第1章　系統軟體

1.1　何謂系統程式？

系統程式這個名詞很難有一個公認的定義，在台灣大專院校的資訊工程系課程當中，系統程式通常被翻譯為 System Software 或 System Programming，但是，這兩個詞彙雖然緊密相關，但卻有著不同的意義。

System Software 的中文是系統軟體，但是，甚麼是系統軟體呢？一個不精確的說法是：『系統軟體是相對於應用軟體而言的，凡是專門設計給程式設計師使用的軟體，就被稱為系統軟體，而設計給一般大眾使用的軟體，則稱為應用軟體』。

根據上述定義，凡是程式設計師專用的軟體，就稱為系統軟體，因此，組譯器、載入器、連結器、巨集處理器、編譯器、直譯器、虛擬機等等，都很明確的被視為是系統軟體。而試算表、排版軟體、瀏覽器等程式，則是給一般大眾使用的，因此被視為是應用軟體。但是對於某些由程式設計師與一般人共同使用的軟體，像是文字編輯器、資料庫管理系統等等，是否納入系統軟體中，則很難有標準的解釋。

System Programming 的中文是系統程式，指的是系統相關的程式設計技術，然而，在此處的系統一詞究竟指的是甚麼系統呢？這有兩種解釋，一種是作業系統、另一種是電腦系統。因此，系統程式是與『作業系統』或『電腦系統』相關的程式開發的技術。

根據此種解釋，以較為狹義的定義，系統程式專指與作業系統相關的程式設計，包含設計作業系統，以及作業系統層次的程式設計，像是 Linux 系統程式，Windows 系統程式等。這些主題包含『行程管理』、『執行緒』、『行程通訊』、『並行控制』、『記憶體管理』、『檔案輸出入』、『驅動程式』等等，這些主題都是作業系統層次的程式設計重點。

然而，若採用較廣義的解釋，系統程式則可以擴大到與電腦系統相關，特別是與硬體相關的程式，這包含『組合語言』、『C 語言』、『嵌入式系統』等主題，都可以被納入到系統程式當中。而與電腦硬體系統較無關的高階應用程式設計，像是遊戲、資料庫、多媒體程式設計等，則不應納入系統程式的範圍。

那麼，『系統軟體』與『系統程式』兩者之間，又有甚麼關係呢？關於這點，必須從程式設計師的角度，才能得到解答。

當程式設計師撰寫程式時，可能使用『高階語言』或『組合語言』。然而，要讓這些程式能實際在電腦上運行，必須使用許多工具，這些工具就是前述的『系統軟體』。

接著，讓我們看看系統軟體與系統程式之間的關係。

1.2 系統程式與系統軟體

高階語言的程式設計師，會使用『編譯器』將程式編譯為『組合語言』，然後組合語言又再度被『組譯器』組譯，於是產生了『目的碼』。接著，這些目的碼經過連結的程序，形成『可執行檔』。只有在可執行檔接著被『載入器』載入到記憶體之後，程式才會真正開始執行。圖 1.1 是這些過程的流程示意圖。

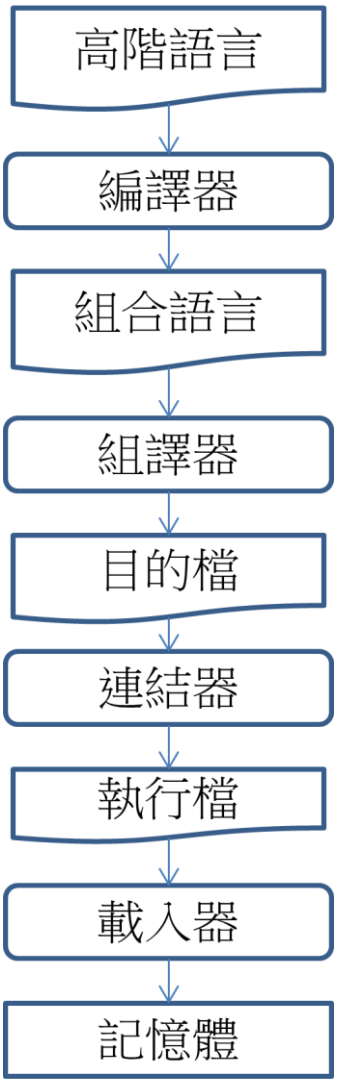

圖 1.1、程式的編譯、組譯、連結、載入之流程

通常，程式開發是一個漸進且冗長的過程，程式人員必須經過撰寫、編譯、測試、修改、再編譯、再測試… 等過程，才能開發出一個有用的程式。因此，這些系統軟體被使用的頻率會非常高。往往在一個軟體的開發過程當中，會產生數百、數千、甚至數萬次的修改。假如程式人員能夠更熟悉這些系統軟體的用法及原理，那麼對其工作將會有莫大的幫助。

從機器語言到組合語言

在電腦的發展史上，最早出現的程式是機器語言，也就是一連串 0 與 1 位元，這也是電腦可以直接執行的語言，舉例而言，機器語言可能如下所示。

範例一： (二進位串列形式的指令) 00010011000100100011000000000000

對於人類而言，通常很難理解這種指令的意義，因為二進位的表示法相當的冗長，難以為人類所辨識。為了讓人們更容易閱讀這些指令，通常我們會將這些二進位碼以每四個一組的方式，中間採用空白隔開，如以下範例所示。這將有助於二進位碼的閱讀。

範例二： (二進位分組形式的指令)：0001 0011 0001 0010 0011 0000 0000 0000

即便如此，對於人們而言，仍然會覺得這樣的表示法過於冗長。為了以更精簡的方式表達上述指令，我們可以用十六進位形式表達上述指令，每四個二進位數字 (0-1) 會對應一個十六進位數字 (0-9, A-F, 其中 A 代表十進位的 10、B 代表 11、以此類推)。因此，範例二的二進位字串將會被改寫為範例三的十六進位字串，讀者可以仔細換算看看。

範例三：(16 進位分組形式的指令)：13 12 30 00

很顯然的，範例三當中的 16 進位形式比起範例二當中的二進位形式短了許多。因此，更容易為人類所閱讀。然而，一般人仍然很難看出該指令的意義。

為了讓人類更容易理解機器指令，於是早期的電腦發明家使用文字的形式，代替這些機器指令。這些用來代替機器指令的文字，就稱為組合語言指令，舉例而言，下列組合語言指令所代表的，正是範例三當中的機器指令。

範例四：ADD R1, R2, R3

你看出來了嗎？範例四的組合語言指令，如何對應到範例三中的機器指令呢？如果我們將這兩個指令寫在一起，會比較容易看出其對應關係，範例五顯示了這個對應關係。

範例五：組合語言與機器指令的對應關係

ADD R1, R2, R3
13 1 2 3 000

在範例五當中，組合語言指令 ADD 被對應到十六進位的指令碼 13，也就是二進位的 0001 0011。暫存器 R1, R2, R3 則直接被編為十六進位碼中的 1, 2, 3，也就是二進位的 0001 0010 0011。最後，由於該指令為 32 位元的指令，因此、我們

必須在未滿 32 位元的部分補上 0 作為結尾。這種對應方式相當明確，只要給出一張指令編碼表，我們也可以透過人腦翻譯，將組合語言轉換為十六進位的指令碼。這個過程，就稱為組譯。

但是，現實生活中的電腦指令，通常沒有這麼容易對應。因為機器指令的格式通常既多樣又複雜，很少像上述範例一樣簡單。還好基本原理是相同的，只要有足夠的耐心，透過一張指令格式與編碼表，人腦仍然可以將電腦的組合語言翻譯為機器指令。

當然，這件事如果要由人類來做，那翻譯指令的人必然為數眾多，而且所做的事情將會非常無趣。所以早期的程式設計師決定讓程式自動完成這件事情，於是撰寫出了組譯器，以便將組合語言轉換為機器指令，再交由電腦執行。

組合語言乃是 CPU 指令集的延伸，撰寫時通常以列為單位，一個指令占據一列，每個指令都是以『指令碼 + 參數』的方式，一列接著一列。例如，以下是一個 ARM 處理器的加法指令。

指令 1：ADD R1, R2, R3

指令 1 的意義相當於 C 語言中的 R1 = R2+R3，但是作用的對象是暫存器，而非 C 語言中的變數，同樣的，ARM 處理器中的減法指令如下所示。

指令 2：SUB R1, R2, R3

指令 2 對應到 C 語言就成了 R1 = R2 – R3。

在 C 語言當中，並沒有暫存器的概念，暫存器是 CPU 當中可用來快速存取的儲存器。通常，如果一個 CPU 是 32 位元的，那其中的暫存器也會是 32 位元的。舉例而言，手機當中常用的 ARM 系列 CPU，通常有 R0, R1, …, R15 等 16 個暫存器，可以供 CPU 儲存運算用的資料。然而，在 CPU 當中，某些暫存器可能具有特殊用途，例如 ARM CPU 中的 R15 就是程式計數器 (Program Counter : PC)，用來儲存指令的記憶體位址，因此也往往被寫成 PC。

有時，由於所使用的組譯器的不同，會導致組合語言的語法也有所不同。舉例而言，假如我們將指令 1 改寫為『目標後置』的形式，則其語法將變更如下。

指令 3：ADD R2, R3, R1

因為在目標後置的組譯器當中，最後一個運算元才是目標暫存器。因此，指令 3 與指令 1 的意義是相同的，同樣對應到 C 語言當中的 R1 = R2+R3。

在指令當中，參數部分並不一定是暫存器，也可能是常數或記憶體位址。舉例而言，下列指令中的 300 就是一個記憶體位址。

指令 4：LD R1, [300]

指令 4 的意義是，要將記憶體位址 300 內的資料值，取出後放入暫存器 R1 當中，其中的 LD 指令代表載入動作，而 [300] 代表位於記憶體 300 當中的值，這個指令若寫成類似 C 語言的寫法，則會是 R1 = [300]。

在某些 CPU 當中，會允許算術指令直接存取記憶體。例如，以下指令會將位址 300 的記憶體內容取出後與 R2 相加，然後再放入到暫存器 R1 當中。

指令 5：ADD R1, R2, [300]

此種設計讓我們可以不用先將記憶體的內容載入，就可以直接進行加法。這個指令若寫成類似 C 語言的寫法，則會是 R1 = R2 + [300]。

在組合語言當中，由於有變數的觀念，所以可以用變數 (像是 COUNT) 取代 [300] 這樣的寫法，以下是一個使用變數的範例。

指令 6：ADD R1, R2, COUNT

在指令 6 當中，我們用變數 COUNT 取代了指令 5 當中的 [300]，其意義相當於 C 語言語法中的 R1 = R2 + COUNT。這樣的寫法較容易閱讀。如果能夠賦予變數適當的名稱，就能更容易的理解變數的意義。

但是，允許算術指令存取記憶體，通常會造成指令長度的增加，因而減慢執行速度。為了避免指令太長，就必須要降低參數個數。例如，將三個參數的加法指令縮減為兩個參數。

指令 7：ADD R1, COUNT

在指令 7 當中，暫存器參數少了一個。那麼，到底變數 COUNT 是與誰相加呢？這個問題的答案是，與第一個參數的暫存器 R1 相加。於是，指令 7 相當於 C 語言中的 R1 = R1 + COUNT。

甚至，對於某些更原始的 CPU 而言，為了把指令格式縮得更短，於是將加法指令的參數縮減為一個，如指令 8 所示。

指令 8：ADD COUNT

與指令 7 相比，指令 8 連目標暫存器都被省略了，只留下變數。那麼，到底 COUNT 是與誰相加呢？加法的結果又該儲存在哪裡呢？

這個問題的答案是，累積器 (Accumulator)。

累積器是一種用來儲存運算結果的暫存器。在使用累積器的 CPU 當中，通常所有的運算都是對累積器進行的。假如累積器是 R0，那麼，指令 8 的意義，就相當於 C 語言當中的 R0 = R0 + COUNT。

組合語言顯然比機器語言容易記憶，而且較容易撰寫，但是，要用組合語言撰寫程式，仍然讓大多數人感到困難無比。

從組合語言到高階語言

為了讓程式設計師能更容易的撰寫程式，於是有人發展出高階語言的概念，第一個出現的高階語言是 Fortran，由 John W. Backus 於 IBM 所發展出來的。後來，許多程式語言陸續被提出來，像是 LISP, Algol, Cobol, C, Prolog, … 等。其中，由 Ken Thompson 與 Dennis Ritchie (合稱 K&R) 在 1972 於貝爾實驗室所發展出來的 C 語言，具有強大的影響力。由於 C 語言是為了開發 UNIX 作業系統而設計的，因此特別適合用來開發系統程式。

C 語言在設計上考慮了許多系統程式的因素，包含與組合語言的銜接、硬體的控制、與執行速度等等。UNIX 的成功也帶動了 C 語言的進一步發展。在今天，C 語言是系統程式、嵌入式系統與作業系統等領域的首選語言。學習 C 語言有助於理解系統程式的概念，像是記憶體映射輸出入，就是一種 C 語言所擅長的系統程式技術。

對程式設計師而言，C 語言比組合語言容易撰寫。在 C 語言當中的一行運算式，可能會需要用許多行組合語言指令才能完成。範例 1.1 顯示了 x=a+3*b-c*d 這個運算式，在轉換成組合語言時，必須用到多達 10 個指令才能完成。

範例 1.1 C 語言與組合語言的對應關係

C 語言	組合語言

x = a + 3 * b - c*d;	LDI R1, 3
	LD R2, b
	MUL R3, R1, R2
	LD R1, c
	LD R2, d
	MUL R4, R1, R2
	LD R1, a
	ADD R2, R1, R3
	SUB R2, R4, R2
	ST R2, x

由於組合語言既難寫又冗長，因此，在目前的產業界當中，只要能使用 C 語言的地方，就會盡可能的避免使用組合語言。

然而，組合語言仍然無法完全被 C 語言所取代，在某些特殊的情況下，仍然必須撰寫組合語言。像是在進行硬體控制時，或者是某些強調速度的應用上，就需要使用組合語言。

以學習的角度而言，組合語言可以幫助人們認識電腦的架構，是學習硬體的捷徑。因此，組合語言也是系統程式課程的核心主題。

在系統程式課程當中，組合語言扮演了軟體與硬體的中介橋樑。由於組合語言是指令集的延伸，整個 CPU 的設計精神可以從組合語言中觀察得到。因此，我們可透過組合語言理解電腦硬體的結構。另一方面，我們也可透過組合語言，學習組譯器、連結器、載入器、編譯器等軟體開發工具。在許多學校的課程安排中，甚至將『系統程式』與『組合語言』合併為同一門課程，這顯示了組合語言與系統程式有極為密切的關聯。

不幸的是，組合語言既多樣且複雜，每種 CPU 都有自己的組合語言。更糟糕的是，為了能與市場中的對手競爭，真實 CPU 通常設計得很複雜。其中最複雜的一種正是目前大多數桌上型電腦所使用的 x86 (IA32) 系列處理器。這對組合語言的初學者而言，無異是雪上加霜，許多學校在教授組合語言時，都被這種情況所困擾，甚至難倒了。

為了降低學習的困難度，在本書中，我們將以一顆簡化過的處理器 CPU0 作為學習對象。由於筆者刻意簡化 CPU0 的指令集與格式，以降低系統程式的學習門檻。

透過這顆簡化過的處理器，我們可以大幅降低組合語言的複雜度，並且讓組譯器的設計變得更簡單，同時也讓目的碼的格式變得更容易閱讀。我們希望能透過這種方式，讓讀者更容易的學會系統軟體與系統程式。

1.3 本書的章節架構

在本書中，我們採取廣義的系統程式定義，同時將 System Software 與 System Programming 的廣義相關主題，盡可能納入本書的各個章節當中。

圖 1.2 顯示了本書的章節導引圖，圖中的組合語言首先被組譯為目的檔，接著連結為執行檔，最後載入到記憶體當中執行，這是組合語言程式的開發流程。

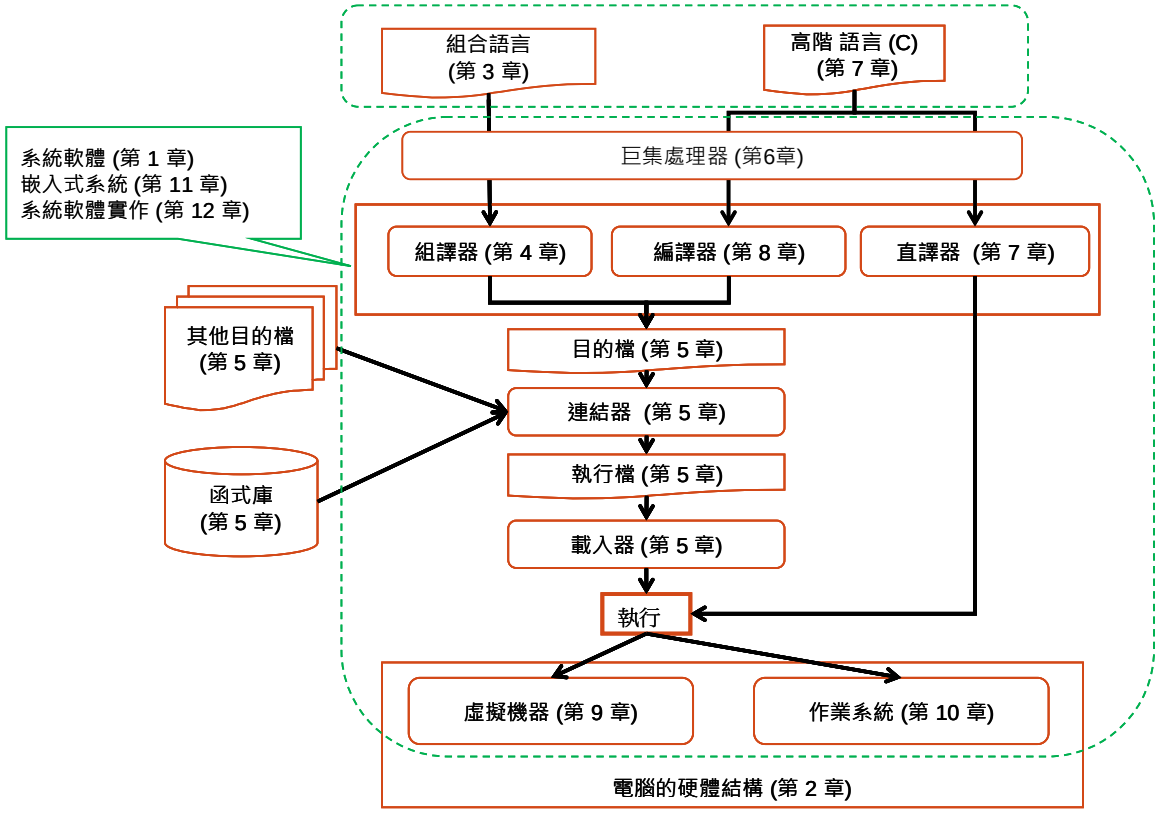

圖 1.2 本書的章節導引圖

在圖 1.2 中同時顯示了高階語言的開發流程。通常，程式語言會在『編譯』與『直譯』兩種執行方式當中選擇一種。編譯式的語言具有較高的執行速度，但是在語法的靈活度與簡潔度上都不如直譯式語言。因此，具有高速度要求的 C 語言通常用編譯的方式，而對速度要求較低，但語法較靈活的 Python、Ruby、Perl 等語言則是使用直譯的方式。

不論是組合語言或高階語言，都需要在某個平台的環境下執行。程式的執行環境可分為三種，第一種是直接在沒有作業系統的電腦硬體上執行 (像是嵌入式系統的程式)，第二種是在有作業系統的電腦，像是個人電腦上執行，最後一種，則是在某個模擬環境當中執行，這類的模擬環境通常稱為虛擬機。一個系統程式設計師，除了要學會程式設計之外，也需要對執行平台有足夠的認識，這是系統程式設計師與一般程式設計師最大的不同點。

在本書的 2-5 章當中，我們會介紹組合語言相關的主題，包含『電腦的硬體結構』(第 2 章)、『組合語言』(第 3 章)、『組譯器』(第 4 章)、『連結與載入』(第 5 章)。這個部份的核心是第 3 章的組合語言，我們會學到組合語言的程式設計方式，電腦的硬體運作原理，以及組合語言相關的軟體工具。

在第 6 章中，我們將討論『巨集處理器』這個主題，這是一個組合語言與高階語言都會使用到的工具，在組合語言中通常會提供 macro 巨集呼叫，而在 C 語言中也有像 #define 這樣的巨集宣告可以使用。

在 7-8 章當中，我們將焦點轉向高階語言，特別是 C 語言上，以便說明『高階語言』(第 7 章)與『編譯器』(第 8 章)的主題。在這個部分，我們將會學習高階語言的語法理論與開發工具，特別是編譯器的的設計原理與使用方式等主題。

接著，我們將焦點轉向程式的執行平台上，說明『虛擬機器』(第 9 章) 與『作業系統』(第 10 章) 的設計原理。在這個部分，我們將學到程式執行平台的原理與特性，包含虛擬機器 (JVM、Dot Net、Virtual PC…) 與作業系統 (Linux、Windows…) 等平台的設計原理。

在第 11 章中，我們會將視野擴大，討論有關『嵌入式系統』的主題。嵌入式系統是一個系統程式的重要領域，這個主題整合了 1 到 10 章的大部分內容，閱讀後會對軟硬體有更深入的理解。這是一個整合性很強的實務領域，也是系統程式的進階主題。

最後，在第 12 章的『系統軟體實作』中，我們將以 C 語言實作本書中重要系統軟體，包含『組譯器』、『虛擬機』、『編譯器』等。以便能透過真實的程式設計範例，更深入的理解這些系統軟體的實作方式。

為了導引讀者進入系統程式的領域，在每本書的每一章當中，都會以理論性的主題開頭，然後在最後一節的『實務案例』中，利用產業界常用的系統軟體，像是『GNU 開發工具』等，進行實務的解說與演練，我們希望讀者不只在理論上能有清楚的認知，也能在實務上有深刻的經驗。

透過這樣的章節安排，我們希望讀者不只能學到系統軟體的設計原理，也能學到系統程式的實務操作。從理論到實務一次貫通，以便在進入產業界之前，就能有足夠的背景知識，以及基本的實作經驗。

1.4 實務案例

現在，就讓我們體會一下系統程式的實務，我們將使用 Dev C++ 與附屬於其中的 GNU 工具，示範系統程式的開發流程，以便讓讀者實際感受系統軟體的使用方式。

1.4.1 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。

本書附錄 D 有 Dev C++ 的使用方式介紹，以供您進一步參考，本節將只用範例導向的方式進行解說。

以微軟的 Windows XP 為例，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。

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++ 的編譯訊息。您會發現 Dev C++ 使用的編譯器是 GNU 的 gcc。圖 1.3 是筆者編譯本書範例 ch01/hello.c 這個程式時所看到的畫面。

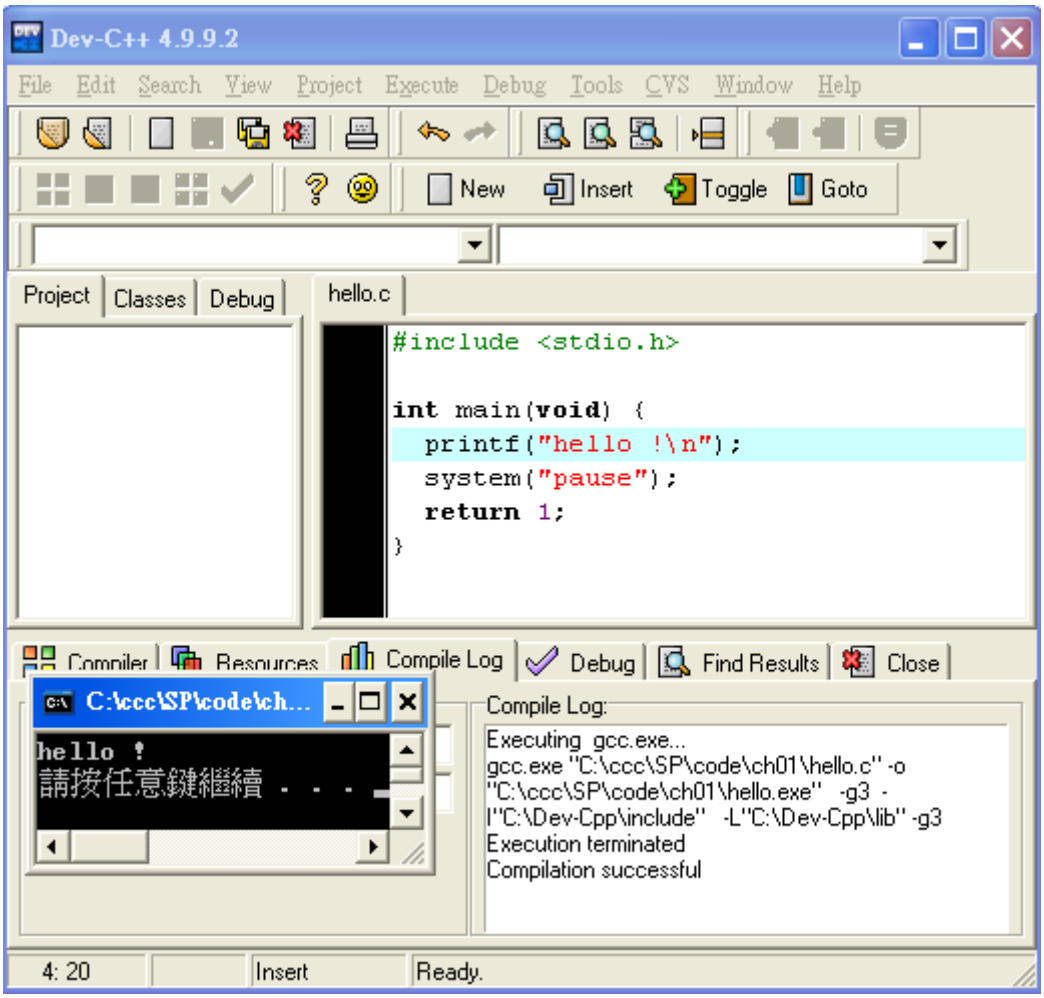

圖 1.3 單一程式檔的 Dev C++ 編譯執行畫面

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，Dev C++ 所使用的編譯器是 GNU 的 gcc 工具，您可以從上圖的 Compile Log 視窗當中看到 gcc 的編譯訊息。

事實上，Dev C++ 所使用的不只是 gcc 工具，而是整套的 GNU 開發工具，這也正是下一節我們將介紹的主題。

1.4.2 GNU 開發工具

GNU 開發工具是由 GNU 組織所設計的，包含 gcc 編譯器、as 組譯器、ld 連結器，make 專案建置工具等。這些工具已廣泛地被使用於系統程式的開發上，甚至，著名的 Linux 作業系統也是用 GNU 工具所開發完成的。

GNU 工具的應用相當廣泛，在 Linux 作業系統中通常預設就安裝了 GNU 工具。在 Windows 系統中，您可以安裝 Dev C++ 或 Cygwin 等軟體，以便使用 GNU 工具。本書的示範將以 Dev C++ 環境為主，但由於 Dev C++ 中的 GNU 工具不支援某些函式庫，特別是像是 fork() 與 thread 等行程管理函數。因此在必要的時候，我們會改用 Cygwin 環境。

在本書的附錄 C 當中，介紹了 GNU 工具的使用方法。以下的操作過程，若有無法理解之處，請參考附錄 C 的解說，以便理解 GNU 相關指令的意義。由於這些操作是在 Dev C++ 的環境之下執行的，也請讀者於必要時先行參考附錄 D 的 Dev C++ 開發環境之主題，以設定 Dev C++ 的命令列環境。如果您需要在 Cygwin 當中使用 GNU 工具，則可以參考附錄 E 的 Cygwin 開發環境一節。

接著讓我們介紹 GNU 工具的使用方法，我們將利用範例導向的方式，導引讀者熟悉 GNU 工具的操作。圖 1.4 顯示了 GNU 工具使用的基本流程。

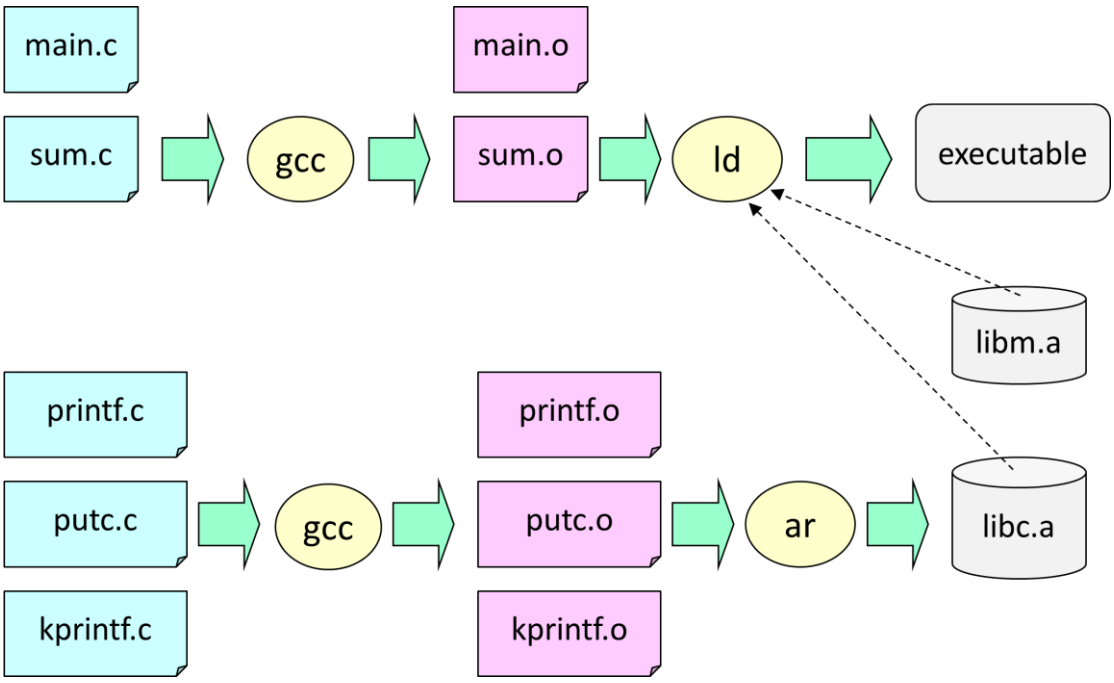

圖 1.4 GNU 工具使用的基本流程

在使用 GNU 工具時，圖 1.4 中的 printf 等函數，通常會是事先被函式庫工具 ar 建立好放入 libc.a 這個檔案中。因此使用者不需再重新建立這些函式庫。程式人員只要負責撰寫 main.c、sum.c 等應用程式，然後利用 gcc 編譯器進行編譯，再用 ld 連結器連結後，輸出執行檔即可。

根據這樣的流程，現在，就請讀者撰寫如範例 1.2 的兩個程式 main.c 與 sum.c。

範例 1.2 程式 main.c 與 sum.c

C 語言主程式 (main.c)	C 語言函數 (sum.c)

#include <stdio.h>
 	
int main(void) {
  int sum1 = sum(10);
  printf("sum=%d\n", sum1);
  system("pause");
  return 1;
}
	int sum(int n) {
	  int s=0;
	  int i;
	  for (i=1; i<=n;i++) {
	  s = s + i;
	  }
	  return s;
	}

接著，您可以使用 GNU 的 gcc 工具對這些檔案進行編譯連結，直接產生執行檔，以下是其執行結果。

```
C:\ch01>gcc sum.c main.c -o sum

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

C:\ch01>dir *.exe
 磁碟區 C 中的磁碟沒有標籤。
 磁碟區序號:  70AE-6E8A

 C:\ch01 的目錄

2010/03/12  上午  09:01                16,019 sum.exe
               1 個檔案              16,019 位元組
               0 個目錄  11,429,384,192 位元組可用

C:\ch01>sum
sum=55
請按任意鍵繼續 . . .
```

圖 1.5 利用 gcc 同時編譯 main.c 與 sum.c 並輸出執行檔

在圖 1.5 中，gcc sum.c main.c -o sum 指令所指定的執行檔名稱為 sum，在 Windows 的環境中，由於預設的輸出副檔名是 .exe，因此您會看到一個 sum.exe 的執行檔被產生出來。[1]

[1] 假如是在 Linux 當中，由於預設副檔名是 .o，因此其輸出檔案名稱將會是 sum.o。

接著，我們將示範如何用 gcc 將 C 語言編譯後轉換為組合語言。這可以利用 gcc 中大寫的 -S 參數完成，參數 -S 用來告訴 gcc 應該產生組合語言而非執行檔。如此，有利於我們觀察組合語言的寫法，以下兩個指令可以分別將程式 sum.c 與 main.c 轉換為組合語言。

```
gcc -S sum.c -o sum.s
gcc -S main.c -o main.s
```

gcc 雖然是個編譯器，但是也可以作為組譯器使用。因此，我們可以利用 gcc 編譯器來『組譯』組合語言。在圖 1.6 中，我們示範了如何利用 gcc 當作組譯器，將 sum.s 與 main.s 兩個組合語言程式，組譯後立即連結為執行檔 sum2。

```
C:\ch01>gcc -S sum.c -o sum.s

C:\ch01>gcc -S main.c -o main.s

C:\ch01>gcc main.s sum.s -o sum2

C:\ch01>sum2
sum=55
請按任意鍵繼續 . . .
```

圖 1.6 將 gcc 當成組譯器使用

除了當組譯器使用之外，gcc 更能將 C 語言與組合語言檔案混合輸入，以單一指令完成編譯、組譯、連結等動作。在圖 1.7 中，我們將示範如何利用 gcc，同時編譯 C 語言檔 main.c 與組合語言檔 sum.s，然後連結並產生執行檔 sum3。

```
C:\ch01>gcc main.c sum.s -o sum3

C:\ch01>sum3
sum=55
請按任意鍵繼續 . . .
```

圖 1.7 利用 gcc 編譯 C 語言 main.c 同時組譯組合語言 sum.s

在本節中，我們介紹了 gcc 的一些基礎用法，然而，gcc 是一套強大而複雜的編譯器，其用法無法在此詳細列出。有興趣的讀者，可以參考附錄 C 中的 gcc 編譯器一節，該小節會有較詳細的 gcc 參數與用法說明。如果這些說明仍然無法

滿足您的求知欲望，您可以參考網路上的 gcc 資源 [2] 與 gcc 官方指南 [3]。

習題

1.1 請說明何謂系統軟體？
1.2 請列出你所知道的系統軟體。
1.3 請說明系統軟體與系統程式兩者有何區別。
1.4 請說明組合語言在系統軟體學習上的角色。
1.5 請說明 C 語言在系統程式上的用途。
1.6 請列出您所經常使用的程式語言，並說明其相關的系統軟體之用法。
1.7 請從網路下載 Dev C++ 軟體，並參照附錄 D 的說明，安裝並使用 Dev C++ 撰寫 C 語言程式，並學習該軟體的用法。
1.8 請找出 Dev C++ 當中的 GNU 工具，並在設定好 PATH 環境變數後，試用 gcc 指令編譯任意一個 C 語言程式 (設定方法請參考本書附錄 D)。

[2] GCC 中文手冊 作者：徐明, http://man.lupaworld.com/content/develop/GCC_zh.htm, 筆者存取時間點為 4/24/2009.
[3] gcc 的官方的說明文件位於 http://gcc.gnu.org/onlinedocs/ , 筆者存取時間點為 4/24/2009.