類別：深入探討 (下) 10

But what, to serve our private ends,
Forbids the cheating of our friends?
—Charles Churchill
Instead of this absurd division into sexes they ought to class people as static and dynamic.
—Evelyn Waugh
Have no friends not equal to yourself.
—Confucius

學習目標

在本章中，你將學到：

- 運用 `const` 物件與 `const` 成員函式。
- 組合其它物件來產生新物件。
- 使用夥伴 (`friend`) 函式與類別。
- 使用 `this` 指標。
- 使用 `static` 資料成員與成員函式。
- 容器類別的概念。
- 輪詢容器元素的循環器 (iterator) 類別。
- 透過代理類別 (proxy classs) 隱藏類別的實作內容。

本章綱要

10.1 簡介
10.2 const 物件和 const 成員函式
10.3 組合：將物件當作類別成員
10.4 夥伴函式與類別
10.5 使用 this 指標
10.6 static 類別成員
10.7 資料抽象化與資訊隱藏
10.8 總結
摘要｜術語｜自我測驗｜自我測驗解答｜習題｜進階習題

10.1 簡介

本章我們要繼續討論和類別及與資料抽象化有關的進階主題。我們透過 `const` 物件與 `const` 成員函式，防止物件被修改，強制實行最小權限原則。我們還會討論軟體再利用的一種形式，稱爲組合 (composition)，讓類別包含其它類別的物件作爲其成員。接下來，我們會討論類別的夥伴關係 (friendship)，讓非成員函式存取類別的非 `public` 成員；運算子多載是此技巧常見的一個應用 (主要因效能上的考量；詳情請見第 11 章)。我們還會討論一個特別的指標 `this`，類別中任何非 `static` 的成員函式，都隱含此指標作爲引數，以便存取正確物件的資料成員，以及其它非 `static` 的成員函式。最後，我們會說明爲何需要 `static` 類別成員，並示範如何在類別中使用 `static` 資料成員與成員函式。

10.2 `const` 物件和 `const` 成員函式

在這一節中，我們要探討如何對物件套用最小權限原則。有些物件需要被修改，有些則不需要。程式設計師可利用關鍵字 `const`，指出該物件是不可修改的，任何嘗試修改的動作都會造成編譯的錯誤。以下敘述

```
const Time noon( 12, 0, 0 );
```

宣告類別 `Time` 的 `const` 物件 `noon`，並將其初始值設爲中午 12 點。

軟體工程的觀點 10.1

因爲任何嘗試修改 `const` 物件的動作，都會在編譯期被發現，故不會造成執行期錯誤。

增進效能的小技巧 10.1

適當地將變數或物件宣告爲 const 能提高程式的執行效能，因爲編譯器能對常數進行無法對一般變數使用的最佳化處理。

除非成員函式亦宣告爲 const，否則 C++編譯器不允許 const 物件呼叫任何的成員函式；即使是不能修改物件內容的 get 成員函式，也是如此。

宣告爲 const 的函式必須在原型 (圖 10.1 第 19-24 行) 與定義 (圖 10.2 第 43、49、55 和 61 行) 中，分別於函式參數列之前，以及函式本體的左大括號前面，寫出 const 關鍵字。

常見的程式設計錯誤 10.1

定義爲 const 的成員函式，若嘗試修改物件的資料成員，會造成編譯錯誤。

常見的程式設計錯誤 10.2

定義爲 const 的成員函式，若呼叫同物件中的其它非 const 成員函式，會造成編譯錯誤。

常見的程式設計錯誤 10.3

對 const 物件呼叫非 const 成員函式，會造成編譯錯誤。

軟體工程的觀點 10.2

我們可對 const 成員函式進行多載，讓它擁有非 const 的版本。編譯器會依叫用函式的物件，決定該使用哪個版本；若物件爲 const，則使用 const 版本，否則使用非 const 版本。

建構子和解構子經常需要修改物件，建構子必須修改物件的內容，以便正確設定物件的初始值。解構子則會在系統清除物件之前，執行資源回收的工作。

常見的程式設計錯誤 10.4

嘗試將建構子或解構子宣告爲 const，會造成編譯錯誤。

定義和使用 const 成員函式

圖 10.1-10.3 的程式對圖 9.8-9.9 中的類別 Time 進行修改，讓 get 與 printUniversal 函式變成 const。在 Time.h 標頭檔 (圖 10.1) 中，程式第 19-21 和第 24 行在每個函式參數列的末端，加上 const 關鍵字。在圖 10.2 中 (分別爲第 43、49、55 和 61 行)，這些函式對應的定義處，我們也在函式參數列之後，寫出關鍵字 const。

```
1  // Fig. 10.1: Time.h
2  // Time class definition with const member functions.
3  // Member functions defined in Time.cpp.
4  #ifndef TIME_H
5  #define TIME_H
6
7  class Time
8  {
9  public:
10    Time( int = 0, int = 0, int = 0 ); // default constructor
11
12    // set functions
13    void setTime( int, int, int ); // set time
14    void setHour( int ); // set hour
15    void setMinute( int ); // set minute
16    void setSecond( int ); // set second
17
18    // get functions (normally declared const)
19    int getHour() const; // return hour
20    int getMinute() const; // return minute
21    int getSecond() const; // return second
22
23    // print functions (normally declared const)
24    void printUniversal() const; // print universal time
25    void printStandard(); // print standard time (should be const)
26 private:
27    int hour; // 0 - 23 (24-hour clock format)
28    int minute; // 0 - 59
29    int second; // 0 - 59
30 }; // end class Time
31
32 #endif
```

圖 10.1　具 const 成員函式的 Time 類別定義

```
1  // Fig. 10.2: Time.cpp
2  // Time class member-function definitions.
3  #include <iostream>
4  #include <iomanip>
5  #include "Time.h" // include definition of class Time
6  using namespace std;
7
8  // constructor function to initialize private data;
9  // calls member function setTime to set variables;
10 // default values are 0 (see class definition)
11 Time::Time( int hour, int minute, int second )
12 {
13    setTime( hour, minute, second );
14 } // end Time constructor
15
16 // set hour, minute and second values
17 void Time::setTime( int hour, int minute, int second )
18 {
19    setHour( hour );
```

圖 10.2　Time 成員函式的定義

```
20       setMinute( minute );
21       setSecond( second );
22    } // end function setTime
23
24    // set hour value
25    void Time::setHour( int h )
26    {
27       hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
28    } // end function setHour
29
30    // set minute value
31    void Time::setMinute( int m )
32    {
33       minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
34    } // end function setMinute
35
36    // set second value
37    void Time::setSecond( int s )
38    {
39       second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
40    } // end function setSecond
41
42    // return hour value
43    int Time::getHour() const // get functions should be const
44    {
45       return hour;
46    } // end function getHour
47
48    // return minute value
49    int Time::getMinute() const
50    {
51       return minute;
52    } // end function getMinute
53
54    // return second value
55    int Time::getSecond() const
56    {
57       return second;
58    } // end function getSecond
59
60    // print Time in universal-time format (HH:MM:SS)
61    void Time::printUniversal() const
62    {
63       cout << setfill( '0' ) << setw( 2 ) << hour << ":"
64          << setw( 2 ) << minute << ":" << setw( 2 ) << second;
65    } // end function printUniversal
66
67    // print Time in standard-time format (HH:MM:SS AM or PM)
68    void Time::printStandard() // note lack of const declaration
69    {
70       cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
71          << ":" << setfill( '0' ) << setw( 2 ) << minute
72          << ":" << setw( 2 ) << second << ( hour < 12 ? " AM" : " PM" );
73    } // end function printStandard
```

圖 10.2　Time 成員函式的定義 (續)

圖 10.3 產生兩個 Time 物件，一個是非 const 物件 wakeUp (第 7 行) ，而另一個則是 const 物件 noon (第 8 行)。此程式嘗試呼叫 const 物件 noon 的非 const 成員函式 setHour (第 13 行) 和 printStandard (第 20 行)。此時，編譯器會產生錯誤訊息。程式也說明其它作用於物件的三種不同成員函式，即呼叫非 const 物件的非 const 成員函式 (第 11 行)、呼叫非 const 物件的 const 成員函式 (第 15 行) 以及呼叫 const 物件的 const 成員函式 (第 17-18 行)。對 const 物件呼叫非 const 成員函式產生的錯誤訊息顯示在輸出視窗之中。

```
1  // Fig. 10.3: fig10_03.cpp
2  // Attempting to access a const object with non-const member functions.
3  #include "Time.h" // include Time class definition
4
5  int main()
6  {
7     Time wakeUp( 6, 45, 0 ); // non-constant object
8     const Time noon( 12, 0, 0 ); // constant object
9
10                              // OBJECT      MEMBER FUNCTION
11    wakeUp.setHour( 18 );     // non-const   non-const
12
13    noon.setHour( 12 );       // const       non-const
14
15    wakeUp.getHour();         // non-const   const
16
17    noon.getMinute();         // const       const
18    noon.printUniversal();    // const       const
19
20    noon.printStandard();     // const       non-const
21 } // end main
```

Microsoft Visual C++ compiler error messages:

```
C:\cpphtp7_examples\ch10\Fig10_01_03\fig10_03.cpp(13) : error C2662:
   'Time::setHour' : cannot convert 'this' pointer from 'const Time' to
   'Time &'
        Conversion loses qualifiers
C:\cpphtp7_examples\ch10\Fig10_01_03\fig10_03.cpp(20) : error C2662:
   'Time::printStandard' : cannot convert 'this' pointer from 'const Time' to
   'Time &'
        Conversion loses qualifiers
```

GNU C++ compiler error messages:

```
fig10_03.cpp:13: error: passing 'const Time' as 'this' argument of
   'void Time::setHour(int)' discards qualifiers
fig10_03.cpp:20: error: passing 'const Time' as 'this' argument of
   'void Time::printStandard()' discards qualifiers
```

圖 10.3 const 物件和 const 成員函式

建構子雖然非 const (圖 10.2 第 11-14 行)，但可用來初始化 const 物件 (圖 10.3 第 8 行)。觀察 Time 建構子的定義 (圖 10.2 第 11-14 行) 可以發現，它呼叫非 const 成員函式 setTime (第 17-22 行)，設定 Time 物件的初始值。在 const 物件的建構子中呼叫非 const 成員函式是合法的。從建構子設定完物件的初始值，到呼叫該物件的解構子期間，該物件都會維持 const (即不能修改的常數)。

請注意，在圖 10.3 第 20 行雖然類別 Time 的成員函式 printStandard 不會修改呼叫它的物件，仍會產生編譯錯誤。不對物件進行修改，並不代表它就自動是 const；必須明確寫出才行。

用成員初始值設定 const 資料成員的初始值

圖 10.4-10.6 的程式用到了**成員初始值語法 (member initializer syntax)**。資料成員可以透過成員初始值列表設定初始值，而 const 資料成員與參照資料成員必須透過成員初始值設定初始值。本章稍後會談到，成員物件也必須如此設定初始值。

```
1  // Fig. 10.4: Increment.h
2  // Definition of class Increment.
3  #ifndef INCREMENT_H
4  #define INCREMENT_H
5
6  class Increment
7  {
8  public:
9     Increment( int c = 0, int i = 1 ); // default constructor
10
11    // function addIncrement definition
12    void addIncrement()
13    {
14       count += increment;
15    } // end function addIncrement
16
17    void print() const; // prints count and increment
18 private:
19    int count;
20    const int increment; // const data member
21 }; // end class Increment
22
23 #endif
```

圖 10.4　類別 Increment 的定義包含非 const 資料成員 count 和 const 資料成員 increment

```
1  // Fig. 10.5: Increment.cpp
2  // Member-function definitions for class Increment demonstrate using a
3  // member initializer to initialize a constant of a built-in data type.
4  #include <iostream>
5  #include "Increment.h" // include definition of class Increment
6  using namespace std;
7
8  // constructor
9  Increment::Increment( int c, int i )
10    : count( c ), // initializer for non-const member
11      increment( i ) // required initializer for const member
12 {
13    // empty body
14 } // end constructor Increment
15
16 // print count and increment values
17 void Increment::print() const
18 {
19    cout << "count = " << count << ", increment = " << increment << endl;
20 } // end function print
```

圖 10.5　用成員初始值列表設定內建型別常數的初始值

```
1  // Fig. 10.6: fig10_06.cpp
2  // Program to test class Increment.
3  #include <iostream>
4  #include "Increment.h" // include definition of class Increment
5  using namespace std;
6
7  int main()
8  {
9     Increment value( 10, 5 );
10
11    cout << "Before incrementing: ";
12    value.print();
13
14    for ( int j = 1; j <= 3; j++ )
15    {
16       value.addIncrement();
17       cout << "After increment " << j << ": ";
18       value.print();
19    } // end for
20 } // end main
```

```
Before incrementing: count = 10, increment = 5
After increment 1: count = 15, increment = 5
After increment 2: count = 20, increment = 5
After increment 3: count = 25, increment = 5
```

圖 10.6　叫用 `Increment` 物件的 `print` 和 `addIncrement` 成員函式

建構子的定義 (圖 10.5 第 9-14 行) 使用**成員初始值列表 (member initializer list)** 設定類別 Increment 的資料成員，非 const 整數 count 與 const 整數 increment 的初始值 (於圖 10.4 第 19-20 行宣告)。成員初始值列表寫在建構子參數列和建構子主體開始的左大括號中間。(圖 10.5 第 10-11 行)，和參數列之間以冒號 (:) 隔開。串列中的每個項目，是由資料成員名稱加上以小括號包圍的初始值所構成。在本例中，count 的初始值設爲建構子參數 c 的值，而 increment 的初始值設爲建構子參數 i 的值。成員初始值之間以逗號隔開。此外，程式會在執行建構子主體之前，先執行成員初始值列表。

軟體工程的觀點 10.3

const 物件不能用賦值敘述修改，所以必須於初始化時設值。若類別含有宣告爲 const 的資料成員，必須提供成員初始值列表，讓建構子正確設定其初始值。宣告爲 const 的參照也一樣。

試圖用指定運算子初始化 const 資料成員

圖 10.7-10.9 的程式顯示，試圖在 increment 建構子中，用指定敘述式初始化 const 資料成員 increment (圖 10.8 第 12 行)，而非使用成員初始值列表時，產生的編譯錯誤。圖 10.8 第 11 行不會產生編譯錯誤，因爲 count 並未宣告爲 const。

常見的程式設計錯誤 10.5

程式若未提供 const 資料成員的成員初始值，會造成編譯錯誤。

軟體工程的觀點 10.4

常數資料成員 (const 物件或變數) 以及宣告成參照的資料成員，必須用成員初始值列表設定初始值，不能在建構子主體中使用賦值敘述。

```
1  // Fig. 10.7: Increment.h
2  // Definition of class Increment.
3  #ifndef INCREMENT_H
4  #define INCREMENT_H
5
6  class Increment
7  {
8  public:
9     Increment( int c = 0, int i = 1 ); // default constructor
10
11    // function addIncrement definition
12    void addIncrement()
```

圖 10.7　類別 Increment 的定義包含非 const 資料成員 count 和 const 資料成員 Increment

```
13    {
14       count += increment;
15    } // end function addIncrement
16
17    void print() const; // prints count and increment
18 private:
19    int count;
20    const int increment; // const data member
21 }; // end class Increment
22
23 #endif
```

圖 10.7 類別 Increment 的定義包含非 const 資料成員 count 和 const 資料成員 Increment (續)

```
1  // Fig. 10.8: Increment.cpp
2  // Erroneous attempt to initialize a constant of a built-in data
3  // type by assignment.
4  #include <iostream>
5  #include "Increment.h" // include definition of class Increment
6  using namespace std;
7
8  // constructor; constant member 'increment' is not initialized
9  Increment::Increment( int c, int i )
10 {
11    count = c; // allowed because count is not constant
12    increment = i; // ERROR: Cannot modify a const object
13 } // end constructor Increment
14
15 // print count and increment values
16 void Increment::print() const
17 {
18    cout << "count = " << count << ", increment = " << increment << endl;
19 } // end function print
```

圖 10.8 錯誤地運用賦值敘述設定內建型別常數的初始值

```
1  // Fig. 10.9: fig10_09.cpp
2  // Program to test class Increment.
3  #include <iostream>
4  #include "Increment.h" // include definition of class Increment
5  using namespace std;
6
7  int main()
8  {
9     Increment value( 10, 5 );
10
11    cout << "Before incrementing: ";
12    value.print();
```

圖 10.9 測試類別 Increment 錯誤訊息的程式

```
13
14      for ( int j = 1; j <= 3; j++ )
15      {
16         value.addIncrement();
17         cout << "After increment " << j << ": ";
18         value.print();
19      } // end for
20   } // end main
```

Microsoft Visual C++ compiler error messages:

```
C:\cpphtp7_examples\ch10\Fig10_07_09\Increment.cpp(10) : error C2758:
   'Increment::increment' : must be initialized in constructor base/member
   initializer list
       C:\cpphtp7_examples\ch10\Fig10_07_09\increment.h(20) : see
          declaration of 'Increment::increment'
C:\cpphtp7_examples\ch10\Fig10_07_09\Increment.cpp(12) : error C2166:
   l-value specifies const object
```

GNU C++ compiler error messages:

```
Increment.cpp:9: error: uninitialized member 'Increment::increment' with
   'const' type 'const int'
Increment.cpp:12: error: assignment of read-only data-member
   'Increment::increment'
```

圖 10.9　測試類別 `Increment` 錯誤訊息的程式 (續)

圖 10.8 第 16-19 行的函式 `print` 宣告爲 `const`。讀者可能會覺的奇怪，爲何要把這個函式標示爲 `const`，因爲程式大概不會宣告 `const Increment` 這樣的物件。然而，程式有可能會使用指向 `Increment` 物件的 `const` 參照，或指向 `Increment` 物件的常數指標。一般來說，當 `Increment` 類別的物件傳遞給函式，或由函式傳回時，就會發生這種情況。在此情況下，只有 `Increment` 類別的 `const` 成員函式才可以透過參照或指標進行呼叫。因此，把函式 `print` 宣告成 `const` 是很合理的；這麼做可以避免錯誤，因爲在這些情況下 `Increment` 物件必須當成 `const` 物件處理。

測試和除錯的小技巧 10.1

請將所有不修改物件內容的類別成員函式宣告成 `const`。雖然這看起來也許有些奇怪，因爲我們並非有意要建立這種類別的 `const` 物件或透過 `const` 參照及指向 `const` 的指標存取其物件。不過，將這種成員函式宣告成 `const` 確實有些好處：如果不小心在函式中修改物件內容，編譯器會發出錯誤訊息。

10.3　組合：將物件當作類別成員

類別物件 AlarmClock 需知道何時應該發出鬧鈴聲音，因此，我們可讓 Time 物件作爲 AlarmClock 的一個成員。這稱爲**組合 (composition)**，又稱爲**「有一個」關係 (has-a relationship)**。類別可以包含其它類別的物件作爲自己的成員。

軟體工程的觀點 10.5

組合是最常見的軟體再利用方式，使類別包含其它類別的物件。

建立物件時，程式會自動呼叫其類別的建構子。前面我們學過如何在 main 中傳遞引數給物件的建構子。本節將說明，物件的建構子如何透過成員初始值列表，將引數傳遞給成員物件的建構子。

軟體工程的觀點 10.6

成員物件會按照宣告的順序建立 (而非成員初始值列表中的順序)；並且，成員物件會在包圍它們的類別物件 (又稱**所屬物件 host objects**) 之前建立。

下一個程式以類別 Date(圖 10.10-10.11) 和類別 Employee (圖 10.12-10.13) 爲例示範組合。類別 Employee 的定義 (圖 10.12) 包含 private 資料成員 firstName、lastName、birthDate 和 hireDate。成員 birthDate 和 hireDate 是類別 Date 的 const 物件，此類別含有 private 資料成員 month、day 和 year。Employee 建構子的標頭 (圖 10.13 第 10-11 行) 指出，建構子接受四個參數 (first、last、dateOfBirth 和 dateOfHire)。前兩個參數透過成員初始值列表，傳遞給 String 類別的建構子。後兩個參數則透過成員初始值列表，傳遞給 Date 類別的建構子。

```
1  // Fig. 10.10: Date.h
2  // Date class definition; Member functions defined in Date.cpp
3  #ifndef DATE_H
4  #define DATE_H
5
6  class Date
7  {
8  public:
9     static const int monthsPerYear = 12; // number of months in a year
10    Date( int = 1, int = 1, int = 1900 ); // default constructor
11    void print() const; // print date in month/day/year format
12    ~Date(); // provided to confirm destruction order
13 private:
14    int month; // 1-12 (January-December)
```

圖 10.10　Date 類別定義

```
15    int day; // 1-31 based on month
16    int year; // any year
17
18    // utility function to check if day is proper for month and year
19    int checkDay( int ) const;
20 }; // end class Date
21
22 #endif
```

圖 10.10　Date 類別定義 (續)

```
1  // Fig. 10.11: Date.cpp
2  // Date class member-function definitions.
3  #include <iostream>
4  #include "Date.h" // include Date class definition
5  using namespace std;
6
7  // constructor confirms proper value for month; calls
8  // utility function checkDay to confirm proper value for day
9  Date::Date( int mn, int dy, int yr )
10 {
11    if ( mn > 0 && mn <= monthsPerYear ) // validate the month
12       month = mn;
13    else
14    {
15       month = 1; // invalid month set to 1
16       cout << "Invalid month (" << mn << ") set to 1.\n";
17    } // end else
18
19    year = yr; // could validate yr
20    day = checkDay( dy ); // validate the day
21
22    // output Date object to show when its constructor is called
23    cout << "Date object constructor for date ";
24    print();
25    cout << endl;
26 } // end Date constructor
27
28 // print Date object in form month/day/year
29 void Date::print() const
30 {
31    cout << month << '/' << day << '/' << year;
32 } // end function print
33
34 // output Date object to show when its destructor is called
35 Date::~Date()
36 {
37    cout << "Date object destructor for date ";
38    print();
39    cout << endl;
```

圖 10.11　Date 成員函式定義

```
40  } // end ~Date destructor
41
42  // utility function to confirm proper day value based on
43  // month and year; handles leap years, too
44  int Date::checkDay( int testDay ) const
45  {
46     static const int daysPerMonth[ monthsPerYear + 1 ] =
47        { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
48
49     // determine whether testDay is valid for specified month
50     if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
51        return testDay;
52
53     // February 29 check for leap year
54     if ( month == 2 && testDay == 29 && ( year % 400 == 0 ||
55        ( year % 4 == 0 && year % 100 != 0 ) ) )
56        return testDay;
57
58     cout << "Invalid day (" << testDay << ") set to 1.\n";
59     return 1; // leave object in consistent state if bad value
60  } // end function checkDay
```

圖 10.11 Date 成員函式定義 (續)

```
1  // Fig. 10.12: Employee.h
2  // Employee class definition showing composition.
3  // Member functions defined in Employee.cpp.
4  #ifndef EMPLOYEE_H
5  #define EMPLOYEE_H
6
7  #include <string>
8  #include "Date.h" // include Date class definition
9  using namespace std;
10
11 class Employee
12 {
13 public:
14    Employee( const string &, const string &,
15       const Date &, const Date & );
16    void print() const;
17    ~Employee(); // provided to confirm destruction order
18 private:
19    string firstName; // composition: member object
20    string lastName; // composition: member object
21    const Date birthDate; // composition: member object
22    const Date hireDate; // composition: member object
23 }; // end class Employee
24
25 #endif
```

圖 10.12 示範組合的 Employee 類別定義

```
1  // Fig. 10.13: Employee.cpp
2  // Employee class member-function definitions.
3  #include <iostream>
4  #include "Employee.h" // Employee class definition
5  #include "Date.h" // Date class definition
6  using namespace std;
7
8  // constructor uses member initializer list to pass initializer
9  // values to constructors of member objects
10 Employee::Employee( const string &first, const string &last,
11    const Date &dateOfBirth, const Date &dateOfHire )
12    : firstName( first ), // initialize firstName
13      lastName( last ), // initialize lastName
14      birthDate( dateOfBirth ), // initialize birthDate
15      hireDate( dateOfHire ) // initialize hireDate
16 {
17    // output Employee object to show when constructor is called
18    cout << "Employee object constructor: "
19       << firstName << ' ' << lastName << endl;
20 } // end Employee constructor
21
22 // print Employee object
23 void Employee::print() const
24 {
25    cout << lastName << ", " << firstName << "  Hired: ";
26    hireDate.print();
27    cout << "  Birthday: ";
28    birthDate.print();
29    cout << endl;
30 } // end function print
31
32 // output Employee object to show when its destructor is called
33 Employee::~Employee()
34 {
35    cout << "Employee object destructor: "
36       << lastName << ", " << firstName << endl;
37 } // end ~Employee destructor
```

圖 10.13　`Employee` 類別的成員函式定義，包括具成員初始值列表的建構子

`Employee` 建構子的成員初始值串列

建構子標頭之後的冒號 (`:`) 接著成員初始值串列 (圖 10.13 第 12 行)。成員初始值串列指出，`Employee` 建構子的參數會傳給 `string` 和 `Date` 資料成員的建構子。參數 `first`、`last`、`dateOfBirth` 和 `dateOfHire` 會分別被傳遞給物件 `firstName` (圖 10.13 第 12 行)、`lastName` (圖 10.13 第 13 行)、`birthDate` (圖 10.13 第 14 行) 和 `hireDate` (圖 10.13 第 15 行) 的建構子。成員初始值串列中的項目以逗號分隔。

Date 類別預設的複製建構子

請注意，Date 類別 (圖 10.10) 並未提供接受參數爲 Date 型別的建構子。那麼，類別 Employee 建構子的成員初始值串列如何能把 Date 物件傳入 Date 建構子，來初始化 birthDate 和 hireDate 物件呢？ 請回想我們在第 9 章提過，編譯器會爲類別提供預設的複製建構子，此建構子會將建構子的引數物件的每個成員，複製到要初始化物件的對應成員。第 11 章會討論如何定義自己的複製建構子。

測試類別 Date 和 Employee

圖 10.14 產生兩個 Date 物件 (第 9-10 行)，並把它們當作引數，傳入於第 11 行產生的 Employee 物件的建構子。程式於第 14 行輸出 Employee 物件的資料。程式在第 9-10 行建立 Date 物件時，於圖 10.11 的第 9-26 行定義的 Date 建構子會顯示一行輸出，指出建構子被呼叫了 (見範例輸出的前兩行)。[請注意：程式圖 10.14 中第 11 行進行兩次額外的 Date 建構子呼叫，它們並沒有出現在程式的輸出中。當 Employee 的 Date 成員物件在其建構子的成員初始值串列 (圖 10.13 第 14-15 行) 中設定初始值時，會呼叫 Date 的預設複製建構子。因爲這個建構子是編譯器自動定義的，所以呼叫它時，不會輸出任何訊息。]

類別 Date 和 Employee 各自擁有解構子 (分別是圖 10.11 的第 35-40 行，以及圖 10.13 的第 33-37 行)，當其物件被清除時，程式會列印出一個訊息。我們就可以從程式的輸出確認物件是由內而外建構起來，而解構則是以相反的順序，由外而內進行 (即先清除 Employee 物件後，才清除成員物件 Date)。請注意圖 10.14 的最後四行輸出。最後兩行分別是 Date 物件 hire (第 10 行) 和 birth (第 9 行) 的 Date 解構子的輸出。這些輸出說明，main 中的三個物件會以它們建構的相反順序，進行解構。(Employee 解構子的輸出是倒數五行)。輸出視窗的倒數第 4 和第 3 行，顯示 Employee 的成員物件 hireDate (圖 10.12 第 22 行) 和 birthDate (圖 10.12 第 21 行) 執行的解構子。我們可以從程式的輸出確認，Employee 物件是由外而內進行解構，也就是說，程式會先執行 Employee 解構子 (輸出視窗的倒數五行)，然後，成員物件會以它們建構的相反順序進行解構。string 類別的解構子中沒有輸出敘述式，因此我們不會看到 firstName 和 lastName 物件被解構。同樣地，圖 10.14 的輸出沒有顯示 birthDate 和 hireDate 物件執行的建構子，因爲它們都是 C++編譯器提供的預設 Date 類別複製建構子。

```
1  // Fig. 10.14: fig10_14.cpp
2  // Demonstrating composition--an object with member objects.
3  #include <iostream>
4  #include "Employee.h" // Employee class definition
5  using namespace std;
6
7  int main()
8  {
9     Date birth( 7, 24, 1949 );
10    Date hire( 3, 12, 1988 );
11    Employee manager( "Bob", "Blue", birth, hire );
12
13    cout << endl;
14    manager.print();
15
16    cout << "\nTest Date constructor with invalid values:\n";
17    Date lastDayOff( 14, 35, 1994 ); // invalid month and day
18    cout << endl;
19 } // end main
```

```
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Employee object constructor: Bob Blue

Blue, Bob  Hired: 3/12/1988  Birthday: 7/24/1949

Test Date constructor with invalid values:
Invalid month (14) set to 1.
Invalid day (35) set to 1.
Date object constructor for date 1/1/1994

Date object destructor for date 1/1/1994
Employee object destructor: Blue, Bob
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
```

There are actually five constructor calls when an `Employee` is constructed—two calls to the `string` class's constructor (lines 12–13 of Fig. 10.13), two calls to the `Date` class's default copy constructor (lines 14–15 of Fig. 10.13) and the call to the `Employee` class's constructor.

圖 10.14　組合 — 包含物件成員的物件

未使用成員初始值串列時，會發生什麼事？

未提供成員初始值給成員物件時，系統會自動呼叫該物件的預設建構子。由預設建構子所設定的數值，可透過 set 函式來重新設定。不過較複雜的初始化，會需要可觀的額外工作和時間。

常見的程式設計錯誤 10.6

成員物件若未用成員初始值串列設定初始值，且其類別未提供預設建構子 (即類別定義一或多個建構子，但未提供預設建構子)，會產生編譯器錯誤。

增進效能的小技巧 10.2

在成員初值串列中明確初始化成員物件。這可以避免「重複初始化」成員物件的額外負擔：第一次是呼叫成員物件的預設建構子，而第二次是稍後在建構子的本體中，用 set 函式設定成員物件的值。

軟體工程的觀點 10.7

類別成員若爲其它類別的物件，將其設成 `public`，不會影響該物件的 `private` 成員之資料封裝性和資訊隱藏性。不過，這會破壞包含該物件所屬類別的封裝性和隱藏性，因此類別的成員物件，應該像其它資料成員一樣設爲 `private`。

10.4 夥伴函式與類別

類別的**夥伴函式 (friend function)** 定義於類別範圍的外部，它可以存取類別的非 `public` (和 `public`) 成員。獨立的函式或其他類別的成員函式可以宣告成另一個類別的夥伴。

使用 `friend` 函式可以增進程式的執行效率。本節會舉出如何使用夥伴函式的詳細例子。在稍後的章節中，我們會介紹如何用夥件函式多載類別物件的運算子 (第 11 章)，且建立循環器類別 (iterator classes) (第 20 章)。循環器類別的物件可以用來連續選取或操作容器類別物件中的項目。容器類別的物件可以儲存多個項目。我們在第 11 章中將會提到，某些時候不適合使用成員函式，這時就可使用夥伴函式。

要將函式宣告爲類別的夥伴函式，必須在類別定義中，在函式原型前面加上關鍵字 `friend`。若要將類別 `ClassTwo` 宣告成類別 `ClassOne` 的夥伴，請將以下的宣告

```
friend class ClassTwo;
```

置於類別 `ClassOne` 的定義中。

軟體工程的觀點 10.8

雖然夥伴函式的原型出現在類別的定義中，但它不是成員函式。

軟體工程的觀點 10.9

成員函式的存取權限如 `private`、`protected` 和 `public` 與夥伴宣告 (`friend declaration`) 無關，所以夥伴宣告可以放在類別定義中任何位置。

良好的程式設計習慣 10.1

請將所有夥伴宣告放在類別定義的最前面，且在宣告的前面不應該出現任何成員存取修飾字。

夥伴關係是由其它類別主動賦予，而非自行取得的權限，也就是說，類別 B 若想成爲類別 A 的夥伴，則類別 A 必須明確宣告類別 B 是它的夥伴。另外，夥伴關係沒有對稱性 (symmetric)，也不具轉移性 (transitive)，意思是說，若類別 A 是類別 B 的夥伴，且類別 B 是類別 C 的夥伴，則你不可以說類別 B 是類別 A 的夥伴 (不具對稱性) 或類別 C 是類別 B 的夥伴(不具對稱性)，或類別 A 是類別 C 的夥伴 (不具轉移性)。

軟體工程的觀點 10.10

有些使用物件導向程式設計者，認爲夥伴關係會破壞資訊隱藏性，且會降低物件導向程式設計方法的價值。接下來，我們會舉出幾個妥善使用夥伴關係的範例。

用夥伴函式修改類別的 `private` 資料

圖 10.15 的例子用 `friend` 函式 `setX` 修改類別 `Count` 的 `private` 資料成員 `x`。此處夥伴宣告 (第 9 行) 依慣例放在類別定義的最前端，甚至在 `public` 成員函式宣告之前；但它其實可出現在類別定義中任何位置。

```
1  // Fig. 10.15: fig10_15.cpp
2  // Friends can access private members of a class.
3  #include <iostream>
4  using namespace std;
5
6  // Count class definition
7  class Count
8  {
9     friend void setX( Count &, int ); // friend declaration
10 public:
11    // constructor
12    Count()
13       : x( 0 ) // initialize x to 0
14    {
15       // empty body
16    } // end constructor Count
17
18    // output x
19    void print() const
20    {
21       cout << x << endl;
22    } // end function print
23 private:
24    int x; // data member
25 }; // end class Count
26
27 // function setX can modify private data of Count
28 // because setX is declared as a friend of Count (line 9)
```

圖 10.15　夥伴函式能夠存取類別的 `private` 成員

```
29  void setX( Count &c, int val )
30  {
31     c.x = val; // allowed because setX is a friend of Count
32  } // end function setX
33
34  int main()
35  {
36     Count counter; // create Count object
37
38     cout << "counter.x after instantiation: ";
39     counter.print();
40
41     setX( counter, 8 ); // set x using a friend function
42     cout << "counter.x after call to setX friend function: ";
43     counter.print();
44  } // end main
```

```
counter.x after instantiation: 0
counter.x after call to setX friend function: 8
```

圖 10.15　夥伴函式能夠存取類別的 private 成員 (續)

請注意，函式 setX (第 29–32 行) 是一個 C 風格的獨立函式，並不是類別 Count 的成員。因此，當我們爲 counter 物件呼叫 setX 時，程式第 41 行會將 counter 當成引數傳給 setX，而不是將它用爲呼叫函式的代表 (如物件名稱) 如下：

```
counter.setX( 8 );
```

假如你刪掉第 9 行的 friend 宣告，就會看到一個錯誤訊息，表示 setX 函式不能修改 Count 類別的 private 資料成員 x。

前面提過，圖 10.15 僅供示範夥伴結構之用。我們通常會將函式 setX 定義爲類別 Count 的成員函式，且將圖 10.15 的程式分割成三個檔案：

1. 標頭檔 (如 Count.h)，內含類別 Count 的定義，其中包含 friend 函式 setX 的原型。
2. 實作檔 (如 Count.cpp)，內含類別 Count 成員函式及 friend 函式 setX 的定義。
3. 包含 main 的測試程式 (如 fig10_15.cpp)。

多載的 friend 函式

多載函式亦可成爲類別的夥伴函式。每個想要成爲夥伴函式的多載函式，都必須在類別的定義中，明確宣告爲該類別的夥伴。

10.5　使用 this 指標

物件的成員函式可以操作物件的資料，但成員函式要怎麼知道它操作的是哪個物件？每個物件可以透過 **this** 指標 (C++關鍵字)，存取自己的記憶體位址。this 指標不是物件自己的一部分，也就是說，對 this 執行 sizeof 運算的結果，與對物件執行 sizeof 運算的結果不同。編譯器會將 this 指標作爲隱含的引數傳給物件的非 static 成員函式。第 10.6 節會介紹 static 類別成員，並解釋爲何 this 指標不傳入 static 成員函式。

物件會自動使用 this 指標 (截至目前爲止我們都這麼做)，也可明確使用之以參照其資料成員和成員函式。this 指標的型別，與物件的型別，以及使用 this 指標的成員函式是否宣告成 const 有關。例如，在類別 Employee 的非 const 成員函式中，this 指標的型別是 Employee *const (指向非常數 Employee 物件的常數指標)。在常數成員函式中，其型別則爲 const Employee*const (指向常數 Empolyee 物件的常數指標)。

本節的第一個例子，將說明如何隱含與明確地使用 this 指標；在本章稍後和第 11 章中，我們會舉出一些使用 this 指標的精巧範例。

隱含和明確地使用 this 指標存取物件的資料成員

圖 10.16 說明如何在程式中隱含和明確地使用 this 指標，讓類別 Test 的成員函式印出 Test 物件的 private 資料 x。

```
1  // Fig. 10.16: fig10_16.cpp
2  // Using the this pointer to refer to object members.
3  #include <iostream>
4  using namespace std;
5
6  class Test
7  {
8  public:
9     Test( int = 0 ); // default constructor
10    void print() const;
11 private:
12    int x;
13 }; // end class Test
14
15 // constructor
16 Test::Test( int value )
17    : x( value ) // initialize x to value
18 {
```

圖 10.16　使用 this 指標隱含與明確地存取物件的成員

```
19      // empty body
20   } // end constructor Test
21
22   // print x using implicit and explicit this pointers;
23   // the parentheses around *this are required
24   void Test::print() const
25   {
26      // implicitly use the this pointer to access the member x
27      cout << "        x = " << x;
28
29      // explicitly use the this pointer and the arrow operator
30      // to access the member x
31      cout << "\n  this->x = " << this->x;
32
33      // explicitly use the dereferenced this pointer and
34      // the dot operator to access the member x
35      cout << "\n(*this).x = " << ( *this ).x << endl;
36   } // end function print
37
38   int main()
39   {
40      Test testObject( 12 ); // instantiate and initialize testObject
41
42      testObject.print();
43   } // end main
```

```
         x = 12
  this->x = 12
(*this).x = 12
```

圖 10.16　使用 this 指標隱含與明確地存取物件的成員 (續)

爲了教學示範，成員函式 print (第 24-36 行) 首先隱含地使用 this 指標 (第 27 行) 印出 x，即只指定資料成員的名稱。接下來，函式 print 會利用 this 指標，以兩種不同的符號來存取 x，即箭號 (->) 運算子加上 this 指標 (第 31 行)，以及點號(.)運算子配合解參照的 this 指標 (第 35 行)。請注意，使用點號成員選擇運算子 (.) 時，*this (第 35 行) 必須加上小括號，點號運算子的優先權比 * 運算子的優先權更高，所以此處的小括號不能省略。若省略小括號，運算式*this.x 的結果會與* (this.x) 相同，這是一種語法錯誤，因爲點號運算子不能與指標一起使用。

常見的程式設計錯誤 10.7

若在指向物件的指標上使用成員選擇運算子 (.)，會產生編譯錯誤，因爲點號成員選擇運算子只能對 lvalue 使用，如物件名稱，指向物件的參照，或解參照的指標。

this 指標的另一項有趣用法，是防止物件賦值給它自己。在第 11 章我們會談到，當物件包含指向動態配置記憶體的指標時，自我賦值會造成嚴重的錯誤。

使用 this 指標進行連續函式呼叫

另一個 this 指標的用法是進行**連續成員函式呼叫 (cascaded member-function calls)**，在同一個敘述中進行多個函式呼叫 (如圖 10.19 中第 12 行)。圖 10.17-10.19 的程式對類別 Time 的 set 函式，如 setTime、setHour、setMinute 和 setSecond 進行修改，使它們傳回 Time 物件的參照，以便進行連續成員函式呼叫。請注意在圖 10.18 中，這些函式的本體最後一個敘述都是傳回*this (第 22、29、36 和 43 行)，且傳回值的型別爲 Time &。

```
1  // Fig. 10.17: Time.h
2  // Cascading member function calls.
3
4  // Time class definition.
5  // Member functions defined in Time.cpp.
6  #ifndef TIME_H
7  #define TIME_H
8
9  class Time
10 {
11 public:
12    Time( int = 0, int = 0, int = 0 ); // default constructor
13
14    // set functions (the Time & return types enable cascading)
15    Time &setTime( int, int, int ); // set hour, minute, second
16    Time &setHour( int ); // set hour
17    Time &setMinute( int ); // set minute
18    Time &setSecond( int ); // set second
19
20    // get functions (normally declared const)
21    int getHour() const; // return hour
22    int getMinute() const; // return minute
23    int getSecond() const; // return second
24
25    // print functions (normally declared const)
26    void printUniversal() const; // print universal time
27    void printStandard() const; // print standard time
28 private:
29    int hour; // 0 - 23 (24-hour clock format)
30    int minute; // 0 - 59
31    int second; // 0 - 59
32 }; // end class Time
33
34 #endif
```

圖 10.17　修改 Time 類別的定義，進行連續成員函式呼叫

```
1  // Fig. 10.18: Time.cpp
2  // Time class member-function definitions.
3  #include <iostream>
4  #include <iomanip>
5  #include "Time.h" // Time class definition
6  using namespace std;
7
8  // constructor function to initialize private data;
9  // calls member function setTime to set variables;
10 // default values are 0 (see class definition)
11 Time::Time( int hr, int min, int sec )
12 {
13    setTime( hr, min, sec );
14 } // end Time constructor
15
16 // set values of hour, minute, and second
17 Time &Time::setTime( int h, int m, int s ) // note Time & return
18 {
19    setHour( h );
20    setMinute( m );
21    setSecond( s );
22    return *this; // enables cascading
23 } // end function setTime
24
25 // set hour value
26 Time &Time::setHour( int h ) // note Time & return
27 {
28    hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
29    return *this; // enables cascading
30 } // end function setHour
31
32 // set minute value
33 Time &Time::setMinute( int m ) // note Time & return
34 {
35    minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
36    return *this; // enables cascading
37 } // end function setMinute
38
39 // set second value
40 Time &Time::setSecond( int s ) // note Time & return
41 {
42    second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
43    return *this; // enables cascading
44 } // end function setSecond
45
46 // get hour value
47 int Time::getHour() const
48 {
49    return hour;
50 } // end function getHour
51
52 // get minute value
```

圖 10.18　修改 `Time` 類別成員函式的定義，進行連續成員函式呼叫

```
53  int Time::getMinute() const
54  {
55     return minute;
56  } // end function getMinute
57
58  // get second value
59  int Time::getSecond() const
60  {
61     return second;
62  } // end function getSecond
63
64  // print Time in universal-time format (HH:MM:SS)
65  void Time::printUniversal() const
66  {
67     cout << setfill( '0' ) << setw( 2 ) << hour << ":"
68        << setw( 2 ) << minute << ":" << setw( 2 ) << second;
69  } // end function printUniversal
70
71  // print Time in standard-time format (HH:MM:SS AM or PM)
72  void Time::printStandard() const
73  {
74     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
75        << ":" << setfill( '0' ) << setw( 2 ) << minute
76        << ":" << setw( 2 ) << second << ( hour < 12 ? " AM" : " PM" );
77  } // end function printStandard
```

圖 10.18　修改 `Time` 類別成員函式的定義，進行連續成員函式呼叫 (續)

```
1  // Fig. 10.19: fig10_19.cpp
2  // Cascading member-function calls with the this pointer.
3  #include <iostream>
4  #include "Time.h" // Time class definition
5  using namespace std;
6
7  int main()
8  {
9     Time t; // create Time object
10
11    // cascaded function calls
12    t.setHour( 18 ).setMinute( 30 ).setSecond( 22 );
13
14    // output time in universal and standard formats
15    cout << "Universal time: ";
16    t.printUniversal();
17
18    cout << "\nStandard time: ";
19    t.printStandard();
20
21    cout << "\n\nNew standard time: ";
22
```

圖 10.19　利用 `this` 指標進行連續成員函式呼叫

```
23      // cascaded function calls
24      t.setTime( 20, 20, 20 ).printStandard();
25      cout << endl;
26   } // end main
```

```
Universal time: 18:30:22
Standard time: 6:30:22 PM

New standard time: 8:20:20 PM
```

圖 10.19　利用 this 指標進行連續成員函式呼叫 (續)

圖 10.19 的程式產生 Time 物件 t (第 9 行)，並於第 12 和 24 行用它進行連續成員函式呼叫。讀者可能要問，爲何這裡可將*this 當作參照傳回？點號運算子 (.) 的結合性是由左至右，故第 12 行首先計算 t.setHour(18)的值，再傳回物件 t 的參照，作爲函式呼叫的傳回值。然後，剩餘的運算式會成爲

```
t.setMinute( 30 ).setSecond( 22 );
```

函式呼叫 t.setMinute(30)執行後，傳回物件 t 的參照。然後，剩下的運算式會成爲

```
t.setSecond( 22 );
```

程式第 24 行也使用連續呼叫的機制。呼叫必須按第 24 行的順序排列，因爲依 printStandard 在類別中的定義，該函式並不傳回指向 t 的參照。若我們在第 24 行在 setTime 之前呼叫 printStandard，會產生編譯錯誤。第 11 章會提供幾個使用連續函式呼叫的實用範例，其中之一是在單一敘述中對 cout 連續使用<<運算子輸出多個數值。

10.6 static 類別成員

一般來說，類別的每個物件都擁有其資料成員的副本。在某些情況下，類別的所有物件必須共享某個變數的單一副本。此時，可使用 **static 資料成員 (static data member)**。這種變數可以用來表示全類別共用的資訊；這種資訊是該類別所有物件一致的特性，而非物件的個別屬性。請回憶我們在第七章中的類別 GradeBook 就使用 static 資料成員儲存常數，代表所有 GradeBook 物件能儲存的成績個數。

屬於全類別共用的資料

我們再用一個例子說明使用「全類別」static 資料的動機與理由。假設在電玩遊戲中，有 Martian 和其他的太空生物等角色。當 Martian 知道有五個以上的 Martian 時，每個 Martian 就會變得很勇敢，並且想要攻擊其它的太空生物。如果少於五個 Martian，則每個 Martian 就會變得十分膽小。所以，每個 Martian 都必須知道 Martian 的數目，以 martianCount 的值表示。我們可以賦予類別 Martian 的每個物件一個 martianCount 資料成員。如此，每個 Martian 都會有一個該資料成員的副本。每當建立新的 Martian 物件時，就必須更新所有 Martian 物件的資料成員 martianCount。這會讓每個 Martian 物件必須處理 (存取) 記憶體中所有其它的 Martian 物件。如此一來，這些重複的副本就會浪費空間及更新每一個 martianCount 副本的時間。比較好的作法是將 martianCount 宣告爲 static，讓 martianCount 變成全類別共用的資料。每個 Martian 都可以存取 martianCount 的值，好像它是每個 Martian 的資料成員一樣，但是 C++只維護一份靜態變數 martianCount 的副本。這樣就可以節省空間。我們透過 Martian 建構子，遞增靜態變數 martianCount 的值，且用 Martian 解構子減少 martianCount 的值。這樣可以節省時間。因爲該類別只有一份副本，所以不需對每個 Martian 物件遞增 martianCount 個別副本的值。

增進效能的小技巧 10.3

當一份資料就可滿足類別所有物件時，請使用 static 資料成員。

使用域以及 static 資料成員的初始化

類別的 static 資料成員和全域變數很像，但它具有類別使用域。此外，static 成員可以宣告成 public、private 或 protected。基本型別的 static 資料成員的預設初始值爲 0。假如你想要設定不同的初始值，static 資料成員只能初始化一次。型別爲 int 或 enum 的 const static 資料成員可以在類別定義中宣告時設定其初始值。其它的 static 資料成員則必須在全域使用域 (global namespace scope) 中定義 (即類別定義本體之外)，而且只能在定義中設定其初始值。若 static 資料成員的型別爲類別，且該類別有預設建構子，則此 static 資料成員不需設定初始值，因爲程式會自動呼叫其預設建構子。

存取 static 資料成員

類別的 private 和 protected static 成員，只能透過類別的 public 成員函式或類別的夥伴函式存取。即使某類別沒有任何物件存在時，該類別的 static 成員仍然存在。類別沒有物件存在時，若要存取其 public static 成員，只要在成員名稱前面寫出類別名稱後接二元使用域解析運算子 (::) 即可。例如，如果先前變數 martianCount 是 public，則程式沒有建立 Martian 物件時，可以用 Martian::martianCount 運算式存取該變數。(當然，一般來說我們不鼓勵使用 public 資料成員)。

我們還可以提供 **public static 成員函式 (static member function)**，如此就可在沒有物件時，用類別名稱、二元使用域解析運算子、以及函式的名稱存取 static 的 private 和 protected 資料成員。static 成員函式可視爲整個類別，而非單一物件，對外界提供的服務。

軟體工程的觀點 10.11

若某類別沒有產生任何物件，則 static 資料成員和 static 成員函式仍然可以單獨使用。

static 資料成員的使用範例

圖 10.20-10.22 的程式示範用 private static 資料成員 count (圖 10.20 第 25 行) 及 public static 成員函式 getCount (圖 10.20 第 19 行)。在圖 10.21 中的第 8 行，程式在全域使用域定義 count 資料成員並將初始值設爲零，而第 12-15 行則定義 static 成員函式 getCount。請注意在第 8 或第 12 行中，程式都沒有寫出關鍵字 static，但它們都用到 static 類別成員。在全域使用域中對某項目使用 static 關鍵字時，該項目就只能在該檔案中使用。類別的 static 成員必須讓外部程式得以使用，所以我們只能在.h 中寫出 static。資料成員 count 會維護已經建立的類別 Employee 的物件個數。當類別 Employee 物件存在時，Employee 物件的任何成員函式都可存取成員 count；在圖 10.21 中，第 22 行的建構子和第 32 行的解構子都會參照 count。

常見的程式設計錯誤 10.8

在全域使用域的 static 資料成員定義中加上關鍵字 static，會造成編譯錯誤。

```
1  // Fig. 10.20: Employee.h
2  // Employee class definition with a static data member to
3  // track the number of Employee objects in memory
4  #ifndef EMPLOYEE_H
5  #define EMPLOYEE_H
6
7  #include <string>
8  using namespace std;
9
10 class Employee
11 {
12 public:
13    Employee( const string &, const string & ); // constructor
14    ~Employee(); // destructor
15    string getFirstName() const; // return first name
16    string getLastName() const; // return last name
17
18    // static member function
19    static int getCount(); // return number of objects instantiated
20 private:
21    string firstName;
22    string lastName;
23
24    // static data
25    static int count; // number of objects instantiated
26 }; // end class Employee
27
28 #endif
```

圖 10.20　修改過的 `Empolyee` 類別定義，用 `static` 資料成員記錄記憶體中 `Empolyee` 物件的個數

```
1  // Fig. 10.21: Employee.cpp
2  // Employee class member-function definitions.
3  #include <iostream>
4  #include "Employee.h" // Employee class definition
5  using namespace std;
6
7  // define and initialize static data member at global namespace scope
8  int Employee::count = 0; // cannot include keyword static
9
10 // define static member function that returns number of
11 // Employee objects instantiated (declared static in Employee.h)
12 int Employee::getCount()
13 {
14    return count;
15 } // end static function getCount
16
17 // constructor initializes non-static data members and
18 // increments static data member count
19 Employee::Employee( const string &first, const string &last )
```

圖 10.21　`Employee` 類別成員函式的定義

```
20     : firstName( first ), lastName( last )
21  {
22     ++count; // increment static count of employees
23     cout << "Employee constructor for " << firstName
24        << ' ' << lastName << " called." << endl;
25  } // end Employee constructor
26
27  // destructor deallocates dynamically allocated memory
28  Employee::~Employee()
29  {
30     cout << "~Employee() called for " << firstName
31        << ' ' << lastName << endl;
32     --count; // decrement static count of employees
33  } // end ~Employee destructor
34
35  // return first name of employee
36  string Employee::getFirstName() const
37  {
38     return firstName; // return copy of first name
39  } // end function getFirstName
40
41  // return last name of employee
42  string Employee::getLastName() const
43  {
44     return lastName; // return copy of last name
45  } // end function getLastName
```

圖 10.21 Employee 類別成員函式的定義 (續)

圖 10.22 使用 static 成員函式 getCount 判斷程式陸續產生的 Employee 物件個數。程式在尚未產生任何 Employee 物件時 (第 12 行)、產生兩個 Employee 物件後 (第 23 行)，以及清除兩個 Employee 物件後 (第 34 行)，都會呼叫 Employee::getCount()函式。main 的第 16-29 行定義了巢狀使用域。我們曾說過，在定義區域變數的使用域之中，該區域變數都是存在的。在本範例中，我們在第 17-18 行的巢狀使用域中建立了兩個 Employee 物件。當每個建構子執行時，都會遞增 Employee 的 static 資料成員 count。在程式抵達第 29 行，Employee 物件被清除時，會執行每個物件的解構子，將 Employee 的 static 資料成員 count 遞減。

```
1  // Fig. 10.22: fig10_22.cpp
2  // static data member tracking the number of objects of a class.
3  #include <iostream>
4  #include "Employee.h" // Employee class definition
5  using namespace std;
6
7  int main()
8  {
```

圖 10.22 記錄類別物件個數的 static 資料成員

```
 9      // no objects exist; use class name and binary scope resolution
10      // operator to access static member function getCount
11      cout << "Number of employees before instantiation of any objects is "
12         << Employee::getCount() << endl; // use class name
13
14      // the following scope creates and destroys
15      // Employee objects before main terminates
16      {
17         Employee e1( "Susan", "Baker" );
18         Employee e2( "Robert", "Jones" );
19
20         // two objects exist; call static member function getCount again
21         // using the class name and the binary scope resolution operator
22         cout << "Number of employees after objects are instantiated is "
23            << Employee::getCount();
24
25         cout << "\n\nEmployee 1: "
26            << e1.getFirstName() << " " << e1.getLastName()
27            << "\nEmployee 2: "
28            << e2.getFirstName() << " " << e2.getLastName() << "\n\n";
29      } // end nested scope in main
30
31      // no objects exist, so call static member function getCount again
32      // using the class name and the binary scope resolution operator
33      cout << "\nNumber of employees after objects are deleted is "
34         << Employee::getCount() << endl;
35   } // end main
```

```
Number of employees before instantiation of any objects is 0
Employee constructor for Susan Baker called.
Employee constructor for Robert Jones called.
Number of employees after objects are instantiated is 2

Employee 1: Susan Baker
Employee 2: Robert Jones

~Employee() called for Robert Jones
~Employee() called for Susan Baker

Number of employees after objects are deleted is 0
```

圖 10.22　記錄類別物件個數的 `static` 資料成員 (續)

成員函式若不存取類別的非 `static` 資料成員或成員函式，則應宣告爲 `static`。`static` 成員函式沒有 `this` 指標 (這與非 `static` 成員函式不同)，因爲 `static` 資料成員和成員函式獨立存在於任何類別物件之外。`this` 指標必需指到類別的某特定物件，但 `static` 成員函式被呼叫時，記憶體中可能尚未產生任何類別的物件。

常見的程式設計錯誤 10.9

在 `static` 成員函式中使用 `this` 指標，會造成編譯錯誤。

常見的程式設計錯誤 10.10

將 static 成員函式宣告成 const，會造成編譯錯誤。修飾子 const 表示，該函式不能對其操作的物件進行修改，但 static 成員函式存在和獨立運作於類別的任何物件上。

10.7 資料抽象化與資訊隱藏

一般來說，外界程式看不到類別的實作內容，此即爲**資訊隱藏(information hiding)**。爲說明這個概念，我們以 6.11 節介紹的堆疊資料結構做爲例子。堆疊是一種後進先出 (last-in, first-out，LIFO) 的資料結構，也就是說，最後推入 (加入) 堆疊的項目會最先從堆疊取出 (移除)。

我們可以利用陣列實現堆疊的行爲，或是其它資料結構如鏈結串列 (linked list)。(我們在第 14 章和第 20 章中介紹堆疊。)使用堆疊的外部程式並不需知道堆疊如何實作。對外部程式來說，只需知道推入資料項目時，是以後入先出的順序排列即可。外部程式只需關心堆疊提供了哪些功能，而非這些功能如何實作。這種觀念稱爲**資料抽象化 (data abstraction)**。雖然你可能會了解類別的實作細節，但在設計上應該避免撰寫出依賴實作細節的程式。這樣一來，就能在不影響系統其它部分的情況下，以新版本的類別 (例如實作堆疊之推入及彈出操作的類別) 替換掉舊的類別。只要類別的 public 服務保持不變 (意思是說，所有原 public 成員函式的標頭在新類別定義中仍保證不變)，系統的其它部分就不會被影響。

抽象資料型別

許多程式語言比較強調「動作」。在這種語言中，資料只是爲了支援程式要進行的動作而存在，因此資料不如動作有趣，資料是「原始的」。它們只支援少數內建資料型別，而且程式設計師不太容易自定型別。C++和一些具有物件導向風格的程式設計語言在對於資料的觀點上，便和傳統程式語言有所不同。在 C++中，物件導向程式設計的主要行爲是產生新型別 (即類別)，並表達這些型別物件之間的互動關係。爲建立這種著重於資料關係的程式語言，我們必須對資料的表示法進行形式化。在這裡我們所要介紹表示法是**抽象資料型別 (abstract data types，ADTs)**，它對程式開發的流程有所幫助。

何謂抽象資料型別呢？請考慮 int 這個型別，一般我們把它看作數學中的整數。不過，int 其實是整數的抽象表示法。跟數學上的整數不同，在 32 bit 電腦上，int 的範圍是-2,147,483,648 到+2,147,483,647。如果計算的結果超出這個範圍，會產生溢位

(overflow) 錯誤，且在不同機器上會有不同的行爲。例如，程式可能會「安靜地」產生一個錯誤的結果，顯示超過 `int` 可表示範圍的數字 (通常稱爲**算數溢位，arithmetic overflow**)。數學上的整數就沒有這個問題。所以，電腦上的 `int` 只是一種很接近眞實世界裡的整數概念。

像 `int`、`double`、`char` 和其它的型別，也都是抽象化資料型態的例子。抽象化資料型別就是用電腦系統來表達眞實世界到某種層次的方法。

抽象資料型別包含兩種概念，**資料表示法 (data representation)**，以及可對這些資料執行的**操作 (operations)**。例如，C++中的 `int` 包含一個整數值 (資料)，並提供加法、減法、乘法、除法以及模數等操作－除以零的行爲未定義。這些操作的行爲與底層電腦系統的一些特性有關，如字 (word) 的寬度。另外一個例子是負整數，一般情況下其操作和資料表示都沒有問題，但取平方根的操作的行爲則未定義。在 C++中，你可以用類別來實作抽象資料型別與其提供的服務。例如，爲實作堆疊抽象資料型別，我們會在第 14 章與第 20 章中實作自己的堆疊類別；我們也會在第 22 章「標準樣板函式庫」中研究標準函式庫提供的 `stack` 類別。

軟體工程的觀點 10.12

程式設計師可以利用類別的機制來建立新的型別。這些新的型別可以設計的和基礎型別一樣容易使用。因此，C++是可擴充的程式語言。用這些新的資料型別擴充語言的功能很簡單，但語言本身的基本觀念不會因此改變。

佇列抽象資料型別

我們每一個人總是在排隊。一排等待中的隊伍也可稱之爲**「佇列」(queue)**。電腦系統內部也使用許多的等待路線，所以我們需要撰寫程式來模擬佇列的功能。佇列是另一個抽象化資料型別的範例。

對外界程式來說，佇列行爲的定義非常明確。外部程式可以叫用佇列的**存入佇列 (enqueue)** 操作，一次放入一個項目，或用**取出佇列 (dequeue)** 操作一次拿出一個項目。理論上，佇列可以無限長，但實際上長度有限。佇列項目的取用是依據**先進先出 (first-in, first out；FIFO)** 的順序；第一個放入的項目就是第一個可以取用的項目。

佇列能記錄目前正在等待的項目，但外界看不到內部實作此功能的資料結構。它提供一組操作佇列的方法，分別是存入佇列 (enqueue) 和取出佇列 (dequeue)。使用者並不關心佇列的實作方式。只要佇列提供它所宣稱的服務即可。將一個新項目存入佇列時，佇列應接受該項目，並按照內部規定，放在某個先進先出的資料結構中。想從佇列

的前端取出下一個項目時，佇列應按 FIFO 的順序 (意思是說，在佇列等待最久的項目，就是下一個取出佇列動作 dequeue 所要傳回的項目)，從它的內部結構中移出這個項目，傳送給客戶端。

佇列的抽象化資料型別能夠保證其內部資料結構的一致性。使用者不能直接處理該資料結構。只有佇列的成員函式才有權存取其內部資料。使用者只能用允許的操作取得資料；若試圖使用抽象資料型別的公開介面未提供的操作，會以適當的方式予以拒絕，如印出錯誤訊息、拋出例外 (請見第 16 章)、結束程式執行或忽略這項操作。

第 20 章會建立我們自己的佇列類別；我們也會在第 22 章研究標準函式庫的 `queue` 類別。

10.8 總結

本章介紹了數個進階的類別與資料抽象化的相關主題。你學到了透過 `const` 物件與 `const` 成員函式，防止物件被修改，強制實行最小權限原則。你也學到，透過組合，類別可以使用另一個類別的物件做爲成員。我們介紹了夥伴關係，並使用一個範例來介紹如何使用 `friend` 函式。

你學到了一個特別的指標 `this`，類別中任何非 `static` 的成員函式，都隱含此指標作爲引數，以便存取正確物件的資料成員，以及其它非 `static` 的成員函式。你也知道如何明確使用 `this` 指標，存取類別的成員以及進行連續成員函式呼叫。我們說明了爲何需要 `static` 資料成員，並示範如何在類別中使用 `static` 資料成員與成員函式。

你學到了兩個基本的物件導向程式設計觀念：資料抽象化以及資訊隱藏。最後我們討論了抽象資料型別，這是電腦系統將眞實世界表達到某種精確度的方式。

在第 11 章中，我們將繼續討論類別和物件，示範如何在物件上運用 C++的運算子，也就是運算子多載。例如，我們將會見到如何「多載」運算子<<，讓它可以不需要明確地使用重覆結構，即能輸出一個完整的陣列。

摘要

10.2 `const` 物件和 `const` 成員函式

- 關鍵字 `const` 可用來指稱物件不可被修改；程式若試圖修改物件內容，會產生編譯錯誤。
- C++編譯器不允許在 `const` 物件上呼叫非 `const` 成員函式。
- 定義爲 `const` 的成員函式，若嘗試修改物件的資料成員，會造成編譯錯誤。

- 成員函式若爲 const，則必須在原型和定義中都寫出 const。
- const 物件需於初始化時賦值。
- 建構子和解構子不能宣告爲 const。
- const 資料成員與參照資料成員一定要用成員初始值串列進行初始化。

10.3 組合：將物件當作類別成員

- 類別可以擁有其它類別的物件作爲成員；這個概念稱爲組合。
- 成員物件按照它們在類別定義中的順序進行建構，並且比包含它們的類別物件更先建立。
- 未提供成員初始值給成員物件時，系統會自動呼叫該物件的預設建構子。

10.4 夥伴函式與類別

- 類別的夥伴函式 (friend function) 定義於類別範圍的外部，它可以存取類別的所有成員。獨立的函式或整個類別可以宣告成另一個類別的夥伴。
- 夥伴宣告可以寫在類別的任何地方。
- 夥伴關係沒有對稱性，也沒有轉移性。

10.5 使用 this 指標

- 所有物件都能透過 this 指標存取自己的位址空間。
- this 指標不是物件自己的一部分，意思是說，對 this 執行 sizeof 運算的結果，與對物件執行 sizeof 運算的結果不一樣。
- 編譯器會將 this 指標作爲隱含的引數傳給物件的非 static 成員函式。
- 物件會自動使用 this 指標 (截至目前爲止我們都這麼做)，也可明確使用之以參照其資料成員和成員函式。
- 透過 this 指標，我們能在單一敘述中藉連續成員函式呼叫多個函式。

10.6 static 類別成員

- static 資料成員代表全類別共用的資訊，意思是類別所有物件共通的特性，而非特定物件的個別屬性。
- static 資料成員具有類別使用域，且可宣告爲 public、private 或 protected。
- 即使某類別沒有任何物件存在時，該類別的 static 成員仍然存在。
- 類別沒有物件存在時，若要存取其 public static 成員，只要在成員名稱前面寫出類別名稱後接二元使用域解析運算子 (::) 即可。

- 成員函式若不存取類別的非 `static` 資料成員或成員函式，則應宣告爲 `static`。`static` 成員函式沒有 `this` 指標 (這與非 `static` 成員函式不同)，因爲 `static` 資料成員和成員函式獨立存在於任何類別物件之外。

10.7 資料抽象化與資訊隱藏

- 抽象資料型別是電腦系統將眞實世界表達到某種精確度的方式。
- 抽象資料型別包含兩種概念：資料表示法，以及可對這些資料執行的操作。

術語

抽象資料型別 (abstract data type，ADT)
算術溢位 (arithmetic overflow)
連續成員函式呼叫 (cascaded member-function calls)
組合 (composition)
資料抽象化 (data abstraction)
資料表示法 (data representation)
取出佇列，佇列運算 (dequeue，queue operation)
存入佇列，佇列運算 (enqueue，queue operation)
先進先出 (FIFO) (first-in, first-out，FIFO)
夥伴函式 (friend function)
「有一個」關係 (has-a relationship)
所屬物件 (host object)
資訊隱藏 (information hiding)
成員初始值 (member initializer)
成員初始值串列 (member initializer list)
成員初始值語法 (member initializer syntax)
成員物件 (member object)
成員物件建構子 (member object constructor)
對 ADT 的操作 (operations in an ADT)
佇列 (queue)
佇列抽象資料型別 (queue abstract data type)
`static` 資料成員 (static data member)
`static` 成員函式 (static member function)
`this` 指標 (this pointer)

自我測驗

10.1 填空題

a) 我們必須用_____設定類別常數成員的初始值。
b) 非類別成員的函式必須宣告成類別的_____，才能夠存取該類別的 `private` 資料成員。
c) 常數物件必須被_____；一旦建立之後，便不能夠再修改。
d) _____資料成員代表全類別共用的資訊。
e) 物件的非 `static` 成員函式可藉_____指標存取自己。
f) 關鍵字_____可以用來指出，一個物件或變數不能夠更改。
g) 如果沒有爲類別的成員物件，提供成員初始值串列，則程式會呼叫該物件的_____。

h) 如果成員函式沒有存取_____類別成員，則可宣告爲 static。

i) static 成員物件建立時機是在包含它的物件建立_____。

10.2 請找出以下程式碼的錯誤，並說明如何更正：

```
class Example
{
public:
   Example( int y = 10 )
      : data( y )
   {
      // empty body
   } // end Example constructor

   int getIncrementedData() const
   int getIncrementedData() const
   {
      return data++;
   } // end function getIncrementedData

   static int getCount()
   {
      cout << "Data is " << data << endl;
      return count;
   } // end function getCount
private:
   int data;
   static int count;
}; // end class Example
```

自我測驗解答

10.1 a) 成員初始值。b) 夥伴。c) 初始化。d) static。e) this。f) const。g) 預設建構子。h) 非 static。i) 之前。

10.2 錯誤：類別 Example 的定義有兩個錯誤。第一個錯誤發生在函式 getIncrementedData 此函式宣告爲 const，但它會修改物件的內容。

更正：要更正第一個錯誤，須將 const 關鍵字從 getIncrementedData 的定義中移除

錯誤：第二個錯誤發生在函式 getCount。此函式宣告爲 static，因此不能存取類別中任何的非 static 成員。

更正：要更正第二個錯誤，須將輸出敘述式從 getCount 定義中移除。

習題

10.3 請說明夥伴關係的概念，以及它的缺點。

10.4 正確的類別 Time 定義是否可包含以下兩個建構子？若答案爲否，請說明理由。

```
Time( int h = 0, int m = 0, int s = 0 );
Time();
```

10.5 若爲建構子或解構子指定傳回值的型別 (即使是 void)，會發生什麼情形？

10.6 (修改 Date 類別) 請修改圖 10.10 的 Date 類別，讓它具有以下功能：

a) 以多種格式來輸出日期，例如
```
DDD YYYY
MM/DD/YY
June 14, 1992
```
b) 使用多載建構子產生 Date 物件，並按 a) 的格式設定初始值。

c) 請建立一個 Date 建構子，使用 <ctime> 定義的標準程式庫函式，讀取系統時間，用來設定 Date 的成員。請參考你手上編譯器的文件，或參考 www.cplusplus.com/ref/ctime/index.html 網站，取得 <ctime> 標頭檔的相關資訊。

在第 11 章中，我們會建立運算子，測試兩個日期是否相等，並且比較兩個日期決定其前後順序。

10.7 (SavingsAccount 類別) 建立 SavingsAccount 類別，用 static 資料成員 annualInterestRate 儲存帳戶的年利率。類別的每個物件都包含一個 private 資料成員 savingsBalance，指出每位帳戶內的目前存款餘額。請提供成員函式 calculateMonthlyInterest，計算每月的存款利息，其計算方式是將 balance 乘以 annualInterestRate 再除以 12；計算後的利息必須再加上 savingsBalance。請提供 static 成員函式 modifyInterestRate，用來設定 static annualInterestRate (年利率) 的新值。請撰寫一個程式來測試 SavingsAccount，產生兩個不同的 SavingsAccount 類別的物件：saver1 和 saver2，其餘額分別爲 $2000.00 和$3000.00。把 annualInterestRate 的值設爲 3%，計算每個月的利息，並且印出每位存款者的新餘額。接下來，請將 annualInterestRate 設定爲 4%，計算下個月的利息，並印出每個存戶的新存款。

10.8 (IntegerSet 類別) 建立類別 IntegerSet，其物件能儲存 0 至 100 範圍內的整數集合。用 bool 型別的 vector 表示集合。如果整數 i 位在該集合內，則陣列元素 a[i]的值爲 true。如果整數 j 不在此集合中，則陣列元素 a[j]爲 false。預設建構子會將集合的初始值設成空集合，意思是說，其元素全部爲 false。

請爲一般的集合運算，提供對應的成員函式。舉例來說，請提供 unionOfSets 成員函式，用來建立兩個集合的聯集 (意思是，如果目前兩個集合的元素之中任一個或兩個都是 true，則請將聯集的元素也設定爲 true；如果目前兩個集合的元素都是 false，則請將聯集的元素都設定爲 false。)

提供 intersectionOfSets 成員函式，建立兩個集合的交集 (意思是，如果目前兩個集合的元素之中任一個或兩個元素都是 false，則交集的元素也都必須設定爲 false；如果目前兩個集合的元素都是 true，則將交集的元素都設定爲 true。)

提供 insertElement 成員函式，將一個新的整數 k 加入集合中 (即將 a[k]設定爲 true)。請提供 deleteElement 成員函式，將整數 m 從集合中刪除 (即將 a[m]設定爲 0)。

提供一個 printSet 成員函式，將集合中的數字，以空白隔開後列印出來。請列印出現在集合中的元素即可 (即所在 vector 位置的值是 true 的元素)。如果遇到空集合，請印出---。

提供 isEqualTo 成員函式，判斷二個集合是否相等。

請提供接受整數陣列和陣列大小的額外建構子，並用陣列設定集合物件的初始值。

現在，撰寫一個驅動程式測試 IntegerSet 類別。建立幾個 IntegerSet 物件，並測試所有成員函式是否能正確運作。

10.9 (修改 Time 類別) 我們可以讓圖 10.17-10.18 定義的 Time 類別，用自午夜開始計時的秒數，作爲類別內部的時間表示法，代替 hour、minute 和 second 三個整數。用戶端可以使用相同的 public 函式，並獲得相同的結果。請修改圖 10.17 的類別 Time，依據從午夜開始計時的秒數實作類別 Time，並說明此類別的使用者不會察覺功能上的改變。[請注意：本習題優雅地示範隱藏實作的好處。]

10.10 (洗牌和發牌) 建立一個程式，能將一副撲克牌洗牌和發牌。本程式應該包含類別 Card、類別 DeckOfCards 和一個測試程式。Card 類別應該提供：

a) int 型別的資料成員 face 和 suit。

b) 一個建構子，接收代表花色和點數的兩個 int 值，並用它們來初始化資料成員。

c) 兩個字串型別的 static 陣列，分別代表花色和點數。

d) toString 函式，將 Card 以字串“face of suit”的格式回傳。你可以用+運算子來串接字串。

DeckOfCards 類別應該包含：

a) Card 型別的 vector，其名稱爲 deck，用來儲存 Card。

b) 整數 currentCard，代表下一張要發的牌。

c) 預設建構子，用來初始化 deck 中的 Card。在每張 Card 建立並初始化之後，建構子應該使用 vector 函式 push_back 將這張牌加入 vector 的尾端。52 張牌都應該執行上述動作。

d) shuffle 函式，用來洗牌。洗牌演算法應該對 vector 中的每張牌執行一次：針對每張牌，隨機選另一張牌與它交換。

e) dealCard 函式會回傳這副牌中的下一張 Card。

f) moreCards 函式會回傳 bool 值，指出是否還有牌要發。

測試程式應該建立一個 DeckOfCards 物件，將一副牌洗牌，並發出 52 張牌。

10.11 (洗牌和發牌) 修改習題 10.10 的程式，使發牌函式能夠一次發出 5 張牌。請撰寫函式完成下列工作：

a) 判斷這 5 張牌裡是否有對子。
b) 判斷這 5 張牌裡是否有雙對子。
c) 判斷這 5 張牌裡是否有三條。(如三張傑克)
d) 判斷這 5 張牌裡是否有四梅。(如四張 **A**)
e) 判斷這 5 張牌裡是否有同花。(即五張同樣花色的牌)
f) 判斷這 5 張牌裡是否有順子。(即五張連續的牌)

洗牌和發牌的專題

10.12 (洗牌和發牌) 使用習題 10.11 所發展的函式撰寫一個程式，讓它能夠發出兩家，每家五張牌的一局牌，然後檢視每付牌，並且判斷哪一付牌較好。

10.13 (洗牌和發牌) 修改習題 10.12 的程式，模仿莊家發牌。五張牌發出時必須面朝下，玩家無法看到牌。然後，程式應檢視莊家手上的牌，按照這付牌，使莊家再抽一、二或三張牌，取代手上不需要的廢牌。然後，程式應再檢視莊家手上的牌。

10.14 (洗牌和發牌) 請修改習題 10.13 的程式，自動處理莊家的牌，但參加者可決定要換掉手中的哪幾張牌。然後，程式應檢視兩付牌，判定誰贏。請使用這個新程式，與電腦進行 20 場比賽。誰贏得比較多場次的比賽，你或電腦？讓你的朋友與電腦進行 20 場比賽。誰贏的場次較多？請依據比賽的結果，適度修改這個撲克牌遊戲程式。再玩 20 場遊戲。修改後的遊戲程式有比較好的成績嗎？

進階習題

10.15 (飛航管理專案) 根據美國國家航管員協會 (National Air Traffic Controllers Association，`www.natca.org/mediacenter/bythenumbers.msp`) 的資料，美國每天有超過 87,000 個客運和貨運航班，以長期趨勢來說，隨著人口的增加，空中交通將會越來越繁忙。在這樣的趨勢下，飛航管理的挑戰性也越來越高，飛航管理的工作包含了監控航班，引導飛行員以確保空中安全。

在本習題中，你將會建立一個 `Flight` 類別，應用在簡單的飛航管理模擬器中。程式中的 `main` 函式會執行飛航管理的工作。你可以參考以下網站

`www.howstuffworks.com/air-traffic-control.htm`

研究飛航管理系統的運作方法。找出在飛航管理系統中，`Flight` 應該有哪些主要屬性。思考一下，飛機從停在機場的登機門開始，直到抵達目的爲止，會經過哪些不同的狀態—停泊、滑行、等候起飛、起飛、爬升...。使用 `FlightStatus` 列舉型別來表示這些狀態。而屬性可能包括了飛機的廠牌型號、目前速度、目前高度、方向、運載量、起

飛時間、預估抵達時間、起點和終點。起點和終點應該用三個字母的標準機場代碼來表示，例如 BOS 代表 Boston，LAX 代表 Los Angeles (你可以在 world-airport-codes.com 找到這些代碼)。提供 set 和 get 函式來處理這些和其他你找到的屬性。接下來，找出類別的行爲，將它們實作爲類別的函式。行爲包括了 changeAltitude、reduceSpeed 和 beginLandingApproach 等等。Flight 建構子應該初始化 Flight 的屬性。你應該寫一個 toString 函式，將 Flight 目前的狀態轉換成 string 的形式 (例如：parked at the gate, taxiing, taking off, changing altitude 等等)。這個字串應該包含物件所有實體變數的值。

應用程式執行時，main 會印出 "Air Traffic Control Simulator" 的訊息，建立三個代表飛機 (飛行中或是正在準備飛行) 的 Flight 物件並與之互動。爲了要簡化這個程式，當呼叫物件的函式時，Flight 會在螢幕上顯示一個訊息，以確認每個動作。舉例來說，假如你呼叫 changeAltitude 函式，它應該執行：

a) 顯示一個訊息，包括航空公司、航班號碼，"changing altitude" 的字樣，目前飛航高度以及新的飛航高度。
b) 更新 status 資料成員的狀態爲 CHANGING_ALTITUDE。
c) 更新 newAltitude 資料成員的值。

在 main 中，建立一些處於不同狀態的 Flight 物件並將它們初始化，例如：有一架飛機在登機門，另一架正準備起飛，第三架正準備降落。main 函式應該傳送訊息給 Flight 物件 (呼叫函式)。當 Flight 物件收到訊息時，被呼叫的函式應該顯示一個確認訊息，像是「[航空公司] [航班號碼] changing altitude from 20000 to 25000 feet」。函式也應該適當地更新 Flight 物件的狀態資訊。例如，假如飛航控制中心傳送了一個訊息，像是(航空公司] [航班號碼] descend to 12000 feet,，程式應該執行一個函式像是 flight1.changeAltitude (12000)，然後顯示一個確認訊息，接著將資料成員 newAltitude 的值設爲 12000 [注意:假設 Flight 的 currentAltitude 資料成員會自動由飛機的高度計所設定。]