Chapter 14

樣板

樣板

- 樣板：

  程式碼製造機制，可以依據使用者設定的基本樣式，製造出近似作用的對應程式碼

- 樣板類型：
  - 樣板函式
  - 樣板類別

template

樣板函式（一）

- 許多平方函式

```
int square( const int& x ) {                    // 整數平方
    return x * x ;
}

double square( const double& x ){                // 浮點數平方
    return x * x ;
}

Fraction square( const Fraction& x ){            // 分數平方
    return x * x ;
}
```

❖ 以上各個平方函式內部的程式碼都一樣，但卻是不同類型的平方函式，不可省略為一，須費力的個別定義

樣板函式（二）

- 平方樣板函式

```
template <class T>
T  square( const T& x ) { return x * x ; }
```

以上 **class** 為樣板資料型別，可用 **typename** 取代，**T** 為型別樣板參數

C++ 編譯器會根據輸入的參數型別不同，在編譯過程中造出不同的平方函式

```
cout << square(2) << endl ;    // T 為整數
cout << square(3.3) << endl ;  // T 為浮點數
```

template function

數字轉字串樣板函式（一）

- 整數轉字串

```
string int_to_str( int no ){
    ostringstream  number ;          // 使用 sstream
    number << no ;
    return number.str() ;
}
```

- 浮點數轉字串

```
string double_to_str( double no ){
    ostringstream  number ;          // 使用 sstream
    number << no ;
    return number.str() ;
}
```

數字轉字串樣板函式（二）

- 樣板函式

```
template <class T>
string num_to_str( const T& no ){
    ostringstream number ;
    number << no ;
    return number.str() ;
}
```

- 使用方式

```
string foo = "氣溫為 " + num_to_str(17.5) + " 度" ;
cout << foo << endl ;
string bar = "氣溫為 " + num_to_str(-3) + " 度" ;
cout << bar << endl ;
```

樣板函式的運作機制（一）

- 使用樣板機制製造函式過程中，型別樣板參數是以直接推導的方式求得，不包含型別轉換

```
template <class T>
T max2(const T& x , const T& y){ return( x > y ? x : y ) ; }
```

```
int main(){
    cout << max2( 1 , -3 )      << endl ;   // T 為 int
    cout << max2( 'A' , 'a' )   << endl ;   // T 為 char
    cout << max2( -2.5 , 3.0 ) << endl ;   // T 為 double
}
```

但 `cout << max2( 1 , 3.0 ) ;` `// T 無法確定`

可修改成以下

```
cout << max2( static_cast<double>(1) , 3.0 ); // T 為 double
```

樣板函式的運作機制（二）

- 若同時有樣板函式與普通函式可選擇，則以樣板函式為最後的選擇方式

```
template <class T>
T max2(const T& x , const T& y){
   return ( x > y ? x : y ) ;
}

int max2(const int &a , const int& b){
   return ( a > b ? a : b ) ;
}

cout << max2( 1 , 3 ) ;        //使用普通函式
cout << max2( 2.0 , 3.0 ) ;    //使用樣板函式
```

樣板函式的運作機制（三）

- 若樣板參數的型別無法由函式的參數型別來確定，則使用者必須明確地設定型別
  - 字串轉數字

```
template <class T>
T str_to_num( const string& foo ){
    T num ;
    istringstream(foo) >> num ;
    return num ;
}
```

  - 使用方法

```
string no1 = "0.0314159e2" ;
double  pi = str_to_num<double>(no1);
string no2 = "99" ;
int   max  = str_to_num<int>(no2);
```

樣板函式的運作機制（四）

- 樣板型別參數推導過程並不包含型別轉換，但普通參數仍可做型別轉換

```
// 計算陣列前 n 個元素的數值和
template <class T>
T sum( T a[] , unsigned int n ){
   T  s = static_cast<T>(0) ;
   for( int i = 0 ; i < n ; ++i )
        s += a[i];
   return s ;
}
```

樣板函式的運作機制（五）

- 在使用上

```
int a[] = { 3 , 1 , 2 , 5 , 7 } ;
double b[] = { 3.1 , 2.2 , 3.3 , 4.2 } ;

unsigned int m = 3 ;

// T 為 int
cout << sum( a , m ) << endl ;

// T 為 double，但 3 會自動轉為 unsigned int
cout << sum( b , 3 ) << endl ;
```

多參數型別樣板函式（一）

- 找出陣列中最大的元素

```
template <class T>
T  max_element( const T foo[] , T size ){
   T max = 0 ;
   for( int i = 0 ; i < size ; ++i )
       if( foo[i] > max ) max = foo[i] ;
   return max ;
}
```

但使用

```
unsigned int bar[S] = { 9 , 6 , 7 } ;
cout << max_element( bar , 3 ) << endl ;     // 錯誤
```

❖ 以上的出錯原因在於樣板參數並不會自動作型別轉換，**bar** 的 **unsigned int** 型別與 **3** 的 **int** 型別不同

多參數型別樣板函式（二）

- 雙參數型別樣板函式

```
template <class T , class S>
T max_element( const T foo[] , S size ){
   T max = 0 ;
   for( int i = 0 ; i < size ; ++i )
       if( foo[i] > max ) max = foo[i];
   return max ;
}
```

- 單參數型別樣板函式

```
cout << max_element( bar ,
                     static_cast<unsigned int>(3) );
```

多參數型別樣板函式範例(一)

- 兩下標間的最大元素

```
template <class T , class S>
T max_element( const T foo[] , S a , S b ){
    T max = foo[a] ;
    for( int i = a+1 ; i <= b ; ++i )
        if( foo[i] > max ) max = foo[i] ;
    return max ;
}
const int S = 5 ;
float foo[S] = { 2.1 , 3.1 , 4.2 , 1.2 , 0.8 } ;
unsigned int a = 2 ;
int b = 1 , c = 3 ;
cout << max_element(foo,b,c)   << endl ; // 正確 b，c 同一型別
cout << max_element(foo,a,c)   << endl ; // 錯誤 b，a 不同型別
cout << max_element(foo,b,S-1) << endl ; // 正確 b，S-1 同型別
```

多參數型別樣板函式範例(二)

- 字串數字和

```
template <class S , class T>
T  string_no_sum( string s ){
    S sum = 0 , no ;
    istringstream  foo(s) ;              // 使用 sstream
    while( foo >> no ) sum += no ;
    return  static_cast<T>(sum) ;
}
```

使用時要明確標明樣板參數型別

```
// 錯誤，須指定樣板參數型別
cout<< string_no_sum( "1.2 2.9" ) << endl ;


// 輸出：12
cout<< string_no_sum<int,double>( "3 6 1 2" ) << endl;
```

樣板類別

- 樣板類別：用來設定類別內所使用的資料型別

  - 集合樣板類別

```
template <class T>
class Set{
   private:
      T  data[10] ;
      int count ;
   public:
      Set():count(0){} ;
      bool insert( const T& element ) ;
};
```

```
Set<int>  foo ;    // 儲存整數的集合物件
Set<char> bar ;    // 儲存字元的集合物件
```

template class

數值樣板參數（一）

- 數值樣板參數

```
template < class T , int SIZE = 10 >
class  Set{
   private:
      T   data[SIZE];   // 可以儲存 SIZE 個 T 型別元素的陣列
      int    count  ;   // 已經儲存的集合元素個數
   ...
};
```

以上的 **SIZE** 為數值樣板參數，預設值為 **10**

- 使用方法

```
Set<char,20>  foo ;   // 可以儲存 20 個字元的集合
Set<int>      bar ;   // 可以儲存 10 個整數的集合
```

value template parameter

數值樣板參數（二）

- 數值樣板參數的數值須在編譯時就已確定

```
int n ;
cin >> n ;

// 錯誤，數值樣板參數須在編譯時確定
Set<int,n> foo;
```

樣板類別的函式語法（一）

- 樣板類別的名稱可加上樣板符號

```
template <class T>
class  Point{
   private:
      T  x , y ;
   public:
     // 建構函式
     Point<T>(const T& a , const T& b ): x(a) , y(b){}
     // 複製：回傳一複製物件
     Point<T> clone() const ;
     // 覆載輸出運算子
     template <class S>
     friend ostream& operator<<( ostream& , const Point<S>& );
};
```

樣板類別的函式語法（二）

- 若函式撰寫在類別之外，則須以樣板函式方式撰寫

```
// 定義在樣板類別外的函式
template <class T>
Point<T> Point<T>::clone(){ return *this; }
```

```
template <class T>
ostream& operator<<( ostream& out ,
                     const Point<T>& pt ){
   out << "( " << pt.x << " , " << pt.y << " )" ;
   return  out ;
}
```

樣板類別的函式語法（三）

- 定義在樣板函式內的夥伴函式可以不須改成樣板函式的模式

```
template <class T>
class Point {
   public:
      ...
      friend ostream& operator<< ( ostream& out ,
                                   const Point<T>& pt ){
          out << "( " << pt.x << " , " << pt.y << " )" ;
          return  out ;
      }
};
```

❖ 以上並不是所有的編譯器都適用

函式使用樣板類別（一）

- 若函式內有使用到樣板類別物件，則須以樣板機制定義此函式

```
template <class T , int S=10>
struct Array{
   T  data[S] ;
};
```

```
template <class T , int S>
ostream& operator<< ( ostream& out ,
                      const Array<T,S>& foo ){
   for( int i = 0 ; i < S ; ++i )
       out << foo.data[i] << ” ” ;
   return out ;
}
```

函式使用樣板類別（二）

- 在使用上

```
int main(){

    Array<int> foo ;
    for( int i=0 ; i<10 ; ++i ) foo.data[i] = i ;
    cout << foo << ' ' ;   // 輸出：0 1 2 ... 9

    Array<char,26> bar ;
    for( int i=0 ; i<26 ; ++i ) bar.data[i] = 'a' + i ;
    cout << bar << ' ' ;   // 輸出：a b c ... z

    return 0 ;

}
```

樣板程式碼儲存的檔案

- 所有的樣板相關的程式碼都須置於程式的標題檔

所有的樣版函式程式碼
`#include "foo.cc"`

`foo.h`

無樣版函式程式碼
`.....`

`foo.cc`

- 原因：

  編譯器在編譯進行時就須確定樣板參數的值

「樣板類別」型別樣板參數

- 型別樣板參數也可以是另一個樣板資料型別

```
// 正確
Set< Point<int> >  ;

// 錯誤，>> 會被當成輸入運算子
Set< Point<float>> ;
```

函式型別樣板參數（一）

- 型別樣板參數也可以是另一個函式
  - 定義三個樣板函式

```
template <class T>
T square( T a ){ return a * a ; }
```

```
template <class T>
T cubic( T a ){ return a * a * a ; }
```

```
template <class F ,class T>
T compute( F fn , T x ){ return fn(x) ; }
```

  - 執行結果

```
cout << compute( square<double> , 1.1 ); // 印出：1.21
cout << compute( cubic<int> , -3 );      // 印出：-27
```

函式型別樣板參數（二）

- 若使用函式指標的方式改寫，則為

```
int square( int a ){ return a * a ; }
int cubic( int a ){ return a * a * a ; }
int compute( int (*fn)(int) , int a ){ return fn(a) ; }
```

- 執行結果

```
cout << compute( square , -2 ) << endl ;    // 印出：4
cout << compute( cubic  ,  3 ) << endl ;    // 印出：27
```

範例：牛頓迭代法（一）

- 牛頓迭代法：利用切線尋求函數的根

  若 F(x) 為一函數，則此函數的根可以由牛頓迭代法計算求得

$$X_2 = X_1 - \frac{F(X_1)}{F'(X_1)}$$

X_1：為迭代前的初值

X_2：為迭代後的新值

$F'(X_1)$：為 F(X) 函式在 X_1 的微分

❖ 在若干步後，當 $|X_2 - X_1|$ 趨近於零，則此迭代為收斂，不收斂的迭代可以在給定新的初值 X_1 後重新展開迭代

範例：牛頓迭代法（二）

- 牛頓迭代法也可以計算函數的複數根
- 複數樣板類別

```
#include <complex>
...
complex<int>    a;                      // a = 0 + 0 i
complex<int>    b(1);                   // b = 1 + 0 i
complex<double> c( 1.5 , 3.2 );         // c = 1.5 + 3.2 i

cout << c.real() << "+" << c.imag() << endl;
```

在 **complex** 標頭檔內定義了許多與數學運算方式相當的複數運算子

如 + , - , * , / , += , -= , *= , /=
也有 == , != , >> , << 等

程式　輸出

範例：樣板函式

- 將函式當成型別樣板參數來使用

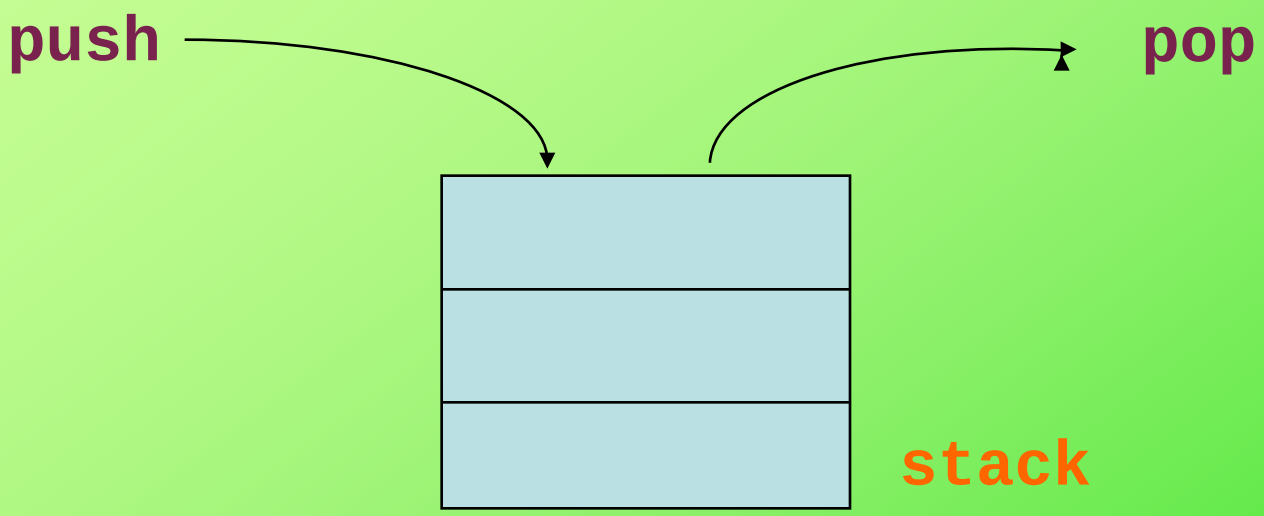
```
template <class Function , class T>
void transform( Function f , T a[] , int size ){
     for( int i = 0 ; i < size ; ++i )
        a[i] = f( a[i] ) ;
}
```

程式

輸出

範例：堆疊樣板類別

- 堆疊



- 特性
  - 先存入 (push) 的元素越晚被取出 (pop)
  - 元素的先後順序會顛倒

stack