例外處理：一窺究竟 24

It is common sense to take a method and try it.If it fails, admit it frankly and try another.But above all, try something.
—***Franklin Delano Roosevelt***

If they're running and they don't look where they're going I have to come out from somewhere and catch them.
—***Jerome David Salinger***

學習目標

在本章中，你將學到：

- 分別使用 `try`、`catch` 以及 `throw`，偵測、處理和指出例外。
- 處理未捕捉到和非預期的例外。
- 宣告新的例外類別。
- 堆疊展開如何讓使用域無法捕捉的例外情況，在另一使用域捕捉。
- 處理 `new` 的錯誤。
- 使用 `unique_ptr` 來防止記憶體遺漏。
- 瞭解標準例外的階層關係。

本章綱要

摘要｜術語｜自我測驗｜自我測驗解答｜習題

24.1 簡介

正如你所知道的，**例外 (exception)** 是程式執行時遇到問題的跡象。**例外處理 (Exception handling)** 讓你建立能化解 (或處理) 例外的應用軟體。在許多情況下，例外處理讓程式得以繼續執行，如同程式沒有遇到問題一樣。本章介紹的功能可以讓你撰寫**堅固 (robust)** 且**容錯的程式 (fault-tolerant programs)**，意即程式可以處理發生的問題且繼續執行，或是優雅的結束。

我們首先透過一個例子來複習例外處理的觀念，於這個例子中展示了如何處理因除以零的而出現的例外我們會討論其他的例外處理問題，像是如何處理建構子或解構子的例外，以及當 new 運算子爲物件配置記憶體空間發生錯誤時，如何處理例外。在本章最後，我們會介紹 C++標準函式庫中，幾個負責例外處理的類別。

軟體工程的觀點 24.1

例外處理提供處理錯誤的標準機制，和大型程式設計團隊一起合作專案時，這個機制特別重要。

軟體工程的觀點 24.2

從一開始就要將例外處理策略納入你的系統。在系統完成之後，就很難將例外處理機制納入。

24.2　範例：處理除零動作

現在讓我們介紹一個簡單的例外處理範例 (圖 24.1–24.2)。這個範例想要避免常見的數學運算難題，就是除零。在 C++ 的整數運算除零，通常會造成程式提前結束。在某些C++實作版本中，浮點數運算的除零是被允許的，結果會顯示 INF 或 -INF。

這個範例會定義一個名稱爲 quotient 的函式，這個函式會接收使用者輸入的兩個整數，然後把第一個 int 參數除以第二個 int 參數。函式在計算除法之前會把第一個 int 參數的值轉型成 double 型別，然後第二個 int 參數在計算時會提昇成爲 double 型別。所以 quotient 函式實際上是做兩個 double 的除法運算，再傳回 double 型別的結果。

雖然浮點數運算允許除零，但是在這個範例，我們把除零當成是錯誤。所以，quotient 函式在進行除法運算之前，要先確定第二個參數不是零。如果第二個參數是零，這個函式要使用例外告訴呼叫的函式發生問題。然後呼叫的函式 (在此是 main) 可以處理這個例外，讓使用者輸入兩個新的值，再呼叫 quotient 函式。依照這種方式的話，即使輸入不當的值，程式還是可以繼續執行，因此更加穩固。

這個範例包括兩個檔案：DivideByZeroException.h (圖 24.1) 定義的例外類別代表可能發生的問題，fig24_02.cpp (圖 24.2) 定義 quotient 函式以及呼叫它的 main 函式。main 函式有示範例外處理的程式碼。

定義一個例外類別來代表可能發生的問題類型

圖 24.1 定義 DivideByZeroException 類別，當做自標準函式庫 **runtime_error** 類別 (定義在標頭檔 **<stdexcept>**) 的衍生類別。runtime_error 類別 (定義在標頭檔**<exception>**)，一個標準函式庫 **exception** 類別的衍生類別，是 C++ 標準的基本類別，用來代表執行時期錯誤。exception 類別是標準 C++基本類別，用來代表所有的例外。(在第 24.11 節會詳細介紹 exception 類別及其衍生類別)。一個衍生自 runtime_error 類別的例外類別通常只有定義建構子 (例如，第 12-13 行)，該建構子會把錯誤訊息字串傳給基本類別 runtime_error 的建構子。每個直接或間接衍生自 exception 的例外類別都有 virtual 函式 **what**，這個函式會傳回例外物件的錯誤訊息。你不一定要從 C++的標準例外類別衍生自訂的例外類別，例如 DivideByZeroException。可是，這樣做的話可以讓你使用 virtual 函式 what 來獲得適當的錯誤訊息。我們在圖 24.2 中使用 DivideByZeroException 類別物件來指出何時發生除零的動作。

```
1  // Fig. 24.1: DivideByZeroException.h
2  // Class DivideByZeroException definition.
3  #include <stdexcept> // stdexcept header contains runtime_error
4  using namespace std;
5
6  // DivideByZeroException objects should be thrown by functions
7  // upon detecting division-by-zero exceptions
8  class DivideByZeroException : public runtime_error
9  {
10 public:
11    // constructor specifies default error message
12    DivideByZeroException()
13       : runtime_error( "attempted to divide by zero" ) {}
14 }; // end class DivideByZeroException
```

圖 24.1 DivideByZeroException 類別的定義

示範例外處理

圖 24.2 中的程式使用例外處理來包裝可能會拋出「除零」例外的程式碼，並且處理這個例外，如果眞的發生例外的話。使用者能夠輸入兩個整數，這會當成引數傳遞給 quotient 函式 (第 10–18 行)。這個函式會將第一個參數 (numerator) 除以第二個參數 (denominator)。假設使用者沒有將除法運算的分母設定爲 0，則 quotient 函式會傳回除法的結果。然而，如果使用者輸入 0 當作分母，則 quotient 函式會拋出例外。在範例輸出中，前兩行會顯示一個成功的計算，而接下來兩行會顯示一個錯誤的計算，因爲程式嘗試執行除零的運算。在例外發生時，程式會通知使用者發生錯誤，並且要使用者輸入兩個新的整數。在介紹完程式碼之後，我們會考慮使用者的輸入，以及產生這種結果的程式流程。

```
1  // Fig. 24.2: fig24_02.cpp
2  // A simple exception-handling example that checks for
3  // divide-by-zero exceptions.
4  #include <iostream>
5  #include "DivideByZeroException.h" // DivideByZeroException class
6  using namespace std;
7
8  // perform division and throw DivideByZeroException object if
9  // divide-by-zero exception occurs
10 double quotient( int numerator, int denominator )
11 {
12    // throw DivideByZeroException if trying to divide by zero
13    if ( denominator == 0 )
14       throw DivideByZeroException(); // terminate function
15
```

圖 24.2 除零時拋出例外的處理範例(1/2)

```
16      // return division result
17      return static_cast< double >( numerator ) / denominator;
18   } // end function quotient
19
20   int main()
21   {
22      int number1; // user-specified numerator
23      int number2; // user-specified denominator
24      double result; // result of division
25
26      cout << "Enter two integers (end-of-file to end): ";
27
28      // enable user to enter two integers to divide
29      while ( cin >> number1 >> number2 )
30      {
31         // try block contains code that might throw exception
32         // and code that will not execute if an exception occurs
33         try
34         {
35            result = quotient( number1, number2 );
36            cout << "The quotient is: " << result << endl;
37         } // end try
38         catch ( DivideByZeroException &divideByZeroException )
39         {
40            cout << "Exception occurred: "
41               << divideByZeroException.what() << endl;
42         } // end catch
43
44         cout << "\nEnter two integers (end-of-file to end): ";
45      } // end while
46
47      cout << endl;
48   } // end main
```

```
Enter two integers (end-of-file to end): 100 7
The quotient is: 14.2857

Enter two integers (end-of-file to end): 100 0
Exception occurred: attempted to divide by zero

Enter two integers (end-of-file to end): ^Z
```

圖 24.2　除零時拋出例外的處理範例(2/2)

在 `try` 區塊內圍繞程式碼

程式會先提示使用者輸入兩個整數，輸入的整數成爲 `while` 迴圈 (第 29 行) 的條件。第 35 行把這兩個數值傳給 `quotient` 函式 (第 10–18 行)，這個函式會計算整數除法並回傳結果，或是在除零時**拋出例外 (throws an exception**，表示有發生錯誤)。例外處理也針對當函式偵測到無法處理的錯誤的處境。

try 區段 (try block) 包含可能造成例外的敘述，以及發生例外就應該跳過的程式碼。請注意，try 區塊 (第 33–37 行) 括住 quotient 函式的呼叫，以及顯示除法結果的敘述。於這個範例中，因爲 quotient 函式在呼叫時 (第 35 行) 可能會拋出例外，所以把這個函式呼叫放在 try 區塊內。把輸出結果的敘述 (第 36 行) 放在 try 區塊內，可以確保只有在 quotient 函式傳回結果的時候才會輸出。

軟體工程的觀點 24.3

例外可能會經由 try 區塊內明確列出的程式碼、或是透過呼叫其他函式、或是透過 try 區塊內深層的巢狀函式呼叫而浮現。

定義 catch 處理常式來處理 DivideByZeroException

例外是由 **catch 處理常式 (catch handler)**。每個 try 區域之後至少要有一個 catch 處理常式(第 38–42 行)。**例外參數 (exception parameter)**係以一個 catch 處理常式可以處理的型別 (此例是 DivideByZeroException) 的參照被宣告。當 try 區塊內有例外發生時，會被執行的 catch 處理常式是其型別與例外型別相符者 (catch 區塊中的型別完全符合例外型別，或是該例外的基本類別)。如果例外參數包含選用性的參數名稱，該 catch 處理常式可以使用這個參數名稱，在 catch 的本體內，也就是大括號 ({ 和 }) 限定的範圍內，和捕捉到的例外物件互動。catch 處理常式通常會向使用者回報錯誤、把錯誤記錄在檔案、優雅地結束程式或是嘗試其他的策略以完成失敗的工作。這個範例的 catch 處理常式只是簡單地回報使用者想要除以零。然後程式會提示使用者輸入兩個新的整數。

常見的程式設計錯誤 24.1

把程式碼放在 try 區塊和對應的 catch 處理常式之間或 catch 處理常式中間，是一種語法錯誤。

常見的程式設計錯誤 24.2

每個 catch 處理常式只能有一個參數。指定一串逗點間隔的例外參數，是一種語法錯誤。

常見的程式設計錯誤 24.3

在單一 try 區塊之後有兩個不同的 catch 處理常式捕捉相同型別的例外，是一種邏輯錯誤。

例外處理的終止模式

如果 try 區塊內某行敘述發生例外，這個 try 區塊就會過期 (就是立刻結束)。接著，程式會搜尋第一個可以處理該例外的 catch 處理常式。程式會透過比較拋出的例外型別與每一個 catch 的例外參數型別，直到找出符合的 catch 區塊。如果例外的型別與拋出例外的型別相同，或者例外的型別是例外參數型別的衍生類別，則是符合的情況。找到符合的情況時，程式會執行對應的 catch 處理常式。當 catch 處理常式到達自己的右括號 (}) 就是處理完成，該例外被視爲已經處理完畢，而在 catch 處理常式之內定義的區域變數 (包括 catch 參數)就脫離了使用的視野。因爲該 try 區塊已經過期。程式流程並不會回到發生例外的地點 (稱爲**拋出點，throw point**)，而是從 try 區塊最後的 catch 處理常式之後第一行敘述 (第 44 行) 繼續執行。這稱爲**例外處理的終止模式 (termination model of exception handling)**。有些語言使用**例外處理的恢復模式 (resumption model of exception handling)**，例外處理後的控制權會從拋出點之後繼續執行。如同其它區段的程式碼，當 try 區段終止時，在該區段定義的區域變數也會脫離使用的視野。

常見的程式設計錯誤 24.4

在例外處理之後，如果你認爲程式的控制權會回到拋出點後第一個敘述，就會發生邏輯錯誤。

測試和除錯的小技巧 24.1

藉由例外處理，程式可以在解決問題之後繼續執行 (而不是終止)。這有助於確保程式的穩固，達成重要任務的處理或是關鍵的商業處理。

如果 try 區塊執行成功 (也就是這個區塊沒有發生例外)，程式就會忽略 catch 處理常式，從這個 try 區塊最後的 catch 之後第一行敘述繼續執行。

如果在 try 區塊發生的例外沒有符合的 catch 處理常式，或者產生例外的敘述沒有在 try 區塊內，則包含該敘述的函式會立刻終止，然後程式會試著尋找呼叫函式內是否有 try 區塊。這個過程稱爲**堆疊展開 (stack unwinding)**，我們會在第 24.6 節介紹。

當使用者輸入非零分母時，程式的執行流程

當使用者輸入分子 100 以及分母 7 時 (也就是圖 24.2 輸出的前兩行)，讓我們來探討程式的流程。quotient 函式在第 13 行判斷分母不等於零，所以在第 17 行執行除法計算，把結果 (14.2857) 以 double 型別傳回到第 35 行。程式會從第 35 行繼續執行，所以第 36 行會顯示除法運算的結果，而第 37 行是 try 區塊的結尾。因爲 try 區塊成功完成而

且沒有拋出例外，所以程式不會執行 catch 處理常式 (第 38–42 行) 內的敘述，繼續從第 44 行 (在 catch 處理常式之後的第一行程式碼) 執行，提示使用者再輸入兩個整數。

當使用者輸入零當分母時，程式的執行流程

現在來探討使用者輸入分子 100 以及分母 0 的情況。quotient 在第 13 行發現 denominator 等於零，這會造成除零的情況。在第 14 行會拋出例外，以 DivideByZeroException 類別 (圖 24.1) 的物件來代表。

要拋出例外，第 14 行使用 **throw** 關鍵字，後面接著一個代表要拋出例外的型別的運算元。一般而言，一個 throw 敘述只會指定一個運算元。(我們在第 24.4 節會討論如何使用沒有運算元的 throw 敘述)。throw 的運算元可以是任何型別。如果是物件的話，稱爲 **例外物件 (exception object)**，這裡的例外物件是 DivideByZeroException 型別的物件。然而，throw 運算元也可以是其它的數值，例如某個運算式的值 (例如 throw x > 5)，或是 int 值 (例如 throw 5)。本章的範例只著重在拋出例外類別的物件。

常見的程式設計錯誤 24.5

當拋出條件運算式 (?:) 的結果時需要特別注意，因爲提昇規則 (promotion rules) 會造成數值的型別和預期不同。例如，同一個條件運算式會拋出 int 或 double 型別時，條件運算式會把 int 轉換成 double 型別。在這種情況下，用來捕捉 int 的 catch 處理常式就永遠不會執行。

做爲拋出例外的一部分，throw 運算元係被建立並用來初使化 catch 處理常式 (很快就會介紹) 的參數。範例第 14 行的 throw 敘述建立一個 DivideByZeroException 類別的物件。當第 14 行拋出例外時，quotient 函式會立刻終止。所以，quotient 函式在執行第 17 行除法運算之前，會先在第 14 行拋出例外。這是例外處理的主要特性：函式應該在錯誤有機會發生**之前**拋出例外。

因爲 quotient 函式呼叫 (第 35 行) 包圍在 try 區塊內，程式會執行到 try 區塊之後的 catch 處理常式 (第 38–42 行)，程式的控制權移轉到這個 catch 處理常式。這個 catch 處理常式當做除零例外的例外處理常式。通常，當在 try 區塊拋出例外之後，與拋出例外型別相符的 catch 區塊會捕捉到這個例外。於這個程式中， catch 處理常式指定要捕捉 DivideByZeroException 物件，符合 quotient 函式拋出的物件型別。catch 處理常式實際上會捕捉指向 DivideByZeroException 物件的參照，這個物件是 quotient 函式的 throw 敘述 (第 14 行) 建立的，使得該 catch 處理常式不需要複製一份該例外物件。

catch 處理常式的內容 (第 40–41 行) 會呼叫 runtime_error 基本類別的 what 函式，印出傳回的錯誤訊息，此函式會傳回由 DivideByZeroException 建構子 (圖 24.1，第 12–13 行) 傳給 runtime_error 的基本類別建構子的字串。

增進效能的小技巧 24.1

捕捉例外物件的參照可以避免複製拋出例外物件的額外負擔。

良好的程式設計習慣 24.1

將每種執行時期錯誤類型，和名稱適合的例外物件關聯在一起，可以讓程式更清晰易懂。

24.3　何時使用例外處理

例外處理的設計是處理**同步的錯誤 (synchronous errors)**，代表敘述執行時發生的錯誤。常見的例子就是超出陣列下標的範圍、算術溢位 (也就是某個數值超出可表示的範圍)、除零、錯誤的函式參數，以及記憶體配置失敗 (因爲記憶體不足)。例外處理的設計並非處理與**非同步事件 (asynchronous event)** 有關的錯誤 (例如，磁碟 I/O 動作完成、網路訊息到達、按下滑鼠鍵和按鍵盤)，這些事件和程式的執行流程同時發生且不受影響。

軟體工程的觀點 24.4

例外處理提供唯一且一致的技術來處理錯誤，幫助大型專案的程式設計者瞭解彼此的錯誤處理碼。

軟體工程的觀點 24.5

請不要把例外處理當成另類的程式流程控制，這些「多餘」的例外會「阻擋」到純粹的錯誤類型例外。

軟體工程的觀點 24.6

例外處理使事先定義的元件能將問題傳至應用程式專屬元件，然後利用應用程式的特定方式處理問題。

當程式與軟體元件 (例如，成員函式、建構子、解構子和類別) 互動時發生問題，例外處理機制也可以幫忙解決。這種軟體元件通常使用例外通知程式發生問題，這讓你可以替各種應用程式實作自訂的錯誤處理。

增進效能的小技巧 24.2

沒有發生例外的時候，例外處理的程式碼對效能的影響很小，或者甚至沒有影響。因此，實作例外處理的程式，比起混合錯誤處理和程式邏輯的程式更有效率。

軟體工程的觀點 24.7

發生常見錯誤情況的函式應該傳回 0 或 NULL (或其它適當的值，如 bool)，而不是拋出例外。呼叫這種函式的程式可以檢查傳回值，判斷呼叫是否成功。

複雜的應用程式通常由事先定義好的元件，以及使用這些事先定義好元件的應用程式專屬 (application-specific) 元件所構成的。當事先定義好的元件碰到問題時，該元件需要能與應用程式專屬元件溝通問題的機制，因爲事先定義好的元件無法提前得知應用程式如何處理發生的問題。

24.4 重新拋出例外

例外處理常式收到例外時可能無法處理，或只能處理例外的一部分。在這種情況下，catch 處理常式可以將例外處理 (或是其中一部分) 延後，交給另一個處理常式。你可以用以下敘述式將**例外重新拋出 (rethrowing the exception)**

```
throw;
```

無論處理常式是否能夠處理某個例外，該常式可以重新拋出這個例外，交給更外圍的處理常式進一步處理。外圍一層的 try 區塊偵測到該重新拋出的例外，會在之後的 catch 處理常式處理這個例外。

常見的程式設計錯誤 24.6

如果在 catch 處理常式之外執行空的 throw 敘述，會呼叫 **terminate** 函式，放棄例外處理，立即結束程式。

圖 24.3 的程式示範如何重新拋出一個例外。在 main 的 try 區塊 (第 29–34 行) ，第 32 行呼叫 throwException 函式 (第 8–24 行)。throwException 函式也包含 try 區塊 (第 11–15 行)，由此在第 14 行的 throw 敘述拋出一個標準函式庫 exception 類別的物件。throwException 函式的 catch 處理常式 (第 16–21 行) 捕捉這個例外，印出錯誤訊息 (第 18–19 行) 再拋出例外 (第 20 行)。這中止 throwException 函式，並將控制權回到 main 函式第 32 行的 try...catch 區塊。這個 try 區塊結束 (所以第 33 行沒有執行)，main 的 catch 處理常式 (第 35–38 行) 捕捉這個例外，印出一段錯誤訊息 (第

37 行)。因爲這個範例的 catch 處理常式沒有使用例外參數，所以省略例外參數名稱，只有指定捕捉例外的型別 (第 16 行和第 35 行)。

```
1  // Fig. 24.3: fig24_03.cpp
2  // Rethrowing an exception.
3  #include <iostream>
4  #include <exception>
5  using namespace std;
6
7  // throw, catch and rethrow exception
8  void throwException()
9  {
10    // throw exception and catch it immediately
11    try
12    {
13       cout << "  Function throwException throws an exception\n";
14       throw exception(); // generate exception
15    } // end try
16    catch ( exception & ) // handle exception
17    {
18       cout << "  Exception handled in function throwException"
19          << "\n  Function throwException rethrows exception";
20       throw; // rethrow exception for further processing
21    } // end catch
22
23    cout << "This also should not print\n";
24 } // end function throwException
25
26 int main()
27 {
28    // throw exception
29    try
30    {
31       cout << "\nmain invokes function throwException\n";
32       throwException();
33       cout << "This should not print\n";
34    } // end try
35    catch ( exception & ) // handle exception
36    {
37       cout << "\n\nException handled in main\n";
38    } // end catch
39
40    cout << "Program control continues after catch in main\n";
41 } // end main
```

```
main invokes function throwException
  Function throwException throws an exception
  Exception handled in function throwException
  Function throwException rethrows exception

Exception handled in main
Program control continues after catch in main
```

圖 24.3　重新拋出例外

24.5 處理非預期的例外

unexpected 函式會呼叫以 **set_unexpected** 函式 (定義在 <exception> 標頭檔) 註冊的函式。假如沒有如此方式註冊的函式，依預設會呼叫 terminate 函式。會呼叫 terminate 函式的狀況包括：

1. 例外機制找不到符合拋出例外的 catch。
2. 在堆疊展開時，解構子要 throw 例外。
3. 在目前沒有處理例外的時候，試圖重新拋出例外。
4. 呼叫 unexpected 函式，預設會呼叫 terminate 函式。

(C++標準文件的第 15.5.1 節介紹其他的狀況)。**set_terminate** 函式可以指定呼叫函式當 terminate 被呼叫時所要呼叫的函式。否則 terminate 就會呼叫 **abort**，該函式會將程式結束，不會呼叫自動儲存空間 (automatic) 或靜態儲存空間 (static storage class) 內現存物件的解構子。當程式提早結束時，這樣的做法可能會造成資源遺漏。

常見的程式設計錯誤 24.7

因爲未捕捉的例外而忘記釋放程式的元件可能會會造成資源浪費，例如檔案串流或 I/O 裝置，這會讓其它的程式無法取得該資源。這就稱爲**「資源遺漏」(resource leak)**。

set_terminate 和 set_unexpected 函式都會傳回一個指標，分別指向函式 terminate 和 unexpected 最後呼叫的函式 (第一次會呼叫會傳回 0)。這讓你可以將此指標儲存起來，使得它稍後可以被復原。set_terminate 和 set_unexpected 函式會將函式指標當作引數，指標指向的函式傳回型別是 void 而且沒有引數。

如果程式設計者自訂終止函式的最後動作不是離開程式，，abort 函式會自動地被呼叫來結束程式的執行。

24.6 堆疊展開

當例外被拋出但卻不能在特定的範圍內被捕捉時，該函式呼叫的堆疊就會「展開」，在下一層外圍的 try atch 區塊就會嘗試 catch 這個例外。展開該函式呼叫堆疊表示，未能捕捉例外的函式會被終止，而且其中所有的區域變數都會被摧毀，控制權會返回原先呼叫這個函式的敘述。如果這個敘述位於某個 try 區塊內，就會嘗試 catch 例外。如果這個敘述沒有 try 區塊包圍時，堆疊展開就會再次發生。如果沒有任何 catch 處理常式

捕捉到這個例外， terminate 函式會被呼叫來終止該程式。圖 24.4 的程式示範堆疊展開的過程。

```
// Fig. 24.4: fig24_04.cpp
// Stack unwinding.
#include <iostream>
#include <stdexcept>
using namespace std;

// function3 throws runtime error
void function3() throw ( runtime_error )
{
   cout << "In function 3" << endl;

   // no try block, stack unwinding occurs, return control to function2
   throw runtime_error( "runtime_error in function3" ); // no print
} // end function3

// function2 invokes function3
void function2() throw ( runtime_error )
{
   cout << "function3 is called inside function2" << endl;
   function3(); // stack unwinding occurs, return control to function1
} // end function2

// function1 invokes function2
void function1() throw ( runtime_error )
{
   cout << "function2 is called inside function1" << endl;
   function2(); // stack unwinding occurs, return control to main
} // end function1

// demonstrate stack unwinding
int main()
{
   // invoke function1
   try
   {
      cout << "function1 is called inside main" << endl;
      function1(); // call function1 which throws runtime_error
   } // end try
   catch ( runtime_error &error ) // handle runtime error
   {
      cout << "Exception occurred: " << error.what() << endl;
      cout << "Exception handled in main" << endl;
   } // end catch
} // end main
```

圖 24.4　堆疊展開(1/2)

```
function1 is called inside main
function2 is called inside function1
function3 is called inside function2
In function 3
Exception occurred: runtime_error in function3
Exception handled in main
```

圖 24.4 堆疊展開(2/2)

main 的 try 區塊 (第 34–38 行) 呼叫 function1 (第 24–28 行)。然後 function1 呼叫 function2 (第 17–21 行)，再呼叫 function3 (第 8–14 行)。function3 在第 13 行拋出 runtime_error 物件。可是，因爲沒有 try 區塊包圍第 13 行的 throw 敘述，所以堆疊會展開，function3 在第 13 行結束，然後控制權傳回 function2 中呼叫 function3 的敘述 (就是第 20 行)。可是沒有 try 區塊包圍第 20 行，所以堆疊會再次展開。function2 在第 20 行結束，然後控制權傳回 function1 中呼叫 function2 的敘述 (就是第 27 行)。因爲程式第 27 行並沒有包圍的 try 區塊，所以再一次進行堆疊展開。function1 在第 27 行結束，然後控制權回到 main 中呼叫 function1 的敘述 (就是第 37 行)。第 34–38 行的 try 區塊包圍這個敘述，所以這個 try 區塊之後第一個符合的 catch 處理常式 (第 39–43 行) 捕捉這個例外並處理。第 41 行使用 what 函式顯示該例外訊息，請記得 what 函式是 exception 類別的 virtual 函式，任何衍生類別都可以多載這個函式，傳回適當的錯誤訊息。

24.7 建構子、解構子和例外處理

首先，讓我們討論一個已經提過，但尚未完全解決的問題：如果在建構子中偵測到發生錯誤，會發生什麼事情？例如，new 無法配置需要的記憶體存放物件的內部表示時，這個物件的建構子應該如何回應？因爲該建構子無法傳回代表錯誤的數值，所以必須選擇其他的方式，表示物件沒有正確地被建構。其中一個方式就是傳回建構不好的物件，希望任何使用它的使用者會予以適當的測試以發現這個物件的狀態不一致。另一個方法就是在建構子的外層設定一些變數。比較好的替代方法是要求建構子 throw 包含錯誤訊息的例外，讓程式有機會處理錯誤。

在建構子拋出例外之前，任何爲已建構完成之 物件一部份成員物件的建構子都會被呼叫。程式會呼叫 try 區塊內所有在拋出例外前所建構的自動物件的解構子。在例外處理常式開始執行時，堆疊展開保證已經完成。如果導因於堆疊展開而被呼叫的解構子拋出一個例外， terminate 會被呼叫。

如果某物件有成員物件，而且如果在外圍物件完成建構之前有個例外被拋出，則那些在例外發生之前已完成建構的成員物件的解構子會被執行。如果一個物件陣列在例外發生時已經局部建構完成，只有該陣列內已建構好的物件的解構子會被呼叫。

例外會阻止於正常情況下會釋放資源 (例如記憶體或檔案) 的程式碼的執行，因此會造成資源遺漏 (resource leak)。解決這個問題的一種技術，就是初始化一個區域物件來取得該資源。當發生例外時該物件的解構子就會被呼叫，以藉此釋放這項資源。

測試和除錯的小技巧 24.2

當一個從 new 運算式所建立之物件的建構子例外被拋出時，用於這個物件的動態配置記憶體都會被釋放。

24.8　例外和繼承

當我們建立 DivideByZeroException 類別爲 exception 類別的衍生類別的時候，各式各樣的例外類別可以自共同的基本類別衍生出來，如同第 24.3 節所曾討論過的。如果 catch 處理常式會捕捉到一個指向基本類別之例外物件的指標或參照，那它也可以捕捉所有指向公開衍生自這個基本類別的物件的指標或參照，如此可以多型地處理相關的錯誤。

測試和除錯的小技巧 24.3

使用具有例外的繼承讓例外處理常式只需使用簡潔的語法就可以捕捉一些相關的錯誤。其中一種方式是分別 catch 指向衍生例外類別物件的每一種型別的指標或參照，可是更簡潔的方式是 catch 指向基本類別物件的指標或參照。此外，捕捉指向衍生類別例外物件的指標或參照很容易出錯，尤其是當你忘記明確地測試這些衍生類別的指標或參照之型別時。

24.9　處理 new 的失敗

C++標準明定，當運算子 new 執行失敗時，會拋出 **bad_alloc** 例外 (定義在標頭檔 <new>)。本節會介紹二個執行 new 失敗的範例。第一個範例使用在失敗時會 throw bad_alloc 例外的 new 版本，第二個範例使用 **set_new_handler** 函式來處理 new 的失敗。[請注意：圖 24.5–圖 24.6 的範例配置大量的動態記憶體，會讓電腦變得很慢。]

new 在失敗時拋出 bad_alloc

圖 24.5 示範執行 new 配置所需記憶體失敗時拋出 bad_alloc 例外。try 區塊內的 for 迴圈 (第 16–20 行) 應該要執行 50 次，在每次迴圈配置有 50,000,000 個 double 值

的陣列。如果 new 失敗會拋出 bad_alloc 例外，迴圈會結束然後程式從第 22 行繼續執行，catch 處理常式捕捉到該例外並處理。第 24–25 行印出訊息「Exception occurred:」，之後是基本類別 exception 的 what 回傳的訊息 (也就是實作定義的例外訊息，例如微軟的 Microsoft Visual C++是「Allocation Failure」)。該輸出結果顯示，在 new 執行失敗之前，迴圈只執行四次就拋出 bad_alloc 例外。你的輸出可能會因你的電腦實體記憶體多寡、虛擬記憶體所需的磁碟剩餘空間，以及使用的編譯器而有所不同。

```
1  // Fig. 24.5: fig24_05.cpp
2  // Demonstrating standard new throwing bad_alloc when memory
3  // cannot be allocated.
4  #include <iostream>
5  #include <new> // bad_alloc class is defined here
6  using namespace std;
7
8  int main()
9  {
10    double *ptr[ 50 ];
11
12    // aim each ptr[i] at a big block of memory
13    try
14    {
15       // allocate memory for ptr[ i ]; new throws bad_alloc on failure
16       for ( int i = 0; i < 50; ++i )
17       {
18          ptr[ i ] = new double[ 50000000 ]; // may throw exception
19          cout << "ptr[" << i << "] points to 50,000,000 new doubles\n";
20       } // end for
21    } // end try
22    catch ( bad_alloc &memoryAllocationException )
23    {
24       cerr << "Exception occurred: "
25          << memoryAllocationException.what() << endl;
26    } // end catch
27 } // end main
```

```
ptr[0] points to 50,000,000 new doubles
ptr[1] points to 50,000,000 new doubles
ptr[2] points to 50,000,000 new doubles
ptr[3] points to 50,000,000 new doubles
Exception occurred: bad allocation
```

圖 24.5　當無法配置記憶體時 new 拋出 bad_alloc

new 在失敗時傳回 0

在記憶體配置失敗傳回 0 的較早版本的 new。C++標準明定編譯器可以使用失敗時傳回 0 的較早版本的 new。爲此，標頭檔 <new> 定義 **nothrow** 物件 (型別是 nothrow_t)，用法如下：

```
double *ptr = new( nothrow ) double[ 50000000 ];
```

這個敘述使用了配置 50,000,000 個 double 型別的陣列時不會拋出 bad_alloc (就是 nothrow) 的 new。

軟體工程的觀點 24.8

爲了讓程式更穩固，請使用失敗時會拋出 bad_alloc 例外的 new 版本。

使用 set_new_handler 函式處理 new 的錯誤

另一個處理 new 錯誤的方式就是 set_new_handler 函式 (原型在標準標頭檔 <new>)。這個函式的引數是一個函式指標，指向的函式沒有引數且傳回 void。如果 new 失敗時會呼叫這個指標指向的函式，如以一來，不管 new 是在程式中何處失敗，都能使用統一的處理方法。一旦 set_new_handler 在程式中註冊了一個 **new 處理常式 (new handler)**，new 運算子在失敗時不會拋出 bad_alloc，而是把錯誤處理交給 new 處理常式。

如果 new 成功地配置記憶體，它就會傳回指向該記憶體的指標。如果 new 無法配置所需的記憶體，而且 set_new_handler 沒有註冊一個 new 處理常式，new 就會拋出 bad_alloc 例外。如果 new 無法配置記憶體空間，並且 new 處理常式已經註冊，這個 new 處理常式就會被呼叫。C++標準明定 new 處理函式必須執行下列工作之一：

1. 刪除其它的動態配置記憶體 (或是告訴使用者關閉其它應用程式)，獲取更多可用的記憶體，然後回到 new 運算子再次配置記憶體。
2. 拋出 bad_alloc 型別的例外。
3. 呼叫 abort 或 exit 函式 (兩者在標頭檔 <cstdlib>) 來終止程式。

圖 24.6 示範如何使用 set_new_handler。customNewHandler 函式 (第 9–13 行) 印出一條錯誤訊息 (第 11 行)，然後呼叫 abort 函式 (第 12 行) 來終止該程式。該輸出結果顯示，在 new 執行失敗之前，迴圈只執行四次，然後呼叫 customNewHandler 函式。你的輸出可能會因你的電腦實體記憶體、虛擬記憶體所需的磁碟剩餘空間，以及使用的編譯器而有所不同。

```
1  // Fig. 24.6: fig24_06.cpp
2  // Demonstrating set_new_handler.
3  #include <iostream>
4  #include <new> // set_new_handler function prototype
5  #include <cstdlib> // abort function prototype
6  using namespace std;
7
8  // handle memory allocation failure
9  void customNewHandler()
10 {
11    cerr << "customNewHandler was called";
12    abort();
13 } // end function customNewHandler
14
15 // using set_new_handler to handle failed memory allocation
16 int main()
17 {
18    double *ptr[ 50 ];
19
20    // specify that customNewHandler should be called on
21    // memory allocation failure
22    set_new_handler( customNewHandler );
23
24    // aim each ptr[i] at a big block of memory; customNewHandler will be
25    // called on failed memory allocation
26    for ( int i = 0; i < 50; ++i )
27    {
28       ptr[ i ] = new double[ 50000000 ]; // may throw exception
29       cout << "ptr[" << i << "] points to 50,000,000 new doubles\n";
30    } // end for
31 } // end main
```

```
ptr[0] points to 50,000,000 new doubles
ptr[1] points to 50,000,000 new doubles
ptr[2] points to 50,000,000 new doubles
ptr[3] points to 50,000,000 new doubles
customNewHandler was called
This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.
```

圖 24.6 `set_new_handler` 指定在 `new` 失敗時呼叫的函式

24.10　unique_ptr 類別和動態記憶體配置[1]

常用的程式設計習慣是配置動態記憶體、將位址指定給某個指標、使用這個指標操作記憶體，不再需要這段記憶體時，使用 delete 釋放記憶體。如果一個例外發生在記憶體配置成功之後，且在還沒執行 delete 敘述之前，就會出現記憶體遺漏的現象。C++標準在標頭檔 **<memory>**提供類別樣版 **unique_ptr** 處理這種狀況。

unique_ptr 類別的物件擁有一個指向動態配置的記憶體的指標。當 unique_ptr 物件的解構子呼叫時 (例如當一個 unique_ptr 物件離開使用域)，它會對自己的指標成員執行 delete 動作。類別樣版 unique_ptr 多載運算子 * 和 ->，使得 nique_ptr 物件可以像一個正常的指標變數使用。圖 24.9 示範一個指向動態配置的 Integer 類別物件 的 unique_ptr 物件 (圖 24.7–24.8)。

```
1  // Fig. 24.7: Integer.h
2  // Integer class definition.
3
4  class Integer
5  {
6  public:
7     Integer( int i = 0 ); // Integer default constructor
8     ~Integer(); // Integer destructor
9     void setInteger( int i ); // functions to set Integer
10    int getInteger() const; // function to return Integer
11 private:
12    int value;
13 }; // end class Integer
```

圖 24.7　Integer 類別定義

```
1  // Fig. 24.8: Integer.cpp
2  // Member function definitions of class Integer.
3  #include <iostream>
4  #include "Integer.h"
5  using namespace std;
6
7  // Integer default constructor
8  Integer::Integer( int i )
```

圖 24.8　Integer 類別成員函式的定義(1/2)

[1] unique_ptr 類別世新版 C++ 標準的一部分，該類別已經實作於 Visual C++ 2010 以及 GNU C++。這個類別取代過時的 auto_ptr 類別。要於 GNU C++ 編譯這個程式，請使用 -std=C++0x 編譯器旗標。

```
 9      : value( i )
10   {
11      cout << "Constructor for Integer " << value << endl;
12   } // end Integer constructor
13
14   // Integer destructor
15   Integer::~Integer()
16   {
17      cout << "Destructor for Integer " << value << endl;
18   } // end Integer destructor
19
20   // set Integer value
21   void Integer::setInteger( int i )
22   {
23      value = i;
24   } // end function setInteger
25
26   // return Integer value
27   int Integer::getInteger() const
28   {
29      return value;
30   } // end function getInteger
```

圖 24.8 Integer 類別成員函式的定義(2/2)

圖 24.9 的第 15 行建立 unique_ptr 物件 ptrToInteger，並將之以一個指向動態配置的 Integer 物件的指標初始化，該物件的內容爲數值 7。第 18 行使用 unique_ptr 多載的 -> 運算子來呼叫 ptrToInteger 管理的 Integer 物件的 setInteger 函式。第 21 行使用 unique_ptr 多載的*運算子來取得 ptrToInteger 指向的值，然後用點號 (.) 運算子呼叫 Integer 物件的 getInteger 函式。unique_ptr 的多載的 -> 和 * 運算子就像一個正常的指標可被用來存取 unique_ptr 指向的物件。

```
 1   // Fig. 24.9: fig24_09.cpp
 2   // unique_ptr object manages dynamically allocated memory.
 3   #include <iostream>
 4   #include <memory>
 5   using namespace std;
 6
 7   #include "Integer.h"
 8
 9   // use unique_ptr to manipulate Integer object
10   int main()
11   {
12      cout << "Creating a unique_ptr object that points to an Integer\n";
13
14      // "aim" unique_ptr at Integer object
15      unique_ptr< Integer > ptrToInteger( new Integer( 7 ) );
```

圖 24.9 unique_ptr 物件管理動態配置記憶體(1/2)

```
16
17      cout << "\nUsing the unique_ptr to manipulate the Integer\n";
18      ptrToInteger->setInteger( 99 ); // use unique_ptr to set Integer value
19
20      // use unique_ptr to get Integer value
21      cout << "Integer after setInteger: " << ( *ptrToInteger ).getInteger()
22   } // end main
```

```
Creating a unique_ptr object that points to an Integer
Constructor for Integer 7

Using the unique_ptr to manipulate the Integer
Integer after setInteger: 99

Destructor for Integer 99
```

圖 24.9　unique_ptr 物件管理動態配置記憶體(2/2)

因爲 ptrToInteger 是 main 的一個區域自動的變數，所以在 main 函式結束後就會被摧毀。unique_ptr 的解構子會強迫 delete 由 ptrToInteger 指標所指向的 Integer 物件，然後會呼叫 Integer 類別的解構子。無論控制權如何離開該區塊 (例如，透過 return 敘述或某個例外)，Integer 佔用的記憶體都會被釋放掉。最重要的是，這項技術可以防止記憶體遺漏。例如，假設函式傳回一個指向某物件的指標。很不幸地，接收這個指標的函式呼叫程式可能不會 delete 物件，因此產生記憶體遺漏。可是，如果這個函式傳回 unique_ptr 予該物件， 當 unique_ptr 物件的解構子被呼叫時，這個物件就會自動刪除。

unique_ptr 一次只能擁有一個動態配置物件，且該物件不能是陣列。透過多載的指定運算子或複製建構子，unique_ptr 可以轉移它管理之動態記憶體的所有權。最後一個保有指向動態記憶體的指標的 unique_ptr 物件會刪除該記憶體。因此 unique_ptr 是一種適於傳回動態配置的記憶體給客戶端程式碼的機制。當 unique_ptr 離開客戶端的程式碼使用域，unique_ptr 的解構子便會刪除動態記憶體。

24.11　標準函式庫的例外階層

經驗告訴我們，例外可很好地區分成許多的種類。C++ 標準函式庫包括一個例外類別的階層，圖 24.10 列出該階層的一部分。如同我們曾在第 24.2 節介紹過的，這個階層的最上層是基本類別 exception (定義在標頭檔<exception>)，該類別包含 virtual 函式 what，衍生類別可以重載這個函式，產生適合的錯誤訊息。

基本類別 exception 直接衍生出 runtime_error 和 **logic_error** 類別 (兩者都定義在標頭檔 <stdexcept>)，這兩個類別也都有好幾個衍生類別。同樣衍生自 exception 的還有 C++運算子拋出的例外，例如，new 拋出的 bad_alloc (第 24.9 節)、

dynamic_cast 所拋出的 **bad_cast**(第 21 章) 和 typeid 拋出的 **bad_typeid**(第 21 章)。在某個函式的 throw 清單納入 **bad_exception**，表示如果發生非預期的例外，unexpected 函式可以拋出 bad_exception 而非終止該程式的執行 (預設的行爲)，或是呼叫 set_unexpected 所指定的另一個函式。

常見的程式設計錯誤 24.8

將一個能夠捕捉基本類別物件的 catch 處理常式，放在只能夠捕捉衍生類別物件的 catch 處理常式之前，是一種邏輯錯誤。捕捉基本類別物件的 catch 處理常式也會捕捉所有衍生自基本類別的物件，所以捕捉衍生類別物件的 catch 處理常式永遠不會被執行。

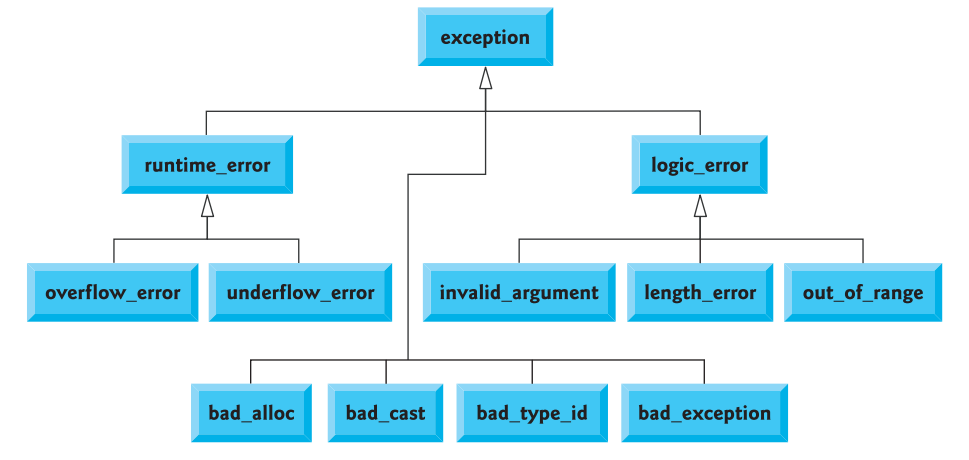

圖 24.10　標準函式庫例外類別的一部分

類別 logic_error 是幾個標準例外類別的基本類別，它會指出程式邏輯的錯誤。例如，**invalid_argument** 類別指出某個傳給函式的引數是無效的。(適當的程式碼當然可以防止將無效引數傳給函式)。**engthh_error** 類別指出所使用的長度大於這個物件允許操作的最大值。**out_of_range** 類別指出傳給陣列的下標超出範圍。

在第 24.8 節介紹的 runtime_error 是其他指出執行時期錯誤之例外類別的基本類別。例如，**overflow_error** 類別描述**算術溢位 (arithmetic overflow error**，就是數學運算的結果超過電腦能儲存的最大數值)，**underflow_error** 類別描述**算術短值 (arithmetic underflow error**，就是數學運算的結果小於電腦能儲存的最小數值)。

常見的程式設計錯誤 24.9

例外類別不一定要衍生自 exception 類別，因此，捕捉型別 exception 並不保證會 catch 程式所有可能遭遇的例外。

測試和除錯的小技巧 24.4

要 catch try 區段拋出的所有例外，請使用 catch(...)。這種捕捉例外的方式有一個缺點，就是在編譯時間不知道被捕捉到之例外的型別。另外一個缺點就是，沒有具名參數的話，無法參照到例外處理常式內部的例外物件。

軟體工程的觀點 24.9

標準 exception 階層是建立例外的好起點，你寫的程式可以設計一個可以 throw 標準例外、throw 標準例外衍生的例外或 throw 自己的例外的程式。

軟體工程的觀點 24.10

使用 catch(...)執行與例外型別無關的回復動作 (例如，釋放一般的資源)。該例外可以重新被拋出以提醒更專門的 catch 處理常式。

24.12　總結

在本章中，你學到了如何使用例外處理機制處理程式中產生的錯誤。你學到了如何利用例外處理機制，將錯誤處理的程式碼從程式的主要執行過程中移除。我們使用除零範例來介紹例外處理。我們也展示了，要如何使用 try 區塊，將可能丟出例外的程式碼包圍起來，以及如何使用 catch 處理常式來處理這些可能發生的例外。你學到了如何重新丟出例外，以及處理建構子中發生的例外。本章中討論了 new 失敗的處理、使用 unique_ptr 類別來配置動態記憶體，以及標準函式庫例外階層。

摘要

24.1　簡介

- 例外是程式執行時遇到問題的表徵。
- 例外處理讓你寫的程式，可以解決執行時期的問題，如果沒有遇到問題的話，就會繼續執行。更嚴重的問題可能在控制的情況下結束之前，需要通知使用者。

24.2 範例：處理除零動作

- exception 類別是例外的標準基本類別。exception 類別提供 virtual 函式 what，可以讓衍生類別覆蓋，傳回適合的錯誤訊息。
- runtime_error 類別 (定義在標頭檔 <stdexcept>) 在 C++標準類別中表示執行時期錯誤。
- C++使用例外處理的終止模式。
- try 區塊是由 try 關鍵字加上大括號({})所組成，定義可能會產生例外的程式碼區塊。try 區塊包含可能造成例外的敘述，以及發生例外就不應該執行的程式碼。
- 每個 try 區域之後至少要有一個 catch 處理常式。每個 catch 處理常式會在小括號中，指定可以處理的例外類型。
- 如果例外參數包含選用性的參數名稱，則 catch 處理常式可以使用該參數名稱來與捕捉的例外物件進行互動。
- 程式發生例外的位置稱爲拋出點。
- 如果 try 區塊中發生例外，這個 try 區塊就會結束，程式的控制權會轉移到第一個例外，於此該例外參數的型別符合被拋出例外的型別。
- try 區塊結束時，該區塊內定義的區域變數會離開使用域。
- try 區塊因爲例外而結束時，程式會尋找第一個符合此例外型別的 catch 處理常式。如果例外的型別與拋出例外的型別相同，或者例外的型別是例外參數型別的衍生類別，則是符合的情況。找到符合的情況時，程式會執行對應的 catch 處理常式。
- 當 catch 處理常式完成時，catch 的參數以及定義在 catch 處理常式的區域變數會離開使用域。其餘相應於該 try 區塊的 catch 處理常式都會被忽略，並且 try … catch 之後的程式會碼的第一行會開始執行。
- 如果 try 區塊沒有發生例外，則程式會忽略該區塊的 catch 處理常式。程式會繼續執行 try … catch 區塊之後的敘述。
- 如果發生在 try 區塊內的例外沒有符合的 catch 處理常式，或者發生於敘述的例外並不是在 try 區塊內，則包含該敘述的函式會立刻終止，然後該程式會試著於呼叫函式內尋找是否有 try 區塊。這個過程稱爲堆疊展開。
- 要拋出一個例外，請使用 throw 關鍵字和一個代表要拋出例外的型別的運算元。throw 的運算元可以是任何型別。

24.3 何時使用例外處理

- 例外處理的是設計來處理同步的錯誤，意即敘述執行時發生的錯誤。
- 例外處理不是設計來處理非同步事件的問題，非同步事件和程式控制流程平行且獨立執行。

24.4　重新拋出例外

- catch 處理常式可以將例外處理 (或是其中一部分) 延後，交給另一個處理常式。無論如何，處理常式可以將例外重新拋出。
- 常見的例子就是超出陣列下標的範圍、算術溢位、除零、錯誤的函式參數，以及記憶體配置失敗。

24.5　處理非預期的例外

- unexpected 函式會呼叫 set_unexpected 函式註冊的函式。假如沒有如此註冊的函式，程式預設會呼叫 terminate 函式。
- set_terminate 函式可以指定呼叫函式 terminate 時要呼叫的函式，否則 terminate 會呼叫 abort，結束程式而且不會呼叫宣告成 static 和 auto 物件的解構子。
- set_terminate 和 set_unexpected 函式都會傳回一個指標，分別指向函式 terminate 和 unexpected 最後呼叫的函式 (第一次會呼叫會傳回 0)。你就可以將此指標儲存起來，方便以後復原時使用。
- set_terminate 和 set_unexpected 函式會將函式指標當作引數，指標指向的函式傳回型別是 void 而且沒有引數。
- 如果程式設計者自訂終止函式的最後動作不是離開程式，在執行完自訂終止函式之後，會呼叫 abort 函式結束程式。

24.6　堆疊展開

- 函式呼叫堆疊展開表示，未能捕捉例外的函式會被終止，而且其中所有區域變數都會被刪除，控制權會返回原先呼叫這個函式的敘述。

24.7　建構子、解構子和例外處理

- 如果在拋出例外之前已經建構部分的物件，建構子拋出的例外會呼叫這些物件的解構子。
- 程式會呼叫 try 區塊在拋出例外前所建構的所有自動物件的解構子。
- 在例外處理開始執行時，堆疊展開已經完成。
- 如果在堆疊展開時拋出例外而呼叫解構子，程式會呼叫 terminate。
- 如果某物件有成員物件，而且在外圍物件完成建構之前拋出例外，就會執行在例外發生之前已完成建構的成員物件的解構子。
- 如果一個物件陣列在例外發生時已經部分建構完成，程式只會呼叫陣列內已建構之物件的解構子。
- 從 new 運算式建立物件的建構子中拋出例外時，這個物件動態配置的記憶體都會被釋放。

24.8 例外和繼承

- 如果 catch 處理常式會捕捉基本類別例外物件的指標或參照，那也可以捕捉所有明確繼承這個基本類別的物件或參照，因此可以處理相關的錯誤。

24.9 處理 new 的失敗

- C++標準規範指明運算子 new 執行失敗時，會拋出 bad_alloc 例外，該例外定義在標頭檔 <new>。
- 函式 set_new_handler 的引數是一個函式指標，指向的函式沒有引數且傳回 void。如果 new 失敗時會呼叫這個指標指向的函式，
- 一旦 set_new_handler 在程式中註冊了 new 處理常式，new 運算子在錯誤時不會拋出 bad_alloc，而是把錯誤處理交給 new 處理常式。
- 如果 new 成功配置記憶體，就會傳回指向該記憶體的指標。
- 在記憶體配置成功，但是還沒執行 delete 敘述之前，如果發生例外就會產生記憶體遺漏。

24.10 unique_ptr 類別和動態記憶體配置

- C++標準函式庫提供類別樣版 unique_ptr 處理記憶體遺漏。
- unique_ptr 類別的物件擁有一個指標，指向動態配置的記憶體。unique_ptr 物件的解構子會對自己的指標成員執行 delete 動作。
- 類別樣版 unique_ptr 多載運算子 * 和 ->，所以可以像正常的指標變數一樣使用。unique_ptr 也可以透過多載的指定運算子和複製建構子，轉移管理的動態記憶體所有權。

24.11 標準函式庫的例外階層

- C++標準函式庫包括例外類別的階層。這個階層的最上曾是基本類別 exception。
- 基本類別 exception 直接衍生出 runtime_error 和 logic_error 類別 (兩者都定義在標頭檔 <stdexcept>)，這兩個類別也都有衍生類別。
- 有些運算子會拋出標準例外，例如 new 運算子拋出 bad_alloc、dynamic_cast 運算子拋出 bad_cast、typeid 運算子拋出 bad_typeid。
- 函式的 throw 清單包含 bad_exception，表示如果發生非預期的例外，unexpected 函式可以拋出 bad_exception 而非終止程式，或是呼叫 set_unexpected 指定的另一個函式。

術語

abort 函式 (abort function)
算術溢位錯誤 (arithmetic overflow error)
算術短值錯誤 (arithmetic underflow error)
非同步事件 (asynchronous event)
auto_ptr 類別樣版 (auto_ptr class template)
bad_alloc 例外 (bad_alloc exception)
bad_cast 例外 (bad_cast exception)

bad_exception 例外　(bad_exception exception)
bad_typeid 例外　(bad_typeid exception)
catch 處理常式　(catch handler)
catch 關鍵字　(catch keyword)
空的例外規格　(empty exception specification)
例外　(exception)
exception 類別　(exception class)
例外處理常式　(exception handler)
例外處理　(exception handling)
<exception>標頭檔　(<exception> header file)
例外物件　(exception object)
例外參數　(exception parameter)
例外規格　(exception specification)
容錯的程式　(fault-tolerant program)
invalid_argument 例外　(invalid_argument exception)
length_error 例外　(length_error exception)
logic_error 例外　(logic_error exception)
<memory> 標頭檔　(<memory> header file)
new 處理常式　(new handler)
nothrow 物件　(nothrow object)
out_of_range 例外　(out_of_range exception)
overflow_error 例外　(overflow exception)
資源遺漏　(resource leak)
例外處理的恢復模式　(resumption model of exception handling)
重新拋出例外　(rethrowing the exception)
堅固的程式　(robust program)
runtime_error 例外　(runtime_error exception)
set_new_handler 函式　(set_new_handler function)
set_terminate 函式　(set_terminate function)
set_unexpected 函式　(set_unexpected function)
堆疊展開　(stack unwinding)
<stdexcept> 標頭檔案　(<stdexcept> header file)
同步錯誤　(synchronous error)
terminate 函式　(terminate function)
例外處理的終止模式　(termination model of exception handling)
throw
拋出例外　(throws an exception)
throw 關鍵字　(throw keyword)
throw 清單　(throw list)
拋出點　(throw point)
try 區塊　(try block)
try 關鍵字　(try keyword)
underflow_error 例外　(underflow_error exception)
unexpected 函式　(unexpected function)
exception 類別的 virtual 函式 what　(what virtual function of class exception)

自我測驗

24.1　例外處理常式允諾了哪些事？

24.2　請解釋爲什麼不把例外處理的技術用在傳統程式控制上。

24.3　爲何例外適合用來處理函式庫函式產生的錯誤？

24.4　什麼是「資源遺漏」？

24.5 如果 try 區塊沒有拋出例外，在執行完 try 區塊後，程式的控制權會移交到何處？

24.6 如果例外是從 try 區塊之外拋出，程式會如何處理？

24.7 請說明使用 catch(...)主要的優點和缺點。

24.8 如果沒有 catch 處理函式符合拋出物件的型別時會發生什麼事情呢？

24.9 如果有幾個 catch 處理函式同時符合拋出物件的型別，會發生什麼事情呢？

24.10 爲什麼你會指定基本類別的型別爲 catch 處理常式的型別，然後拋出衍生類別型別的物件呢？

24.11 假設存在某個與例外物件型別完全吻合的 catch 處理常式。則必須在何種環境之下，才會由另一個不同的處理常式來處理該型別的例外？

24.12 拋出例外一定會終止程式嗎？

24.13 當 catch 處理常式 throw 例外時，會發生什麼事情呢？

24.14 敘述 throw; 有何作用？

自我測驗解答

24.1 它允諾提供更穩固與容錯的程式，該程式能面對問題並持續執行或優雅地中止。

24.2 (a) 例外處理是設計來處理不常發生的狀況，而這些狀況常會導致程式終止，所以我們不會要求編譯器的設計者將例外處理設計的很完美。 (b) 傳統的流程控制通常比例外處理更清楚，也更有效率。 (c) 發生例外時，因爲堆疊展開可能會產生問題，且在例外發生之前所配置的資源可能無法釋放。 (d)「額外的」例外會讓程式設計者更難以處理大量的例外情況。

24.3 要求函式庫函式的錯誤處理方式能夠滿足所有使用者特殊的需求是不可能的。

24.4 突然結束的程式會造成其他程式無法獲得資源，或者是自己無法重新取得「遺漏的」資源。

24.5 程式會跳過 try 區塊後續所有 catch 區塊的例外處理常式，然後將執行權交給最後一個 catch 區塊的下一個敘述。

24.6 如果從 try 區塊之外拋出例外，程式會呼叫函式 terminate 終止程式。

24.7 catch(...)的格式可以捕捉 try 區塊拋出的任何型別例外。優點是可以捕捉所有可能的例外。缺點是這種 catch 沒有參數，因此無法引用拋出物件的資訊，也無法知道發生錯誤的原因。

24.8 這會使得程式繼續在外層下一個 try 區塊，找尋符合的處理常式。如果繼續下去，程式最後會發現沒有處理常式符合拋出物件的型別；在此狀況下，程式會呼叫終止函式

terminate，而 terminate 函式預設會呼叫 abort 函式。terminate 函式的另一個取代方法，就是將另一個函式當作 set_terminate 函式的引數。

24.9 try 區塊之後第一個符合的 catch 區塊會執行。

24.10 這是一個 catch 相關例外型別的好方法。

24.11 基本類別的處理常式可以捕捉所有的衍生類別物件。

24.12 不會，但是拋出例外的區塊會結束。

24.13 這個例外會由另外一個 catch 處理常式 (如果存在的話) 來處理，而此 catch 處理常式又與偵測例外的 try 區塊 (如果存在) 相關聯。

24.14 如果例外出現在 catch 處理常式，會重新拋出這個例外，否則會呼叫 unexpected 函式。

習題

24.15（例外情況） 列出本文介紹的各種例外發生條件，儘可能地列出各種例外情形，對於每個例外請簡短描述，程式通常會如何利用本章討論的技術處理。一些典型的例外包括除零、算術運算溢位、陣列附標超出範圍、儲存空間用完等等。

24.16（Catch 參數） 在何種狀況之下，你在處理常式中定義要捕捉的物件型別時，不會提供參數名稱？

24.17（throw 敘述） 包含下列敘述的程式

```
throw;
```

你一般認爲這樣的敘述式會出現在程式的哪個部分？如果該敘述是出現在程式的其他不同位置，後果又是如何？

24.18（例外處理與其他方式） 將例外處理方法與本書所討論的其他錯誤處理方式加以比較。

24.19（例外除理與程式控制） 爲什麼不應該把例外當成另一種程式流程控制？

24.20（new 運算子失敗） 如果 new 運算子失敗，會拋出什麼樣的例外？

24.21（從 catch 拋出例外） 假設程式 throw 一個例外，而且適當的例外處理常式開始執行。現在假設例外處理常式本身也 throw 同樣的例外。這會產生無窮遞迴嗎？請撰寫一個程式驗證你的觀點。

24.22（捕捉除零例外） 請撰寫一個程式捕捉出現除零的例外。

24.23（拋出條件運算式的結果） 試撰寫一個程式，拋出條件運算式的結果，該結果可能爲 double 或 int。提供 int 的 catch 處理常式，以及 double 的 catch 處理常式。請說明不論傳回的是 int 或 double，程式都只會執行 double 的 catch 處理常式。

24.24（區域變數解構子） 撰寫一個程式，說明在區塊拋出例外之前，程式會呼叫區塊中所有建構過的物件的解構子。

24.25（成員物件解構子）撰寫一個程式，說明在例外發生之前，程式只會呼叫已經建構好的成員物件的解構子。

24.26（捕捉所有的例外）撰寫一個程式，示範 `catch(...)`處理常式捕捉一些例外型別。

24.27（例外處理常式的順序）撰寫一個程式，說例外處理常式的排列順序是很重要的。程式會執行第一個符合的處理常式。以兩種不同的方法編譯和執行你的程式，說明這兩種不同處理常式的不同效果。

24.28（拋出例外的建構子）撰寫一個程式，說明建構子如何將執行失敗的資訊，傳遞給 `try` 區塊後續的處理程式。

24.29（再次拋出例外）請寫一個程式，示範再次拋出例外。

24.30（未補捉的例外）撰寫一個程式，說明函式雖然擁有自己的 `try` 區塊，但是不一定能夠捕捉 `try` 區塊中所產生的每個可能錯誤。一些例外可能會漏失到外面的區域，讓外界的處理常式處理。

24.31（堆疊展開）撰寫一個程式，從深層巢狀結構內的函式拋出例外，然後在外圍的 `try` 區塊後續的 `catch` 處理常式加以捕捉並且處理。