

附錄A CPU0 處理器

A.1 處理器

CPU0 是一個簡易的 32 位元處理器，其架構如圖 A.1 所示，包含 R0..R15, IR, MAR, MDR 等暫存器，其中 IR 是指令暫存器，R0 是一個永遠為常數 0 的唯讀暫存器，R15 是程式計數器 (Program Counter : PC)，R14 是連結暫存器 (Link Register : LR)，R13 是堆疊指標暫存器 (Stack Pointer : SP)，而 R12 是狀態暫存器 (Status Word : SW)。

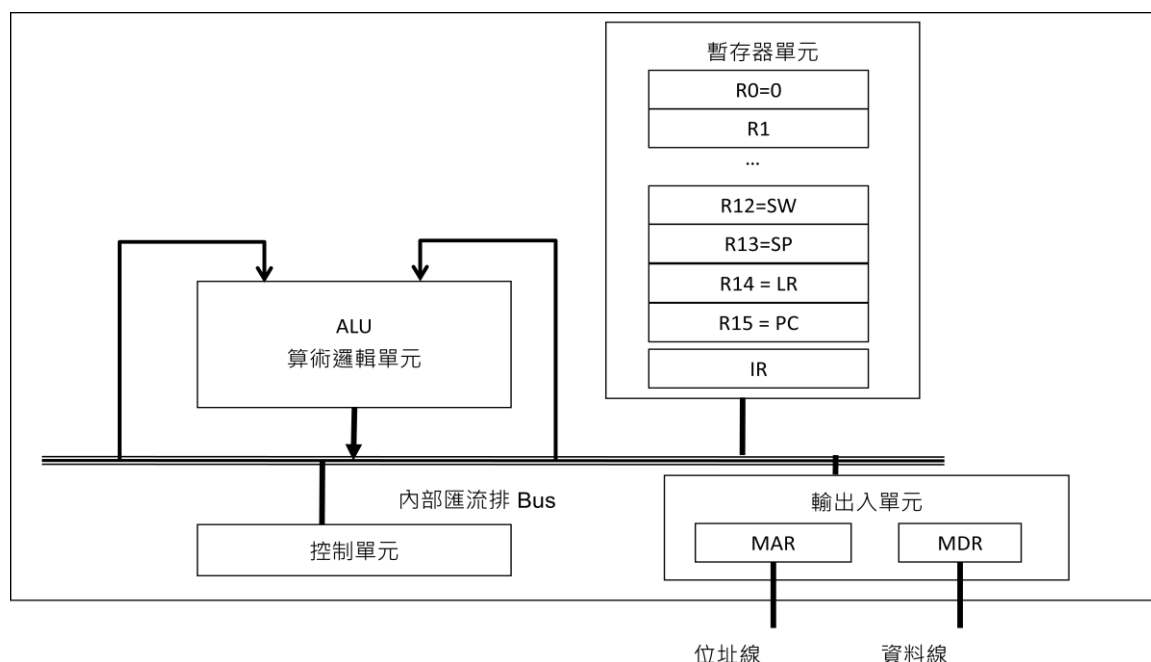


圖 A.1 CPU0 的架構圖

A.2 指令表

CPU0 包含『載入儲存』、『運算指令』、『跳躍指令』、『堆疊指令』等四大類指令，表格 A.1 是 CPU0 的指令編碼表，記載了 CPU0 的指令集與每個指令的編碼。

表格 A.1 CPU0 的指令編碼表

類型	格式	指令	OP	說明	語法	語意
載入儲存	L	LD ¹	00	載入 word	LD Ra, [Rb+Cx]	Ra ← [Rb+ Cx]
	L	ST	01	儲存 word	ST Ra, [Rb+ Cx]	Ra → [Rb+ Cx]
	L	LDB	02	載入 byte	LDB Ra, [Rb+ Cx]	Ra ← (byte)[Rb+ Cx]
	L	STB	03	儲存 byte	STB Ra, [Rb+ Cx]	Ra → (byte)[Rb+ Cx]
	A	LDR	04	LD 的暫存器版	LDR Ra, [Rb+Rc]	Ra → [Rb+ Rc]
	A	STR	05	ST 的暫存器版	STR Ra, [Rb+Rc]	Ra → [Rb+ Rc]
	A	LBR	06	LDB 的暫存器版	LBR Ra, [Rb+Rc]	Ra ← (byte)[Rb+ Rc]
	A	SBR	07	STB 的暫存器版	SBR Ra, [Rb+Rc]	Ra → (byte)[Rb+ Rc]
	L	LDI	08	立即載入	LDI Ra, Rb+Cx	Ra ← Rb + Cx
運算指令	A	CMP ²	10	比較	CMP Ra, Rb	SW ← Ra >= < Rb
	A	MOV	12	移動	MOV Ra, Rb	Ra ← Rb
	A	ADD	13	加法	ADD Ra, Rb, Rc	Ra ← Rb+Rc
	A	SUB	14	減法	SUB Ra, Rb, Rc	Ra ← Rb-Rc
	A	MUL	15	乘法	MUL Ra, Rb, Rc	Ra ← Rb*Rc
	A	DIV	16	除法	DIV Ra, Rb, Rc	Ra ← Rb/Rc
	A	AND	18	邏輯 AND	AND Ra, Rb, Rc	Ra ← Rb and Rc
	A	OR	19	邏輯 OR	OR Ra, Rb, Rc	Ra ← Rb or Rc
	A	XOR	1A	邏輯 XOR	XOR Ra, Rb, Rc	Ra ← Rb xor Rc
	A	ROL ³	1C	向左旋轉	ROL Ra, Rb, Cx	Ra ← Rb rol Cx
	A	ROR	1D	向右旋轉	ROR Ra, Rb, Cx	Ra ← Rb ror Cx
	A	SHL	1E	向左移位	SHL Ra, Rb, Cx	Ra ← Rb << Cx
	A	SHR	1F	向右移位	SHR Ra, Rb, Cx	Ra ← Rb >> Cx
跳躍指令	J	JEQ ⁴	20	跳躍 (相等)	JEQ Cx	if SW(=) PC ← PC+Cx
	J	JNE	21	跳躍 (不相等)	JNE Cx	if SW(!=) PC ← PC+Cx
	J	JLT	22	跳躍 (<)	JLT Cx	if SW(<) PC ← PC+Cx
	J	JGT	23	跳躍 (>)	JGT Cx	If SW(>) PC ← PC+Cx
	J	JLE	24	跳躍 (<=)	JLE Cx	if SW(<=) PC ← PC+Cx
	J	JGE	25	跳躍 (>=)	JGE Cx	If SW(>=) PC ← PC+Cx

¹ LD 為 LOAD 的縮寫，ST 代表 STORE，因此 LDB 就是 Load byte，STB 就是 Store byte，而 LDI 則是 Load immediate value，LDR 則是 Load by Register。

² CMP 為 Compare 的縮寫，MOV 代表 Move，SUB 代表 Subtract，MUL 代表 Multiply，DIV 代表 Divide。

³ ROL 為 Rotate Left 的縮寫，ROR 的全名為 Rotate Right，SHL 代表 Shift Left，SHR 則是 Shift Right。

⁴ JEQ 為 Jump if Equal 的縮寫，JNE 的全名為 Jump if Not Equal，JLT 則是 Jump if Less Than，JGT 則是 Jump if Greater Than，JLE 則是 Jump if Less or Equal，JGE 代表 Jump if Greater or Equal，JMP 則代表 Jump。

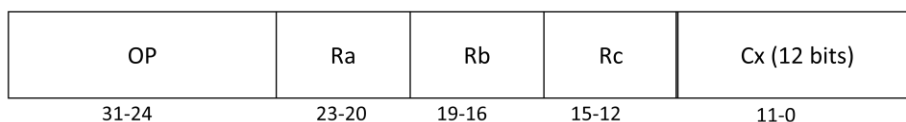
	J	JMP	26	跳躍 (無條件)	JMP Cx	$PC \leftarrow PC + Cx$
	J	SWI ⁵	2A	軟體中斷	SWI Cx	$LR \leftarrow PC; PC \leftarrow Cx; INT \leftarrow 1$
	J	CALL	2B	跳到副程式	CALL Cx	$LR \leftarrow PC; PC \leftarrow PC + Cx$
	J	RET	2C	返回	RET	$PC \leftarrow LR$
	J	IRET	2D	中斷返回	IRET	$PC \leftarrow LR; INT \leftarrow 0$
堆疊指令	A	PUSH	30	推入 word	PUSH Ra	$SP -= 4; [SP] = Ra;$
	A	POP	31	彈出 word	POP Ra	$Ra = [SP]; SP += 4;$
	A	PUSHB	32	推入 byte	PUSHB Ra	$SP --; [SP] = Ra; (byte)$
	A	POPB	33	彈出 byte	POPB Ra	$Ra = [SP]; SP ++; (byte)$

A.3 指令格式

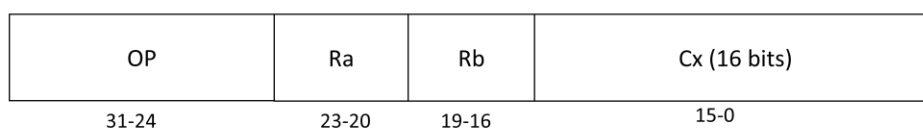
CPU0 所有指令長度均為 32 位元，這些指令也可根據編碼方式分成三種不同的格式，分別是 A 型、J 型與 L 型。

大部分的運算指令屬於 A (Arithmetic) 型，而載入儲存指令通常屬於 L (Load & Store) 型，跳躍指令則通常屬於 J (Jump) 型，這三種型態的指令格式如圖 A.2 所示。

A 型



L 型



J 型

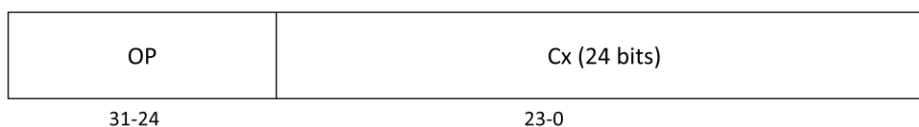


圖 A.2 CPU0 的指令格式

⁵ SWI 為 SoftWare Interrupt 的縮寫，CALL 則是 Call Subroutine，RET 則是 return 的縮寫。

A.4 狀態暫存器

R12 狀態暫存器 (Status Word : SW) 是用來儲存 CPU 的狀態值，這些狀態是許多旗標的組合。例如，零旗標 (Zero，簡寫為 Z) 代表比較的結果為 0，負旗標 (Negative，簡寫為 N) 代表比較的結果為負值，另外常見的旗標還有進位旗標 (Carry，簡寫為 C)，溢位旗標 (Overflow，簡寫為 V) 等等。圖 A.3 顯示了 CPU0 的狀態暫存器格式，最前面的四個位元 N、Z、C、V 所代表的，正是上述的幾個旗標值。



圖 A.3 CPU0 中狀態暫存器 SW 的結構

條件旗標的 N、Z 旗標值可以用來代表比較結果是大於 (>)、等於 (=) 還是小於 (<)，當執行 CMP Ra, Rb 動作後，會有下列三種可能的情形。

1. 若 $Ra > Rb$ ，則 $N=0, Z=0$ 。
2. 若 $Ra < Rb$ ，則 $N=1, Z=0$ 。
3. 若 $Ra = Rb$ ，則 $N=0, Z=1$ 。

如此，用來進行條件跳躍的 JGT、JGE、JLT、JLE、JEQ、JNE 指令，就可以根據 SW 暫存器當中的 N、Z 等旗標決定是否進行跳躍。

SW 中還包含中斷控制旗標 I (Interrupt) 與 T (Trap)，用以控制中斷的啟動與禁止等行為，假如將 I 旗標設定為 0，則 CPU0 將禁止所有種類的中斷，也就是對任何中斷都不會起反應。但如果只是將 T 旗標設定為 0，則只會禁止軟體中斷指令 SWI (Software Interrupt)，不會禁止由硬體觸發的中斷。

SW 中還儲存有『處理器模式』的欄位，M=0 時為『使用者模式』 (user mode) 與 M=1 時為『特權模式』 (super mode)等，這在作業系統的設計上經常被用來製作安全保護功能。在使用者模式當中，任何設定狀態暫存器 R12 的動作都會被視為是非法的，這是為了進行保護功能的緣故。但是在特權模式中，允許進行任何動作，包含設定中斷旗標與處理器模式等位元，通常作業系統會使用特權模式 (M=1)，而一般程式只能處於使用者模式 (M=0)。

A.5 位元組順序

CPU0 採用大者優先 (Big Endian) 的位元組順序 (Byte Ordering)，因此代表值越大的位元組會在記憶體的前面 (低位址處)，代表值小者會在高位址處。

由於 CPU0 是 32 位元的電腦，因此，一個字組 (Word) 占用 4 個位元組 (Byte)，因此，像 LD R1, [100] 這樣的指令，其實是將記憶體 100-103 中的字組取出，存入到暫存器 R1 當中。

LDB 與 STB 等指令，其中的 B 是指 Byte，因此，LDB R1, [100] 會將記憶體 100 中的 byte 取出，載入到 R1 當中。但是，由於 R1 的大小是 32 bits，相當於 4 個 byte，此時，LDB 與 STB 指令到底是存取四個 byte 當中的哪一個 byte 呢？這個問題的答案是 byte 3，也就是最後的一個 byte。

高位元

低位元

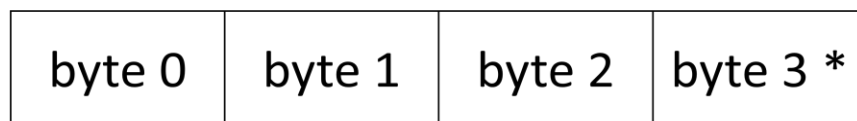


圖 A.4 CPU0 的暫存器位元組順序圖

A.6 中斷程序

CPU0 的中斷為不可重入式中斷，其中斷分為軟體中斷 SWI (Trap) 與硬體中斷 HWI (Interrupt) 兩類。

硬體中斷發生時，中段代號 INT_ADDR 會從中段線路傳入，此時執行下列動作：

1. $LR \leftarrow PC; INT \leftarrow 1$
2. $PC \leftarrow INT_ADDR$

軟體中斷 SWI Cx 發生時，會執行下列動作：

1. $LR \leftarrow PC; INT \leftarrow 1$
2. $PC \leftarrow Cx;$

中斷最後可以使用 IRET 返回，返回前會設定允許中斷狀態。

1. $PC \leftarrow LR; INT \leftarrow 0$

附錄B C0 語言的語法

為了說明編譯器與剖析器的設計原理，我們設計了一個簡化版的 C 語言，稱為 C0 語言。該語言總共包含 11 條規則，可以用來撰寫一些小型的程式。

B.1 C0 語言的範例

在 C0 語言當中，包含了指定、運算與多層的 for 迴圈等語句，範例 B.1 顯示了一個具有兩層 for 迴圈的 C0 語言程式。

範例 B.1 一個具多層結構 C0 語言程式

C0 語言程式
<pre>sum = 0; for (i=1; i<=9; i++) { for (j=1; j<=9; j++) { p = i * j; sum = sum + p; } } return sum;</pre>

B.2 EBNF 語法規則

在本書中，我們使用 EBNF 語法描述 C0 語言，該語法圖 B.1 所示，其中包含了 11 條規則，規則中的星號 * 代表重複比對數次 (包含零次)，加號 + 代表重複比對一次以上 (包含一次)，而問號 ? 則代表可出現零次或一次。這些符號可作用在圓括號 () 所框起來的規則區塊，或者由方括號 [] 所框起來的字元集合當中，用以代表這些區域可重複比對的次數。

	EBNF 語法規則
1	PROG = BaseList
2	BaseList = (BASE)*

3	BASE	= FOR STMT ';' ;
4	FOR	= 'for' '(' STMT ';' COND ';' STMT ')' BLOCK
5	STMT	= 'return' id id '=' EXP id ('++' '--')
6	BLOCK	= '{' BaseList '}'
7	EXP	= ITEM ([+-*/] ITEM)?
8	COND	= EXP ('==' '!=' '<=' '>=' '<' '>') EXP
9	ITEM	= id number
10	id	= [A-Za-z_][A-Za-z0-9_]*
11	number	= [0-9]+

圖 B.1 C0 語言的 EBNF 規則

FOR 規則中有三個重要的部分，也就是 STMT、COND 與 BLOCK 等三者，STMT 用來描述 `i=0; i++` 等敘述，而 COND 則描述條件判斷部分，像是 `i<=10` 等，而最後的 BLOCK 則是 for 迴圈的主體部分，BLOCK 乃是由一對大括號 `{...}` 夾住的 BaseList 區段所組成，於是透過 BaseList 又遞迴的定義了下一層的完整程式區段。

必須注意的是，C0 語言當中缺乏某些重要的結構，像是 IF 語句，函數呼叫等等，C0 語言甚至沒有支援完整的數學運算語法，因此每條運算式只能有一個加減乘除符號，所以在 C0 語言當中無法撰寫像 `sum=sum+i*j` 這樣的複雜算式，您必須自行將該語句拆成兩條指定敘述，採用像 `p = i*j; sum=sum+p;` 這樣的寫法。

您可以自行擴充 C0 語言，以支援 IF 及函數呼叫等語句，或者支援更完整的數學運算式語法，在本書中，為了簡單起見，我們將只用這 11 條規則，以避免實作上太過複雜。

附錄C GNU 開發工具

在本書中，我們以 GNU 工具作為主要的系統軟體工具，在 UNIX/Linux 當中，預設就包含 GNU 工具，但在 MS. Windows 當中，則可以安裝 Dev C++ (附錄 D) 或 Cygwin 環境 (附錄 E)，以便使用 GNU 工具。

在 GNU 工具中有一些常用的檔案命名方式，舉例而言，函式庫的附檔名通常是 *.a，而組合語言的附檔名通常是 *.s，以下是 GNU 平台中常見的一些附檔名使用慣例，如表格 C.1 所示。

表格 C.1 GNU/Linux/UNIX 的檔案命名慣例

附檔名	檔案類型說明
.c	C 語言的程式，像是 sum.c 等。
.a	函式庫，像是 libc.a、libm.a 等。
.cpp	C++ 的程式，像是 sum.cpp 等。
.h	引用標頭檔，像是 stdio.h 等
.i	經過巨集展開後的 C 語言程式
.ii	經過巨集展開後的 C++ 語言程式
.s	組合語言程式
.S	經過巨集展開後的組合語言程式

C.1 常用的 GNU 工具

GNU 程式集包含許多工具程式，其中，在本書中會用到的有 gcc, as, ld, ar, nm, objdump, objcopy, strip, strings, ltrace 等工具，這些工具的基本用法如表格 C.2 所示。

表格 C.2 GNU 的工具與用法

工具	工具類型	說明
gcc	C 語言編譯器 GNU C Compiler	範例：gcc hello.c -o hello.o
as ⁶	組譯器	範例：as hello.s -o hello.o

⁶ 請注意！在 cygwin 中 as 不好用，因為在 Windows 當中，一個可執行的組合語言需要使用很多函式庫，因此建議直接用 gcc 進行組譯，因為 gcc 會自動連結必要的 C 語言函式庫，其語法範例為 gcc hello.s -o hello.o。

	Assembler	說明：將 <code>hello.s</code> 組譯為 <code>hello.o</code>
<code>ld</code> ⁷	連結器 Linker	範例： <code>ld -o abc.o a.o b.o c.o</code> 說明：將 <code>a.o</code> , <code>b.o</code> , <code>c.o</code> 連結成執行檔 <code>abc.o</code>
<code>ar</code>	函式庫製作 Archive	範例： <code>ar -r libabc.a a.o b.o c.o</code> 說明：將 <code>a.o</code> , <code>b.o</code> , <code>c.o</code> 包裝成函數庫 <code>libabc.a</code>
<code>nm</code>	name mangling ⁸ 目標檔中的符號	範例： <code>nm hello.o</code> 說明：看 <code>hello.o</code> 目標檔的符號表。
<code>objdump</code>	Object File Dump 目標檔傾印	範例： <code>objdump -x hello.o</code> 說明：查看目標檔資訊
<code>objcopy</code>	Object File Copy 複製/轉換目標檔	範例： <code>objcopy -O binary hello.elf hello.bin</code> 說明：將 <code>elf</code> 檔轉換為 <code>binary</code> 檔
<code>strip</code>	Strip 去除除錯資訊	範例： <code>strip a.o</code> 說明：把 <code>a.o</code> 當中的符號表與除錯資訊去除。
<code>strings</code>	觀看字串表	範例： <code>strings a.o</code> 說明：觀看 <code>a.o</code> 檔中的字串表，會顯示符號名稱與分段名稱。
<code>ltrace</code>	追蹤函數呼叫路徑	範例： <code>ltrace a.o</code> 說明：追蹤函數呼叫路徑（在 <code>Cygwin</code> 中沒有）。

GNU 的目的檔工具可分為觀察工具（像是 `objdump`、`nm`、`strings`）與修改工具（像是 `objcopy`、`strip`）等兩類。系統程式設計師通常會用 `objdump` 觀察目的檔，然後利用 `objcopy` 轉換目的檔。

以下，我們將介紹這些工具的基本用法，包含 `gcc` (C.2 節)、`ld` (C.3 節)、`ar` (C.4 節)、`objdump` (C.5 節) 與 `objcopy` (C.6 節) 等，以便讀者在需要使用到這些工具時可以快速查閱之用。

C.2 編譯器 gcc 的用法

GNU 的 C 語言編譯器稱為 `gcc`，這是一個相當強大的編譯工具，同時也具備了組譯與連結的功能，表格 C.3 顯示了 `gcc` 常用的參數名稱與使用方法。

表格 C.3 `gcc` 編譯器的常用參數及其意義

⁷ `ld` 是 Loader 的縮寫，在 UNIX 當中，連結器 (Linker) 與載入器 (Loader) 常是一體的，因此，會用 `ld` 作為連結指令。

⁸ name mangling 又被稱為 name decoration (名稱裝飾)，是在目標檔當中嵌入變數名稱的一種方式，在此，`nm` 被用來列出目標檔中的符號，有關 name mangling 一詞的來源請參考維基百科 http://en.wikipedia.org/wiki/Name_mangling。

參數	範例	說明
-S	gcc -S sum.c -o sum.s	要求 gcc 產生組合語言程式碼
-E	gcc -E hello.c -o hello.i	只執行巨集展開，但不產生目的檔
-D	gcc -DDEBUG sum.c -o sum gcc -DCPU=x86 sum.c -o sum	定義 #define DEBUG 後才編譯 定義 #define CPU x86 後才編譯
-g	gcc -g sum.c -o sum	編譯時加入除錯資訊，讓 gdb 可遠端除錯
-c	gcc -c hello.c -o hello.o	編譯並組譯程式碼，但不做連結
-I	gcc -c -I /home/cxx/include -o hello.o hello.c	指定引用檔 (*.h) 的路徑
-L	gcc -L /home/cxx/lib -o hello hello.o	指定函式庫 (*.a) 的路徑
-l	gcc -L /home/cxx/lib -lm -lpthread -o hello hello.o	指定函式庫的名稱
-shared	gcc -shared a.o b.o c.o -o libabc.so	產生共享函式庫 (*.so)
-fPIC	gcc -g -rdynamic -fPIC -o test test.c	輸出 position-independent code，一般在輸出動態連結函式庫時使用
-Werror	gcc -Werror sum.c -o sum.s	將警告視為錯誤，一但有警告就不輸出目標檔
-O0	gcc -S -O0 sum.c -o sum.s	不進行最佳化 (預設)
-O1	gcc -S -O1 sum.c -o sum.s	第 1 級的最佳化 (較差)
-O2	gcc -S -O2 sum.c -o sum.s	第 2 級的最佳化
-O3	gcc -S -O3 sum.c -o sum.s	第 3 級的最佳化 (最高等級)
-dr	gcc -c -dr sum.c -o sum.o	輸出 RTL 中間碼

C.3 連結器 ld 的用法

GNU 的主要連結工具是 ld 指令，但是在實務上，通常我們會直接用 gcc 進行連結動作，因為 gcc 會幫忙傳送給 ld 去執行，表格 C.4 是 gcc 與 ld 連結時常用的參數表。

表格 C.4 連結指令 ld 的常用參數表

參數	範例	說明
-o <file>	ld -o ab a.o b.o	連結 a.o, b.o 為執行檔 ab
-L <path>	ld -o ab a.o b.o -L /home/lib	指定函式庫搜尋路徑為 /home/lib
-l<name>	ld -o ab a.o b.o -lm	連結函式庫 lib<name>.a，

		本範例連結的是 libm.a
-e <offset>	ld -e 0x10000 -o hello crt0.o hello.o	設定連結啟始位址為 0x10000
-s	ld -s -o ab a.o b.o	移除所有符號
-S	ld -S -o ab a.o b.o	移除除錯符號
-r	ld -r -o romfs.o romfs.img	輸出可重定位 (relocatable) 的檔案
-Map	ld -o ab a.o b.o -Map ab.map	產生連結後的符號表
-T <linkscript>	ld -o ab a.o b.o -T ab.ld	指定 link script 為 ab.ld
-T<段><位址>	ld -o ab a.o b.o -Ttext 0x0 ld -o ab a.o b.o -Ttext 0x1000	指定 text 段位址 指定 data 段位址
-T	ld -o ab a.o b.o -Ttext 0x0 -Tdata 0x1000 -Tbss 0x3000	指定 text 段位址為 0x0、data 段位址為 0x1000、bss 段位址為 0x3000

C.4 函式庫 ar 工具的使用法

ar 指令是 archive 的縮寫，其語法為 ar [options] <archive_file> <src_files>，其中的 [options] 參數如表格 C.5 所示，您也可以使用 ar --help 觀看其詳細的使用方式。

表格 C.5 函式庫指令 ar 的參數表

參數	範例	說明
-r	ar -r libx.a a.o b.o	將 a.o 與 b.o 包裝為函式庫 libx.a
-tv	ar -tv libx.a	查看 libx.a 函式庫的內容
-x	ar -x libx.a a.o	取出 libx.a 中的目的檔 a.o

C.5 目的檔 objdump 觀察工具的使用法

Objdump 是 GNU 的主要目的檔觀察工具，您可用 objdump 顯示目的檔的檔頭、區段、內容、符號表等資訊，表格 C.6 顯示了其使用方法。

表格 C.6 objdump 指令的使用方法與常用參數

語法：objdump <option(s)> <file(s)>		
參數	範例	說明
-i	objdump -i	顯示支援的檔案格式與機器架構

-f	objdump -f a.o	顯示檔頭資訊 (--file-headers)
-h	objdump -h a.o	顯示區段表頭 (--[section-]header)
-x	objdump -x a.o	顯示所有表頭 (--all-headers)
-d	objdump -d a.o	反組譯程式段 (--disassemble)
-D	objdump -D a.o	反組譯全部區段 (--disassemble-all)
-t	objdump -t a.o	顯示符號表 (--syms)
-r	objdump -r a.o	顯示重定位記錄 (--reloc)

C.6 目的檔 objcopy 複製工具的用法

Objcopy 是 GNU 的主要目的檔複製工具，該工具不只可以進行複製，還可以對目的檔進行修改與格式轉換等處理，表格 C.7 顯示了其使用方法⁹。

表格 C.7 objcopy 指令的使用方法與常用參數

語法：objcopy [參數] infile [outfile]	
參數	說明
-I	指定輸入檔案格式 (--input-target)
-O	指定輸出檔案格式 (--output-target)
-B	指定機器架構 (--binary architecture)
-S	去除全部符號資訊 (--strip-all)
-g	去除全部除錯資訊 (--strip-debug)
-j <section name>	只抽取指定區段 (--only-section)
-R <section name>	去除特定區段 (--remove-section)

⁹ 關於 objcopy 的參考資料，請看 <http://www.cmlab.csie.ntu.edu.tw/~daniel/linux/objcopy.html>

附錄D Dev C++ 開發環境

Dev C++ 是學習 C/C++ 語言的學生常用的開發環境，是由 Bloodshed Software 公司所設計的，您可以從 <http://www.bloodshed.net/devcpp.html> 網頁當中下載這個免費的開發工具。

Dev C++ 使用了 GNU 的 gcc，並且使用另外還使用了 Mingw 的函式庫，當您安裝完 Dev C++ 之後，可以從『開始/所有程式/Bloodshed Dev C++』功能表選項中，啟動 Dev C++ 開發環境。

D.1 單一程式的編譯與執行

當您寫了一個 C 語言程式，並且按下功能表中的『Execute/Compile&Run』時，您可以從 Compile Log 這個視窗當中，看到 Dev C++ 所使用的編譯器，GNU 的 gcc 的編譯訊息，圖 D.1 是筆者編譯本書範例 ch01/hello.c 這個程式時所看到的畫面。

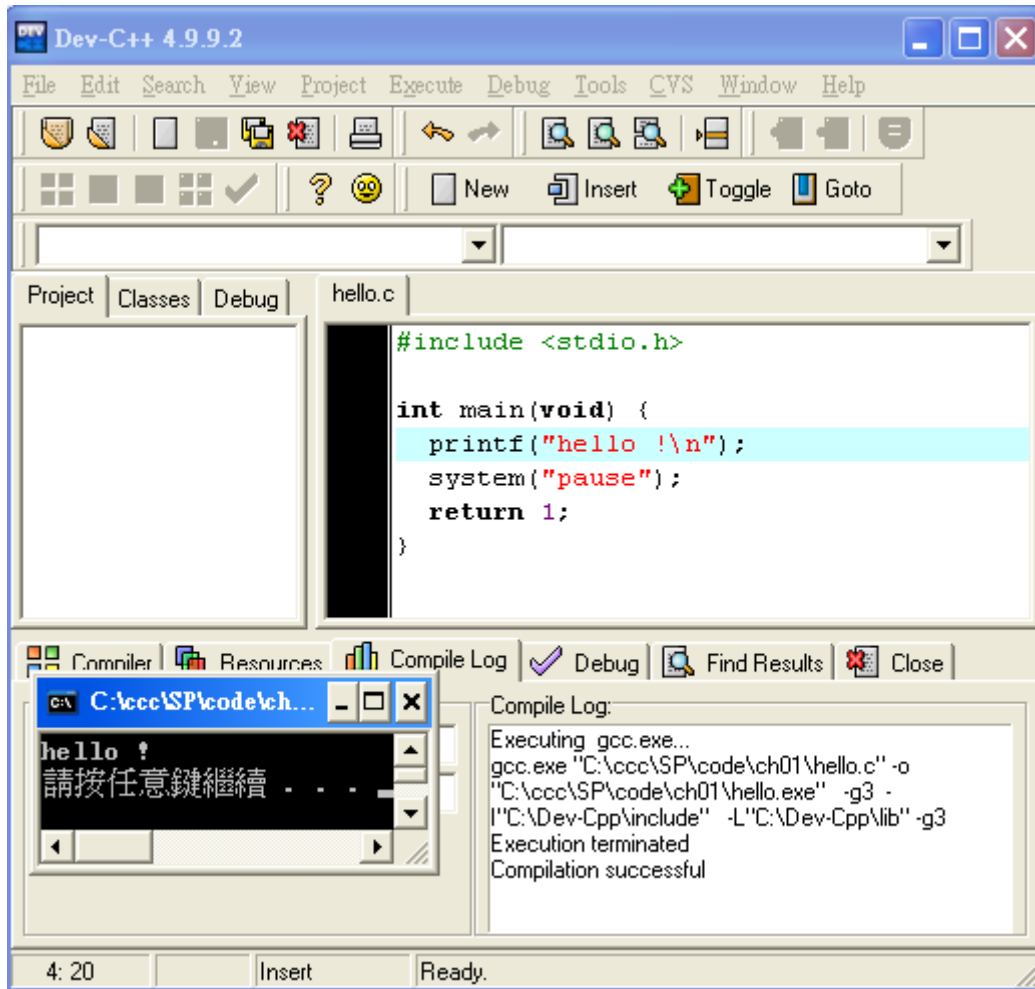


圖 D.1 單一程式檔的 Dev C++ 編譯執行畫面

使用 Dev C++ 撰寫單一程式時，可以直接打開該程式並進行編譯。但是如果有數個程式，就必須先建立專案之後再進行編譯。

D.2 多個程式的編譯與執行

假如您有數個程式要進行編譯與連結，此時您可以用 Dev C++ 建立專案¹⁰，然後同樣按下『Execute/Compile&Run』選項進行編譯。但是，此時 Compile Log 視窗中會顯示 make 的指令訊息，而不再是使用 gcc。這是因為 Dev C++ 自動幫您建立了一個專案檔¹¹，並且利用專案建置工具 make 進行整個專案的編譯動作。圖 D.2 是筆者以 Dev C++ 建置 ch01.dev 專案時所擷取的畫面，您可以看到其中的 Compile Log 視窗當中有 make 指令的訊息。

¹⁰ 您可以於 Dev C++ 當中按下『File/New/Project』選項建立新專案。

¹¹ 您會看到該專案的目錄下會出現一個 Makefile.win 的文字檔，這個文字檔其實就是 GNU 的 makefile 工具所使用的專案檔。

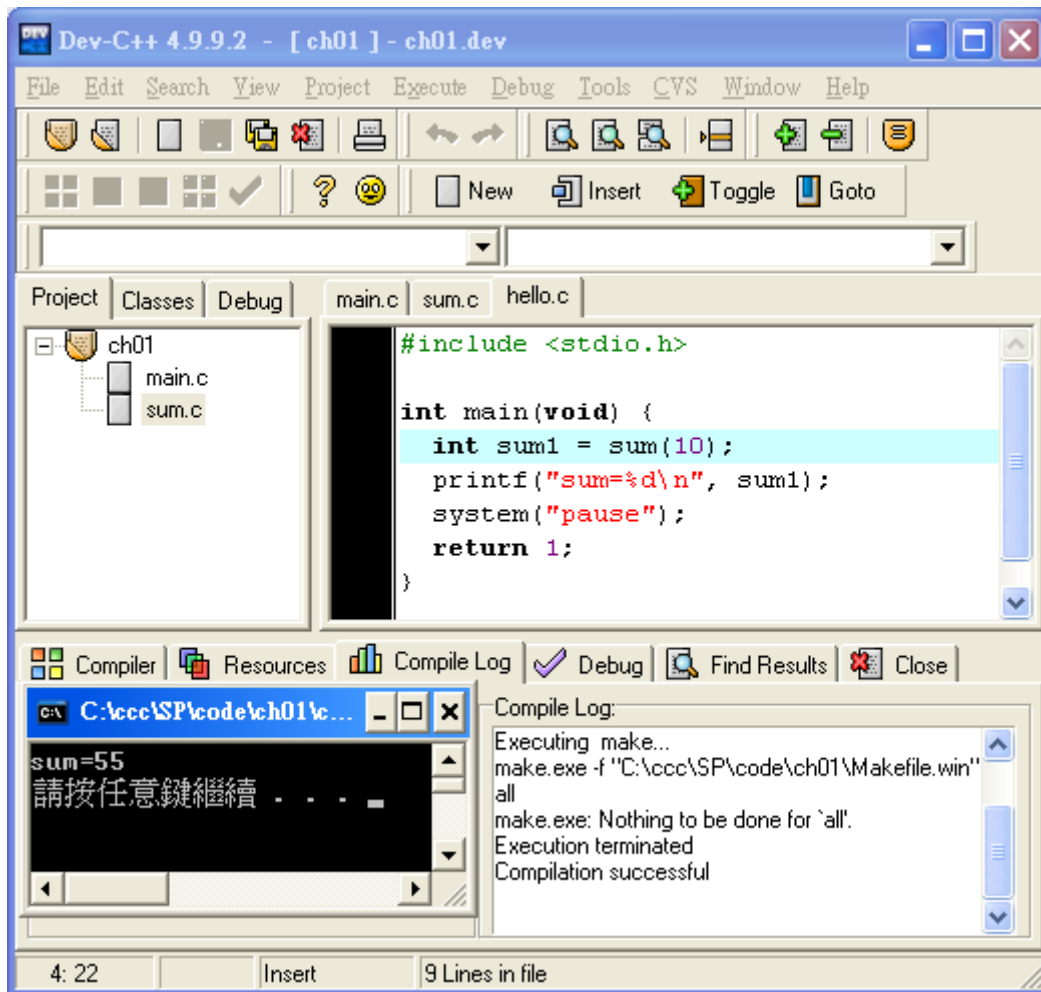


圖 D.2 專案檔的 Dev C++ 編譯執行畫面

從以上的操作當中，您應該可以很容易的看出 Dev C++ 與 GNU 工具的關係，Dev C++ 相當於程式設計師與 GNU 工具之間的一個視窗介面，當使用者按下視窗的選項進行操作時，Dev C++ 會呼叫 gcc、make、gdb 等 GNU 工具以執行編譯與除錯的動作，然後再將訊息回應到相關的視窗當中。

從圖 D.2 的 Compile Log 視窗當中，您可以看到 Dev C++ 是呼叫 `make.exe -f "C:\ccc\SP\code\ch01\Makefile.win"` 指令，以編譯整個專案的。如果您檢視專案所在的資料夾當中，您會發現一個附檔名為 `.dev` 的專案檔，以及一個名稱為 `Makefile.win` 的建置檔。

D.3 自己撰寫專案建置檔

假如您想使用自己寫的專案建置檔，您可以將 Dev C++ 功能選項 Project/Project Options/Makefile 當中的 Use custom Makefile 設定為自己寫的 Makefile，這樣就能讓 Dev C++ 根據自己寫的 Makefile 進行編譯動作，而不會採用 Dev C++ 所

自動產生的 Makefile.win 檔案了。

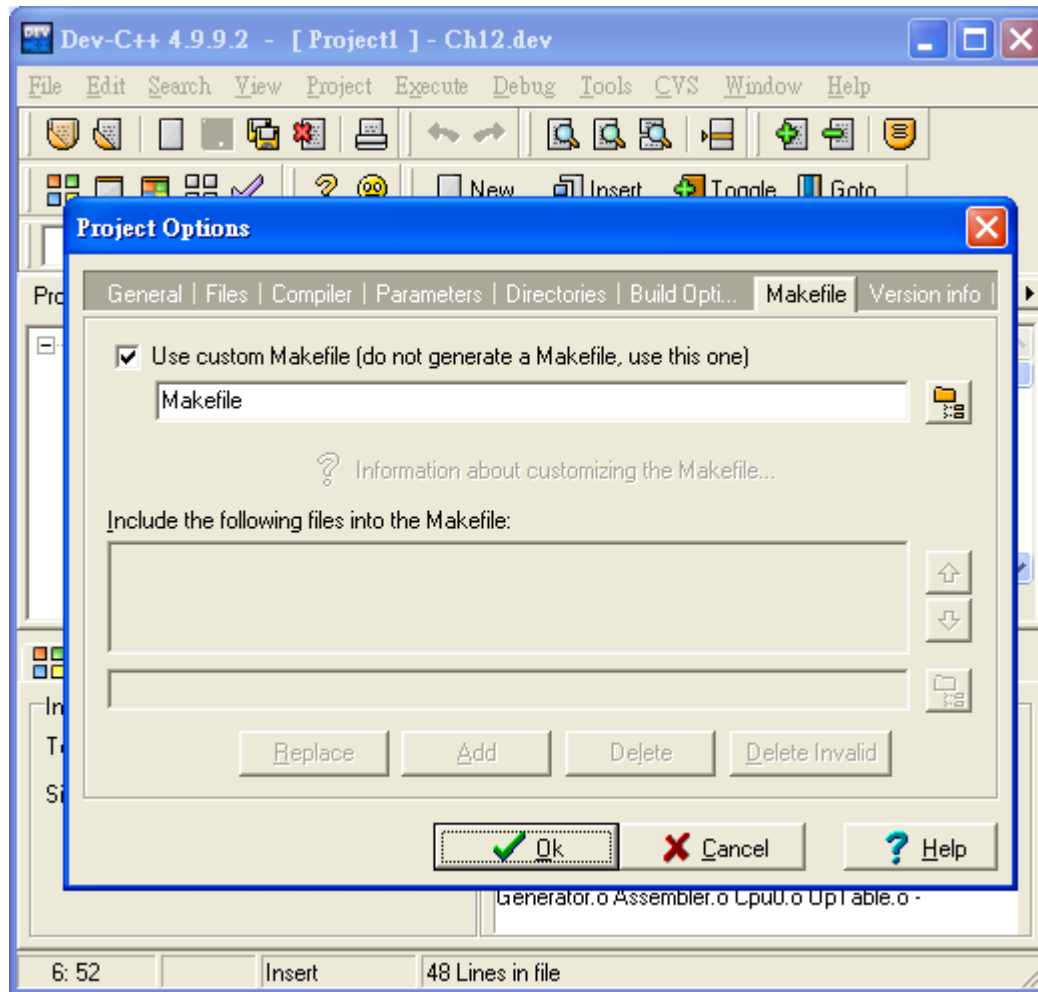


圖 D.3 在 Dev C++ 當中設定使用自己寫的 Makefile

如果您希望讓 Dev C++ 能在按下 Execute/Run 功能時執行自己撰寫的 Makefile 所產生的執行檔，那麼您可以設定 Project/Project Options/Build Options 當中的 override output filename 為該執行檔，如圖 D.4 畫面中的 test.exe 所示。於是當您按下 Run 選項時，就會執行該執行檔 (也就是 test.exe)，如此，您就可以將整個 make 指令的建置與執行過程，直接嵌入到 Dev C++ 的整合環境當中，而不需要回到命令列當中進行操作。

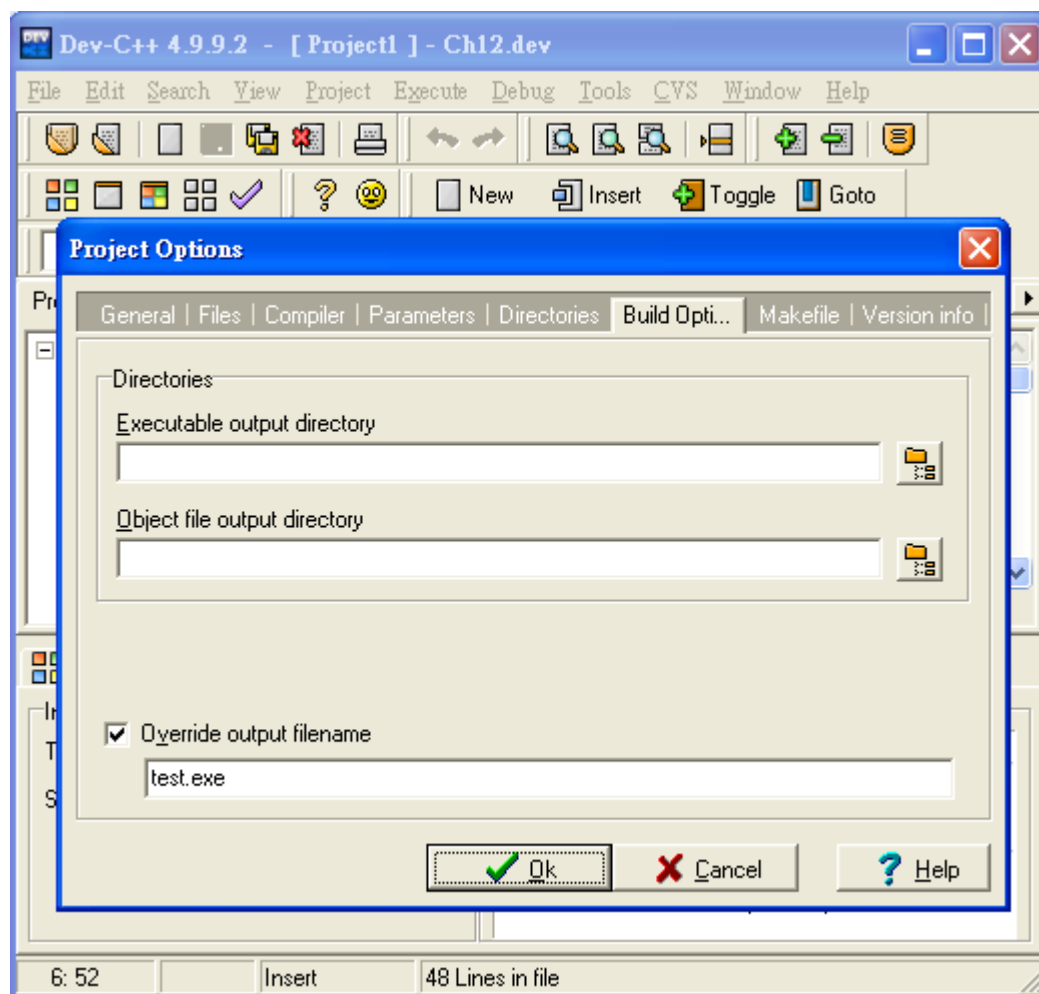


圖 D.4 在 Dev C++ 中設定執行的輸出檔案為 test.exe

D.4 GNU 工具的安裝路徑

在 Dev C++ 當中，GNU 工具到底被安裝在哪裡呢？如果你打開 Dev C++ 的資料夾，您將會看到一個稱為 bin 的子資料夾，裏面包含了 gcc、g++、as、gdb、ld、make 等 GNU 工具，如圖 D.5 所示。

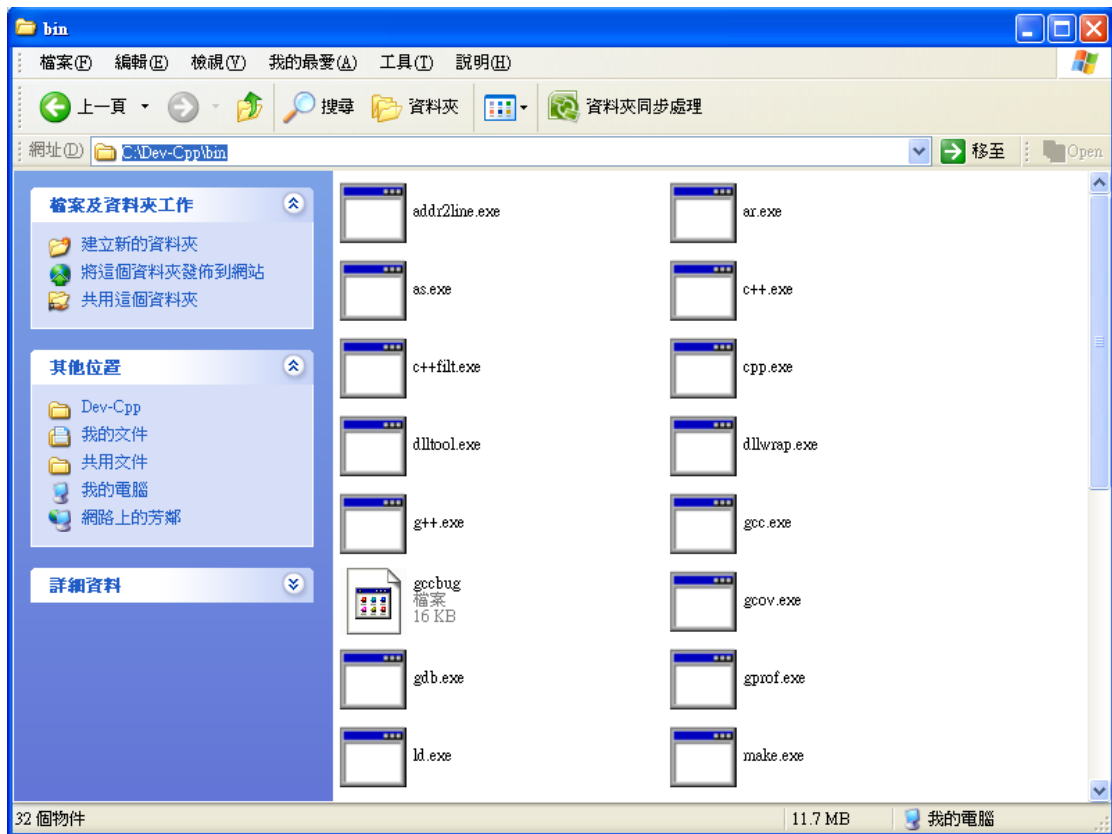


圖 D.5 Dev C++ 中的 GNU 工具

您只要將這個路徑加入到 Windows 系統的 PATH 環境變數內，如圖 1.6 所示，就可以在命令列中使用 gcc、make 等工具，進行編譯與專案建置的動作，而不一定要依靠 Dev C++ 的視窗型開發環境了。

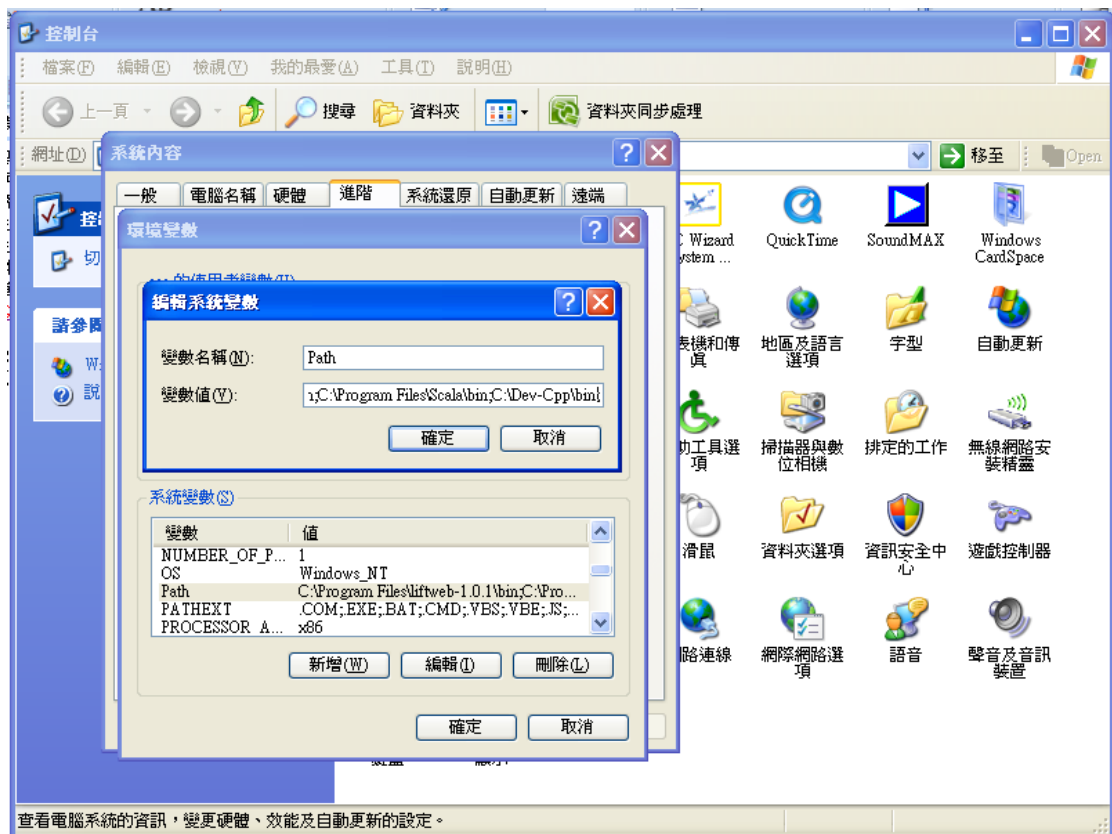
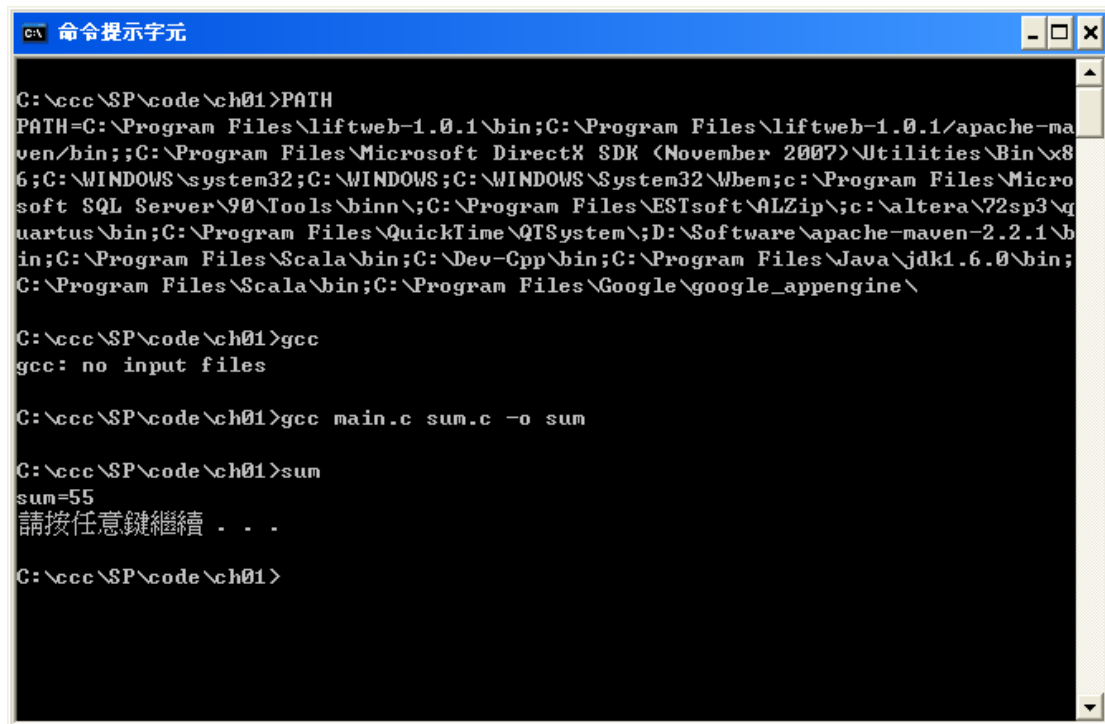


圖 D.6 使用控制台設定 PATH 系統變數

當您設好環境變數後，可以於 MS. Windows 系統中選擇『開始/附屬應用程式/命令提示字元』，然後利用 PATH 指令先檢查看看 \Dev-Cpp\bin 的路徑是否已經設定好了。如果沒有問題，您就可以輸入 gcc 指令看看，假如回應訊息是『'gcc'不是內部或外部命令、可執行的程式或批次檔』，那麼代表您的路徑設定有誤，但是如果顯示的訊息是『gcc: no input files』，那麼代表 \Dev-Cpp\bin 的路徑設定正確，於是您就可以在命令列中使用 GNU 工具了。圖 D.7 顯示了上述的檢查步驟與編譯執行過程。



```
C:\ccc\SP\code\ch01>PATH
PATH=C:\Program Files\liftweb-1.0.1\bin;C:\Program Files\liftweb-1.0.1\apache-ma
ven\bin;;C:\Program Files\Microsoft DirectX SDK (November 2007)\Utilities\Bin\x8
6;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;c:\Program Files\Micro
soft SQL Server\90\Tools\bin\;C:\Program Files\ESTsoft\ALZip\;c:\altera\72sp3\q
uartus\bin;C:\Program Files\QuickTime\QTSystem\;D:\Software\apache-maven-2.2.1\
in;C:\Program Files\Scala\bin;C:\Dev-Cpp\bin;C:\Program Files\Java\jdk1.6.0\bin;
C:\Program Files\Scala\bin;C:\Program Files\Google\google_appengine\

C:\ccc\SP\code\ch01>gcc
gcc: no input files

C:\ccc\SP\code\ch01>gcc main.c sum.c -o sum

C:\ccc\SP\code\ch01>sum
sum=55
請按任意鍵繼續 . . .

C:\ccc\SP\code\ch01>
```

圖 D.7 啟動命令列後開始使用 GNU 工具

Dev C++ 只能在 MS. Windows 下執行，使用的函式庫是 Windows 上的函式庫，因此缺乏許多 Linux 上的函式庫功能。還好，在 MS. Windows 底下有另一套仿造 Linux 所建立的 GNU 工具環境 Cygwin，假如您需要使用到行程管理或執行緒等函式庫，您可以改用 Cygwin 開發環境。

附錄E Cygwin 開發環境

Cygwin 是 Cygnus 公司為 MS Windows 所建構的命令列開發環境，此環境將 Linux 上大部份的工具都移植到 MS Windows 底下，其使用方式幾乎完全仿造 Linux 中的用法。因此，在 Cygwin 中開發出來的程式通常也能直接放入 Linux 下進行編譯執行，反之亦然，這對系統程式的學習而言，有相當大的幫助。

Cygwin 並沒有將 Linux 作業系統整個放入 MS. Windows 中，而是將 Linux 上面的工具程式，放入到 Windows 環境下，並且將許多 Linux 當中的函式庫移植到 Windows 當中，因此，還是有些 Linux 當中的程式無法被放入 Cygwin 當中執行，但 Cygwin 預設所支援的函式庫比 Dev C++ 多一些，像是 fork()、thread 等函式庫，都可以直接在 Cygwin 當中使用。

安裝 Cygwin 這套軟體時，請確認是否勾選了 All/Devel/{gcc、binutils、make} 等選項，如圖 E.1 所示。如此，即可在 Window 當中以 GNU 工具開發 C 語言與組合語言程式，其用法與 Linux 當中幾乎大部分都相同，本書中的範例也都可以 Cygwin 上執行。

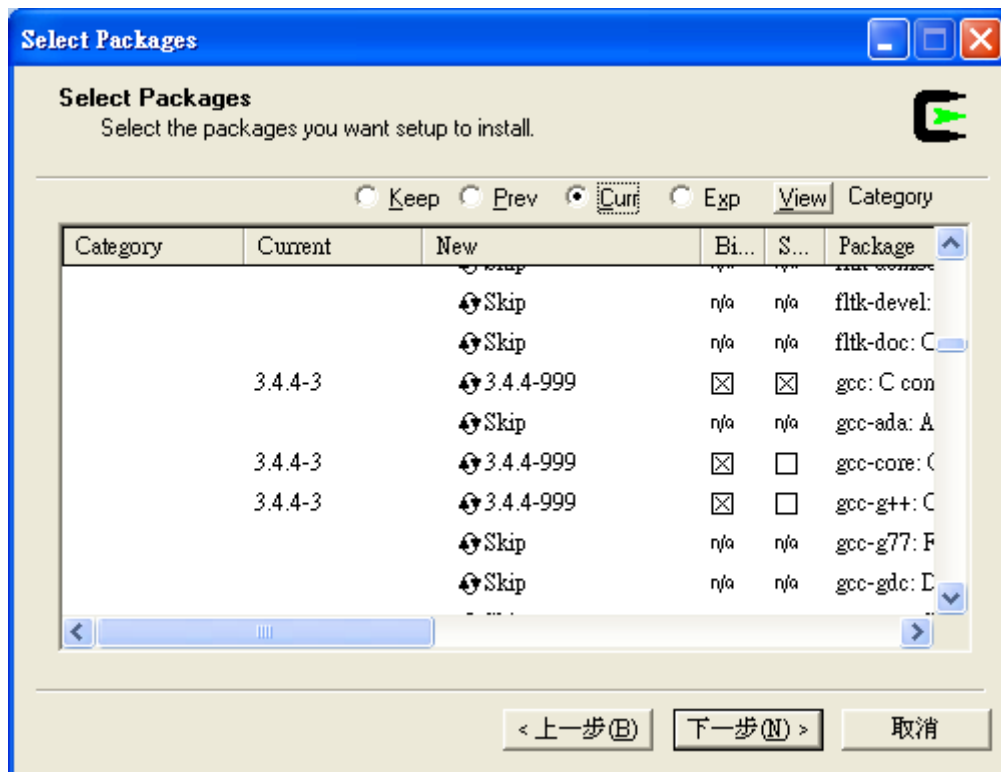


圖 E.1 開發環境 Cygwin 安裝時，請確認勾選了 gcc, make, binutils 等三個選項

當您安裝好 Cygwin 軟體環境後，請選取『開始/所有程式/Cygwin/Cygwin Bash Shell』項目，以啟動 Cygwin 的命令列環境，接著，會顯示出如圖 E.2 的命令列視窗。

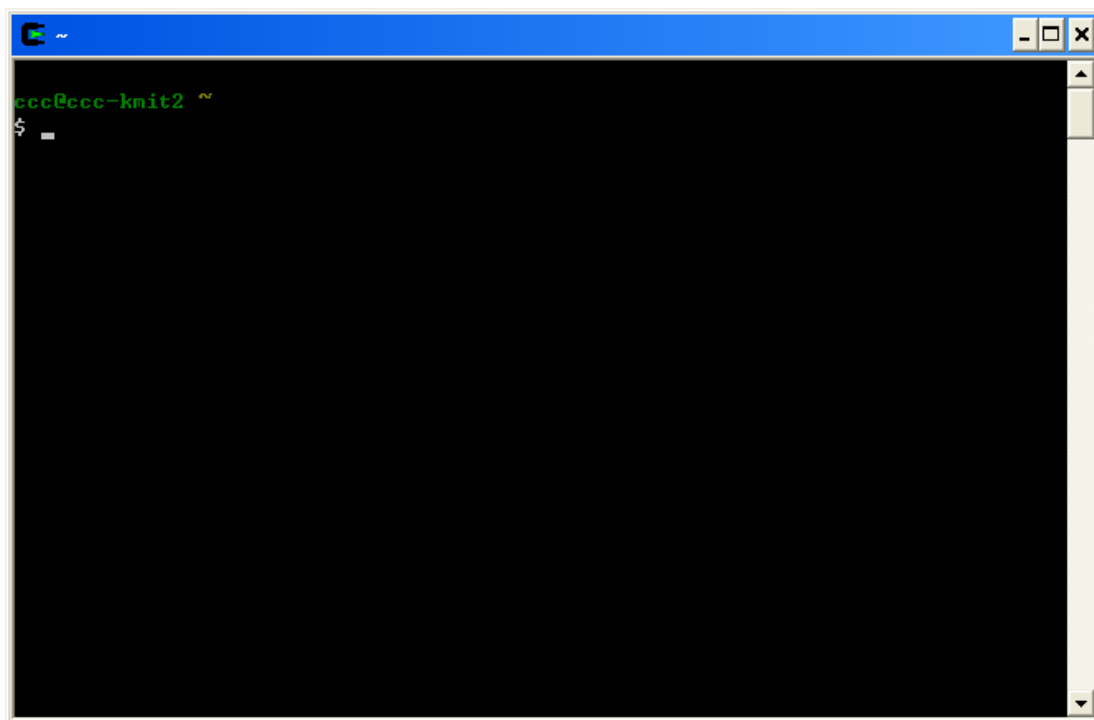


圖 E.2 Cygwin 開發環境的初始進入畫面

E.1 仿照 Linux 的指令環境

在安裝完 Cygwin 環境之後，您就可以開始使用 GNU 工具了。但是，對於習慣使用 MS. Windows 的使用者而言，使用 Cygwin 將會需要學習一些新的指令，因為 Cygwin 的指令名稱與用法，都是仿照 Linux 的指令所設計的。如果您原本就會使用 Linux，那麼您將會感到如魚得水。

Cygwin 所使用的命令列，乃是仿照 Linux 中的 Bash Shell 環境 (此後簡稱 Shell)，Shell 與 Microsoft Windows 的 DOS 命令列有所不同。

在 DOS 環境當中，`dir` 可用來顯示目前的資料夾，但在 Shell 中則必須使用 `ls` 命令，還好，大部分的 DOS 命令都可被對應到某些 Shell 命令，以下是一些常用指令在 DOS 與 Shell 兩個環境下的對照表。

表格 E.1 Linux Shell 與 MS. DOS 與的常用指令對照表

功能	DOS 命令	UNIX/Linux 命令	UNIX/Linux 使用範例	範例說明
檔案列表	<code>dir</code>	<code>ls</code>	<code>ls -all</code>	列出詳細的檔案列表
切換路徑	<code>cd</code>	<code>cd</code>	<code>cd ../</code>	切換到父目錄
複製檔案	<code>copy</code>	<code>cp</code>	<code>cp a.o b.o</code>	將 a.o 複製到 b.o
重新命名	<code>ren</code>	<code>mv</code>	<code>mv a.o b.o</code>	將 a.o 更名為 b.o

刪除檔案	del	rm	rm a.o	將 a.o 移除
建立目錄	md	mkdir	mkdir dir1	建立目錄 dir1
刪除目錄	rd	rmdir	rmdir dir1	移除目錄 dir1
執行檔案	<file>	./<file>	./test	執行檔案 test.exe

在 Cygwin 環境當中，假如您用 `gcc test.c -o test` 這個指令編譯出一個執行檔 `test.exe` 後，您必須用 `./test` 才能執行。這是由於 Shell 採用了 Linux 的習慣的原因，您不能直接輸入檔名 `test` 去執行該檔案。

通常 Cygwin 會被安裝在 `C:\cygwin` 資料夾底下，當您啟動 Cygwin 命令列之後，所在的資料夾會是 `C:\cygwin\home\<user>` 這個資料夾。舉例而言，筆者電腦中的 Cygwin 使用者資料夾就是 `C:\cygwin\home\ccc\` 這個資料夾。您可以直接用檔案總管開啟該資料夾，然後在其中建立一個如範例 E.1 的 `hello.c` 程式並存檔，我們將示範如何在 Cygwin 當中編譯並執行這個檔案。(當然您也可以將程式從其他資料夾複製到 cygwin 的使用者資料夾中，您同樣可以在 Shell 命令列中看到這些檔案)

範例 E.1 程式 `hello.c`

```
#include <stdio.h>

int main(void) {
    printf("hello !\n");
    system("pause");
    return 1;
}
```

當您撰寫好 `hello.c` 之後，您應該將該檔案儲存在 Cygwin 的使用者資料夾中，如圖 E.3 所示。接著，讓我們回到 Cygwin 的命令列環境查看。

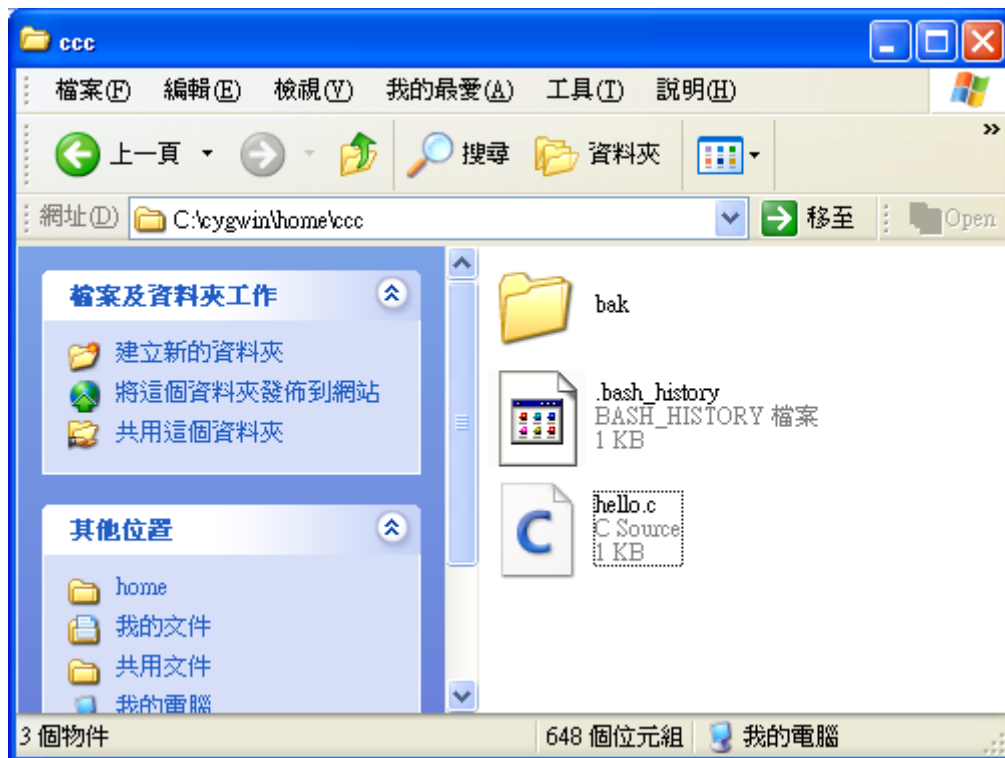


圖 E.3 Cygwin 開發環境的使用者資料夾

如果您在 Cygwin 命令列環境中使用 `ls` 指令列出檔案，您將會看到檔案 `hello.c` 已經存在。此時，您可以使用 `gcc hello.c -o hello` 這個指令編譯 `hello.c`，完成之後再度用 `ls` 列出檔案，看到多出了一個 `hello.exe` 的檔案，然後，您可以使用 `./hello` 這個命令執行該檔案。圖 E.4 顯示了這個編譯與執行的過程。

```

ccc@ccc-kmit3 ~
$ ls
bak hello.c

ccc@ccc-kmit3 ~
$ gcc hello.c -o hello

ccc@ccc-kmit3 ~
$ ./hello
hello !
sh: pause: command not found

ccc@ccc-kmit3 ~
$ ls
bak hello.c hello.exe

ccc@ccc-kmit3 ~
$

```

圖 E.4 在 Cygwin 開發環境中用 gcc 進行編譯與執行的情況

Cygwin 環境中的函式庫與 Dev C++ 有些不同。在使用 Dev C++ 時，我們可以用 `system("pause")` 這樣的函數讓 DOS 暫停，但是在 Cygwin 當中的 `system()` 函數則根本不認識 `pause` 這個指令，因此會顯示『sh: pause: command not found』。這是由於 Cygwin 的 shell 環境沒有 `pause` 指令，因此無法表現出暫停功能。

還好，在大部分的情況之下，如果您使用的是與系統平台無關的標準 C 語言函數，那麼同樣的程式在 Cygwin 與 Dev C++ 當中，會展現出同樣的行為。因此，除了少數與平台相關的範例，像是 Linux 作業系統中的 `fork` 與 `thread` 等主題外，本書中大部分的程式都可以同時在這兩個平台下編譯與執行。