

前置處理器

E

Hold thou the good; define it well.

—Alfred, Lord Tennyson

*I have found you an argument;
but I am not obliged to find you
an understanding.*

—Samuel Johnson

*A good symbol is the best
argument, and is a missionary to
persuade thousands.*

—Ralph Waldo Emerson

*Conditions are fundamentally
sound.*

—Herbert Hoover [December
1929]

學習目標

在本章中，你將學到：

- 使用 `#include` 來開發大型程式。
- 使用 `#define` 來建立巨集和具有引數的巨集。
- 了解條件式編譯。
- 在條件式編譯的過程中顯示錯誤訊息。
- 使用斷言 (assertion) 來測試運算式的數值是否正確。



本章綱要

- E.1 簡介
- E.2 `#include` 前置處理器命令
- E.3 `#define` 前置處理器命令：符號常數
- E.4 `#define` 前置處理器命令：巨集
- E.5 條件式編譯
- E.6 `#error` 和 `#pragma` 前置處理器命令
- E.7 `#` 和 `##` 運算子
- E.8 事先定義的符號常數
- E.9 斷言
- E.10 總結

摘要 | 術語 | 自我測驗 | 自我測驗解答 | 習題

E.1 簡介

本章將介紹**前置處理器 (preprocessor)**。前置處理會在程式編譯之前進行。一些可能的動作包括了：將其它的檔案含入編譯的檔案中、定義**符號常數 (symbolic constant)**和**巨集 (macro)**、程式碼的**條件式編譯 (conditional compilation)**、以及**前置處理器命令的條件式執行 (conditional execution of preprocessor directives)**。所有前置處理器命令都會以`#`開始，並且同一行前置處理器命令之前只能夠出現空白字元。前置處理命令不是 C++敘述，因此不以分號 (`;`) 做結。前置處理器命令必須在編譯開始之前完成處理。



常見的程式設計錯誤 E.1

在前置處理器命令之後放置一個分號，可能會導致各種錯誤（視前置處理器命令的類別而定）。



軟體工程的觀點 E.1

許多前置處理器功能（特別是巨集）比較適合 C 程式設計師而非 C++程式設計師。C++程式設計師應該熟悉前置處理器的使用，因為他們可能需要利用傳統的 C 程式碼。

E.2 #include 前置處理器命令

本書已經使用過 **#include 前置處理器命令 (#include preprocessor directive)**。這個#include 命令會將特定檔案複製到命令所在的位置。#include 命令具有兩種格式：

```
#include <filename>
#include "filename"
```

這兩種格式的差異在於前置處理器搜尋含入檔案的位置。如果檔案名稱以角括號<和>括起來（用於標準的函式庫標頭檔，standard library headers），則搜尋通常以實作環境相依的方式來執行，通常是在事先指定的資料夾中。如果檔案名稱以雙引號括起來，則前置處理器會先到編譯檔案所在的目錄底下搜尋指定的含入檔，接著才會用角括號括起來的檔案一樣的方式，以實作環境相依的方式來搜尋。這種方式通常會用來含入程式設計師自定的標頭檔。

#include 命令可用來含入標準標頭檔，像是 <iostream> 和 <iomanip>。#include 命令也可以用於含有數個原始碼的程式，而這些程式會一起編譯。標頭檔含有不同程式檔中的共同宣告和定義，此標頭檔通常會建立並且含入程式檔中。這種宣告和定義的例子就是類別、結構、union、列舉型別、函式原型、常數以及串流物件（例如 cin）。

E.3 #define 前置處理器命令：符號常數

#define 前置處理器命令 (#define preprocessor directive) 可以建立**符號常數 (symbolic constants)**，以符號來表示常數）以及巨集 (macros，定義成符號的運算)。#define 前置處理器命令的格式如下：

```
#define identifier replacement-text
```

當這一行出現在某個檔案時，之後所有該識別字 (identifier) 出現的位置（字串中的除外），都會在編譯之前自動取代成代換文字 (replacement-text)。例如

```
#define PI 3.14159
```

會將之後所有出現的符號常數 PI，都替換成數字常數 3.14159。符號常數讓你能夠為常數命名，並且在程式中使用該名稱。如果該常數需要更改，則只需要更改 #define 命令即可，當程式重新編譯時，程式中所有的常數都會被修改。[請注意：所有位於符號常

E-4 C++程式設計藝術(第七版)(國際版)

數名稱右邊的文字，都會取代該符號常數。]例如，`#define PI = 3.14159` 會讓前置處理器將每個出現識別字 `PI` 的地方都換成 `= 3.14159`。這種替換方式是許多邏輯和語法錯誤的原因。] 將某個符號常數重新定義成新的數值，而沒有先解除它的定義，也是一種錯誤。注意，在 C++ 中，我們比較建議你使用 `const` 變數而非符號常數。`const` 變數有特定的資料型別，而且可以由除錯器看到其名稱。一旦符號常數以替代文字取代之後，除錯器就只能看到替代文字了。`const` 變數的缺點是需要其資料型別大小的記憶體位置，符號常數則不需要額外的記憶體。



常見的程式設計錯誤 E.2

在沒有定義符號常數的檔案中，使用該符號常數是一種編譯錯誤（除非它們從標頭檔被 `#included` 進去了）。



良好的程式設計習慣 E.1

使用有意義的名稱來做為符號常數，將有助於程式本身的註解能力。

E.4 `#define` 前置處理器命令：巨集

[請注意：本節是為需要用到傳統 C 程式碼的 C++ 程式設計師而加入的。在 C++ 中，巨集通常可以用樣板和行內函式來取代。] 巨集 (macro) 是由 `#define` 前置處理器命令所定義的運算。**巨集識別字 (macro-identifier)** 與符號常數一樣，程式裡的巨集識別字也會在編譯之前替換成**代換文字 (replacement-text)**。巨集可以定義成具有**引數 (arguments)** 或不具有引數。沒有引數的巨集的處理方式與符號常數一樣。在具有引數的巨集中，引數會先代換到替換文字中，然後才將巨集展開 (expand)，也就是說，代換文字會取代程式中的巨集識別字和引數。巨集引數沒有型別檢查的功能。巨集僅供文字替換之用。

考慮以下具有一個引數的巨集定義，其功用為計算圓的面積：

```
#define CIRCLE_AREA( x ) ( PI * ( x ) * ( x ) )
```

每當 `CIRCLE_AREA (y)` 出現在檔案中時，`y` 的值將取代代換文字中的 `x`，符號常數 `PI` 將取代成它的數值（先前定義過），並且在程式中展開此巨集。例如，以下的敘述

```
area = CIRCLE_AREA( 4 );
```

會展開成

```
area = ( 3.14159 * ( 4 ) * ( 4 ) );
```

這個運算式只包含常數，因此在編譯時期就會計算運算式的值，然後在執行時期將結果設定給 `area`。當巨集引數為運算式時，代換文字中每個 `x` 和整個運算式外圍的小括號可以確保正確的運算順序。例如，以下的敘述

```
area = CIRCLE_AREA( c + 2 );
```

會展開成

```
area = ( 3.14159 * ( c + 2 ) * ( c + 2 ) );
```

這可以正確運算，因為小括號確保正確的運算順序。如果省略代換文字中的小括號，則巨集會展開成

```
area = 3.14159 * c + 2 * c + 2;
```

這相當於

```
area = ( 3.14159 * c ) + ( 2 * c ) + 2;
```

因為運算子的優先權，所以其結果是錯的。



常見的程式設計錯誤 E.3

忘記在代換文字中，將巨集引數用小括號包起來可能會造成錯誤。

巨集 `CIRCLE_AREA` 也可以定義成函式。以下的函式 `circleArea`：

```
double circleArea( double x ) { return 3.14159 * x * x; }
```

會與巨集 `CIRCLE_AREA` 執行相同的計算，但是它會因函式呼叫而產生一些額外的負擔。巨集 `CIRCLE_AREA` 的優點就是巨集會將程式碼直接加入程式中，這避免函式呼叫的額外負擔，因為巨集 `CIRCLE_AREA` 的定義是獨立的，且名稱是有意義的，所以程式仍然會保持相當的可讀性。其缺點就是它的引數需要計算兩次。而且，每次在程式中出現巨集時，就要將它展開。假如此巨集很大，就會大幅增加程式的長度。因此，在執行速度和程式大小（假如磁碟空間很小）之間存在有取捨關係。注意，行內函式（第 6 章）可以同時擁有巨集的效能以及函式的軟體工程優點。



增進效能的小技巧 E.1

在執行時期之前，巨集有時候能以行內程式碼取代函式。這可以消除函式呼叫的額外負擔。我們比較建議使用行內函式來取代巨集，因為它們提供了函式的型別檢查服務。

E-6 C++程式設計藝術(第七版)(國際版)

以下的巨集定義有 2 個引數，其功用為計算矩形的面積：

```
#define RECTANGLE_AREA( x, y ) ( ( x ) * ( y ) )
```

每當程式中出現 `RECTANGLE_AREA(a, b)` 時，`a` 和 `b` 的值會取代巨集代換文字中的 `a` 和 `b`，並且每個巨集名稱都展開這個巨集。例如，以下的敘述

```
rectArea = RECTANGLE_AREA( a + 4, b + 7 );
```

會展開成

```
rectArea = ( ( a + 4 ) * ( b + 7 ) );
```

程式會計算運算式的數值，並且將其結果指定給 `rectArea` 變數。

巨集或符號常數的代換文字通常是在 `#define` 命令同一行中，位於識別字之後的所有文字。如果巨集或符號常數的代換文字太長，以致於無法於一行內撰寫完成，則你必須在行末加上一個反斜線(`\`)，來表示下一行仍然是代換文字 (最後一行不用)。

符號常數和巨集可以使用 **`#undef` 前置處理器命令**來移除。`#undef` 命令會取消某個符號常數或巨集名稱的定義。符號常數或巨集的範圍 (scope) 是從它定義的位置開始，一直到以 `#undef` 取消定義，或直到檔案結束為止。一旦某個名稱取消定義之後，該名稱可以再以 `#define` 重新定義。

請注意，具有副作用的運算式 (也就是會更改變數值) 不應該傳給巨集，因為巨集的引數可能會計算一次以上。



常見的程式設計錯誤 E.4

巨集經常會替換到原本不是用來做為巨集，但是正好相同的名稱。這可能會造成意外而難解的編譯和語法錯誤。

E.5 條件式編譯

條件式編譯 (conditional compilation) 讓你能夠控制前置處理器命令的執行，以及程式碼的編譯。條件前置處理器命令會計算某個常數整數運算式，然後判斷是否應執行程式碼。強制轉換運算式、`sizeof` 運算式、以及列舉常數都不能夠在前置處理器命令中進行計算，因為它們都是由編譯器和編譯前的前置處理所決定的。

條件式前置處理器的結構很類似 `if` 選擇結構。請考慮以下的前置處理器命令碼：

```
#ifndef NULL
#define NULL 0
#endif
```

它會判斷符號常數 `NULL` 是否已經定義過了。假如 `NULL` 沒有被定義，則運算式 `#ifndef NULL` 會含括到 `#endif` 的運算式，假如 `NULL` 已經被定義了，則會跳過這段程式碼。每個 `#if` 結構都必須以 `#endif` 結束。命令 `#ifdef` 和 `#ifndef` 是 `#if defined(name)` 和 `#if !defined(name)` 的縮寫。多重的條件前置處理器結構可以使用 `#elif` (相當於 `if` 結構中的 `else if`) 和 `#else` (相當於 `if` 結構中的 `else`) 命令來進行測試。

在程式的開發過程中，程式設計師通常會將一大段的程式碼變成「註解」以免編譯它。如果程式碼含有 C 風格的註解，則我們不能以 `/*` 和 `*/` 來將整段程式註解，因為碰到的第一個 `*/` 會結束註解。取而代之地，程式設計師可以使用以下的前置處理器結構：

```
#if 0
    code prevented from compiling
#endif
```

若想編譯這段程式碼，只要將 `0` 換成 `1` 即可。

條件式編譯常用來幫助偵錯。輸出敘述通常可以用來印出變數值以及確認控制流程。輸出敘述可以包含在條件式前置處理器命令中，所以這種敘述式只有在偵錯過程尚未結束時才會進行編譯。例如

```
#ifdef DEBUG
    cerr << "Variable x = " << x << endl;
#endif
```

如果在命令 `#ifdef DEBUG` 之前定義了符號常數 `DEBUG`，則程式中的 `cerr` 敘述式才會編譯。這個符號常是通常是由命令列編譯器或是 IDE (例如 Visual Studio) 所設定，而不是由 `#define` 定義的。當偵錯完成之後，程式會將 `#define` 命令從原始檔中移除，而用來協助偵錯的輸出敘述式，在編譯期間會被忽略。在較大型的程式中，我們可能希望定義幾個不同的符號常數，來控制原始檔的不同區段的條件式編譯。



常見的程式設計錯誤 E.5

在只能夠存放一個敘述式的位置，加入數個協助偵錯的條件式編譯輸出敘述式，這樣會導致語法錯誤或邏輯錯誤。在這種情況下，條件式編譯的敘述式應該使用大括號構成的複合敘述式。因此，當編譯加上偵錯敘述式的程式時，程式的控制流程才不會改變。

E.6 #error 和 #pragma 前置處理器命令

命令 #error

```
#error tokens
```

會印出與實作環境相依的訊息，其中含有命令中指定的 token。token 是一連串以空白分隔的字元。例如

```
#error 1 - Out of range error
```

包含 6 個 token。在某個常用的 C++編譯器中，當處理 #error 命令時，命令中的 token 將列印出來成為錯誤訊息，前置處理動作停止，並且程式不會進行編譯。

命令 #pragma

```
#pragma tokens
```

會產生實作環境定義的動作。實作環境無法辨識的 pragma 會被忽略。例如，某個特定的 C++編譯器可以辨識出 pragma，讓你利用編譯器的特殊功能。若想瞭解更多關於 #error 和 #pragma 的資訊，請查閱你所使用 C++的說明文件。

E.7 # 和 ## 運算子

和 ## 這兩個前置處理器運算子只有 C++和 ANSI/ISO C 才能夠使用。# 運算子會將某個代換文字 token 轉換成由雙引號包圍起來的字串。請看以下的巨集定義：

```
#define HELLO( x ) cout << "Hello, " #x << endl;
```

當 HELLO(John)出現在程式檔案時，它會展開成

```
cout << "Hello, " "John" << endl;
```

字串 "John" 會取代代換文字中的#x。在前置處理期間，由空白字元所分隔的字串會串接起來，所以上述的敘述式相當於

```
cout << "Hello, John" << endl;
```

請注意，# 運算子必須用於具有引數的巨集，因為#的運算元會參考巨集的某個引數。

運算子可用來串接兩個 token。請看以下的巨集定義：


```
cout << "Hello, John" << endl;
#define TOKENCONCAT( x, y ) x ## y
```

當 `TOKENCONCAT` 出現在程式中時，它的兩個引數會串接在一起，然後取代該巨集。例如，程式中的 `TOKENCONCAT(0 ,K)` 將會取代成 `OK`。## 運算子必須有兩個運算元。

E.8 事先定義的符號常數

有六個事先定義的符號常數 (predefined symbolic constants，圖 E.1)。這些符號常數的識別字都是以兩個底線字元開始和結束 (`__cplusplus` 除外)。這些識別字和前置處理運算子 (第 E.5 節)，都不能用於 `#define` 和 `#undef` 命令。

符號常數	說明
<code>__LINE__</code>	目前原始碼的行數 (整數常數)。
<code>__FILE__</code>	原始程式檔案的假設名稱 (字串)。
<code>__DATE__</code>	原始程式檔編譯時的日期 (字串格式為 <code>Mmm dd yyyy</code> ，例如 <code>Aug 19 2002</code>)。
<code>__STDC__</code>	指出程式是否支援 ANSI/ISO C 標準。假如支援，其值為 1；否則未定義。
<code>__TIME__</code>	原始檔案編譯的時間 (一個字串常數，其格式為 <code>hh:mm:ss</code>)。
<code>__cplusplus</code>	其值為 199711L (ISO C++ 標準通過的日期)，假如檔案不是由 C++ 編譯器所編譯的，則其值為未定義的。可以用來設定檔案是由 C 或 C++ 編譯的。

圖 E.1 事先定義的符號常數

E.9 斷言

assert 巨集 (定義在 `<cassert>` 標頭檔) 會測試某個運算式的數值。如果運算式的值為 0 (偽)，則 `assert` 將印出一段錯誤訊息並且呼叫 `abort` 函式 (屬於一般的公用函式庫 - `<cstdlib>`) 來終止程式的執行。這是一個很有用的偵錯工具，可以用來測試某個變數是否為正確的數值。例如，假設在程式中，變數 `x` 的值應該不會大於 10。我們可以用一個斷言來測試 `x` 的值，如果 `x` 的值不正確，則印出錯誤訊息。敘述式為

E-10 C++程式設計藝術(第七版)(國際版)

```
assert( x <= 10 );
```

如果執行到該敘述式時， x 大於 10，則程式會印出一段含有此行號和檔案名稱的錯誤訊息，然後結束執行。然後，你可以查看其鄰近的程式碼，找出錯誤的位置。如果定義了符號常數 `NDEBUG`，則其後的斷言都會忽略。因此當我們不再需要斷言時（例如當偵錯完成時），你只需要將

```
#define NDEBUG
```

加到程式檔，而不必刪除每個斷言。`NDEBUG` 跟 `DEBUG` 符號常數一樣，通常是由編譯器命令列選項或是 IDE 所設定的。

大多數的 C++ 編譯器目前都提供例外處理的功能。C++ 程式設計師會使用例外處理來代替斷言。但是對於會使用到傳統 C 程式碼的 C++ 程式設計師來說，還是需要了解斷言。

E.10 總結

本附錄討論了 `#include` 命令，通常用在開發大型程式。你也學到了 `#define` 命令，可以用來產生巨集。我們也介紹了條件式編譯、顯示錯誤訊息以及斷言。

摘要

E.2 `#include` 前置處理器命令

- 在程式編譯之前，會先處理以`#`開頭的每一行前置處理器命令。
- 在同一行中，只有空白字元可以出現在前置處理器命令之前。
- `#include` 命令會含入所指定檔案的副本。如果檔案名稱以雙引號括起來，則前置處理器會到編譯檔案所在的目錄底下搜尋指定的含入檔。如果檔案名稱是以角括號 `<`和`>` 包含起來，則搜尋會按照實作環境定義的方式來執行。

E.3 `#define` 前置處理器命令：符號常數

- `#define` 前置處理器命令可以用來產生符號常數和巨集。
- 符號常數是某個常數的名稱。

E.4 #define 前置處理器命令：巨集

- 巨集 (macro) 是由 #define 前置處理器命令所定義的運算。巨集可以定義成具有引數或不具有引數。
- 巨集或符號常數的代換文字通常是在 #define 命令同一行中，位於識別字 (和巨集引數列) 之後的所有文字。如果巨集或符號常式的代換文字太長，以致於無法於一行內撰寫完成，則你必須在行末加上一個反斜線(\)，來表示下一行仍然是代換文字。
- 符號常數和巨集可以使用 #undef 前置處理器命令來移除。#undef 命令會取消某個符號常數或巨集名稱的定義。
- 符號常數或巨集的範圍 (scope) 是從它定義的位置開始，一直到以 #undef 取消定義，或直到檔案結束為止。

E.5 條件式編譯

- 條件式編譯 (conditional compilation) 讓你能夠控制前置處理器命令的執行，以及程式碼的編譯。
- 條件前置處理器命令會計算某個常數整數運算式。強制轉換運算式、sizeof 運算式、以及列舉常數都不能夠在前置處理器命令中進行計算。
- 每個 #if 結構都必須以 #endif 結束。
- 命令 #ifdef 和 #ifndef 是 #if defined (name)和 #if !defined (name)的縮寫。
- 多重條件前置處理器結構可以使用 #elif 和 #else 來進行測試。

E.6 #error 和 #pragma 前置處理器命令

- #error 命令會印出含有此命令所指定之 token 的實作環境錯誤訊息，然後終止前置處理和編譯。
- #pragma 命令會產生實作環境定義的動作。如果實作環境無法辨識某個 pragma，則這個 pragma 將會忽略。

E.7 # 和 ## 運算子

- #運算子會將代換文字 token 轉換成由雙引號所包圍的字串。#運算子必須使用於具有引數的巨集，因為#的運算元一定是巨集的某個引數。
- ##運算子可用來串接兩個 token。## 運算子必須有兩個運算元。

E.8 事先定義的符號常數

- 有六個事先定義的符號常數。常數 `__LINE__` 是目前原始程式碼的行數 (整數)。常數 `__FILE__` 是檔案的假設名稱 (字串)。常數 `__DATE__` 是原始程式檔編譯時的日期 (字串)。常數 `__TIME__` 是原始程式檔編譯時的時間 (字串)。事先定義的符號常數都是以兩個底線字元開始和結束 (`__cplusplus` 除外)。

E.9 斷言

- `assert` 巨集 (定義在 `<cassert>` 標頭檔) 會測試某個運算式的數值。如果運算式的值為 0 (偽)，則 `assert` 會印出一段錯誤訊息，並且呼叫函式 `abort` 來終止程式的執行。

術語

<code>#</code>	展開巨集 (expand a macro)
<code>##</code>	<code>__FILE__</code>
<code>\</code> (反斜線)連續字元 <code>[\ (backslash) continuation character]</code>	標頭檔 (header file)
<code>abort</code>	<code>#if</code>
引數 (argument)	<code>#ifdef</code>
<code>assert</code> 巨集 (assert macro)	<code>#ifndef</code>
<code><cassert></code>	<code>#include</code> 前置處理器命令 (#include Preprocessor Directive)
條件式編譯 (conditional compilation)	<code>__LINE__</code>
前置處理器命令的條件式執行 (conditional execution of preprocessor directives)	巨集 (macro)
轉換成字串前置處理器命令 <code>__cplusplus</code> (convert-to-string preprocessor directive <code>__cplusplus</code>)	巨集識別字 (macro identifier)
<code><cstdio></code>	具有引數的巨集 (macro with arguments)
<code><cstdlib></code>	<code>#pragma</code> 命令 (#pragma directive)
<code>__DATE__</code>	事先定義的符號常數 (predefined symbolic constants)
除錯器 (debugger)	前置處理命令 (preprocessing directive)
前置處理指令 <code>#define</code> (#define preprocessor directive)	前置處理器 (preprocessor)
命令 (directives)	代換文字 (replacement text)
<code>#elif</code>	符號常數或巨集的範圍 (scope of a symbolic constant or macro)
<code>#else</code>	標準函式庫標頭檔 (standard library header files)
<code>#endif</code>	符號常數 (symbolic constant)
<code>#error</code> 命令 (#error directive)	<code>__TIME__</code>
	<code>#undef</code> 前置處理器命令 (#undef Preprocessor Directive)

自我測驗

E.1 填寫以下空格：

- a) 每個前置處理器命令都必須以_____開始。
- b) 條件式編譯結構可以用_____和_____命令來擴充成多重狀況。
- c) _____命令可以用來產生巨集和符號常數。
- d) 在同一行中，只有_____字元可以出現在前置處理器命令之前。
- e) 命令_____可以用來捨棄符號常數和巨集名稱。
- f) _____和_____命令是`#if defined (name)` 和`#if !defined (name)` 的縮寫。
- g) _____讓你能夠控制前置處理器命令的執行，以及程式碼的編譯。
- h) 如果巨集計算運算式的值為 0，則_____巨集會印出一段訊息並且結束程式的執行。
- i) _____命令會將一個檔案加入另一個檔案。
- j) _____運算子將它的兩個引數串接。
- k) _____運算子會將它的運算元轉換成字串。
- l) _____字元表示符號常數或巨集的代換文字會在下一行中繼續定義。

E.2 寫一個程式，印出以下的事先定義的符號常數：__LINE__、__FILE__、__DATE__ (圖 E.1)。

E.3 為以下各項工作撰寫前置處理器命令。

- a) 將符號常數 YES 的值定義為 1。
- b) 將符號常數 NO 的值定義為 0。
- c) 含入標頭檔 common.h。這個檔案與即將編譯的檔案位於相同目錄中。
- d) 如果符號常數 TRUE 已定義，則取消它的定義，然後再將它重新定義為 1。請勿使用 `#ifdef`。
- e) 如果符號常數 TRUE 已定義，則取消它的定義，然後再將它重新定義為 1。請使用 `#ifdef` 前置處理器命令。
- f) 如果符號常數 ACTIVE 不為 0，請將符號常數 INACTIVE 定義為 0。否則，請將 INACTIVE 定義為 1。
- g) 定義 CUBE_VOLUME 巨集，其功用為計算一個立方體的體積 (接收一個引數)。

自我測驗解答

E.1 a) #。 b) `#elif`，`#else`。 c) `#define`。 d) 空白。 e) `#undef`。 f) `#ifdef`，`#ifndef`。 g) 條件式編譯。 h) `assert`。 i) `#include`。 j) `##`。 k) #。 l) \。

E-14 C++程式設計藝術(第七版)(國際版)

E.2 (參考以下程式)

```
1 // exF_02.cpp
2 // Self-Review Exercise E.2 solution.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     cout << "__LINE__ = " << __LINE__ << endl
9         << "__FILE__ = " << __FILE__ << endl
10        << "__DATE__ = " << __DATE__ << endl
11        << "__TIME__ = " << __TIME__ << endl
12        << "__cplusplus = " << __cplusplus << endl;
13 }
```

```
__LINE__ = 9
__FILE__ = c:\cpp4e\ch19\ex19_02.CPP
__DATE__ = Jul 17 2002
__TIME__ = 09:55:58
__cplusplus = 199711L
```

- E.3
- a) `#define YES 1`
 - b) `#define NO 0`
 - c) `#include "common.h"`
 - d) `#if defined(TRUE)`
 `#undef TRUE`
 `#define TRUE 1`
 `#endif`
 - e) `#ifdef TRUE`
 `#undef TRUE`
 `#define TRUE 1`
 `#endif`
 - f) `#if ACTIVE`
 `#define INACTIVE 0`
 `#else`
 `#define INACTIVE 1`
 `#endif`
 - g) `#define CUBE_VOLUME(x) ((x) * (x) * (x))`

習題

- E.4 撰寫一個程式，其中定義一個接收一個引數的巨集，計算一個球體的體積。你的程式應該計算半徑 1~10 的球體體積，並以表格列出結果。球體體積的計算公式如下：

$$\left(\frac{4.0}{3} \right) * \pi * r^3$$

其中 π 等於 3.14159。

E.5 撰寫一個程式產生以下的輸出：

```
The sum of x and y is 13
```

程式應該定義具有引數 *x* 和 *y* 的巨集 `SUM`，並且使用 `SUM` 來產生輸出。

E.6 撰寫一個定義和使用巨集 `MINIMUM2` 的程式，來找出兩個數中較小的數。請由鍵盤輸入數值。

E.7 撰寫一個使用巨集 `MINIMUM3` 的程式，來找出三個數中最小的數。`MINIMUM3` 巨集應使用習題 F.6 所定義的 `MINIMUM2` 巨集來找出最小的數。請由鍵盤輸入數值。

E.8 撰寫一個程式，使用 `PRINT` 巨集列印出字符串值。

E.9 撰寫一個使用巨集 `PRINTARRAY` 印出整數陣列的程式。此巨集的引數應該為陣列以及陣列的元素個數。

E.10 撰寫一個使用巨集 `SUMARRAY` 的程式，來計算數值陣列的數值總和。此巨集的引數應該為陣列以及陣列的元素個數。

E.11 修改 E.4-E.10 的答案，將成員函式行內化 (`inline`)。

E.12 請找出前置處理器在展開下列巨集時，可能會發生的錯誤：

- a) `#define SQR(x) x * x`
- b) `#define SQR(x) (x * x)`
- c) `#define SQR(x) (x) * (x)`
- d) `#define SQR(x) ((x) * (x))`

E-16 C++程式設計藝術(第七版)(國際版)