24

其它主題

What's in a name? that which we call a rose By any other name would smell as sweet.
—William Shakespeare
O Diamond!Diamond! thou little knowest the mischief done!
—Sir Isaac Newton

學習目標

在本章中，你將學到：

- 使用 `const_cast` 將 `const` 物件暫時處理成非 `const` 物件。
- 如何使用 `namespace`。
- 如何使用運算子關鍵字。
- 如何在 `const` 物件中，使用 `mutable` 成員。
- 如何使用類別成員指標運算子 `.*`和`->*`。
- 如何使用多重繼承。
- 在多重繼承中，`virtual` 基本類別的角色。

本章綱要

24.1 簡介

我們現在考慮幾種進階的 C++ 功能。首先，我們將學習 const_cast 運算子的相關細節，這個運算子能讓你增加或移除某個變數的 const 限制。其次，我們將討論 namespace，程式設計者可以利用它來確保在程式中的每一個識別字都是唯一的名稱，而且它能幫助我們解決經由使用函式庫而導致的名稱衝突問題，因爲各函式庫有可能具有相同變數、函式或類別名稱。然後，我們將提出幾個運算子關鍵字，對於其鍵盤不支援某些運算子符號，例如!、&、^、|等符號的程式設計者而言，這些運算子關鍵字將很有用。接著我們又討論 mutable 的儲存空間類別修飾詞，這個修飾詞能讓你指定一個資料成員永遠都可以被修改，即使這個資料成員所屬的物件目前被程式視爲 const 物件。其次，我們會介紹兩個特殊運算子，我們可以將這兩個運算子，與指向類別成員的指標搭配使用，以便在無法預先知道資料成員或成員函式名稱的情形下，還能運用它們。最後，我們將介紹多重繼承，這項功能可以讓一個衍生類別同時繼承幾個基本類別的成員。我們另外還會討論多重繼承的潛在問題，也會討論如何利用 virtual 繼承來解決這些問題。

24.2 const_cast 運算子

C++ 提供 **const_cast** 運算子，用於取消 const 或 volatile 的限定。當程式預期使某個變數經由編譯器所不知道的硬體，或其他程式來加以修改的時候，該程式可以使用 **volatile** 修飾詞來宣告此變數。將變數宣告爲 volatile，意謂著編譯器不應該針

對此變數的使用進行最佳化，因爲這麼做會影響其他會存取和修改此 volatile 變數的程式的能力。

一般而言，因爲 const_cast 運算子允許程式能修改被宣告爲 const 的變數，然而對於 const 變數，我們會預設它是不能修改的，所以使用 const_cast 運算子是有危險的。在一些情況下，取消 const 限定是值得冒險的，甚至是必要的。舉例來說，比較舊的 C 和 C++ 函式庫可能提供具有非 const 參數，而且不能修改其參數的函式。如果我們想要傳遞 const 資料給這樣的函式，我們會需要將資料的 const 限定取消掉；否則編譯器將顯示出錯誤訊息。

同樣地，我們可能會將非 const 資料傳遞給會將資料當作常數來處理，然後將該資料當作常數傳回的函式。在這樣的情形下，我們可能需要取消傳回資料的常數限定，如同我們在圖 24.1 所示範說明的。

```
1  // Fig. 24.1: fig24_01.cpp
2  // Demonstrating const_cast.
3  #include <iostream>
4  #include <cstring> // contains prototypes for functions strcmp and strlen
5  #include <cctype> // contains prototype for function toupper
6  using namespace std;
7
8  // returns the larger of two C-style strings
9  const char *maximum( const char *first, const char *second )
10 {
11    return ( strcmp( first, second ) >= 0 ? first : second );
12 } // end function maximum
13
14 int main()
15 {
16    char s1[] = "hello"; // modifiable array of characters
17    char s2[] = "goodbye"; // modifiable array of characters
18
19    // const_cast required to allow the const char * returned by maximum
20    // to be assigned to the char * variable maxPtr
21    char *maxPtr = const_cast< char * >( maximum( s1, s2 ) );
22
23    cout << "The larger string is: " << maxPtr << endl;
24
25    for ( size_t i = 0; i < strlen( maxPtr ); i++ )
26       maxPtr[ i ] = toupper( maxPtr[ i ] );
27
28    cout << "The larger string capitalized is: " << maxPtr << endl;
29 } // end main
```

```
The larger string is: hello
The larger string capitalized is: HELLO
```

圖 24.1　示範說明運算子 const_cast

在這個程式中，函式 maximum (第 9–12 行) 接收了兩個 C 風格字串的 const char* 參數，並且傳回一個 const char*，指向上述兩個字串中比較大的字串。函式 main 將兩個 C 風格字串宣告成非 const char 陣列 (第 16–17 行)；因此，這些陣列是可以修改的。在函式 main 中，我們想要輸出兩個 C 風格字串中比較大的字串，然後經由將它轉換成大寫字母，來修改此 C 風格字串。

函式 maximum 的兩個參數是 const char* 型別，所以函式傳回的型別也必須宣告爲 const char* 型別。如果傳回的型別只指定爲 char*，則編譯器會發出錯誤訊息，指出正要傳回的數值無法從 const char* 轉換成 char*；此時編譯器將函式視爲 const 的資料，當作非 const 的資料來處理，所以這是一個危險的轉換。

即使函式 maximum 相信資料是常數，但是我們知道在 main 中的原始陣列並不包含常數資料。所以，在需要的時候，main 應該要能修正那些陣列的內容。既然我們知道這些陣列是可以修改的，我們使用 const_cast (第 21 行) 取消 maximum 傳回指標的 const 限定，然後，我們便可以修改代表兩個 C 風格字串中比較大的字串的陣列資料。然後我們可以使用 for 迴圈 (第 25–26 行) 中字元陣列名稱的指標，將比較大字串的內容轉換成大寫。因爲 C++不允許我們將型別 const char* 的指標，指定給 char* 型別的指標，所以如果沒有第 21 行的 const_cast，這個程式將無法編譯。

測試和除錯的小技巧 24.1

一般而言，只有在我們已經預先知道原始資料不是常數的時候，我們才能使用 const_cast。否則，將發生無法預期的結果。

24.3 可變類別成員 (mutable Class Members)

在第 24.2 節中，我們介紹了 const_cast 運算子，此運算子允許我們解除一個型別的「const 限定」。對於 const 物件的資料成員而言，當此物件被用於該物件所屬類別的 const 成員函式主體時，const_cast 操作也可以施加於此資料成員。這樣做將允許 const 成員函式能修改資料成員，即使資料成員所屬物件在此函式主體中被視爲 const 也是如此。當一個物件的大部分資料成員應該視爲 const，但其中一個特別的資料成員仍然需要加以修改的時候，我們就可以進行這樣的操作。

這裡舉一個例子，我們考慮一個內容會經過排序的鏈結串列。對此鏈結串列進行搜尋，並不需要修改其內的資料，所以搜尋函式可以是此鏈結串列所屬類別的 const 成員函式。然而，我們可以想像得到，如果要讓未來的搜尋更有效率，一個鏈結串列應該

要能紀錄上一個成功比對的位置。如果下一次搜尋操作企圖找出出現在串列中比較後面的項目，則搜尋動作可以從上一次成功比對的位置開始執行，而不需要從串列的頭端重新開始。爲了達成這個功能，執行搜尋工作的 const 成員函式，必須要能記錄上一次成功比對的資料成員。

如果如上述這樣的資料成員需要永遠能進行修改，則 C++ 提供了儲存類別修飾字 **mutable**，它可以作爲 const_cast 的替代操作方式。mutable 資料成員隨時可以進行修改，即使在 const 成員函式或 const 物件中也是如此。這樣作可以減少對於取消 const 限定的需求。

可攜性的小技巧 24.1

嘗試對定義成常數的物件進行修改所造成的影響，會隨著編譯器的種類而有所不同，不論這樣的修改是透過 const_cast 或 C 風格的強制轉換都是如此。

mutable 和 const_cast 兩者都允許資料成員進行修改；不過它們用於不同的程式狀況中。對於不含 mutable 資料成員的 const 物件而言，每一次要修改其成員時，程式都必須使用運算子 const_cast。因爲這種成員並非隨時都能加以修改，所以這會大幅降低意外修改成員的可能性。涉及 const_cast 的操作，通常會隱藏在成員函式的運作中。類別的使用者可能無法察覺到該成員正在被修改。

軟體工程的觀點 24.1

在擁有「機密」實作細節的類別中，如果它不會影響客戶端使用物件，則使用 mutable 成員是很有用的。

mutable 資料成員使用技巧的說明

圖 24.2 將示範說明 mutable 成員的用法。此程式定義了類別 TestMutable (第 7–21 行)，它包含一個建構子，函式 getValue，以及一個宣告成 mutable 的 private 資料成員 value。程式第 15–18 行將函式 getValue 定義成能傳回 value 副本的 const 成員函式。請注意，在 return 敘述中，此函式會使 mutable 資料成員 value 遞增 1。一般而言，const 成員函式是無法修改資料成員的，除非此函式所操作的物件 (也就是 this 指標所指向的物件) 使用 const_cast 轉換成非 const 型別。然而因爲 value 是 mutable，所以這個 const 函式能夠修改資料。

```
1  // Fig. 24.2: fig24_02.cpp
2  // Demonstrating storage-class specifier mutable.
3  #include <iostream>
4  using namespace std;
5
6  // class TestMutable definition
7  class TestMutable
8  {
9  public:
10    TestMutable( int v = 0 )
11    {
12       value = v;
13    } // end TestMutable constructor
14
15    int getValue() const
16    {
17       return value++; // increments value
18    } // end function getValue
19 private:
20    mutable int value; // mutable member
21 }; // end class TestMutable
22
23 int main()
24 {
25    const TestMutable test( 99 );
26
27    cout << "Initial value: " << test.getValue();
28    cout << "\nModified value: " << test.getValue() << endl;
29 } // end main
```

```
Initial value: 99
Modified value: 100
```

圖 24.2　示範說明 mutable 資料成員的使用方法

程式第 25 行宣告了 const TestMutable 物件 test，並且將其初始值設定成 99。第 27 行將呼叫 const 成員函式 getValue，此函式會對 value 加一，並且傳回其原先的數值。請注意，編譯器允許針對物件 test，呼叫成員函式 getValue，這是因爲 test 是 const 物件，而且 getValue 是一個 const 成員函式的緣故。然而，getValue 卻修改了變數 value。因此，當程式第 28 行再一次呼叫 getValue 的時候，程式會輸出新的 value (100)，以便證明 mutable 資料成員確實已經被修改。

24.4 命名空間

程式包含許多定義於不同使用域的識別字。有時候，定義在某個使用域的變數會和定義在不同使用域，但是具有相同名稱的變數「重疊」(也就是發生使用域的抵觸)，因而可

能產生名稱上的衝突。這種重疊現象會出現在許多層次上。識別字重疊的現象，經常發生在第三方提供的函式庫中，這些函式庫恰好以相同的名稱作爲全域識別字 (例如函式)。這可能導致編譯時期的錯誤。

良好的程式設計習慣 24.1

避免使用以底線字元作爲起始字元的識別字，這樣的識別字可能導致連結時期的錯誤。許多程式碼函式庫 (code libraries) 使用以底線字元作爲起始字元的名稱。

C++ 規範嘗試使用**命名空間 (namespace)** 來解決這個問題。每一個 namespace 會定義放置識別字和變數的使用域。若要使用 **namespace 成員 (namespace member)**，其用法有兩種，其中一種，必須將成員名稱使用 namespace 名稱以及二元使用域解析運算子 (::) 加以修飾，如下所示

```
MyNameSpace::member
```

另一種用法爲，在程式中使用某個名稱以前，必須先寫出相關的 using 宣告或 using 指令。一般來說，這樣的 using 敘述會放置在 namespace 成員所使用的檔案開頭部分。舉例來說，將下列 using 指令放在原始碼檔案的開頭

```
using namespace MyNameSpace;
```

指的是 namespace MyNameSpace 的成員可以在不需要在每一個成員前面放置 MyNameSpace 和使用域解析運算子 (::) 的情形下，於檔案中使用它。

using 宣告 (例如，using std::cout;) 可以將一個名稱帶進宣告出現的使用域中。using 指令 (例如，using namespace std;) 則會將來自被標示的命名空間中所有的名稱，都帶進該指令出現的使用域中。

軟體工程的觀點 24.2

理想上，在大型程式中，每個實體 (entity) 都應該宣告在類別、函式、程式區塊或 namespace 中。這可以幫助釐清每一個實體所扮演的角色。

測試和除錯的小技巧 24.2

如果存在名稱衝突的可能性，則必須在成員的前面加上它的 namespace 名稱和使用域解析運算子 (::)。

並非所有的 namespace 名稱都保證是唯一的。兩個不相關的程式設計廠商有可能會不慎替他們的 namespace 名稱取相同的識別字。圖 24.3 將示範說明 namespace 的使用方法。

```
1  // Fig. 24.3: fig24_03.cpp
2  // Demonstrating namespaces.
3  #include <iostream>
4  using namespace std;
5
6  int integer1 = 98; // global variable
7
8  // create namespace Example
9  namespace Example
10 {
11    // declare two constants and one variable
12    const double PI = 3.14159;
13    const double E = 2.71828;
14    int integer1 = 8;
15
16    void printValues(); // prototype
17
18    // nested namespace
19    namespace Inner
20    {
21       // define enumeration
22       enum Years { FISCAL1 = 1990, FISCAL2, FISCAL3 };
23    } // end Inner namespace
24 } // end Example namespace
25
26 // create unnamed namespace
27 namespace
28 {
29    double doubleInUnnamed = 88.22; // declare variable
30 } // end unnamed namespace
31
32 int main()
33 {
34    // output value doubleInUnnamed of unnamed namespace
35    cout << "doubleInUnnamed = " << doubleInUnnamed;
36
37    // output global variable
38    cout << "\n(global) integer1 = " << integer1;
39
40    // output values of Example namespace
41    cout << "\nPI = " << Example::PI << "\nE = " << Example::E
42       << "\ninteger1 = " << Example::integer1 << "\nFISCAL3 = "
43       << Example::Inner::FISCAL3 << endl;
44
45    Example::printValues(); // invoke printValues function
46 } // end main
47
48 // display variable and constant values
49 void Example::printValues()
50 {
51    cout << "\nIn printValues:\ninteger1 = " << integer1 << "\nPI = "
52       << PI << "\nE = " << E << "\ndoubleInUnnamed = "
```

圖 24.3 namespaces 使用方法的示範說明

```
53          << doubleInUnnamed << "\n(global) integer1 = " << ::integer1
54          << "\nFISCAL3 = " << Inner::FISCAL3 << endl;
55    } // end printValues
```

```
doubleInUnnamed = 88.22
(global) integer1 = 98
PI = 3.14159
E = 2.71828
integer1 = 8
FISCAL3 = 1992

In printValues:
integer1 = 8
PI = 3.14159
E = 2.71828
doubleInUnnamed = 88.22
(global) integer1 = 98
FISCAL3 = 1992
```

圖 24.3　namespaces 使用方法的示範說明 (續)

定義命名空間

第 9-24 行使用關鍵字 namespace 來定義命名空間 Example。namespace 的本體是由左大括弧和右大括弧 ({}) 圈圍起來。命名空間 Example 的成員包含兩個常數 (第 12-13 行的 PI 和 E)，一個 int (第 14 行的 integer1)，一個函式 (第 16 行的 printValues) 和一個**巢狀命名空間 (nested namespace)** (第 19-23 行的 Inner)。請注意，成員 integer1 與全域變數 integer1 (第 6 行) 具有相同名稱。具有相同名稱的變數必須具有不同的使用域，否則就會產生語法錯誤。namespace 可以包含常數、資料、類別、巢狀 namespace、函式等等。namespace 的定義必須擁有全域使用域，或巢狀安置在其它 namespace 內。跟類別不一樣的是，不同的命名空間成員可以定義在個別的命名空間區塊中，每一個標準函式庫標頭檔都有一個命名空間區塊，將它的內容放在 namespace std。

程式第 27-30 行建立了一個**不具名的 namespace (unnamed namespace)**，它包含成員 doubleInUnnamed。不具名 namespace 中的變數、類別和函式只能在目前的**編譯單元 (translation unit)** 中被存取 (a.cpp 檔以及它 include 的檔案)。然而，不具名 namespace 的變數、類別或函式與靜態連結的不同，它們可能被用來當作樣板引數。不具名的 namespace 具有隱喻的 using 指令，所以其成員擁有**全域 namespace (global namespace)**，能夠直接存取，而且不需要使用 namespace 名稱來加以限定。全域變數 (Global variables) 也是全域 namespace 的一部分，程式可以在檔案中宣告位置之後的所有使用域使用它們。

軟體工程的觀點 24.3

每一個個別的編譯單位都有自己獨一無二的不具名 namespace；也就是說，不具名 namespace 取代了 static 連結修飾子。

以修飾名稱存取命名空間的成員

程式第 35 行會輸出變數 doubleInUnnamed 的值，此變數是當作不具名命名空間的一部分來直接存取。第 38 行會輸出全域變數 integer1 的值。針對這兩個變數，編譯器會先試圖找出它們在 main 函式中區域性宣告的位置。因爲沒有任何區域性宣告，所以編譯器會假設這些變數位在全域性命名空間中。

程式第 41-43 行輸出了位於命名空間 Example 的 PI、E、integer1 和 FISCAL3 的值。請注意，所以前述每一個成員都必須 Example::修飾。因爲這個程式並沒有提供任何 using 指令或宣告，來指出程式將使用命名空間 Example 的成員，除此之外，因爲有另一個全域變數具有相同名稱，所以成員 integer1 必須加上修飾詞。否則，程式將會輸出全域變數的值。請注意，因爲 FISCAL3 是巢狀命名空間 Inner 的成員，所以它必須以 Example::Inner::加以修飾。

函式 printValues (定義於第 49–55 行) 是 Example 的成員，所以它可以在不使用命名空間修飾的情形下，直接存取 Example 命名空間的成員。第 51–54 行的輸出敘述將輸出 integer1、PI、E、doubleInUnnamed、全域變數 integer1 和 FISCAL3。請注意，PI 和 E 並沒有以 Example 修飾。因爲變數 doubleInUnnamed 是位於未命名命名空間中，而且此變數的名稱沒有與命名空間 Example 的任何其他成員相衝突，所以變數 doubleInUnnamed 仍然是可存取的。而 integer1 的全域變數版本必須加上一元使用域解析運算子 (::) 作爲修飾，是因爲其名稱與命名空間 Example 的一個成員相衝突的緣故。此外，FISCAL3 必須以 Inner::加以修飾。在存取巢狀 namespace 的成員時，這些成員必須加上 namespace 名稱 (除非成員當時是在巢狀 namespace 內部中被使用) 加以修飾和限定。

常見的程式設計錯誤 24.1

將 main 放在 namespace 中，是一種編譯時期的錯誤。

Namespace 名稱的別名

命名空間也可以具有別名。例如，以下的敘述

```
namespace CPPHTP = CPlusPlusHowToProgram;
```

將建立 `CPlusPlusHowToProgram` 的**命名空間別名 (namespace alias)** `CPPHTP`。

24.5　運算子關鍵字

C++ 標準提供了**運算子關鍵字** (**operator keywords**，圖 24.4)，它們可以用來取代幾種 C++ 運算子。對於其鍵盤不支援某些像 !、&、^、"|、等等這類字元的程式設計者而言，運算子關鍵字是很有用的。

運算子	運算子關鍵字	說明
邏輯運算子關鍵字		
&&	and	邏輯 AND
||	or	邏輯 OR
!	not	邏輯 NOT
不等式運算子關鍵字		
!=	not_eq	不等式
逐位元運算子關鍵字		
&	bitand	逐位元 AND
|	bitor	逐位元包含 OR
^	xor	逐位元互斥 OR
~	compl	逐位元補數
逐位元指定運算子關鍵字		
&=	and_eq	逐位元 AND 指定
|=	or_eq	逐位元包含 OR 指定
^=	xor_eq	逐位元互斥 OR 指定

圖 24.4　可以替代運算子符號的運算子關鍵字

圖 24.5 將示範說明運算子關鍵字。微軟 Microsoft Visual C++ 2008 編譯器必須先含入標頭檔 `<ciso646>` (第 4 行) 以便使用運算子關鍵字。在 GNU C++ 中，這個標頭檔是空的，因爲運算子關鍵字是永久定義的。

```
1  // Fig. 24.5: fig24_05.cpp
2  // Demonstrating operator keywords.
3  #include <iostream>
4  #include <ciso646> // enables operator keywords in Microsoft Visual C++
5  using namespace std;
6
7  int main()
8  {
9     bool a = true;
10    bool b = false;
11    int c = 2;
12    int d = 3;
13
14    // sticky setting that causes bool values to display as true or false
15    cout << boolalpha;
16
17    cout << "a = " << a << "; b = " << b
18       << "; c = " << c << "; d = " << d;
19
20    cout << "\n\nLogical operator keywords:";
21    cout << "\n   a and a: " << ( a and a );
22    cout << "\n   a and b: " << ( a and b );
23    cout << "\n    a or a: " << ( a or a );
24    cout << "\n    a or b: " << ( a or b );
25    cout << "\n     not a: " << ( not a );
26    cout << "\n     not b: " << ( not b );
27    cout << "\na not_eq b: " << ( a not_eq b );
28
29    cout << "\n\nBitwise operator keywords:";
30    cout << "\nc bitand d: " << ( c bitand d );
31    cout << "\nc bit_or d: " << ( c bitor d );
32    cout << "\n   c xor d: " << ( c xor d );
33    cout << "\n   compl c: " << ( compl c );
34    cout << "\nc and_eq d: " << ( c and_eq d );
35    cout << "\n c or_eq d: " << ( c or_eq d );
36    cout << "\nc xor_eq d: " << ( c xor_eq d ) << endl;
37 } // end main
```

```
a = true; b = false; c = 2; d = 3

Logical operator keywords:
   a and a: true
   a and b: false
    a or a: true
    a or b: true
     not a: false
     not b: true
a not_eq b: true
```

圖 24.5　運算子關鍵字的示範說明

```
Bitwise operator keywords:
c bitand d: 2
c bit_or d: 3
   c xor d: 1
   compl c: -3
c and_eq d: 2
 c or_eq d: 3
c xor_eq d: 0
```

圖 24.5　　運算子關鍵字的示範說明 (續)

這個程式宣告並且初始化了兩個 `bool` 變數，以及兩個整數變數 (第 9–12 行)。其中的邏輯運算 (第 21–27 行) 針對 `bool` 變數 `a` 和 `b`，利用各種不同的邏輯運算子關鍵字加以執行。其中的逐位元運算 (第 30–36 行) 則針對 `bool` 變數 `c` 和 `d`，利用各種不同的逐位元運算子關鍵字加以執行。程式會輸出每一個運算的結果。

24.6　指向類別成員的指標 (.* 和 ->*)

C++ 提供 `.*` 和 `->*` 運算子，用於透過指標來存取類別成員。這是很少使用的功能，通常只有進階的 C++ 程式設計者會用到它。這裡我們只提供一個使用指標指向類別成員的範例。圖 24.6 的程式用於示範說明指向類別成員指標的運算子。

```
1  // Fig. 24.6: fig24_06.cpp
2  // Demonstrating operators .* and ->*.
3  #include <iostream>
4  using namespace std;
5
6  // class Test definition
7  class Test
8  {
9  public:
10    void func()
11    {
12       cout << "In func\n";
13    } // end function func
14
15    int value; // public data member
16 }; // end class Test
17
18 void arrowStar( Test * ); // prototype
19 void dotStar( Test * ); // prototype
20
21 int main()
22 {
```

圖 24.6　　示範說明 `.*` 和 `->*` 運算子

```
23     Test test;
24     test.value = 8; // assign value 8
25     arrowStar( &test ); // pass address to arrowStar
26     dotStar( &test ); // pass address to dotStar
27  } // end main
28
29  // access member function of Test object using ->*
30  void arrowStar( Test *testPtr )
31  {
32     void ( Test::*memberPtr )() = &Test::func; // declare function pointer
33     ( testPtr->*memberPtr )(); // invoke function indirectly
34  } // end arrowStar
35
36  // access members of Test object data member using .*
37  void dotStar( Test *testPtr2 )
38  {
39     int Test::*vPtr = &Test::value; // declare pointer
40     cout << ( *testPtr2 ).*vPtr << endl; // access value
41  } // end dotStar
```

```
In test function
8
```

圖 24.6 示範說明 .* 和 ->* 運算子 (續)

這個程式宣告了類別 Test (第 7–16 行)，它提供 public 成員函式 test 和 public 資料成員 value。程式第 18–19 行提供了函式 arrowStar (定義於第 30–34 行) 和函式 dotStar (定義於第 37–41 行) 的原型，它們分別用來示範說明 ->* 和 .* 運算子的用法。第 23 行建立了物件 test，第 24 行將資料成員 value 設定成 8。程式第 25–26 行呼叫函式 arrowStar 和 dotStar，並且將物件 test 的位址傳遞給這兩個函式。

函式 arrowStar 的第 32 行將變數 memPtr 宣告成一個指向成員函式的指標，並且對它進行初始化的工作。在這個宣告中，Test::* 意指變數 memPtr 爲一個指向類別 Test 的成員的指標。要宣告指向函式的指標，則應該讓*位於指標的名稱之前，然後再用小括弧圈圍起來，如 (Test::*memPtr)。指向函式的指標必須標示出其指向的函式的傳回型別，以及該函式的參數列，如同其型別的一部分。函式的傳回型別標示在指標宣告的左小括弧的左側，而參數列則標示在指標宣告右側的一組獨立小括弧中。在這個例子中，函式的傳回型別是 void，而且函式不具參數。指標 memPtr 以類別 Test 的成員函式 test 的位址，加以初始化。函式的標頭必須與函式指標的宣告相匹配；也就是，函式 test 必須具有 void 傳回型別，並且沒有參數。請注意，指定運算子的右側使用了取址運算子 (&) 來取得成員函式 test 的位址。另外也請注意，第 32 行指定運算子的左側和右側都沒有參照類別 Test 的特定物件。只有類別名稱會與二元使用域解

析運算子 (::) 一起配合使用。程式第 33 行使用 ->* 運算子，呼叫儲存在 memPtr (也就是 test) 內的成員函式。因爲 memPtr 是一個指向類別成員的指標，所以必須使用 ->* 運算子而不是 -> 運算子來呼叫函式。

程式第 39 行將 vPtr 宣告成指向 Test 類別的 int 資料成員的指標。並且對它進行初始化。賦值敘述的右邊指出資料成員 value 的位址。第 40 行將指標 testPtr2 予以解參照，然後使用 .* 運算子來存取 vPtr 所指向的成員。用戶程式碼只可以替用戶程式碼能夠存取的類別成員，建立指向類別成員的指標。在這個範例中，成員函式 test 和資料成員 value 兩者都是可以公開存取的。

常見的程式設計錯誤 24.2

在宣告成員函式指標時，沒有將指標名稱放在小括弧內，是一種語法錯誤。

常見的程式設計錯誤 24.3

在宣告成員函式指標的時候，如果沒有在指標名稱之前放置一個類別名稱，然後於類別名稱之後加上使用域解析運算子 (::)，將是一種語法錯誤。

常見的程式設計錯誤 24.4

企圖將 -> 或 * 運算子與指向類別成員的指標一起搭配使用，會產生語法錯誤。

24.7　多重繼承

在第 12 和 13 章中，我們探討過單一繼承，在這類繼承方式中，每一個類別只能由恰好一個基本類別衍生出來。在 C++ 中，一個類別可以從超過一個以上的基本類別衍生出來，這是一種稱爲**多重繼承 (multiple inheritance)** 的技術，使用這種技術的時候，衍生類別可以繼承兩個或兩個以上基本類別的成員。這種強大的功能激勵了各種軟體再利用方式的產生，但是它可能會造成各種歧義的問題。多重繼承是一種不易學習的概念，應該只適合有經驗的程式設計者使用它。事實上，與多重繼承糾結在一起的問題，是相當不易捉摸的，因此使得像 Java 和 C# 這類比較新的程式設計語言，都不允許類別衍生自一個以上的基本類別。

良好的程式設計習慣 24.2

如果使用得宜，多重繼承是很強大的功能。當新類別和兩個或兩個以上現存類別之間，存在著「是一種 (is a)」關係時 (也就是說，型別 A「是一種」型別 B，而且型別 A「是一種」型別 C)，程式應該使用多重繼承。

軟體工程的觀點 24.4

多重繼承可能導致系統變得複雜。在設計系統時，我們需要更謹慎地使用多重繼承；當單一繼承就足以完成工作時，程式不應該使用多重繼承。

多重繼承的一個常見問題是，每一個基本類別可能含有名稱相同的資料成員或成員函式。當我們要進行編譯的時候，這會導致歧義的問題。讓我們考慮多重繼承的範例 (圖 24.7、圖 24.8、圖 24.9、圖 24.10 和圖 24.11)。類別 `Base1` (圖 24.7) 含有一個 `protected int` 資料成員 `value` (第 20 行)，一個用來設定 `value` 的建構子 (第 10-13 行)，以及 `public` 成員函式 `getData` (第 15-18 行)，其中 `getData` 將會傳回 `value`。

```
1  // Fig. 24.7: Base1.h
2  // Definition of class Base1
3  #ifndef BASE1_H
4  #define BASE1_H
5
6  // class Base1 definition
7  class Base1
8  {
9  public:
10    Base1( int parameterValue )
11    {
12       value = parameterValue;
13    } // end Base1 constructor
14
15    int getData() const
16    {
17       return value;
18    } // end function getData
19 protected: // accessible to derived classes
20    int value; // inherited by derived class
21 }; // end class Base1
22
23 #endif // BASE1_H
```

圖 24.7　示範說明多重繼承—`Base1.h`

除了類別 `Base2` 的 `protected` 資料名稱爲 `letter` 而且型別爲 `char` (第 20 行) 以外，類別 `Base2` (圖 24.8) 與類別 `Base1` 相似。就像類別 `Base1` 一樣，`Base2` 也有 `public` 成員函式 `getData`，但是這個函式傳回的是 `char` 資料成員 `letter` 的值。

```
1  // Fig. 24.8: Base2.h
2  // Definition of class Base2
3  #ifndef BASE2_H
4  #define BASE2_H
5
6  // class Base2 definition
7  class Base2
8  {
9  public:
10    Base2( char characterData )
11    {
12       letter = characterData;
13    } // end Base2 constructor
14
15    char getData() const
16    {
17       return letter;
18    } // end function getData
19 protected: // accessible to derived classes
20    char letter; // inherited by derived class
21 }; // end class Base2
22
23 #endif // BASE2_H
```

圖 24.8　示範說明多重繼承—Base2.h

類別 Derived (圖 24.9-24.10) 是透過多重繼承的方式，由類別 Base1 和類別 Base2 繼承而來。類別 Derived 含有型別爲 double 的 private 資料成員 real (第 20 行)，能初始化類別 Derived 所有資料的建構子，以及 public 成員函式 getReal，其中函式 getReal 會傳回 double 變數 real 的值。

我們在 class Derived 後面的冒號 (:) 之後 (第 13 行)，寫出以逗號爲分隔的基本類別，來指明多重繼承的關係。在圖 24.10 中，我們應該注意建構子 Derived 使用成員初始值語法 (第 9 行)，明確地替它的每一個基本類別 Base1 和 Base2，呼叫基本類別建構子。基本類別建構子的呼叫是按照繼承關係標示的順序來進行，而不是按照其建構子寫出的順序來進行；此外，如果基本類別建構子沒有在成員初始值列表中明確地予以呼叫，則程式將隱喻地呼叫它們的預設建構子。

```
1  // Fig. 24.9: Derived.h
2  // Definition of class Derived which inherits
3  // multiple base classes (Base1 and Base2).
4  #ifndef DERIVED_H
5  #define DERIVED_H
6
7  #include <iostream>
```

圖 24.9　示範說明多重繼承—Derived.h

```
 8  #include "Base1.h"
 9  #include "Base2.h"
10  using namespace std;
11
12  // class Derived definition
13  class Derived : public Base1, public Base2
14  {
15     friend ostream &operator<<( ostream &, const Derived & );
16  public:
17     Derived( int, char, double );
18     double getReal() const;
19  private:
20     double real; // derived class's private data
21  }; // end class Derived
22
23  #endif // DERIVED_H
```

圖 24.9 示範說明多重繼承—Derived.h (續)

```
 1  // Fig. 24.10: Derived.cpp
 2  // Member-function definitions for class Derived
 3  #include "Derived.h"
 4
 5  // constructor for Derived calls constructors for
 6  // class Base1 and class Base2.
 7  // use member initializers to call base-class constructors
 8  Derived::Derived( int integer, char character, double double1 )
 9     : Base1( integer ), Base2( character ), real( double1 ) { }
10
11  // return real
12  double Derived::getReal() const
13  {
14     return real;
15  } // end function getReal
16
17  // display all data members of Derived
18  ostream &operator<<( ostream &output, const Derived &derived )
19  {
20     output << "    Integer: " << derived.value << "\n  Character: "
21          << derived.letter << "\nReal number: " << derived.real;
22     return output; // enables cascaded calls
23  } // end operator<<
```

圖 24.10 示範說明多重繼承—Derived.h

多載的串流插入運算子 (圖 24.10 第 18-23 行) 使用其第二個引數，顯示 Derived 物件的資料，其中第二個引數是指向 Derived 物件的參照。這個運算子函式是 Derived 的 friend 函式，所以 operator<<可以直接存取類別 Derived 的 protected 和 private 成員，其中包括 protected 資料成員 value (繼承自類別 Base1)，protected

資料成員 letter (繼承自類別 Base2)，以及 private 資料成員 real (宣告於類別 Derived 中)。

現在讓我們來檢視 main 函式 (圖 24.11)，此函式用來測試圖 24.7-24.10 中的類別。程式第 11 行建立了 Base1 物件 base1，並且將它初始化成 int 數值 10，然後建立指標 base1Ptr，並且將它初始化成空指標 (也就是 0)。第 12 行建立 Base2 物件 base2，並且將它初始化成 char 數值 'Z'，然後建立指標 base2Ptr，並且將它初始化成空指標。程式第 13 行將建立 Derived 物件 derived，經過初始化以後，它含有 int 數值 7，char 數值 'A'，以及 double 數值 3.5。

```
1  // Fig. 24.11: fig24_11.cpp
2  // Driver for multiple-inheritance example.
3  #include <iostream>
4  #include "Base1.h"
5  #include "Base2.h"
6  #include "Derived.h"
7  using namespace std;
8
9  int main()
10 {
11    Base1 base1( 10 ), *base1Ptr = 0; // create Base1 object
12    Base2 base2( 'Z' ), *base2Ptr = 0; // create Base2 object
13    Derived derived( 7, 'A', 3.5 ); // create Derived object
14
15    // print data members of base-class objects
16    cout << "Object base1 contains integer " << base1.getData()
17       << "\nObject base2 contains character " << base2.getData()
18       << "\nObject derived contains:\n" << derived << "\n\n";
19
20    // print data members of derived-class object
21    // scope resolution operator resolves getData ambiguity
22    cout << "Data members of Derived can be accessed individually:"
23       << "\n    Integer: " << derived.Base1::getData()
24       << "\n  Character: " << derived.Base2::getData()
25       << "\nReal number: " << derived.getReal() << "\n\n";
26    cout << "Derived can be treated as an object of either base class:\n";
27
28    // treat Derived as a Base1 object
29    base1Ptr = &derived;
30    cout << "base1Ptr->getData() yields " << base1Ptr->getData() << '\n';
31
32    // treat Derived as a Base2 object
33    base2Ptr = &derived;
34    cout << "base2Ptr->getData() yields " << base2Ptr->getData() << endl;
35 } // end main
```

圖 24.11　示範說明多重繼承

```
Object base1 contains integer 10
Object base2 contains character Z
Object derived contains:
    Integer: 7
  Character: A
Real number: 3.5

Data members of Derived can be accessed individually:
    Integer: 7
  Character: A
Real number: 3.5

Derived can be treated as an object of either base class:
base1Ptr->getData() yields 7
base2Ptr->getData() yields A
```

圖 24.11 示範說明多重繼承 (續)

程式第 16–18 行用於顯示每一個物件的資料數值。針對物件 base1 和 base2，我們呼叫每個物件的 getData 成員函式。即使這個範例有兩個 getData 函式，呼叫的執行過程也不會有任何歧義的情形。在第 16 行中，編譯器知道 base1 是類別 Base1 的物件，所以呼叫的是類別 Base1 的 getData。在第 17 行中，編譯器知道 base2 是類別 Base2 的物件，所以呼叫的是類別 Base2 的 getData。第 18 行則利用多載的串流插入運算子，來顯示物件 derived 的內容。

解決當衍生類別從多個基本類別繼承了名稱相同的成員函式時，所產生的歧義問題

程式第 22–25 行利用類別 Derived 的成員函式 get，再一次輸出物件 derived 的內容。然而，這裡有一個歧義的問題，因爲這個物件含有兩個 getData 函式，其中一個繼承自類別 Base1，另一個則繼承自類別 Base2。透過二元使用域解析運算子，可以輕易解決這個問題。運算式 derived.Base1::getData() 取得的是繼承自類別 Base1 的變數值 (也就是 int 變數 value)，而 derived.Base2::getData() 取得的是繼承自類別 Base2 的變數值 (也就是 char 變數 letter)。在 real 中的 double 數值則利用呼叫 derived.getReal()，毫無歧義地列印出來；在繼承階層中，並沒有具有該名稱的其他成員函式。

示範說明在多重繼承中的「是一種」關係

單一繼承的是一種 (is-a) 關係也適用於多重繼承的關係。爲了示範說明這一點，程式第 29 行將物件 derived 的位址，指定給 Base1 指標 base1Ptr。因爲類別 Derived 的

物件是一種類別 Base1 的物件，所以這是允許的。程式第 30 行透過 base1Ptr 呼叫 Base1 成員函式 getData，只取得物件 derived 的 Base1 部分的數值。第 33 行將物件 derived 的位址指定給 Base2 指標 base2Ptr。因爲類別 Derived 的物件是一種類別 Base2 的物件，所以這是允許的。第 34 行透過 base2Ptr 呼叫 Base2 成員函式 getData，只取得物件 derived 的 Base2 部分的數值。

24.8　多重繼承和 virtual 基本類別

在第 24.7 節中，我們討論過多重繼承，透過此機制，一個類別能繼承兩個或更多個類別。例如，C++ 標準函式庫使用多重繼承來組成類別 basic_iostream (圖 24.12)。

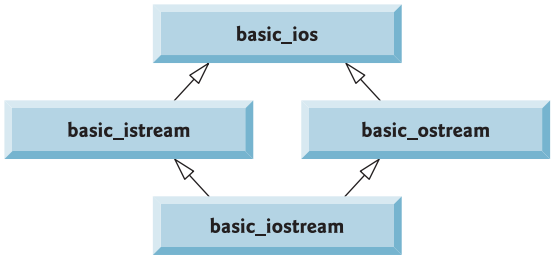

圖 24.12　形成類別 basic_iostream 的多重繼承

類別 basic_ios 是 basic_istream 和 basic_ostream 兩者的基本類別，這兩個衍生類別都是利用單一繼承的方式產生。類別 basic_iostream 則是繼承自 basic_istream 和 basic_ostream 兩者。這允許類別 basic_iostream 的物件能提供 basic_istreams 與 basic_ostreams 兩者的功能。在多重繼承的階層關係中，圖 24.12 的這種狀況稱爲**菱形繼承 (diamond inheritance)**。

因爲類別 basic_istream 和 basic_ostream 都繼承自 basic_ios，所以對於 basic_iostream，會存在一個潛在問題。類別 basic_iostream 會含有類別 basic_ios 成員的兩個副本；一個透過類別 basic_istream 繼承而來，一個透過 basic_ostream 繼承而來。因爲編譯器不知道應該使用哪一個版本的 basic_ios 類別成員，所以這將導致歧義的情形，並且產生編譯時期的錯誤。當然，basic_iostream 實際上並不會遭遇我們所提到的這個問題。在這一節中，我們將探討如何使用 virtual 基本類別，來解決從一個間接基本類別，繼承得到重複副本的問題。

發生在菱形繼承中的歧義所引起的編譯時期錯誤

圖 24.13 將示範說明在菱形繼承中可能發生的歧義情形。這個程式定義了類別 Base (第 8–12 行)，此類別含有純粹 `virtual` 函式 `print` (第 11 行)。類別 `DerivedOne` (第 15–23 行) 和 `DerivedTwo` (第 26–34 行) 都以 `public` 的方式繼承了類別 `Base`，並且重載了 `print` 函式。類別 `DerivedOne` 和類別 `DerivedTwo` 都含有 C++ 標準稱之爲**基本類別子物件 (base-class subobject)** 的項目，也就是這個範例中類別 `Base` 的成員。

```
1  // Fig. 24.13: fig24_13.cpp
2  // Attempting to polymorphically call a function that is
3  // multiply inherited from two base classes.
4  #include <iostream>
5  using namespace std;
6
7  // class Base definition
8  class Base
9  {
10 public:
11    virtual void print() const = 0; // pure virtual
12 }; // end class Base
13
14 // class DerivedOne definition
15 class DerivedOne : public Base
16 {
17 public:
18    // override print function
19    void print() const
20    {
21       cout << "DerivedOne\n";
22    } // end function print
23 }; // end class DerivedOne
24
25 // class DerivedTwo definition
26 class DerivedTwo : public Base
27 {
28 public:
29    // override print function
30    void print() const
31    {
32       cout << "DerivedTwo\n";
33    } // end function print
34 }; // end class DerivedTwo
```

圖 24.13　嘗試使用多型的方式呼叫多重繼承的函式

```
35
36  // class Multiple definition
37  class Multiple : public DerivedOne, public DerivedTwo
38  {
39  public:
40     // qualify which version of function print
41     void print() const
42     {
43        DerivedTwo::print();
44     } // end function print
45  }; // end class Multiple
46
47  int main()
48  {
49     Multiple both; // instantiate Multiple object
50     DerivedOne one; // instantiate DerivedOne object
51     DerivedTwo two; // instantiate DerivedTwo object
52     Base *array[ 3 ]; // create array of base-class pointers
53
54     array[ 0 ] = &both; // ERROR--ambiguous
55     array[ 1 ] = &one;
56     array[ 2 ] = &two;
57
58     // polymorphically invoke print
59     for ( int i = 0; i < 3; i++ )
60        array[ i ] -> print();
61  } // end main
```

Microsoft Visual C++ compiler error message:

```
c:\cpphtp7_examples\ch25\Fig24_13\fig24_13.cpp(54) : error C2594: '=' :
   ambiguous conversions from 'Multiple *' to 'Base *'
```

GNU C++ compiler error message:

```
fig24_13.cpp: In function ‘int main()’:
fig24_13.cpp:54: error: ‘Base’ is an ambiguous base of ‘Multiple’
```

圖 24.13　嘗試使用多型的方式呼叫多重繼承的函式 (續)

類別 `Multiple` (第 37–45 行) 繼承自 `DerivedOne` 和 `DerivedTwo` 兩個類別。在類別 `Multiple` 中，函式 `print` 已經被重載成呼叫 `DerivedTwo` 的 `print` (第 43 行)。請注意，我們必須以類別名稱 `DerivedTwo` 限定 `print` 的呼叫，藉此指出呼叫的是哪一個版本。

函式 `main` (第 47–61 行) 宣告了類別 `Multiple` (第 49 行)，`DerivedOne` (第 50 行) 和 `DerivedTwo` (第 51 行) 的物件。第 52 行將宣告一個 `Base*` 指標的陣列。每一個陣列元素都會以一個物件的位址予以初始化 (第 54–56 行)。我們將 `Multiple` 類別的物件 `both` 的位址指定給 `array[0]`時，會發生錯誤。物件 `both` 實際上含有型別

Base 的兩個子物件，所以編譯器不知道指標 array[0]應該指向哪一個子物件，而且編譯器會產生能指出歧義轉換的編譯時期錯誤。

利用 virtual 基本類別繼承方式消除重複的子物件

重複出現子物件的問題，可以使用 virtual 繼承方式來加以解決。當基本類別是以 virtual 方式被繼承的時候，只有一個子物件會出現在衍生類別中，這種繼承方式稱爲 **virtual 基本類別繼承 (virtual base-class inheritance)**。圖 24.14 將圖 24.13 的程式改寫成採用 virtual 基本類別。

```
1  // Fig. 24.14: fig24_14.cpp
2  // Using virtual base classes.
3  #include <iostream>
4  using namespace std;
5
6  // class Base definition
7  class Base
8  {
9  public:
10    virtual void print() const = 0; // pure virtual
11 }; // end class Base
12
13 // class DerivedOne definition
14 class DerivedOne : virtual public Base
15 {
16 public:
17    // override print function
18    void print() const
19    {
20       cout << "DerivedOne\n";
21    } // end function print
22 }; // end DerivedOne class
23
24 // class DerivedTwo definition
25 class DerivedTwo : virtual public Base
26 {
27 public:
28    // override print function
29    void print() const
30    {
31       cout << "DerivedTwo\n";
32    } // end function print
33 }; // end DerivedTwo class
34
35 // class Multiple definition
36 class Multiple : public DerivedOne, public DerivedTwo
37 {
```

圖 24.14　使用 virtual 基本類別

```
38  public:
39     // qualify which version of function print
40     void print() const
41     {
42        DerivedTwo::print();
43     } // end function print
44  }; // end Multiple class
45
46  int main()
47  {
48     Multiple both; // instantiate Multiple object
49     DerivedOne one; // instantiate DerivedOne object
50     DerivedTwo two; // instantiate DerivedTwo object
51
52     // declare array of base-class pointers and initialize
53     // each element to a derived-class type
54     Base *array[ 3 ];
55     array[ 0 ] = &both;
56     array[ 1 ] = &one;
57     array[ 2 ] = &two;
58
59     // polymorphically invoke function print
60     for ( int i = 0; i < 3; i++ )
61        array[ i ]->print();
62  } // end main
```

```
DerivedTwo
DerivedOne
DerivedTwo
```

圖 24.14　使用 virtual 基本類別 (續)

這個程式的關鍵性改變是，類別 DerivedOne (第 14 行) 和 DerivedTwo (第 25 行) 都經由標示 virtual public Base，繼承類別 Base。既然這兩個類別都繼承自 Base，所以它們都含有一個 Base 子物件。virtual 繼承方式的優點並不明顯，直到類別 Multiple 繼承自 DerivedOne 和 DerivedTwo 兩個類別 (第 36 行)，才明顯顯示出其差別。既然這兩個基本類別都使用 virtual 繼承方式來繼承類別 Base 的成員，所以編譯器會確保只有一個型別 Base 的子物件繼承到類別 Multiple 中。這樣作將消除在圖 24.13 中會由編譯器所產生的歧義錯誤。現在編譯器允許以隱喻的方式，將 main 函式第 55 行的衍生類別指標 (&both)，轉換成基本類別指標 array[0]。在第 60-61 行的 for 迴圈則會以多型的方式，替每一個物件呼叫 print 函式。

使用 virtual 基本類別的多重繼承階層中的建構子

如果針對基本類別使用的是預設建構子，則利用 virtual 基本類別實作繼承階層，會變得比較簡單。圖 24.13 和 24.14 的範例使用了編譯器所產生的預設建構子。如果

`virtual` 基本類別提供的是需要引數的建構子，因爲**最末層衍生類別 (most derived class)** 必須明確地呼叫 `virtual` 基本類別建構子，這將導致衍生類別的實作變得更加複雜。

軟體工程的觀點 24.5

爲 `virtual` 基本類別提供預設建構子，可以簡化類別階層的設計工作。

關於多重繼承的額外資訊

多重繼承是一個複雜的主題，一般都是在更高等的 C++ 課本中才會討論它。請造訪我們的 C++ 資源中心，以得到更多關於多重繼承的資訊：

```
www.deitel.com/cplusplus/
```

你可以在「C++ Multiple Inheritance」類別中找到一些文章和資源的連結，包括多重繼承 FAQ 以及使用多重繼承的技巧。

24.9 總結

你在本章中學到了如何使用 `const_cast` 運算子來移除變數的 `const` 特性。接著我們介紹了如何使用命名空間來確保程式中的識別字是唯一的，我們也解釋了它們如何幫忙解決名稱衝突的問題。你學到了幾個運算子關鍵字，對於其鍵盤不支援某些運算子符號，例如!、&、^、|等符號的程式設計者而言，這些運算子關鍵字將很有用。我們接著討論 `mutable` 的儲存空間類別修飾詞，這個修飾詞能讓你指定一個資料成員永遠都可以被修改，即使這個資料成員所屬的物件目前被程式視爲 `const` 物件。我們也示範了如何使用指向類別成員的指標，以及 `->*` 和 `.*` 運算子。最後，我們介紹了多重繼承，以及讓一個衍生類別同時繼承幾個基本類別的成員時所會產生的問題。同時我們討論了如何利用 `virtual` 繼承來解決這些問題。

摘要

24.2 `const_cast` 運算子

- C++ 提供 `const_cast` 運算子，用於取消 `const` 或 `volatile` 的限定。
- 當程式期望某個變數能讓其他程式加以修改的時候，此程式可以利用 `volatile` 修飾詞來宣告這個變數。將變數宣告爲 `volatile`，意謂著編譯器不應該針對此變數的使用進行最佳化，因爲這麼做會影響其他會存取和修改此 `volatile` 變數的程式的能力。

- 一般而言，因爲 const_cast 運算子允許程式能修改被宣告爲 const 的變數，然而對於 const 變數，我們會預設它是不能修改的，所以使用 const_cast 運算子是有危險的。
- 在一些情況下，取消 const 限定是值得冒險的，甚至是必要的。舉例來說，比較舊的 C 和 C++函式庫可能提供具有非 const 參數，而且不能修改其參數的函式。如果我們想要傳遞 const 資料給這樣的函式，我們會需要將資料的 const 限定取消掉；否則編譯器將顯示出錯誤訊息。
- 如果我們將非 const 資料傳遞給一個將資料當成常數來處理的函式，而且此函式會將該資料以常數的形式予以傳回，則我們可能需要取消傳回資料的 const 限定，以便存取和修改該資料。

24.3　可變類別成員 (mutable Class Members)

- 如果如上述這樣的資料成員需要永遠能進行修改，則 C++ 提供了儲存空間類別的修飾子 mutable，它可以作爲 const_cast 的替代操作方式。mutable 資料成員隨時可以進行修改，即使在 const 成員函式或 const 物件中也是如此。這樣作可以減少對於取消 const 限定的需求。
- mutable 和 const_cast 兩者都允許資料成員進行修改；不過它們用於不同的程式狀況中。對於不含 mutable 資料成員的 const 物件而言，每一次要修改其成員時，程式都必須使用運算子 const_cast。因爲這種成員並非隨時都能加以修改，所以這會大幅降低意外修改成員的可能性。
- 涉及 const_cast 的操作，通常會隱藏在成員函式的運作中。類別的使用者可能無法察覺到該成員正在被修改。

24.4　命名空間

- 程式包含許多定義於不同使用域的識別字。有時候，定義在某個使用域的變數會和定義在不同使用域，但是具有相同名稱的變數「重疊」(也就是發生使用域的抵觸)，因而可能產生名稱上的衝突。C++ 規範嘗試使用命名空間 (namespace) 來解決這個問題。
- 每一個 namespace 會定義放置識別字和變數的使用域。若要使用 namespace 的成員，成員名稱必須利用 namespace 名稱和二元使用域解析運算子 (::) 予以修飾和限定，或者在程式使用到成員名稱之前，先寫出 using 指令或宣告。
- 一般來說，這樣的 using 敘述會放置在 namespace 成員所使用的檔案開頭部分。
- 並非所有的 namespace 名稱都保證是唯一的。兩個不相關的程式設計廠商有可能會不慎替他們的 namespace 名稱取相同的識別字。
- namespace 可以包含常數、資料、類別、巢狀 namespace、函式等等。namespace 的定義必須擁有全域使用域，或巢狀安置在其它 namespace 內。

- 不具名的 namespace 具有隱喻的 using 指令，所以其成員擁有全域 namespace，能夠直接存取，而且不需要使用 namespace 名稱來加以限定。全域變數也是全域命名空間的一部分。
- 在存取巢狀 namespace 的成員時，這些成員必須加上 namespace 名稱 (除非成員當時是在巢狀 namespace 內部中被使用) 加以修飾和限定。
- 命名空間也可以具有別名。

24.5 運算子關鍵字

- C++標準提供了運算子關鍵字，它們可以用來取代幾種 C++ 運算子。對於其鍵盤不支援某些像!、&、^、"|、等等這類字元的程式設計者而言，運算子關鍵字是很有用的。

24.6 指向類別成員的指標 (.* 和 ->*)

- C++ 提供 .* 和 ->* 運算子，用於透過指標來存取類別成員。這是很少使用的功能，通常只有進階的 C++程式設計者會用到它。
- 要宣告一個指向函式的指標，必須在指標名稱前面附加 *，然後將它們放置在小括弧內。指向函式的指標必須標示出其指向的函式的傳回型別，以及該函式的參數列，如同其型別的一部分。

24.7 多重繼承

- 在 C++ 中，一個類別可以從超過一個以上的基本類別衍生出來，這是一種稱爲多重繼承的技術，使用這種技術的時候，衍生類別可以繼承兩個或兩個以上基本類別的成員。
- 多重繼承的一個常見問題是，每一個基本類別可能含有名稱相同的資料成員或成員函式。當我們要進行編譯的時候，這會導致歧義的問題。
- 單一繼承的是一種 (is-a) 關係也適用於多重繼承的關係。
- 舉例來說，多重繼承也使用於 C++ 標準函式庫中，藉此形成類別 basic_iostream。類別 basic_ios 是 basic_istream 和 basic_ostream 兩者的基本類別，這兩個衍生類別都是利用單一繼承的方式產生。類別 basic_iostream 則是繼承自 basic_istream 和 basic_ostream 兩者。這允許類別 basic_iostream 的物件能提供 basic_istreams 與 basic_ostreams 兩者的功能。在多重繼承的階層關係中，這種狀況稱爲菱形繼承。
- 因爲類別 basic_istream 和 basic_ostream 都繼承自 basic_ios，所以對於 basic_iostream，會存在一個潛在問題。如果沒有正確地加以實作，則類別 basic_iostream 可能含有類別 basic_ios 的成員的兩個副本，其中一個透過類別 basic_istream 繼承得來，另一個則透過類別 basic_ostream 繼承而來。因爲編譯器不知道應該使用哪一個版本的 basic_ios 類別成員，所以這將導致歧義的情形，並且產生編譯時期的錯誤。

24.8　多重繼承和 virtual 基本類別

- 當一個衍生類別的物件繼承了兩個或兩個以上基本類別子物件的時候，就可能發生菱形繼承關係的歧義情形。重複出現子物件的問題，可以使用 virtual 繼承方式來加以解決。當基本類別是以 virtual 方式被繼承的時候，只有一個子物件會出現在衍生類別中，這種繼承方式稱爲 virtual 基本類別繼承 (virtual base-class inheritance)。
- 如果針對基本類別使用的是預設建構子，則利用 virtual 基本類別實作繼承階層，會變得比較簡單。如果 virtual 基本類別提供的是需要引數的建構子，因爲最末層衍生類別 (most derived class) 必須明確地呼叫 virtual 基本類別建構子，以初始化繼承自 virtual 基本類別的成員，這將導致衍生類別的實作變得更加複雜。

術語

* 運算子 (* operator)
->* 運算子 (->* operator)
and 運算子關鍵字 (and operator keyword)
and_eq 運算子關鍵字 (and_eq operator keyword)
基本類別子物件 (base-class subobject)
bitand 運算子關鍵字 (bitand operator keyword)
bitor 運算子關鍵字 (bitor operator keyword)
compl 運算子關鍵字 (compl operator keyword)
const_cast 運算子 (const_cast operator)
菱形繼承 (diamond inheritance)
全域命名空間 (global namespace)
最末層衍生類別 (most derived class)
多重繼承 (multiple inheritance)
mutable 關鍵字 (mutable keyword)
命名空間 (namespace)
命名空間的別名 (namespace alias)
命名空間的成員 (namespace member)
巢狀命名空間 (nested namespace)
not 運算子關鍵字 (not operator keyword)
not_eq 運算子關鍵字 (not_eq operator keyword)
operator 關鍵字 (operator keyword)
or 運算子關鍵字 (or operator keyword)
or_eq 運算子關鍵字 (or_eq operator keyword)
編譯單元 (translation unit)
不具名命名空間 (unnamed namespace)
virtual 基本類別繼承 (virtual base class inheritance)
volatile 修飾詞 (volatile qualifier)
xor 運算子關鍵字 (xor operator keyword)
xor_eq 運算子關鍵字 (xor_eq operator keyword)

自我測驗

24.1 請在以下空格填入解答：

a) _____運算子能利用成員所屬的 namespace 來限定此成員。
b)_____運算子能將物件的「const」限定取消掉。
c) 因爲未命名 namespace 具有隱喻的 using 指令，所以其成員看起來是位於_____，能夠直接加以存取，而且不需要使用 namespace 名稱來修飾和限定。
d) 運算子_____是不等號的運算子關鍵字。

e) _____一個類別可以從超過一個以上的基本類別衍生出來，這樣的衍生方式稱爲_____。

f) 當基本類別以_____形式被繼承時，只會有一個基本類別的子物件出現在衍生類別中。

24.2 說明下列何者爲對，何者爲錯。如果答案是錯，請解釋爲什麼。

a) 非 const 引數要被傳遞給 const 函式的時候，應該利用 const_cast 運算子將函式的「const 限定」取消掉。

b) mutable 資料成員不可以在 const 成員函式中進行修改。

c) namespace 保證是獨一無二的。

d) 就像類別主體一樣，namespace 的主體也是以分號作爲結尾。

e) namespace 不能具有 namespace 作爲成員。

自我測驗解答

24.1 a) 二元的使用域解析 (::)。b) const_cast。c) 全域的 namespace。d) not_eq。e) 多重繼承。f) virtual。

24.2 a) 錯。將非 const 引數傳遞給 const 函式是符合程式語法的。不過，當我們將 const 參照或指標傳遞給非 const 函式的時候，應該使用 const_cast 運算子取消該參照或指標的「const 限定」。

b) 錯。mutable 資料成員永遠可以加以修改，即使是在 const 成員函式中也是如此。

c) 錯。程式設計者可能會不愼選擇已經使用了的 namespace 名稱。

d) 錯。namespace 主體並沒有以分號作爲結尾。

e) 錯。Namespace 可以具有巢狀結構。

習題

24.3 請在以下空格填入解答：

a) 關鍵字_____指明 namespace 或 namespace 的成員將被使用。

b) 運算子_____是邏輯 OR 的運算子關鍵字。

c) 儲存空間修飾子_____允許 const 物件的成員能進行修改。

d) _____修飾詞指出，某個物件可以讓其他程式修改。

e) 如果可能發生使用域互相抵觸的情形，應該在成員的前面加上_____名稱和使用域解析運算子_____。

f) namespace 的本體是利用_____加以圈圍起來。

g) 對於不含 mutable 資料成員的 const 物件而言，每一次要修改其成員時，程式都必須使用運算子_____。

24.4 (Currency 命名空間) 撰寫一個 namespace Currency，其中定義了常數成員 ONE、TWO、FIVE、TEN、TWENTY、FIFTY 和 HUNDRED。撰寫兩個使用到 Currency 的小程式。其中一個程式必須讓所有常數都能加以使用，而另一個程式則必須只讓 FIVE 能加以使用。

24.5 已知圖 24.15 的 namespace，請回答下列每一個敘述是眞或僞。如果答案是僞，請說明其理由。

a) 在 namespace Data 裡面可以使用變數 kilometers。

b) 在 namespace Data 裡面可以使用物件 string1。

c) 在 namespace Data 裡面可以使用常數 POLAND。

d) 在 namespace Data 裡面可以使用常數 GERMANY。

e) 在 namespace Data 裡面可以使用函式 function。

f) 在 namespace CountryInformation 裡面可以使用命名空間 Data。

g) 在 namespace CountryInformation 裡面可以使用物件 map。

h) 在 namespace RegionalInformation 裡面可以使用物件 string1。

```
1  namespace CountryInformation
2  {
3     using namespace std;
4     enum Countries { POLAND, SWITZERLAND, GERMANY,
5                      AUSTRIA, CZECH_REPUBLIC };
6     int kilometers;
7     string string1;
8
9     namespace RegionalInformation
10    {
11       short getPopulation(); // assume definition exists
12       MapData map; // assume definition exists
13    } // end RegionalInformation
14 } // end CountryInformation
15
16 namespace Data
17 {
18    using namespace CountryInformation::RegionalInformation;
19    void *function( void *, int );
20 } // end Data
```

圖 24.15　習題 24.5 的命名空間

24.6 比較並列出 mutable 和 const_cast 的差異。最少提供一個例子，說明何時其中一個比另一個更適用。[請注意：這個習題並不需要撰寫任何程式碼。]

24.7 (更改 `const` 變數) 撰寫一個程式，讓這個程式使用 `const_cast` 來修改一個宣告成 `const` 的變數。[提示：在你的解答中，使用指向 `const` 識別字的指標。]

24.8 使用 `virtual` 基本類別可以解決什麼樣的問題？

24.9 (`virtual` 基本類別) 撰寫一個使用 `virtual` 基本類別的程式。位於繼承階層最頂端的類別必須提供一個能接收最少一個引數的建構子 (也就是，不提供預設建構子)。對於此繼承階層而言，這項要求會提出什麼樣的挑戰？

24.10 找出以下敘述的錯誤：如果可以的話，請說明如何更正每一個錯誤。

a)
```
namespace Name {
   int x;
   int y;
   mutable int z;
};
```

b)
```
int integer = const_cast< int >( double );
```

c)
```
namespace PCM( 111, "hello" ); // construct namespace
```