樣版

22

Behind that outside pattern the dim shapes get clearer every day. It is always the same shape, only very numerous.
—***Charlotte Perkins Gilman***

Every man of genius sees the world at a different angle from his fellows.
—***Havelock Ellis***

…our special individuality, as distinguished from our generic humanity.
—***Oliver Wendell Holmes, Sr.***

學習目標

在本章中，你將學到：

- 使用函式樣版快速建立一群相關的 (多載) 函式。
- 分辨「函式樣版」和「特殊化函式樣版」的差異。
- 使用類別樣版建立一群相關型別。
- 分辨「類別樣版」和「特殊化類別樣版」的差異。
- 將函式樣版多載。

本章綱要

22.1 簡介

本章會討論 C++中一個更強大的軟體再利用功能，也就是**樣版 (template)**。**函式樣版 (Function template)** 和**類別樣版 (class template)** 可讓你透過一段程式碼，來指定全部相關(多載)函式的範圍，稱爲**特殊化函式樣版 (function-template specializations)**，或指定全部相關類別的範圍，稱爲**特殊化類別樣版 (class-template specializations)**。這種技巧叫做**泛型程式設計 (generic programming)**。

我們只要撰寫一個適用於陣列排序的函式樣版，然後讓 C++產生個別的特殊化函式樣版，它可以處理 `int` 陣列、`float` 陣列和 `string` 陣列的排序。第 15 章曾經介紹過函式樣版。本章將進一步探討它，並介紹範例。

我們可以撰寫一個適用於堆疊類別的類別樣版，然後讓 C++產生個別的特殊化類別樣版，例如 `int` 堆疊類別、`float` 堆疊類別和 `string` 堆疊類別等。

請注意「樣版」和「特殊化樣版」之間的差異：函式樣版和類別樣版很像我們繪製圖形時所用的型版；而特殊化函式樣版和特殊化類別樣版就像是個別的複製圖片，它們雖然有相同的圖形，但可以塗上不同的顏色。本章會介紹函式樣版和類別樣版。

軟體工程的觀點 22.1

大部分的 C++編譯器都要求使用樣版的用戶端程式碼檔案中，必須包含完整的樣版定義。爲了此原因以及再利用性的考量，樣版通常都定義在標頭檔中，該標頭檔然後會 `#include` 於用戶端程式碼檔案。對類別樣版而言，這意謂著成員函式也被定義在標頭檔中。

22.2　函式樣版

多載的函式通常會在不同型別的資料上執行**類似**或**相同**的操作。如果這些操作對於每個型別都是**相同**的，則可以使用函式樣版 (function templates)，以更嚴謹、更方便地進行操作。剛開始時，你撰寫一份函式樣版的定義。依據明確地提供的引數型別，或從該函式呼叫所推論出的引數型別，編譯器會產生個別的原始碼函式 (也就是特殊化函式樣版)，以正確處理每種函式呼叫。在 C 語言中，程式可以利用前置處理指令 `#define` 所建立的**巨集 (macros)** 來執行這項工作。然而，巨集可能會產生嚴重的副作用，而且它無法讓編譯器執行型別檢查。

測試和除錯的小技巧 22.1

函式樣版，就像巨集一樣，可讓我們再利用軟體。與巨集不同的是，函式樣版有助於消除許多型別的錯誤，因爲 C++會仔細進行型別檢查。

所有**函式樣版的定義 (function-template definitions)** 都是以關鍵字 **`template`** 開始，隨後接著在**角括號 (angle bracket，< 和 >)** 中放入傳給函式樣版的**樣版參數 (template parameter)** 名單；每個樣版參數必須在前面加上關鍵字 `class` 或 **`typename`** (兩者皆可) ，如

```
template< typename T >
```

或者

```
template< class ElementType >
```

或者

```
template< typename BorderType, typename FillType >
```

函式樣版定義的「型別樣版參數」可用來指定傳給函式的引數型別、指定函式的傳回型別和宣告函式中的變數。此函式定義就跟一般函式定義一樣。用來指定函式樣版參數的關鍵字 `typename` 和 `class`，其實意思指的是「任何內建型別或使用者自訂型別」。

常見的程式設計錯誤 22.1

忘了在函式樣版的每個型別樣版參數前面加上 `class` 或 `typename` 關鍵字，是一種語法錯誤。

範例：函式樣版 `printArray`

讓我們檢視圖 22.1 中第 7–14 行的函式樣版 `printArray`。函式樣版 `printArray` 宣告 (第 7 行) 一個樣版參數 `T` (`T` 可能是任何有效的識別字) 來表示函式 `printArray` 所要列印的陣列型別；`T` 叫做**型別樣版參數 (type template parameter)** 或型別參數 (type parameter)。第 22.5 節會介紹非型別樣版參數。

```
1  // Fig. 22.1: fig22_01.cpp
2  // Function-template specializations of function template printArray.
3  #include <iostream>
4  using namespace std;
5
6  // function template printArray definition
7  template< typename T >
8  void printArray( const T * const array, int count )
9  {
10    for ( int i = 0; i < count; ++i )
11       cout << array[ i ] << " ";
12
13    cout << endl;
14 } // end function template printArray
15
16 int main()
17 {
18    const int aCount = 5; // size of array a
19    const int bCount = 7; // size of array b
20    const int cCount = 6; // size of array c
21
22    int a[ aCount ] = { 1, 2, 3, 4, 5 };
23    double b[ bCount ] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
24    char c[ cCount ] = "HELLO"; // 6th position for null
25
26    cout << "Array a contains:" << endl;
27
28    // call integer function-template specialization
29    printArray( a, aCount );
30
31    cout << "Array b contains:" << endl;
32
33    // call double function-template specialization
34    printArray( b, bCount );
35
36    cout << "Array c contains:" << endl;
37
38    // call character function-template specialization
39    printArray( c, cCount );
40 } // end main
```

圖 22.1 函式樣版 `printArray` 的特殊化函式樣版(1/2)

```
Array a contains:
1 2 3 4 5
Array b contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7
Array c contains:
H E L L O
```

圖 22.1　函式樣版 printArray 的特殊化函式樣版(2/2)

當編譯器在用戶端程式碼偵測到 printArray 函式呼叫時 (如第 29、34 行)，編譯器會透過其多載解析能力，找出最符合此函式呼叫的 printArray 函式定義。此例中，唯一具有正確參數個數的 printArray 函式，就是 printArray 函式樣版 (第 7-14 行)。請看第 29 行的函式呼叫。編譯器會比對 printArray 第一個引數的型別 (第 29 行的 int *) 與 printArray 函式樣版的第一個引數參數 (第 8 行的 const T *)，推斷出只要把型別參數 T 換成 int，引數就符合參數了。接著，編譯器在整個樣版定義中把 T 換成 int，並編譯一個特殊化的 printArray，以顯示 int 陣列的值。在圖 22.1 中，編譯器會產生兩種特殊化 printArray 函式，一個會接收 int 陣列，一個會接收 double 陣列。例如，int 型別的特殊化函式樣版如下：

```
void printArray( const int * const array, int count )
{
   for ( int i = 0; i < count; ++i )
      cout << array[ i ] << " ";

   cout << endl;
} // end function printArray
```

跟函式參數一樣，樣版參數名稱必須在樣版定義中是獨一無二的。不同函式樣版之間所用的樣版參數名稱則不一定要是獨一無二的。

圖 22.1 中示範了函式樣版 printArray (第 7–14 行)。程式一開始宣告擁有 5 個元素的 int 陣列 a、7 個元素的 double 陣列 b (第 22–23 行)。接著，程式呼叫 printArray 印出每個陣列，一個用 int *型別的引數 a (第 29 行)，一個用 double *型別的引數 b (第 34 行)。例如，第 29 行的呼叫會讓編譯器推斷 T 是 int 型別，並實體化一個 printArray 特殊化函式樣版，其型別參數 T 是 int。第 34 行的呼叫會讓編譯器推斷 T 是 double 型別，並實體化第二個 printArray 特殊化函式樣版，其型別參數 T 是 double。特別注意，若 T (第 7 行) 代表的是使用者自訂型別 (圖 22.1 中沒有)，那麼一定要撰寫該型別的多載串流插入運算子；不然第 11 行的第一個串流插入運算子就沒辦法被編譯。

常見的程式設計錯誤 22.2

如果透過使用者自訂型別來呼叫樣版，並且此樣版使用該類別物件的一些函式或運算子 (例如==、+、=)，則那些函式與算子必須針對這些使用者自訂型別進行多載。忘記多載這種運算子會產生編譯錯誤。

在本例中，有了樣版機制，你就不用以下列的函式原型，撰寫兩個不同的多載函式

```
void printArray( const int * const, int );
void printArray( const double * const, int );
void printArray( const char * const, int );
```

它們的程式碼都一樣，只有 T 的型別不同 (第 8 行所用的)。

增進效能的小技巧 22.1

雖然樣版提供了軟體再利用的優點，但是請記住，儘管我們事實上只撰寫了一次樣版，但是多個特殊化函式樣版和特殊化類別樣版會於程式中被實體化 (在編譯時)。這些副本可能會消耗可觀的記憶體空間。不過這通常不成問題，因爲由樣版產生的程式碼的大小，跟你自己撰寫數個多載函式所用者是一樣的。

22.3 將函式樣版多載

函式樣版和多載是密切相關的。從函式樣版產生的特殊化函式樣版都擁有相同的名稱，所以編譯器就得使用多載解析來呼叫正確的函式。

函式樣版可以數種方式予以多載。我們可指定相同的函式名稱但是不同的函式參數，來提供其它的函式樣版。舉例來說，圖 22.1 的 `printArray` 函式樣版只需提供額外的參數 `lowSubscript` 和 `highSubscript`，就能多載 `printArray` 函式樣版，以指定要輸出陣列的哪些部分 (請參閱習題 22.4) 。

函式樣版也可用其它具有相同的函式名稱但是不同的函式引數的非樣版函式來予以多載。例如，圖 22.1 的 `printArray` 函式樣版可以採用另一個非樣版函式版本予以多載，該函式會以簡潔的、表格化的欄位，專門印出一群字元字串(參閱習題 22.5)。

當某個函式被呼叫時，編譯器會執行配對處理，來判斷哪一個函式應該被呼叫。首先，編譯器會找出是否有函式名稱和引數型別完全與被呼叫之函式吻合的函式。假如沒有，編譯器會判斷是否有適合的函式樣版，可以用來建立一個不論是函式名稱或是引數型別都吻合該函式呼叫的特殊化函式樣版。假如找到了如是的樣版，編譯器就會建立並使用該合用的特殊化函式樣版。假如沒有，編譯器會產生錯誤訊息。若有多個函式都符合該呼叫，編譯器會認爲此呼叫模稜兩可，並產生錯誤訊息。

常見的程式設計錯誤 22.3

若找不到可適用於某個特別的函式呼叫的函式定義，或如果有多個符合的函式定義而讓編譯器混淆，則編譯器會產生錯誤。

22.4　類別樣版

我們可以瞭解，「堆疊」(是一種資料結構，它可以依序放入資料項，然後再依照「後進先出」的原則取出這些資料項)與放在堆疊中的資料型別無關的概念。然而，要實體化一個堆疊，資料的型別必須予以指明。這就讓軟體的再使用性有了絕佳的表現機會。我們需要方法來一般性地描述堆疊的概念，並且實體化此泛型堆疊類別的特定型別版本。C++ 透過類別樣版提供此功能。

軟體工程的觀點 22.2

讓泛型類別的某個特別型別版本可被實體化，因此得以增進了軟體再使用性。

類別樣版又稱爲**參數化型別 (parameterized types)**，因爲它們需要一或多個型別參數，來指出如何訂做「泛型類別」樣版以形成一個特殊化類別樣版。你只要寫一次類別樣版的定義，就可產生各種特殊化類別樣版。需要新的特殊化類別樣版時，你使用一種簡明的表示法，編譯器便能爲該表示法產生原始碼。舉例來說，`Stack` 類別樣版因此成爲用於建立使用於程式中的許多 `Stack` 類別的基礎 (例如「`double` 型別的 `Stack`」、「`int` 型別的 `Stack`」、「`char` 型別的 `Stack`」、「`Employee` 型別的 `Stack`」等)。

建立類別樣版 `Stack<T>`

注意圖 22.2 中 `Stack` 類別樣版的定義。它看起來很像一般的類別定義，但前面有以下這行標頭 (第 6 行)

```
template< typename T >
```

以便敘明一個具有型別參數 `T` 的類別樣版定義，該型別參數 `T` 在這裡當作 `Stack` 類別的型別佔位元。你不一定要用 `T` 當識別字，用任何有效的識別字都可以。`Stack` 中所儲存的元素型別，在整個 `Stack` 類別標頭和成員函式定義中，會統一使用識別字 `T` 表示。我們稍後會展示 `T` 如何關聯到特定的型別 (例如 `double` 或 `int`)。由於此類別樣版的設計方式，所以對於使用此 `Stack` 要的類別型別，得遵守兩個條件：它們必須有預設建構子 (用在第 44 行，以建立儲存堆疊元素的陣列)，而且它們的指定運算子必須能正確地將物件複製到 `Stack` 中 (第 56 行和 70 行)。

```
1  // Fig. 22.2: Stack.h
2  // Stack class template.
3  #ifndef STACK_H
4  #define STACK_H
5
6  template< typename T >
7  class Stack
8  {
9  public:
10    explicit Stack( int = 10 ); // default constructor (Stack size 10)
11
12    // destructor
13    ~Stack()
14    {
15       delete [] stackPtr; // deallocate internal space for Stack
16    } // end ~Stack destructor
17
18    bool push( const T & ); // push an element onto the Stack
19    bool pop( T & ); // pop an element off the Stack
20
21    // determine whether Stack is empty
22    bool isEmpty() const
23    {
24       return top == -1;
25    } // end function isEmpty
26
27    // determine whether Stack is full
28    bool isFull() const
29    {
30       return top == size - 1;
31    } // end function isFull
32
33 private:
34    int size; // # of elements in the Stack
35    int top; // location of the top element (-1 means empty)
36    T *stackPtr; // pointer to internal representation of the Stack
37 }; // end class template Stack
38
39 // constructor template
40 template< typename T >
41 Stack< T >::Stack( int s )
42    : size( s > 0 ? s : 10 ), // validate size
43      top( -1 ), // Stack initially empty
44      stackPtr( new T[ size ] ) // allocate memory for elements
45 {
46    // empty body
47 } // end Stack constructor template
48
49 // push element onto Stack;
50 // if successful, return true; otherwise, return false
51 template< typename T >
52 bool Stack< T >::push( const T &pushValue )
```

圖 22.2 Stack 類別樣版(1/2)

```
53  {
54     if ( !isFull() )
55     {
56        stackPtr[ ++top ] = pushValue; // place item on Stack
57        return true; // push successful
58     } // end if
59
60     return false; // push unsuccessful
61  } // end function template push
62
63  // pop element off Stack;
64  // if successful, return true; otherwise, return false
65  template< typename T >
66  bool Stack< T >::pop( T &popValue )
67  {
68     if ( !isEmpty() )
69     {
70        popValue = stackPtr[ top-- ]; // remove item from Stack
71        return true; // pop successful
72     } // end if
73
74     return false; // pop unsuccessful
75  } // end function template pop
76
77  #endif
```

圖 22.2　Stack 類別樣版(2/2)

類別樣版的成員函式定義就是函式樣版。出現在類別樣版定義外部的成員函式定義，每個前面要加上標頭

```
template< typename T >
```

(第 40、51 和 65 行)。因此，每個定義就跟一般函式定義很像了，除了 Stack 元素型別都用型別參數 T 表示之外。在類別樣版名稱 Stack<T> 旁使用了二元使用域解析運算子 (第 41 行、52 和 66 行)，以將每個成員函式定義綁到類別樣版的使用域上。此例中，一般化的類別名稱就是 Stack<T>。當實體化 Stack <double> 型別的 doubleStack 時，Stack 特殊化函式樣版的建構子會使用 new 建立一個元素型別爲 double 的陣列，來代表此堆疊 (第 44 行)。Stack 類別樣版定義中的敘述式，

```
stackPtr( new T[ size ] );
```

在 Stack<double> 特殊化類別樣版中，編譯器會把它產生成

```
stackPtr( new double[ size ] );
```

建立程式來測試類別樣版 Stack<T>

現在，讓我們探討 Stack 類別樣版的測試程式 (圖 22.3)。該測試程式首先會建立大小爲 5 的 doubleStack 物件 (第 9 行)。此物件會被宣告成類別 Stack<double> (念作「double 型別的 Stack」)。編譯器會將型別 double 與類別樣版中的型別參數 T 連結，產生 double 型別之 Stack 類別的原始碼。雖然該樣版提供了軟體再利用的優點，但是請記住，儘管我們在程式碼中只寫了一次樣版的程式碼，但在程式中會 (在編譯時) 實體化多個特殊化類別樣版。

```
1  // Fig. 22.3: fig22_03.cpp
2  // Stack class template test program.
3  #include <iostream>
4  #include "Stack.h" // Stack class template definition
5  using namespace std;
6
7  int main()
8  {
9     Stack< double > doubleStack( 5 ); // size 5
10    double doubleValue = 1.1;
11
12    cout << "Pushing elements onto doubleStack\n";
13
14    // push 5 doubles onto doubleStack
15    while ( doubleStack.push( doubleValue ) )
16    {
17       cout << doubleValue << ' ';
18       doubleValue += 1.1;
19    } // end while
20
21    cout << "\nStack is full. Cannot push " << doubleValue
22       << "\n\nPopping elements from doubleStack\n";
23
24    // pop elements from doubleStack
25    while ( doubleStack.pop( doubleValue ) )
26       cout << doubleValue << ' ';
27
28    cout << "\nStack is empty. Cannot pop\n";
29
30    Stack< int > intStack; // default size 10
31    int intValue = 1;
32    cout << "\nPushing elements onto intStack\n";
33
34    // push 10 integers onto intStack
35    while ( intStack.push( intValue ) )
36    {
37       cout << intValue++ << ' ';
38    } // end while
39
```

圖 22.3 Stack 類別樣版的測試程式(1/2)

```
40      cout << "\nStack is full. Cannot push " << intValue
41         << "\n\nPopping elements from intStack\n";
42
43      // pop elements from intStack
44      while ( intStack.pop( intValue ) )
45         cout << intValue << ' ';
46
47      cout << "\nStack is empty. Cannot pop" << endl;
48   } // end main
```

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop
```

圖 22.3　Stack 類別樣版的測試程式(2/2)

第 15–19 行會呼叫 push 將 double 數值 1.1、2.2、3.3、4.4 和 5.5 放到 doubleStack。當測試程式要 push 第六個數到 doubleStack 時，while 迴圈就會終止(因爲 doubleStack 滿了，它最多只能放 5 個元素)。當程式無法將數值推入堆疊時[1]，push 函式會傳回 false。

第 25–26 行會在 while 迴圈中呼叫 pop 從堆疊取出 5 個數值 (注意，圖 22.3 的輸出中，數值會以後進先出的順序取出)。當測試程式要取出第六個數值時，因爲 doubleStack 是空的，所以 pop 迴圈就會終止。

第 30 行以下列宣告，實體化一個整數堆疊 intStack

```
Stack< int > intStack;
```

(唸做「intStack 是個 int 型別的 Stack」)。因爲沒有指定堆疊大小，所以按預設建構子的規定設爲 10 (圖 22.2 的第 10 行)。第 35–38 行會重複執行，並呼叫 push 將數值推

[1] Stack 類別 (圖 22.2) 提供 isFull 函式，你可以在執行推入操作之前，用它判斷堆疊是否滿了。如此做可避免將資料推入滿的堆疊而造成錯誤。也可以讓 push 函式「拋出例外」。你可以撰寫程式來「補捉例外」，然後決定應用程式如何適當處理它。嘗試從空的堆疊中取出項目時，pop 函式亦可使用相同的技術。

入堆疊物件 intStack，直到它塞滿爲止；然後第 44–45 行會再重複執行，呼叫 pop 將數值從 intStack 取出，直到它變成空的。再次注意，在輸出中，數值是以「後進先出」的方式取出。

建立函式樣版來測試類別樣版 Stack<T>

請注意圖 22.3 的 main 函式中的程式碼與第 9–28 行的 doubleStack 操作程式以及第 30–47 行的 intStack 操作程式幾乎相同。這提供了另一個使用函式樣版的機會。圖 22.4 定義了函式樣版 testStack (第 10–34 行)以便執行跟圖 22.3 的 main 函式一樣的工作，就是把一串數值推入 Stack<T>，還有把數值從 Stack<T> 取出。函式樣版 testStack 也使用樣版參數 T (於第 10 行指定) 來代表 Stack<T> 儲存的資料型別。此函式樣版有四個引數 (第 12–15 行)，一個是指到型別 Stack<T> 的物件的參照，一個型別是 T 的數值，將會首先被推入 Stack<T>，一個型別是 T 的數值，代表每次被推入 Stack<T> 的數值增加量，最後一個引數型別是字串，代表要用於輸出的 Stack<T> 物件名稱。main 函式 (第 36–43 行) 會實體化一個型別 Stack<double> 的物件，叫做 doubleStack (第 38 行) 以及一個型別 Stack<int> 的物件，叫做 intStack (第 39 行)，並在第 41 和 42 行使用這些物件。編譯器由實體化該函式第一個引數的型別，來推論 testStack 的 T 型別。圖 22.4 的輸出跟圖 22.3 完全吻合。

```
1  // Fig. 22.4: fig22_04.cpp
2  // Stack class template test program. Function main uses a
3  // function template to manipulate objects of type Stack< T >.
4  #include <iostream>
5  #include <string>
6  #include "Stack.h" // Stack class template definition
7  using namespace std;
8
9  // function template to manipulate Stack< T >
10 template< typename T >
11 void testStack(
12    Stack< T > &theStack, // reference to Stack< T >
13    T value, // initial value to push
14    T increment, // increment for subsequent values
15    const string stackName ) // name of the Stack< T > object
16 {
17    cout << "\nPushing elements onto " << stackName << '\n';
18
19    // push element onto Stack
20    while ( theStack.push( value ) )
21    {
22       cout << value << ' ';
23       value += increment;
```

圖 22.4　將 Stack 樣版物件傳給一個函式樣版(1/2)

```
24       } // end while
25
26       cout << "\nStack is full. Cannot push " << value
27          << "\n\nPopping elements from " << stackName << '\n';
28
29       // pop elements from Stack
30       while ( theStack.pop( value ) )
31          cout << value << ' ';
32
33       cout << "\nStack is empty. Cannot pop" << endl;
34    } // end function template testStack
35
36    int main()
37    {
38       Stack< double > doubleStack( 5 ); // size 5
39       Stack< int > intStack; // default size 10
40
41       testStack( doubleStack, 1.1, 1.1, "doubleStack" );
42       testStack( intStack, 1, 1, "intStack" );
43    } // end main
```

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop
```

圖 22.4　將 Stack 樣版物件傳給一個函式樣版(2/2)

22.5　類別樣版的非型別參數和預設型別

第 22.4 節的類別樣版 Stack 在樣版標頭中只使用一個型別參數 (圖 22.2，第 6 行)。但亦可使用**非型別樣版參數 (nontype template parameter)**，它們可以有預設引數，並且當作 const。例如，樣版標頭可修改成接收一個 int elements 參數，如下所述：

```
template< typename T, int elements > // nontype parameter elements
```

然後，如下的宣告

```
Stack< double, 100 > mostRecentSalesFigures;
```

可在編譯時期實體化一個具有 100 個 double 數值元素的 Stack 特殊化類別樣版，稱爲 mostRecentSalesFigures；這個特殊化類別樣版的型別就是 Stack<double,100>。該類別定義因而可能擁有一個如下的陣列宣告的 private 資料成員

```
T stackHolder[ elements ]; // array to hold Stack contents
```

此外，型別參數可指定**預設的型別 (default type)**。例如，

```
template< typename T = string > // defaults to type string
```

會指定 Stack 預設包含 string 物件。然後，如下的宣告

```
Stack<> jobDescriptions;
```

可用來實體化一個 string 型別的 Stack 特殊化類別樣版，叫做 jobDescriptions；此特殊化類別樣版的型別是 Stack<string>。預設型別參數須是樣版的型別參數列中最右邊的參數。當某人正在實體化一個具有兩個以上的預設型別的類別時，若被省略的不是型別參數清單中位於最右邊的型別參數，則所有位在此型別右邊的型別參數也須被省略。

增進效能的小技巧 22.2

請盡量在編譯期間指定容器類別 (例如陣列類別或堆疊類別) 的大小 (可能是透過非型別樣版參數)。可消除以 new 動態建立空間的執行時期負荷。

軟體工程的觀點 22.3

在編譯期間指定容器大小，可避免在 new 無法取得記憶體時，所發生的潛在致命執行期錯誤。

本章習題會要求您使用非型別參數，來建立第 19 章 Array 類別的樣版。此樣版可讓我們在編譯期間，就建立一個用於第 19 章的 Array 類別的樣版。這個樣版使得 Array 物件得以以指定的型別與指定的元素數量被實體化，而不是在執行時期才建立 Array 物件的空間。

有時候，我們無法在類別樣版使用特定型別。舉例來說，Stack 樣版 (圖 22.2) 要求欲儲存在此 Stack 中的類別型別必須提供預設建構子與能正確地複製物件的指定運算子。若某種特別的使用者自訂型別無法使用在我們的 Stack 樣版上，或需要特別的處理，您可爲該特別的型別定義一個此類別樣版的**「明確特殊化 (explicit specialization)」**。假設我們要建立一個明確特殊化的 Stack，以存放 Employee 物件。要做這件事情，請以 Stack<Employee> 名稱建立一個新類別，如下所示：

```
template<>
class Stack< Employee >
{
   // body of class definition
};
```

此明確特殊化的 Stack<Employee> 完全取代了 Stack 類別樣版，專供 Employee 型別使用，它沒用到任何原有類別樣版的功能，甚至可擁有不同的成員。

22.6　總結

本章介紹了 C++中一個非常強大的功能，樣版。你學到了要如何使用函式樣版來讓編譯器產生一組特殊化函式樣版，用來代表一組相關的多載函式。我們也討論了要如何多載一個函式樣版以便產生一個函式的特殊化版本，使之得以以一個不同於其它特殊化函式樣版的方式來處理特殊的資料型別。接下來，你學到了類別樣版以及特殊化類別樣版。你見到了數個如何使用類別樣版來建立一組相關而可以於不同的資料型別執行相同操作之型別的範例。在下一章中，我們將會討論許多 C++的 I/O 功能，並示範幾個用來執行各種格式化的串流操作子。

摘要

22.1　簡介

- 樣版可讓我們指定一群相關的(多載的)函式，稱爲特殊化函式樣版，或一群相關的類別，稱爲特殊化類別樣版。

22.2　函式樣版

- 若要使用特殊化函式樣版，你會撰寫一個單一函式樣版定義。根據此函式呼叫中所提供的引數型別，C++會產生個別的特殊化函式，適當地處理每種型別的呼叫。
- 所有函式樣版的定義都是以關鍵字 template 開始，接著在角括號 (< 和 >) 中放入傳給函式樣版的樣版參數；每個用來代表型別的樣版參數必須在前面加上關鍵字 class 或 typename。指定函式樣版參數的關鍵字 typename 和 class 的意思是指「任何內建型別或使用者自訂型別」。
- 樣版定義的樣版參數可用來指定傳給函式的引數種類、函式的傳回型別以及宣告函式中的變數。
- 跟函式參數一樣，在樣版定義中的樣版參數名稱必須是獨一無二的。不同函式樣版之間的樣版參數名稱則不一定要是獨一無二的。

22.3 將函式樣版多載

- 函式樣版可以數種方式予以多載。我們可指定相同的函式名稱與不同的函式參數，來提供其它的函式樣版。函式樣版也可藉由提供另一個具有相同的函式名稱以及不同的函式參數的非樣版函式的方式予以多載。若該樣版與非樣版版本都同時符合該呼叫，則非樣版版本會被使用。

22.4 類別樣版

- 類別樣版提供了一種可一般性的描述某種類別，並實體化此泛型類別的特定型別版本的方法。
- 類別樣版又稱爲參數化型別 (parameterized types)，它們需要型別參數來指出如何訂做一個的「泛型類別」樣版以形成一個特定的特殊化類別樣版。
- 想要使用特殊化類別樣版，程式設計者撰寫一個類別樣版。當程式設計者需要一個新的特定型別的類別時，編譯器會爲該特殊化類別樣版產生原始碼。
- 類別樣版定義看起來很像是傳統的類別定義，但是它前面會加上 `template<typename T>` (或 `template<class T>`以表示它是一個類別樣版定義。型別參數 `T` 的作用宛如該類別型別的佔位字元。型別 `T` 在類別定義和成員函式定義中從頭到尾都被視爲一個一般化的型別名稱。
- 在類別樣版外部的每個成員函式，每個都必須以 `template<typename T>` (或 `template<class T>`) 開頭。因此，每個函式定義類似於傳統的函式定義，除了該類別中的一般化資料都用型別參數 `T` 表示之外。在類別樣版名稱旁要使用二元使用域解析運算子，以將每個成員函式定義綁到類別樣版的使用域上。

22.5 類別樣版的非型別參數和預設型別

- 在類別標頭或函式樣版中，可使用非型別參數。
- 你可以爲型別參數清單內的型別參數指定預設的型別。
- 我們可提供某類別樣版的明確特殊化 (explicit specialization)，以爲特定型別建立新樣版，重載取代原有的類別樣版。

術語

角括號 (< 和 >) [angle brackets (< and >)]
樣版型別參數中的關鍵字 `class` (`class` keyword in a template type parameter)
類別樣版 (class template)
類別樣版定義 (class-template definition)
特殊化類別樣版 (class-template specialization)
型別參數的預設型別 (default type for a type parameter)
明確特殊化 (explicit specialization)
樣版的 `friend` (`friend` of a template)
函式樣版 (function template)
函式樣版定義 (function-template definition)

特殊化函式樣版　(function-template specialization)
泛型程式設計　(generic programming)
巨集　(macro)
特殊化類別樣版的成員函式　(member function of a class-template specialization
非型別樣版參數　(nontype template parameter)
將函式樣版多載　(overloading a function template)
參數化型別　(parameterized type)
樣版　(template)
關鍵字 `template`　(`template` keyword)
樣版參數　(template parameter)
型別參數　(type parameter)
型別樣版參數　(type template parameter)
關鍵字 `typename`　(`typename` keyword)

自我測驗

22.1　是非題。如果答案爲非，請解釋爲什麼。

a) 函式樣版定義的型別樣版參數可用來指定傳給函式的引數型別、函式的傳回型別和宣告函式中的變數。
b) 關鍵字 `class` 和 `typename` 如果與型別樣版參數一起使用，就是特別指「任何使用者自訂的類別型別」。
c) 函式樣版可透過另一個具有相同函式名稱的函式樣版予以多載。
d) 各樣版定義之間所用的樣版參數名稱必須是獨一無二的。
e) 每個在類別樣版外部的成員函式定義，必須以樣版標頭開頭。

22.2　請填入以下題目的空格：

a) 樣版可讓我們只用一小段程式碼，就能指定全部相關函式的範圍，稱爲_____，或指定全部相關類別的範圍，稱爲_____。
b) 所有函式樣版定義均以關鍵字_____開頭，接著在_____中放入傳給函式樣版的樣版參數。
c) 從函式樣版產生的相關函式都具有相同的名稱，所以編譯器就得使用_____解析方法來呼叫正確的函式。
d) 類別樣版也稱作_____型別。
e) _____運算子會和類別樣版名稱合併使用，將每個成員函式定義綁到類別樣版的使用域。

自我測驗解答

22.1　a) 是　b) 非。這裡的關鍵字 `typename` 與 `class` 亦可使用內建型別的型別參數　c) 是　d) 非。各函式樣版之間所用的樣版參數名稱不一定要是獨一無二的　e) 是

22.2　a) 特殊化函式樣版，特殊化類別樣版　b) `template`，角括號 (< 和 >)　c) 多載　d) 參數化　e) 二元使用域解析

習題

22.3 **（選擇排序函式樣版）**根據附錄 E.1 的排序技巧，寫一個函式樣版 selectionSort。寫一個測試程式來輸入、排序和輸出 int 陣列和 float 陣列。

22.4 **（陣列列印範圍）**將圖 22.1 的函式樣版 printArray 多載，讓它可多取兩個整數引數，叫做 int lowSubscript 和 int highSubscript。呼叫這個函式只會印出陣列的指定部分內容。檢查 lowSubscript 和 highSubscript 的有效性；若其中一個超過陣列的範圍，或 highSubscript 小於或等於 lowSubscript，則 printArray 函式應該傳回 0；否則，printArray 應該傳回列印元素的個數。接著，修改 main 以在陣列 a、b、c (圖 22.1 的第 22–24 行) 上測試這兩版的 printArray。。

22.5 **（函式樣版多載）**以非樣版方式將圖 22.1 的函式樣版 printArray 多載，該樣版以特別用整齊、表格化的格式列印字元字串陣列。

22.6 **（樣版的運算子多載）**撰寫一個判斷函式 isEqualTo 的簡單函式樣版，該樣版利用等號運算子 (==)比較它的二個相同型別的引數是，如果相等則傳回 true，如果不相等則傳回 false。在程式中使用這個函式樣板，該程式僅以各種內建的型別呼叫 isEqualTo 函式。現在撰寫一個該程式的不同版本，該程式呼叫使用者自訂類別的 isEqualTo 函式，但是不要多載等號運算子。當您要執行這個程式時，會產生什麼現象？現在多載等號運算子 (以運算子函式) operator==。當您要執行這個程式時，會產生什麼現象？

22.7 **（陣列類別樣版）**重新實作從圖 19.10-19.11 的 Array 類別成爲類別樣版。請於程式中展示新的 Array 類別樣版。

22.8 請分辨「函式樣版」和「特殊化函式樣版」的差異。

22.9 請解釋哪一個最像型版，是類別樣版還是特殊化類別樣版？

22.10 函式樣版和多載之間存在何種關係？

22.11 爲什麼會選擇使用函式樣版來取代巨集呢？

22.12 使用函式樣版和類別樣版會產生哪些執行性能的問題？

22.13 呼叫某個函式時，編譯器會執行配對處理，來判斷應該呼叫哪一個特殊化函式樣版。在何種情況下，嘗試進行比對過程會產生編譯錯誤呢？

22.14 爲什麼可以把類別樣版稱作參數化型別？

22.15 解釋爲何 C++程式會使用以下敘述

```
Array< Employee > workerList( 60 );
```

22.16 回頭檢視習題 22.15 的答案。解釋爲何 C++程式會使用以下敘述

```
Array< Student > studentNumber;
```

22.17 請解釋在 C++程式中以下表示法的使用：

```
template< typename T > Array< T >::Array(char p )
```

22.18 要產生一個像陣列或堆疊的容器類別時，爲何會在類別樣版中使用非型別參數？