第6章 巨集處理器

巨集處理器 (Macro Processor) 乃是一種方便程式撰寫者使用，避免重複撰寫程式的工具，在程式被編譯前，巨集處理器會先將程式當中的巨集展開，然後再交給編譯器或組譯器進一步處理。

在組合語言當中，非常仰賴巨集的使用，因為組合語言沒有強大的編譯功能可以提供高層次的語法，巨集就成了組合語言當中唯一可以採用的較高層次工具。

在本章中，我們將介紹組合語言中巨集的用途與作法，並且介紹巨集處理的演算法。最後，在實務案例中，我們將焦點轉向高階語言，介紹 C 語言當中的巨集，並利用 gcc 觀察巨集展開的過程。

6.1 組合語言的巨集

為了說明巨集在組合語言當中的功能，先讓我們來看一個具有巨集的組合語言範例，我們以 max(x,y) 這樣一個函數為例，說明巨集的用法。範例 6.1 (a) 顯示了一個具有巨集的 CPU0 組合語言，其中的 MAX 是巨集，該巨集被定義後被呼叫了兩次。範例 6.1 (b) 是該程式被巨集處理器展開後的結果。

範例 6.1 具有巨集的組合語言 - 展開前與展開後的狀況

	(a). 巨集展開前	(b) 巨集展開後
1	MAX MACRO &X, &Y, &Z	// MAX(A,B,C)
2	LD R1, &X	LD R1, A
3	LD R2, &Y	LD R2, B
4	CMP R1, R2	CMP R1, R2
5	JLE $ELSE	JLE $ELSE1
6	ST R1, &Z	ST R1, C
7	JMP $END	JMP $END2
8	$ELSE:	$ELSE1:
9	ST R2, &Z	ST R2, C
10	$END:	$END2:
11	MEND	// MAX(C,D,E)
12		LD R1, C
13	MAX(A, B, C)	LD R2, D

14	MAX(C, D, E)	CMP R1, R2
15	RET	JLE $ELSE3
16		ST R1, E
17	A: WORD 5	JMP $END4
18	B: WORD 3	$ELSE3:
19	C: RESW 1	ST R2, E
20	D: WORD 7	$END4:
21	E: RESW 1	RET
22		
23		A: WORD 5
24		B: WORD 3
25		C: RESW 1
26		D: WORD 7
27		E: RESW 1

在範例 6.1 當中，左半部 (a) 是包含巨集的原始程式，而右半部 (b) 則是經過巨集處理器展開後的狀況。讀者可以很清楚的看到，在第 1 行當中，我們用 MACRO 指令定義了一個名稱為 MAX 的巨集，而且在第 11 行中，我們用 MEND 指令結束巨集定義。接著，在 13 行當中我們以 MAX(A,B,C) 的方式呼叫了該巨集一次，接著又在 14 行當中以 MAX(C,D,E) 第二次呼叫該巨集。於是，導致了該巨集在程式當中被呼叫了兩次。

因此，在範例 6.1 右半部的 (b) 當中，MAX 巨集被展開了兩次，第一次是在的第 1 行，第二次是在第 11 行。在第一次巨集展開的程式內 (1-10 行)，參數 &X 被取代為 A, &Y 被取代為 B, 而 &Z 被取代為 C。同樣的，在第二次巨集展開的程式碼內 (11-20 行)，參數 &X, &Y, &Z 則分別被取代為 C, D, E。

除了參數的取代之外，為了避免同一巨集多次展開所造成的標記重複現象，因此，巨集處理器會將標記加上編號，以避免重複的狀況。

在巨集當中，標記的前面必須被加上錢字號 $，舉例而言，像 $ELSE 與 $END 等兩個標記就被加上了錢字號，這是用來提醒巨集處理器的一種方法。這兩個標記展開後變成 $ELSE1, $END2, $ELSE3, $END4，如此就可以避免到標記重複的現象。

6.2 巨集處理的演算法

單層的巨集處理器只能容許一層巨集呼叫，不能容許在巨集當中再度呼叫巨集。這種巨集處理器的設計非常簡單，只要針對每一個巨集指令進行單層展開即可。其演算法如圖 6.1 所示。

演算法	說明
Algorithm MacroProcessor	單層巨集處理器
Input sourceFile, expandFile	輸入原始程式、輸出展開檔
inFile = open(sourcFile)	開啟輸入檔 (原始程式)
outFile = create(expandFile)	建立輸出檔 (展開程式)
pass1()	第一輪：定義巨集
pass2()	第二輪：展開巨集
End Algorithm	
Function pass1	第一輪：定義巨集
while (not inFile.isEnd)	當輸入檔未結束時
line = getLine()	讀取一行
if isMacroDefine(line)	如果是巨集定義
macro = readMacro();	讀取整個巨集
macroTable[macro.name] =macro	記錄到巨集表當中
end while	
End Function	
Function pass2	第二輪：展開巨集
inFile.goTop();	回到輸入檔開頭
while (not inFile.isEnd)	當輸入檔未結束時
line = getLine()	讀取一行
op = opCode(line)	取得指令部分
search macroTable for op	看看是否為巨集呼叫
if found	如果是巨集呼叫
macroCall = parseMacroCall(line);	剖析巨集呼叫
macro = macroTable[macroName];	取得巨集內容
body = replace macroCall.args in macro.body	取代內容中的參數
replace label with label+id in body	為標記加上編號
write body to outFile	將取代後的內容輸出
else	如果不是巨集呼叫

write source line to outFile end while End Function	將該指令直接輸出

圖 6.1 單層巨集處理器的演算法

圖 6.1 的演算法使用到兩種記錄結構，巨集記錄 macro 與呼叫結構 macroCall，並且使用到一個符號表 macroTable。利用這些資料結構，巨集處理器先在第一輪的讀取過程中建立巨集定義表，然後在第二輪的展開過程當中，展開每一行巨集呼叫，輸出展開後的結果。

大部分的商業用巨集處理器會支援多層的巨集展開功能，這種多層的展開方式雖然較為複雜，但是其基本動作與圖 6.1 相當類似，只是必須對每一行以遞迴的方式進行展開呼叫，如果在巨集展開的過程當中又發現巨集呼叫的指令時，就必須在呼叫巨集展開函數，以達成遞迴展開的功能。

6.3 實務案例

在本節中，我們將透過 gcc 的巨集展開功能，觀察 C 語言巨集的展開過程，以便理解巨集展開的實務操作方式。

6.3.1 C 語言的巨集

在 C 語言的設計中，有兩種巨集宣告方式，您可以使用 #define 指令宣告巨集函數，也可以利用 inline 指令，讓一般函數改為巨集函數，直接展開到程式當中。通常，#define 指令是用來撰寫較短的巨集定義，而 inline 指令則用來撰寫較長的巨集函數。

範例 6.2 中顯示了一個 C 語言的巨集定義與呼叫程式，其中定義了兩個巨集函數 max(a,b) 與 min(a,b)，分別傳回 a,b 兩者中的較大值與較小值。範例 6.2 左半部的 (a) 部分是原始程式，而右半部的 (b) 部分是展開後的結果，請讀者對照查看，以便理解 C 語言中的巨集運作方式。

我們可以使用 GNU 工具中的 gcc 編譯器，加上 -E 參數，以便將程式中的巨集展開，但不執行編譯動作。範例 6.2 就是我們用指令 gcc -E macro.c -o macro_E.c 將 (a) 展開為 (b) 的結果，讀者可以親自操作看看。

範例 6.2 具有巨集的 C 語言 - 展開前與展開後的狀況

指令：gcc -E macro.c -o macro_E.c	
(a) 展開前：macro.c	(b) 展開後：macro_E.c

#define max(a,b) (a>b?a:b)
#define min(a,b) (a<b?a:b)
 	
int main() {
  int x = max(3,5);
  int y = min(3,5);
  printf("max(3,5)=%d,
  min(3,5)=%d\n",x,y);
}
	int main() {
	  int x = (3>5?3:5);
	  int y = (3<5?3:5);
	  printf("max(3,5)=%d,
	  min(3,5)=%d\n",x,y);
	}

條件式展開

C 語言當中的巨集處理器，支援條件式展開的功能，這種功能對 C 語言相當重要，尤其在專案管理上更是不可或缺，以下，我們將說明 C 語言中條件式展開的巨集處理器之用途。首先，請讀者看範例 6.3 (a) 的程式，該程式用 #ifdef 條件式巨集指令，定義了 bugs 變數與 error(msg) 巨集函數，並且在程式的最後利用 error() 函數印出錯誤訊息，然後報告總共有幾個錯誤。

接著，請讀者利用 gcc 加上 -E 參數的方式，編譯該程式，指令如下所示。

```
gcc -E macroDebug.c -o MacroDebug_E.c
```

該指令會將程式的巨集展開，但因為編譯時沒有定義 _DEBUG_ 這個符號，因此，展開後的程式不會印出除錯訊息，其原始碼如範例 6.3 (b) 所示。

範例 6.3 具有條件式巨集的 C 語言 - 展開前與展開後的狀況

(a) 檔案：MacroDebug.c	(b) 檔案：MacroDebug_E.c

#ifdef _DEBUG_
  int bugs = 0;
  #define error(msg) {printf(msg);bugs++;}
#endif
 	
#define max(a,b) (a>b?a:b)
#define min(a,b) (a<b?a:b)
	int main() {
	  int x = (3>5?3:5);
	  int y = (3<5?3:5);
	  printf("max(3,5)=%d\n",x);
	  printf("min(3,5)=%d\n",y);
	}

	(c) 檔案：MacroDebug_DEBUG_E.c

```
int main() {
   int x = max(3,5);
   int y = min(3,5);
   printf("max(3,5)=%d\n",x);
   printf("min(3,5)=%d\n",y);
#ifdef _DEBUG_
   if (x!=5)
      error("max(3,5)");
   if (y!=3)
      error("min(3,5)");
   printf("共有%d 個錯誤",bugs);
#endif
}
```

```
int bugs = 0;

int main() {
   int x = (3>5?3:5);
   int y = (3<5?3:5);
   printf("max(3,5)=%d\n",x);
   printf("min(3,5)=%d\n",y);

   if (x!=5)
      {printf("max(3,5)");bugs++;};
   if (y!=3)
      {printf("min(3,5)");bugs++;};
   printf("共有 %d 個錯誤", bugs);

}
```

接著，再請讀者利用下列指令編譯該程式，其中的 –D_DEBUG_ 參數會動態的定義一個 _DEBUG_ 巨集符號傳給 gcc 編譯器。

```
gcc -E -D_DEBUG_ macroDebug.c -o MacroDebug_DEBUG_E.c
```

此時，由於 _DEBUG_ 符號已被定義，因此，展開後的程式會定義 bugs 變數, error(msg) 函數，並印出除錯訊息。其原始碼如範例 6.3 (c) 所示。

這種條件式的巨集定義的方式，在 C 語言當中相當常見，利用這種方式，我們可以編譯出程式碼較大的除錯版本，以便於除錯時使用。然後，在程式要發行時，編譯一個沒有 _DEBUG_ 符號的版本，如此，發行的軟體程式會較為精簡，也不會動不動就跑出錯誤訊息來困擾使用者。這也是使用 C 語言開發軟體時必須具備的巨集知識。

習題

6.1 請說明巨集處理器的輸入、輸出與功能為何？
6.2 請說明巨集處理器會如何處理巨集參數？
6.3 請說明巨集處理器在展開標記時會產生甚麼問題，應如何解決？
6.4 請使用 gcc 工具將範例 6.2 展開，觀察展開後的檔案，並說明展開前後的對應關係。

6.5 請使用 gcc 工具將範例 6.3 展開，觀察展開後的檔案，並說明展開前後的對應關係。