

第十章

C++

前置處理器

本章內容：

- * #define 敘述
- * 條件式編譯
- * # include 檔
- * 參數化的巨集
- * 高階功能
- * 摘要
- * 程式設計練習
- * 問題解答

演說家就像華麗刺繡的壁毯商人一樣，以誇張的言辭誇耀花色的繁美以求人們購買；
但在其鍍金的言辭背後，往往是被扭曲的事實。

— 俗諺

早期的 C 編譯器並沒有常數或 inline 函式。隨著 C 的快速發展，人們很快發現，C 需要處理常數、巨集及含括 (include) 檔。解決的方式是建立一個前置處理器 (preprocessor)，在將程式送到 C 編譯器之前先予以處理。前置處理器算是一種特殊的文字編輯器。它的語法和 C 完全不同，它也不需要了解 C 語言的架構。它只是一種原始的文字編輯器。

#define 敘述

#define 敘述可用來定義常數。例如，下面兩行功能類似：

```
#define SIZE 20           // 陣列大小是 20
const int SIZE = 20;     // 陣列大小是 20
```

實際上，#define SIZE 20 的動作，就是要求前置處理器將每個 SIZE 變成 20，這簡化了許多額外的處理程序。

前置處理器的前面都會加上“#”符號。C++ 是一種格式自由的語言，語言的元素可放在一行的任何位置，行的結束符號會視為空白。但前置處理器並不是自由格式，你必須將“#”符號放在每一行的前面。後面會談到，前置處理器並不了解 C++ 的語法。

警告

前置處理器不屬於 C++ 編譯器的一部份。它用的是完全不同的語法。將前置處理器當做 C++ 程式碼，會導致許多問題。

前置處理器是以行為單位，C++ 的敘述是用分號(;)來結束。但前置處理器並不是用分號來結束，加入分號的話，會造成奇怪的結果。前置處理器的指示可在行的尾端放入倒斜線(\)來連接下一行。一般來說，你可以定義一個取代的巨集，例如：

```
#define FOO bar
```

前置處理器會將程式中的“FOO”這個字用“bar”取代。在實務上，一般會用大寫英文字母當作巨集名稱。如此才容易分辨是一個變數(全部小寫)或是一個巨集(全部大寫)。

一般 define 敘述的用法如下：

```
#define Name Substitute-Text
```

Name 可以是任何合法的變數名稱，Substitute-Text 可以是任何內容，只要一行中可以放得下。Substitute-Text 可包含空白字元、運算子和其它字元。

也可以使用下面的定義：

```
#define FOR_ALL for (i = 0; i < ARRAY_SIZE; ++i)
```

用法如下：

```
/*
 * 清除陣列的內容
 */
FOR_ALL {
    data[i] = 0;
}
```

這種定義巨集的方式並不理想，反而使得程式的控制流程糾纏不清。以這個例子來說，如果程式設計師想明白迴圈的功能，他必須先找到程式開始的位置，看看 FOR_ALL 的定義。

若是濫用這種取代的做法來代替基本的程式架構，會造成反效果。例如，你可以定義：

```
#define BEGIN {
#define END }

. . .

    if (index == 0)
        BEGIN
            cout << "Starting\n";
        END
```

你根本不是在寫 C++ 程式，而是 C++ 與 PASCAL 的混合體！

前置處理器會導致難以預測的問題，因為它並沒有檢查 C++ 的語法。例如，範例 10-1 會在第 11 行發生錯誤：

範例 10-1. big/big.cc

```
1 #define BIG_NUMBER 10 ** 10
2
3 main()
```

範例 10-1. big/big.cc (續)

```
4 {
5     // index for our calculations
6     int    index;
7
8     index = 0;
9
10    // 下一行有語法錯誤
11    while (index < BIG_NUMBER) {
12        index = index * 8;
13    }
14    return (0);
15 }
```

問題是在第 1 行的 #define 敘述，但錯誤訊息會指出第 11 行的位置。第 1 行的定義會使前置處理器將第 11 行展開為：

```
while (index < 10 ** 10)
```

因為 ** 是不合法的運算子，所以會導致語法錯誤。

問題 10-1：下面程式的執行結果是 47，而不是預計的答案 144，怎麼搞的？

範例 10-2. first/first.cc

```
#include <iostream.h>

#define FIRST_PART    7
#define LAST_PART    5
#define ALL_PARTS    FIRST_PART + LAST_PART

main() {
    cout << "The square of all the parts is " <<
        ALL_PARTS * ALL_PARTS << '\n';
    return (0);
}
```

提示：

```
CC -E prog.cc
```

會將前置處理器的輸出結果送到螢幕上。

在 MS-DOS 下，可使用這個命令：

```
cpp prog.cpp
```

它會建立一個檔名為 prog.i 的檔案，包含前置處理器的輸出。

執行範例 10-2 之後，前置處理器的輸出為：

範例 10-3 : first/first-ed.out

```
# 1 "First.cc"
# 1 "/usr/local/lib/g++=include/iostream.h" 1 3

// 大約跳過 900 行的 #include 資料

inline ios& oct(ios& i)
{ i.setf(ios::oct, ios::dec|ios::hex|ios::oct); return i; }

# 1 "first.cc" 2

main() {
    cout << "The square of all the parts is " <<
        7 + 5 * 7 + 5 << '\n';
    return (0);
}
```

注意

C++ 前置處理器的輸出含有許多資訊，大部份可以忽略。在這？，你需要檢查輸出直到 cout 敘述這一行。這裡你會找到問題的原因。

問題 10-2：範例 10-2 會產生一個警告訊息，指出 counter 變數在使用前尚未設定其內容值。但是 for 迴圈明明有指定其值。你也會見到第 11 行有一個奇怪的警告訊息“null effect”。

範例 10-4.

```
1 // 警告! 空白是關鍵
2
3 #include <iostream.h>
4
5 #define MAX 10
6
7 main()
8 {
9     int counter;
10
11     for (counter =MAX; counter > 0;
12         --counter)
13         cout << "Hi there\n";
14
15     return (0);
16 }
```

提示：請檢查前置處理器的輸出。

問題 10-3：範例 10-3 算出的 size 內容值不對，為什麼？

範例 10-5. size/size.cc

```
#include <iostream.h>

#define SIZE    10;
#define FUDGE   SIZE -2;

main()
{
    int size;// 使用 size 變數

    size = FUDGE;
    cout << "Size is " << size << '\n';
    return (0);
}
```

問題 10-4：下面的程式在輸入不正確的資料時會印出訊息“Fatal Error: Abort”，並且結束執行。但是輸入正確的資料時也會如此，為什麼？

範例 10-6. die/die.cc

```
1 #include <iostream.h>
2 #include <stdlib.h> /* 只需要 ANSI 標準 */
3
4 #define DIE \
5   cerr << "Fatal Error:Abort\n";exit(8);
6
7 main() {
8   // 用來測試的任意值
9   int value;
10
11   value = 1;
12   if (value < 0)
13     DIE;
14
15   cerr << "We did not die\n";
16   return (0);
17 }
```

#define 與 const

在 `const` 之前，`#define` 是定義常數的唯一方式，因此，早期的程式碼都是使用 `#define`。然而，`const` 比 `#define` 更理想。首先，C++ 會立即檢查 `const` 敘述的語法；而 `#define` 指示一直到使用巨集時才會檢查。`const` 是採用 C++ 語法，`#define` 則有自己的語法；最後，`const` 適用 C++ 變數的有效範圍法則，而用 `#define` 所定義的常數會延伸到整個程式【譯註】。

在大多數情況下，`const` 比 `#define` 理想。底下有兩種個方式來定義同樣的常數：

```
#define MAX 10          // 使用前置處理器定義一個值
                       // 這很容易出問題

const int MAX = 10;    // 定義一個 C++ 的常數整數
                       // ( 比較安全 )
```

譯註 沒有區域和全域的差別

`#define` 只能定義簡單的常數。`const` 可定義幾乎每一種 C++ 常數的型態，包含結構類別。例如：

```
struct box {
    int width, height;           // 方塊的大小，單位為點
};
const box pink_box(1.0, 4.5);   // 輸入粉紅色方塊的大小
```

`#define` 在條件式編譯和其它特定應用上十分有用。

條件式編譯

程式設計師往往必須面對跨平台的問題，就是讓程式在各種機器上執行。理論上，C++ 的可攜性很高；但許多機器具有特殊的架構。例如，本書包含了 UNIX、MS-DOS 及 Windows 編譯器，表面上十分類似，但仍有差異存在。第二十五章將會探討這些問題。

前置處理器允許你透過條件式編譯 (conditional compilation)，賦予你設計上的彈性。假設你要在測試時將除錯用的程式碼放進來，然後在正式發行時予以刪除，就可以使用 `#ifdef-#endif` 的方式：

```
#ifdef DEBUG
    cout << "In compute_hash, value " << value << " hash " << hash << "\n";
#endif /* DEBUG */
```

注意

`#endif` 後面不必加上 `/* DEBUG */`，但這可以當作註解。

程式開頭如果包含：

```
#define DEBUG /* Turn debugging on */
```

就會包含 `cout` 這一行。若程式含有下面的指示：

```
#undef DEBUG /* Turn debugging off */
```

就會跳過 `cout`。

嚴格來說，`#undef DEBUG` 不是必要的。若是沒有 `#define DEBUG` 敘述，`DEBUG` 就是未定義的狀態。`#undef DEBUG` 可以明確指出：`DEBUG` 是用來進行條件式編譯，目前是屬於關閉狀態。

如果不定義該符號，`#ifndef` 指示會編譯這段程式碼。

`#else` 表示條件式的反向。例如：

```
#ifdef DEBUG
    cout << "Test version. Debugging is on\n";
#else /* DEBUG */
    cout << "Production version\n";
#endif /* DEBUG */
```

程式設計師往往需要暫時刪除部份的程式碼，常用的方法是用 `/* */` 括住，把部份程式暫時當成註解。這可能會出問題，例如下面的例子：

```
/**** 將這一小段當做註解
    section_report();
    /* 處理後面的部份 */
    dump_table();
**** 該段註解結束 */
```

第 5 行會出現語法上的錯誤，為什麼？

更好的方式是使用 `#ifdef` 將這段內容刪除。

```
#ifdef UNDEF
    section_report();
    /* 處理後面的部份 */
    dump_table();
#endif /* UNDEF */
```

(當然，若是有人定義了 `UNDEF`，這個程式碼就會被包含進來；這樣做肯定會被人毒打一頓。)

編譯器的設定選項 `-Dsymbol` 允許你定義符號，例如下面的命令：

```
CC -DDEBUG -g -o prog prog.cc
```

雖然程式中沒有 `#define DEBUG`，但是編譯 `prog.c` 程式會包含 `#ifdef DEBUG` 到 `#endif /* DEBUG */` 之間的程式。Turbo C++ 同樣功能的命令為：

```
tcc -DDEBUG -g -N -eprog.exe prog.c
```

一般形式的選項是 `-Dsymbol` 或 `-Dsymbol=value`。例如，下面會設定 `MAX` 為 10：

```
CC -DMAX=10 -o prog prog.c
```

大部份的 C++ 編譯程式會自動定義一些與系統相關的符號。像是 Turbo C++ 定義 `__TURBOC__`，MS-DOS 則定義 `__MSDOS__`。ANSI 標準的 C 編譯器定義了符號 `__STDC__`。C++ 編譯器定義了 `__cplusplus`。大多數的 UNIX 編譯器都定義了系統的名稱（如 Sun、VAX、celerity 等等），但很少有文件提到這一點。unix 這個符號，在所有的 UNIX 機器上都有定義。

注意

命令列的選項只會指定符號的初始值。程式中任何的 `#define` 及 `#undef` 指示，都可以改變符號的值。例如：

```
#undef DEBUG
```

不論你是否使用 `-DDEBUG`，都會使 `DEBUG` 變成未定義。

#include 檔

#include 指示允許程式使用另一個程式的原始碼。

例如，程式中使用了這個指示：

```
#include <iostream.h>
```

它會要求前置處理器取出 `iostream.h` 這個檔案，並且放入目前的程式中。可被其他檔案包含的檔案，稱為標頭檔 (header file)，大部份的 #include 指示出現在程式頂端。在 UNIX 中，這些檔案位於 `/usr/include`；在 Turbo C++ 中，則是位於 Turbo C++ 安裝的目錄。

標準包含檔 (include file) 是用來定義程式庫中函式的資料結構和巨集。例如，`cout` 是一個標準類別，可將資料印到標準輸出裝置。`cout` 所用 `ostream` 的類別定義和相關程序，都定義在 `iostream.h`。

你或許需要設計自己的包含檔。當你必須將程式拆成多個檔案時，或儲存常數和資料結構，本地 (local) 包含檔都是非常好用的。它可以讓同一組專案的設計師分享許多資訊 (請參考第二十三章)。

你可用雙引號括住檔案名稱，來指定含括檔：

```
#include "defs.h"
```

檔名 ("`defs.h`") 可為任何合法的檔案名稱，可以是一個簡單的檔案 "`defs.h`"；相對的路徑 "`../data.h`"；或絕對路徑 "`/root/include/const.h`"。(在 DOS/Windows 環境中，必須使用倒斜線 (`\`) 而不是斜線 (`/`) 來做為目錄的分隔符號。)

包含檔可為巢狀方式，但這可能很容易會造成問題。假設你在 `const.h` 檔案中定義多個常數，若是檔案 `data.h` 及 `io.h` 兩者都含括 `const.h`，然後你的程式中加入了：

```
#include "data.h"
#include "io.h"
```

由於前置處理器設定 `const.h` 中的定義兩次，所以會發生錯誤。定義常數兩次並非嚴重錯誤 (fatal error)；然而定義一個資料結構或是 `union` 兩次，則會導致嚴重錯誤，必須極力避免。

解決的方法之一是檢查 `const.h`，看看是否它已經被包含進來，並且沒有重覆定義的符號。

請看下面這段程式碼：

```
#ifndef _CONST_H_INCLUDED_

/* 定義常數 */

#define _CONST_H_INCLUDED_
#endif /* _CONST_H_INCLUDED_ */
```

當 `const.h` 被包含進來，它定義了 `_CONST_H_INCLUDED_`。若是這個符號已經被定義了，`#ifndef` 的條件式會隱藏其他的定義，以避免問題發生。

注意

你也可以將程式碼放在標頭檔中，但這不是好習慣。通常程式碼是放在 `.cc` 檔案裡面，定義、宣告、巨集及 `inline` 函式則放在 `.h` 檔案。

參數化的巨集

目前為止，討論的只有簡單的 `#define` 和巨集。巨集也可以接受參數，下面的巨集是計算一個數字的平方：

```
#define SQR(x) ((x) * (x)) // 數值的平方計算
```

巨集會用引數的字串來取代 `x`，`SQR(5)` 會展開成為 `((5)*(5))`。最好在巨集的參數旁邊加上括號，範例 10-7 說明了不用這個方式所產生的問題：

範例 10-7：`sqr/sqr.cc`

```
#include <iostream.h>
#define SQR(x) (x * x)

main()
{
```

```
int counter;    // 迴圈的計數器

for (counter = 0; counter < 5; ++counter) {
    cout << "x " << counter+1 <<
        " x squared " << SQR(counter+1) << '\n';
}
return (0);
}
```

問題 10-5：上面程式的輸出結果是什麼？（請在你的機器上測試）為什麼會是這個結果？（請檢查前置處理器的輸出）

在使用（++）和（--）運算子必須小心，避免混入太複雜的演算式，應考慮邊際效應以及難以預測的結果，如範例 10-8。

範例 10-8：sqr-i/sqr-i.cc

```
#include <iostream.h>
#define SQR(x) ((x) * (x))

main()
{
    int counter;    /* 迴圈的計數器 */

    counter = 0;
    while (counter < 5)
        cout << "x " << counter+1 <<
            " x squared " << SQR(++counter) << '\n';
    return (0);
}
```

為什麼它會造成無法預測的結果？這個計數器每次會增加多少？

在範例 10-8 中，SQR(++counter) 會展開為 ((++counter)*(++counter))。它使得計數器在每次進行一次迴圈時遞增 2 次。該算式實際產生的結果，是依據系統而定。

問題 10-6：下面的程式指出有一個未定義的變數，但我們明明只有一個變數 counter，怎麼回事？

範例 10-9 : rec/rec.cc

```
#include <iostream.h>
#define RECIPROCAL (number) (1.0 / (number))

main()
{
    float counter;

    for (counter = 0.0; counter < 10.0;
        counter += 1.0) {

        cout << "1/" << counter << " = " <<
            RECIPROCAL(counter) << "\n";
    }
    return (0);
}
```

運算子

運算子用在參數化的巨集中，可以將引數變成字串。例如：

```
#define STR(data) #data
STR(hello)
```

會產生：

```
"hello"
```

第二十六章有範例，進一步討論這個運算子。

參數化的巨集與 inline 函式

大部份的情形下，最好使用 inline 函式而不要用參數化的巨集。要避免參數化巨集所造成的問題，最好是透過 inline 函式。例如，SQR 巨集可同時處理 float 及 int 兩種資料型態，但你必須寫兩個 inline 函式，才能達到同樣的功能。

```
#define SQR(x) ((x) * (x)) // 參數化的巨集
// 可以處理，但是會有問題

// 相同作用的 inline 函式
inline int sqr(const int x) {
    return (x * x);
}
```

高階功能

本書沒有包含 C++ 前置處理器指示的完整列表。它包括 #if 某些進階的特性，可以處理條件式的編譯；以及 #pragma 指示，將特定編譯器的命令放進檔案裡。關於這些功能的詳細說明，請參考 C++ 的技術文件。

摘要

前置處理器對於 C++ 十分重要，它看起來和 C++ 幾乎完全不同。它並不屬於 C++ 編譯器的一部分。

在定義巨集時所導致的問題，在定義當時並不容易看出來，而是在執行時才出錯。透過下面幾種簡單的法則，可以降低問題產生的可能性：

1. 在每個地方加上括號。特別是在 #define 常數及巨集的參數時。
2. 定義巨集時若超過一個敘述，將程式碼加上 {}。
3. 前置處理器並不是 C++，不要使用 = 或是 ;。

```
#define X = 5           // 錯誤
#define X 5;           // 錯誤
#define X = 5;         // 大錯特錯

#define X 5            // 正確
```

如果你已經看到這了，表示最困難的部份已經過了。

程式設計練習

練習 10-1：C++ 的標準包含一個 `boolean` 型態 (`bool`) 它定義了 `true` 和 `false` 值。問題是大部分的 C++ 編譯器並沒有支援這種型態。請你利用 `#define`，來定義 `BOOLEAN`、`TRUE` 及 `FALSE` 的值。

練習 10-2：寫一個巨集來判斷，若是參數可被 10 整除則傳回 `TRUE`，否則傳回 `FALSE`。

練習 10-3：寫一個巨集 `is_digit`，若引數是十進位數則傳回 `TRUE`。寫另一個巨集 `is_hex`，若是引數是十六進位的數字 (`0-9 A-F a-f`)，則傳回 `TRUE`。第二個巨集應該參考第一個。

練習 10-4：寫一個前置處理器的巨集，可交換兩個整數。（如果你是屬於駭客級，就不要透過宣告在巨集外部的暫時變數。）

問題解答

答案 10-1：當程式被前置處理器處理過之後，`cout` 敘述展開如下：

```
cout << "The square of all the parts is " << 7 + 5 * 7 + 5 << '\n';
```

算式 `7 + 5 * 7 + 5` 等於 47。將巨集中的算式加入括號是個好習慣。若是改變 `ALL_PARTS` 的定義為：

```
#define ALL_PARTS (FIRST_PART + LAST_PART)
```

程式就可以正確執行。

答案 10-2：前置處理器的處理邏輯十分單純。在定義巨集時，會把識別字當做是巨集的一部份。在這個情形下，`MAX` 的定義就是 "`=10`"。當 `for` 敘述展開的結果如下：

```
for (counter == 10; counter > 0; --counter)
```

C++ 允許你算出一個結果，而且不去處理。在這個敘述中，程式只檢查 `counter` 是否為 10，而不去管它。在定義中刪除 `=` 之後，問題就解決了。

答案 10-3：前一個問題中，前置處理器並未依循 C++ 的語法。程式設計師使用分號來結束該敘述，但前置處理器會把它當成 `size` 定義的一部份。`size` 的指定敘述會展開為：

```
size = 10; -2;;
```

後面的這兩個分號並不會造成任何影響，中間那一個才是問題，這一行要求 C++ 做兩件事：

(1) 將 `size` 變數設定為 10，以及 (2) 計算 `-2` 的值並且不去管它（不會有警告訊息發生）。將分號刪除之後，就可以解決問題。

答案 10-4：前置處理器的輸出如下：

```
void exit();

main() {
    int value;

    value = 1;
    if (value < 0)
        "Fatal Error:Abort\n";exit(8);

    cerr << "We did not die\n";
    return (0);
}
```

問題是在 `if` 之後的兩個敘述，通常會分成兩行。若是將程式做正確的縮排，其結果如下：

範例 10-10：die3/die.cc

```
#include <iostream.h>
#include <stdlib.h>

main() {
    int value; // 用來測試的任意值

    value = 1;
    if (value < 0)
```

範例 10-10 : die3/die.cc (續)

```
        cout << "Fatal Error:Abort\n";

        exit(8);

        cout << "We did not die\n";
        return (0);
    }
```

如此一來，就很容易找出程式跳離的原因。實際上，if 後面的兩個敘述被前置處理器巨集藏起來了。解決這類問題的方法，是在含有多個敘述的巨集中加入大括號。

```
#define DIE \
    {cout << "Fatal Error: About\n"; exit(8);}
```

答案 10-5：問題是前置處理器根本不懂 C++ 的語法。每當迴圈處理一次，計數器會增加兩次。巨集呼叫：

```
SQR(counter+1)
```

會展開為：

```
(counter+1 * counter+1)
```

答案和 $(counter+1)*(counter+1)$ 不一樣。要避免這種問題，可採用 inline 函式，而不要使用參數化的巨集。

```
inline int SQR(int x) {return (x*x); }
```

如果你堅持要用參數化的巨集，記得將每一個參數加上括號：

```
#define SQR(x)    ((x) * (x))
```

答案 10-6：參數化的巨集與沒有參數的巨集的差異，在於括號會接在巨集名稱之後。在這？，RECIPROCAL 定義之後有一個空白字元，所以它不是一個參數化的巨集，而是一個單純的文字設定巨集，將 RECIPROCAL 用下面的文字取代：

```
(number) (1.0 / number)
```

將 RECIPROCAL 和 (number) 之間的空白刪除，就可以修正這個問題。