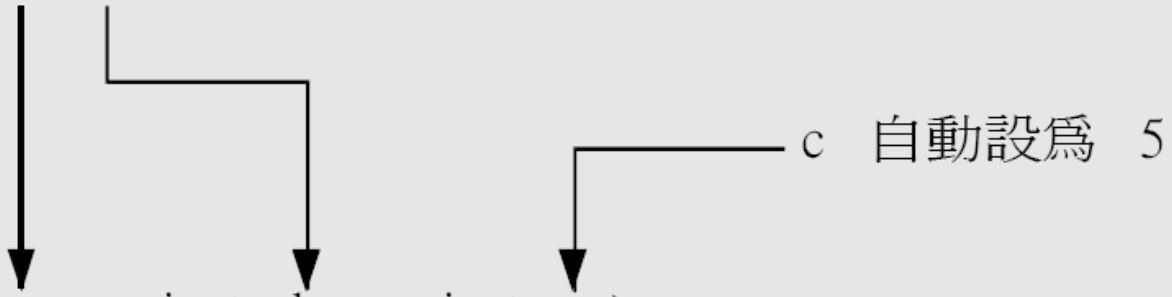
參數的預設值

- 函式的參數可以有預設值, 當呼叫函式時, 若省略於定義中有設定預設值的參數, 則該參數將自動設為預設之值：

參數的預設值

```
funct(int a, int b, int c=5);   // 宣告函式時即設定預設值
...
int main()
{
   ...
   funct( 1, 2);
   ...
}

funct(int a,   int b,   int c)
{ ... }
```



參數的預設值

- 具有預設值的參數可以有多個, 但都必須集中在參數列的**最右邊**, 例如以上面的 funct() 函式而言, 不可以 a、b 有設預設值, 而 c 沒有設；或是 a、c 有預設值, 而 b 沒有。此外要注意, 如果在程式檔內同時有函式的宣告與定義, 每個參數的預設值設定只能出現在一個地方, 不可重複設定：

參數的預設值

```
funct(int  a=5);      //  宣告時指定預設值
...
...
funct(int  a=5)       //  錯誤：在定義函式時又設定一次
{...}
```

程式 Ch06-04.cpp 參數有預設值的函式

```
01 #include <iostream>
02 using namespace std;
03
```

參數的預設值

```
04 double newV(double t, double a=9.8, double v0 = 0)
05 {                                  // 兩個參數有預設值
06    return v0 + a*t;
07 }
08
09 int main()
10 {
11    cout << "速度與加速度的計算示範：V=V0+at" << endl;
12
13    cout << "若 V0 = 100, a = 2.8, t =15, 則 "
```

參數的預設值

```
14          << "V = " << newV(15,2.8,100) << endl;
15
16     cout << "若 V0 = 0   , a = 9.8, t =15, 則 "
17          << "V = " << newV(15);   // 只傳一個參數
18  }
```

執行結果

速度與加速度的計算示範：V=V0+at

若 V0 = 100, a = 2.8, t =15, 則 V = 142

若 V0 = 0   , a = 9.8, t =15, 則 V = 147

變數的作用範圍與生命週期

- 為什麼呼叫函式時要傳參數給它？最主要的原因, 就是函式不能直接存取我們在 main() 中所宣告的變數, 同理, main() 也不能存取在其它自訂函式中所宣告的變數。在 C++ 中, 所有的變數依宣告方式的不同, 而有不同的作用範圍 (scope) 及生命週期 (life time)：
  - **作用範圍**：或稱為視野, 就是個體的活動範圍, 也就是我們可接觸到該個體的區間。在程式執行時, 有些變數在任何地方都可以存取到, 有些則只能在固定的檔案內、函式內、或由 { } 括起來的區塊內才能存取。

變數的作用範圍與生命週期

- **生命期**：就是個體存在的時間。在程式執行時，有些變數會持續存在，直到程式結束為止；另一些變數則在執行到某段程式才被建立，離開該段程式後就自動消失；還有一些變數則可以只在需要時才建立，直到不需要時再將其釋回。

- 這兩個特性合稱為變數的儲存等級 (Storage class)，以下我們就來看各種變數的宣告方式，及其與函式的關係。

局部變數

- 到目前為止, 我們所學的變數宣告方式, 就稱為局部變數 (local variable), 意思是指這些變數只能用在程式中的某個區域, 在函式內部定義的變數就是局部變數, 它們只能用在函式的大括號所括起來的範圍內。局部變數也稱為自動 (auto) 變數, 在宣告時前面可加上關鍵字 auto 註明, 不過一般都省略掉, 因為只要宣告變數時未加上儲存等級關鍵字, 編譯器都會視為局部 (自動) 變數：

局部變數

```
auto int a;      // 兩種寫法都是局部變數
int b;
```

- 局部變數的作用範圍只及於所在的函式內部,所以其它函式無法使用這些變數。

```
funct(int a, char b)
{
    int a;   // 錯誤：名稱重複
    ...
}
```

局部變數

- 局部變數的生命期是由所屬函式被呼叫開始, 至函式結束執行而消失。
- 局部變數是以堆疊 (stack) 來存放的。當函式被呼叫時, 函式的堆疊才會被配置, 因此局部變數才有存放的空間並開始啟用。而在函式結束時, 函式的堆疊會被取消, 因此局部變數也同時被消滅, 不但其數值不保存, 且其所佔空間 (堆疊) 也不被保留。請參考以下的範例：

局部變數

程式 Ch06-05.cpp 在函式中累加局部變數

```
01 #include <iostream>
02 using namespace std;
03 void adding();                    // 沒有傳回值, 需註明爲 void
04
05 int main()
06 {
07    for (int i=0;i<3;i++)          // 呼叫 adding() 三次
08       adding();
09 }
10
11 void adding(void)
12 {
```

局部變數

```
13      int number=100;                    // 局部變數, 有初始值
14      cout << "number = " << number++ << endl;
15  }
```

執行結果

```
number = 100
number = 100
number = 100
```

局部變數

- 雖然在第 14 行用 number++ 改變了 number 的值, 但第 23 次呼叫函式時仍顯示 number 的值為 100 而非 101 或 102。這是因為在前 1 次呼叫函式輸出 number 並遞增後, 函式就結束了, 變數 number 的生命期也跟著結束, 所以累加的結果並未保存下來。下次呼叫函式時, number 又重新被建立並設為初始值 100, 所以每次呼叫函式時都只會看到 number 等於 100。

局部變數

- 此外, 在 for、while 等迴圈中所定義的變數也是局部變數, 且其作用範圍僅及於迴圈的大括號中。
- 函式的宣告和定義也一樣可放在別的函式的大括號內, 如此一來函式的視野就變小, 變得和局部變數一樣無法供整個檔案使用。例如：

局部變數

```
int  func1(int);

int  main()
{
   int  func2(int);     // 宣告在 main() 函式內
}

int  func1(int  x) {...}   // 看不到 func2() 的宣告, 所以無法呼叫 func2()


int  func2(int  x) {...}
```

靜態局部變數

- 如果要讓局部變數的生命期變長, 使變數的值到下一次函式呼叫時仍能保存下來, 不會因為函式結束, 變數的值就消失不見, 則可在宣告局部變數時, 加上 **static** 關鍵字, 讓變數成為**靜態局部變數**。

靜態局部變數

- 靜態局部變數的作用範圍與一般局部變數無異, 但生命期則不同。指定為 static 後的局部變數不再存放於堆疊, 而是在程式一開始執行時就存於固定的記憶體位址, 因此並不隨函式的結束而消逝, 而函式再次執行時, 上次保留下來的變數值也可以繼續使用。

靜態局部變數

程式 Ch06-06.cpp 在函式中累加靜態局部變數

```
01 #include <iostream>
02 using namespace std;
03 void adding(void);          //沒有傳回值及參數
04
05 int main()
06 {
07    for (int i=0;i<3;i++)          // 呼叫 adding() 三次
08       adding();
09 }
10
11 void adding(void)
```

靜態局部變數

```
12 {
13     static int number=100;    // 靜態局部變數
14     cout << "number = " << number++ << endl;
15 }
```

執行結果

```
number = 100
number = 101
number = 102
```

靜態局部變數

- 在第 13 行加上 static 關鍵字後, number 變數的生命期就不再受限於單次的函式呼叫, 變數的值也會保存下來, 所以每次呼叫都會輸出前一次 ++ 運算的結果。
- 請注意! static 變數只會在第一次建立時做初值設定, 下次函式再呼叫時便不會再做初值設定了, 所以其值得以持續保留, 這也是和一般局部變數不同之處。

全域變數

- 與局部變數相對應的變數類型稱為全域變數 (Global variable)。全域變數是定義在函式大括號以外的變數, 也和靜態局部變數一樣, 是自程式開始就存在的, 但其視野則是自定義開始, 一直到程式最後, 所以定義在程式開頭的全域變數, 程式中所有的函式都

```
#include <iostream>
...
int a;    // 全域變數, 以下所有函式都可存取
```

全域變數

```
int  main( )
{
   ...     //  可存取變數  a
}

int  b;    //  全域變數,  func( )  可存取

int  func( )
{
  ...    // 可存取變數 a 和 b
```

全域變數

- 由於全域變數生命期是自程式開始一直到程式結束，所以其值也可一直保存下來。利用這項特性，我們可將所有函式都會用到的變數宣告成全域變數，如此可減少宣告變數的次數。例如前面的溫度轉換函式，利用全域變數的方法，可改寫成：

程式 Ch06-07.cpp 利用全域變數存放共用的數值

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

全域變數

```
03 double f;            // 用來儲存華氏溫度的全域變數
04
05 double FtoC (void)                  // 不需傳遞參數
06 {
07    return (f - 32) * 5 / 9;    // 存取全域變數
08 }
09
10 int main()
11 {
12    cout << "請輸入華氏的溫度：";
13    cin >> f;                          // 將輸入的值存入全域變數
14
15    cout << "換算成攝氏溫度爲 " << FtoC() << " 度";
16 }
```

全域變數

執行結果

```
請輸入華氏的溫度：14
換算成攝氏溫度爲  -10  度
```

- 第 3 行宣告的全域變數 f 可供全部的函式存取, 所以 main() 函式不用再宣告存放華氏溫度的局部變數, 也不用傳參數給轉換函式 FtoC(), 後者也可直接從變數 f 取得計算所需的華氏溫度值。

外部變數

- 如果變數定義在程式後面, 或甚至是定義在其它的程式檔中, 則可以經由特別的宣告來拓展該變數的作用範圍, 讓程式可順利存取該變數, 此種變數稱為外部 (external) 變數。外部變數的宣告方法是在資料型別前加上 extern 關鍵字：

```
extern    int    exta;      // 因爲只是宣告，所以不可設定初始值
```

- 這樣外部變數的宣告便可以和定義有所區別了 (因宣告的前頭有加 extern)。注意, 若以 extern 宣告時有設定初值, 則會被視為**定義**而非宣告。

全域變數與局部變數

- 第 2 章曾提過變數名稱不可重複, 不過這其實是指同一視野內的變數而言, 例如同一函式內的局部變數名稱彼此不能重複, 全域變數名稱彼此不能重複。但是不同函式的局部變數, 以及全域變數與局部變數之間, 則可以有同名的變數。

全域變數與局部變數

- 不同函式間的同名局部變數並沒有什麼特別，因為這些變數的視野都只在所宣告的函式內，所以不會互相影響。但是如果全域變數與局部變數名稱相同，那麼我們在函式中存取該名稱的變數時，究竟是存取全域變數或局部變數？
- 當局部變數與全域變數同名時，會產生遮蔽效應，也就是說函式只能**看到**局部變數，所以存取時也是存取到局部變數。如果需要使用同名的全域變數，則必須使用**範圍解析運算子 ::** 來標示變數名稱，表示要存取同名的全域變數：

全域變數與局部變數

程式 Ch06-08.cpp 局部變數與全域變數同名

```
01 #include <iostream>
02 using namespace std;
03 int x=10;    // 全域變數
04
05 int main()
06 {
07    int x=100;
08    cout << "局部變數 x = " << x << '\t'
09         << "全域變數 x = " << ::x << endl;   // 用 :: 存取全域變數
10
11    x += ::x++;
12
13    cout << "局部變數 x = " << x << '\t'
14         << "全域變數 x = " << ::x << endl;
15 }
```

全域變數與局部變數

執行結果

```
局部變數 x = 100          全域變數 x = 10
局部變數 x = 110          全域變數 x = 11
```

- 程式第 9、11、14 行存取 x 時, 都在變數名稱前加上範圍解析運算子, 表示存取的是全域變數, 就不會像只寫 x 時只會存取到局部變數。
- 不過這種寫法容易造成程式除錯困難, 且會降低程式可讀性, 因此應避免使用。

6-3 函式傳回值

- 我們可利用傳回值將函式處理的結果傳回呼叫者, 本節就來介紹一些有關傳回值的函式設計方式。
  - 無傳回值
  - 多個 return 敘述

無傳回值

- 如果函式做好處理工作後, 不需將結果傳回呼叫者, 此時可將函式宣告為沒有傳回值, 也就是將函式的型別宣告為 void。宣告為 void 的函式, 因為不必傳回任何數值給呼叫者, 所以不必用到 return 敘述, 而且如果真的用 re-turn 傳回某個數值, 編譯器還會視為錯誤。以下就是一個沒有傳回值的函式應用範例：

無傳回值

程式 Ch06-09.cpp 無傳回值的函式

```
01 #include <iostream>
02 using namespace std;
03 void showresult(double);          // 沒有傳回值的函式
04                                    // 要宣告爲 void
05 int main()
06 {
07   int sex;                        // 代表性別選項
08   double height, weight;          // 身高與體重
09
10   do {   // 利用迴圈要求使用者一定要選擇 1 或 2
11      cout << "性別:(1)男(2)女：";
12      cin >> sex;
```

無傳回值

```
13    } while (sex!=1 && sex!=2);
14
15    cout << "請輸入身高(公分)：";
16    cin >> height;
17
18    if (sex == 1)
19       weight = (height - 80)*0.7; //  男性的標準體重公式
20    else
21       weight = (height - 70)*0.6; //  女性的標準體重公式
23
24    showresult(weight);                //  呼叫顯示結果的函式
25 }
26
27 void showresult(double result)   //  定義輸出結果的函式
```

無傳回值

```
28 {
29     cout << "您的標準體重範圍是 " << endl << result * 0.9
30          << " 公斤至 " << result * 1.1 << " 公斤之間";
31 }
```

執行結果

```
性別(1)男(2)女：1
請輸入身高(公分)：199
您的標準體重範圍是
74.97 公斤至 91.63 公斤之間
```

無傳回值

- 範例程式中的 showresult() 函式的工作只是單純顯示標準體重資訊, 無需傳回任何資料給呼叫者, 所以一開始的原型宣告就將之宣告為 void, 且函式中沒有 return 敘述。

多個 return 敘述

- 雖然函式只能有 1 個傳回值, 但不表示函式中只能有 1 個 return 敘述, 我們可利用條件分支的方式, 在不同狀況下傳回不同的計算結果, 如此就可設計出具有多個 return 敘述的函式。當然, 函式在執行時, 仍只會執行到 1 個 return 敘述是無庸置疑的。我們將計算標準體重的程式略做修改, 示範多個 return 敘述的應用：

多個 return 敘述

程式 Ch06-10.cpp 計算標準體重的函式

```
01 #include <iostream>
02 using namespace std;
03
04 double stdWeight(int sex, double height)   // 體重計算函式
05 {
06   if (sex == 1)       // 男
07     return (height - 80) * 0.7;
08   else                // 女
09     return (height - 70) * 0.6;
10 }
11
12 int main()
```

多個 return 敘述

```
13 {
14    int sex;                          // 代表性別選項
15    double height, weight; //  身高及體重
16
17    do {   // 一定要選擇 1 或 2
18       cout << "性別(1)男(2)女：";
19       cin >> sex;
20    } while (sex!=1 && sex!=2);
21
22    cout << "請輸入身高(公分)：";
23    cin >> height;
24
25    weight = stdWeight(sex, height);
26    cout << "您的標準體重範圍是 " << endl << weight * 0.9
27         << " 公斤至 " << weight * 1.1 << " 公斤之間";
28 }
```

執行結果

```
性別(1)男(2)女：2
請輸入身高(公分)：170
您的標準體重範圍是
54 公斤至 66 公斤之間
```

6-4 行內函式

- 定義行內函式
- 行內函式對程式的影響

定義行內函式

- 行內函式 (inline function) 是一種特殊的函式, 從外觀看, 它和一般的函式無異, 只是在定義時, 函式傳回值型別前要加上一個 inline 關鍵字。但編譯器對行內函式的處理則是迥異於一般的函式, 因為當編譯器看到函式被定義為 inline 時, 就會對函式做**行內擴展** (inline expand) 的動作, 也就是將程式中每一個呼叫函式的敘述, 都代換成對應的行內函式本體內容：

定義行內函式

```
inline getmax(int a, int b)
   { return (a>b ? a : b); }
....
a = getmax(i, j);    //  編譯器會將此行代換成
                     //    a = (i>j ? i : j);
```

- 編譯器在處理行內函式的呼叫時, 會試圖以最經濟的方式來將函式內的動作直接展開在程式中。如果有參數, 當然也會做型別的檢查和自動轉換。

定義行內函式

程式 Ch06-11.cpp 將華氏溫度轉成攝氏溫度的行內函式

```
01 #include <iostream>
02 using namespace std;
03
04 inline double FtoC (double f)     // 定義爲行內函式
05 {
06   return (f - 32) * 5 / 9;
07 }
08
09 int main()
10 {
11   double F;
```

定義行內函式

```
12    cout << "請輸入華氏的溫度：";
13    cin >> F;
14
15    cout << "換算成攝氏溫度爲 " << FtoC(F) << " 度";
16 }
```

- 這個範例程式基本上與之前的範例沒什麼不同，主要的差異是在此範例中我們將 FtoC() 定義為行內函式，因此編譯器會試著做行內擴展，以提高程式執行效率。

定義行內函式

- 請特別注意, 並非加上 inline 關鍵字, 函式就一定會變成行內函式。基本上 inline 對編譯器是**參考用**, 編譯器仍會自行判斷該函式是否合適做為行內函式。舉例來說, 如果函式的內容很複雜, 例如是計算最大公因數的函式, 即使加上 inline 關鍵字, 編譯器仍不會將該函式設為行內函式, 自然也不會做行內擴展的處理。

行內函式對程式的影響

- 行內函式由於會被代換成函式本體的敘述，所以實際執行時並沒有像使用一般函式時，程式執行流程會在呼叫函式時跳到函式中、執行完畢後返回呼叫者的切換動作，所以程式的執行效率會因此而提升。
- 但執行速度加快的代價，則是編譯後產生的執行檔會因為行內擴展而使程式檔變大，所以一般只有內容很少的函式我們才會考慮使用行內函式。

行內函式對程式的影響

- 此外，若專案中有數個程式檔都要使用同一個行內函式，此時每個檔案中都要有該行內函式的定義，而且定義內容必須一致。因此在這種情況下，通常會將行內函式的定義放入自訂的表頭檔，然後讓每個程式檔都去含括這個表頭檔。

6-5 巨集

- 定義及使用巨集
- 使用巨集的注意事項

定義及使用巨集

- 巨集 (macro, 或稱巨集函式) 是另一類似行內函式機制的程式寫法。它的功能和第 3 章介紹的巨集常數類似, 我們可將一簡單的運算式定義成巨集, 當前置處理器在處理程式檔時, 就會將巨集的名稱代換成指定的運算式, 就和代換巨集常數的動作一樣。
- 例如程式中可能要計算變 數 i 的三次方多次, 每次都要寫 "x*x*x" 實在有點累, 就可將它寫成如下的巨集：

定義及使用巨集

```
#define   x-cube    x * x * x
...
int main()
{
   int x;
   ...
   int y = x-cube;          // 會被換成 "int y = x * x * x;"
   ...
}
```

定義及使用巨集

- 但這種寫法實用性不高, 因此巨集也可像函式一樣用括號指定參數, 但不用指定參數型別。例如：

```
#define   cube(x)    x * x * x    // 計算立方的巨集
```

- 使用此巨集時就像使用函式一樣, 將要計算立方的變數代入 cube(x) 中的 x 即可：

```
int i = 3;
cout << cube(i);   // 相當於 cout << i * i * i;
```

定義及使用巨集

- 在上例中，我們以 #define 前置處理指令把 cube(x) 定義成 "x * x * x"，所以程式中的 cube(x) 在編譯之前就會被前置處裡器代換成 "i * i * i"。請參考以下的範例：

程式 Ch06-12.cpp 利用巨集計算立方

```
01 #include <iostream>
02 using namespace std;
03 #define  cube(x)    x * x * x    // 計算立方的巨集
04
```

定義及使用巨集

```
05 int main()
06 {
07    for(int i=1;i<10;i+=2)
08       cout << i << " 的三次方等於 "
09            << cube(i) << endl;
10 }
```

執行結果

```
1 的三次方等於 1
3 的三次方等於 27
5 的三次方等於 125
7 的三次方等於 343
9 的三次方等於 729
```

定義及使用巨集

- 巨集函式的好處和行內函式一樣，就是以展開後的程式碼取代巨集，因此不像使用函式要浪費函式呼叫 / 返回的時間，因而加快程式執行的速度。而使用巨集取代特定的運算式，也有助於提高程式的可讀性。

使用巨集的注意事項

- 使用巨集和函式相比, 有一項不方便之處, 就是巨集參數若是代入運算式, 可能會造成運算結果錯誤。以參數呼叫函式時, 不管參數是變數或運算式, 編譯器都是將其值 (運算式的執行結果) 傳遞給函式的參數；但巨集的運作則只是由前置處理器將參數**代換**到巨集中參數出現的位置, 若參數是運算式, 前置處理器也不會處理 (也無法處理, 因為前置處理器不懂 C++ 敘述), 而只是單純做文字的替換而已。

使用巨集的注意事項

- 所以前述的求立方巨集, 如果使用 cube(i+1) 這樣的呼叫, 則會出現錯誤的結果。因為此時巨集會被展開成：

```
#define   cube(x)     x * x * x     // 計算立方的巨集
...
cube(i+1) ──────────► i+1 * i+1 * i+1
          替換成
```

- 由於運算子 * 的優先性較高, 此式相當於 i+i+i+1, 根本不是計算 (i+1) 的立方。

使用巨集的注意事項

- 解決的方法之一是在 #define 時, 就將每個參數都加 () 括號包圍起來, 由於 () 的優先性最高, 這就能保證巨集的運算不致因運算子優先性的問題而導致結果錯誤。比如：

```
#define   cube(x)     (x)*(x)*(x)     //  計算立方的巨集
```

- 不過這樣也只能解決部份的問題。如果巨集的參數中有遞增或遞減運算, 仍是會造成結果不正確, 例如：

```
int  i  =  3;
...
cout  <<  cube(i++);
```

使用巨集的注意事項

- 此程式片段要計算的應是 3 的立方, 並在算完後將 i 遞增。然而實際上會被替換成：

```
(i++)*(i++)*(i++)
```

- 雖然計算結果仍是 3 的立方, 但 i 最後卻遞增了 3 次變成 6。如果是用前置遞增 / 遞減運算子, 則連立方的計算結果都不正確, 因為會變成計算 (++i)*(++i)*(++i), 依 Visual C++ 2005 編譯器的處理方式, 最後計算立方的 3 個 i 都會用 6 代入, 所以變成計算 6 的立方。

使用巨集的注意事項

- 由於使用巨集有這些潛在的問題, 且使用巨集時, 編譯器不會替我們做參數型別的檢查及自動轉換, 這些問題都嚴重侵害到程式本身的安全性。因此, 建議儘量使用行內函式來代替巨集。

6-6 C++ 標準函式庫

- 認識了函式的使用方式後，大家一定會想如果有人預先設計好具有特定功能的函式，那寫程式就輕鬆多了。其實程式語言的設計者早就想到這個問題，每種程式語言都或多或少提供了各種工具函式，讓程式設計人員可直接取用，而不必再重複設計這些常用的函式。

6-6 C++ 標準函式庫

- C++ 語言也提供相當多的工具函式, 它們都分類放在不同的標準函式庫 (function library), 就像萬用的工具箱一樣。使用這些函式時, 只要在程式開頭含括該函式宣告所在的含括檔, 接著就可在程式中呼叫這些函式, 就像我們使用 cin/cout 要先含括 <iostream> 一樣。

數學運算

- 幾乎所有的程式都要進行或簡或繁的數學運算, 因此標準函式庫就提供一些基本的數學計算函式。
  - 次方、平方根、指數、對數
  - 三角函數

次方、平方根、指數、對數

- 次方、平方根、指數、對數常見於各式各樣的科學、金融、統計等計算中，當我們需要用到這些基本的運算時，即可含括 <cmath> 含括檔，使用下列函式進行計算：

```
double exp   (double x)          // 傳回自然對數 e 的 x 次方值
double log   (double x)          // 傳回 x 的對數值 (以自然對數 e 爲基底)
double log10(double x)           // 傳回 x 的對數值 (以 10 爲基底)
double pow   (double x, double y)    // 傳回 x 的 y 次方
double sqrt (double x)           // 傳回 x 的平方根 (√x)
```

次方、平方根、指數、對數

- 使用這些函式時請注意一些數學及 C++ 語言資料型別的限制。舉例來說, 如果 x 為負數, 則 sqrt(x) 將會傳回錯誤的結果, 因為負數開根號會出現虛數, 這不是 double 可表現的資料範圍。同理, 若 x 為負數且 y 有小數點, 則 pow (x,y) 也會傳回錯誤。
- 例如以下就是利用求平方根的函式來計算直角三角形的斜邊長度：

次方、平方根、指數、對數

程式 Ch06-13.cpp 計算直角三角型的斜邊

```
01 #include <iostream>
02 #include <cmath>
03 using namespace std;
04
05 int main()
06 {
07   double a, b;
08
09   cout << "本程式可計算直角三角形的斜邊長" << endl;
10   cout << "請輸入直角三角型第 1 個短邊的邊長：";
11   cin >> a;
12   cout << "請輸入直角三角型第 2 個短邊的邊長：";
```

次方、平方根、指數、對數

```
13    cin >> b;
14    cout << "斜邊長爲：" << sqrt(a*a + b*b);
15 }
```

執行結果

```
本程式可計算直角三角形的斜邊長
請輸入直角三角型第 1 個短邊的邊長：100
請輸入直角三角型第 2 個短邊的邊長：105
斜邊長爲：145
```

三角函數

- <cmath> 含括檔中宣告的數學函式中另一類在各種計算也常用到的就是三角函數的函式, 這些函數包括：

三角函數

```
double sin(double x) // 傳回 x 的正弦函數值
double cos(double x) // 傳回 x 的餘弦函數值
double tan(double x) // 傳回 x 的正切函數值

double asin(double x) // 傳回 x 的反正弦函數值
double acos(double x) // 傳回 x 的反餘弦函數值
double atan(double x) // 傳回 x 的反正切函數值
double atan2(double y,double x) // 傳回 y/x 的反正切函數值

double sinh(double x) // 傳回 x 的雙曲正弦函數值
double cosh(double x) // 傳回 x 的雙曲餘弦函數值
double tanh(double x) // 傳回 x 的雙曲正切函數值
```

三角函數

- 使用這類函式時要注意兩點，首先是前面提過的參數範圍，以反三角函數為例，必需以 -1～1 之間的參數值，才會傳回合理的角度值；此外這些標準函式所使用的角度單位並非一般習慣使用的角度，而是使用**弳度**為單位，也就是以 π (=180 度) 的角度單位來計算。因此為了方便計算，建議要用到這些函式時，先在程式中定義代表 π 或角度 / 弳度換算用的巨集常數，請參考以下範例：

三角函數

程式 Ch06-14.cpp 測試三角函數的計算結果

```
01 #include <iostream>
02 #include <cmath>
03 using namespace std;
04 #define PI    3.141592653589793        // 定義常數 π
05
06 int main()
07 {
08    cout << "角度\tsin()" << endl;
09
10    for(int i=30;i<=180;i+=30) {  // 計算 30、60、90...度的正弦函數值
11       cout << i << '\t';
12       cout << sin(i *PI / 180) << endl;
13    }
14 }
```

三角函數

執行結果

角度	sin()
30	0.5
60	0.866025
90	1
120	0.866025
150	0.5
180	1.22461e-016

三角函數

1. 第 10～13 行以迴圈的方式輸出 30、60、90…至 180 度的 sin() 正弦函數值。
2. 第 12 行使用 PI 常數以便將角度轉成弧度再呼叫 sin() 方法。

- 讀者可發現, 當角度為 180 度 () 時, sin 函數的值應該為 0, 但範例程式算出來的結果只是**非常小**的數值, 但仍不是 0。這是電腦浮點數處理方式的先天限制, 並非 C++ 的算術不好, 讀者若在進行較精密的計算時, 需注意這個問題。

時間函式

- 時間類型的函式也是寫作程式常會用到的, C++ 提供了幾個與時間有關的函式, 例如有一個是由中央處理器的時脈 (clock) 數傳回目前時間的函式：

```
clock_t   clock ( void ); // 傳回以時脈數表示的目前時間
```

- 其中 clock_t 為使用 typedef 定義的型別, 通常就是 long 型別。此外有個巨集常數 CLK_TCK, 表示每秒的時脈數, 因此我們可利用它們來計算程式執行的時間, 或是控制程式等待的時間。

時間函式

程式 Ch06-15.cpp 計算程式執行時間

```
01 #include <iostream>
02 #include <ctime>
03 using namespace std;
04
05 int main()
06 {
07   cout << "執行加法及比較運算一億次需要 ";
08   clock_t starttime = clock();
09   for(int i=0;i<100000000;i++) ;     // 沒有做事的迴圈
10   clock_t endtime = clock();
11
12   cout << (double)(endtime - starttime) / CLK_TCK << " 秒" ;
13 }
```

時間函式

執行結果

執行加法及比較運算一億次需要 0.831 秒

1. 第 9 行的迴圈雖然沒做什麼事, 但運算式每次都會將 i 遞增並做比較, 所以等於做加法及比較運算一億次。
2. 第 12 行將執行迴圈前後的系統時間 (以時脈為單位) 相減, 再除以 CLK_TCK 常數即可得到執行的秒數。由於參與計算的數值都是整數型別, 所以用強迫轉型的方式, 以取得商數的小數點。

6-7 函式多載 (Overloading)

- 函式的參數列中各參數的型別也稱為函式的**簽名** (Signature)。因為在 C++ 的程式中, 函式名稱可以重複, 但同名的函式其簽名必須彼此不同, 每個名稱及簽名的組合必須是唯一而不可重複的。使用相同的名稱來定義不同的函式就稱為**多載** (Overload), 當程式中有多載的函式時, C++ 的編譯器就會依呼叫函式時的參數型別, 來判斷程式到底是呼叫哪一個函式。

函式多載 (Overloading)

- 舉例來說, 剛剛介紹的數學函式, 其實就有 3 種版本：

```
float sqrt (float x)
double sqrt (double x)
long double sqrt (long double x)
```

- 以上三個函式的宣告均為合法, 因為它們的參數列不同; 因此編譯時編器會替函式名稱加上不同的簽名以茲分辨。當我們呼叫函式時, 編譯器也會依照呼叫函式時的參數型別, 來決定要使用哪一個函式。除了參數型別不同外, **參數數量不同**也可以是簽名的依據：

函式多載 (Overloading)

程式 Ch06-16.cpp 多載函式的應用

```
01 #include <iostream>
02 using namespace std;
03
04 void volume (double r) // 計算球體體積
05 {
06    cout << "半徑 " << r << " 的球體體積爲 "
07         << 4 / 3 * 3.14159 * r * r * r << endl;
08 }
09                              // 計算長方體體積
10 void volume (double l, double w, double h)
11 {
```

函式多載 (Overloading)

```
12     cout << "長 " << l << " 寬 " << w << " 高 " << h
13          << " 的長方體體積爲 "<< l * w * h << endl;
14 }
15
16 int main()
17 {
18     volume(15);
19     volume(5,15,25);
20 }
```

執行結果

```
半徑 15 的球體體積爲 10602.9
長 5 寬 15 高 25 的長方體體積爲 1875
```

函式多載 (Overloading)

1. 第 4～8 行為計算並顯示球體體積的函式 volume(), 呼叫時需傳遞球體的半徑為參數。
2. 第 10～14 行為計算並顯示長方體體積的函式 volume(), 呼叫時需傳遞長、寬、高為參數。
3. 第 18、19 行分別以不同數量的參數呼叫 volume() 函式, 結果分別會執行到不同的計算體積函式, 如**執行結果**所示。

函式多載 (Overloading)

- 在設計多載函式時請注意, 函式的簽名是依據參數的型別, 所以只有傳回值型別不同是無法成為多載函式：

```
long   calc(int);
double calc(int);     // 重複宣告函式, 只有傳回值不同不能當多載函式
```

- 多載函式的應用是物件導向程式設計三項特性中**多面性** (Polymorphism) 的一環, 在後面章節進入物件導向的主題後, 將會介紹更多的複載函式應用。

6-8 綜合演練

- 遞迴函式
- 數學函式的應用：求任意次方根
- 解二元一次聯立方程式的巨集

遞迴函式

- 在函式的應用中, 有一種特殊的設計方式稱為遞迴 (Recursive), 意思是指在函式本體中的敘述, 會再呼叫函式本身, 如此一直循環呼叫下去, 因此稱為遞迴。當然為了要避免像無窮迴圈這種程式停不下來的狀況, 遞迴函式也必須設計一個可停止再呼叫自己的狀況, 讓函式能一層層地返回。以下就是一個無法停下來的遞迴函式：

遞迴函式

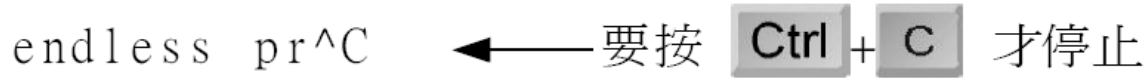
程式 Ch06-17.cpp 無窮的遞迴函式

```
01 #include <iostream>
02
03 int main()
04 {
05     void func();     // 宣告函式
06     func();          // 呼叫函式
07 }
08
09 void  func()        // 遞迴函式
10 {
11     std::cout << "This is a endless program\n";
12     func();      // 呼叫自己
13 }
```

遞迴函式

執行結果

```
This is a endless program
This is a endless program
This is a endless program
This is a endless program
This is a endless pr^C
```



- 如果不按 Ctrl + C 讓程式一直執行下去，幾分鐘後電腦竟然當機了！這是因為程式呼叫函式的層數太深了，而使得堆疊的容量不夠所致。所以在遞迴函式中，必須設定一個條件式來控制呼叫的層數，而非隨心所欲地去呼叫。底下是以遞迴方式設計的求階乘的程式：

遞迴函式

程式 Ch06-18.cpp 計算階乘的遞迴函式

```
01 #include <iostream>
02 using namespace std;
03
04 long double fact(int n)   // 遞迴式函式
05 {
06    if(n == 1)              // 在 n==1 時停止往下遞迴
07      return 1;             // 傳回 1
08    else
09      return ( n * fact(n-1));    // 將參數減 1 再呼叫自己
10 }
11
12 int main()
13 {
```

遞迴函式

```
14      int x;
15      while (true) {
16        cout << "請輸入一小於170的整數(輸入0結束程式)：";
17        cin >> x;
18        if(x == 0) break;          // 輸入 0 時跳出迴圈、結束程式
19        cout << x << "! = " << fact(x) << endl;
20      }
21  }
```

執行結果

```
請輸入一小於170的整數(輸入0結束程式)：5
5! = 120
請輸入一小於170的整數(輸入0結束程式)：55
55! = 1.26964e+073
請輸入一小於170的整數(輸入0結束程式)：0
```

遞迴函式

1. 第 4～10 行為以遞迴方式計算階乘的函式, 當參數 n 的值為 1 時即直接傳回 1 其它狀況則將參數值減 1 再呼叫自己。
2. 第 9 行呼叫自己的動作其實不難理解, 因為 n! 的值就相當於是 (n* (n-1)!), 所以函式就再呼叫自己來計算 (n-1)! 的值；而 (n-1)! 又可看成是 (n-1) 乘上 (n-2)!..., 如此一直呼叫下去, 當呼叫函式的參數值為 1 時, 就會傳回 1, 如此層層傳回, 程式就會得到 n * (n-1) * (n-2) * ... * 2 * 1 的計算結果, 也就是 n! 的值。

遞迴函式

3. 第 15～20 行以 while 迴圈重複執行計算階乘的動作, 只有當使用者輸入 0 時, 才會跳出迴圈結束程式。
4. 第 17 行取得輸入資料後, 即在第 18 行檢查是否為 0, 不是就在第 20 行呼叫 fact() 函式, 並顯示其傳回的階乘值。

數學函式的應用：求任意次方根

- <cmath> 中只有一個 sqrt() 函式讓我們求平方根, 那要求其它次方根怎麼辦？其實不難, 只要換個方式來思考, 開 n 次方根, 就相當於計算該數值的 (1/n) 次方, 所以我們只要倒過來用 pow() 來求 (1/n) 次方即可。我們可以把這個簡單的計算設計成行內函式來使用：

數學函式的應用：求任意次方根

程式 Ch06-19.cpp 求任意數的 N 次方根

```
01 #include <iostream>
02 #include <cmath>
03 using namespace std;
04
05 inline double root(double x, int n) { return pow(x,1.0/n); }
06
07 int main()
08 {
09    int n;
10    double x;
11    while (true)
```

數學函式的應用：求任意次方根

```
12    {
13      cout << "請輸入要求 n 次方根的正實數(輸入0則結束程式)：";
14      cin >> x;
15      cout << "要求幾次方根(限輸入整數)：";
16      cin >> n;
17
18      if(x == 0 || n==0)        // 輸入 0 時跳出迴圈、結束程式
19        break;
20      else if (x < 0)           // 若 x 爲負值, 將其變成正值
21        x *= -1;                 // 也可呼叫 <cmath> 的絕對值函式 abs()
22      cout << x << " 的 " << n << " 次方根爲 " << root(x,n) << endl;
23    }
24  }
```

數學函式的應用：求任意次方根

執行結果

```
請輸入要算開 n 次方根的正實數(輸入 0 則結束程式)：243
要求幾次方根(限輸入整數)：5
243 的 5 次方根爲 3
請輸入要求 n 次方根的正實數(輸入 0 則結束程式)：128
要求幾次方根(限輸入整數)：7
128 的 7 次方根爲 2
請輸入要求 n 次方根的正實數(輸入 0 則結束程式)：0
要求幾次方根(限輸入整數)：0
```

數學函式的應用：求任意次方根

- 第 5 行就是用 pow() 計算開 n 次方根的行內函式。另外為了避免負數開 n 次方可能出現虛數的情況, 程式特別在第 20 行檢查 x 是否為負值, 是就將之轉成正數。

解二元一次聯立方程式的巨集

- 在數學中，二元一次聯立方程式是對兩個具有相同的兩個未知數 (x,y) 的多項式求 x、y 的值，例如：

$$ax + by = c$$
$$dx + ey = f$$

- 其中 a、b、c、d、e、f 是方程式的係數，為已知的常數值，未知數為 x、y。而我們可將此聯立方程式的解寫成如下的公式：

$$x=(c*e-f*b)/(a*e-d*b)$$
$$y=(a*f-d*c)/(a*e-d*b)$$

解二元一次聯立方程式的巨集

- 我們將這兩個公式定義成巨集便可以求出各種聯立方程式的解, 程式如下：

程式 Ch06-20.cpp 解二元一次聯立方程式

```
01 #include <iostream>
02 using namespace std;
03 #define XX (c*e- f*b)/(a*e- d*b)   // 定義解 x 的巨集
04 #define YY (a*f- d*c)/(a*e- d*b)   // 定義解 y 的巨集
05
06 int main()
07 {
08    float a,b,c,d,e,f;                    // 二元一次方程式的係數
```

解二元一次聯立方程式的巨集

```
09    char go = 'y';
10
11    cout << "解聯立方程式" << endl
12         << "ax + by = c" << endl
13         << "dx + ey = f" << endl;
14
15    while (go == 'y' || go =='Y') {
16       cout << "請輸入a的值："; cin >> a;
17       cout << "請輸入b的值："; cin >> b;
18       cout << "請輸入c的值："; cin >> c;
19       cout << "請輸入d的值："; cin >> d;
20       cout << "請輸入e的值："; cin >> e;
21       cout << "請輸入f的值："; cin >> f;
22
```

解二元一次聯立方程式的巨集

```
23        if ((a*e- d*b)== 0)          // 避免分母爲 0
24           continue;
25
26        cout << a << "x + " << b << "y = " << c << endl;
27        cout << d << "x + " << e << "y = " << f << endl;
28        cout << "的解爲 x = " << XX << endl
29             << "          y = " << YY << endl;
30
31       cout << "還要再算嗎？(y/n)：";
32       cin >> go;
33    }
34 }
```

解二元一次聯立方程式的巨集

執行結果

```
解聯立方程式
ax + by = c
dx + ey = f
請輸入 a 的值：15
請輸入 b 的值：30
請輸入 c 的值：0
請輸入 d 的值：2
請輸入 e 的值：8
請輸入 f 的值：20
15x + 30y = 0
2x + 8y = 20
的解爲  x = -10
        y = 5
還要再算嗎？(y/n)：n
```