
現代 C 語言程式設計

Release 1.2.1

Michelle Chen

Mar 18, 2022

目錄

版權聲明 (Copyright)	1
免責聲明 (Disclaimer)	3
版本演進 (History)	5
本書所使用的記述方式	7
C 語言簡介	9
前言	9
為什麼要學 C 語言？	9
C 語言是高階語言還是低階語言？	10
C 語言是跨平台語言	10
C 語言的演進	11
主流系統的 C 編譯器	13
搭配特定硬體的 C 編譯器	14
C 專案	15
C 套件	15
和 C 語言相關的程式語言	15
在 Windows 上建立開發環境	17
前言	17
C 編譯器	17
編輯器或 IDE	19
管理 C 專案	21
安裝第三方函式庫	21
如何選擇？	22
動手做時間：使用 MinGW (MSYS2) 搭配 Code::Blocks	22
撰寫第一個程式	27
在 Mac OS 上建立開發環境	29
前言	29
C 編譯器	29

編輯器或 IDE	29
管理 C 專案	31
安裝第三方函式庫	31
動手做時間：使用 Xcode 建立 C 專案	32
寫第一個程式	34
在 GNU/Linux 上建立開發環境	35
前言	35
C 編譯器	35
編輯器或 IDE	35
管理 C 專案	36
安裝第三方函式庫	37
動手做時間：用 Anjuta 建立 C 專案	37
撰寫第一個程式	41
在命令列使用 GCC 或 Clang	43
前言	43
基本的使用方式	43
檢查編譯器的版本	43
開啟警告訊息	44
開啟除錯相關資訊	44
開啟剖析 (Profiling) 相關資訊	44
選擇編譯最佳化策略	44
編譯多個檔案	45
指定 C 標準 (C Standard) 的版本	45
加入額外的函式庫	46
加入額外的標頭檔或二進位檔位置	46
編譯函式庫	47
小結	47
在命令列使用 Visual C++	49
前言	49
基本的使用方式	49
檢查編譯器版本	49
開啟警告訊息	50
開啟除錯訊息	50
選擇最佳化策略	50
指定 C 標準 (C Standard) 的版本	50
連結外部檔案	51
編譯函式庫	51
用 Intel C++ Compiler 編譯 C 程式	53
前言	53
Intel C++ Compiler 的系統需求	53
取得 Intel C++ Compiler	54

安裝 Intel System Studio	54
在命令列使用 Intel C++ Compiler 編譯 C 或 C++ 程式	61
在命令列使用 Intel Inspector 檢查 C 或 C++ 程式	62
使用 Intel System Studio 隨附的 IDE	63
繼續深入	67
附記	67
善用開發工具改善 C 程式專案	69
前言	69
試著用多種 C 編譯器編譯 C 程式碼	69
編譯自動化軟體 (Build Automation)	69
除錯器 (Debugger)	70
自動程式碼重排 (Code Formatting)	71
靜態程式碼檢查 (Static Code Analysis)	72
記憶體用量檢查 (Memory Checking)	73
測試程式 (Testing)	74
撰寫程式文件 (Documentation)	75
版本控制軟體 (Version Control)	75
基本概念	77
前言	77
C 程式所用的副檔名	77
第一個 C 程式	77
編譯 C 程式的步驟	78
大小寫敏感性 (Case Sensitivity)	78
空白 (Spaces)、縮進 (Indentations)、換行 (End of Line)	79
註解 (Comments)	79
撰碼風格 (Coding Style)	80
標準函式庫	80
引入函式庫	80
主函式 (Main Function)	81
表達式 (Expression) 和敘述 (Statement)	82
離開狀態 (Exit Status)	82
C 執行期函式庫	82
資料型態 (Data Type)	83
前言	83
C 語言的資料型態	83
布林數 (Boolean) (C99)	84
整數 (Integer)	84
固定寬度整數 (C99)	87
浮點數 (Floating-Point Number)	87
複數 (Complex Number) (C99)	90
字元 (Character)	91
字串 (String)	92

列舉 (Enumeration)	92
陣列 (Array)	93
結構體 (Structure)	93
聯合體 (Union)	93
指標 (Pointer)	94
變數 (Variable)	95
前言	95
實字 (Literal)	95
宣告變數	95
使用變數	96
宣告常數 (Constant)	97
合理的變數名稱	97
識別字的命名風格	97
保留字 (Keywords)	98
可視度 (Scope)	99
運算子 (Operators)	101
前言	101
代數運算子	101
二元運算子	102
比較運算子	104
邏輯運算子	105
指派運算子	107
三元運算子	108
其他運算子	108
運算子優先順序	109
選擇控制結構 (Selection Control Structure)	111
前言	111
if 敘述	111
switch 敘述	113
迭代控制結構 (Iteration Control Structure)	117
前言	117
while 敘述	117
do { ... } while 敘述	120
for 敘述	122
break 敘述	124
continue 敘述	125
goto 敘述	125
(選讀實例) 終極密碼	126
指標 (Pointer) 和記憶體管理 (Memory Management)	129
前言	129

記憶體階層 (Memory Layout)	129
靜態記憶體配置 (Static Memory Allocation)	130
自動記憶體配置 (Automatic Memory Allocation)	131
動態記憶體配置 (Dynamic Memory Allocation)	131
空指標 (Null Pointer)	133
比較指標是否相等	134
野指標 (Wild Pointer)	135
迷途指標 (Dangling Pointer)	137
結語	139
陣列 (Array)	141
前言	141
宣告陣列	141
存取陣列元素	142
走訪陣列	143
計算陣列大小	144
動態配置的陣列	146
多維陣列	148
字串 (String)	151
前言	151
C 語言的字串方案	151
C 字串微觀	152
計算 C 字串長度	152
複製 C 字串	153
相接兩個 C 字串	154
檢查兩個 C 字串是否相等	156
尋找子字串	157
結語	158
結構 (Struct)	159
前言	159
宣告結構	159
存取結構內屬性	161
內嵌在結構內的結構	161
儲存結構的陣列	162
宣告指向指標的結構	163
存取結構指標的屬性	164
(選讀案例) 無函式的堆疊操作	165
聯合 (Union)	169
前言	169
宣告聯合	169
存取聯合中的元素	170
內嵌在結構內的聯合	171

內嵌在聯合內的結構	172
列舉 (Enumeration)	175
前言	175
宣告列舉	175
讀取列舉的數字	177
列舉不具有型別安全	177
函式 (Function)	181
前言	181
使用函式的益處	181
宣告函式	181
使用 <code>const</code> 修飾字防止不當修改	184
函式原型	185
不定參數函式	186
遞迴函式	188
傳值呼叫 vs. 傳址呼叫	190
回傳指標	193
回傳多個值	195
函式指標 (Function Pointer)	197
巨集 (macro) 或前置處理器 (Preprocessor)	201
前言	201
閱讀經前置處理器處理過的 C 程式碼	201
用 <code>#include</code> 引入函式庫	201
用 <code>#define</code> 告白巨集	202
(無用) 用巨集創造語法	204
泛型型別巨集 (C11)	206
用巨集進行條件編譯	206
在編譯時期引發錯誤	209
引入各編譯器特有的特性	209
預先定義的巨集	210
巨集對於 C 程式的意義	210
結語	210
函式庫 (Library)	211
函式庫和套件的差異	211
標準函式庫和第三方函式庫	211
C 函式庫的檔案格式	211
實例：二元搜尋樹	212
標頭檔和程式碼間的連動	214
延伸閱讀	215
基本輸出入 (Input and Output)	217
前言	217

學習 <code>stdio.h</code> 函式庫	217
標準串流 (Standard Streams)	217
檔案輸出入	218
格式化輸出的格式	219
格式化輸出函式	220
格式化輸入函式	221
未格式化輸出函式	221
未格式化輸入函式	222
更進一步	227
前言	227
標準函式庫 (Standard Library)	227
資料結構 (Data Structure)	227
演算法 (Algorithm)	228
命令列工具 (Console Application)	228
圖形介面程式 (GUI Application)	228
網頁應用程式 (Web Application)	229
嵌入式裝置 (Embedded System) 和物聯網 (Internet of Thing)	229
編譯器 (Compiler) 或直譯器 (Interpreter)	229
科學運算 (Scientific Computing)	230
學習另一個程式語言	230
附錄：在 Visual Studio 2022 中建立和編譯 C 專案	231
前言	231
安裝必要的工作負載和套件	231
建立主控台專案	232
設置專案屬性	233
使用現代 C 語言的特性撰寫範例程式	234
附錄：實用的 C 巨集	237
前言	237
附帶換行符號的終端機輸出	237
不附帶換行符號的終端機輸出	237
用於除錯的終端機輸出	238
指定範圍的隨機整數	238
基本的數學運算	238
複數	239
無限大、無限小、非數字	240

版權聲明 (COPYRIGHT)

著作權 © 2021 Michelle Chen，保留一切權利。未經授權任意拷貝、引用、翻印、散佈，均屬違法。

若未另外聲明，本書所有的程式碼皆採用 Apache 2.0 授權，歡迎各位讀者在符合授權的前提下使用本書的程式碼。若本書中的程式有使用到第三方軟體，則以該軟體原本的授權方式為準。

免責聲明 (DISCLAIMER)

本書的內容(文字、圖片、電腦程式等)僅為一般性質的資訊，而非正式的技術文件。我們致力於保持本書的內容是即時和正確的，但我們無法保證本書內容的完整性、即時性、正確性、可靠性。本書的內容仍可能因人為錯誤、技術性問題等因素造成錯誤。

此外，我們也無法擔保本書使用者因使用本書或直接或間接受到本書的內容所致的任何損失或傷害。本書使用者應自行評估、判斷本書的內容對自己的風險。

本書使用者可以透過本書的超連結前往外部網站，但我們無法控制外部網站的性質、內容和可得性。我們在本網站中加入這些連結不代表我們推薦這些連結的內容或為這些連結的內容背書。我們無法擔保這些連結的內容。

當你使用本書時，表示你同意本書的聲明，會自行評估、判斷使用本書所導致的風險。

版本演進 (HISTORY)

- 1.2.1
 - 在「實用的 C 巨集」新增跨平台的複數型態
 - 用基於 x86_64 的 Windows 的 Visual Studio 2022 檢查 C 範例程式碼的相容性
 - 用基於 Apple Silicon 的 macOS Big Sur 的 Clang 檢查 C 範例程式碼的相容性
- 1.2.0
 - 改版內容：在 Visual Studio 2022 中建立 C 專案
 - 微調一些版面
- 1.1.0
 - 新增內容：在 Visual Studio 2019 中使用 Clang
 - 新增內容：實用的 C 巨集
- 1.0.5
 - 修改和修正少量文字
- 1.0.4
 - 小幅增修基本概念的章節
- 1.0.3
 - 修改和 Visual C++ 相關的文字
 - 小幅修改 GCC 或 Clang 的章節
- 1.0.2
 - 小幅修改 GCC 或 Clang 的章節
 - 小幅修改 Visual C++ 的章節
 - 小幅修改資料型態的章節
 - 新增計算陣列大小的方式
 - 修復基本輸出入的範例程式
- 1.0.1
 - 修改 C 語言簡介的章節
 - 修改基本概念的章節
 - 修改前置處理器的章節

- 1.0.0
 - 首次發佈

本書所使用的記述方式

對於 C 程式碼，會以語法高亮來輔助閱讀：

```
#include <main>

int main(void)
{
    printf("Hello World\n");

    return 0;
}
```

同樣地，對於 C 程式碼片段，也會以語法高亮來輔助閱讀：

```
/* Excerpt */
assert(0 != strcmp("hello", "goodbye"));
```

為了便於在電子書閱讀器上閱讀本書，我們的範例程式碼不會完全遵守 K&R 風格。我們會儘可能地確保排列過的程式碼仍可正確運作。

按照 Unix 的慣例，終端機會以 \$ 符號來表示指令提示符：

```
$ cd path/to/project
```

當使用 root 操作 Unix 終端機時，則會改用 # 來表示指令提示符：

```
# apt install gcc
```

按照 Windows 的慣例，終端機會以 > 來表示指令提示符。為了簡化，不顯示工作目錄：

```
> cd path\to\project
```


前言

在本章中，我們不急著動手寫程式。而會先對 C 語言做一些概念上的介紹。

為什麼要學 C 語言？

除了大專院校會把 C 當成教學工具外，C 和他的大兄弟 C++ 可說是資訊界最重要的兩個語言。許多重要的軟體專案是以 C 寫成：

- [GNU/Linux](#) 的核心¹
- [Apache](#)²：目前市佔率最高的網頁伺服器
- [Nginx](#)³：另一個快速的網頁伺服器
- [MySQL](#)⁴：知名的關連式資料庫
- [GNOME](#) 桌面系統⁵

許多高階語言的編譯器或直譯器也是以 C 寫成：

- [GCC](#)⁶：GNU 計劃的 C 和 C++ 編譯器
- [Perl](#)⁷：具有強大的文字處理能力的程式語言和命令列工具
- [Python](#)⁸：資料科學的第一把交椅
- [Ruby](#)⁹：Ruby on Rails 的母語
- [PHP](#)¹⁰：全世界約 78% 的網站使用 PHP 實作，包括 Facebook 和 Wikipedia
- [Lua](#)¹¹：最受歡迎的內嵌腳本語言

¹ <https://www.linux.org/>

² <https://httpd.apache.org/>

³ <http://nginx.org/>

⁴ <https://www.mysql.com/>

⁵ <https://www.gnome.org/>

⁶ <https://gcc.gnu.org/>

⁷ <https://www.perl.org/>

⁸ <https://www.python.org/>

⁹ <https://www.ruby-lang.org/>

¹⁰ <https://www.php.net/>

¹¹ <https://www.lua.org/>

許多高階語言的延伸模組內部以 C、C++、Fortran 等編譯語言實作，再用 C API 做高階語言的 binding。絕大部分的高階語言都可以用 C API 寫延伸模組，即使平常甚少寫 C 程式，這個語言的知識不會完全無用。

此外，在嵌入式系統中，由於系統資源受限，通常會用組合語言和/or C 來寫程式，較少用 C++ 或其他的程式語言來實作。

在所有程式語言中，除了組合語言外，最貼近電腦硬體的語言就是 C。對於學習電腦相關知識來說，學習 C 語言會比其他高階語言更有幫助。

雖然現在有一些新的系統語言，像是 D 語言或 Rust，C 並沒有因這些語言出現就衰退掉。至少現有的大型軟體專案不會因為新語言出現就全面重寫。這些新興系統語言只是實作新專案的選項之一，而非用來替代現有的軟體專案。

我們在市面上的基礎教材和大專院校中所學到的 C 算是入門級內容，和前述的重量級軟體專案有一大段落差。此外，一般人用的桌面系統 Windows 對 C 的支援不佳。所以會給人 C 不實用的錯覺。

C 語言是高階語言還是低階語言？

先說答案，C 是高階語言。

真正的低階語言是組合語言 (assembly language)。所謂的組合語言就是直接以處理器 (CPU) 的指令集 (instruction set) 來寫程式。由於每種處理器有自己獨特的指令集，故組語無法跨平台。

相較於組合語言，C 的語法無法直接和處理器的指令集做一對一的轉換，而要經過 C 編譯器將 C 程式碼轉換成等效的組語程式碼後，再由組語程式碼轉為機械碼。所以 C 是高階語言。

有些程式人會認定 C 是中階語言。實際上，沒有什麼所謂的中階語言。C 仍然是高階語言，和其他高階語言的差別只是 C 語言的抽象化程度比較少。由於抽象化程度較少，所以 C 程式碼在編譯時能夠轉換成體積小、效能佳的機械碼。

C 語言是跨平台語言

C 原本是要用來撰寫 Unix 系統的系統程式語言，而非應用程式語言。

Unix 以 C 語言重新實作後，就具有跨平台的能力。當 Unix 要移植到新的硬體上時，只要重寫以組語撰寫的部分即可，以 C 撰寫的部分可以原封不動移到新的平台上。

但撰寫跨平台的 C 程式，不像 Java、Golang 或其他高階語言那麼容易。因為 C 要直接面對異質的系統 C API。除了在語法及標準函式庫層次可自動跨平台之外，往往需要程式設計者費心安排程式碼，以寫出跨平台的 C 程式。

當新的硬體平台出現時，第一個移植的高階語言往往是 C 語言。因為 C 的特性精簡，編譯器相對易實作。做出 C 編譯器後，就可以利用 C 跨平台的特性繼續移植其他的高階語言。

C 語言的演進

有些人以為 C 是老古板，但其實 C 仍在持續演化。我們這裡會簡介到西元 2020 年為止的 C 標準演進過程。

K&R C

K&R C 是 C 尚未標準化前的非正式標準。該規格記載在 Brian Kernighan 和 Dennis Ritchie 所著的 C 經典教材 *The C Programming Language* 第一版中。由於 C 已經標準化了，不需再刻意追隨這個版本的 C。

C89 或 ANSI C

C89 是第一個正式的 C 標準，許多人對 C 的印象就是基於這個版本的 C。由於絕大部分的 C 編譯器至少都支援 C89，有些很在意程式碼可攜性的軟體專案會刻意守在這個版本，像是 Lua。

C99

C99 是第一個 C 標準的重大改版，加入許多新的功能。包括但不限於：

- 新增布林數 (boolean)
- 新增複數 (complex number)
- 新增 long long 整數
- 新增以 // 開頭的單行註解
- 可在 for 迴圈內初始化變數
- 可用自動變數決定陣列長度
- 新增一些函式庫

如果沒寫過 C 程式碼，可能會無法理解這些特性。先大略看過，學一陣子 C 自然會了解。大抵上，C99 相容於 C89，但新增一些功能。

C11

C11 是第二個 C 標準的重大改版，加入許多新的功能。包括但不限於：

- 新型別安全的泛型程式，使用 `_Generic` 保留字
- 支援多執行緒程式
- 加入更多浮點數運算相關的巨集 (macro)
- 支援匿名結構體 (anonymous structure) 和匿名聯合 (anonymous union)
- 改善對統一碼 (unicode) 的支援

同樣地，C11 大抵上相容於先前的 C 標準。

現代 C 語言 (modern C) 是指充份利用 C99 和 C11 的特性來撰寫 C 程式碼，不用刻意守在 ANSI C。善用現代 C 語言所帶來的特性，會讓程式碼更簡潔易讀。

除非專案需要守在 ANSI C 或是所用的編譯器無法充分支援現代 C 語言的特性，我們應該善用現代 C 語言所帶來的便利性。

C17 或 C18

C18 是一個小改版，沒有引入新的語法特性，僅修復一些先前的問題。

C2x

C2x 是最新的 C 標準，這個版本將會引入一些新的特性。但這個版本還沒有定案，所以，不用急著去追新。

Embedded C

初學 C 語言時，會預設 C 程式在個人電腦上運行。相對來說，embedded C 是用於嵌入式裝置的 C 標準，和一般 C 語言有些差異。剛學 C 時不用刻意學這套標準，等到熟悉 C 後再學也不遲。

POSIX

POSIX¹² 不是 C 標準，而是類 Unix 系統的標準，用於 GNU/Linux 等系統。POSIX 包括一套共通的 C API，所以 C 程式碼在不同類 Unix 系統間大抵上是可攜的。由於 GNU/Linux 等類 Unix 系統相當普遍，所以 POSIX 值得關注。

主流系統的 C 編譯器

C、C++、Lisp 等具有歷史的語言往往有語言標準但沒有官方實作品(編譯器)。語言標準只是一份(嚴謹的)技術文件，標準上所列的特性是否能用，還得看所用的 C 編譯器是否有實作。

本節來看目前市面上主要的 C 編譯器對 C 標準的支援情形。由於這裡的資訊會隨時間變動，建議最好查閱一下各編譯器的文件，不要只依賴本文的資訊。

Visual C++

Visual C++ 是隨附在 Visual Studio¹³ 內的 C 和 C++ 編譯器。由於 Visual Studio Community 裝好就有完整的 C 編譯器和 IDE(整合性開發環境)，是很多人學習 C 時所用的工具。

原本 Visual C++ 對 C 標準的支援相對落後，最近(西元 2020 年 11 月)已經修正這個長期議題。現在 Visual C++ 支援最新的 C17(出處¹⁴)。

GCC

GCC¹⁵ 是 GNU/Linux 等類 Unix 系統所用的 C 和 C++ 編譯器，也有移植到 Windows 的版本。GCC 還蠻認真在支援 C 標準上，ANSI C、C99 和 C11 均有支援，而且可藉由參數指定 C 標準。

此外，GCC 自帶特有的 C extension¹⁶，用來強化 C 語言的特性。要注意這些特性不是 C 標準的一部分，會使得 C 程式碼可攜性變差。如果已確認專案只會使用 GCC 來編譯，仍可考慮使用這些特性。

¹² <http://get posixcertified.ieee.org/>

¹³ <https://visualstudio.microsoft.com/zh-hant/vs/>

¹⁴ <https://docs.microsoft.com/en-us/visualstudio/releases/2019/release-notes#16.8.0>

¹⁵ <https://gcc.gnu.org/>

¹⁶ <https://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html>

Clang

Clang¹⁷ 是 MacOS 預設的 C 和 C++ 編譯器，也有移植到 Windows 和類 Unix 系統上。Clang 刻意在參數上和 GCC 相容，但部分特性仍和 GCC 相異，故無法完全取代 GCC。Clang 有支援 ANSI C、C99 和 C11。

站在學習的觀點上，可以考慮用 Clang 來學習 C 語言。因為 Clang 吐出的錯誤訊息比 GCC 友善，對 C 語言初學者來說比較容易除錯。

搭配特定硬體的 C 編譯器

除了上述通用型 C 編譯器，還有一些針對特定硬體所製作的 C 編譯器。我們在本節中介紹幾個常見的例子。

Intel C++ Compiler

顧名思義，Intel C++ Compiler¹⁸ 會對使用 Intel 處理器的電腦做程式碼優化。但對非 Intel 處理器的電腦就吃不到這些福利。根據 Intel 所發佈的調校數據¹⁹，Intel C++ Compiler 所編譯出來的程式的確會比 GCC 或 Clang 所編譯的快一些。

Intel C++ Compiler 主要的特長是優化 C 程式碼，對於 C 學習者來說，不需要刻意用這套編譯器來練習 C。如果對這套編譯器有興趣，Intel System Studio²⁰ 可免費或付費使用，而 Intel Parallel Studio XE²¹ 則需付費使用。

CUDA

CUDA²² 是使用 NVIDIA 的顯示卡進行平行運算的技術。CUDA C 會在原本的 C 語言之外加入延伸語法²³，用來撰寫平行運算相關的程式碼。由於 CUDA 運算需搭配 NVIDIA 顯示卡，而且會用到非標準的 C 延伸語法，一開始不用刻意學 CUDA C。

¹⁷ <https://clang.llvm.org/>

¹⁸ <https://software.intel.com/en-us/c-compilers>

¹⁹ <https://software.intel.com/en-us/c-compilers/features/benchmarks>

²⁰ <https://software.intel.com/en-us/system-studio>

²¹ <https://software.intel.com/en-us/parallel-studio-xe>

²² <https://developer.nvidia.com/cuda-zone>

²³ <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#c-language-extensions>

Arduino

Arduino²⁴ 是開放原始碼的單板電腦 (single board computer)，程式人可使用 C 或 C++ 撰寫在 Arduino 上執行的程式。但 Arduino 程式是用 C 和 C++ 的子集來撰寫，和標準 C 有差異。除非對嵌入式系統有興趣，不需一開始就用 Arduino 學 C。

C 專案

軟體專案將程式碼及編譯組態包在一起，程式設計者可以快速地編譯和執行軟體專案所產出的程式。現代程式語言大都內建軟體專案管理工具，程式設計者就不需使用第三方工具管理軟體專案。

但 C 本身沒有專案的概念，所謂的 C 專案是開發工具後設的功能。當程式設計者用 Visual Studio 寫 C 程式時，C 專案組態是 Visual Studio 內的開發工具來管理，而非 C 程式碼的一部分。

對於初學者來說，不建議直接使用 Make、CMake 等專案管理工具。先用 IDE 內建的功能來管理 C 專案，把心力放在語法的學習上。等到學熟語法後，再開始學習用 Make、CMake 等工具建立跨平台 C 專案。

C 套件

套件是現代程式語言的主要特性之一。套件會將可重覆使用的程式碼區塊包裝起來，搭配套件管理程式自動化安裝和管理。現代語言為了擴展該語言的特性，會內建套件格式及套件管理程式。

但 C、C++ 等具有歷史的語言沒有套件的概念。在 GNU/Linux、BSD 等 Unix 上可見的函式庫套件是系統後設的功能，而非 C 語言的原生特性。由於本書的目標是學習 C 的核心語法，不會用到第三方套件，所以本書不深入討論這個議題。

和 C 語言相關的程式語言

C 語言簡潔的設計，間接影響了許多高階語言的特性。在這些高階語言中，C++ 和 Objective-C 直接和 C 語言相關，可視為 C 語言的超集合。由於 C 語言缺乏內建的物件系統，這兩個語言使用不同的物件系統來改善這項缺失。

²⁴ <https://www.arduino.cc/>

C++

C++ 早期是以 C++ 程式碼轉 C 程式碼的轉譯器來實作，後來則變成實質的編譯器。在 C++ 演進的過程中，加入了許多特性，使得 C++ 成為兼具中階和高階特性的程式語言，相當龐大且複雜。

現在的 C++ 程式包含以下四種面向：

- 命令式程式設計，這部分和 C 語言重疊
- 物件導向程式設計
- 泛型程式設計
- 函數式程式設計

近年來，C++ 飛快地發展，和 C 語言差異越來越大。而且，C++ 並不是 C 語言的嚴格超集。因此，現在普遍認為，學 C++ 時不需先學 C²⁵。

由於 C++ 兼容 C 的特性，在高階語法特性上又比 C 豐富得多，許多程式人在學完 C 之後，轉而學習和使用 C++。除了資源受限等不適合用 C++ 的情境，基本上 C 程式碼都可以用 C++ 改寫。

Objective-C

Objective-C²⁶ 早期使用 C 前置處理器來撰寫，後來演化成實質的編譯器。

在 Swift²⁷ 問世前，Objective-C 是 Mac 家族系統主要的程式語言。目前為了支援現有的軟體，Mac 家族系統仍保留 Objecitve-C API。但由蘋果公司這幾年的發展趨勢來看，蘋果公司幾乎不會再為 Objective-C 開發新特性。

Objective-C 的語法是 C 語言的嚴格超集，但 Objective-C 所依賴的物件庫，目前只在 Mac 和 iOS 上的 Cocoa 有最完整的實作。在非 Mac 家族系統上，只能用 GNUstep²⁸ 這套 Cocoa 的仿作品來替代。但 Cocoa 和 GNUstep 的 API 目前不完全相容。

此外，目前支援 Objective-C 的編譯器 GCC 和 Clang 兩者對 Objective-C 的特性不完全相容。有些新的特性只有在 Clang 上實作，但 GCC 沒有跟上來。所以，Objective-C 沒有辦法像 C 或 C++ 般，成為真正的通用型程式語言。在實務上，可將 Objective-C 視為 Mac 家族系統專用語言。

²⁵ http://www.stroustrup.com/new_learning.pdf

²⁶ https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html#/apple_ref/doc/uid/TP40011210

²⁷ <https://swift.org/>

²⁸ <http://www.gnustep.org/>

在 WINDOWS 上建立開發環境

前言

Windows 有多種 C 編譯器系統，沒有那一種是最佳的方案，而是視自身需求來選擇。目前來說，有以下四種選項：

- Visual C++
- Cygwin
- WSL (Windows Subsystem for Linux)
- MinGW + MSYS

選擇 C 開發環境時，會從 C 編譯器、編輯器、專案管理軟體、套件管理軟體等方面來思考。本文列出在 Windows 上常見的選擇。

C 編譯器

C 編譯器包括編譯器本身和標準函式庫兩部分。這兩個部分會包在一起，安裝 C 編譯器時即一併安裝。本節介紹數個常見的選項。

Visual C++

Visual C++ 是隨附在 Visual Studio 內的 C 和 C++ 編譯器。安裝 Visual Studio Community 時選擇並安裝「C++ 的桌面開發」工作負載 (workload) 即可。安裝完後，會得到 C 和 C++ 編譯器及一套完整的 IDE (整合式開發環境)，可以在完全不使用命令列工具的前提下學 C 語言，所以相當受歡迎。

原本 Visual C++ 無法選擇 C 標準，且在 C 標準上較落後。最近 (西元 2020 年 11 月) 這項議題已經修正了，現在 Visual C++ 支援最新的 C17 (出處²⁹)。

如果只是想用 Visual C++ 編譯 C 程式，但是不需要 Visual Studio 的 IDE 的話，可以考慮下載 Build Tools for Visual Studio。目前 (西元 2021 年三月下旬) 能下載到的有兩個版本：

- Visual Studio Build Tools 2017: https://aka.ms/vs/15/release/vs_buildtools.exe
- Visual Studio Build Tools 2019: https://aka.ms/vs/16/release/vs_buildtools.exe

²⁹ <https://docs.microsoft.com/en-us/visualstudio/releases/2019/release-notes#16.8.0>

下載後，同樣可以按照自己的需求選擇所需的元件，就和使用 Visual Studio 的安裝器雷同。

當我們安裝 Build Tools for Visual Studio 後，由於沒有 IDE 可用，得自行搭配其他的 IDE。常見的選項有 Code::Blocks 或 KDevelop 等。

微軟希望大家直接去用 Visual Studio，所以對 Build Tools for Visual Studio 不太去宣傳，只是低調地放個連結給人下載。但微軟也沒把這些連結拿掉，代表這個工具還是有其需求。

依筆者自己的推測，Build Tools for Visual Studio 主要是用在 Docker for Windows 這類純命令列環境的情境，而不是給一般開發者使用。

Cygwin 內的 GCC

Cygwin³⁰ 是一套運行在 Windows 上的類 Unix 子系統，主要的用途是在 Windows 上使用類 Unix 系統的命令列環境、命令列工具、開發工具等。為了要讓 Cygwin 支援 POSIX，Cygwin 附帶了一隻額外的 DLL (動態連結函式庫)。故在 Cygwin 下編譯的程式皆會依賴該 DLL。

我們會把 Cygwin 視為非 Windows 非 GNU/Linux 的特殊子系統。如果只是要拿內附的 GCC 等開發工具來練習 C 語言倒是無妨。如果想要開發 Windows 原生程式的話，還是使用其他的方案比較好。

Windows Subsystem for Linux 內的 GCC

註：Windows 10 限定。

WSL (Windows Subsystem for Linux)³¹ 是一套運行在 Windows 上的 GNU/Linux 子系統，主要目的是提供類 Unix 系統上的開發工具。

原本的 WSL 是以 Linux 相容介面來運行，內部沒有真正的 Linux 核心。在 WSL2 後，會在該子系統內放入真正的 Linux 核心，利用虛擬化技術來運行該子系統。

由於 WSL 是相對新穎的軟體專案，該專案仍在持續演進中。有時候會在 WSL 上碰到在原生類 Unix 系統上沒有的問題，這些問題可能難以解決或無法解決。故現階段不建議使用 WSL 來練習 C。

³⁰ <https://www.cygwin.com/>

³¹ <https://docs.microsoft.com/en-us/windows/wsl/about>

MinGW + MSYS

MinGW³² 是 GCC 在 Windows 上的移植品，提供 C 和 C++ 編譯器、C 和 C++ 標準函式庫、開發工具等。但沒有第三方函式庫也沒有編輯器，這些部分要另外安裝。至於 MSYS 則是隨附在 MinGW 內的小型類 Unix 子系統，主要用來移植來自類 Unix 系統的自由軟體。

原本的 MinGW 較不親民，後來有開發團隊製作 MSYS2³³。相較於 MinGW，MSYS2 除了提供 C 開發環境外，還加入套件管理軟體，要安裝第三方函式庫就方便得多了。但是編輯器的部分仍需另外安裝。

編輯器或 IDE

編輯器或 IDE 是用來撰寫程式碼的軟體。兩者的差別在於前者功能較單純，專注在撰寫程式碼本身；後者則加入較多功能，像是除錯、重構、專案管理、版本控制等。其實編輯器或 IDE 和 C 編譯器是脫勾的，但 Visual Studio 是少數將兩者同捆的例外之一。

本節介紹一些在 Windows 上常見的選項。

Visual Studio

Visual Studio³⁴ 是微軟官方的 C 和 C++ IDE。由於裝好 IDE 後就可以取得完整的開發環境，故很受歡迎。

除了使用微軟自家的 Visual C++ 外，現在也可以在 Visual Studio 中使用 Clang。原本 Visual Studio 只使用自家的 MSBuild 來管理專案，後來採取較開放的態度，開始接受基於 CMake 組態的專案。

CLions

CLions³⁵ 是一套跨平台的 C 和 C++ IDE。該 IDE 是商業軟體，但提供 30 天試用，可充份體驗後再決定是否要付費。CLions 沒有內建的 C 編譯器，但主流的 C 編譯器均有支援。除了 C 和 C++，CLions 還支援 Objective-C、Python、網頁前端等多種語言。

為了能夠在不同平台下管理專案，CLions 採用 CMake。雖然有一些 CLions 使用者希望 CLions 能夠加入 Makefile 的支援，但 CLions 目前不支援 Makefile。可能的原因是 Make 程式在不同平台間的歧異較大。

不過，CLions 提供一些外掛 (plugins) 來間接支援 Makefile³⁶，有需要的讀者可以參考一下。

³² <http://www.mingw.org/>

³³ <https://www.msys2.org/>

³⁴ <https://visualstudio.microsoft.com/zh-hant/vs/>

³⁵ <https://www.jetbrains.com/clion/>

³⁶ <https://www.jetbrains.com/help/clion/managing-makefile-projects.html>

Code::Blocks

Code::Blocks³⁷ 是一套跨平台的 C、C++、Fortran IDE，不僅開放原始碼還可免費取得。該 IDE 沒有內建的 C 編譯器，可和多種 C 編譯器搭配，包括 MinGW。然而，Code::Blocks 不接受 CMake 或 Make 組態的專案，而是使用自家的專案管理程式。

KDevelop

KDevelop³⁸ 是一套基於 Qt 的跨平台 C 和 C++ IDE，本身是可免費取得的自由軟體。該 IDE 沒有內建的 C 編譯器，可搭配多種 C 編譯器，包括 Visual C++ 及 MinGW。雖然 KDevelop 的名氣不若前述各 IDE 響亮，此 IDE 支援 CMake、QMake、Make 等多種組態的專案，相當方便。

不要用 Dev-C++

Dev-C++ 是一套免費的 C 或 C++ IDE，在台灣的大專院校間很流行。但我們不應再使用 Dev-C++。

Dev-C++ 會紅的原因是早期的 Visual Studio 很貴，免費的 Express 版本功能限制多，所以大家會轉而使用 Dev-C++。但 Dev-C++ 本身已經不再維護了，而且 Visual Studio Community 的功能也相當完整，沒有理由繼續使用 Dev-C++。

其他編輯器

大部分編輯器對 C 都有基本的支援。對於小型檔案，不一定要開 IDE 來用。以下是一些常見的編輯器：

- Atom
- Sublime Text
- Visual Studio Code (VSCode)
- Nodepad++ (Windows 限定)
- Vim
- Emacs

在這些編輯器中，VSCode 是近年來竄升最快的編輯器。該編輯器不僅跨平台，有著豐富的外掛，還支援多種語言，是輕量編輯的最佳選擇。

³⁷ <http://www.codeblocks.org/>

³⁸ <https://www.kdevelop.org/>

管理 C 專案

C 語言本身沒有專案的概念，由各個系統自行提供專案管理方案。常見的有以下數種方案：

- 使用 IDE 內建的專案管理軟體
- 使用 Make
- 使用 CMake
- 使用其他專案管理軟體

對於初學者來說，當下的學習重點在於學會 C 的核心功能，不應耗費過多時間在專案組態上。所以直接使用 Visual Studio 或 Code::Blocks 這類內建專案管理功能的 IDE 無妨。

若是要對外發佈的專案，則不應綁死特定 IDE，這時候應該用 Make、CMake 等不綁定 IDE 的專案管理工具。直接使用 Visual Studio 來管理專案的前提是該專案只會在 Windows 上發佈。

安裝第三方函式庫

在程式寫到一定規模後，標準函式庫會不敷使用，這時候就會尋求適合的第三方函式庫。C 語言有函式庫的格式但沒有套件的概念。在 Windows 上也沒有 `/usr/include` 或 `/usr/lib` 等系統內定函式庫路徑，而是用手動的方式將函式庫的標頭檔及二進位檔拷貝到專案中。

但純手動不易管理，所以發展出下列安裝及管理第三方套件的方式：

- 使用 vcpkg
- 使用 MSYS2 內附的 Pacman

vcpkg 是微軟自家的第三方套件管理軟體。該專案為許多受歡迎的 C 或 C++ 函式庫寫 CMake 組態，再利用 vcpkg 主程式來自動安裝這些函式庫。但目前的 vcpkg 無法自行設置套件的編譯參數，而且只能搭配 Visual C++。

Pacman 原本是 Arch Linux 的套件管理軟體，MSYS2 專案將其移植到 Windows 上，用來搭配 MinGW。同樣地，MSYS2 專案預先將許多受歡迎的 C 或 C++ 函式庫包成套件，所以可以自動安裝。

為什麼會有兩套重覆功能的套件管理軟體？這是為了搭配不同的 C 編譯器所致。

如何選擇？

由於 Cygwin 及 WSL 需在子系統中運行，故先排除。實際在選擇時，就是在 Visual C++ (MSVC) 及 MinGW (GCC) 兩大 C 編譯器系統擇一。

如果程式只要在 Windows 上跑，可以直接用 Visual Studio 全家餐，就會有完整的 C 編譯器 (Visaul C++) 及 IDE。頂多在需要第三方函式庫時拉 vcpkg 來用。

如果程式需要在多個平台上運行，就應該用 MinGW (GCC)。這時候的組合會是 MSYS2 + Make (或 CMake) + 第三方編輯器 (或 IDE)。

對於初學者來說，Visual Studio 或 Code::Blocks 這類內建專案管理功能的 IDE 比較方便。但在跨過新手階段後，應該要試著用 CLions 或 KDevelop 這類可跨平台的 IDE，並用 CMake 或 Make 管理專案。

動手做時間：使用 MinGW (MSYS2) 搭配 Code::Blocks

綜合以上的想法，筆者建議以下的組合 (擇一即可)：

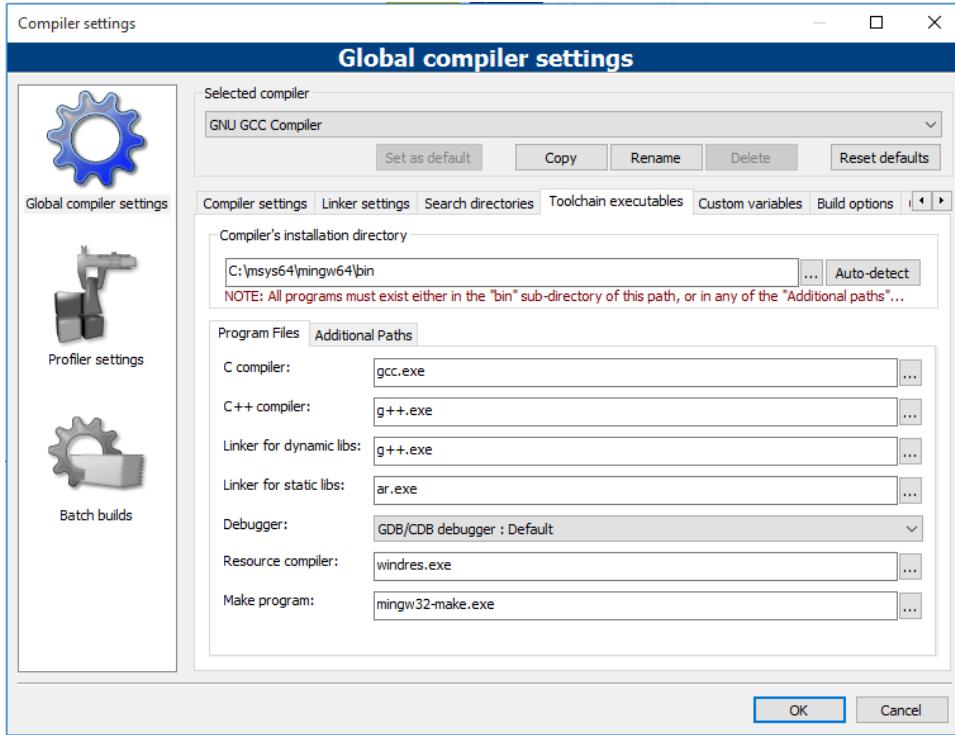
- MSYS2 搭配 Code::Blocks
- MSYS2 搭配 Visual Studio Code

Code::Blocks 的好處在於本身為跨平台軟體，可在 Windows、GNU/Linux、Mac 下執行。對初學者來說，先用 Code::Blocks 內建的專案管理系統來減輕學習的負擔，上手後再慢慢轉到 Makefile 這類純手工的專案管理工具即可。

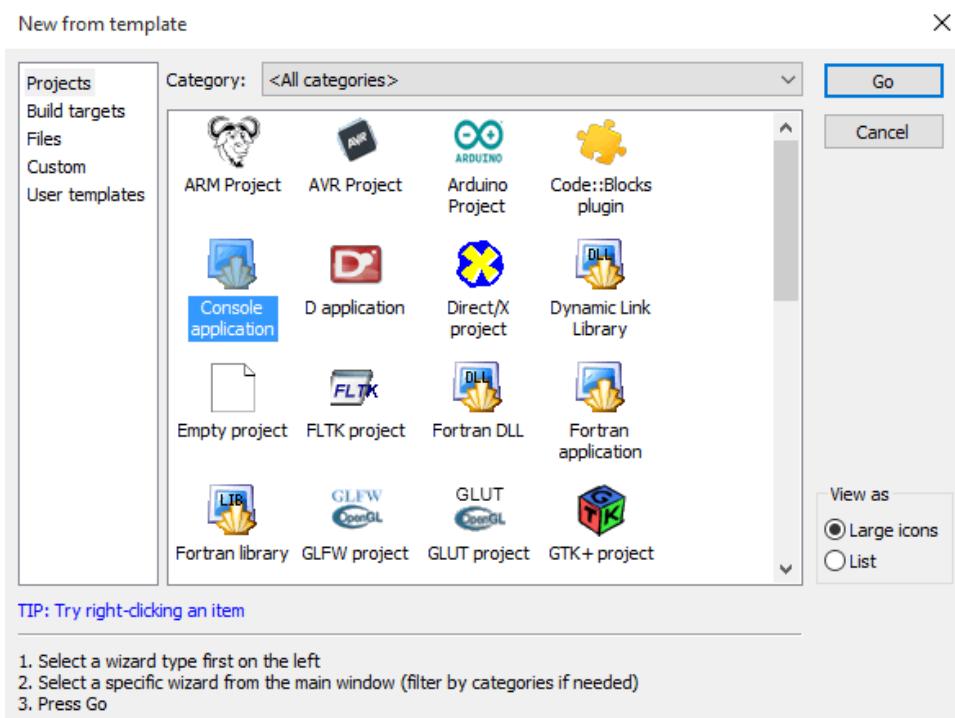
如果要用 Visual Studio Code，建議學一些基本的 Makefile 撰寫方式，以簡化編譯軟體的過程。

接下來，我們用經典的 Hello World 範例來說明如何在 Windows 下撰寫 C 語言。使用 IDE 的讀者，請建立一個終端機程式的專案，使用編輯器的讀者，可建立 *hello.c* 檔案 (也可用其他名稱，建議用英文來命名)。我們這裡以 Code::Blocks 為例，來展示建立專案過程。

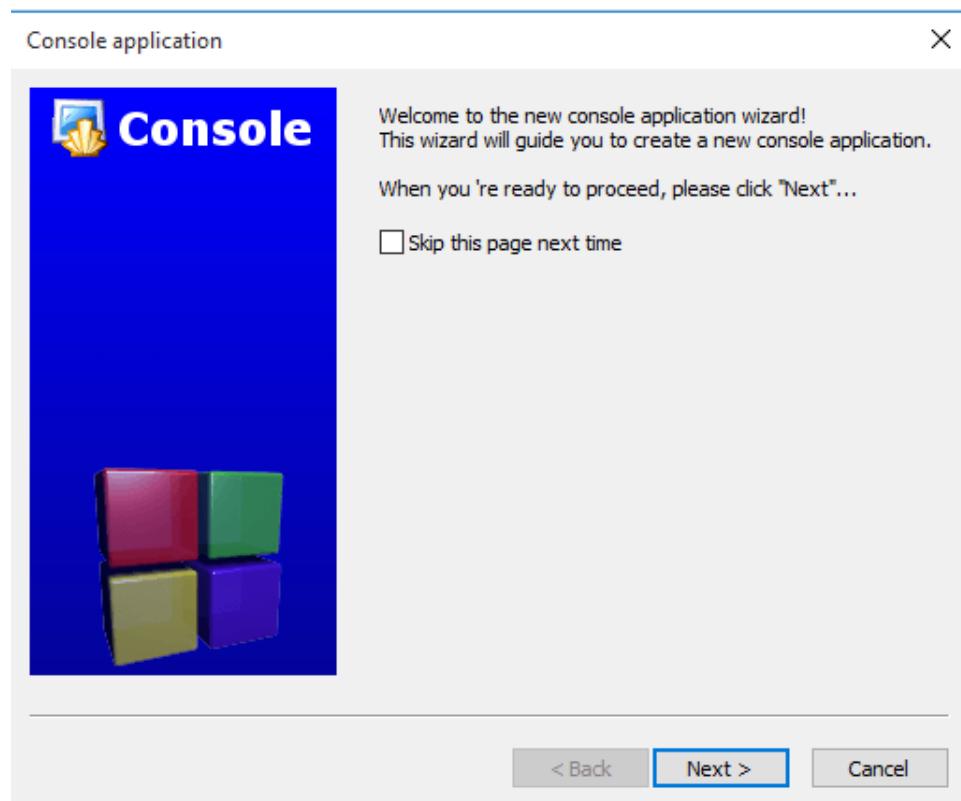
如果 Code::Blocks 無法抓到 GCC (MinGW) 的路徑，需手動修改 GCC 路徑 (從 Settings 的 Compiler 選單選擇)：



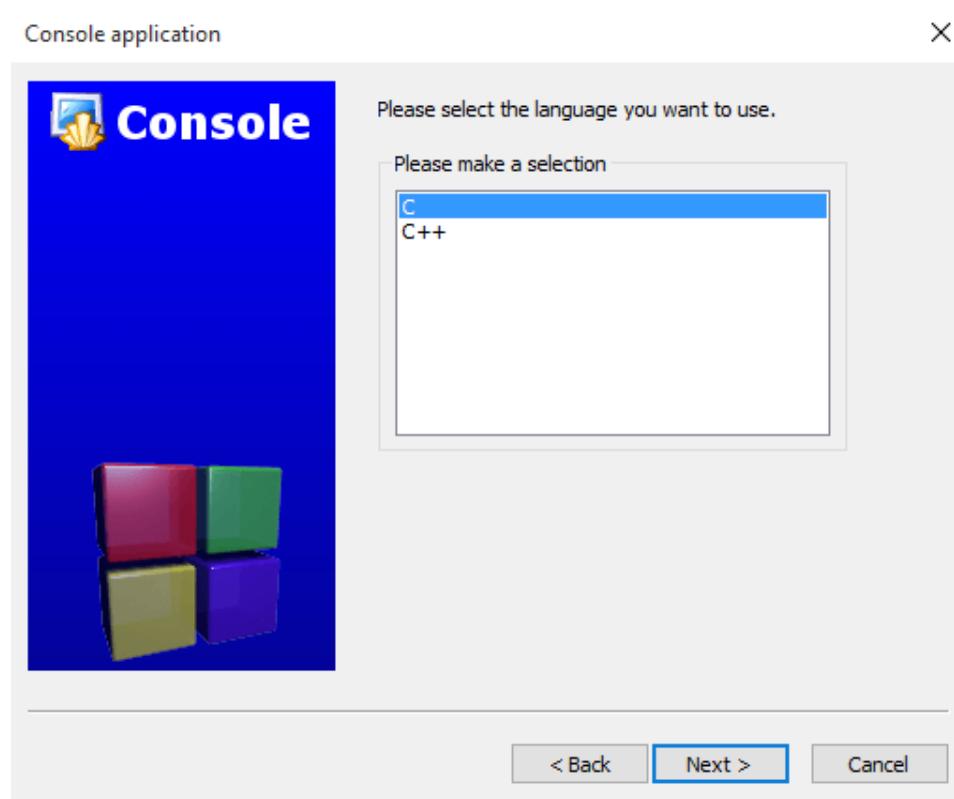
選擇專案類型，這裡選「Console application」：



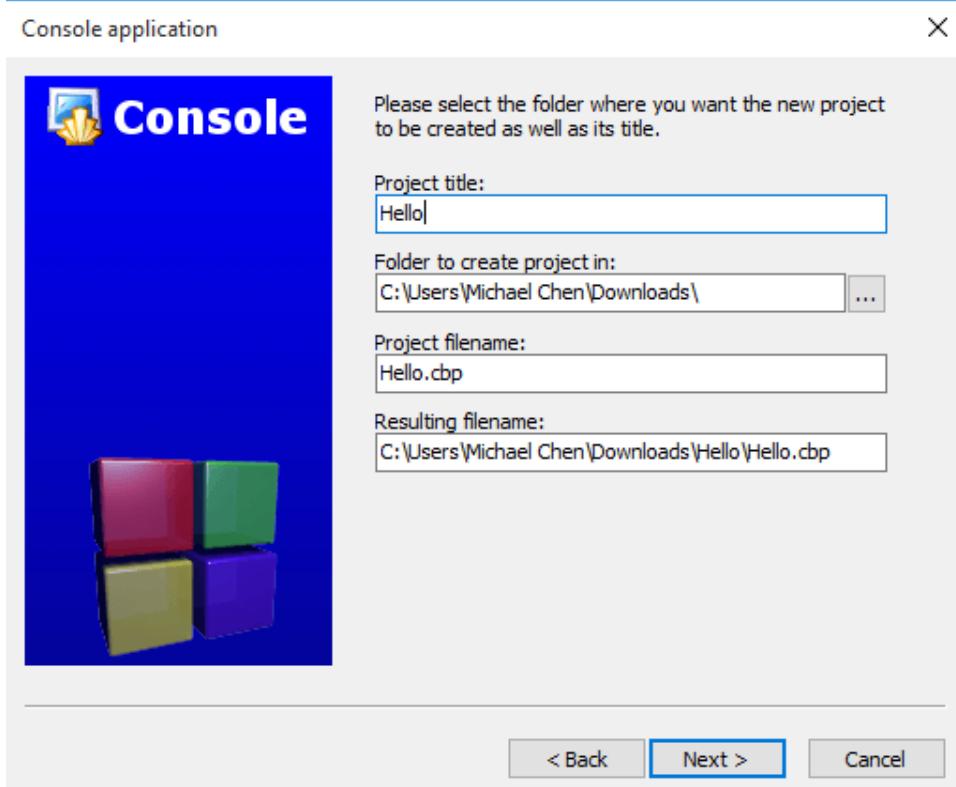
這裡會出現一個額外的提示畫面，僅僅是歡迎訊息，直接選 Next 即可：



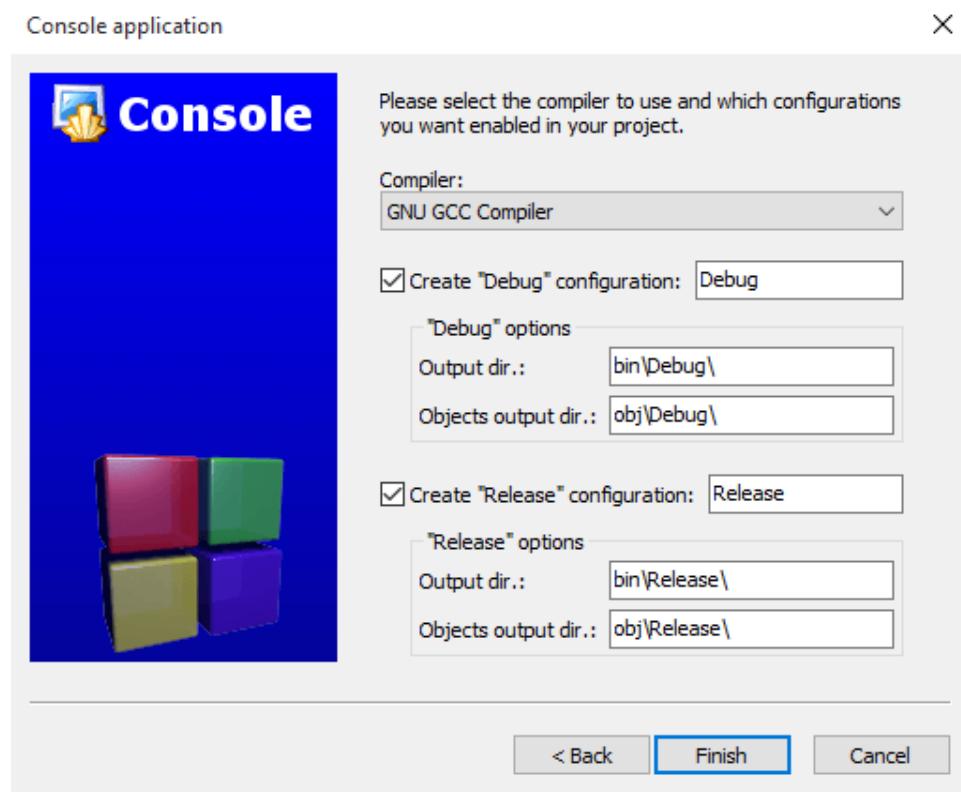
選擇專案語言，記得要選 C，不要選 C++，因為 C++ 並不是 C 的嚴格超集合 (strict superset)：



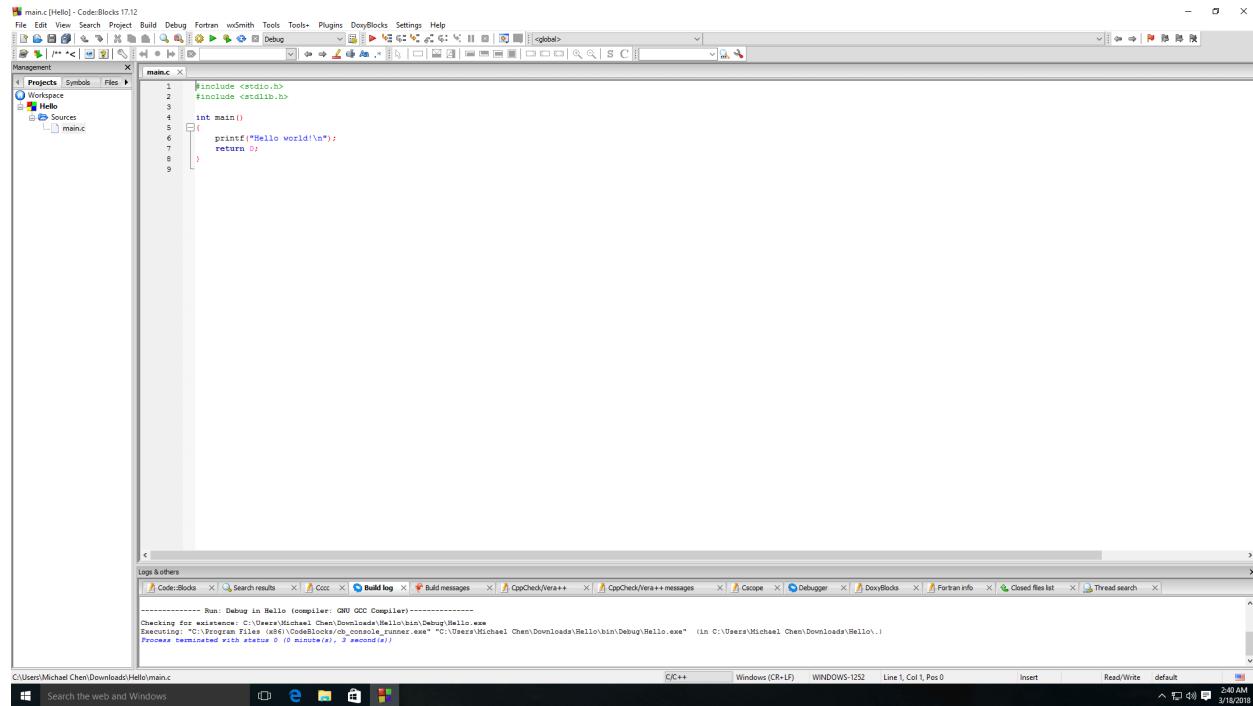
設置專案名稱和路徑：



選擇編譯器，這裡選 GCC 即可：



進入 Code::Blocks 的編輯器，開始撰寫程式：



撰寫第一個程式

我們這裡展示 Hello World 程式，暫時不要管程式碼的意義，這裡的重點是確保程式可順利運行：

```
#include <stdio.h>

int main()
{
    printf("Hello World\n");

    return 0;
}
```

如果使用 IDE 的讀者，選取執行 (Run) 或等效的指令即可執行。如果使用編輯器的讀者，可參考以下終端機指令：

```
> gcc -o hello hello.c
> hello
Hello World
```

對於初期的練習，使用上述方法應該足夠。但若讀者在終端機環境中編譯 C 程式碼，久了應該會覺得反覆打指令很費時。建議使用 Make 等自動編譯軟體來減輕我們的工作量。

在 MACOS 上建立開發環境

前言

MacOS 本質上是一種 BSD 系統，在 C 語言的支援上自然比 Windows 好得多。但蘋果公司不甚注重傳統 Unix 程式設計的發展，等於是只有半套功能的 Unix 系統。所幸，有社群方案可以補足不足的地方，使用起來和 GNU/Linux 等類 Unix 系統的體驗不會差太多。

C 編譯器

MacOS 上預設的 C 編譯器是 Clang，但是也可以自行安裝 GCC 來用。雖然 Clang 有許多參數刻意和 GCC 相容，但兩者仍有一些特性是相異的，所以兩者無法直接相互取代。

編輯器或 IDE

雖然 macOS 也有命令列環境可用，但 macOS 一定會有桌面環境可用，所以不需刻意去用命令列環境中的編輯器。本文介紹數個在 macOS 中可用的編輯器或 IDE。

Xcode

Xcode 是 macOS 預設的 IDE。由於歷史因素，Xcode 可用來寫 C、C++、Objective-C、Swift 等語言。在 Mac 生態圈的思維中，使用者介面是以 Objective-C 或 Swift 來寫，C 或 C++ 只用在非使用者介面的部分。

早期的 C 使用者介面函式庫 Carbon 已經在 macOS Catalina (10.15) 中移除了，所以不應該再去學這套 API。

但 Xcode 算是肥大的軟體，如果沒有要寫 Objective-C 或 Swift 程式的話，可以用其他較輕量的 IDE 來代替。

CLions

CLions 是一套跨平台的 C 和 C++ 商業 IDE，也可以在 macOS 上使用。CLions 本身不自帶 C 編譯器，但可搭配多種外部 C 編譯器，包括 Clang。

為了要在異質平台上管理 C 專案，CLions 直接使用 CMake 來管理專案。雖然有些 CLions 使用者希望 CLions 能支援 Make，但 CLions 官方團隊目前沒有支援 Make 的打算。如果真的想用 Make 的話，倒是可以透過一些外掛來獲得對 Makefile 的間接支援³⁹。

Code::Blocks

Code::Blocks 是一套跨平台的 C、C++、Fortran 等多種語言的 IDE，也可以在 macOS 上使用。但 Code::Blocks 開發團隊目前似乎缺 macOS 版本的維護者，在筆者撰寫文章的時候（西元 2020 年四月上旬），macOS 版本的 Code::Blocks 版本有點滯後。

Code::Blocks 不自帶 C 編譯器，可搭配多種 C 編譯器使用，包括 Visual C++、GCC、Clang 等。但 Code::Blocks 使用自帶的專案管理軟體，無法直接使用 CMake 或 Make。如果專案有跨平台的需求，就不適合用 Code::Blocks 來管理。

KDevelop

KDevelop 是一套跨平台的 C、C++、Python、PHP 等多種語言的 IDE。目前在 macOS 上的版本略為落後。

KDevelop 不自帶 C 編譯器，可和系統預設的 C 編譯器搭配使用。此外，KDevelop 可以直接吃入以 Make、CMake、QMake 等多種自動編譯軟體管理的專案。所以，如果需要一個免費 IDE，且專案有跨平台的需求，倒是可以考慮使用 KDevelop。

其他編輯器

除了上述 IDE 外，對於小型練習程式來說，也可以直接用編輯器來寫。以下是一些常見的跨平台編輯器：

- Atom
- Sublime Text
- Visual Studio Code (VSCode)
- Vim
- Emacs

³⁹ <https://www.jetbrains.com/help/clion/managing-makefile-projects.html>

由於 Vim 和 Emacs 上手難度較高，沒學過的讀者不用一開始就刻意去學。可以先試其他比較易上手的編輯器。

管理 C 專案

在寫 C 程式時，要根據自己的目的來選 C 專案的管理方式。對於初期的語法練習，直接使用 IDE 內建的專案管理比較簡單。因為這時候的重心在學習語法上，管理專案算是額外的工作，能省就省。

如果是要公開發佈或是團隊協作的 C 專案，則要慎選自動編譯軟體。多會使用 Make 或 CMake 這類跨平台的自動編譯軟體來管理專案，而不會用 IDE 來管理專案。因為要考慮團隊成員可能使用相異的平台或工具來撰寫程式。

安裝第三方函式庫

雖然 macOS 是 BSD 系統，卻沒有官方的套件管理軟體。這是因為蘋果公司把 macOS 視為開發 iOS 和 macOS 軟體的平台，不注意傳統的 Unix 程式設計。目前這個議題由社群方案來補足。最常見的套件管理軟體是 Homebrew 和 MacPorts。

Homebrew

Homebrew 是一套以 Ruby 實作的套件管理軟體，應該是目前最多人使用的套件管理軟體。由於 Homebrew 預設把套件裝在 */usr/local* 目錄的子目錄，不會覆寫系統上的同名軟體。此外，Homebrew 在預設情形下會避開系統上已有的 C 函式庫，不用擔心在編譯 C 程式時相依到某個 Homebrew 套件。

MacPorts

MacPorts 是一套以 Tcl 實作的套件管理軟體，目前 MacPorts 使用者沒有 Homebrew 使用者來得多，但仍然有持續維護，而且套件也不少。

MacPorts 在編譯及安裝時可指定安裝位置。按照 Unix 系統的慣例，常見的安裝位置有 */usr/local* 或 */opt*。讀者也可以視需求將 MacPorts 安裝到其他位置。MacPorts 會把軟體相依性限制在安裝位置內，不會影響到系統目錄及檔案。

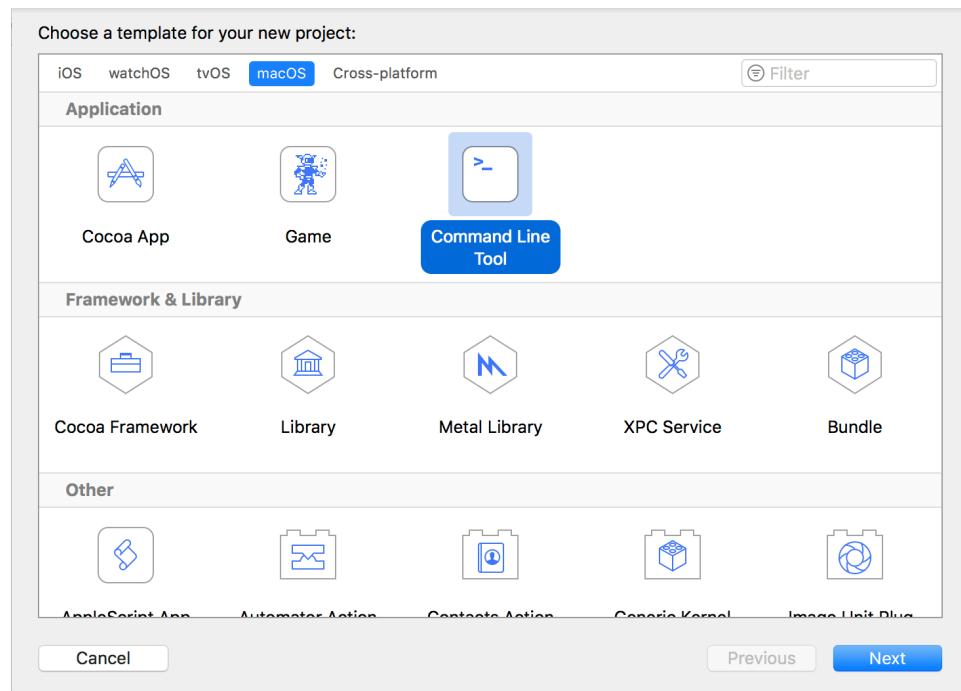
使用 Homebrew 或 MacPorts 的注意事項

最好不要混用套件管理軟體，以免發生預期外的錯誤。標準做法是移除其中一套軟體後，再安裝另外一套。

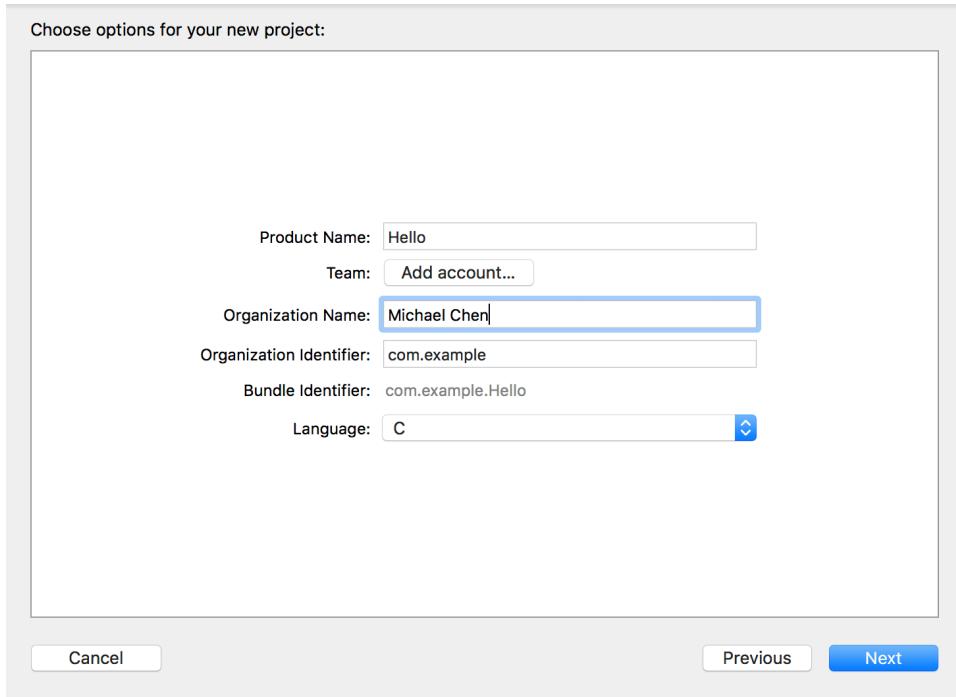
動手做時間：使用 Xcode 建立 C 專案

接下來，我們用經典的 Hello World 範例來說明如何在 Mac 下撰寫 C 語言。如果使用 IDE 的讀者，請自行建立一個終端機程式類型專案，如果使用編輯器的讀者，建立一個檔名，像是 *hello.c* (可取其他檔名，建議用英文命名)。我們這裡以 Xcode 為例：

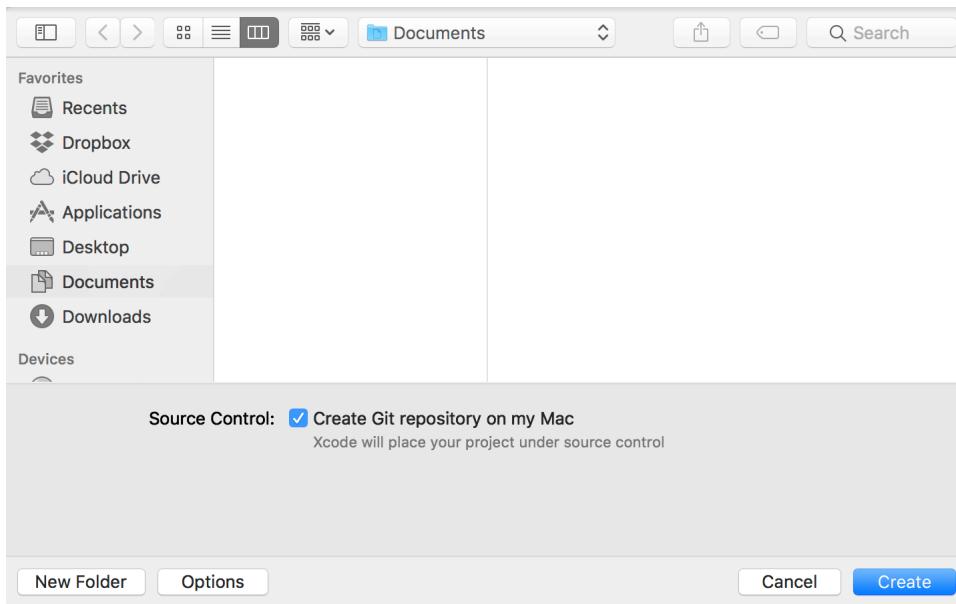
一開始，先選專案類型，這裡選「macOS」的「Command-Line Tool」：



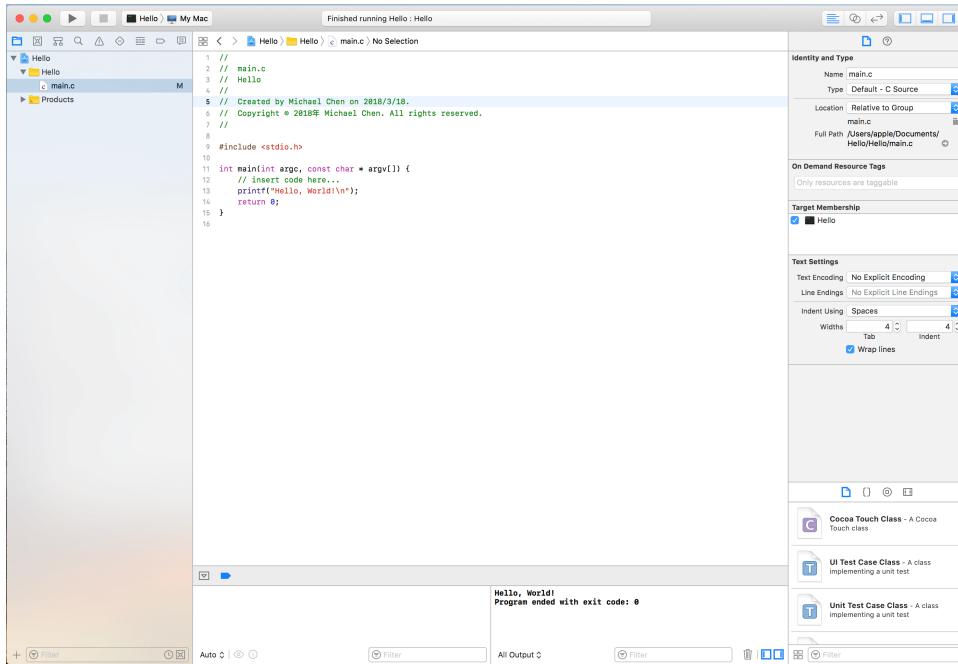
設置專案的名稱：



設置專案存放的位置，這裡選 *Documents* (可選其他位置)：



進入 Xcode 的編輯器，可以開始撰寫程式碼：



寫第一個程式

我們這裡展示 Hello World 程式，暫時不要管程式碼的意義，這裡的重點是確保程式可順利運行：

```
#include <stdio.h>

int main()
{
    printf("Hello World\n");

    return 0;
}
```

如果使用 IDE 的讀者，選取執行 (Run) 或等效的指令即可執行。如果使用編輯器的讀者，可參考以下終端機指令：

```
$ gcc -o hello hello.c
$ ./hello
Hello World
```

註：Mac 中的 GCC 預設是指向 Clang 的連結，而非 GNU 的 GCC。

如果程式順利執行，代表環境建置成功。若執行失敗，則需根據錯誤訊息來處理。

對於初期的練習，使用上述方法應該足夠。但若讀者在終端機環境中編譯 C 程式碼，久了應該會覺得反覆打指令很費時。建議使用 Make 等自動編譯軟體來減輕我們的工作量。

在 GNU/LINUX 上建立開發環境

前言

GNU/Linux 承襲 Unix 的文化，對於 C 語言支援相當良好。除了剛開始要花一些時間學習如何使用系統外，GNU/Linux 是相當適合用來學習程式設計的平台，當然也包括 C 語言在內。

C 編譯器

GNU/Linux 預設的 C 編譯器是 GCC，但是也可以用其他的 C 編譯器，像是 Clang。有些學習者會刻意用 Clang 來學 C 語言，因為 Clang 吐出來的錯誤訊息比較友善。

編輯器或 IDE

使用 GNU/Linux 時，會考量命令列環境和桌面環境等不同情境。因為有些 GNU/Linux 用於伺服器，這時候系統上不會有桌面環境，得用命令列環境的編輯器。

適用於命令列環境的編輯器

以下是命令列環境中常見的編輯器：

- Vi (或 Vim)
- Emacs
- Nano
- JOE (Joe's Own Editor)

在這些編輯器中，至少要會基本的 Vi (或 Vim)，因為幾乎所有的 GNU/Linux 主機都會裝 Vi (或 Vim)。但 Vi 家族編輯器的操作比較複雜，對於簡單的編輯任務，可以用 Nano 就好。由於 GNU/Linux 系統上預裝 Emacs 或 JOE 的機會比較少，學習順序可以擺後面一點。

適用於桌面環境的編輯器或 IDE

如果有桌面環境可用，倒不需要堅持使用命令列環境的編輯器。因為 IDE 有編輯器沒有的好處，像是重構、自動補完、程式碼瀏覽等整合性的功能等。為了要在編輯器中模擬 IDE 的功能，我們得安裝不少外掛或是用複雜的設定。但在 IDE 中這些都是立即可用的功能。

以下是桌面環境中常見的編輯器或 IDE：

- Geany
- Anjuta
- KDevelop
- CLions (商業軟體)
- Visual Studio Code (VSCode)

筆者比較推薦 KDevelop、CLions、VSCode 等開發工具，因為這些軟體本身是跨平台的，在不同系統上可取得一致的操作邏輯。

雲端 IDE

雲端 IDE 是指網頁 IDE + GNU/Linux 虛擬機器的組合，由於雲端運算在近年來相當流行，雲端 IDE 成為一個便利的選項。最知名的雲端 IDE 就是被 Amazon 收購的 Cloud 9。當然也有一些其他的方案，讀者可自行上網搜尋比較。

管理 C 專案

C 語言本身沒有專案的概念，需用第三方工具來管理 C 專案。以下是常見的選項：

- Make
- Autotools
- CMake

傳統上的專案管理工具是 Make，但要在不同 Unix 或類 Unix 系統間寫出通用的 Makefile 相對困難，因而出現 Autotools。Autotools 的工作原理是自動偵測當下環境並產生相對應的 Makefile。但 Autotools 本身是以 shell 程式寫成，僅能用在 Unix 或類 Unix 系統上。

相對來說，CMake 也可以自動生成 Makefile。除此之外，CMake 還可自動生成其他平台的專案管理組態，在泛用性上比 Autotools 好。所以 CMake 在近年來相當流行。

安裝第三方函式庫

我們不會只用標準函式庫來寫程式，而會在必要時使用合適的第三方函式庫。C 語言本身有既定的函式庫格式，但沒有套件的概念。現行的解決方式，是直接用 GNU/Linux 系統上的套件管理軟體來安裝 C 函式庫。

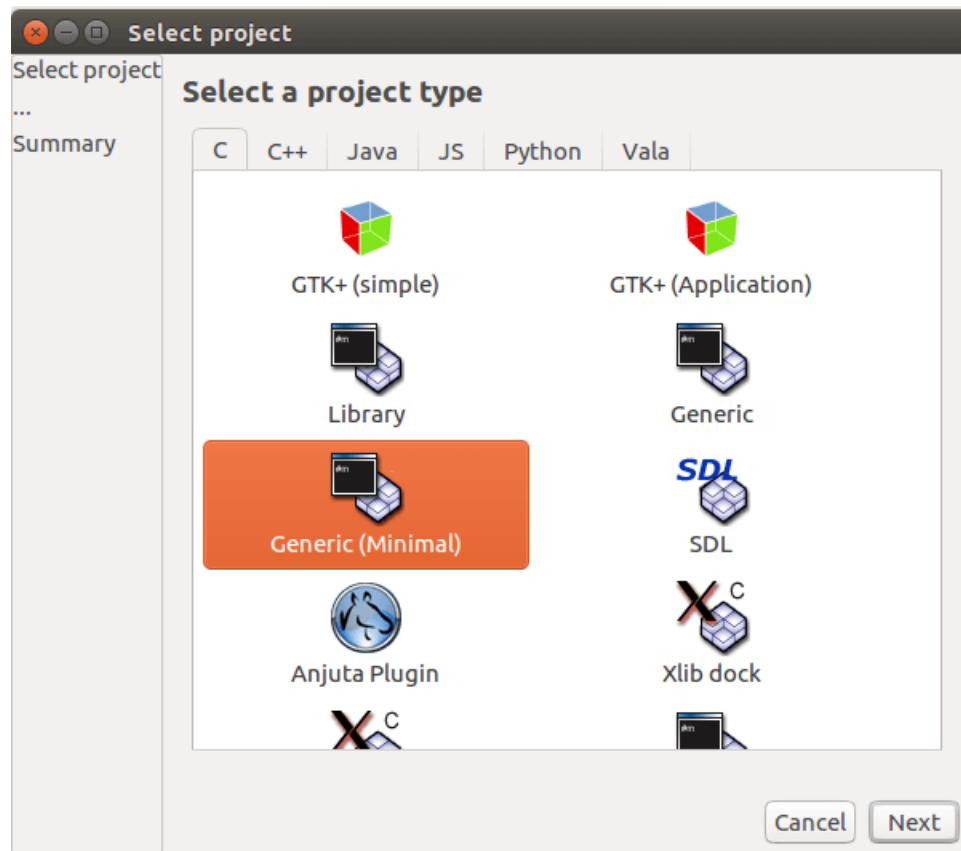
一般情形下，使用系統的套件管理軟體來安裝 C 函式庫比較方便。但套件是預編好的，套件維護者會以適用大部分使用者需求的考量來編譯。如果需要特殊的編譯選項，就得自己抓原始碼來編譯。

此外，套件的 C 函式庫版本可能不是最新的，如果需要較新的版本，還是要自行抓原始碼來編譯。此外，有些較小眾的 C 函式庫，不會有現成的套件可用，這時候還是得使用上游原始碼。

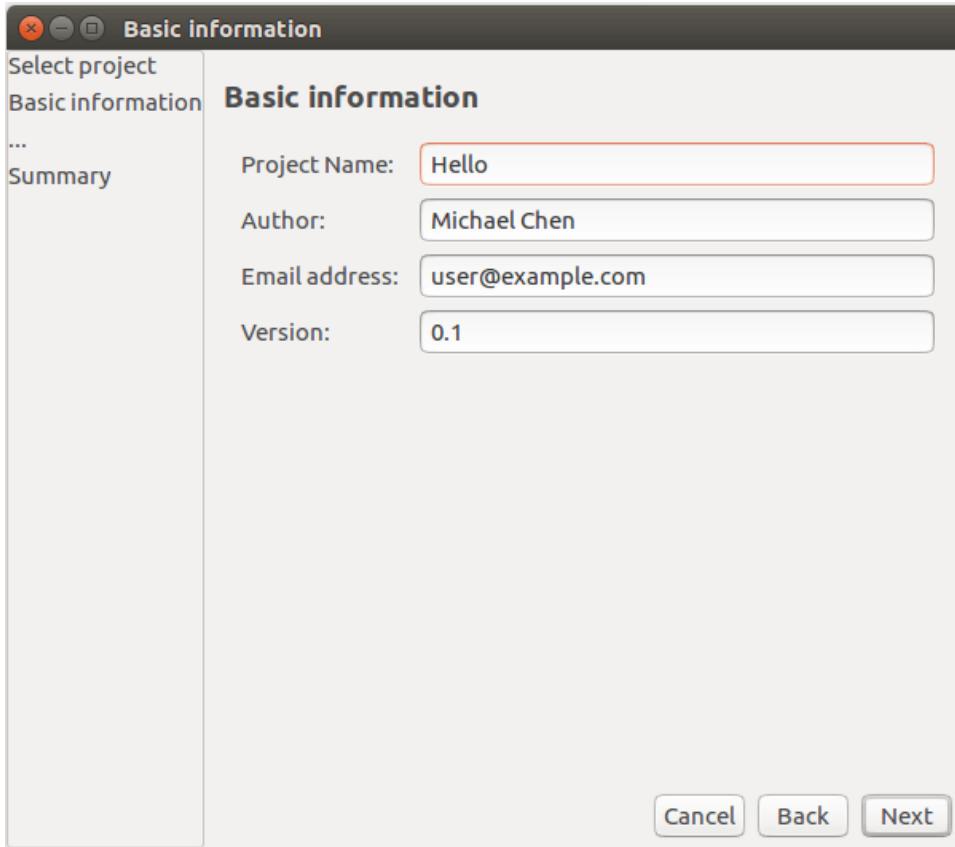
動手做時間：用 Anjuta 建立 C 專案

對於初心者來說，使用 IDE 可以省下撰寫專案設定檔 (Makefile 等) 的心力，將心思專注在學習語法上。使用 Code::Blocks 等跨平台 IDE 可以省下重學新的 IDE 的心力。我們這裡以 Anjuta 為例，說明如何用 IDE 寫 C 程式。

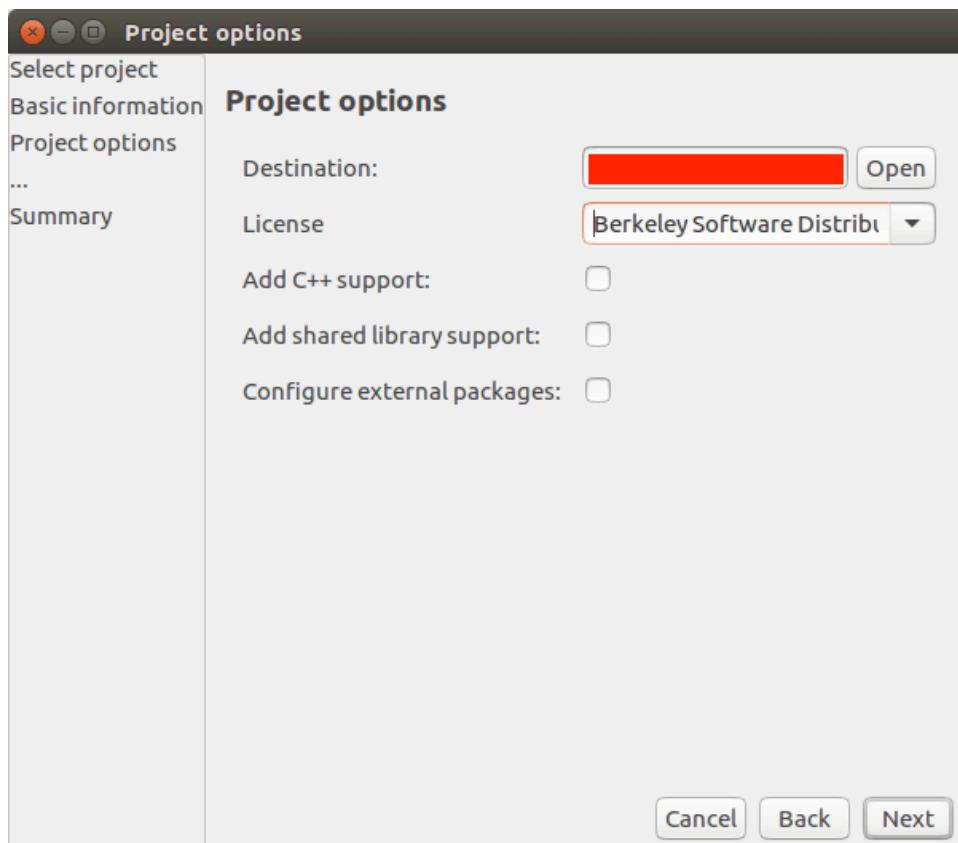
在語言類別中選 C 語言，選擇「Generic (Minimal)」：



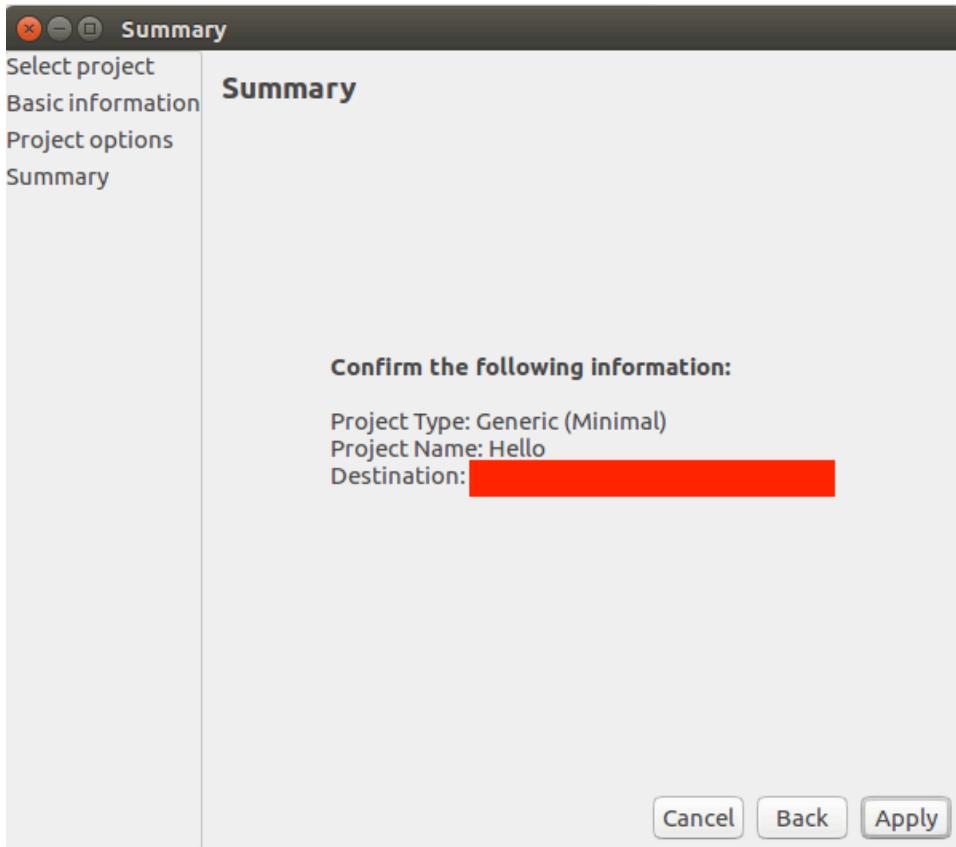
設定專案相關資訊：



設定專案位置、授權和其他資訊：



確認專案設定正確與否：



開始編輯 C 程式碼：

The screenshot shows the main interface of the Anjuta IDE. The title bar says 'main.c (~/Documents/hello) - hello - Anjuta'. The left sidebar shows a project tree with a single file 'main.c' selected. The main editor window displays the content of 'main.c':

```
/* -*- Mode: C; indent-tabs-mode: t; c-basic-offset: 4; tab-width: 4 -*- */
/*
 * main.c
 * Copyright (C) 2018 Michael Chen <user@example.com>
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. Neither the name "Michael Chen" nor the name of any other
 *    contributor may be used to endorse or promote products derived
 *    from this software without specific prior written permission.
 *
 * Hello IS PROVIDED BY Michael Chen "AS IS" AND ANY EXPRESS
 * OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
 * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL Michael Chen OR ANY OTHER CONTRIBUTORS
 * BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
 * BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
 * OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
 * ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */
#include <stdio.h>
int main()
{
    printf("Hello world\n");
    return 0;
}
```

At the bottom, there are tabs for 'Files', 'Project', and 'Symbols', and status information: 'Line: 0001 Mode: INS Col: 000 Project: hello'.

撰寫第一個程式

我們這裡展示 Hello World 程式，暫時不要管程式碼的意義，這裡的重點是確保程式可順利運行：

```
#include <stdio.h>

int main()
{
    printf("Hello World\n");

    return 0;
}
```

如果使用 IDE 的讀者，選取執行 (Run) 或等效的指令即可執行。如果使用編輯器的讀者，可參考以下終端機指令：

```
$ gcc -o hello hello.c
$ ./hello
Hello World
```

如果程式順利執行，代表環境建置成功。若執行失敗，則需根據錯誤訊息來處理。

對於初期的練習，使用上述方法應該足夠。但若讀者在終端機環境中編譯 C 程式碼，久了應該會覺得反覆打指令很費時。建議使用 Make 等自動編譯軟體來減輕我們的工作量。

在命令列使用 GCC 或 CLANG

前言

本文會選 GCC 而非其他 C 編譯器是因為 GCC 在 GNU/Linux 等類 Unix 系統上具有代表性。如果讀者使用 Clang，因 Clang 參數刻意相容於 GCC，仍然可以參考本文來學習 Clang；而且 Clang 的錯誤訊息比 GCC 友善，倒也不失為一個學習 C 語言的替代工具。

基本的使用方式

假定 C 程式碼為 *source.c*，編譯出的執行檔為 *program*。先記住以下基本指令即可：

```
$ gcc -o program source.c
```

本文的目的是整理一些常見的 GCC 或 Clang 的使用方式，初學者覺得難以吸收或用不到的話可以先跳過沒關係。

檢查編譯器的版本

使用 `--version` 參數即可。可參考以下指令：

```
$ gcc --version
gcc (Ubuntu 4.8.4-2ubuntu1~14.04.4) 4.8.4
Copyright (C) 2013 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There
is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.
```

從編譯器的版本可得知該編譯器所有的特性和臭蟲。在線上討論區討論問題時，可以貼上自己所用的 C 編譯器的版本，有時會從中得到一些有用的回應。

開啟警告訊息

除了編譯程式碼外，編譯器也會做一些靜態程式碼檢查。使用 `-Wall` 參數會開啟許多有用的警告，一開始練習時建議開啟，可以從錯誤訊息中學習。除此之外，還可以開啟 `-Wextra`，會出現更多的錯誤訊息。參考以下指令：

```
$ gcc -Wall -Wextra -g -o program source.c
```

`-Werror` 會將警告訊息轉為錯誤，初期對練習寫 C 程式會有一些幫助。不過，GCC 有些警告訊息其實不會造成實質的影響，筆者會去看 GCC 的警告訊息，但不會開啟這個選項。

`-pedantic` 對非標準 C 的語法會出現錯誤訊息，若注重程式碼的相容性可開啟此選項。

開啟除錯相關資訊

加入 `-g` 可以在編譯出來的程式中加上除錯相關的資訊，如果要對編譯出來的執行檔使用 GDB 等除錯器 (debugger) 除錯時就需要在編譯時加入此參數。

由於這個參數對程式運行本身沒有幫助，但是會讓程式的體積變大，最後要發布實際要上線的程式時，建議關掉這項參數重新編譯一次。

開啟剖析 (Profiling) 相關資訊

加入 `-pg` 可在編譯時加入 *gprof* 程式可用的訊息，可和 `-g` 併用。

由於在程式中加入剖析相關資訊會影響程式運行效能，程式要正式上線時，要關掉這個參數後重新編譯一次。

選擇編譯最佳化策略

GCC 有許多和最佳化相關的參數，這些參數相當複雜。為了簡化最佳化的過程，GCC 提供一些預先配置好的「套餐」。常見的選項如下：

- `-O0` (關閉最佳化)
- `-O1`
- `-O2`
- `-O3`
- `-Os` (空間最佳化)

`-O2` 是保守但可用的最佳化策略，要效能可試著用 `-O3` 來編譯程式。但 `-O3` 有時會優化過頭，導致程式發生預期外的錯誤。除非是效能魔人，不建議用 `-O3` 來編譯程式。

除此之外，還有一些細部的選項可以把玩。在學習 C 語言的時候不需要耗費時間在這裡，完全不優化程式也沒關係。

編譯多個檔案

初學時會把所有的程式碼寫在同一個檔案中，但實務上的程式會拆成多個檔案。可參考以下指令：

```
$ gcc -Wall -g -o program main.c lib_a.c lib_b.c lib_c.c
```

相關的標頭檔 (header) 要預先撰寫，否則 GCC 無法自動串連多個檔案。於後文會再說明。

指定 C 標準 (C Standard) 的版本

參考以下指令：

```
$ gcc -Wall -g -std=c11 -o program source.c
```

GCC 中常見的 C 語言標準：

- c89 或 c90 或 `-ansi`
- c99
- c11
- c17 或 c18

除此之外，還可以加上 GNU 特有的 extension：

- gnu89 或 gnu90：c89 加上 GNU C extension
- gnu99：c99 加上 GNU C extension
- gnu11：c11 加上 GNU C extension
- gnu17：c17 加上 GNU C extension

一開始建議使用 c90，只有在需要新的特性時才改用新的 C 標準，以維持相容性。GNU C extension 不是 C 標準的語法，除非很確定該專案只會用到 GCC 來編譯，不建議任意地使用。

加入額外的函式庫

除了少數內建的函式庫以外，編譯時要加入相關的參數。常見的例子有

- `-lm`：連結 *math.h*
- `-lstdc++`：混合編譯 C 和 C++ 程式碼時需連結的函式庫
- `-lpthread`：連結 GNU/Linux 的多執行緒函式庫
- `-lrt`：連結 POSIX 運行期函式庫
- `-ldl`：連結動態函式庫

參考以下指令：

```
$ gcc -Wall -g -o program source.c -lm
```

以 `-lm` 來說，其讀法為 `-l` (函式庫) 加上 `m` (數學函式庫)，其他函式庫同理可知。絕大部分連結函式庫的參數名稱都很規律，不要死背這些參數。

加入額外的標頭檔或二進位檔位置

有些函式庫不是位於系統內建的位置上，則要另外加上 `-I` (記成 include) 和 `-L` (記成 library)。參考以下指令：

```
$ gcc -o program source.c -I/path/to/include -L/path/to/lib -lsomething
```

`pkg-config` 是一個用來簡化編譯第三方函式庫的小工具，透過這套工具，我們不用手寫 `-I` 和 `-L` 等參數。

例如，我們以 `pkg-config` 自動產生適用於 *libpng* 的參數 (於 Mac 上測試)：

```
$ pkg-config --libs --cflags libpng  
-I/usr/local/Cellar/libpng/1.6.34/include/libpng16 -L/usr/local/Cellar/  
libpng/1.6.34/lib -lpng16 -lz
```

編譯時將此段參數插入指令之中即可：

```
$ gcc -o program source.c `pkg-config --libs --cflags libpng`
```

編譯函式庫

函式庫是 C (或 C++) 分享程式的方式，分為靜態連結 (static linking) 和動態連結 (dynamic linking) 兩種。前者會將程式直接編入主程式中，後者在執行期時才動態連結。

編譯靜態連結函式庫可參考以下指令：

```
$ gcc -c -o a.o a.c
...
$ ar rcs libsomething.a a.o ...
```

在 macOS 上時則改用 libtool 來編譯靜態函式庫：

```
$ libtool -static -o libsomething.a a.o ...
```

編譯動態連結函式庫可參考以下指令：

```
$ gcc -fPIC -c -o a.o a.c
...
$ gcc -shared -o libsomething.so a.o ...
```

在 macOS 上最好按照 macOS 的慣例，將動態連結函式的副檔名改為 `.dylib`。

小結

本文所述大概僅佔 GCC 參數的一小部分，但一些冷門的參數其實也很少用，需要時再去查詢即可。如果每次都要手動輸入編譯指令其實蠻辛苦的，建議使用 GNU Make，等自動編譯軟體來減少手動輸入指令的負擔。

在命令列使用 VISUAL C++

前言

Visual Studio 內部的 C 編譯器為 `cl.exe`，微軟網站有 `cl.exe` 參數等相關資料 (像是這份文件⁴⁰)。

在大部分情形下，程式設計者會透過 Visual Studio 間接使用此編譯器，甚少直接從命令列呼叫該編譯器。不過，我們有時會從終端機呼叫 `cl.exe`，像是要撰寫跨平台的 Makefile 時，就會用到 `cl.exe` 的命令列參數。此外，熟悉 `cl.exe` 的參數後，也可在 Visual Studio 中調整相關參數。

基本的使用方式

最簡單的使用方式如下：

```
> cl hello.c  
> .\hello.exe  
Hello World
```

本文的目的是整理一些常見的使用情境，一開始沒用到的話也不用死背這些指令。

檢查編譯器版本

單獨輸入 `cl` 指令可檢視該編譯器的版本：

```
> cl  
Microsoft (R) C/C++ Optimizing Compiler Version 19.13.26131.1 for x64  
Copyright (C) Microsoft Corporation. All rights reserved.  
  
usage: cl [ option... ] filename... [ /link linkoption... ]
```

如果在程式討論區，提供編譯器的版本偶爾會得到一些有用的回應。比較簡單的替代方法是提供 Visual Studio 的版本。

⁴⁰ <https://docs.microsoft.com/en-us/cpp/build/reference/compiler-command-line-syntax>

開啟警告訊息

理論上，使用 `/Wall` 參數可以開啟所有警告訊息，對修改程式有一些幫助；但 `/Wall` 會有過多的偽陽性，在實務上其實不好用，比較好用的參數是 `/W4` (參考[這個討論串⁴¹](#))。

使用 `/sdl` 參數可再多開啟一些和安全性相關的警告訊息。

使用範例如下：

```
> cl /W4 /sdl /Fe:program.exe source.c
```

開啟除錯訊息

使用 `/Zi` 參數可在編譯程式時額外產生一些除錯相關的訊息。

選擇最佳化策略

Visual C++ 常見的最佳化參數套餐如下：

- `/Od`：關閉最佳化，為預設情境
- `/O1`：最省空間的最佳化
- `/O2`：最佳速度的最佳化
- `/Os`：偏向節省空間的最佳化
- `/Ot`：偏向改善速度的最佳化

除此之外，還有一些細節選項可調。一開始時不用耗費過多時間在調最佳化參數上面，在最後要發布程式時再來費心選擇即可。

指定 C 標準 (C Standard) 的版本

Visual C++ 可以指定以下 C 標準：

- `/std:c11`
- `/std:c17`

⁴¹ <https://stackoverflow.com/questions/45579176/how-to-use-the-highest-warning-level-wall-for-visual-studio-2017-when-its-inc>

不使用參數時，則使用原本的 MSVC 所支援的 C 標準，相當於 C99 和部分 C11。

Visual C++ 預設會開啟 Microsoft 延伸特性。如果想要在 ANSI C 模式下編譯 C 程式碼，可以加上 /za 參數。但撰寫 Win32 API 程式時，基本上無法使用 /za 參數，因為 Win32 API 本身就不遵守 ANSI C 標準。

連結外部檔案

使用 /I 參數可在編譯時加入外部路徑，如下例：

```
> for %x in (*.c) do cl /c %x
> cl /Fe:program.exe *.obj /I\include
```

編譯函式庫

編譯靜態函式庫可參考以下指令：

```
> for %x in (*.c) do cl /c %x
> lib /out:something.lib *.obj
```

Visual C++ 的靜態函式庫的副檔名是 .lib ，而非 .a ，而且檔案名稱沒有 lib 前綴。

編譯動態函式庫可參考以下指令：

```
> for %x in (*.c) do cl /c %x
> link /DLL /OUT:something.dll *.obj
```

Visual++ 的動態函式庫的副檔名是 .dll 而非 .so ，而且檔案名稱沒有 lib 前綴。除了 .dll 檔外，還有搭配的 .lib 檔和 .exp (export) 檔等。

用 INTEL C++ COMPILER 編譯 C 程式

前言

除了桌面系統預設的 C 或 C++ 編譯器外，也有一些針對特定硬體而設計的 C 或 C++ 編譯器，像是 Intel C++ Compiler、CUDA C、Arduino C 等。比起通用型 C 或 C++ 編譯器，針對特定硬體的 C 或 C++ 編譯器會針對特定硬體去優化，可以享受特定硬體所帶來的益處。

Intel C++ Compiler 是針對 Intel 平台而設計的 C 或 C++ 編譯器。由於 Intel 平台相當普遍，所以這個編譯器值得注意。

Intel C++ Compiler 的系統需求

Intel C++ Compiler 的目標硬體為基於 Intel 處理器的個人電腦。雖然使用其他 Intel 相容處理器 (AMD、VIA 等) 的個人電腦也可以裝 Intel C++ Compiler，但在非 Intel 平台上會吃不到 Intel C++ Compiler 的優化，故不建議使用。

支援的系統會區分為宿主系統 (host system) 和目標系統 (target system) 兩種。宿主系統是安裝 Intel C++ Compiler 時所用的系統，而目標系統是編譯出來的執行檔可用的系統。當兩者相同時，就是一般的編譯，而兩者相異時為交叉編譯。

以下是 Intel C++ Compiler 所支援的宿主系統：

- Windows
- GNU/Linux
- macOS

基本上就是主流的桌面系統。在 Windows 上可搭配 Visual Studio 2017 或 2019 來使用。

以下是 Intel C++ Compiler 所支援的目標系統：

- Windows
- GNU/Linux
- Android

要注意目標系統沒有 macOS。

取得 Intel C++ Compiler

Intel C++ Compiler 會隨附在 Intel Parallel Studio 或 Intel System Studio 中。兩者皆是商業軟體，後者可長期免費使用。Intel System Studio 的使用許可需每年更新一次，目前可無限更新使用許可，但不知道 Intel 日後會不會回收這項福利。

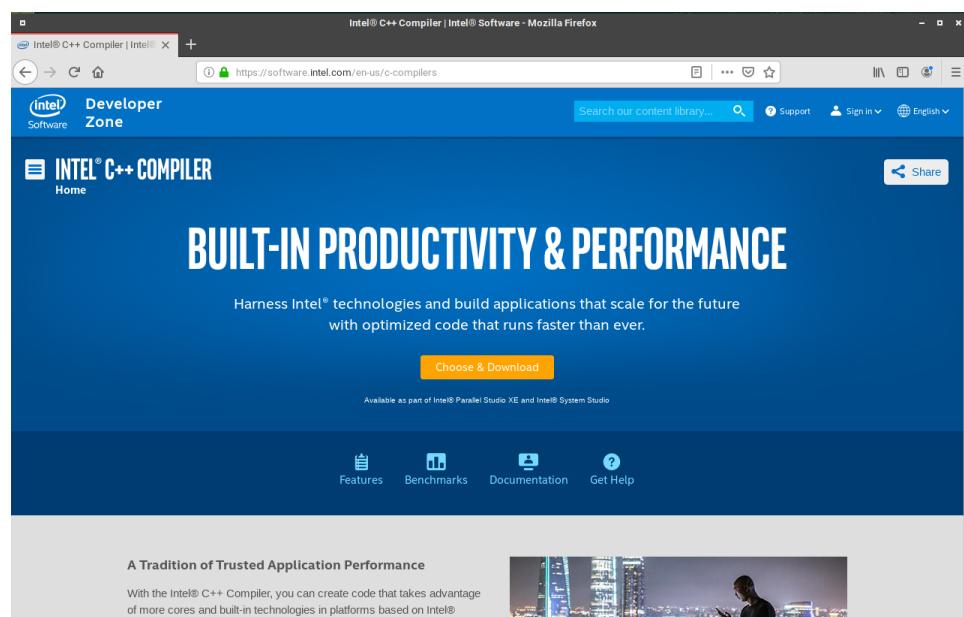
如果只是要用 Intel C++ Compiler 的話，先用免費的 Intel System Studio 即可。確認 Intel System Studio 無法滿足自己的需求後，再買 Intel Parallel Studio 也不遲。

安裝 Intel System Studio

本筆展示安裝 Intel System Studio 的過程。

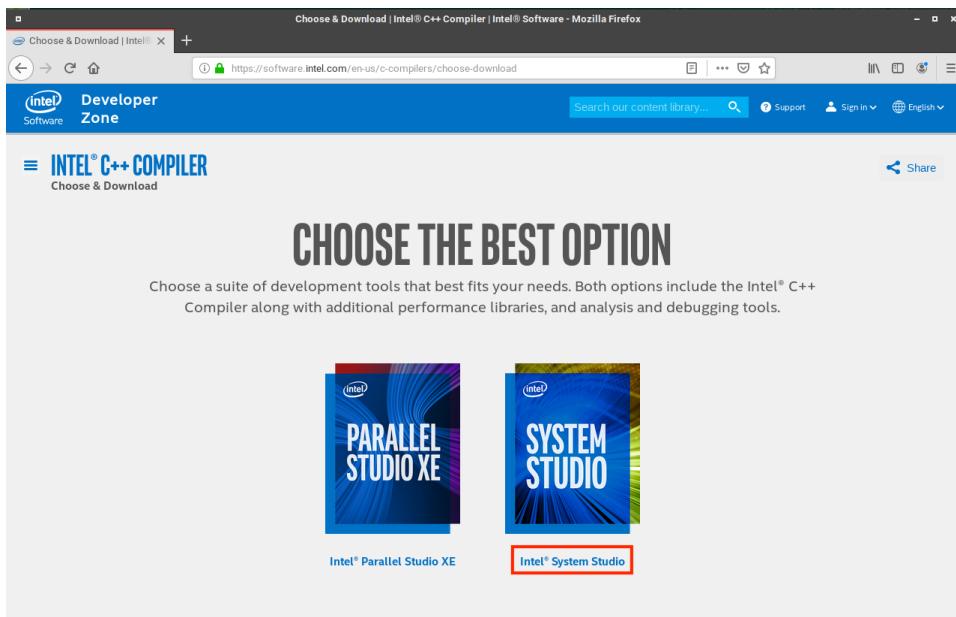
註：筆者的 Windows 主機因顯卡老舊，畫面有殘影，故改用 GNU/Linux 虛擬主機做截圖。

到 Intel C++ Compiler 的專頁⁴²下載 Intel C++ Compiler：

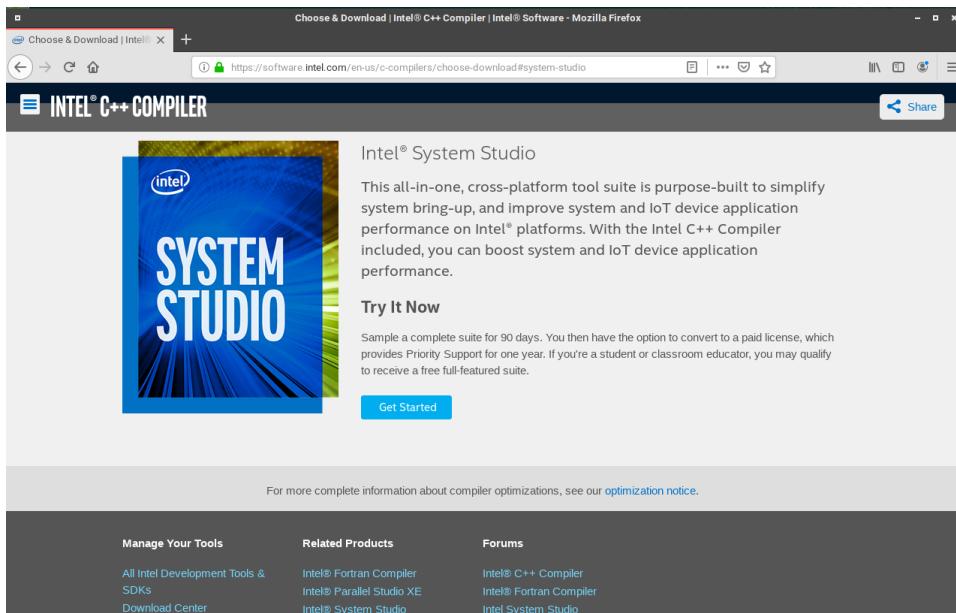


可以在 Intel Parallel Studio 和 Intel System Studio 中擇一。我們的目的只是要用 C 編譯器，故下載 Intel System Studio 即可：

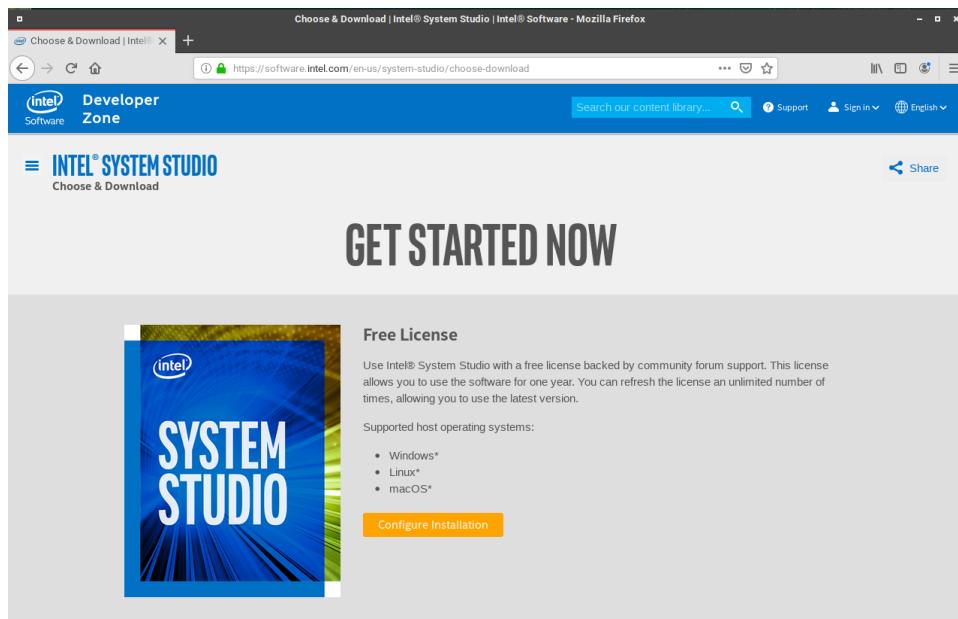
⁴² <https://software.intel.com/en-us/c-compilers>



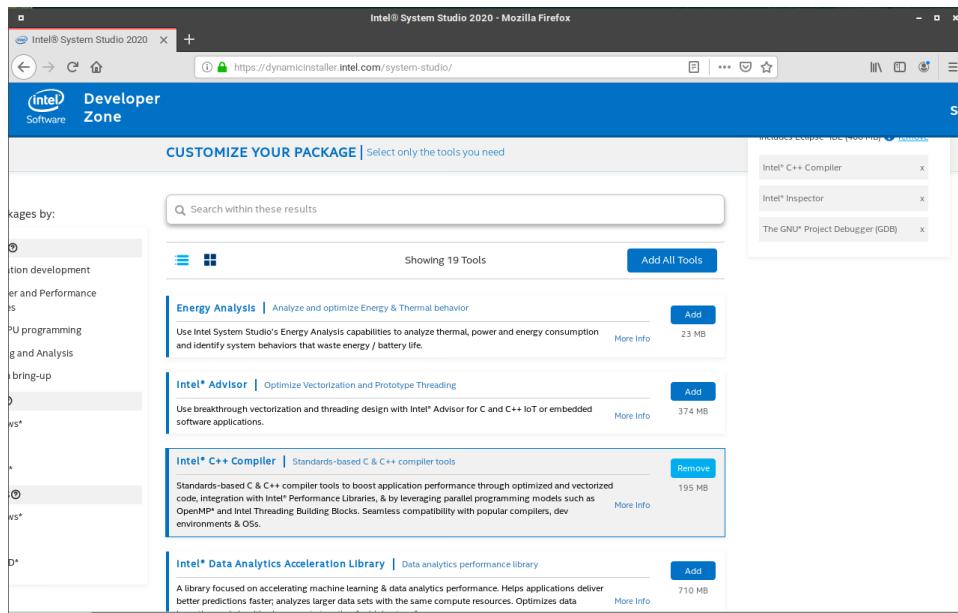
這裡只是簡單的說明頁。繼續按下一步即可：



這裡特別說到 Intel System Studio 可以免費用，只要每年更新使用許可就可以繼續使用。按下一步開始選擇所要安裝的軟體：



在安裝 Intel System Studio 時，不是一口氣就安裝所有的軟體，而是像點菜般選擇自己所需的部分：



我們只選擇了 Intel C++ Compiler、Intel Inspector、GDB：

Selected Tools

Continue to Download

[Clear All Tools](#)

Includes Eclipse* IDE (400 MB) [remove](#)

- Intel® C++ Compiler [x](#)
- Intel® Inspector [x](#)
- The GNU* Project Debugger (GDB) [x](#)

讀者不一定要和我們選一樣的軟體，可以隨自己需求更動軟體清單。
註冊不是強制的，所以我們不註冊，直接下載軟體：

Get Your Download

Log in to your **My Intel** account.

Don't have an account? Create one now—quick and easy.

When you do, here's what you'll get:

- Fast access to Intel software product downloads
- Ability to engage in discussion forums with experts and colleagues
- Timely notification of the latest software tools, technologies, and industry trends
- Advance invitations to technical webinars, training, and local events
- Opportunities to help Intel make its next-gen tools better, including beta testing pre-release products

[Log In >](#)

using My Intel web account

[Sign Up >](#)

for a My Intel web account

Maybe next time. Please take me to my download.

根據自己的宿主系統和目標系統來選擇即可：

The screenshot shows the Intel System Studio 2020 download page. At the top, there's a navigation bar with tabs for 'Developer Zone' and 'Sign In'. Below it, a section titled 'YOUR TOOLS ARE READY FOR DOWNLOAD' lists several tools with their compatibility across different host and target platforms:

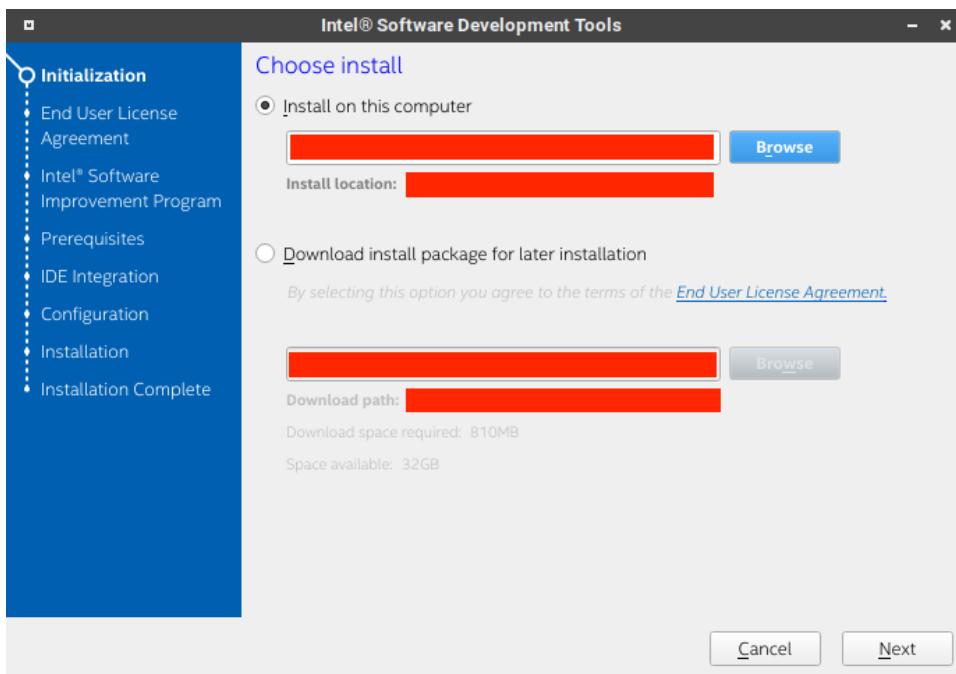
	Windows® Host Windows® Target (Plugin for Visual Studio®)	Windows® Host Linux® Target (Eclipse-based tools)	Linux® Host Linux® Target	macOS® Host Linux® Target
Object Debugger (GDB)	—	✓	✓	—
or	✓	✓	✓	—
mpiler <small>Threading Building Blocks as a dependency.</small>	✓	✓	✓	—

Below the table, there are four orange 'Download' buttons, each with a small icon and a link labeled 'Standalone Offline'. The entire interface is designed for cross-platform development.

由於筆者是在 GNU/Linux 上展示安裝過程，故下載到 tarball。若是在 Windows 上，則會下載到安裝程式：

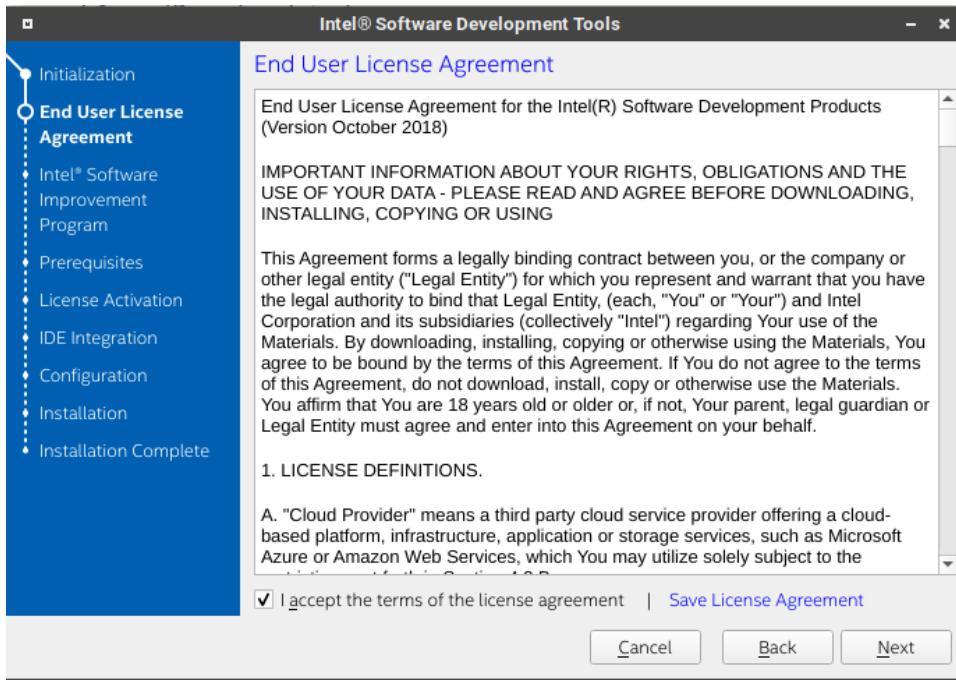
A Firefox dialog box titled 'Opening intel-sw-tools-installation-bundle-linux-linux.zip' is shown. It asks, 'You have chosen to open:' followed by the file name 'intel-sw-tools-installation-bundle-linux-linux.zip'. Below that, it says 'which is: Zip archive (18.7 MB)' and 'from: blob:'. The main question is 'What should Firefox do with this file?'. There are two radio button options: 'Open with Engrampa Archive Manager (default)' and 'Save File'. The 'Save File' option is selected and highlighted with a red border. Below the radio buttons is a checkbox 'Do this automatically for files like this from now on.' At the bottom are 'Cancel' and 'OK' buttons.

接著，啟動安裝程式。一開始先選擇安裝位置：

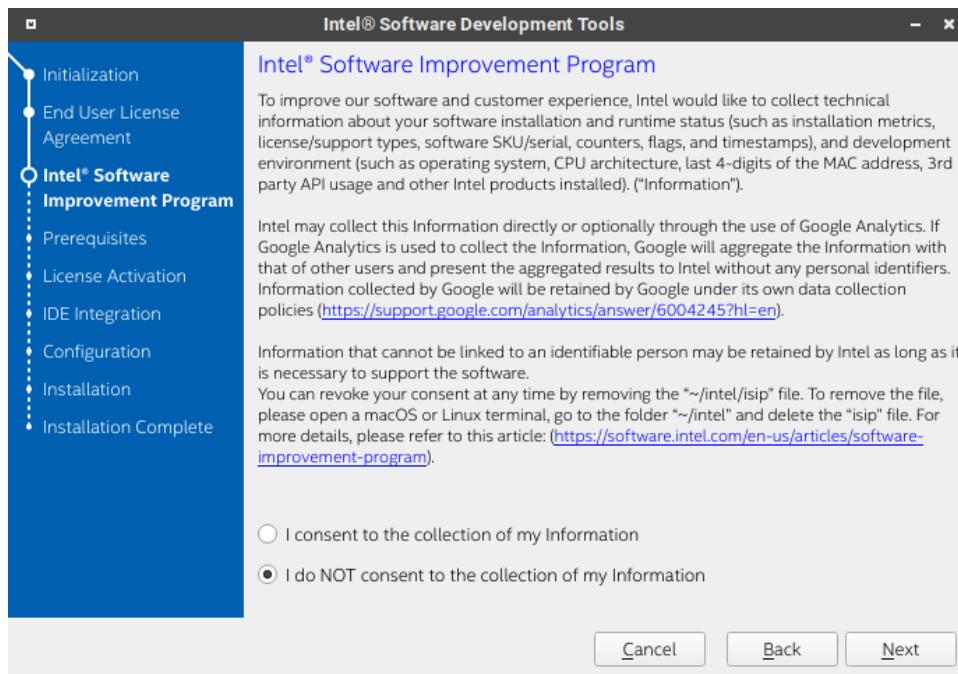


在 GNU/Linux 安裝 Intel System Studio 時，預設會安裝到 `$HOME/intel` 上。

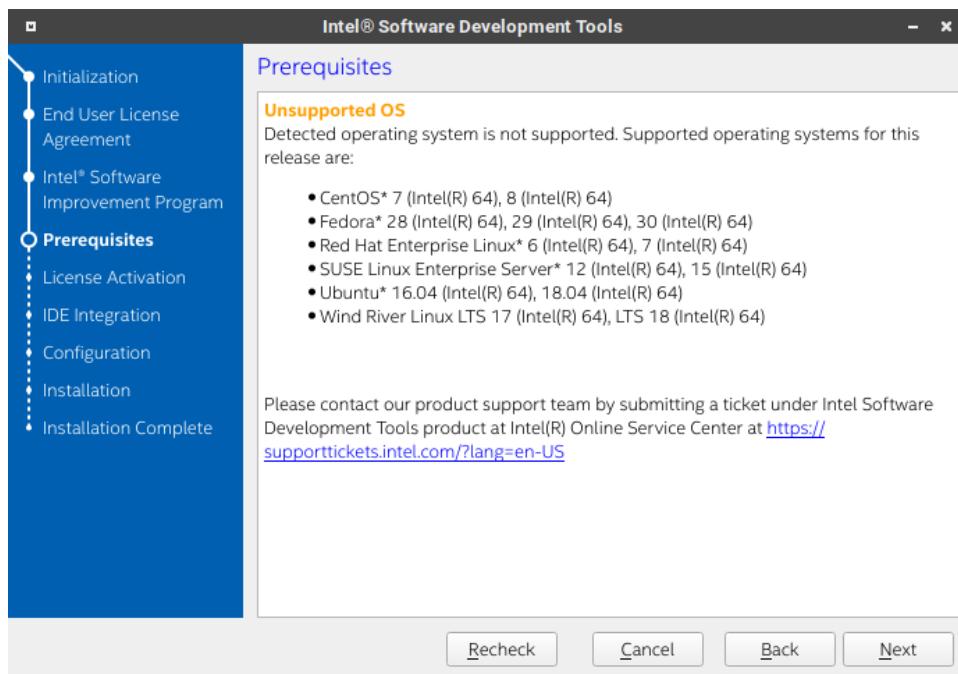
按照慣例，會有落落長的軟體授權文件。有興趣的讀者可以自己看一下：



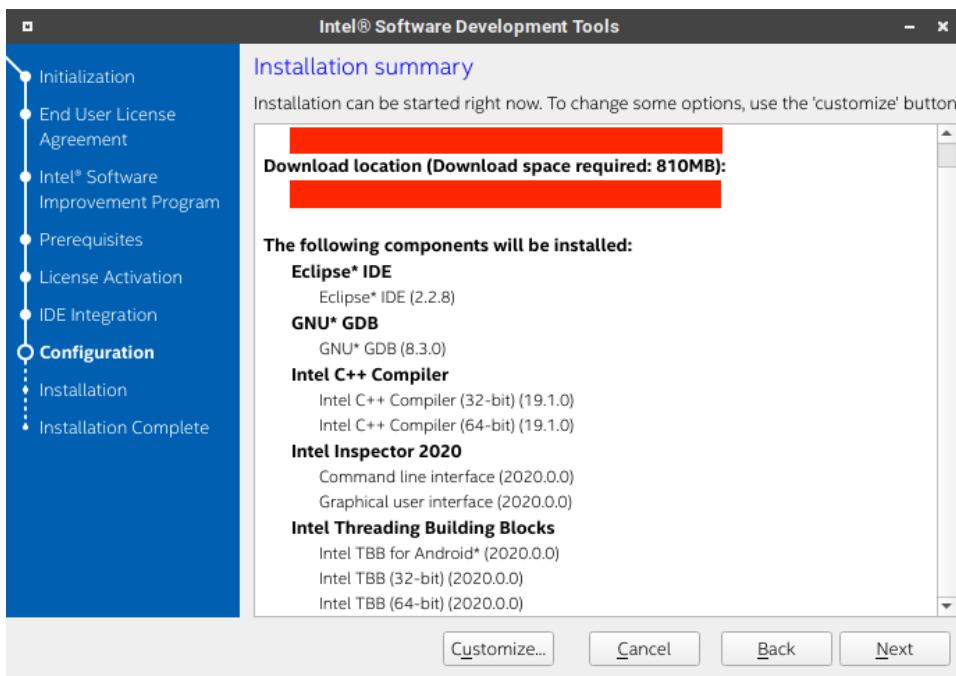
Intel System Studio 會收集一些匿名資料，讀者可自行決定要不要提供這些資料：



筆者在測試時使用 openSUSE Leap 15.1。雖然 openSUSE Leap 並非 Intel System Studio 官方支援的系統。實際上仍然可以使用：



安裝精靈會展示出安裝的軟體：



安裝完後即可結束安裝精靈。

我們實際上會從命令列使用 Intel C++ Compiler，至於 IDE 則不是必備的。

在命令列使用 Intel C++ Compiler 編譯 C 或 C++ 程式

Intel C++ Compiler 在使用前要先初始化環境，這點很像 Visual C++。在 Windows 上，初始化的 Batch 命令稿為 *compilervars.bat*。以筆者的電腦為例，完整的指令如下：

```
> "C:\Program Files (x86)\IntelsWTools\sw_dev_tools\compilers_and_
libraries\windows\bin\compilervars.bat" intel64
```

由於讀者所安裝的軟體可能相異，請不要死背這條指令，自己試著在系統上找一下 *compilervars.bat* 的位置。

在 Unix 系統上，初始化的 shell 命令稿為 *compilervars.sh* 或 *compilarvars.csh*，視 shell 環境而定。以筆者的電腦為例，完整的指令如下：

```
$ . $HOME/intel/sw_dev_tools/bin/compilervars.sh intel64
```

同樣地，請不要死背指令。

初始化環境後，就可以在命令列環境使用 Intel C++ Compiler。

這個編譯器比較特別，在 Windows 上的指令為 *icl*，但在 Unix 上的指令為 *icc*。此外，在 Windows 上使用 DOS 風格的參數，且參數相容於 Visual C++；但在 Unix 上使用 GNU

風格的參數，參數相容於 GCC。Intel C++ Compiler 會故意設計成這樣，是為了讓不同平台的使用者較快轉移編譯器。

註：Visual C++ 的命令列程式為 *cl.exe*，而 Windows 版本的 Intel C++ Compiler 的命令列程式為 *icl.exe*。

例如，在 Unix 上將 CC 參數設為 *icc*，就可以用 Intel C++ Compiler 取代 GCC：

```
$ make CC=icc
```

當然，*Makefile* 要預先寫好。本文重點不在於寫 *Makefile*，故不詳細說明。

在命令列使用 Intel Inspector 檢查 C 或 C++ 程式

除了使用 Intel C++ Compiler 編譯 C 或 C++ 之外，還可以用 Intel Inspector 檢查 C 或 C++ 程式。這套軟體可以用來偵測記憶體或執行緒問題。

使用 Intel Inspector 前也是要初始化環境。在 Windows 上，用來初始化的 Batch 命令稿為 *inspxe-vars.bat*。以下是筆者在電腦上執行的指令：

```
> "C:\Program Files (x86)\IntelSWTools\sw_dev_tools\Inspector\inspxe-
→vars.bat"
```

Unix 的初始化命令稿為 *inspxe-vars.sh* 或 *inspxe-vars.csh*，視 shell 環境而定。以下是筆者在 Unix 系統上執行的指令：

```
$ . $HOME/intel/sw_dev_tools/inspector/inspxe-vars.sh
```

初始化環境後，就可以在命令列使用 Intel Inspector。

Intel Inspector 使用時要分兩步驟操作。第一步先從目標程式收集資料，第二步會根據收到的資料產生報告。

假定我們的目標程式為 *program*。以下指令會偵測是否有記憶體洩露：

```
$ inspxe-cl -collect mi1 -- ./program
```

inspxe-cl 可指定 *-collect* 或 *-report* 兩種動作 (action) 之一，但不能同時指定。*-collect* 可收集的對象有六種⁴³，其中三種和記憶體相關，三種和執行緒相關。

在命令列程式中，-- 代表主指令的參數到此為止，之後的參數會傳到目標指令上。假定指令如下：

```
$ inspxe-cl -collect mi1 -- ./program --opt var path/to/file
```

⁴³ <https://software.intel.com/en-us/inspector-user-guide-linux-collect>

這時後，主指令為 `inspxe-cl -collect mi1`，目標指令為 `./program --opt var path/to/file`，兩者間以 `--` 相隔。

執行完程式後，Intel Inspector 會將收集的資料放在資料夾中，像是 `r000mi1`。該資料夾內有很多文字檔案，這些檔案不是要直接觀看的，而是要餵給 Intel Inspector 產生報告的。

接著，再執行一次 Intel Inspector 以產生報表：

```
$ inspxe-cl -report problems -result-dir ./r000mi1
```

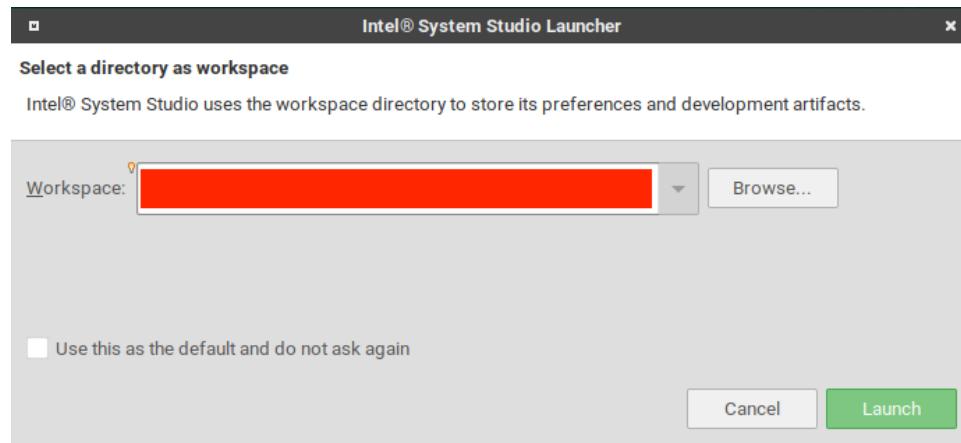
報表的形式有好幾種⁴⁴，請依自己的需求選擇。

筆者平常使用 Valgrind 而非 Intel Inspector。但 Valgrind 僅支援 GNU/Linux 系統，對 Windows 使用者來說，可以試著用 Intel Inspector 檢查 C 或 C++ 程式。

使用 Intel System Studio 隨附的 IDE

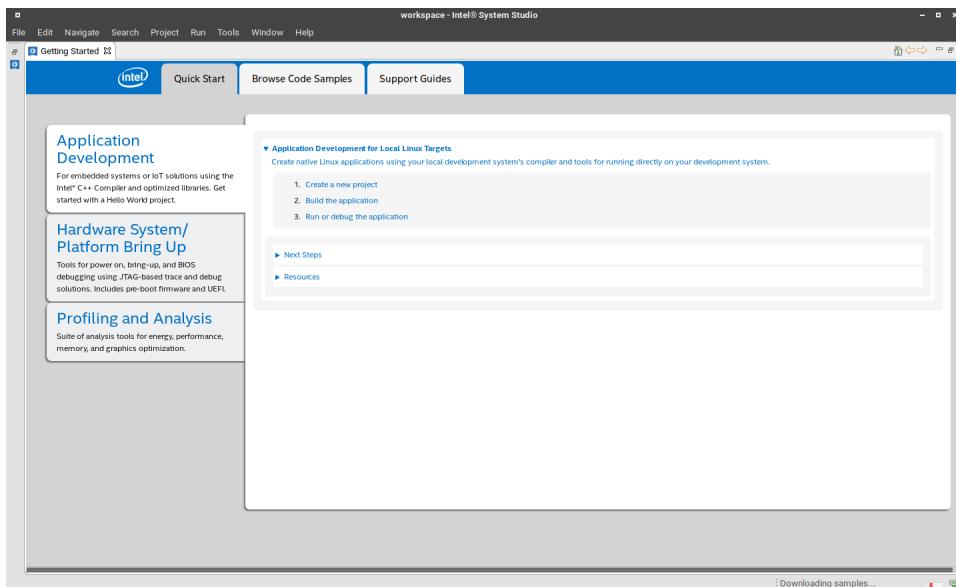
除了從命令列使用 Intel C++ Compiler 外，也可以使用 Intel System Studio 隨附的 IDE。基本上這套 IDE 是特化版的 Eclipse。原本 Eclipse 是 Java IDE，後來演變為多語言 IDE，當然也包括 C 或 C++。

開啟 Intel System Studio 時先選擇 workplace：

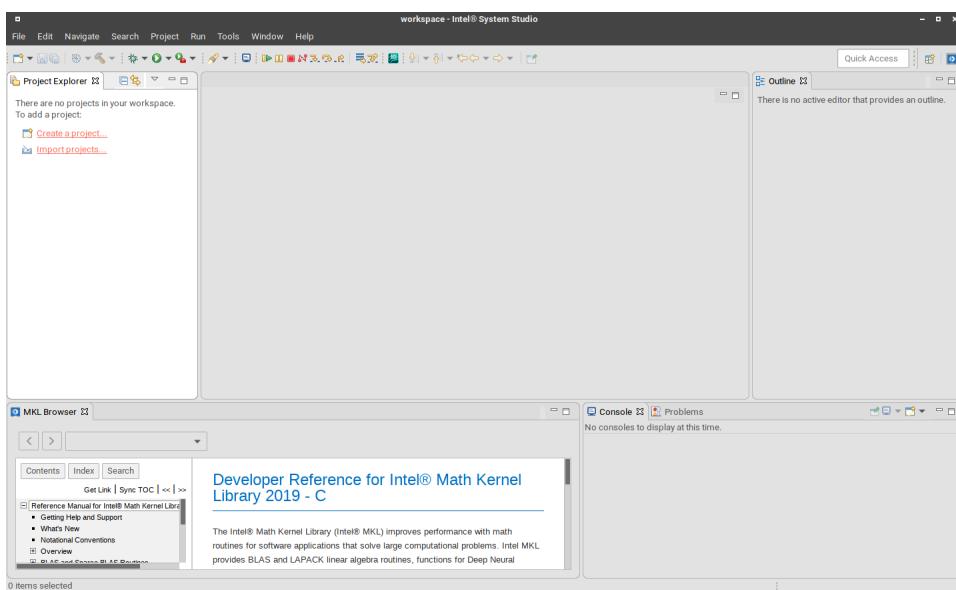


第一次開啟 Intel System Studio 時會有 Getting Started 的歡迎畫面：

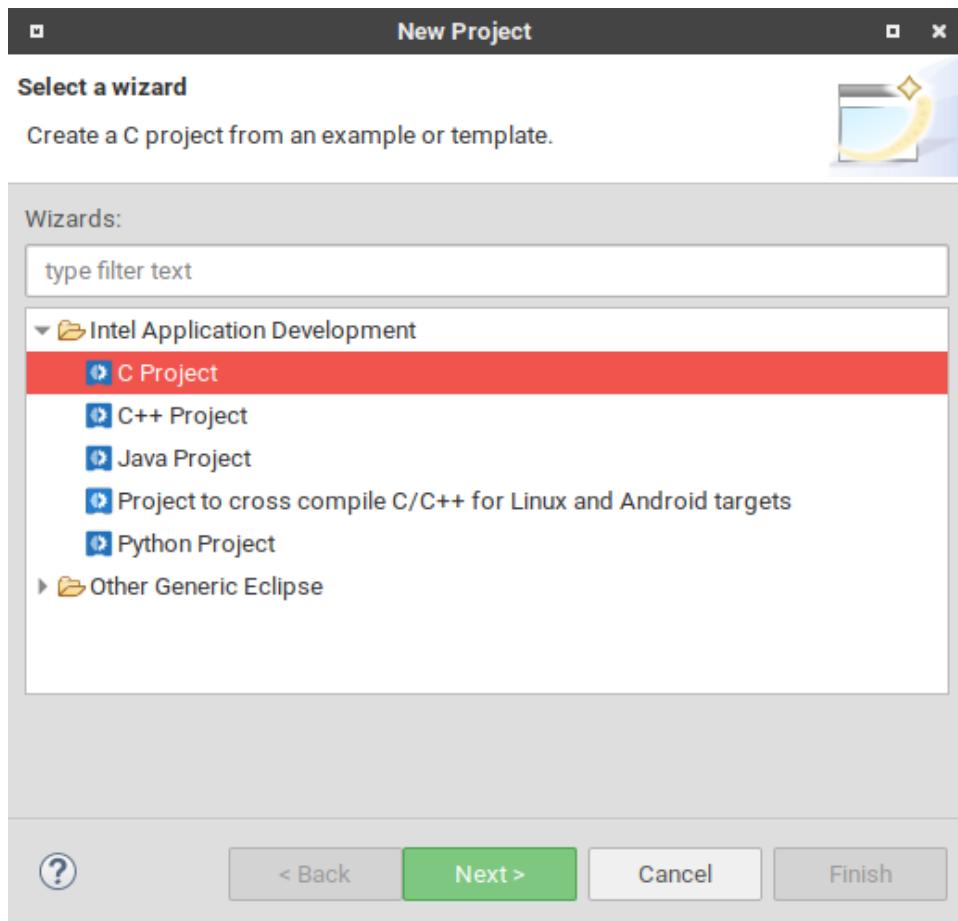
⁴⁴ <https://software.intel.com/en-us/inspector-user-guide-linux-report>



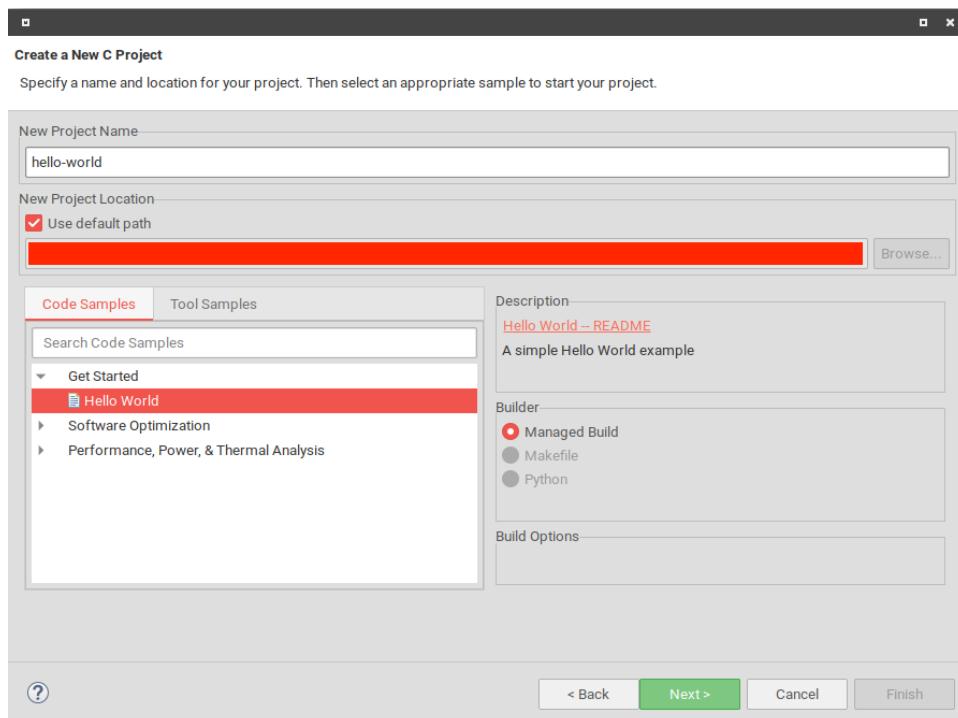
實際的編輯器界面如下：



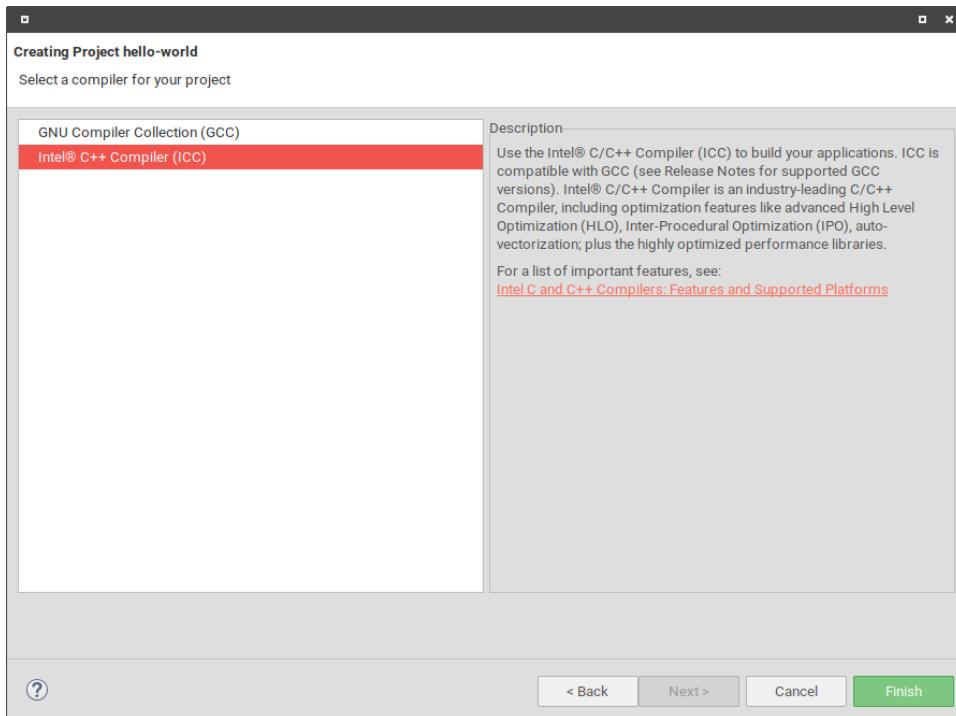
在本節中，我們選擇 C Project：



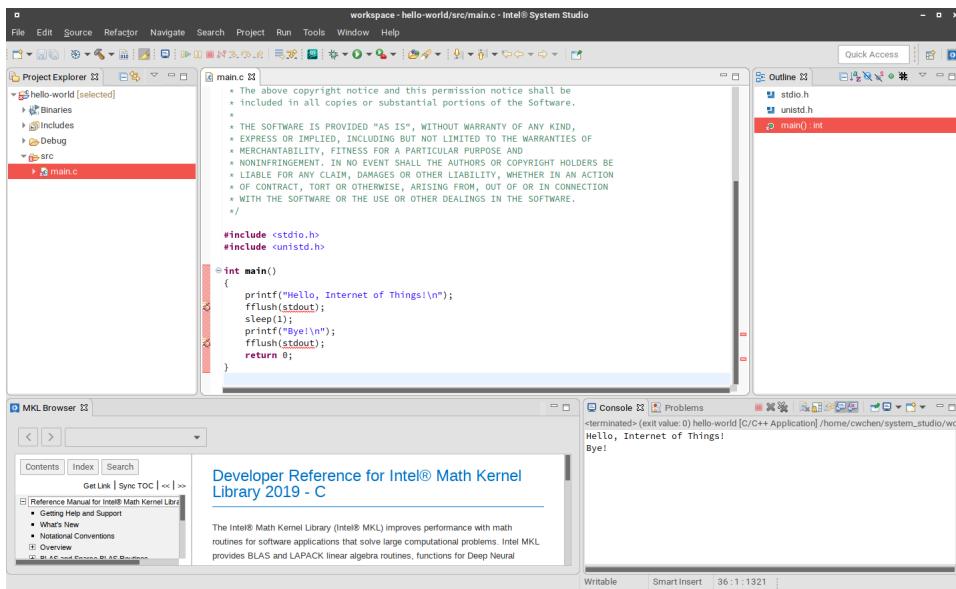
從 Hello World 模板中建立專案：



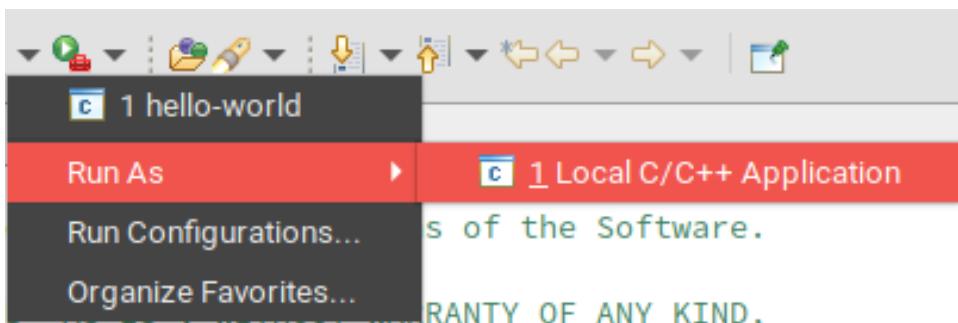
選擇專案所用的 C 編譯器：



一開始專案的程式碼只有簡單幾行：



按下 IDE 上方的 Run As 即可編譯程式：



筆者平常不會依賴 IDE 來管理 C 或 C++ 專案，而會自己手寫 GNU Make 或 CMake 設定檔。本節內容僅留給需要的讀者參考。

繼續深入

在本文中，我們只介紹 Intel C++ Compiler 和其他的開發工具，但 Intel System Studio 還有一些可以注意的地方，像是數理運算的函式庫、多執行緒函式庫、加密函式庫、物聯網開發工具等。如果讀者的程式會在 Intel 平台上跑，可以去學習這些函式庫或開發工具的用法。

附記

在使用 Intel C++ Compiler 和 Intel Inspector 的過程中，筆者發現了一些小錯誤。像是 Intel Inspector 會有偽陽性，這些錯誤可能來自 Intel C++ Compiler 或外部函式庫。網路論壇中也提到有時 Intel C++ Compiler 編譯出來的程式會崩潰、無法使用。

由於 Intel C++ Compiler 的使用率較低，所以錯誤報告不多，無法確切得知 Intel C++ Compiler 的品質如何。如果程式只是用來跑科學運算，使用 Intel C++ Compiler 編譯應該無妨。如果是要對外發佈的程式或伺服程式，可能還是用 GCC 或 Clang 編譯較佳。

善用開發工具改善 C 程式專案

前言

除了編譯器和編輯器等必要的軟體外，還有許多和撰寫 C 程式相關的開發工具。由於這些軟體不是必備的，所以一些初階的 C 語言教材不會介紹這些軟體。但這些工具對於撰寫 C 專案有不同面向的助益，行有餘力的話，可以試著關注一下這些工具，慢慢學習這些工具的使用方式。

試著用多種 C 編譯器編譯 C 程式碼

在大部分情形下，我們把 C 編譯器當成老師，根據 C 編譯器所吐出的訊息來修正我們的程式碼。但在很偶然的情形下，C 編譯器也會出錯。

筆者曾經在某個知名的類 Unix 系統上用 GCC 和 Clang 編譯同一份 C 專案，GCC 編譯出來的執行檔有誤，但 Clang 編譯出來的結果是正常的。事後用 Valgrind 和其他類 Unix 系統的 GCC 確認過，不是 C 專案本身的問題。

所以，最好用 GNU Make 或 CMake 這類跨平台軟體設置 C 專案，將專案設置成可應對多種 C 編譯器，這時候就可以交叉使用不同 C 編譯器來編譯同一份 C 專案。一般來說，專案最好能夠對應 GCC、Clang、Visual C++ 等主流 C 編譯器。詳見下節。

編譯自動化軟體 (Build Automation)

剛開始學 C 語言時，我們會直接用 IDE 來管理專案，這樣做的好處在於不需要手動寫設定檔，可以專心在學習語法上。但是，過度依賴 IDE 的專案管理功能，會使得專案的可攜性降低。IDE 並不是 C 專案必要的部分，我們可以使用完全不依賴 IDE 的編譯自動化軟體來管理管案，讓專案和 IDE 脫勾。

C 語言本身沒有官方的專案格式，但我們可利用一些社群方案來管理 C 專案。常見的編譯自動化軟體如下：

- Make
- Autotools⁴⁵
- CMake⁴⁶

⁴⁵ <https://www.gnu.org/software/automake/>

⁴⁶ <https://cmake.org/>

Make 算是編譯自動化軟體的鼻祖，也是 POSIX 標準的項目之一⁴⁷，在類 Unix 系統上相當普遍。但 Make 算是純命令列工具，無法結合 Visual Studio 等 IDE 的專案管理功能。其實我們可以用 Make 直接在命令列操作 Visual C++，只是實務上會這樣做的程式人甚少。

由於寫跨平台的 Makefile 相當困難，因而出現 Autotools 這類 Makefile 生成器。Autotools 會根據系統當時的情境和使用者所設置的參數，生成相對應的 Makefile。但 Autotools 是設計給類 Unix 系統使用的，Windows 上則無法使用。

相較於 Make，CMake 不是真正的編譯自動化軟體，而是編譯自動化設定檔生成器。透過 CMake，我們可以從單一設置產生 Make、Visual Studio、Xcode、Code::Blocks 等多種軟體的編譯自動化設定檔，再分別利用這些軟體去編譯 C 專案，利用這種機制來間接達成跨平台的特性。

Visual Studio 和 CLions 等商業 IDE 都支援 CMake，代表 CMake 的確是受到重視的軟體專案。由於 CMake 可以涵蓋 Make 的功能，但反之則否，我們應優先學習 CMake。

除錯器 (Debugger)

程式往往不會一次就寫對，在程式發生錯誤時，就要對程式除錯。實際上，除錯在整個撰寫程式的過程中，甚至會占掉一半以上的時間。

除錯器 (debugger) 就是用來輔助除錯的軟體。除錯器會在程式執行到中斷點 (breaking point) 時將程式中止，這時候程式設計者就可以觀察程式內部狀態。觀察狀態的方式是觀察變數執行到中斷點時的值。

對於 C 語言來說，GDB 是相當常見的除錯器。若要使用 GDB 除錯，在編譯程式時要加上 -g 選項：

```
$ gcc -g -o program source.c
```

接著，用 gdb 程式去執行該程式：

```
$ gdb ./program
```

這時候會來到 GDB 的互動式環境：

```
(gdb)
```

輸入 list 指令加上行數可以跳到特定行數所在的位置：

```
(gdb) list 35
```

輸入 break 指令加上行數可以在特定行數設置中斷點：

⁴⁷ <https://pubs.opengroup.org/onlinepubs/009695399/utilities/make.html>

```
(gdb) break 34
```

輸入 `run` 指令會啟動程式至中斷點所在的位置：

```
(gdb) run
```

輸入 `print` 指令並指定變數即可讀出該變數在程式執行到中斷點時的狀態：

```
(gdb) print out
```

這裡的 `out` 是變數名稱，而非指令的一部分，請讀者不要死記這個變數名稱。

確認完後，用 `clear` 清除中斷點：

```
(gdb) clear
```

執行 `quit` 即可離開 GDB。

```
(gdb) quit
```

有些程式設計者完全不用除錯器除錯，而用 `printf` 等函式在終端機中印出資料來觀察程式狀態。但使用 `printf` 函式印出資料的方式，無法將程式中止在特定步驟，和使用除錯器仍有差異。此外，使用 `printf` 函式會在程式中留下額外的程式碼，而使用除錯器不會。

GDB 本身是命令列工具，而有些程式設計者不習慣在命令列環境下除錯。許多 IDE 都內建除錯器的功能，就可以在圖形介面環境下除錯。不論是使用 GDB 或其他除錯器，最好還是花一點時間學習除錯器的用法。

自動程式碼重排 (Code Formatting)

我們在撰寫 C 程式碼時，通常會把 C 程式碼排列整齊。排列後的 C 程式碼不僅易於閱讀，之後要修改也比較容易。

但排列程式碼是相當機械化的動作。對於多人協作的專案來說，要保持一致的程式碼風格更是困難。利用自動程式碼重排軟體，我們可以省下排列程式碼的時間和心力，輕鬆取得較一致的撰碼風格。

常見的 C 語言自動程式碼重排工具如下：

- `indent`⁴⁸
- `astyle`⁴⁹

⁴⁸ <https://www.gnu.org/software/indent/>

⁴⁹ <http://astyle.sourceforge.net/>

- ClangFormat⁵⁰
- uncrustify⁵¹

以 `indent` 為例，使用方式如下：

```
$ indent file.c
```

這時候會產生排列過的 `file.c` 和備分檔 `file.c.~`。如果對排列的結果不滿意，可以及時用備分檔回復。

由於 `indent` 是 GNU 專案，預設使用 GNU 風格。但台灣的 C 語言教材甚少使用 GNU 風格，較常用 K&R 風格。可以用以下參數來更動排版風格：

```
$ indent -kr file.c
```

其實 `indent` 的選項很多，為了節省程式人的時間，會用前述的風格 (style) 快速設置一系列選項。常見的風格如下：

- `-gnu`：GNU 專案所用的風格
- `-kr`：K&R 風格，台灣的 C 語言教材常用
- `-orig`：BSD 風格

不論使用那套程式碼重排軟體，在多人協作時，建議先配置好共通的設定檔，在團隊中皆使用該設定檔，就可以取得一致的撰碼風格。

靜態程式碼檢查 (Static Code Analysis)

由於 C 語言是靜態型別語言，本來就可以比動態型別語言抓出更多錯誤。此外，GCC 和 Clang 等編譯器也內建許多非錯誤的警告訊息，可透過參數開啟這些資訊。不過，仍然有一些給 C 或 C++ 使用的靜態程式碼檢查軟體。以下是常見的方案：

- Clang Static Analyzer⁵²
- Splint⁵³
- cppcheck⁵⁴
- infer⁵⁵ (限 GNU/Linux 及 macOS)

註：目前 Splint 已經停止維護，故不建議繼續使用。

⁵⁰ <https://clang.llvm.org/docs/ClangFormat.html>

⁵¹ <https://github.com/uncrustify/uncrustify>

⁵² <https://clang-analyzer.llvm.org/>

⁵³ <http://lclint.cs.virginia.edu/>

⁵⁴ <http://cppcheck.sourceforge.net/>

⁵⁵ <https://fbinfer.com/>

以下是使用 `cppcheck` 檢查 C 程式碼的範例指令：

```
$ cppcheck --enable=warning,style,performance,portability --std=c89
  ↳path/to/file.c
```

在此指令中，`cppcheck` 會檢查四個項目，並檢查 C 程式碼是否符合 ANSI C (C89)。

以下是使用 `infer` 檢查 C 專案的範例指令：

```
$ infer run -- make
```

這時候 `infer` 會在編譯專案的過程中掃描編譯的 C 程式碼，並回報有問題的程式碼。若要再次進行檢查，得先清除專案的暫存物件 (`.o`)。

由於靜態程式碼檢查軟體會有偽陽性，這些軟體檢查的結果，無法取代我們自己對程式碼的了解。如果對這些軟體吐出的訊息有疑問，還是得自己去查詢相關的資訊。

記憶體用量檢查 (Memory Checking)

C 語言在預設情境下需要手動管理記憶體，記憶體處理不當就成了 bug 的來源之一。所幸，我們可以使用記憶體管理軟體來檢查我們寫的 C 程式碼是否有錯。

在類 Unix 系統上最知名的記憶體檢查軟體就是 `Valgrind`⁵⁶。由於 Valgrind 已經問世多年，算是相當成熟的軟體。當 Valgrind 報錯時，九成以上是程式碼本身的問題，只有一成以內是 Valgrind 誤判。

假定我們的程式為 `program`，搭配 Valgrind 的指令如下：

```
$ valgrind ./program
```

基本上就是把要檢查的程式當成 Valgrind 軟體的第一個參數即可。如果執行目標程式時要加參數，也可以直接加在第二個之後的參數。

如果沒有記憶體洩露，Valgrind 會出現類似以下的報告：

```
==5030== HEAP SUMMARY:
==5030==     in use at exit: 0 bytes in 0 blocks
==5030==   total heap usage: 2,071 allocs, 2,071 frees, 56,470 bytes
  ↳allocated
==5030==
==5030== All heap blocks were freed -- no leaks are possible
```

如果出現記憶體洩露，Valgrind 則會出現類似以下的報告：

⁵⁶ <http://valgrind.org/>

```
==5295== HEAP SUMMARY:
==5295==     in use at exit: 24 bytes in 1 blocks
==5295==   total heap usage: 2,071 allocs, 2,070 frees, 56,470 bytes allocated
==5295==
==5295== LEAK SUMMARY:
==5295==   definitely lost: 24 bytes in 1 blocks
==5295==   indirectly lost: 0 bytes in 0 blocks
==5295==   possibly lost: 0 bytes in 0 blocks
==5295==   still reachable: 0 bytes in 0 blocks
==5295==   suppressed: 0 bytes in 0 blocks
==5295== Rerun with --leak-check=full to see details of leaked memory
```

Valgrind 會吐出程式執行的堆疊，讓我們知道記憶體洩露的發生位置，但仍然要程式人自己去檢查程式碼，從中找出錯誤。

Windows 和 macOS 無法使用 Valgrind，得使用其他替代軟體，像是 Dr. Memory⁵⁷。

如果知道自己的程式有用到手動記憶體配置，最好花一下時間用 Valgrind 等軟體檢查一下，並自行修復錯誤。透過這樣的過程，養成自己良好的撰碼習慣。

其實 C 語言有第三方 GC (垃圾回收器) 函式庫，像是 Boehm GC⁵⁸。但真正會使用 Boehm GC 或其他 GC 函式庫的 C 專案很少。此外，現在已有 Golang 或 Rust 等自動管理記憶體的編譯語言，使用起來更加簡單。現行的主流手法仍是好好地用 Valgrind 等軟體檢查自己的 C 程式碼是否有記憶體使用的問題。

測試程式 (Testing)

測試程式是用來檢查主程式是否有問題的程式，不會隨主程式一起發佈出去，只是在開發時期做為內部使用。初學時程式規模很小，往往會過度依賴手動檢查，忽略自動化測試所帶來的便利性，養成不良的習慣。

寫測試程式是一個先苦後甘的過程。因為寫測試程式不會直接增加程式人的產能，算是額外做的工。但我們日後要重構 (refactoring) 專案時，若專案當時有留下測試程式，可讓重構的過程更加順暢。

以下是一些常見的 C 語言測試框架：

- Check⁵⁹ 
- CUnit⁶⁰

如果不想用額外的函式庫，也可以自己用 C 內建的控制結構寫一些簡單的測試程式。

⁵⁷ <https://drmemory.org/>

⁵⁸ <https://www.hboehm.info/gc/>

⁵⁹ <https://libcheck.github.io/check/>

⁶⁰ <http://cunit.sourceforge.net/>

撰寫程式文件 (Documentation)

如果專案是要對外發佈的，除了寫程式碼外，也要幫專案寫文件。外部專案使用者不會在缺乏足夠的文件、未了解專案如何使用時，自動自發去閱讀專案的程式碼。使用者只會在喜歡或信任該專案，但想進一步了解該專案的實作細節時，才會願意花時間去閱讀專案程式碼。

以下是兩種不同的專案文件撰寫工具：

- Doxygen⁶¹：撰寫軟體 API 文件
- Sphinx⁶²：撰寫軟體說明文件

API 文件如同軟體專案的指引，會詳細地展示每個函式或物件的參數、回傳值等資訊。但不一定會有範例程式。API 文件是對專案已有一定熟悉度，想要查詢特定函式的用法時會查詢的文件。

Doxygen 是用於 C、C++、Java 等程式語言的 API 文件生成工具。Doxygen 文件的原始碼是以註解的形式存在於專案原始碼中，所以不需另外準備一份文件檔案。由於 Doxygen 文件的資訊來自註解，不會自動隨著程式碼更新而更新，負責專案的程式人得自己更新 Doxygen 文件的內容。

說明文件則是軟體專案的教程，目的是讓對專案不熟的程式人學習該專案的用法，所以也有推廣專案的用意。由於說明文件可能有多種格式，像是 HTML 文件、PDF 文件、EPUB 電子書等。所以會利用 Sphinx 等說明文件軟體，以單一文件原始碼產生多種格式的說明文件。

版本控制軟體 (Version Control)

版本控制軟體對使用 C 專案來說，不是必備的，但對管理 C 專案卻相當有幫助，尤其是在管理多人協作的專案。版本控制軟體的基本概念是幫軟體專案額外加上狀態，就好像是玩電腦遊戲到某個段落時存取遊戲的狀態般。

當專案具有狀態的概念時，可以在寫錯程式碼的時候將程式碼回溯到先前的版本，也可以在程式碼出現衝突時偵測衝突事件，並提醒程式設計者。除此之外，版本控制還有許多功能，用來處理各式各樣的管理情境。

目前最廣泛使用的版本控制軟體是 Git⁶³。像是目前最受歡迎的專案管理網站 GitHub 就是使用 Git 來傳輸專案。另一個知名的專案管理網站 BitBucket 甚至棄用原本有支援的 Mercurial，全面改用 Git 來管理專案。

⁶¹ <http://www.doxygen.nl/>

⁶² <http://www.sphinx-doc.org/>

⁶³ <https://git-scm.com/>

前言

在本文中，我們以 Hello World 程式為例，說明 C 語言的基本概念。

C 程式所用的副檔名

C 程式碼使用兩種副檔名，原始碼 (source) 使用 .c 為副檔名，標頭檔 (header) 使用 .h 為副檔名。原始碼中會放入實作 (implementation)，標頭檔則是擺放宣告 (declaration)。

C 語言是編譯語言，編譯出來的產物是執行檔。Unix 的執行檔不加副檔名，Windows 的執行檔的副檔名為 .exe 。執行檔的副檔名不是 C 程式特有的，所有的編譯語言所編譯出來的執行檔都使用相同的副檔名。

原始碼和標頭檔的檔案名稱沒有特別限制，只要是作業系統中合法的檔名即可。一般建議用簡短、有意義的檔名，並用底線取代空白。

一開始練習的程式規模很小，將所有的程式碼寫在原始碼中無妨。待程式規模變大後，會將原始碼拆分到多個檔案中，這時候就會用到標頭檔。

第一個 C 程式

最小的 C 程式如下：

```
int main() {}
```

但這個程式什麼事也沒做，而且看不出來 C 程式的基本架構。我們略為修改成加上註解的 Hello World 程式如下：

```
/* Include the library for standard input and output. */
#include <stdio.h>

/* Entry to the main program. */
int main(void)
{
    /* Print some string to console. */
    printf("Hello World\n");
```

(continues on next page)

(continued from previous page)

```
/* Exit the program successfully. */
return 0;
}
```

我們會以這個程式為基準，說明 C 語言的基本概念。

編譯 C 程式的步驟

表面上，編譯 C 程式只是一行指令或一個 IDE 的按鈕就解決的事。但編譯 C 程式實際上分為四個步驟：

- 前處理 (Preprocessing)
- 編譯 (Compilation)
- 組譯 (Assembly)
- 連結 (Linking)

前處理將含巨集 (macro) 的 C 程式碼轉換成沒有巨集的 C 程式碼。嚴格上來說，前處理不算編譯的一部分，只是經由一連串字串代換的動作改寫原始碼而已。我們會在後文介紹 C 巨集。

編譯 (compilation) 是整個編譯 (building) 的前半段。在這個階段，C 編譯器會將 C 原始碼轉換成等效的組合語言原始碼。由此可知，C 仍然是高階語言，只是寫起來不若現代語言那麼方便。

組譯會將組合語言原始碼轉換成機械碼。轉換後的檔案為目的檔 (object files)。目的檔只是編譯的中間產物，無法使用。

最後，透過連結，將目的檔轉為執行檔 (executable)。執行檔即為可使用的電腦程式 (program)。

剛開始練習的程式很簡短，只會用到標準函式庫的函式，所以編譯的步驟較簡單。隨著程式的規模上升，會使用模組化的方式將程式碼分散在多個檔案中，也有可能會用到外部函式庫，這時候編譯的步驟就會變複雜。

大小寫敏感性 (Case Sensitivity)

C 語言的程式碼會區分大小寫。所以，`foo`、`Foo` 和 `FOO` 視為不同的識別字。在實務上，大部分的識別字會用小寫字母。全大寫字母僅用於常數 (constant) 和列舉 (enumeration)。按照 C 社群的慣例，幾乎不用 CapitalCase 來寫識別字。

即使 C 語言會區分大小寫，我們也不應濫用這項特性，以免寫出難以維護的程式碼。

空白 (Spaces)、縮進 (Indentations)、換行 (End of Line)

C 語言對於空白、縮進、換行等版面安排相當自由，並沒有 Python 那種強制縮進的規則。甚至還有故意寫出難以閱讀的 C 程式碼的比賽，像是 IOCCC (The International Obfuscated C Code Contest)⁶⁴。但我們仍然鼓勵讀者在寫 C 程式碼時排版程式碼，或是使用程式碼自動重排軟體。因為整齊的程式碼對於日後維護會有所幫助。

由於 C 程式碼在編譯後就會轉成機械碼，執行編譯出來的程式不需要再重新解析原始碼。因此，對 C 程式碼刻意做縮小化 (minification) 只會影響到編譯程式碼的時間，對程式執行速度沒有幫助。基本上這是沒有意義的行為。

註解 (Comments)

程式碼中的註解不會執行，所以可寫入一般文字。C 語言有兩種風格的註解。ANSI C 是使用一對 /* 和 */ 把註解文字包起來，像是以下範例：

```
/* Some comment. */
```

由於傳統的註解能夠跨越多行，故未完全淘汰。

在 C99 後，引入單行註解。單行註解以 // 開頭，之後同一行內的所有文字皆視為註解。像是以下範例：

```
// Some single-line comment.
```

單行註解比較易寫，但無法跨越多行，所以未取代傳統的註解。

除了這兩種正統的註解外，還有另一種利用巨集來註解掉整段程式碼的手法：

```
#if 0
    printf("It won't compile\n");
#endif
```

這樣操作的原理在於巨集會在編譯前就執行，相關程式碼會被抹去，等同於這段程式碼不存在。

註解的用途是記錄程式人撰寫程式碼時的意圖或想法。對於教學用的程式，註解可放入教學用文字。

有時候我們會用註解暫時隱藏掉一段程式碼，防止該段程式碼編譯及運作。例如，我們在除錯時，利用註解遮蔽掉可能有問題的程式碼，然後逐步縮小註解的範圍，直到找到錯誤為止。

⁶⁴ <https://www.ioccc.org/>

撰碼風格 (Coding Style)

在程式碼中使用一致的撰碼風格，會增加程式碼的可讀性。實務上，會以開發團隊所用的撰碼風格為準。對於獨立開發者來說，則在自身的程式碼中保持一致即可。

C 常見的撰碼風格有 K&R、BSD、GNU 等。台灣的 C 教材多使用 K&R 風格。本書的程式碼會以 K&R 風格為主。

標準函式庫

C 標準函式庫是一組預寫好的宣告、函式、巨集等。由於標準函式庫已經預先實作好一些常見的功能，C 程式設計者就不需重覆實作。除非有好的理由，應優先使用標準函式庫的功能，而非重造輪子。

由於 C 標準函式庫相當精簡，很多功能無法透過標準函式庫取得。這時候會需要使用第三方函式庫。若沒有合適的第三方函式庫，則需自行實作所需的功能。

引入函式庫

C 語言的語法刻意保持精簡，大部分的功能由函式庫來實現。甚至標準輸出入這種基本的功能也是藉由函式庫來實現。所以，學 C 語法不會花太久時間，但要寫到熟練則需要反覆練習。

引入函式庫的語法是 `#include`。該語法是一種巨集 (macro)，但我們一開始不需要在意巨集的寫法，因為 `#include` 敘述的寫法是固定的。

引入函式庫時，有兩種寫法。一種是以一對角括號 < 和 > 包住函式庫名稱。一種則是用一對分號 " 和 " 包住函式庫名稱。C 語言沒有強制要使用那一種方式。

一般來說，使用標準函式庫或外部函式庫時，使用角括號：

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

使用內部函式庫時，則使用分號：

```
#include "mylib.h"
```

當使用分號引用函式庫時，C 編譯器會優先從 C 程式碼所在的路徑找標頭檔，故可用來引用內部函式庫。

主函式 (Main Function)

主函式是 C 程式的起始點，一般的 C 語言皆有主函式。但在嵌入式系統或是作業系統本身的 C 程式碼中，主函式則不是必需的。

如果該 C 程式不需要接收命令列參數，則使用以下方式來寫：

```
int main(void)
{
    /* Implement your code here. */
}
```

若該 C 程式需要接收命令列參數，則改用以下方式來寫：

```
int main(int argc, char *argv[])
{
    /* Implement your code here. */
}
```

除了命令列參數外，如果想要接收環境變數，可以用以下方式改寫：

```
int main(int argc, char *argv[], char *env[])
{
    /* Implement your code here. */
}
```

現在這種寫法比較少見，因為標準 C 可用 getenv()⁶⁵ 函式取得環境變數，沒有必要再用這種寫法。

至於以下寫法是錯誤的：

```
void main(void)
{
    /* DON'T USE THIS IN PRODUCTION CODE. */
}
```

雖然 Visual C++ 可編譯通過，但這種寫法並非標準 C 的一部分，故不建議使用。

⁶⁵ <https://en.cppreference.com/w/c/program/getenv>

表達式 (Expression) 和敘述 (Statement)

表達式會回傳值，像是 "Hello World\n" 是表達式，該表達式是 "Hello World" 字串再附帶一個換行符號。而敘述代表單一指令，像是 `printf("Hello World\n");`；是一條敘述。C 語言的單行敘述會用 ; 結尾。

表達式可以當成敘述使用，但反之則不一定成立。C 語言是命令式語言，每條指令都是以敘述來寫。

C 程式由一條條敘述組成，預設情形下，程式會由上往下依序執行敘述。但我們有許多改變程式行進流程的方式，像是控制結構或函式呼叫等。

離開狀態 (Exit Status)

我們在程式結束時回傳 0：

```
return 0;
```

這代表程式正常結束。

C 程式在結束時，會向系統回傳一個整數，用來代表程式的狀態。一般來說，回傳 0 代表程式正常結束，回傳非零值 1 代表程式異常結束。

有些程式會用不同的回傳值表示不同的異常狀態，但除了回傳 0 表示正常結束以及回傳 1 表示異常結束外，目前 C 語言對其他的回傳值沒有共識。程式使用者需查詢該程式的使用手冊才能知道回傳值的意義。

由此可知，用回傳值來判斷程式狀態不具有可攜性。我們不鼓勵讀者使用複雜的回傳值來表示程式的離開狀態。

C 執行期函式庫

雖然 C 程式表面上不需要虛擬機器，但其實 C 程式仍然有執行期函式庫。像是 C 程式具有主函式，且主函式會回傳離開狀態，C 程式需要執行期函式庫來處理主函式。

C 語言的執行期函式庫很小，通常是以組合語言來撰寫，會在編譯 C 程式碼時一併加入 C 程式中，所以不需要另外安裝虛擬機器。C 語言的運行期函式庫稱為 crt0。

MSVC (Visual C++) 在不同版本的 C 編譯器使用不同的執行階段函式庫，所以在 Windows 上安裝以 C 撰寫的應用程式時，有時會需要安裝特定版本的執行階段函式庫。在 MSVC 中的 C 執行階段函式庫稱為 Visual C++ 可轉散發套件。

資料型態 (DATA TYPE)

前言

絕大部分的程式語言都有資料型態 (data type) 的特性。資料型態是資料的標註 (annotation)，用來規範電腦程式對該資料的合理操作。在本文中，我們會介紹 C 語言的資料型態。

C 語言的資料型態

以下是 C 語言的資料型態：

- 基礎型態 (fundamental types)
 - 布林數 (boolean) (**C99**)
 - 整數 (integer)
 - 浮點數 (floating-point number)
 - 複數 (complex number) (**C99**)
 - 字元 (character)
 - 列舉 (enumeration)
- 衍生型態 (derived types)
 - 陣列 (array)
 - 結構體 (structure)
 - 聯合體 (union)
 - 指標 (pointer)
 - 函式 (function)

基礎型態用來標註純量 (scalar) 形態，像是整數、浮點數、字元等。基礎型態無法再化簡成更小的單位。不同資料型別在記憶體中有不同的儲存方式。

相對來說，衍生型態則是基於其他型態所建立的型態，像陣列是資料的容器等。衍生型態的特質在於保留使用者自訂的彈性。

雖然我們可以用結構體等型態來模擬其他高階語言的類別 (class)，嚴格來說，C 語言沒有類別的概念。所謂的物件導向 C (object-oriented C) 只是一種利用物件導向程式的概念整理 C 程式碼的方式。

除了陣列以外，C 語言沒有其他的內建資料結構型態。如果 C 程式設計者需要某種資料結構，就要自己實作。雖然有些社群發佈的資料結構函式庫，這些函式庫並未形成共識。所以，實作資料結構對於 C 程式設計者來說仍然是相當重要的議題。

布林數 (Boolean) (C99)

C 語言沒有原生的布林數 (boolean)，但 C 語言有布林語境 (boolean context)，像是 $5 > 3$ 等關係運算。在這個例子中， $5 > 3$ 為真，故回傳 1。相對來說， $3 > 5$ 為偽，會回傳 0。而 1 在條件句中視為真，0 視為偽。

在 C99 之前，常見的手法是用巨集自行宣告真值和偽值，如下例：

```
#define TRUE    1  
#define FALSE   0
```

C99 的 `stdbool.h`⁶⁶ 函式庫，就是把巨集宣告這件事標準化，避免各軟體專案各自為政的情形。在 C 程式碼引入該函式庫後，可以得到以下三個巨集宣告：

- `true`：展開為常數 1
- `false`：展開為常數 0
- `bool`：展開為 `_Bool` 型別

由於 `stdbool.h` 是標準函式庫的一部分，只要自己使用的 C 編譯器有支援，應優先使用。

整數 (Integer)

根據正負號的有無和記憶體容量的大小，C 語言的整數細分為數個型態：

- 無號整數 (`unsigned integer`)
 - `unsigned short`
 - `unsigned int`
 - `unsigned long`
 - `unsigned long long (C99)`
- 帶號整數 (`signed integer`)
 - `short` 或 `signed short`
 - `int` 或 `signed int`
 - `long` 或 `signed long`

⁶⁶ <https://en.cppreference.com/w/c/types/boolean>

- long long 或 signed long long (**C99**)

會分那麼細主要是為了節約系統資源。實際使用時，只要知道該整數型態的範圍即可。一開始不會用時，先一律用 int 型態，之後再慢慢練習細分即可。

實際上整數型態的範圍大小會隨系統而異，並非一成不變。在 C 語言的 **limits.h** 提供數個和整數型別上下限相關的巨集宣告。我們可以利用這些巨集宣告來檢查自己系統的整數型別的上下限。參考以下範例程式：

```
#include <limits.h>
#include <stdio.h>

int main(void)
{
    printf(
        "Max of signed char: %d\n",
        CHAR_MAX);
    printf(
        "Min of signed char: %d\n",
        CHAR_MIN);

    printf(
        "Max of signed short: %d\n",
        SHRT_MAX);
    printf(
        "Min of signed short: %d\n",
        SHRT_MIN);

    printf(
        "Max of signed int: %d\n",
        INT_MAX);
    printf(
        "Min of signed int: %d\n",
        INT_MIN);

    printf(
        "Max of signed long: %ld\n",
        LONG_MAX);
    printf(
        "Min of signed long: %ld\n",
        LONG_MIN);

#if __STDC_VERSION__ >= 199901L
    printf(
        "Max of signed long long: %lld\n",
        LLONG_MAX);
    printf(

```

(continues on next page)

(continued from previous page)

```

    "Min of signed long long: %lld\n",
    LLONG_MIN);
#endif

printf("\n"); /* Line separator. */

printf(
    "Max of unsigned char: %u\n",
    UCHAR_MAX);

printf(
    "Max of unsigned short: %u\n",
    USHRT_MAX);

printf(
    "Max of unsigned int: %u\n",
    UINT_MAX);

printf(
    "Max of unsigned long: %lu\n",
    ULONG_MAX);

#if __STDC_VERSION__ >= 199901L
printf(
    "Max of unsigned long long: %llu\n",
    ULLONG_MAX);
#endif

return 0;
}

```

對於無號整數來說，最小值一律為 0，故未列出。

筆者自己在測試時，long 和 long long 的範圍是相同，但在其他系統上這兩者可能會有差異。讀者可以在自己的電腦上試跑這個小程式看看。

細心的讀者可能有發現我們在這裡列入字元型態 char。因為 char 內部仍然是以整數來儲存，我們可以把 char 當成小範圍的整數來用，可節約系統資源。

如果我們需要更大位數的整數呢？這時候就要透過大數 (big number) 函式庫來運算。大數是以軟體模擬的，不受硬體的先天限制，但速度會比內建數字型態慢一些。像 GMP⁶⁷ 就是一個大數函式庫。

⁶⁷ <https://gmplib.org/>

固定寬度整數 (C99)

ANSI C 語言不保證整數的寬度，會因系統而異。著眼於這項議題，C99 新增 stdint.h⁶⁸ 函式庫，提供固定寬度的整數。有需要的讀者可自行參考。

浮點數 (Floating-Point Number)

C 語言的浮點數細分為三種：

- float
- double
- long double

三種浮點數在最大值、最小值、精準位數等會有一些差異。一開始時，先一律用 double 即可，之後再學著依情境細分。

同樣地，不要硬背浮點數的範圍、精準度等資訊。可以試著寫小程式在自己系統上面跑。參考以下範例程式：

```
#include <float.h>
#include <stdio.h>

int main(void)
{
    printf(
        "Max of float: %e\n",
        FLT_MAX);
    printf(
        "Min pos of float: %e\n",
        FLT_MIN);

    printf(
        "Max of double: %e\n",
        DBL_MAX);
    printf(
        "Min pos of double: %e\n",
        DBL_MIN);

    printf(
        "Max of long double: %e\n",
        LDBL_MAX);
    printf(

```

(continues on next page)

⁶⁸ <https://en.cppreference.com/w/c/types/integer>

(continued from previous page)

```

    "Min pos of long double: %e\n",
    LDBL_MIN);

/* Line separator. */
printf("\n");

printf(
    "Digit of float: %u\n",
    FLT_DIG);
printf(
    "Digit of double: %u\n",
    DBL_DIG);
printf(
    "Digit of long double: %u\n",
    LDBL_DIG);

return 0;
}

```

那麼，如何用 C 語言表示無窮大 (infinity)、負無窮大 (negative infinity) 和非數字 (not a number) 呢？C 語言沒有內建的表示法，但我們可以透過簡單的運算取得這些特殊數字。參考以下範例程式：

```

#include <stdio.h>
#ifndef _MSC_VER
#include <math.h>
#endif

int main(void)
{
    /* Positive infinity. */
#ifndef _MSC_VER
    double PosInf = -log(0.0);
#else
    double PosInf = 1.0 / 0.0;
#endif

    /* Negative infinity. */
#ifndef _MSC_VER
    double NegInf = log(0.0);
#else
    double NegInf = -1.0 / 0.0;
#endif

    /* Not a number. */

```

(continues on next page)

(continued from previous page)

```

#define _MSC_VER
double NaN = nan("NaN");
#else
double NaN = 0.0 / 0.0;
#endif

printf("%e\n", PosInf);
printf("%e\n", NegInf);
printf("%e\n", NaN);

/* Line separator. */
printf("\n");

printf(
    "inf + inf = %e\n",
    PosInf + PosInf);
printf(
    "-inf + -inf = %e\n",
    NegInf + NegInf);
printf(
    "inf + -inf = %e\n",
    PosInf + NegInf);

printf(
    "inf + nan = %e\n",
    PosInf + NaN);

return 0;
}

```

在一些舊的 C 編譯器上，這樣的範例程式會引發程式的錯誤。不過，近年來新的 C 編譯器都不再把除以 0.0 視為程式錯誤，所以可以用這種方法來取得這些特殊數字。

雖然新版的 Visual C++ (MSVC) 支援 C11 及 C17，但為了相容現存程式碼，仍然保持舊編譯器的行為。為了處理 MSVC 的編譯器行為，此處用到前置處理器的語法。我們會在後續的章節談到前置處理器。這裡先知道此範例程式碼會根據 MSVC 和非 MSVC 使用不同的程式碼即可。

複數 (Complex Number) (C99)

在 C99 之前，C 語言沒有原生的複數。當時的手法是自己用其他型別來模擬，像是以下的結構體宣告：

```
struct complex_t {
    double real;
    double imag;
};
```

在此結構體宣告中，real 表示實部，imag 表示虛部。

在 C99 後，C 語言支援原生的複數，我們就不用自己手刻複數型別了。在使用複數時，通常會引入 [complex.h⁶⁹](#) 函式庫，可得到額外的巨集宣告和複數運算相關函式。

以下是一個簡短的範例程式：

```
#include <assert.h>
#include <complex.h>
#include <math.h>

int main(void)
{
    complex double p = 3 + 4 * I;

    assert(3 == creal(p));
    assert(4 == cimag(p));

    complex o = 0 + 0 * I;

    double dx = \
        creal(p) - creal(o);
    double dy = \
        cimag(p) - cimag(o);
    double d = \
        sqrt(pow(dx, 2) + pow(dy, 2));
    assert(5.0 == d);

    return 0;
}
```

在此程式中，我們分別以 `creal()` 函式和 `cimag()` 函式取出複數的實部和虛部，再計算兩複數的距離。

雖然新版的 Visual C++ (MSVC) 支援 C11 和 C17，但為了相容舊程式碼，不提供複數型態。若要考慮 MSVC 相容性的話，上述程式可改寫如下：

⁶⁹ <https://en.cppreference.com/w/c/numeric/complex>

```

#include <assert.h>
#include <complex.h>
#include <math.h>

#ifndef _MSC_VER
#define COMPLEX_T _Dcomplex
#define COMPLEX_NEW(real, imag) ((_Dcomplex){(real), (imag)})
#else
#define COMPLEX_T complex double
#define COMPLEX_NEW(real, imag) ((real) + (imag) * I)
#endif

int main(void)
{
    COMPLEX_T p = COMPLEX_NEW(3, 4);

    assert(3 == creal(p));
    assert(4 == cimag(p));

    COMPLEX_T o = COMPLEX_NEW(0, 0);

    double dx = \
        creal(p) - creal(o);
    double dy = \
        cimag(p) - cimag(o);
    double d = \
        sqrt(pow(dx, 2) + pow(dy, 2));
    assert(5.0 == d);

    return 0;
}

```

這是因為 MSVC 的複數型態是結構體，和標準 C 不相容，所以要用前置處理器來處理歧異性。我們還沒學到前置處理器的使用方式，先知道前置處理器是一種基於字串代換的巨集即可。

字元 (Character)

字元代表單一的字母 (letter) 或符號 (symbol)，像是 'c' 代表英文字母的 c。C 語言中有三種字元型態：

- `char`：用於 ASCII 編碼或其他多字元編碼
- `wchar_t`：寬字元



初學 C 語言時，重點在於核心概念，先會用 `char` 即可。後兩種字元主要用於國際化 (internationalization) 相關議題，一開始不用急著馬上學。

字串 (String)

C 語言沒有真正的字串型別，而是用以零結尾的字元陣列 (null-terminated `char` array) 來表示字串。比起其他的高階語言，這種字串表示法相對低階，處理起來比較瑣碎。

在以下範例程式中，我們用兩種方式來撰寫相同的字串：

```
#include <assert.h>
#include <string.h>

int main(void)
{
    char s1[] = "Hello World";
    char s2[] = \
        {'H', 'e', 'l', 'l', 'o', ' ', \
         'W', 'o', 'r', 'l', 'd', '\0'};

    assert(0 == strcmp(s1, s2));

    return 0;
}
```

在這個範例中，我們直接以字串實字 "Hello World" 對 `s1` 賦值，但刻意用字元陣列來對 `s2` 賦值。以 `strcmp()` 函式比較兩個字串，確認其回傳值為 0，表示兩字串相等。

列舉 (Enumeration)

列舉是由使用者自訂的型態，用來表達有限數量 (finite)、離散的 (discrete) 資料。像是性別 (gender)、星期幾 (day of week) 等。在下列例子中，我們用列舉定義交通號誌：

```
enum traffic_light_t {
    TRAFFIC_LIGHT_GREEN,
    TRAFFIC_LIGHT_YELLOW,
    TRAFFIC_LIGHT_RED
};
```

現行的交通號誌有綠、黃、紅三種，故在本範例中我們定義三個值。

在這個例子中，`TRAFFIC_LIGHT_` 是我們自訂的前綴 (prefix)。因為 C 語言沒有命名空間也沒有物件的概念，所以我們用自訂的前綴來模擬命名空間。這在 C 語言不是強制的，而是一種撰碼風格。

列舉所定義的識別字，在程式中視為一種獨一無二的符號。這些識別字本身的值不是重點，而是取其符號上的意義。

陣列 (Array)

陣列是由使用者定義的線性容器。陣列的特性是 C 語言內建的，但陣列的型別則由使用者來決定。

例如，以下範例宣告一個長度為 5 的整數 (int) 陣列：

```
int arr[] = {1, 2, 3, 4, 5};
```

我們將於後文介紹陣列的使用方式，故這裡不詳談。

結構體 (Structure)

結構體是由使用者定義的複合型態。結構體內部會包含一至多個欄位 (field)，這些欄位有可能是同質或異質的。

例如，下列結構體用來模擬平面座標上的點 (point)：

```
struct point_t {
    double x;
    double y;
};
```

在此結構體宣告中，我們定義了兩個屬性 x 和 y，分別表示點的 x 座標和 y 座標。

聯合體 (Union)

聯合體是另一種由使用者定義的複合型態。聯合體內部包含一至多個欄位 (field)，這些欄位有可能是同質或異質。但聯合體內的屬性是共用的，在同一時間內同一聯合體只能用其欄位中其中一個欄位。

例如，以下聯合體定義了兩個欄位：

```
union data_t {
    double d1;
    int i;
};
```

使用者可選擇使用 d1 或 i，但兩者不能共存。

指標 (Pointer)

指標 (pointer) 用來儲存資料在記憶體中的虛擬位址，在 C、C++、Rust 等系統程式語言中都有指標的概念。這項特性主要的用途是管理記憶體。我們會在後續文章中介紹指標，故此處不重覆說明。

前言

電腦程式大部分的工作都和資料處理相關。在程式語言中，變數 (variable) 相當於資料的標籤，我們可透過變數來操作資料。在本文中，我們介紹在 C 語言中使用變數的方式。

實字 (Literal)

實字是指直接寫死在程式裡的資料。在教學用範例程式裡時常會看到實字，因為從外界讀取資料的程式會比較複雜，直接使用程式內部寫好的資料則簡單得多。

以下是一些實字的例子：

- 12345 (整數)
- 123.456 (浮點數)
- "Hello World" (C 字串)
- {1, 2, 3, 4, 5} (整數陣列)

宣告變數

C 語言的變數包含以下要件：

- 資料型態 (data type)
- 識別字 (identifier)
- 值 (value)

絕大部分的程式語言都有資料型態的概念。資料型別用來規範資料在程式中合理的操作。我們在前一篇文章介紹過資料型態，此處不詳述。

在電腦程式中，編譯器或直譯器透過識別字來了解程式碼的意義。有些識別字是內建的，像是保留字 (keywords)、內建變數、內建函式、標準函式庫的函式等。有些識別字是程式設計者宣告後創造出來的，像是變數。

變數是資料的標籤，而非資料本身。電腦程式很大一部分是在操作資料，變數在本質上是用來操作資料的一種語法特性。

以下變數 `i` 的資料型態是 `int`，值為 3：

```
int i = 3;
```

以下變數 s 的資料型態是指向 char 的指標，值為 C 字串實字 "Hello World"：

```
char *s = "Hello World";
```

請注意 C 語言沒有原生的字串型態。我們會在後續文章中介紹 C 語言的字元和字串，這裡先觀察一下變數的宣告方式即可。

使用變數

我們以簡短的例子來看如何使用變數：

```
#include <assert.h>          /* 1 */
#include <stdio.h>           /* 2 */

int main(void)               /* 3 */
{
    int a = 4;                /* 4 */
    int b = 3;                /* 5 */

    printf("4 + 3 = %d\n", a + b); /* 6 */
    printf("4 - 3 = %d\n", a - b); /* 7 */

    assert(a == 4);           /* 8 */
    assert(b == 3);           /* 9 */

    return 0;                  /* 10 */
}
```

我們在第 5 行和第 6 行宣告整數型態變數 a 和另一個整數型態變數 b，並分別對兩變數賦值。變數對於 C 程式來說，是由程式設計者自行定義的，所以要先宣告才能使用。

接著，我們在第 7 行和第 8 行使用兩變數進行加法和減法運算，並將運算當成參數結果傳入 printf() 函式。實際的效果是在終端機印出運算結果。

最後，我們在第 9 行和第 10 行用 assert() 巨集確認變數 a 和變數 b 在運算過程中都沒有實質的變化。因為我們沒有把運算的結果回存到 a 或 b 上。

在這個簡短的例子中，我們重覆使用 a 和 b 多次。其實這兩個變數本身只是一個符號，我們透過這兩個變數操作整數形態資料 4 和 3。

宣告常數 (Constant)

我們在宣告變數後，無法預防變數遭到預期外的修改。若確認某變數的值在整個程式中不會變化，可以加上 `const` 保留字來修飾該變數，這時候該變數視為常數。我們若在程式中修改常數，C 編譯器會引發錯誤，阻止程式編譯，藉此提醒程式設計者該修正程式碼了。

參考以下反例：

```
const double pi = 3.1415927;
pi = 4.0; /* Error! */
```

我們宣告了型別為 `double` 的變數 `pi`，由於我們預期 `pi` 的值是定值，故加上 `const` 修飾 `pi`。事後我們意圖修改 `pi` 的值時，引發該程式的錯誤，這時候我們就知道程式碼出問題了。

由於 C 標準函式庫 `math.h` 已經定義了較精準的 `pi` 常數 `M_PI`，我們不應該另行宣告一個常數 `pi`。這裡只是展示 `const` 的用法。

合理的變數名稱

變數命名有固定的條件：

- 第一個字元是英文字母或底線
- 第二個以後的字元是英文字母、底線或數字

由於 C 語言會把一些新的保留字用首字底線的方式來命名，像 `_Bool`，不建議用首字底線來命名變數。除此之外，識別字名稱有一些風格上的建議，詳見下一節。

識別字的命名風格

撰碼風格 (coding style) 是讓程式碼易讀的一種撰碼方式，這是建議事項，而非強制的。識別字命名風格也是撰碼風格的一環。以下是常見的識別字命名風格：

- PascalCase
- camelCase
- snake_case
- kebab-case (C 語言無法使用)

Pascal case 和 Camel case 在 C 語言較少使用，這種風格主要在 Java 或 C# (C sharp) 圈子中流行。

C 語言常用 snake case 或是縮寫來命名變數或函式。像是 `isalnum()`⁷⁰ 函式用來檢查某個字元是否為字串或數字。由 `gtk_init()`⁷¹ 的名稱可知，這是一個用來初始化的函式，並在函式名稱使用 `gtk_` 前綴來模擬命名空間 (namespace)。

C 語言無法使用 kebab case，這種風格主要用在 Lisp 家族語言。

保留字 (Keywords)

保留字在程式碼中有特別的意義，故不能用來命名變數。我們在本節中列出 C 語言保留字的清單⁷²。

ANSI C 的保留字：

```
auto      break     case      char
const     continue  default   do
double    else      enum      extern
float     for       goto     if
int       long      register return
short     signed    sizeof    static
struct    switch   typedef  union
unsigned  void      volatile while
```

C99 新增的保留字：

```
inline    restrict _Bool      _Complex
           _Imaginary
```

C11 新增的保留字：

```
_Alignas _Alignof _Atomic   _Generic
_Noreturn        _Static_assert
_Thread_local
```

原本的 C 語言使用小寫字當成保留字，現代 C 語言為了保持現有程式的相容性，新的保留字改以首字底線來命名，再加上選擇性的巨集宣告。

不要刻意背誦保留字，要在學習程式設計的過程中自然地學會。此外，好的程式碼編輯器會用顏色提示保留字所在的地方，我們可以很容易地看到保留字所在的位置。

⁷⁰ <https://en.cppreference.com/w/c/string/byte/isalnum>

⁷¹ <https://developer.gnome.org/gtk3/stable/gtk3-General.html#gtk-init>

⁷² <https://en.cppreference.com/w/c/keyword>

可視度 (Scope)

可視度是一個隱含在程式碼中的概念。由小至大分為以下數種：

- 區塊可見 (block scope)
- 函式可見 (function scope)
- 檔案可見 (file scope)
- 全域可見 (global scope)

例如，下列整數型態變數 `n` 的可視度是函式可見：

```
int main(void)
{
    int n = 3;

    /* Use `n` here. */

    return 0;
}
```

下列無號整數型態變數 `i` 的可視度則是區塊可見：

```
#include <stddef.h>

int main(void)
{
    {
        size_t i;
        for (i = 0; i < 10; ++i) {
            /* Repeat something here. */
        }
    }

    /* `i` is not visible here. */

    return 0;
}
```

實際上，上述範例程式碼是在 **ANSI C** 中使用 `for` 迴圈的計數器的小技巧。在 **C99** 中則會改寫如下：

```
#include <stddef.h>

int main(void)
{
```

(continues on next page)

(continued from previous page)

```
for (size_t i = 0; i < 10; ++i) {
    /* Repeat something here. */
}

/* `i` is not visible here. */

return 0;
}
```

剛開始學習 C 語言的時候，程式碼很簡短，不會感受到可視度的重要。當程式變長後，可視度的重要性會慢慢浮現出來。簡單的原則是僅開放最低限度的可視度，最好不要使用全域變數。

運算子 (OPERATORS)

前言

在程式語言中，運算子多以符號表示，通常都無法再化約成更小的單位，所以運算子可視為該語言的基礎指令。本文介紹 C 語言的運算子。

代數運算子

代數運算子用來進行日常的十進位代數運算。包括以下運算子：

- +：相加
- -：相減
- *：相乘
- /：相除
- %：取餘數

以下是簡短範例：

```
#include <assert.h>

int main(void)
{
    assert(4 + 3 == 7);
    assert(4 - 3 == 1);
    assert(4 * 3 == 12);
    assert(4 / 3 == 1);
    assert(4 % 3 == 1);

    return 0;
}
```

許多 C 語言教材會用 `printf` 將資料輸出終端機，但我們刻意用 `assert` 巨集檢查運算結果是否正確。因為透過 `assert` 巨集可自動檢查程式是否正確，但使用 `printf` 函式得手動檢查。而且 `assert` 敘述可以表明我們的意圖。

除了上述運算子，還有遞增和遞減運算子：

- ++
- --

遞增、遞減運算子會帶著隱性的狀態改變，在使用時要注意。以下是範例：

```
#include <assert.h>

int main(void)
{
    unsigned n = 3;

    assert (++n == 4);
    assert (n++ == 4);
    assert (n == 5);

    return 0;
}
```

在這個範例中，第一個 assert 敘述使用 `++n`，會先加 1 再比較值。但第二個 assert 敘述使用 `n++`，會先比較值才加 1。

由於遞增、遞減運算子會造成隱性的狀態改變，應儘量把敘述寫簡單一些，以免發生預期外的結果。

二元運算子

二元運算子用在二進位數運算。包括以下運算子：

- `&` : bitwise AND
- `|` : bitwise OR
- `^` : bitwise XOR
- `~` : 取補數
- `<<` : 左移 (left shift)
- `>>` : 右移 (right shift)

二元運算有其規則。以下是 `&` (bitwise AND) 的運算規則：

p	q	p & q
1	1	1
1	0	0
0	1	0
0	0	0

歸納起來，就是「兩者皆為真時才為真」。

以下是 `|` (bitwise OR) 的運算規則：

p	q	$p \mid q$
1	1	1
1	0	1
0	1	1
0	0	0

歸納起來，就是「兩者任一為真即為真」。

以下是 \wedge (bitwise XOR) 的運算規則：

p	q	$p \wedge q$
1	1	0
1	0	1
0	1	1
0	0	0

以下是 \sim (取補數) 的規則：

p	$\sim p$
1	0
0	1

只看規則會覺得有點抽象，所以我們補上範例程式，並把運算的過程寫在註解裡：

```
#include <assert.h>

int main(void)
{
    int a = 3; /* 0000 0011 */
    int b = 5; /* 0000 0101 */

    /*      0000 0011
     * &) 0000 0101
     * -----
     *      0000 0001 */
    assert(1 == (a & b));

    /*      0000 0011
     * |) 0000 0101
     * -----
     *      0000 0111 */
    assert(7 == (a | b));

    /*      0000 0011
     * ~) 0000 0001
     * -----
     *      1111 1110 */
    assert(22 == (~a));
}
```

(continues on next page)

(continued from previous page)

```

    ^) 0000 0101
    -----
      0000 0110 */
assert(6 == (a ^ b));

/* <<) 0000 0101
-----
      0000 1010 */
assert(10 == (b << 1));

/* >>) 0000 0101
-----
      0000 0010 */
assert(2 == (b >> 1));

return 0;
}

```

二進位運算比十進位運算不直觀，但速度較快。故二進位運算多用於注意速度的低階運算，平常用不太到。

比較運算子

比較運算子用來比較兩個值之間的相對關係。包含以下運算子：

- ==：相等
- !=：相異
- >：大於
- >=：大於或等於
- <：小於
- <=：小於或等於

以下是簡短的範例程式：

```
#include <assert.h>

int main(void)
{
    assert(4 == 4);
    assert(4 != 5);
}
```

(continues on next page)

(continued from previous page)

```

assert(5 > 4);
assert(5 >= 4);

assert(3 < 4);
assert(3 <= 4);

return 0;
}

```

要注意比較運算子只能用來比較基礎型別，而且要兩者間相容。對於其他型態的資料，要自行撰寫比較函式。像是標準函式庫中的 `string.h` 函式庫就有比較 C 字串的 `strcmp()`⁷³ 函式。其範例如下：

```

#include <assert.h>
#include <string.h>

int main(void)
{
    assert(strcmp("C", "C++") < 0);
    assert(strcmp("C", "Ada") > 0);

    return 0;
}

```

邏輯運算子

邏輯運算子用來進行布林運算。以下是 C 語言的邏輯運算子：

- `&&` : logic AND
- `||` : logic OR
- `!` : logic NOT

C 語言雖然沒有真正的布林數，仍然有布林語境，故仍可進行布林運算。

邏輯運算子和二元運算子規則類似，符號也有點類似，要注意不要寫錯。

以下是 `&&` (logic AND) 的運算規則：

⁷³ <https://en.cppreference.com/w/c/string/byte/strcmp>

p	q	$p \&& q$
true	true	true
true	false	false
false	true	false
false	false	false

以下是 `||` (logic OR) 的運算規則：

p	q	$p \parallel q$
true	true	true
true	false	true
false	true	true
false	false	false

以下是 `!` (logic NOT) 的運算規則：

p	$!p$
true	false
false	true

我們使用 **C99** 引入的 `stdbool.h` 函式庫做一些基礎布林運算：

```
#include <assert.h>
#include <stdbool.h>

int main(void)
{
    assert(
        (true && true) == true);
    assert(
        (true && false) == false);
    assert(
        (false && true) == false);
    assert(
        (false && false) == false);

    assert(
        (true || true) == true);
    assert(
        (true || false) == true);
    assert(
        (false || true) == true);
    assert(
        (false || false) == false);
}
```

(continues on next page)

(continued from previous page)

```
(false || false) == false);

assert((!true) == false);
assert((!false) == true);

return 0;
}
```

要注意 `true` 和 `false` 都是巨集宣告，這個範例不代表 C 語言有布林數。

指派運算子

指派運算子分為一般指派運算和複合指派運算。一般指派運算就是單純的賦值。複合指派運算則是小小的語法糖。C 語言包含以下指派運算子：

- `=`：一般賦值
- `+=`：相加後賦值
- `-=`：相減後賦值
- `*=`：相乘後賦值
- `/=`：相除後賦值
- `%=`：取餘數後賦值

以下是簡短的範例：

```
#include <assert.h>

int main(void)
{
    unsigned short n = 4;

    n += 3;
    assert(7 == n);

    return 0;
}
```

在這個例子中，`n` 原本是 4，後來經 `+=` (相加後賦值) 變成 7。

三元運算子

三元運算子類似於小型的 if 敘述。其虛擬碼如下：

```
condition ? a : b
```

但三元運算子是表達式，而不是敘述。所以，三元運算子會回傳值。

以下範例用三元運算子求得兩值中較大者：

```
#include <assert.h>

int main(void)
{
    unsigned max = 5 > 3 ? 5 : 3;

    assert(max == 5);

    return 0;
}
```

其他運算子

以下是一些未歸類的運算子：

- sizeof：取得資料或型態的寬度
- *：宣告指標變數、取值
- &：取址
- .：取結構體的欄位
- ->：取結構體指標的欄位，是一種語法糖

以下實例用 sizeof 求得不同資料型態的寬度：

```
#include <stdbool.h>
#include <stdio.h>

int main(void)
{
    printf(
        "Size of bool: %u\n",
        sizeof(true));
```

(continues on next page)

(continued from previous page)

```

printf(
    "Size of char: %u\n",
    sizeof('c'));

printf(
    "Size of short: %u\n",
    sizeof((short) 3));
printf(
    "Size of int: %u\n",
    sizeof(3));
printf(
    "Size of long: %u\n",
    sizeof(31));
printf(
    "Size of long long: %u\n",
    sizeof(311));

printf(
    "Size of float: %u\n",
    sizeof(3.4f));
printf(
    "Size of double: %u\n",
    sizeof(3.4));
printf(
    "Size of double: %u\n",
    sizeof(3.4l));

return 0;
}

```

`sizeof` 長得很像函式，但是 `sizeof` 是運算子，需注意。

運算子優先順序

如果同一行敘述內用到多個運算子時，要如何確認那個運算子會先運算呢？正統的方法是透過查詢運算子優先順序⁷⁴來確認。

實際上程式設計者甚少背誦運算子優先順序。因為：

- 運算子的優先順序和數學的概念相同
- 藉由簡化敘述來簡化運算子的使用
- 適度使用括號來改變運算子優先順序

⁷⁴ https://en.cppreference.com/w/c/language/operator_precedence

即使自己很熟運算子的優先順序，也不保證團隊成員都很熟。我們應該要撰寫簡單明瞭的敘述，減少大家對程式碼的猜測。

選擇控制結構 (SELECTION CONTROL STRUCTURE)

前言

選擇控制結構的用途在於選擇性地執行某段程式碼區塊的指令。只有在符合該控制結構內的條件句時才執行該控制結構的區塊內的指令。藉由選擇控制結構，我們可以根據情境控制某段程式碼的執行與否。

C 語言有 `if` 和 `switch` 兩種選擇控制結構。我們會在本文中學習如何使用。

`if` 敘述

`if` 最簡單的形式是單元敘述，只有符合特定條件時才執行該區塊內的指令。將這概念寫成 C 虛擬碼如下：

```
if (condition) {  
    statement;  
}
```

在本例中，只有 `condition` 為真時，才會執行 `statement`。實際上 `statement` 可能是多行敘述，這裡為了簡化只寫成單行敘述。

除了前述形式外，我們還可以選擇性加上 `else` 區塊，這時候會形成二元敘述。將其寫成 C 虛擬碼如下：

```
if (condition) {  
    statement_a;  
}  
else {  
    statement_b;  
}
```

在本例中，當 `condition` 為真時，會執行 `statement_a`。反之，則執行 `statement_b`。

我們還可以加上一至多個 `else if` 區塊，這時候會形成多元敘述。將其寫成 C 虛擬碼如下：

```
if (condition_a) {  
    statement_a;  
}  
else if (condition_b) {  
    statement_b;
```

(continues on next page)

(continued from previous page)

```

}
else {
    statement_c;
}

```

在本例中，當 *condition_a* 為真時，執行 *statement_a*。而 *condition_b* 為真時，執行 *statement_b*。當 *condition_a* 及 *condition_b* 皆不為真時，執行 *statement_c*。

注意在多元敘述中，條件的先後是有意義的。以本例來說，只要 *condition_a* 為真，不論 *condition_b* 是否為真，皆會執行 *statement_a* 後即離開該 *if* 敘述，而不會執行 *statement_b*。

我們先前提過，*else* 是選擇性的。所以，下列 C 虛擬碼是合理的：

```

if (condition_a) {
    statement_a;
}
else if (condition_b) {
    statement_b;
}
else if (condition_c) {
    statement_c;
}

```

這時候，有可能因 *condition_a*、*condition_b*、*condition_c* 皆不符合，而該 *if* 敘述沒有執行任何指令。

如同先前的例子，條件的先後順序是有意義的，我們需根據程式的需求安排條件的先後。

我們已經看了不少虛擬碼，來看一個實例：

```

#include <stdlib.h>
#include <stdio.h>
#include <time.h>

int main(void)
{
    /* Set a random seed according to
       current system time. */
    srand((unsigned) time(NULL));

    /* Get a random number
       between 0 and 10. */
    unsigned short num = rand() % 11;

    /* Show some message according to
       the value of `num`. */
    if (num == 0) {

```

(continues on next page)

(continued from previous page)

```

        printf("Zero\n");
    }
else if (num % 2 == 0) {
    printf("%d is even\n", num);
}
else {
    printf("%d is odd\n", num);
}

return 0;
}

```

一開始我們會以系統目前時間做為參數放入 `srand()`⁷⁵ 函式以生成亂數種子 (random seed)，這是生成亂數時常見的手法。因為每次執行該程式時，系統時間皆相異，可確保亂數種子的隨機性。

接著，我們呼叫 `rand()`⁷⁶ 函式以取得隨機整數。在預設情形下，`rand()` 函式會回傳一個 0 至 `RAND_MAX`⁷⁷ 之間的整數。常見的手法是用取餘數 (modulus) 的方式限縮亂數的範圍。

最後，我們寫一段 `if` 敘述，該敘述會根據 `num` 的值輸出不同的訊息。細節請讀者自行閱讀。

switch 敘述

在多重選擇時，雖然可以直接用 `if` 敘述來寫，用 `if` 敘述來寫的程式碼會比較長。相對來說，`switch` 敘述可簡化多重選擇的程式碼。其虛擬碼如下：

```

switch (expression) {
case a:
    statement_a;
    break;
case b:
    statement_b;
    break;
case c:
    statement_c;
    break;
default:
    statement;
}

```

⁷⁵ <https://en.cppreference.com/w/c/numeric/random/srand>

⁷⁶ <https://en.cppreference.com/w/c/numeric/random/rand>

⁷⁷ https://en.cppreference.com/w/c/numeric/random/RAND_MAX

expression 可放入單一值或是表達式。當 *expression* 的值為 *a* 時，執行 *statement_a*。當 *expression* 的值為 *b* 時，執行 *statement_b*。當 *expression* 的值為 *c* 時，執行 *statement_c*。若所有值皆不符合時，執行 *default* 所在的子區塊內的指令。

同樣地，*case* 區塊的先後順序是有意義的。在本例中，當 *expression* 符合 *a* 時，不論 *expression* 是否符合 *b* 或 *c*，皆會執行 *statement_a* 後離開 *switch* 敘述。所以，我們要根據情境適當地安排 *case* 區塊的順序。

default 區塊是選擇性的，所以下列 *switch* 虛擬碼是合理的：

```
switch (expression) {  
    case a:  
        statement_a;  
        break;  
    case b:  
        statement_b;  
        break;  
    case c:  
        statement_c;  
        break;  
}
```

要注意 *switch* 敘述內的 *break* 敘述不可任意省略。若 *break* 敘述省略掉，在執行完該 *case* 敘述完後會繼續執行下一段 *case* 敘述，直到碰到 *break* 敘述為止。這種特性稱為貫穿 (fallthrough)。我們來看一段 *switch* 虛擬碼：

```
switch (expression) {  
    case a:  
    case b:  
        statement_b;  
    case c:  
        statement_c;  
    case d:  
        statement_d;  
        break;  
}
```

在這個例子中，當 *expression* 等於 *a* 或 *b* 時，會依序執行 *statement_b*、*statement_c*、*statement_d*。因為前兩個區塊沒有用 *break* 中斷程式的執行。同理，當 *expression* 等於 *c* 時，則會依序執行 *statement_c* 和 *statement_d*。

貫穿可能是我們預期的效果或是不小心漏寫 *break* 敘述所造成的 bug，需注意。

我們已經看了不少虛擬碼，現在來看實例：

```
#include <stdio.h>  
#include <time.h>
```

(continues on next page)

(continued from previous page)

```

int main(void)
{
    /* Get the object
       of current time. */
    time_t t = time(NULL);
    struct tm *now = localtime(&t);

    /* Show some message
       according to
       current day of week. */
    switch (now->tm_wday) {
        case 0: /* Sunday. */
            /* Fallthrough. */
        case 6: /* Saturday. */
            printf("Weekend\n");
            break;
        case 5: /* Friday. */
            printf(
                "Thank God. "
                "It's Friday.\n");
            break;
        case 3: /* Wednesday. */
            printf("Hump day\n");
            break;
        case 1: /* Monday. */
            printf("Monday blue\n");
            break;
        default:
            /* All other days of week. */
            printf("Week\n");
            break;
    }

    return 0;
}

```

我們先取得代表目前系統時間的物件 `t`，再將該物件轉為代表時間的結構體 `now`。這是為了從時間物件中取出當時的日期 (day of week)。

接著，我們用一個 `switch` 敘述來檢查 `now` 的 `tm_wday` 屬性，根據不同日期吐出不同的訊息。實作細節請讀者自行閱讀。

讀者可試著把該範例用 `if` 敘述改寫，就可以體會 `if` 和 `switch` 兩者的差異。我們把這段改寫過程留給讀者自行練習，不展示其程式碼。

由於 `case` 敘述的對象一定要是整數，所以並非所有的 `if` 敘述都可換成等效的 `switch` 敘述。反之，所有的 `switch` 敘述都可換成等效的 `if` 敘述。

附帶一提，在優化 C 程式的觀點上，使用 `switch` 敘述會比使用等效的 `if` 敘述來得好一些。因為 `if` 敘述需要逐一地針對每項條件句去測試，但 `switch` 敘述可自動跳躍至符合條件的子區塊。如果很在意程式效能的話，應在能用 `switch` 敘述的地方就不要用 `if` 敘述。

迭代控制結構 (ITERATION CONTROL STRUCTURE)

前言

在電腦程式中，迴圈 (loop) 或迭代控制結構 (iteration control structure) 用來反覆執行同一段程式碼區段。

迴圈可透過條件句 (conditional)、計數器 (counter)、迭代器 (iterator) 等方式來決定執行的次數。C 語言內建前兩種形式的迴圈，迭代器則要自己寫。由於迭代器算是資料結構的項目之一，本文不會談到。

C 語言有 `while` 和 `for` 兩種形式的迴圈，我們會在本文中介紹。

`while` 敘述

`while` 的使用方式

`while` 是以條件句來控制迭代的迴圈敘述。以下是 `while` 的 C 語言虛擬碼：

```
while (condition) {
    statement;
}
```

只有在 *condition* 為真時，才會執行 `while` 區塊內的程式碼。

以下是簡短的實例：

```
#include <stdio.h>

int main(void)
{
    unsigned i = 1;

    while (i <= 10) {
        printf("%u\n", i);

        i += 1;
    }

    return 0;
}
```

在這個範例中，我們用 `while` 迴圈控制變數 `i` 的終止狀態，在每次迭代時印出變數 `i` 的值。

實際範例

我們來看一個較長的例子。這個例子會印出反白的菱形：

此範例程式的參考程式碼如下：

```
#include <stddef.h>          /* 1 */
#include <stdio.h>           /* 2 */

int main(void)                /* 3 */
{
    size_t i = 0;              /* 4 */
    size_t j = 0;              /* 5 */

    while (i <= 20) {          /* 6 */
        size_t j = 0;          /* 7 */

        if (i <= 10) {          /* 8 */
            while (j <= 20) {  /* 9 */
                if (10 - i <= j  /* 10 */
                     && j <= 10 + i) { /* 11 */
                    /* ... */
                }
            }
        }
    }
}
```

(continues on next page)

(continued from previous page)

```

        printf(" ");
        /* 12 */
    }
    /* 13 */
else {
    printf("*");
    /* 14 */
}
/* 15 */
/* 16 */

    ++j;
}
/* 17 */
/* 18 */
/* 19 */
else {
    /* 20 */
while (j <= 20) {
    if (i - 10 <= j
        && j <= 21 - i + 9) {
        printf(" ");
        /* 21 */
        /* 22 */
        /* 23 */
        /* 24 */
    }
    /* 25 */
else {
    printf("*");
    /* 26 */
    /* 27 */
}
/* 28 */

    ++j;
}
/* 29 */
/* 30 */
/* 31 */

printf("\n");
/* 32 */

++i;
/* 33 */
/* 34 */

return 0;
/* 35 */
/* 36 */
}

```

在這個例子中，整個程式置於一個大的 `while` 迴圈中，此 `while` 迴圈位於第 6 行至第 34 行。

在這個 `while` 迴圈中，菱形上半部和下半部各由不同的 `while` 迴圈來控制。上半部的 `while` 迴圈位於第 8 行至第 19 行。下半部的 `while` 迴圈位於第 20 行至第 31 行。共通的部分位於第 32 行和第 33 行。

在每一次迭代中，我們會印出一行符號，所以需要用一個內部 `while` 迴圈印出一行符號。每個「游標」印出的字元可能是 " " (空白) 或 "*" (星號)。另外，在每行結束時要印出換行符號，讓「游標」移動到下一行。實作細節請讀者自行追蹤程式碼。

do { ... } while 敘述

do { ... } while 的使用方式

do { ... } while 是 while 的變體。其 C 虛擬碼如下：

```
do {
    statement;
} while (condition);
```

do { ... } while 的使用方式和 while 相似，差別在於無論 *condition* 是否為真，do { ... } while 至少會執行一次迭代。

實際範例

以下是實際的例子：

```
#include <stddef.h> /* 1 */
#include <stdio.h> /* 2 */

int main(void)
{
    char cmd = 'q';

RESTART:
    do {
        printf(
            "===== "
            "System Menu"
            " ======\n");
        printf(" New File (n)\n"); /* 9 */
        printf(" Open File (o)\n"); /* 10 */
        printf(" Set Preference (p)\n"); /* 11 */
        printf(" Quit (q)\n"); /* 12 */
        printf(
            "Select next action "
            "(default to q): "); /* 13 */

        size_t i = 0; /* 14 */
        char c; /* 15 */
        while ((c = getchar()) != '\n') { /* 16 */
            if (i > 1) { /* 17 */
                printf("Invalid command\n"); /* 18 */
                printf(

```

(continues on next page)

(continued from previous page)

```

"=====
"=====\\n\\n"); /* 19 */
cmd = 'q'; /* 20 */

/* Clear the buffer. */
while (getchar() != '\\n'); /* 21 */

goto RESTART; /* 22 */
} /* 23 */

cmd = c; /* 24 */
i++; /* 25 */
} /* 26 */

switch (cmd) { /* 27 */
case 'n': /* 28 */
printf("Create a new file\\n"); /* 29 */
break; /* 30 */
case 'o': /* 31 */
printf("Open a old file\\n"); /* 32 */
break; /* 33 */
case 'p': /* 34 */
printf(
    "Set system preference\\n"); /* 35 */
break; /* 36 */
case 'q': /* 37 */
printf("Quit the system\\n"); /* 38 */
break; /* 39 */
default: /* 40 */
printf(
    "Unknown command: %c\\n",
    cmd); /* 41 */
} /* 42 */

printf(
"=====
"=====\\n\\n"); /* 43 */
} while (cmd != 'q'); /* 44 */

return 0; /* 45 */
} /* 46 */

```

在這個例子中，我們用 `do { ... } while` 來模擬系統選單。整個程式由一個大的 `do { ... } while` 迴圈組成。該迴圈位於第 7 行至 44 行。

該迴圈分為三個部分。第一個部分為系統選單提示訊息，位於第 8 行至第 13 行。第二個

部分用於接收使用者輸入，位於第 14 行至第 26 行。最後一個部分則用於回饋使用者的輸入，位於第 27 行至第 43 行。

為了簡化範例，此程式沒有實際的功能，用終端機訊息來取代。

for 敘述

for 的使用方式

for 迴圈使用計數器來迭代。其虛擬碼如下：

```
for (init; end; next) {
    statement;
}
```

其實 for 迴圈分為四個區塊，在一對 (和) (中括號) 內有 *init* 、*end* 、*next* 三個區塊，而一對 { 和 } (大括號) 內則為每次迭代時要執行的指令。

init 、*end* 、*next* 三個區塊內放置和計數器相關的程式碼。*init* 區塊將計數器初始化，*end* 區塊決定迭代結束的條件，*next* 區塊將計數器遞增或遞減。

在 C99 後，可以在 *init* 區塊內建立新變數，所以可以看到以下片段：

```
for (int i = 0; i < 10; ++i) {
    /* Repeat some code here. */
}
```

在這段程式碼中，計數器為 *i*。該變數的起始值為 0，該變數小於 10 時可繼續迭代，每輪迭代 *i* 會遞增 1。

在 C89 時，不能在 *init* 區塊內建立新變數，所以會用以下替代性寫法：

```
{
    int i;
    for (i = 0; i < 10; i++) {
        /* Repeat some code here. */
    }
}
```

在這段程式碼中，外部區塊用來建立新的局部可視度 (local scope)。變數 *i* 在離開該區塊後會自動消失。這個手法的好處在於程式設計者可在不同 for 迴圈中重覆使用變數 *i*，不需要為每個迴圈建立新的變數。

實際範例

接著，我們來看一個較長的範例。該範例會印出以下 ASCII 圖形：

A decorative border consisting of a repeating pattern of five-pointed stars arranged in a grid-like fashion, creating a frame around the page.

該範例的參考程式碼如下：

```
#include <stddef.h> /* 1 */
#include <stdio.h> /* 2 */

int main(void) /* 3 */
{
    for (size_t i = 0; /* 4 */
         i <= 20; /* 5 */
         ++i) { /* 6 */
        if (i <= 10) /* 7 */
            for (size_t j = 0; /* 8 */
                  j < 10 - i; /* 9 */
                  ++j) /* 10 */
                printf(" "); /* 11 */
        printf("\n"); /* 12 */
    } /* 13 */
    for (size_t j = 0; /* 14 */
         j < 2 * i + 1; /* 15 */
         ++j) /* 16 */
        printf("*"); /* 17 */
}
```

(continues on next page)

(continued from previous page)

```

else {                                /* 18 */
    for (size_t j = 0;                /* 19 */
          j < i - 10;                 /* 20 */
          ++j)                      /* 21 */
        printf(" ");                /* 22 */

    for (size_t j = 0;                /* 23 */
          j < 2 * (20 - i) + 1;      /* 24 */
          ++j)                      /* 25 */
        printf("*");                /* 26 */
    }

    printf("\n");                    /* 28 */
}

return 0;                                /* 30 */
}                                         /* 31 */

```

此範例程式放在一個大的 `for` 迴圈當中，該迴圈的程式碼位於第 5 行至第 29 行間。

該迴圈內部分為上下兩部分，分別繪製半邊的圖形。上半部的程式碼位於第 8 行至第 17 行。下半部的程式碼位於第 18 行至第 27 行。共通的部分位於第 28 行。

在每個「游標」中，會根據「游標」所在的位置決定要繪出 " " (空白) 或 "*" (星號)。此外，在每行結束時，要繪製換行符號，讓「游標」移動到下一行。實作細節請讀者自行閱讀程式碼。

break 敘述

`break` 敘述的用途在於提早結束迴圈。由於單獨使用 `break` 敘述意義不大，多會搭配選擇控制結構來使用。以下是簡短的實例：

```

#include <stdio.h>

int main(void)
{
    for (size_t i = 1; i <= 10; i++) {
        if (i > 5)
            break;

        printf("%zu\n", i);
    }
}

```

(continues on next page)

(continued from previous page)

```
    return 0;
}
```

在這個例子中，當 `i` 大於 5 時，會執行 `break` 敘述，中斷此 `for` 迴圈。

continue 敘述

`continue` 敘述用來略去該次迭代剩下的指令，直接進入下一輪迭代。單獨使用 `continue` 敘述意義不大，故會搭配選擇控制結構來使用。以下是簡短的範例：

```
#include <stdio.h>

int main(void)
{
    for (size_t i = 1; i <= 10; i++) {
        if (i % 2 != 0)
            continue;

        printf("%zu\n", i);
    }

    return 0;
}
```

當 `i` 不為偶數時，會執行 `continue` 敘述，就不會印出 `i` 的值。實際的效果就是只印偶數。

goto 敘述

`goto` 敘述可以在同一函式內任意跳躍到標籤 (label) 所在的位置，算是最自由的控制結構。但濫用 `goto` 敘述易寫出難以維護的程式碼，故應謹慎使用。適合使用 `goto` 敘述的場合包括模擬錯誤處理或是釋放系統資源等。

(選讀實例) 終極密碼

在看完各種控制結構的用法後，我們用一個稍長的範例程式來展示如何使用控制結構。本節的範例程式是終極密碼⁷⁸，這是一個常見的小遊戲，很常當成程式設計的範例題目。

以下是範例程式的程式碼：

```
#include <stdbool.h>          /* 1 */
#include <stdio.h>            /* 2 */
#include <stdlib.h>            /* 3 */
#include <time.h>              /* 4 */

int main(void)                /* 5 */
{
    srand((unsigned) time(NULL)); /* 6 */

    int max = 100;               /* 8 */
    int min = 1;                 /* 9 */

    /* Compute a random answer
       between min and max. */
    int answer = \
        rand() % (max - min + 1) + min; /* 10 */

    int guess;                  /* 11 */

    /* Keep the game running until
       the user answers correctly. */
    while (true) {               /* 12 */
        bool getGuess = false;   /* 13 */

        /* Keep prompting until
           the user inputs a
           valid guess. */
        while (!getGuess) {      /* 14 */
            printf(               /* 15 */
                "Please input a number"
                " between %d and %d: ", /* 16 */
                min, max);           /* 17 */

            if (scanf(" %d", &guess)) { /* 19 */
                getGuess = true;     /* 20 */

                /* Check whether
                   the number is valid. */
            }
        }
    }
}
```

(continues on next page)

⁷⁸ <https://zh.wikipedia.org/wiki/%E7%B5%82%E6%A5%B5%E5%AF%86%E7%A2%BC>

(continued from previous page)

```

if (getchar() != '\n') { /* 21 */
    printf( /* 22 */
        "Not a valid number\n"); /* 23 */

    /* Clean the buffer. */ /* 24 */
    while (getchar() != '\n'); /* 25 */

    getGuess = false; /* 26 */
}
} /* 28 */

/* Check for
   non-number string. */
else { /* 29 */
    /* Clean the buffer. */ /* 30 */
    while (getchar() != '\n'); /* 31 */

    printf("Not a number\n"); /* 32 */
}

/* 33 */

if (getGuess
    && (guess < min
        || max < guess)) { /* 34 */
    printf( /* 35 */
        "Invalid number: %d\n",
        guess);
    getGuess = false; /* 36 */
}
} /* 37 */
} /* 38 */
} /* 39 */
} /* 40 */
} /* 41 */
} /* 42 */

/* Compare the guess
   with our answer. */
if (guess == answer) { /* 43 */
    printf("You got it\n");
    break;
}
} /* 44 */
else if (guess > answer) { /* 45 */
    printf("Too large\n");
    max = guess;
}
} /* 46 */
else {
    printf("Too small\n");
    min = guess;
}
} /* 47 */
} /* 48 */
} /* 49 */
} /* 50 */
} /* 51 */
} /* 52 */
} /* 53 */
} /* 54 */
} /* 55 */
}

```

(continues on next page)

(continued from previous page)

```
return 0;          /* 56 */  
}                  /* 57 */
```

在第 7 行中，範例程式用系統時間建立亂數種子。由於每次執行時系統時間會相異，產生的亂數也會相異，可造成隨機的效果。

在第 10 行中，範例程式建立隨機 answer。程式使用者無法預先知道 answer 的值。

整個遊戲迴圈位在第 12 行至第 55 行。該遊戲迴圈分為兩部分。前半部負責接收及檢查使用者輸入，位於第 14 行至第 42 行。後半部負責檢查 guess 和 answer 是否相符，然後給予相對應的回饋。後半段程式碼位於第 43 行至第 54 行。

輸入資料時，可能有四種情境：

- 符合範圍的整數
- 不符合範圍的整數
- 浮點數
- 非數字字串

除了第一種情境外，其他的輸入都是不合法的 (invalid)。當輸入不合法時，會請使用者重新輸入，直到使用者輸入合法的數值。

根據 guess 和 answer 的關係，可能有三種情境：

- 兩者相等
- guess 大於 answer
- guess 小於 answer

第一種情境發生時，代表遊戲結束，將迴圈中止。除此之外，會回饋相對應的提示，更改變數的範圍，然後繼續下一輪遊戲。

指標 (POINTER) 和記憶體管理 (MEMORY MANAGEMENT)

前言

指標 (pointer) 是 C 語言的衍生型別之一。指標的值並非資料本身，而是另一塊記憶體的虛擬位址 (virtual address)。我們可利用指標間接存該指標所指向的記憶體的值。

在 C 語言中，有些和記憶體相關的操作必需使用指標，實作動態資料結構時也會用到指標，所以我們學 C 時無法避開指標。

指標在現實生活中沒有直接對應的概念，是純粹電腦程式的特性，所以要花一些時間來適應。我們在本文中展示一些簡短的範例，讓讀者消除對指標的陰影。

許多 C 語言教材將指標放在整本書的後半段，集中在一章到兩章來講。但我們很早就介紹指標，並在日後介紹其他 C 語言特性時，順便提到指標相關的內容。這樣的編排是希望讀者能儘早習慣指標的使用方式。

記憶體階層 (Memory Layout)

使用指標，不必然要手動管理記憶體。記憶體管理的方式，得看資料在 C 程式中的記憶體階層而定。C 語言有三種記憶體階層：

- 靜態記憶體配置 (static memory allocation)
- 自動記憶體配置 (automatic memory allocation)
- 動態記憶體配置 (dynamic memory allocation)

靜態記憶體儲存程式本身和全域變數，會自動配置和釋放，但容量有限。

自動記憶體儲存函式內的局部變數，會自動配置和釋放，在函式結束時自動釋放，容量有限。

動態記憶體儲存函式內的變數，需手動配置和釋放，可跨越函式的生命週期，可於執行期動態決定記憶體容量，可用容量約略等於系統的記憶體量。

雖然靜態記憶體和自動記憶體可自動配置，但各自受到一些限制，故仍要會用動態記憶體。

靜態記憶體配置 (Static Memory Allocation)

我們來看一個使用靜態記憶體配置的簡短範例：

```
#include <assert.h>          /* 1 */

/* DON'T DO THIS
   IN PRODUCTION CODE. */
int i = 3;                  /* 2 */

int main(void)              /* 3 */
{
    int *i_p = &i;           /* 4 */
    assert(*i_p == 3);       /* 5 */

    *i_p = 4;               /* 6 */
    assert(i == 4);          /* 7 */

    return 0;                /* 8 */
}                            /* 9 */
                           /* 10 */
```

在本範例的第 2 行，我們宣告變數 `i` 為 3 時，C 程式自動為我們配置記憶體。由於該變數屬於全域變數，會自動配置記憶體，不需人為介入。

在範例程式的第 5 行中，我們宣告了指向 `int` 的指標 `i_p`。

`int *` 表示指向 `int` 型別的指標。C 語言需要知道指標所指向的型別，才能預知記憶體的大小。

`&i` 代表回傳變數 `i` 的虛擬記憶體位址。因為指標的值是記憶體位址，在本例中，我們的指標 `i_p` 指向一塊已經配置好的記憶體，即為變數 `i` 所在的位址。

接著，我們在範例的第 6 行確認確認指標 `i_p` 所指向的值的確是 3。在這行敘述中，`*i_p` 代表解址，解址後會取出該位址的值。在本例中即取回 3。

接著，我們在第 7 行藉由修改指標 `i_p` 所指向的記憶體間接修改變數 `i` 的值。

最後，我們在第 8 行藉由 `assert(i == 4);` 敘述確認指標 `i_p` 確實間接修改到變數 `i`。代表兩者的確指向同一塊記憶體。

如果讀者用 Valgrind 或其他記憶體檢查軟體去檢查此程式，可發現此範例程式沒有配置動態記憶體，也沒有記憶體洩露的問題。代表使用指標不必然要手動管理記憶體。

附帶一提，在本範例中，我們使用了全域變數。這在撰碼上是不良的習慣，因全域變數很容易在不經意的情形下誤改。我們的程式只是為了展示靜態記憶體的特性，不建議在實務上使用全域變數。

自動記憶體配置 (Automatic Memory Allocation)

接著，我們來看一個使用自動記憶體配置的實例：

```
#include <assert.h>      /* 1 */
int main(void)           /* 2 */
{
    int i = 3;           /* 3 */
    int *i_p = &i;        /* 4 */
    assert(*i_p == 3);   /* 5 */
    *i_p = 4;            /* 6 */
    assert(i == 4);      /* 7 */
    return 0;             /* 8 */
}                         /* 9 */
/* 10 */
```

在本例的第 4 行中，當我們宣告變數 `i` 的值為 3 時，同樣會自動配置記憶體。由於變數 `i` 存在於主函式中，故使用自動記憶體配置。

在本例的第 5 行中，我們宣告了指向變數 `i` 的指標 `i_p`。這時候 `i` 的記憶體已經配置好了。

這個範例除了記憶體配置的方式外，其他的指令和前一節的範例雷同，故不再詳細說明，請讀者自行閱讀。

如果讀者用 Valgrind 或其他記憶體檢查軟體檢查此範例程式，同樣可發現此程式沒有配置動態記憶體，也沒有記憶體洩露的問題。

動態記憶體配置 (Dynamic Memory Allocation)

我們來看一個動態記憶體配置的實例：

```
#include <assert.h>          /* 1 */
#include <stdio.h>            /* 2 */
#include <stdlib.h>           /* 3 */
int main(void)               /* 4 */
{
    int *i_p = \               /* 5 */
        (int *) malloc(sizeof(int)); /* 6 */
    if (!i_p) {                /* 7 */
        fprintf(
```

(continues on next page)

(continued from previous page)

```

        stderr,
        "Failed to allocate memory\n"); /* 8 */
    goto ERROR; /* 9 */
}

*i_p = 3; /* 10 */
if ((*i_p == 3)) {
    fprintf(
        stderr,
        "Wrong value: %d\n",
        *i_p); /* 13 */
    goto ERROR; /* 14 */
}

free(i_p); /* 15 */

return 0; /* 17 */

ERROR: /* 18 */
if (i_p)
    free(i_p); /* 19 */
/* 20 */

return 1; /* 21 */
/* 22 */
}

```

在第 6 行中，我們配置一塊大小為 int 的記憶體。

C 標準函式庫中配置記憶體的函式為 `malloc()`，該函式接收的參數為記憶體的大小。我們甚少手動寫死記憶體的大小，而會使用 `sizeof` 直接取得特定資料型別的大小。這幾乎是固定的手法了。

`malloc()` 回傳的型別是 `void *`，即為大小未定的指標。我們會自行手動轉型為指向特定型別的指標。有些 C 編譯器會自動幫我們轉型，就不用自行轉型。

由於配置記憶體是有可能失敗的動作，我們在第 7 行至第 10 行檢查是否成功地配置記憶體。

當 `malloc()` 失敗時，會回傳 `NULL`。而 `NULL` 在布林語境中會視為偽，故 `!i_p` 在 `i_p` 的值為 `NULL` 時會變為真。

若 `!i_p` 為真，代表 `malloc()` 未成功配置記憶體，這時候我們會中止一般的流程，改走錯誤處理流程。我們先在標準錯誤印出錯誤訊息，然後用 `goto` 跳到標籤 `ERROR` 所在的地方。由於 C 沒有內建的錯誤處理機制，使用 `goto` 跳離一般程式流程算是窮人版的例外 (exception)。

`printf()` 會在標準輸出印出訊息，而 `fprintf()` 敘述可以選擇輸出目標。本範例會在標準錯誤印出訊息。

如果 `malloc()` 成功地配置記憶體，我們就繼續一般的程式流程。我們在第 11 行將 `i_p` 指向的記憶體賦值為 3，然後在第 12 行至第 15 行檢查是否正確地賦值。一般來說，這行敘述是不需檢查的。這裡僅是展示這項特性。

由於 `i_p` 所指向的記憶體是手動配置的，我們在第 16 行釋放 `i_p` 所占用的記憶體。

基本上，`malloc()` 和 `free()` 應成對出現。每配置一次記憶體就要在確定不使用時釋放回去。由於這個範例相當短，這似乎顯而易見。但是在撰寫動態資料結構時，會跨過多個函式，比這個範例複雜得多，就有可能會忘了釋放記憶體。

當程式發成錯誤時，我們會改走錯誤處理的流程。在本例中，錯誤流程在第 18 行至第 21 行。在進行錯誤處理時，我們同樣會釋放記憶體，但程式最後會回傳非零值 1，代表程式異常結束。

由於在錯誤發生時，我們無法確認 `i_p` 是否已配置記憶體，所以要用 `if` 敘述來確認。請讀者再回頭把整個程式運行的流程看一次，即可了解。

空指標 (Null Pointer)

當 C 程式試圖去存取系統資源時，該指令有可能失敗，故要撰寫錯誤處理相關的程式碼。以下範例程式試圖打開一個文字檔案 `file.txt`：

```
#include <stdio.h> /* 1 */

int main(void) /* 2 */
{
    FILE *fp; /* 3 */

    fp = fopen("file.txt", "r"); /* 5 */
    if (!fp) { /* 6 */
        fprintf(
            stderr,
            "Failed to open a file\n"); /* 7 */
        return 1; /* 8 */
    } /* 9 */

    fclose(fp); /* 10 */

    return 0; /* 11 */
} /* 12 */
```

在第 5 行中，我們試圖用 `fopen()` 函式以讀取模式開啟 `file.txt`。當檔案無法開啟時，會回傳空指標 `NULL`。所以我們在第 6 行至第 9 行檢查指標 `fp` 是否為空。

當 `fp` 為空指標時，`!fp` 會負負得正，這時候程式會中止一般流程，改走錯誤處理流程。在這個範例中，我們在終端機印出錯誤訊息，並回傳非零值 1 代表程式異常結束。

以下是另一種檢查空指標的寫法：

```
fp = fopen("file.txt", "r");
if (NULL == fp) {
    fprintf(stderr, "Failed to open the file\n");
    return 1;
}
```

基本上，兩者皆可使用，這僅是風格上的差異。

比較指標是否相等

當我們使用指標時，會處理兩個值，一個是指標所指向的位址，另一個是指標所指向的值。我們用以下範例來看兩者的差別：

```
#include <stdio.h>          /* 1 */
int main(void)             /* 2 */
{
    int a = 3;              /* 3 */
    int b = 3;              /* 4 */

    int *p = &a;            /* 5 */
    int *q = &a;            /* 6 */
    int *r = &b;            /* 7 */

    if (! (p == q)) {        /* 9 */
        fprintf(
            stderr,
            "Wrong addresses: %p %p\n",
            p, q);              /* 10 */
        goto ERROR;           /* 11 */
    }                         /* 12 */

    if (p == r) {             /* 13 */
        fprintf(
            stderr,
            "Wrong addresses: %p %p\n",
            p, r);              /* 14 */
        goto ERROR;           /* 15 */
    }                         /* 16 */

    if (! (*p == *q)) {       /* 17 */
        fprintf(
            stderr,
```

(continues on next page)

(continued from previous page)

```

    "Wrong values: %d %d\n",
    *p, *q); /* 18 */
    goto ERROR; /* 19 */
}
/* 20 */

if (!(*p == *r)) { /* 21 */
    fprintf(
        stderr,
        "Wrong values: %d %d\n",
        *p, *r); /* 22 */
    goto ERROR; /* 23 */
}
/* 24 */

return 0; /* 25 */

ERROR: /* 26 */
    return 1; /* 27 */
}
/* 28 */

```

一開始，我們分別在第 4 行及第 5 行配置兩塊自動記憶體。雖然變數 `a` 和變數 `b` 的值是相同的，但兩者存在於不同的記憶體區塊。

接著，我們第 6 行至第 8 行宣告三個指標，分別指向這兩個變數。

我們可以預期 `p` 和 `q` 同時會指向變數 `a` 所在的記憶體，而 `r` 則指向變數 `b` 所在的記憶體。但 `p`、`q`、`r` 所指向的值皆相等。從範例程式中即可確認這樣的狀態，讀者可自行閱讀一下。

野指標 (Wild Pointer)

若我們宣告了指標但未對指標賦值，這時候指標的位址是未定義的，由各家 C 編譯器自行決定其行為。宣告但未賦值的指標稱為野指標，這時候指標的值視為無意義的垃圾值，不應依賴其結果。

我們來看一個野指標的範例程式：

```

#include <stdio.h> /* 1 */
#include <stdlib.h> /* 2 */

#define PPUTS(format, ...) { \
    fprintf(stdout, \
        "(%s:%d) " format "\n", \
        __FILE__, __LINE__, \
        ##__VA_ARGS__); \
}

```

(continues on next page)

(continued from previous page)

```

}

/* 3 */

#define PUTERR(format, ...) { \
    fprintf(stderr, \
        "(%s:%d) " format "\n", \
        __FILE__, __LINE__, \
        ##__VA_ARGS__); \
}

int main(void)
{
    /* Wild pointer. */
    int *i_p;

    if (NULL == i_p) {
        PUTS("i_p is NULL");
    }

    if (!i_p) {
        PUTS("i_p is EMPTY");
    }

    /* NULL pointer. */
    i_p = NULL;

    if (NULL == i_p) {
        PUTS("i_p is NULL");
    }

    if (!i_p) {
        PUTS("i_p is EMPTY");
    }

    i_p = \
        (int *) malloc(sizeof(int));
    if (!i_p) {
        PUTERR(
            "Failed to allocate memory");
        goto ERROR;
    }

    if (NULL == i_p) {
        PUTS("i_p is NULL");
    }
}

```

(continues on next page)

(continued from previous page)

```

if (!i_p) {                                /* 32 */
    PUTS("i_p is EMPTY");
}                                              /* 33 */
                                                /* 34 */

free(i_p);                                     /* 35 */

return 0;                                      /* 36 */

ERROR:                                     /* 37 */
if (i_p)
    free(i_p);                                /* 38 */
                                                /* 39 */

return 1;                                      /* 40 */
}                                              /* 41 */

```

我們在本範例中宣告了兩個巨集，該巨集的用意是節省版面。我們在後續的文章會詳細介紹巨集的使用方式。

在第 8 行時，指標 `i_p` 尚未賦值，其值為垃圾值。使用不同 C 編譯器編譯此範例程式時，會得到不同的結果。

在第 16 行時，我們將 `i_p` 以 `NULL` 賦值，這時候 `i_p` 就不再是野指標，轉為空指標。

在第 23 行時，我們手動配置了一塊記憶體，這時候 `i_p` 就是一般的指標。

由這個例子可知，我們在宣告指標時，若未馬上配置記憶體或其他系統資源時，應該立即以 `NULL` 賦值，讓該指標成為空指標。

迷途指標 (Dangling Pointer)

原本指向某塊記憶體的指標，當該記憶體中途消失時，該指標所指向的位址不再合法，這時候的指標就成為迷途指標。如同野指標，迷途指標所指向的值視為垃圾值，不應依賴其結果。

以下是一個迷途指標的範例程式：

```

#include <stdio.h>                                /* 1 */

int main(void)                                /* 2 */
{
    int *i_p = NULL;                            /* 3 */
                                                /* 4 */

    {
        int i = 3;                               /* 5 */
                                                /* 6 */

```

(continues on next page)

(continued from previous page)

```

    i_p = &i;                                /* 7 */
}                                         /* 8 */

/* i_p is now a
   dangling pointer. */
if (*i_p == 3) {                           /* 9 */
    fprintf(                                /* 10 */
        stderr,                            /* 11 */
        "It should not be 3\n");          /* 12 */
    return 1;                             /* 13 */
}

return 0;                                /* 15 */
}                                         /* 16 */

```

在第 7 行時，指標 `i_p` 指向 `i`。但在第 8 行時，該區塊結束，`i` 的記憶體會自動釋放掉，這時候 `i_p` 所指向的記憶體不再合法，`i_p` 變成迷途指標。之後的運算基本上是無意義且不可靠的。

我們再來看另一個迷途指標的例子：

```

#include <stdio.h>                                /* 1 */
#include <stdlib.h>                               /* 2 */

#define PUTERR(format, ...) { \
    fprintf(stderr, \
        "(%s:%d) " format "\n", \
        __FILE__, __LINE__, \
        ##__VA_ARGS__); \
}

int main(void)                                     /* 4 */
{
    int *i_p = \
        (int *) malloc(sizeof(int)); /* 6 */
    if (!i_p) {                                    /* 7 */
        PUTERR(                                /* 8 */
            "Failed to allocate int"); /* 9 */
        goto ERROR;                         /* 10 */
    }                                         /* 11 */

    free(i_p);                                /* 12 */

    /* i_p is now
       a dangling pointer. */                  /* 13 */
    *i_p = 3;
}

```

(continues on next page)

(continued from previous page)

```
if (*i_p == 3) {                                /* 14 */
    PUTERR(                                         /* 15 */
        "It should not be 3");                      /* 16 */
    goto ERROR;                                     /* 17 */
}                                                 /* 18 */

return 0;                                         /* 19 */

ERROR:
if (i_p)                                         /* 20 */
    free(i_p);                                    /* 21 */
                                                /* 22 */

return 1;                                         /* 23 */
}                                                 /* 24 */
```

在第 6 行時，我們配置一塊記憶體到 `i_p`。在第 12 行時這塊記憶體釋放掉了，這時候 `i_p` 所指的位址不再合法，故 `i_p` 成為迷途指標。即使之後的運算能夠成功，那也只是一時僥倖而已。

結語

在本文中，我們介紹了數個指標的基本用法。我們把指標相關的內容放在前半部，是為了要讓大家及早適應指標。在後續介紹各種 C 語言的特性時，我們會再加入指標的使用方式。

陣列 (ARRAY)

前言

C 語言的陣列是線性 (linear)、連續 (continuous)、同質的 (homogeneous) 資料結構，使用零或正整數為索引來存取其中元素。

在 C 語言中，陣列是唯一的內建資料結構，其優點為隨機存取的效率很好。除了陣列外，其他的動態資料結構皆需程式設計者自行實作。本文介紹陣列的使用方式。

宣告陣列

以下敘述建立一個長度為 5、元素型別為 int 的陣列 arr：

```
int arr[5];
```

要注意這時候陣列元素尚未初始化。陣列未初始化時所存的值視為垃圾值，其運算結果不可靠。

我們也可以在宣告陣列時一併賦值：

```
int arr[5] = {3, 4, 5, 6, 7};
```

或者是用稍微取巧的方式來初始化：

```
int arr[] = {3, 4, 5, 6, 7};
```

這時候陣列 arr 的長度由賦值自動決定，在本範例敘述中即為 5。

如果想要明確地列出陣列元素所在的位置，可以用下列方式來宣告陣列：

```
int arr[] = {
    [0] = 3,
    [1] = 4,
    [2] = 5,
    [3] = 6,
    [4] = 7,
};
```

這樣寫程式碼會變長，好處是可明確看出每個元素所在的位置。

但要注意陣列的長度不能由變數決定：

```

int main(void)
{
    unsigned sz = 5;

    /* Error. */
    int arr[sz] = {1, 2, 3, 4, 5};

    return 0;
}

```

因為陣列的長度要在編譯期就決定好。如果想要在執行期動態生成陣列，要用動態配置記憶體的方式。我們後文會談如何動態配置陣列。

存取陣列元素

陣列使用零或正整數存取陣列元素。參考以下範例：

```

#include <assert.h>          /* 1 */

int main(void)              /* 2 */
{
    /* 3 */
    int arr[] = {3, 4, 5};   /* 4 */

    assert(3 == arr[0]);     /* 5 */
    assert(4 == arr[1]);     /* 6 */
    assert(5 == arr[2]);     /* 7 */

    return 0;                /* 8 */
}                            /* 9 */

```

我們在第 3 行宣告了長度為 3，元素型別為 `int` 的陣列 `arr`。然後在第 5 行至第 7 行間分別對其中元素以索引取值。利用斷言確認取出的值是正確的。

注意取索引時，第一個元素的索引值從 0 開始，而非 1，這是因為索引是一種偏移值 (offset) 的概念。

但 C 語言不會檢查索引是否逾越陣列的邊界。參考以下反例：

```

#include <stdio.h>          /* 1 */

int main(void)              /* 2 */
{
    /* 3 */
    int arr[] = {3, 4, 5};   /* 4 */

    /* Error */             /* 5 */
}

```

(continues on next page)

(continued from previous page)

```
printf("%d\n", arr[3]); /* 5 */
return 0; /* 6 */
/* 7 */
```

陣列 `arr` 的長度為 3，最大的合法索引值應為 2，但本例在第 5 行時故意取索引值為 3 的值。這時候取出的值是垃圾值，其運算結果不可靠。

由於逾越邊界 (out of bound) 算是常見的錯誤，資訊界出現過數個 C 方言 (C dialect)，意圖改善 C 常見的錯誤。其中一個例子是微軟的研究項目 Checked C⁷⁹。但這些 C 方言，除了展示一些對 C 語言的想法外，幾乎沒用程式設計師將其用在實務上。

如果讀者真的很在意陣列邊界的問題，現階段的方式就是自行實作工具函式或陣列物件，在這些自製函式或物件中加入邊界檢查的功能。

走訪陣列

走訪陣列元素的方式是使用 `for` 迴圈搭配計數器 (counter)。參考下例：

```
#include <stddef.h>
#include <stdio.h>

int main(void)
{
    int arr[] = {3, 4, 5, 6, 7};

    for (size_t i = 0; i < 5; i++) {
        printf("%d\n", arr[i]);
    }

    return 0;
}
```

這裡我們把計數器的邊界寫死在程式中，實務上不會用這樣的方式。取得陣列大小的方式請看下一節。

⁷⁹ <https://www.microsoft.com/en-us/research/project/checked-c/>

計算陣列大小

C 陣列本身沒有儲存陣列大小的資訊。如果想要知道陣列的大小，得自行計算。參考下例：

```
#include <assert.h>          /* 1 */
#include <stddef.h>           /* 2 */

int main(void)                /* 3 */
{
    int arr[] = {3, 4, 5, 6, 7}; /* 4 */
                                /* 5 */

    size_t sz = \
        sizeof(arr) / sizeof(arr[0]); /* 6 */

    assert(5 == sz);             /* 7 */

    return 0;                   /* 8 */
}                                /* 9 */
```

在此範例程式中，我們在第 6 行分別計算陣列大小和陣列元素大小，將其相除後即可得陣列長度。在本例中其值為 5。

有些 C 程式設計者把這個小技巧包成以下巨集：

```
#define ARRAY_SIZE(arr) (sizeof(arr) / sizeof(arr[0]))
```

但這個巨集是很不牢靠的。因為本節所介紹的方式只對自動配置記憶體的陣列有效，若陣列使用動態配置記憶體，則無法使用這個方法。

如果我們想儲存陣列長度的資訊，需將長度存在另一個變數中。由於陣列和陣列長度兩者是連動的，我們會用結構體把兩個變數包在一起。例如，以下結構體宣告一個動態陣列：

```
struct array_t {
    size_t size;
    size_t capacity;
    int *elems;
};
```

當我們改變陣列大小時，就多寫幾行指令修改 `size` 和 `capacity` 的大小，就可以儲存陣列的長度和容量。我們會在後文介紹結構體。至於實作動態陣列的方式，則需參考資料結構方面的教材。

還有一個方式是在陣列的尾端放入一個旗標 (flag)。當程式走訪到該旗標時，就代表到達陣列的尾端。例如，以下的 `float` 陣列在尾端塞入 `Nan` (非數字) 做為該陣列的旗標：

```
float arr[] = {3.0, 4.0, 5.0, 6.0, 7.0, NaN};
```

由於 *NaN* 不屬於任何浮點數，走訪陣列到該元素時，就知道到達陣列尾端了。

那麼，*NaN* 是怎麼來的呢？在 C 語言中可以利用一些在數學上會求得非數字的計算取得 *NaN*，像是 $0.0 / 0.0$ 或是 $\log(-1)$ 等。

我們延續這個概念，寫成一個完整的範例：

```
#include <assert.h>                                /* 1 */
#include <stdio.h>                                 /* 2 */

#define NaN (0.0 / 0.0)                            /* 3 */

#define IS_NaN(n) \
    (((n) > (n)) ? 0 : \
     ((n) == (n)) ? 0 : \
     ((n) < (n)) ? 0 : 1)                         /* 4 */

int main(void)                                     /* 5 */
{
    float arr[] = \
        {3.0, 4.0, 5.0, 6.0, 7.0, NaN};          /* 6 */

    size_t sz = 0;                                /* 7 */
    {
        size_t i;
        for (i = 0; !IS_NaN(arr[i]); ++i) {       /* 11 */
            printf("(%u) %.2f\n", i+1, arr[i]);   /* 12 */
            ++sz;                               /* 13 */
        }                                     /* 14 */
    }                                         /* 15 */

    assert(5 == sz);                            /* 16 */

    return 0;                                    /* 17 */
}                                              /* 18 */
```

在第 3 行中，我們宣告巨集常數 *NaN*，求得非數字的方式是用 $0.0 / 0.0$ 。

由於 *NaN* 對 *NaN* 進行大於、小於、等於比較都為偽，我們利用這個特性在第 4 行寫出 *IS_NaN* 巨集。我們這裡還沒詳細說明巨集的用法，先想成一種擬函式即可。

在第 7 行中，我們將 *NaN* 當成 *float* 陣列 *arr* 的旗標。

在第 11 行至第 14 行間，我們走訪陣列 *arr*。在走到 *NaN* 時，陣列會停止走訪。在第 16 行確認陣列 *arr* 的長度為 5，未計入旗標。

但旗標的泛用性不佳，因為每種資料型態的陣列能用的旗標相異。也有些陣列找不到合用

的旗標。最好的方式還是用資料結構中建立動態陣列的模式，因為動態陣列本來就會儲存陣列長度和容量的資訊。

動態配置的陣列

我們先前的範例中，陣列使用自動配置記憶體。但我們若要在執行期動態生成陣列，則要改用動態配置記憶體的方式。

我們同樣用 `malloc()` 函式來配置記憶體。參考以下敘述：

```
int *arr = (int *) malloc(size * sizeof(int));
```

我們以 `sizeof` 求得單一元素的大小後，乘上陣列的長度 `size` 即可配置一塊足夠大小的記憶體，用來儲存陣列 `arr` 的元素。由此可知，陣列在電腦中以是一整塊連續的記憶體來儲存，所以可以用索引值快速存取。

如果想要在配置記憶體時一併將元素初始化為 0，改用 `calloc()` 函式即可。但 `calloc()` 函式的參數略有不同：

```
int *arr = (int *) calloc(size, sizeof(int));
```

由於多了初始化的動作，`calloc()` 函式會比 `malloc()` 函式慢一點點。

使用完後同樣要釋放記憶體：

```
free(arr);
```

由於陣列內部是單一且連續的記憶體區塊，所以可在單一 `free()` 函式呼叫中釋放掉。不論使用 `malloc()` 或 `calloc()`，皆使用 `free()` 來釋放記憶體。

我們來看一個動態配置記憶體陣列的範例：

```
#include <stdlib.h> /* 1 */
#include <stdio.h> /* 2 */

int main(void) /* 3 */
{
    size_t sz = 5; /* 4 */

    int *arr = \
        (int *) malloc(sz * sizeof(int)); /* 5 */
    if (!arr) { /* 6 */
        perror( /* 7 */
            "Failed to allocate"
            " an array\n"); /* 8 */
        goto ERROR; /* 9 */
    }
}
```

(continues on next page)

(continued from previous page)

```

}

for (size_t i = 0; i < sz; i++) { /* 10 */
    arr[i] = 3 + i; /* 11 */
} /* 12 */
/* 13 */

int data[] = {3, 4, 5, 6, 7}; /* 14 */
for (size_t i = 0;
      i < sizeof(data) / sizeof(int);
      i++) { /* 15 */
    if (arr[i] != data[i]) { /* 16 */
        fprintf(
            stderr,
            "Unequal values: %d %d\n",
            arr[i], data[i]); /* 17 */
        goto ERROR; /* 18 */
    } /* 19 */
} /* 20 */

free(arr); /* 21 */

return 0; /* 22 */

ERROR: /* 23 */
if (arr) /* 24 */
    free(arr); /* 25 */

return 1; /* 26 */
} /* 27 */

```

我們在第 6 行動態配置陣列 `arr`。由於動態配置記憶體可能失敗，我們在第 7 行至第 10 行間檢查 `arr` 是否存在。當配置記憶體未成功時，放棄一般的程式流程，改走錯誤處理流程。

接著，我們在第 11 行至第 13 行間對 `arr` 的元素逐一賦值。

我們在第 14 行至第 20 行間檢查 `arr` 的值是否正確，若發生錯誤，印出錯誤值並中斷一般程式流程。

最後，在第 21 行釋放掉 `arr` 所占用的記憶體。

多維陣列

先前的範例皆為一維陣列，但 C 語言允許多維陣列。參考以下宣告多維陣列的敘述：

```
int mtx[3][2] = {
    {1, 2},
    {3, 4},
    {5, 6}
};
```

我們同樣可以對多維陣列存取索引值：

```
#include <assert.h>

int main(void)
{
    int mtx[3][2] = {
        {1, 2},
        {3, 4},
        {5, 6}
    };

    assert(4 == mtx[1][1]);

    return 0;
}
```

上述範例的記憶體是自動配置的。那如果我們要動態配置記憶體呢？這時候有兩種策略，其中一種較直觀的策略是動態配置陣列 ()。參考以下範例：

```
#include <stdio.h>          /* 1 */
#include <stdlib.h>          /* 2 */

int main(void)              /* 3 */
{
    size_t row = 3;          /* 4 */
    size_t col = 2;          /* 5 */

    int **mtx = \               /* 6 */
        (int **) malloc(      /* 7 */
            row * sizeof(int *)); /* 8 */
    if (!mtx) {                /* 9 */
        perror(                /* 10 */
            "Failed to allocate rows\n"); /* 11 */
        goto ERROR;
    }
```

(continues on next page)

(continued from previous page)

```

for (size_t i = 0; i < row; i++) { /* 12 */
    mtx[i] = \
        (int *) malloc(
            col * sizeof(int)); /* 13 */
    if (!mtx[i]) { /* 14 */
        fprintf(
            stderr,
            "Failed to allocate col\n"); /* 15 */
        goto ERROR; /* 16 */
    } /* 17 */
} /* 18 */

int n = 1; /* 19 */
for (size_t i = 0; i < row; i++) { /* 20 */
    for (size_t j = 0; j < col; j++) { /* 21 */
        mtx[i][j] = n; /* 22 */

        n++;
    } /* 24 */
} /* 25 */

/* {{1, 2},
   {3, 4},
   {5, 6}} */
if (!(4 == mtx[1][1])) { /* 26 */
    fprintf(
        stderr,
        "Wrong value: %d\n",
        mtx[1][1]);
    goto ERROR; /* 28 */
} /* 29 */

for (size_t i = 0; i < row; i++) { /* 30 */
    free(mtx[i]); /* 31 */
} /* 32 */

free(mtx); /* 33 */

return 0; /* 34 */

ERROR: /* 35 */
if (mtx) {
    for (size_t i = 0; i < row; i++) { /* 37 */
        if (mtx[i]) /* 38 */

```

(continues on next page)

(continued from previous page)

```

    free(mtx[i]);           /* 39 */
}
/* 40 */

free(mtx);               /* 41 */
/* 42 */

return 1;                /* 43 */
/* 44 */
}

```

本範例的多維陣列的型別是 `int **`。這其實是指向指標的指標。第一層指標是指向 `int *` 的指標，第二層指標則是指向 `int` 的指標。

我們在第 7 行配置第一層陣列，其長度為 `row`，陣列元素的大小為 `int *`。

我們在第 12 行至第 18 行間配置第二層陣列，其長度為 `col`，陣列元素的大小為 `int`。由於我們要配置多次，故我們使用迴圈來重覆進行相同的任務。

接著，我們在第 19 行至第 25 行間用雙層迴圈逐一對多維陣列 `mtx` 賦值。

本範例為了簡化，只在第 26 行至第 28 行間檢查其中一個元素。如果讀者有興趣，可試著改寫這個程式，改成對每個元素逐一檢查。

最後在第 30 行至 33 行間釋放記憶體。注意由內而外逐一釋放。若我們先釋放外部陣列，就沒有合法的指標可指向內部陣列的記憶體，造成記憶體洩露。

我們先前有提過，多維陣列有兩種宣告方式。除了前述的陣列的陣列外，我們仍然可以用一維陣列儲存多維陣列，再將外部多維座標經計算轉換為內部一維座標。參考以下結構體宣告：

```

struct matrix_t {
    size_t row;
    size_t col;
    double *elements;
};

```

在這個結構體中，我們內部使用一維陣列 `elements` 來儲存陣列元素。要存取陣列元素時得將座標進行轉換。至於實作的方式，屬於資料結構的範圍，此處不詳談。

前言

學完陣列和指標後，就有足夠的預備知識學習 C 字串。C 語言沒有獨立的字串型別，而 C 字串是以 `char` 或 `wchar_t` 為基礎型別的陣列，所以要有先前文章的鋪陳才容易學習 C 字串。

C 語言的字串方案

在 C 語言中，常見的字串方案有以下數種：

- 固定寬度字元陣列
 - 字元陣列 (character array)
 - 寬字元陣列 (wide character array)，使用 `wchar.h` 函式庫
- 多字節編碼 (multibyte encodings)，像是 Big5 (大五碼) 或 GB2312 等
- 統一碼 (Unicode)：包括 UTF-8、UTF-16、UTF-32 等

有些方案為等寬字元，有些方案則採不等寬字元編碼。預設的字元陣列僅能處理英文文字，其他方案則是為了處理多國語文文字而產生的。

例如，在支援 Unicode 的終端機環境，可以透過 `wchar_t` 印出中文字串：

```
#include <locale.h>
#include <wchar.h>

int main(void)
{
    /* Trick to print multibyte strings. */
    setlocale(LC_CTYPE, "");

    wchar_t *s = L"你好，世界";
    printf("%ls\n", s);

    return 0;
}
```

由於本文的目的是了解字串的基本操作，我們仍然是以預設的字元陣列為準，不考慮多國語言的情境。

C 字串微觀

我們由 "Hello World" 字串來看 C 字串的組成：

'H'	'e'	'l'	'l'	'o'	' '	'W'	'o'	'r'	'l'	'd'	'\0'
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	------

由上圖可知，C 字串除了依序儲存每個字元外，在尾端還會額外加上一個 '\0' 字元，代表字串結束。由於 C 字串需要尾端的 '\0' 字元來判斷字串結束，我們在處理字串時，別忘了在字串尾端加上該字元。

C 語言不會幫程式設計者檢查字串是否正確，而會直接認定字串尾端有附加 '\0'。當字串尾端沒有附加 '\0' 時，會造成字串相關演算法的錯誤。

我們把上圖所示的字元陣列寫成以下的程式碼：

```
#include <assert.h>
#include <string.h>

int main(void)
{
    char s[] = \
        {'H', 'e', 'l', 'l', 'o', ' ', \
         'W', 'o', 'r', 'l', 'd', '\0'};
    assert(0 == strcmp(s, "Hello World"));

    return 0;
}
```

由此可知 C 字串在本質上是尾端為 NULL 的字元陣列。當我們在實作字串相關演算法時也要以這個想法來寫程式。

接下來，我們會介紹數個字串操作的情境。由於 C 標準函式庫已經有 **string.h** 函式庫，在操作字串時應優先使用該函式庫，而非重造輪子；本文展示的程式僅供參考。

計算 C 字串長度

計算字串長度時，不包含尾端的結束字尾，所以 C 字串 "happy" 的字串長度為 5 而非 6。可參考以下的範例程式碼：

```
#include <assert.h>                      /* 1 */
#include <stddef.h>                        /* 2 */
#include <string.h>                         /* 3 */
```

(continues on next page)

(continued from previous page)

```

int main(void)                                /* 4 */
{
    char s[] = "hello";                      /* 5 */
    size_t sz = 0;                           /* 6 */

    for (size_t i = 0; s[i]; i++) {          /* 8 */
        sz++;                                /* 9 */
    }                                       /* 10 */

    assert(sz == strlen(s));                /* 11 */

    return 0;                                /* 12 */
}
                                         /* 13 */

```

我們藉由走訪字串來計算字串的長度，這段動作位於第 8 行至第 10 行。當字串走到尾端的零值時，`s[i]` 會回傳零，這時迴圈會結束。藉由走訪字串陣列一輪就可以知道字串長度。

複製 C 字串

一般使用 `strcpy` 函式的範例，都是預先配置某個長度的字元陣列；本例略加修改，先動態計算來源字串的長度，再由堆積(heap)動態配置一塊新的字元陣列，將原本的字元逐一複製到目標字串即完成。參考以下程式碼：

```

#include <assert.h>                                /* 1 */
#include <stddef.h>                                 /* 2 */
#include <stdio.h>                                  /* 3 */
#include <stdlib.h>                                 /* 4 */
#include <string.h>                                 /* 5 */

int main(void)                                     /* 6 */
{
    char s[] = "Hello World";                     /* 7 */
    size_t sz_s = strlen(s);                      /* 8 */

    /* Add trailing zero. */                      /* 9 */
    size_t sz = sz_s + 1;                          /* 10 */

    char *out = \
        malloc(sz * sizeof(char));                /* 11 */
    if (!out) {                                    /* 12 */

```

(continues on next page)

(continued from previous page)

```

fprintf(
    stderr,
    "Failed to allocate C string\n"); /* 13 */
return 1; /* 14 */
/* 15 */

for (size_t i = 0; i < sz_s; i++) { /* 16 */
    out[i] = s[i]; /* 17 */
} /* 18 */

out[sz-1] = '\0'; /* 19 */

assert(
    0 == strcmp(out, "Hello World")); /* 20 */

free(out); /* 21 */

return 0; /* 22 */
/* 23 */
}

```

要配置記憶體前，要先知道記憶體的長度，所以我們在第 10 行計算字串的長度。

接著，我們在第 11 行為字串配置記憶體。

拷貝字串的方式為逐一拷貝字元陣列內的字元，該動作位於第 16 行至第 18 行。

此外，還要為字元加上尾端的零值，該動作為於第 19 行。

在本例中，由於 `out` 是由 heap 配置記憶體，使用完要記得手動釋放，該動作位於第 21 行。

相接兩個 C 字串

原本的 `strcat` 函式需預先估計目標字串的長度，筆者略為修改，採用動態計算字串長度後生成所需長度的字元陣列，最後將原本的字串逐一複製過去。範例程式碼如下：

```

#include <assert.h> /* 1 */
#include <stdio.h> /* 2 */
#include <stdlib.h> /* 3 */
#include <string.h> /* 4 */

int main(void)
{
    char s_a[] = "Hello "; /* 5 */
    char s_b[] = "World"; /* 6 */
    /* 7 */

```

(continues on next page)

(continued from previous page)

```

size_t sz_a = strlen(s_a);           /* 8 */
size_t sz_b = strlen(s_b);           /* 9 */
size_t sz = sz_a + sz_b + 1;         /* 10 */

char *out = \
    malloc(sz * sizeof(char));        /* 11 */
if (!out) {                          /* 12 */
    fprintf(
        stderr,
        "Failed to allocate"
        " a C string\n");
    return 1;                         /* 15 */
}

for (size_t i = 0; i < sz_a; i++) {   /* 17 */
    out[i] = s_a[i];                 /* 18 */
}                                       /* 19 */

for (size_t i = 0; i < sz_b; i++) {   /* 20 */
    out[i+sz_a] = s_b[i];            /* 21 */
}                                       /* 22 */

out[sz-1] = '\0';                     /* 23 */

assert(
    0 == strcmp(out, "Hello World"));  /* 24 */

free(out);                           /* 25 */

return 0;                            /* 26 */
}                                     /* 27 */

```

如同上一節的例子，我們要先計算記憶體的長度，所以我們在第 10 行計算字串長度，並預先加入尾端零值的長度。

本例有兩段字串，所以拷貝字串的動作要分兩輪。在第 17 行至第 19 行間拷貝第一個字串。在第 20 行至 22 行間拷貝第二個字串。

別忘了加上尾端零值，這段動作發生在第 23 行。

在本例中，由於 `out` 是由 heap 配置記憶體，使用完要記得手動釋放。本動作在第 25 行完成。

檢查兩個 C 字串是否相等

檢查字串相等的方式為逐一檢查字元陣列的字元是否相等。可參考以下範例程式碼：

```
#include <assert.h>          /* 1 */
#include <stdbool.h>          /* 2 */

int main(void)                /* 3 */
{
    char a[] = "happy";        /* 4 */
    char b[] = "happy hour";   /* 5 */

    bool is_equal = true;      /* 6 */

    char *p = a;               /* 7 */
    char *q = b;               /* 8 */
    while (*p && *q) {        /* 9 */
        /* Unequal character. */ /* 10 */
        if (*p != *q) {        /* 11 */
            is_equal = false;  /* 12 */
            goto END;          /* 13 */
        }
        /* 14 */
    }
    /* 15 */

    p++; q++;                 /* 16 */
}                                /* 17 */

/* Unequal string length. */ /* 18 */
if (*p != *q) {                /* 19 */
    is_equal = false;          /* 20 */
    goto END;                  /* 21 */
}
/* 22 */

END:                           /* 23 */
assert(false == is_equal);     /* 24 */

return 0;                      /* 25 */
}                                /* 26 */
```

我們不希望影響原本的字元陣列，所以我們在第 8 行及第 9 行分別拷貝兩字元陣列的位址。

檢查字元陣列的動作位於第 10 行至第 17 行。當字元不相等時，直接結束比較字元陣列的迴圈。

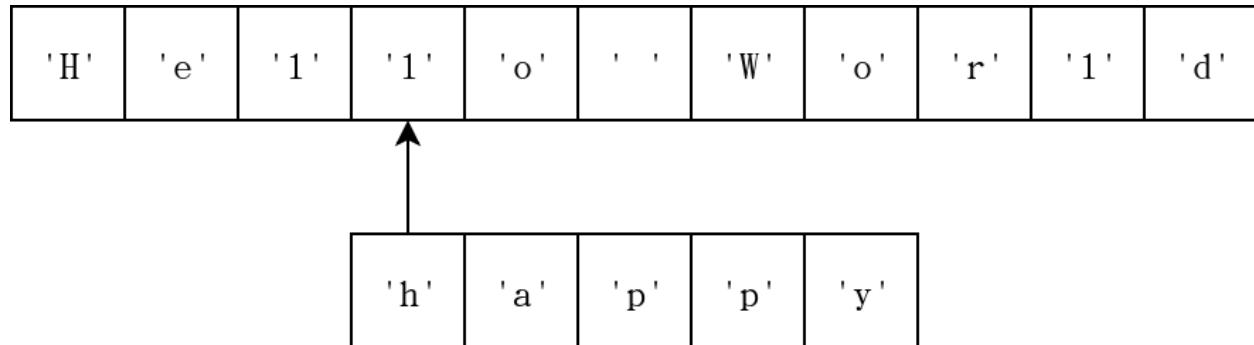
兩字元有可能部分相等，但兩者不等長。所以我們在第 19 行至第 21 行檢查兩字元陣列的尾端是否相等。

在這裡我們不直接檢查字串長度，因為這樣會多走訪一輪字串。我們先逐一檢查字元是否

相等，在尾段則檢查兩字串長度是否相等。

尋找子字串

尋找子字串的示意圖如下：



本命題的想法相當簡單，我們逐一走訪原字串，在每個位置檢查是否符合子字串。以下是參考實作：

```
#include <assert.h>          /* 1 */
#include <stdbool.h>          /* 2 */

int main(void)                /* 3 */
{
    /* Original string. */      /* 4 */
    char s[] =                 /* 5 */
        "The quick brown fox "
        "jumps over the lazy dog"; /* 6 */
    /* Substring. */           /* 7 */
    char ss[] = "lazy";         /* 8 */

    bool is_found = false;       /* 9 */

    char *p = s;                /* 10 */
    while (*p) {                /* 11 */
        bool temp = true;        /* 12 */

        char *q = p;              /* 13 */
        char *r = ss;              /* 14 */
        while (*q && *r) {        /* 15 */
            if (*q != *r) {        /* 16 */
                temp = false;        /* 17 */
                break;                  /* 18 */
            }                      /* 19 */
        }
    }
}
```

(continues on next page)

(continued from previous page)

```
    q++; r++;          /* 20 */
}

if (!(*r)) {           /* 22 */
    is_found = true;   /* 23 */
    break;             /* 24 */
}

p++;                  /* 26 */
/* 27 */

assert(is_found);     /* 28 */

return 0;              /* 29 */
/* 30 */
```

走訪原字串的迴圈位於第 11 行至第 27 行。在走訪時，我們在每個位置逐一比較子字串是否相符。比較子字串的迴圈位於第 15 行至第 21 行。

要注意每次走訪子字串時，都要重新拷貝字串的位址，才可以重覆走訪。

結語

在本文中，我們學習數個字串相關的基本演算法。學習這些演算法的目的不是要取代內建字串處理函式，而是要在沒有內建函式可用時，有能力自己實作新的函式。

前言

相對於先前介紹的基本型別 (primitive data type)，結構 (structure) 是一種複合型別 (derived data type)，用來表達由多個屬性組成的型別，而這些屬性可以是基本型別或是另一個複合型別所組成。

在我們先前的程式中，大多都僅使用基本型別，透過結構，我們可以創造新的型別。由於 C 沒有內建的物件導向語法，使用指向結構的指標來模擬 C++ (或 Java 或 C#) 的 *this* 指標是相當常見的手法。

宣告結構

使用 `struct` 保留字可以宣告結構，如下例：

```
struct person_t {
    char *name;
    unsigned age;
};

int main(void)
{
    struct person_t p = \
        { "Michelle", 37 };

    return 0;
}
```

但這種初始化結構的方式寫死該結構的屬性的位置，若屬性有更動需一併更動相關程式碼，在軟工觀點上不佳。可以改用以下的方法來初始化結構：

```
struct person_t {
    char *name;
    unsigned age;
};

int main(void)
{
    struct person_t p = {
        .name = "Michelle",
        .age = 37,
    };
}
```

(continues on next page)

(continued from previous page)

```
.age = 37  
};  
  
return 0;  
}
```

透過這種方式，不用寫死結構中各個屬性的相對位置，日後要更動屬性時，可將要修改的部分降低。

此外，可以搭配 `typedef` 來簡化結構的型別名稱：

```
/* Forward declaration. */  
typedef struct person_t person_t;  
  
struct person_t {  
    char *name;  
    unsigned age;  
};  
  
int main(void)  
{  
    person_t p = {  
        .name = "Michelle",  
        .age = 37  
    };  
  
    return 0;  
}
```

如果想節省命名空間，可進一步使用以下方式來宣告結構：

```
typedef struct {  
    char *name;  
    unsigned age;  
} person_t;
```

這時候的結構是匿名結構 (anonymous structure)，不會占用命名空間。這是 C11 新增的特性。

存取結構內屬性

使用 .(點號) 可存取結構內的屬性，如下例：

```
#include <assert.h>

typedef struct point_t point_t;

struct point_t {
    double x;
    double y;
};

int main(void)
{
    point_t pt = {
        .x = 3.0,
        .y = 4.0
    };

    assert(3 == pt.x);
    assert(4 == pt.y);

    return 0;
}
```

內嵌在結構內的結構

由於結構視為一種型別，我們可以在結構內嵌入另一個結構，如下例：

```
#include <assert.h>
#include <string.h>

typedef struct person_t person_t;

struct person_t {
    char *name;
    unsigned age;
};

typedef struct employee_t employee_t;

struct employee_t {
```

(continues on next page)

(continued from previous page)

```

person_t p;
double salary;
};

int main(void)
{
    employee_t ee = {
        .p = {
            .name = "Michelle",
            .age = 37
        },
        .salary = 100.0
    };

    assert(
        0 == strcmp(ee.p.name, "Michelle"));
    assert(37 == ee.p.age);
    assert(100.0 == ee.salary);

    return 0;
}

```

但結構內不能嵌入同一個結構，也就是結構不能遞迴宣告。相對來說，結構內可以放入指向同一結構的指針，在資料結構上常見這種寫法。

儲存結構的陣列

由於結構在整體上視為一個新型別，我們可以用陣列來儲存多個結構，如下例：

```

#include <assert.h>
#include <stddef.h>
#include <stdio.h>

typedef struct point_t point_t;

struct point_t {
    double x;
    double y;
};

int main(void)
{
    point_t pts[] = {

```

(continues on next page)

(continued from previous page)

```

{ .x = 0.0, .y = 0.0 },
{ .x = 1.0, .y = 2.0 },
{ .x = 3.0, .y = 4.0 }
};

for (size_t i = 0; i < 3; i++) {
    printf("(%.2f, %.2f)\n",
           pts[i].x, pts[i].y);
}

return 0;
}

```

宣告指向指標的結構

以下是一個宣告結構指標 (pointer to struct) 的簡單實例：

```

#include <stdlib.h>

typedef struct point_t point_t;

struct point_t {
    double x;
    double y;
};

int main(void)
{
    point_t *pt = \
        malloc(sizeof(point_t));
    if (!pt)
        return 1;

    free(pt);

    return 0;
}

```

由於我們從堆積 (heap) 動態配置記憶體，在程式尾段要記得將記憶體釋放掉。

有些 C 語言教材會將型別名稱進一步簡化，如下例：

```

typedef struct point_t point_t;
typedef point_t * point_p;

```

這不是硬性規定，僅為個人風格；筆者目前沒有採用這種方式，因為這樣的別名會讓人忘了該型別其實是指標；如果要這樣做，建議在尾端加上 `_p` 或 `Ptr` 等可以辨識的型別名稱。

存取結構指標的屬性

存取結構指標內的屬性有兩種方式：

- 解指標 (dereference)
- 使用 `->` (箭號)

第二種方式在語法上比較簡潔，故較受歡迎，可以當做一種語法糖。在我們這個例子中，我們兩種方法都使用，供讀者參考：

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

typedef struct point_t point_t;

struct point_t {
    double x;
    double y;
};

int main(void)
{
    point_t *pt = \
        malloc(sizeof(point_t));
    if (!pt)
        return 1;

    /* Init x and y
       with dereference. */
    (*pt).x = 0.0;
    (*pt).y = 0.0;

    /* Access fields of pt
       with dereference. */
    assert(0.0 == (*pt).x);
    assert(0.0 == (*pt).y);

    /* Mutate x and y with `->` */
    pt->x = 3.0;
    pt->y = 4.0;
}
```

(continues on next page)

(continued from previous page)

```

/* Access fields of pt
   with `->` */
assert(3.0 == pt->x);
assert(4.0 == pt->y);

free(pt);

return 0;
}

```

(選讀案例) 無函式的堆疊操作

本節不是必需的，若覺得沒有需求或過於艱難可先略過不讀。

我們一般在練習資料結構和演算法時，我們都會將程式碼包在函式(或方法)中，因為我們心中假定這些資料結構和演算法是可重覆利用的程式碼區塊，事實上也是如此。

但初學者一開始時可以嘗試直接將資料結構或演算法的步驟直接寫在主程式中，在某些情境下這樣做反而會比較簡單，日後在改寫成函式時也可以了解兩者的差異。如果已經熟悉 C 核心語法的讀者，倒不需要刻意這樣做。

由於整個程式碼略長，我們將完整的程式碼在這裡⁸⁰展示，有需要的讀者可自行前往觀看。接下來，我們會分段說明。

堆疊 (stack) 是一種先進後出 (FILO, 即 first-in, last-out) 的線性 (linear) 資料結構，可以想成是一個桶子，先放進去的東西會放置在下面，後放進去的東西會放置在上方。

典型的 C 資料結構會採取以下的方式來宣告堆疊型別：

```

typedef struct node_t node_t;

struct node_t {
    int data;
    node_t *next;
};

typedef struct stack_t stack_t;

struct stack_t {
    node_t *top;
};

```

`node_t` 物件是用來儲存資料的盒子，而 `stack_t` 型別會將 `node_t` 物件包在裡面，日後用函式寫資料結構時，`node_t` 物件不會外露。

⁸⁰ <https://gist.github.com/cwchentw/4ee8500df8d46afecec61aa16bcd7123>

在我們這個例子中，我們採用單向連結串列 (singly-linked list) 將堆疊串起來，如果讀者一開始不知道這是什麼意思也無妨，這只是描述某種串連節點的方式的術語。

由這個例子可以發現，雖然結構不能內嵌同一結構，但可內嵌指向同一結構的指標。

有些 C 語言教材會採用以下策略：

```
typedef struct node_t node_t;

struct node_t {
    int data;
    node_t *next;
};

typedef node_t *stack_t;
```

這樣的做法就是將指向 node_t 的指標外露，在這個例子是可行的；然而，在許多資料結構，會用到超過一項的屬性，這個方法就行不通。讀者可以自行練習用這種方法改寫本範例。

首先，我們建立堆疊物件 st：

```
/* Program state. */
bool failed = false;

/* Create a `stack_t` object. */
stack_t *st = malloc(sizeof(stack_t));
if (!st)
    return EXIT_FAILURE;

st->top = NULL;
```

同樣也是用 malloc 搭配 sizeof 即可配置記憶體。至於 failed 僅是用來表示整個程式運作狀態的旗標，和堆疊操作無關。

接著，建立並插入第一個節點：

```
/* Insert an element. */
{
    /* Create a new node. */
    node_t *node = malloc(sizeof(node_t));
    if (!node) {
        perror("Failed to allocate a node");
        failed = true;
        goto STACK_FREE;
    }

    node->data = 9;
```

(continues on next page)

(continued from previous page)

```

node->next = NULL;

/* Point st->top to node. */
st->top = node;
}

```

在這段程式碼中，我們刻意將整段程式碼包在一個區塊中，這是為了減少命名空間的汙染；簡單地說，變數 `node` 的名稱在這個區塊結束後就無效，不會影響到後續的程式碼。

其實配置記憶體的動作是有可能失敗的，所以要考慮失敗時的處理方法。在本程式中，我們在配置記憶體失敗時，就將先前已配置的記憶體釋放掉，並以失敗狀態結束本程式。在這個例子中，我們優雅地用 `goto` 來減少重覆的程式碼，讀者可追蹤本例完整的程式碼即知其使用方式。

接著，我們檢查堆疊頂端的資料：

```

/* Peek top element. */
if (st->top->data != 9) {
    perror("It should be 9");
    failed = true;
    goto STACK_FREE;
}

```

我們以撰寫測試程式的精神撰寫此段程式碼，當程式不符預期時就進行結束程式相關的動作。

我們中間有略過一些程式碼。接著來看從堆疊中取出節點的方法：

```

/* Pop top element. */
do {
    node_t *curr = st->top;
    if (!curr)
        break;

    /* Update st->top. */
    st->top = curr->next;

    if (curr->data != 5) {
        perror("It should be 5");
        failed = true;
        goto STACK_FREE;
    }

    free(curr);
} while (0);

```

我們在這裡用單次執行的 `do ... while` 區塊包住程式碼，因為在 `st->top` 為 `NULL` 時

可以利用 `break` 直接結束這段程式碼。

在取出該節點後，我們要更新 `st->top` 所指的節點。另外，取出的節點之後沒有指標會指到，故需在此處即釋放其記憶體。

最後，我們展示釋放堆疊物件的程式碼：

```
STACK_FREE:  
{  
    node_t *curr = st->top;  
    node_t *temp;  
    while (curr) {  
        temp = curr;  
        curr = curr->next;  
        free(temp);  
    }  
  
    free(st);  
}
```

如果我們先前提過的，釋放記憶體要由內而外。我們先將內側的節點逐一釋放，最後將外部的堆疊物件釋放掉。

透過本節的範例，我們不僅可以學會指標函式的使用方式，也可以使用一個替代性的方法來練習資料結構和演算法。

然而，從本程式可觀察到，隨著程式變長，不免開始出現一些重覆的程式碼，這也是為什麼我們要用函式(或方法)包覆程式以減少重覆的程式碼。本節的方法僅止於早期程式短小時的練習，之後還是要用函式和模組等手法組織程式碼。

前言

聯合 (union)乍看和結構 (structure)有點像，但聯合內的屬性共用同一塊記憶體，故同一時間內僅能用聯合內其中一種屬性。聯合主要用來表示同概念但不同資料類型的實體。

宣告聯合

使用 union 保留字可宣告聯合，如下例：

```
union sample_t {
    float f;
    int i;
    char ch;
};

int main(void)
{
    union sample_t s;

    s.i = 3;

    return 0;
}
```

我們可以用 typedef 來簡化聯合的型別名稱：

```
/* Forward declaration. */
typedef union sample_t sample_t;

union sample_t {
    float f;
    int i;
    char ch;
};

int main(void)
{
    sample_t s;
```

(continues on next page)

(continued from previous page)

```
s.f = 3.0;

return 0;
}
```

如果想節省命名空間的符號量，可改用以下方法來宣告：

```
typedef union {
    float f;
    int i;
    char ch;
} sample_t;
```

這時候的聯合是匿名聯合 (anonymous union)，故不占用命名空間。這是 C11 新增的特性。

存取聯合中的元素

我們先前提過，聯合在同一時間同僅能儲存其中一個屬性，故以下程式會引發錯誤：

```
#include <assert.h>

typedef union sample_t sample_t;

union sample_t {
    float f;
    int i;
    char ch;
};

int main(void)
{
    sample_t s;

    s.i = 3;

    assert(s.i == 3);

    /* Update `s` */
    s.f = 5.0;

    assert(s.f == 5.0);
```

(continues on next page)

(continued from previous page)

```
/* Error! */
assert(s.i == 3);

return 0;
}
```

內嵌在結構內的聯合

聯合和結構相似，都是一種複合型別，我們可以在結構內嵌入聯合，這時候的好處在於我們可以用一個額外的欄位來記錄目前聯合中使用的型別，如以下實例：

```
#include <stddef.h>
#include <stdio.h>

typedef union amount_t amount_t;

union amount_t {
    unsigned unit;
    float liter;
};

typedef struct item_t item_t;

struct item_t {
    char *name;
    unsigned short amountType;
    amount_t howmuch;
};

int main(void)
{
    item_t books = {
        .name = "C Programming Tutorial",
        .amountType = 1,
        .howmuch.unit = 4
    };

    item_t apples = {
        .name = "Apple",
        .amountType = 1,
        .howmuch.unit = 6
    };
}
```

(continues on next page)

(continued from previous page)

```

item_t juices = {
    .name = "Orange Juice",
    .amountType = 2,
    .howmuch.liter = 3.2
};

item_t items[] = {books, apples, juices};

for (size_t i = 0; i < 3; i++) {
    printf("%s: ", items[i].name);

    if (1 == items[i].amountType) {
        printf("%d units",
               items[i].howmuch.unit);
    } else {
        printf("%.2f liters",
               items[i].howmuch.liter);
    }

    printf("\n"); /* trailing newline. */
}

return 0;
}

```

內嵌在聯合內的結構

除了聯合可嵌在結構內，結構也可以嵌在聯合內。不過，我們為了記錄聯合所用的型別，外部會再用一層結構包住該聯合，就會形成三層的複合型別，如下例：

```

#include <stddef.h>
#include <stdio.h>

typedef struct rgb_t rgb_t;

struct rgb_t {
    unsigned short r;
    unsigned short g;
    unsigned short b;
};

typedef union color_data_t color_data_t;

```

(continues on next page)

(continued from previous page)

```
union color_data_t {
    char *description;
    rgb_t rgb;
};

typedef struct color_t color_t;

struct color_t {
    unsigned short type;
    color_data_t data;
};

int main(void)
{
    color_t red = {
        .type = 1,
        .data.description = "red"
    };

    color_t orange = {
        .type = 1,
        .data.description = "orange"
    };

    color_t beige = {
        .type = 2,
        .data.rgb = { 245, 245, 220 }
    };

    color_t colors[] = { red, orange, beige };

    for (size_t i = 0; i < 3; i++) {
        if (colors[i].type == 1) {
            printf("%s\n",
                   colors[i].data.description);
        } else {
            printf("(%u, %u, %u)\n",
                   colors[i].data.rgb.r,
                   colors[i].data.rgb.g,
                   colors[i].data.rgb.b);
        }
    }

    return 0;
}
```


列舉 (ENUMERATION)

前言

列舉 (enum 或 enumeration) 是另一種複合型別，主要是用在宣告僅有有限值的型態，像是一星期內的日期 (day of week) 或是一年內的月份等。透過列舉，我們可以在程式中定義數個獨一無二的符號 (symbol)。

宣告列舉

使用 enum 保留字可以宣告列舉，如下例：

```
enum direction {
    North,
    South,
    East,
    West
};

int main(void)
{
    enum direction dest = East;

    return 0;
}
```

列舉同樣可用 typedef 簡化型別名稱，如下例：

```
/* Forward declaration. */
typedef enum direction Direction;

enum direction {
    North,
    South,
    East,
    West
};

int main(void)
{
```

(continues on next page)

(continued from previous page)

```
Direction dest = East;

return 0;
}
```

由於 C 語言的列舉本身沒有前綴，可以自行加入前綴，如下例：

```
typedef enum direction Direction;

/* Enum with prefix. */
enum direction {
    Direction_North,
    Direction_South,
    Direction_East,
    Direction_West
};

int main(void)
{
    Direction dest = Direction_East;

    return 0;
}
```

雖然前綴不是強制規定，許多程式設計者偏好此種風格，以減少命名空間衝突。

有些程式設計者會將列舉用全大寫來表示：

```
enum direction {
    DIRECTION_NORTH,
    DIRECTION_SOUTH,
    DIRECTION_EAST,
    DIRECTION_WEST
};
```

這種觀點將列舉視為一種常數 (constant)，實際上也是如此。這種寫法不是硬性規定，而是一種撰碼風格，讀者可自由選用。

讀取列舉的數字

一般來說，我們使用列舉時，將其視為一種符號，不會在意其內部的數值；但必要時也可指定列舉的值，如下例：

```
#include <assert.h>

enum mode {
    MODE_READ = 4,
    MODE_WRITE = 2,
    MODE_EXEC = 1
};

int main(void)
{
    assert(6 == (MODE_READ ^ MODE_WRITE));

    return 0;
}
```

在這個例子中，我們刻意安排列舉的值，透過二進位運算，就可以把列舉的項目視為旗標(flag) 使用。

列舉不具有型別安全

以下理當要引發錯誤的程式碼，其實是「正確」的：

```
typedef enum direction_t direction_t;

enum direction_t {
    DIRECTION_NORTH,
    DIRECTION_SOUTH,
    DIRECTION_EAST,
    DIRECTION_WEST,
};

int main(void)
{
    /* Wrongly correct. */
    direction_t d = 6;

    return 0;
}
```

這是因為 C 語言的列舉在內部是以 `int` 儲存，而且整數值會自動轉型成相對應的列舉型別。由於這項奇異的特性是 C 標準的一部分，為了相容性考量，基本上是不會修改的。

有一派的程式人直接放棄列舉，改用巨集宣告：

```
/* `unsigned char` is
   a small-range number. */
typedef unsigned char DIRECTION;

#define DIRECTION_NORTH 0
#define DIRECTION_SOUTH 1
#define DIRECTION_EAST  2
#define DIRECTION_WEST  3
```

使用巨集未嘗不可。但不論使用列舉或是巨集，我們的目的都是在創造符號，而且兩者都不具有型別安全。

另一個替代的方式是改用結構體包住列舉。將本節的列舉改寫如下：

```
typedef struct direction_t direction_t;

struct direction_t {
    enum {
        _DIRECTION_NORTH,
        _DIRECTION_SOUTH,
        _DIRECTION_EAST,
        _DIRECTION_WEST,
    } value;
};

const direction_t \
DIRECTION_NORTH = { _DIRECTION_NORTH };
const direction_t \
DIRECTION_SOUTH = { _DIRECTION_SOUTH };
const direction_t \
DIRECTION_EAST = { _DIRECTION_EAST };
const direction_t \
DIRECTION_WEST = { _DIRECTION_WEST };
```

由於整數不會轉型成結構體，這樣做的確可以達到型別安全的要求。但是結構體間無法直接比較，程式人得自己寫工具函式來比較。這樣的工具函式不會太難寫：

```
bool is_direction_equal(
    direction_t a,
    direction_t b)
{
    return a.value == b.value;
}
```

如果沒很在意型別安全的議題，就不必要特地寫這樣的程式碼。因為這樣寫程式碼變長，而且效能不會比直接用常數來得好。

前言

在先前的文章中，絕大部分的程式的程式碼全都寫在主函式裡，在規模短小的程式這樣子做並沒有什麼不好，但隨著程式規模成長，這種模式就漸漸行不通了。這時候，我們會利用函式將程式碼分離開來。

使用函式的益處

函式 (function) 是程序抽象化 (procedure abstraction) 的實踐方式，使用函式有以下的好處：

- 減少撰寫重覆的程式碼
- 將程式碼以有意義的方式組織起來
- 在相同的流程下，可藉由參數調整程式的行為
- 藉由函式庫可組織和分享程式碼
- 做為資料結構 (data structures) 和物件 (objects) 的基礎

宣告函式

C 不使用額外的保留字來宣告函式，而是用固定的格式來宣告函式。其格式可參考以下虛擬碼：

```
return_type function_name(parameters)
{
    /* Function body. */
}
```

函式包含以下數個部分：

- 函式的名稱 (identifier)
- 函式的參數 (parameters)，相當於輸入
- 函式的回傳值 (return value)，相當於輸出
- 函式的本體 (body)

電腦程式中的函式會改變程式的狀態，不像數學的函式那麼純粹 (pure)。

只看虛擬碼會覺得有點抽象，我們看幾個範例。以下是一個指數運算的函式：

```
double power(double base, int expo)
{
    assert(base);

    if (expo == 0)
        return 1;

    double result = 1.0;

    if (expo > 0) {
        for (int i = 0; i < expo; ++i)
            result *= base;
    }
    else if (expo < 0) {
        for (int i = 0; i < -expo; ++i)
            result /= base;
    }

    return result;
}
```

這個函式的

- 名稱是 power
- 參數有兩個，分別是 base (double 型態) 和 expo (int 型態)
- 回傳值的型態是 double

使用實例如下：

```
assert(power(3, 2) == 9);
assert(
    fabs(power(3, -2) - 1.0/9.0)
    < 0.000001);
```

我們另外看一個打招呼程式：

```
#include <stdbool.h>          /* 1 */
#include <stdlib.h>             /* 2 */
#include <stdio.h>              /* 3 */
#include <string.h>              /* 4 */

char * hello(char name[])
{                                     /* 5 */
    /* 6 */
```

(continues on next page)

(continued from previous page)

```

char s[] = "Hello ";           /* 7 */
size_t sz_s = strlen(s);      /* 8 */
size_t sz_n = strlen(name);   /* 9 */

char *out = \
    malloc(
        (sz_s + sz_n + 1)
        * sizeof(char));       /* 10 */
if (!out)                      /* 11 */
    return out;                /* 12 */

for (size_t i = 0;
    i < sz_s;
    i++)                         /* 13 */
    out[i] = s[i];               /* 14 */

for (size_t i = 0;
    i < sz_n;
    i++)                         /* 15 */
    out[i+sz_s] = name[i];      /* 16 */

out[(sz_s+sz_n)] = '\0';        /* 17 */

return out;                  /* 18 */
}                                /* 19 */

int main(void)              /* 20 */
{
    char *s_a = hello("Foo");   /* 21 */
    if (!s_a)                  /* 22 */
        goto ERROR;            /* 23 */

    printf("%s\n", s_a);        /* 24 */
    /* 25 */

    char *s_b = hello("Bar");  /* 26 */
    if (!s_b)                  /* 27 */
        goto ERROR;            /* 28 */

    printf("%s\n", s_b);        /* 29 */
    /* 30 */

    char *s_c = hello("Qux");  /* 31 */
    if (!s_c)                  /* 32 */
        goto ERROR;

```

(continues on next page)

(continued from previous page)

```

printf("%s\n", s_c);           /* 33 */
free(s_c);                    /* 34 */
free(s_b);                    /* 35 */
free(s_a);                    /* 36 */

return 0;                      /* 37 */

ERROR:                         /* 38 */
if (s_c)                        /* 39 */
    free(s_c);                  /* 40 */

if (s_b)                        /* 41 */
    free(s_b);                  /* 42 */

if (s_a)                        /* 43 */
    free(s_a);                  /* 44 */

return 1;                        /* 45 */
}                                /* 46 */

```

在這個例子中，第 5 行至第 19 行的部分是函式 `hello()`。這個函式類似於一個自製的字串相接函式，只是第一段文字皆為 "Hello "。

第 20 行至第 46 行的部分為主函式。在主函式中，我們分別在第 22 行、第 26 行、第 30 行呼叫 `hello()` 函式，每次皆傳入相異的參數。

由於這個版本的 `hello()` 回傳一個由堆積 (heap) 動態配置的字串，不能直接將其導向 `printf()`，要用字元指標去接，之後透過該指標才能順利釋放記憶體。我們在第 34 行至第 36 行間逐一釋放掉配置的記憶體。

此外，本例使用 `goto` 來簡化錯誤處理的流程。

如果要更泛用，就是直接寫一個自製的 `strcat()` 函式，傳入兩個字串為參數，將其相接成一個字串，讀者可自行嘗試。

使用 `const` 修飾字防止不當修改

在函式參數中加入 `const` 修飾字，可以防止程式修改這個參數的值。由於函式會拷貝值，這個修飾詞用在函式參數時，對基本型別沒有意義，會用於指標型別。如下例：

```

bool stack_is_empty(const stack_t *self)
{
    assert(self);
}

```

(continues on next page)

(continued from previous page)

```

return !(self->top) ? true : false;
}

```

函式原型

在先前的例子中，長長的函式後才接到主程式程式碼，其實不是很好閱讀，如果函式一多這情形會更嚴重。利用 C 語言的函式原型 (function prototype) 可以改善這個現象。將上述的 C 程式碼以函式原型改寫：

```

#include <stdbool.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/* Declare function prototype. */
char * hello(char []);

int main(void)
{
    /* Implement our main program
       here. */
}

char * hello(char name[])
{
    /* Implement the function
       `hello()` here. */
}

```

為了突顯函式原型的部分，我們將所有的實作部分拿掉。在這個 C 虛擬碼中，我們在程式碼上方加上函式原型的宣告，就不需要直接實作 hello 函式，可以先寫主函式，再程式碼下方補上 hello 的實作即可。

在撰寫程式時，建議將主要的程式寫在最上方，而將實作內容寫在下方，這樣在閱讀程式碼時，就可以由上而下逐漸追到實作細節。剛好 C 語言的函式原型可以協助我們達到這個撰碼方式。這並不是筆者隨意提出的想法，讀者可在一些軟工的書籍看到類似的思維。

在撰寫函式原型時，參數不需加名稱，只要寫入型別標註 (annotation) 即可；有些 C 程式碼會在函式原型加參數名稱，基本上僅是為了閱讀方便，函式原型的參數名稱不需和函式實作的參數名稱相同。

不定參數函式

有時候我們會看到這樣的函式：

```
double max(double a, double b)
{
    return a > b ? a : b;
}
```

這種函式，僅能固定接收兩個參數，通用性不是很好。我們希望可以做到這樣：

```
/* C pseudocode. */

/* max with 3 items. */
m = max(3.0, 1.0, 2.0);

/* max with 5 items. */
n = max(4.0, 2.0, 1.0, 5.0, 3.0);
```

目前的 C 語言無法做到上述虛擬碼的樣子，但也相差不遠，大概可以做到下面這個例子：

```
/* C pseudocode. */

/* Def: max(size, n_1, n_2, ...) */
/* Call max() with 3 items. */
m = max(3, 3.0, 1.0, 2.0);

/* Call max() with 5 items. */
n = max(5, 4.0, 2.0, 1.0, 5.0, 3.0);
```

以下是實例：

```
#include <assert.h>                      /* 1 */
#include <stdarg.h>                        /* 2 */
#include <stddef.h>                         /* 3 */

double max(
    size_t sz,
    double value,
    ...);                                     /* 4 */

int main(void)                                /* 5 */
{
    /* Call max() with 3 items. */           /* 6 */
    assert(
```

(continues on next page)

(continued from previous page)

```

7.0 == max(3, 7.0, 4.0, 6.0)); /* 8 */

/* Call max() with 5 items. */ /* 9 */
assert(
    5.0 == max(5, 4.0, 2.0, 1.0,
                5.0, 3.0)); /* 10 */

return 0; /* 11 */
/* 12 */

double max(
    size_t sz,
    double value,
    ...) /* 13 */
{
    /* Declare args. */ /* 14 */
    va_list args; /* 15 */

    /* Get the first item. */ /* 16 */
    va_start(args, value); /* 17 */

    double max = value; /* 18 */
    double temp; /* 19 */
    for (size_t i = 1; i < sz; i++) { /* 20 */
        /* Get each subsequent item. */ /* 21 */
        temp = va_arg(args, double); /* 22 */

        max = max > temp ? max : temp; /* 23 */
    } /* 24 */

    /* Clean args. */ /* 25 */
    va_end(args); /* 26 */

    return max; /* 27 */
} /* 28 */
/* 29 */

```

第 4 行為函式原型，其目的是要將函式本體下移，就可以把主要的程式寫在上方。

第 5 行至第 12 行為主函式。值得注意的部分在兩次呼叫 `max()` 函式時參數的數量是相異的。

第 13 行至第 29 行的部分為 `max()` 函式的實作。這個函式的關鍵在於使用不定參數的特性來接收不等數量的參數。

不定參數函式需要 `stdarg.h` 函式庫的協助，過程如下：

- 以 `va_list` 宣告代表不定參數的變數，位於第 16 行

- 以 `va_start` 取得第 1 項參數，位於第 18 行
- 以 `va_arg` 取得之後的參數，位於第 19 至第 25 行
- 以 `va_end` 清理該變數，位於第 27 行

由於不定參數本身無法預先取得參數數量的資訊，要由外部傳入；另外，也要補足參數型別的資訊，像是我們在第 23 行的指令。

遞迴函式

遞迴 (recursion) 是指將某個問題分解成更小的子問題來解決該問題的方式。由於 C 語言有實作遞迴，有些問題就可以透過遞迴很優雅地解決掉。

初心者一開始往往不知道遞迴函式如何寫，基本上有兩個要件：

- 終止條件
- 縮小問題的方式

我們透過 Fibonacci 數⁸¹來看遞迴函式怎麼寫，這是一個常見的例子：

```
#include <assert.h>          /* 1 */
#include <stddef.h>           /* 2 */

typedef unsigned int uint;    /* 3 */

/* Function prototype. */     /* 4 */
uint fib(uint);             /* 5 */

int main(void)             /* 6 */
{
    uint arr[] = \           /* 7 */
        {0, 1, 1, 2, 3,       /* 8 */
         5, 8, 13, 21};      /* 8 */

    for (size_t i = 0; i < 9; i++) /* 9 */
        assert(fib(i+1) == arr[i]); /* 10 */

    return 0;                /* 11 */
}                                /* 12 */

uint fib(uint n)           /* 13 */
{
    /* `n` starts from 1. */   /* 14 */
    assert(n > 0);            /* 15 */
                                /* 16 */
}
```

(continues on next page)

⁸¹ https://en.wikipedia.org/wiki/Fibonacci_number

(continued from previous page)

```

if (n == 1) /* 17 */
    return 0; /* 18 */

if (n == 2) /* 19 */
    return 1; /* 20 */

return fib(n - 1) \
+ fib(n - 2); /* 21 */
} /* 22 */

```

第 6 行至第 12 行的部分為主函式。在這裡我們檢查前 9 個 Fibonacci 數，確認 `fib()` 函式的值是正確的。

第 13 行至第 22 行的部分是 `fib()` 函式的實作。這個函式的重點在於使用遞迴讓函式更加簡潔。

`fib()` 的終止條件有兩個：

- n 等於 1 時，回傳 0，位於第 17 行至第 18 行
- n 等於 2 時，回傳 1，位於第 19 行至第 20 行

除了這個條件外，碰到其他的 n 就用遞迴呼叫逐漸將問題縮小到前述條件，位於第 21 行。

我們人工追蹤一下這個函式，來拆解遞迴函式。當 n 等於 1 時，結果如下：

```
fib(1) -> 0
```

當 n 等於 2 時，結果如下：

```
fib(2) -> 1
```

當 n 等於 3 時，結果如下：

```

fib(3) -> fib(2) + fib(1)
-> 1 + 0
-> 1

```

當 n 等於 4 時，結果如下：

```

fib(4) -> fib(3) + fib(2)
-> (fib(2) + fib(1)) + 1
-> (1 + 0) + 1
-> 2

```

當 n 等於 5 時，結果如下：

```
fib(5) -> fib(4) + fib(3)
-> (fib(3) + fib(2)) + (fib(2) + fib(1))
-> ((fib(2) + fib(1)) + 1) + (1 + 0)
-> ((1 + 0) + 1) + 1
-> 3
```

還可以再繼續追蹤下去，有興趣的讀者可以自行嘗試。

由於遞迴在電腦科學中很重要，有空時最好自行練習一下。以下是一些常見的例子：

- 階乘 (factorial)
- Fibonacci 數
- 最大公因數 (greatest common divisor)
- 二元搜尋樹 (binary search tree)
- 河內塔 (towers of hanoi)
- 走訪特定資料夾內的所有資料夾和檔案

傳值呼叫 vs. 傳址呼叫

我們想寫一個將兩數互換的函式，但這樣寫行不通：

```
#include <assert.h>

/* Useless swap. */
void swap(int, int);

int main(void)
{
    int a = 3;
    int b = 4;

    /* No real effect. */
    swap(a, b);

    /* Error! */
    assert(a == 4);
    assert(b == 3);

    return 0;
}

void swap(int a, int b)
```

(continues on next page)

(continued from previous page)

```
{
    int temp = a;
    a = b;
    b = temp;
}
```

這牽涉一些從語法上看不出來的函式的運行期行為。我們這次加上一些額外的訊息：

```
#include <assert.h>
#include <stdio.h>

/* Useless swap. */
void swap(int, int);

int main(void)
{
    int a = 3;
    int b = 4;

    fprintf(
        stderr,
        "a in main, before swap: %d\n",
        a);
    fprintf(
        stderr,
        "b in main, before swap: %d\n",
        b);

    /* No real effect. */
    swap(a, b);

    fprintf(
        stderr,
        "a in main, after swap: %d\n",
        a);
    fprintf(
        stderr,
        "b in main, after swap: %d\n",
        b);

    return 0;
}

void swap(int a, int b)
{
```

(continues on next page)

(continued from previous page)

```

fprintf(
    stderr,
    "a in swap, before swap: %d\n",
    a);
fprintf(
    stderr,
    "b in swap, before swap: %d\n",
    b);

int temp = a;
a = b;
b = temp;

fprintf(
    stderr,
    "a in swap, after swap: %d\n",
    a);
fprintf(
    stderr,
    "b in swap, after swap: %d\n",
    b);
}

```

印出訊息如下：

```

a in main, before swap: 3
b in main, before swap: 4
a in swap, before swap: 3
b in swap, before swap: 4
a in swap, after swap: 4
b in swap, after swap: 3
a in main, after swap: 3
b in main, after swap: 4

```

我們發現 `a` 和 `b` 在 `swap` 函式中的確有互換，但在函式結束後卻沒有實際的效果。這是因為函式在傳遞參數時，並不是直接將參數傳進去函式內部，而是將其複製一份後傳入。以本例來說，我們只是交換 `a` 和 `b` 的複製品而已。

將函式的參數修改成傳遞指標，如下：

```

#include <assert.h>
#include <stdio.h>

/* It really works. */
void swap(int *, int *);

```

(continues on next page)

(continued from previous page)

```

int main(void)
{
    int a = 3;
    int b = 4;

    swap(&a, &b);

    assert(a == 4);
    assert(b == 3);

    return 0;
}

void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

```

這個程式如同我們預期的方式來運作。

有些 C 語言教材會用傳值呼叫 (pass by value) 和傳址呼叫 (pass by reference) 來區分；但實際上 C 的函式呼叫皆為傳值呼叫，只是在傳遞指標時的「值」是記憶體位址，簡單地說，C 在傳指標時，將指標的位址複製一份後傳入函式內部。

回傳指標

我們想寫一個字串相接的函式，但以下程式不會正常運作：

```

#include <assert.h>          /* 1 */
#include <stddef.h>           /* 2 */
#include <stdio.h>            /* 3 */
#include <stdlib.h>           /* 4 */
#include <string.h>            /* 5 */

/* Useless string concat. */    /* 6 */
char *
my_strcat(char [], char []);    /* 7 */

int main()                  /* 8 */
{
    char *s = \

```

(continues on next page)

(continued from previous page)

```

my_strcat(
    "Hello ", "World");           /* 10 */

printf("%s\n", s);                /* 11 */

return 0;                         /* 12 */
}                                 /* 13 */

char *
my_strcat(char a[], char b[])
{
    /* `s` is a local variable. */ /* 16 */
    char s[256];                 /* 17 */

    strcat(s, a);                /* 18 */
    strcat(s, b);                /* 19 */

    /* `s` is a junk value. */   /* 20 */
    return s;                     /* 21 */
}                                 /* 22 */

```

我們在第 17 行配置一塊型態為字串(字元陣列)的自動記憶體，並試圖在第 21 行回傳該變數。以下是此範例程式的編譯錯誤訊息：

```

returnArr.c:34:5: warning: function returns address of local variable
 ↳ [-Wreturn-local-addr]
return s;

```

在本例中，`my_strcat()` 函式的變數 `s` 從堆疊(stack)自動配置記憶體，在函式結束時，`s` 已經被清除，回傳的值是垃圾值(junk value)，沒有實質意義。

修改成以下版本即可正常運作：

```

#include <stdio.h>                  /* 1 */
#include <stdlib.h>                  /* 2 */
#include <string.h>                  /* 3 */

char *
my_strcat(char [], char []);        /* 4 */

int main(void)                      /* 5 */
{
    char *s = \                      /* 6 */
    my_strcat(                      /* 7 */
        "Hello ", "World");

```

(continues on next page)

(continued from previous page)

```

printf("%s\n", s);           /* 8 */
free(s);                    /* 9 */
return 0;                   /* 10 */
}                           /* 11 */

char * my_strcat(
    char a[], char b[])
{
    size_t sz_a = strlen(a);   /* 14 */
    size_t sz_b = strlen(b);   /* 15 */
    size_t sz = sz_a + sz_b + 1; /* 16 */

    char *s = \
        calloc(sz, sizeof(char)); /* 17 */

    strcat(s, a);             /* 18 */
    strcat(s, b);             /* 19 */

    return s;                 /* 20 */
}                           /* 21 */

```

我們在第 19 行時手動配置字串 (字元陣列) 型態的記憶體 `s`，並在第 22 行回傳。

在本例中，我們回傳字元指標。由於我們從堆積 (heap) 配置記憶體，在 `my_strcat()` 函式結束時，記憶體不會清除，故程式可正常運作。只是主函式結束時要記得手動釋放記憶體。

回傳多個值

C 語言的函式僅能回傳單一值，那麼，當我們需要回傳多個值時要如何處理呢？一個常見的方法就是在參數中使用指標，透過指派至該參數來回傳值。以下是一個確認陣列物件是否有特定值的函式：

```

bool array_contains(
    const array_t *self,
    int value,
    size_t *index)
{
    assert(self);

    for (size_t i = 0;

```

(continues on next page)

(continued from previous page)

```

        i < array_size(self);
        i++)
    {
        if (value
            == array_at(self, i)) {
            *index = i;
            return true;
        }
    }

    *index = 0;
    return false;
}

```

在這個函式中，我們回傳 `bool` 值表示函式中的確含有這個值 `value`，但我們額外用一個指標 `index` 來接收 `value` 所在的索引值。這個函式的使用方式如下：

```

size_t *index = \
    (size_t *) malloc(sizeof(size_t));
if (!index) {
    /* Error handling. */
}

if (!array_contains(arr, 13, index)) {
    /* `index` is invalid here. */
    /* Handle the error. */
}

/* Free `index` later. */

```

另外一個方式是回傳一個含有兩個屬性的結構體：

```

typedef struct result_t result_t;

struct result_t {
    size_t index;
    bool has_value;
};

```

在這個情境下，每次要使用 `index` 前，都要檢查 `has_value` 為真。

函式指標 (Function Pointer)

函式也可以做為型別，透過函式指標可以宣告函式型別。像以下宣告：

```
typedef int (*comp_fn)(int, int);
```

透過這個宣告，我們宣告了 `comp_fn` 型別，該型別是一個函式，接收兩個整數，回傳一個整數。我們以這個型別寫一個實際的例子：

```
#include <assert.h>                                /* 1 */

typedef int (*comp_fn)(int, int);    /* 2 */

int compute(comp_fn, int, int);    /* 3 */
int add(int a, int b);            /* 4 */
int sub(int a, int b);            /* 5 */

int main(void)                  /* 6 */
{
    assert(                                /* 7 */
        7 == compute(add, 3, 4));        /* 8 */
    assert(                                /* 9 */
        -1 == compute(sub, 3, 4));      /* 10 */
    return 0;                      /* 11 */
}

int compute(
    comp_fn fn,
    int a, int b)                /* 12 */
{
    return fn(a, b);              /* 13 */
}                                              /* 14 */

int add(int a, int b)          /* 15 */
{
    return a + b;                /* 16 */
}                                              /* 17 */

int sub(int a, int b)          /* 18 */
{
    return a - b;                /* 19 */
}                                              /* 20 */
```

在第 2 行中，我們宣告函式指標型態的別名 `comp_fn`。接著，在第 3 行中，把 `comp_fn` 當成第一個參數。

如果我們不用別名的話，第 3 行需改寫如下：

```
int compute(int (*)(int, int), int, int);
```

雖然效果相同，但在軟體工程的觀點上，這樣寫比較不好。因為 `int (*)(int, int)` 對程式設計者來說是沒有意義的符號。

在本例中，函式 `compute()` 實際的行為由參數 `fn` 決定，我們只要傳入合於 `comp_fn` 型別的函式即可改變函式 `compute()` 的運作方式。雖然 C 語言不是函數式語言，可以透過這項特性做一些高階函式，我們於後續文章會說明。

我們將上例修改如下：

```
#include <assert.h>          /* 1 */
#include <stdbool.h>         /* 2 */
#include <string.h>           /* 3 */

typedef
int (*comp_fn)(int, int);    /* 4 */

int
compute(char [], int, int);  /* 5 */

int main()                  /* 6 */
{
    assert(
        7 == compute(
            "+", 3, 4));      /* 8 */
    assert(
        -1 == compute(
            "-", 3, 4));      /* 9 */

    return 0;                /* 10 */
}

int add(int a, int b);     /* 12 */
int sub(int a, int b);     /* 13 */

int compute(
    char comp[],
    int a, int b)           /* 14 */
{
    comp_fn fn;             /* 15 */
    /* 16 */

    if (0 == strcmp(
        comp, "+")
        || 0 == strcmp(
            comp, "add"))

```

(continues on next page)

(continued from previous page)

```

{
    fn = add;                      /* 17 */
    /* 18 */
    /* 19 */
} else if (
    0 == strcmp(
        comp, "-")
    || 0 == strcmp(
        comp, "sub"))
{
    fn = sub;                      /* 20 */
    /* 21 */
    /* 22 */
}
else
    assert(
        "No valid comp"
        && false);                  /* 23 */

return fn(a, b);                  /* 25 */
/* 26 */

int add(int a, int b)            /* 27 */
{
    return a + b;                 /* 28 */
    /* 29 */
    /* 30 */
}

int sub(int a, int b)            /* 31 */
{
    return a - b;                 /* 32 */
    /* 33 */
    /* 34 */
}

```

在這個例子中，compute() 傳入的值為字串，該函式藉由傳入的參數動態修改變數 fn 的值，這段動作位於第 17 行至第 26 行。

從這個例子可以看到，函式就像值一樣可傳遞，函數式程式的前提就是建立在此項特性上。

巨集 (MACRO) 或前置處理器 (PREPROCESSOR)

前言

前置處理器是在 C 或 C++ 中所使用的巨集 (macro) 語言。嚴格說來，前置處理器的語法不是 C 語言，而是一個和 C 語言共生的小型語言。在本文中，我們介紹數種常見的前置處理器用法。

閱讀經前置處理器處理過的 C 程式碼

在 C 編譯器中，前置處理器和實質的 C 編譯器是分開的。C 程式碼會經過前置處理器預處理 (preprocessing) 過後，再轉給真正的 C 編譯器，進行編譯的動作。

預處理在本質上是一種字串代換的過程。前置處理器會將 C 程式碼中巨集宣告的部分，代換成不含巨集的 C 程式碼。之後再將處理過的 C 程式碼導給 C 編譯器，進行真正的編譯。

所幸，預處理在一些 C 編譯器中是可獨立執行的步驟。以 GCC 為例，我們可以把前處理這一步獨立出來，觀察預處理後的程式碼。下列的範例指令將程式碼前處理後，用 indent 程式以 K&R 風格重新排版：

```
$ gcc -E -o file.i file.c
$ indent -kr file.i
```

藉由閱讀整理過的 *file.i* 文字檔，我們可以了解前置處理器做了什麼事情，有利於除錯巨集。

用 #include 引入函式庫

#include 敘述用來引入外部函式庫，這算是單純的敘述。在引入函式庫時，有兩種語法可用，如下例：

```
/* Include some standard
   or third-party library. */
#include <stdlib.h>

/* Include some internal library. */
#include "something.h"
```

有些程式會將外部函式庫以一對角括號 < 和 > 將標頭檔名稱括起來，專案內部的模組用則成對雙引號 " 和 "，在視覺上可簡單地區分。這只是撰碼風格，非強制規範。

用 `#define` 宣告巨集

`#define` 敘述用來宣告巨集。這應該是前置處理器中最具可玩性的部分。有些程式人會用巨集寫擬函式，甚至會用巨集創造語法。基本上，用巨集創造語法算是走火入魔了，我們不鼓勵讀者這麼做，知道有這件事即可。

用巨集宣告定值

最簡單的巨集是宣告定值：

```
#define SIZE 10
```

實際上，在轉換後的 C 程式中，並沒有 SIZE 這個變數。每個 SIZE 所在的位置會經前置處理器代換為 10。

承上，我們來看一個相關的範例：

```
#include <stdio.h> /* 1 */

#define SIZE 5 /* 2

int main(void) /* 3
{
    int arr[SIZE]; /* 4

    for (int i = 0; i < SIZE; i++) { /* 6
        arr[i] = i + 3; /* 7
    } /* 8

    for (int i = 0; i < SIZE; i++) { /* 9
        printf("%d\n", arr[i]); /* 10
    } /* 11

    return 0; /* 12
}
```

我們知道陣列不能用變數初始化，但第 2 行所宣告的巨集變數 SIZE 會在前處理時將程式碼中 SIZE 出現的位置轉換成定值 5。實數編譯時陣列的長度是定值 5 而非巨集變數 SIZE，所以程式可正確編譯和運行。

用巨集宣告單行擬函數

稍微進階一點的用法是用巨集寫簡單的擬函式。像是以下的 MAX 巨集：

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

實際上，該巨集會代換為三元運算子，所以可以像函式般回傳值。

使用巨集寫擬函式的好處在於巨集是字串處理，不受到 C 資料型態的限制。所以，用巨集寫出來的擬函式可以模擬(較不安全的)泛型程式。

但巨集是很不牢靠的，像是以下的誤用例：

```
m = MAX(a++, b++);
```

該巨集會展開成以下 C 程式碼：

```
m = ((a++) > (b++) ? (a++) : (b++));
```

由於遞增運算子會隱微地改變變數的狀態，這行程式會產生預期外的錯誤結果。

用巨集宣告多行擬函數

巨集也可以用來跨越多行，這時候就更像函式了。在以下例子中，我們定義一個可重覆執行的區塊：

```
#define TIMES(count, block) { \
    size_t i; \
    for (i = 0; i < (count); ++i) { \
        (block); \
    } \
}
```

此巨集利用區塊創造局部可視域，所以計數器 *i* 不會汙染程式的命名空間。

之後，就可以用比較簡潔的方式重覆執行特定程式碼：

```
TIMES(3, {
    printf("Hello World\n");
});
```

擬函式巨集很容易被濫用，所以要謹慎使用。我們會在後文看到使用擬函數巨集的經典反例。

從多行巨集取得回傳值

巨集不是函式，無法用 `return` 敘述回傳值。單行巨集算是表達式，巧妙地閃開了這個議題。但多行巨集就會碰到這個問題。

應對的方式是在巨集宣告中額外加入一個變數，把該變數當成「回傳值」。參考以下範例程式：

```
#include <assert.h>

#define FAC(n, out) { \
    out = 1; \
    unsigned i; \
    for (i = 1; i <= (n); ++i) \
        (out) = (out) * i; \
}

int main(void)
{
    unsigned out;

    /* Receive `out` from `FAC` the macro. */
    FAC(5, out);

    assert(120 == out);

    return 0;
}
```

在這個範例程式中，`FAC` 巨集用來計算階乘 (factorial)。使用該巨集時，要額外加上變數 `out` 充當該巨集的「回傳值」。雖然會多用一個變數，但巨集使用者可以自行控制變數名稱，所以不是什麼大問題。

(無用) 用巨集創造語法

程式設計者可以利用 `#define` 為 C 語言創造新語法。這種用法算是經典的反模式 (anti-pattern)，所以看看就好，不要深入學習。

例如，我們用巨集將中序運算子「轉換」為前序運算子，就可以在 C 程式碼中「寫」Lisp：

```
/* DON'T DO THIS IN PRODUCTION CODE. */
#include <assert.h>
#include <stdio.h>

typedef unsigned int uint;
```

(continues on next page)

(continued from previous page)

```

/* Arithmetic operators. */
#define ADD(a, b) ((a) + (b))
#define SUB(a, b) ((a) - (b))

/* Relational operators. */
#define GT(a, b) ((a) > (b))
#define LE(a, b) ((a) <= (b))
#define EQUAL(a, b) ((a) == (b))

/* Assignment operator. */
#define SETQ(a, b) ((a) = (b))

/* Function implementation. */
#define DEFUN(fn, t, params, body) \
t fn params { body }

DEFUN(fib, uint, (uint n),
      assert(
        GT(n, 0));
      if (EQUAL(n, 1))
        return 0;
      else if (EQUAL(n, 2))
        return 1;
      else
        return
          ADD(fib(SUB(n, 1)),
              fib(SUB(n, 2))));
      )

DEFUN(main, int, (void),
      uint i;
      for (SETQ(i, 1); LE(i, 20); ++i)
        printf(
          "%u\n", fib(i));
      return 0;
      )

```

在巨集重重包裝下，這個程式已經看不到原本的 C 程式碼。這樣寫事後要除錯很很困難，所以算是經典的誤用例。

泛型型別巨集 (C11)

泛型型別巨集是 **C11** 所引入的新特性，用於模擬泛型程式。為什麼我們說是模擬泛型呢？我們來看以下的泛型 `log10` 函式：

```
#define log10(x) _Generic((x), \
    log double: log10l, \
    float: log10f, \
    default: log10)(x)
```

在此範例程式中，巨集 `log10` 是一個利用泛型型別巨集宣告的擬函式。在我們帶入不同型別的參數時，該巨集會將參數導向適合該參數的型別的函式。藉由這種方式，達成泛型的效果。

也就是說，我們雖然有泛型程式的型別安全，但卻無法真正節省實作的時間。因為我們仍然要針對不同型別重覆實作相同演算法的函式。為什麼 C 標準要引入這樣的特性呢？應該是為了兼具相容性和現代語法所做的妥協吧。

用巨集進行條件編譯

利用巨集中有關條件編譯的語法，我們可以根據不同情境改變巨集的輸出。也就是說，我們可以利用這項特性保留所需的程式碼，去除不需要的程式碼。

避免重覆引入標頭檔

一個經典的實例是標頭檔的 `#include guard`：

```
#ifndef SOMETHING_H
#define SOMETHING_H

/* Declare some data types
   and public functions. */

#endif /* SOMETHING_H */
```

當前置處理器第一次讀到此標頭檔時，`SOMETHING_H` 是未定義的，這時候前置處理器會繼續執行下一行敘述。在下一行我們定義了 `SOMETHING_H`。

反之，當前置處理器第二次讀到此標頭檔時，由於 `SOMETHING_H` 已定義了，前置處理器不會繼續執行後續的內容，巧妙地避開了重覆引入的議題。

使用 `#include guard` 時，標頭檔最尾端的 `#endif` 是要和開頭的 `#ifndef` 成對出現的固定語法。初學者有時會忘了加上去。

在使用 `#include guard` 時，有微小的機會會發生巨集名稱衝突。像是以下的例子：

```
#ifndef UTILS_H
#define UTILS_H

/* Declare some data types
   and public functions. */

#endif /* UTILS_H */
```

C 專案或多或少會有一些無法歸類的工具函式，一個常見的方式是將這些工具函式放在同一個模組中集中管理。而 `UTILS_H` 又是很普遍的名稱，就有可能引發巨集名稱衝突。

替代的方式是在標頭檔第一行加入以下敘述：

```
#pragma once
```

`#pragma once` 在效果上等同於 `#include guard`，但不會引發巨集名稱衝突。雖然 `#pragma once` 非標準 C 語法，許多 C 編譯器都有實作這項功能。除非手上的 C 專案要支援一些冷門的 C 編譯器，可以考慮使用這個語法替代 `#include guard`。

混合 C 和 C++ 程式碼

另外一個經典的例子是 `extern "C"` 敘述：

```
#ifdef __cplusplus
extern "C" {
#endif

/* Some declarations. */

#ifndef __cplusplus
}
#endif
```

`extern "C"` 敘述是 C++ 的語法，用於混合 C 和 C++ 的專案。

C++ 為了處理命名空間 (namespace)、函式重載 (function overloading) 等語法特性，會將函式名稱 mangling。但我們不希望 C++ 編譯器將 C 函式庫的標頭檔內的函式宣告也 mangling，所以我們用 `extern "C"` 敘述告知 C++ 編譯器不要對該區塊內的函式名稱 mangling。

`__cplusplus` 是 C++ 編譯器中才會出現的巨集變數。使用 C 編譯器去讀這段宣告時，不會出現 `extern "C"` 敘述。使用 C++ 編譯器時則會出現該敘述。

由於 `extern "C"` 是一個選擇性的區塊，所以我們用兩段巨集宣告把函式宣告包起來。這已經算是寫 C 和 C++ 混合程式時固定使用的手法了。

偵測宿主系統

條件編譯也常用來偵測編譯程式時的系統環境。如以下的例子：

```
#if defined(_WIN32)
    #define PLATFORM_NAME "Windows"
#elif defined(__CYGWIN__)
    #define PLATFORM_NAME "Cygwin"
#elif defined(__linux__)
    #define PLATFORM_NAME "GNU/Linux"
#elif defined(__APPLE__)
    #define PLATFORM_NAME "Mac"
#elif defined(__unix__)
    #define PLATFORM_NAME "Unix"
#else
    #define PLATFORM_NAME "Other OS"
#endif
```

在這個例子中，我們僅僅用條件編譯來決定 `PLATFORM_NAME` 的值，但我們可以進一步用條件編譯篩選不同系統下的程式碼，撰寫跨平台程式碼。

條件編譯對於撰寫跨平台程式碼相當重要。因為不同系統的 C API 往往不會相等。我們可以用條件編譯的方式，針對不同平台撰寫不同的程式碼，滿足跨平台的需求。

我們的例子中已經列出幾個常見的系統，[這裡⁸²](#)則列出更多系統名稱，有興趣的讀者可以參考一下。

註解掉一大段程式碼

我們還可以用條件編譯來註解掉一段程式碼。如下例：

```
#if 0
    printf("It won't print\n");
#endif
```

由於 `#if 0` 為偽，從 `#if 0` 到 `#endif` 之間的 C 程式碼會被前置處理器自動忽略掉，達到註解的效果。

⁸² <https://sourceforge.net/p/predef/wiki/OperatingSystems/>

用條件編譯輔助除錯

條件編譯常用來輔助除錯。參考以下例子：

```
#ifdef DEBUG
    fprintf(stderr, "Some message\n");
#endif
```

當巨集 DEBUG 為真時，會印出錯誤訊息。反之，則不會印出來。

我們在編譯 C 程式碼時，可以在參數中開啟 DEBUG 宣告：

```
$ gcc -DDEBUG -o file file.c
```

這時候巨集 DEBUG 視為真，印出錯誤訊息。最後程式要發佈前，關掉這個參數再編譯一次即可。

在編譯時期引發錯誤

我們可以在巨集中引發錯誤，中止程式編譯。參考以下實例：

```
#if __unix__
    #error "Unsupported OS"
#endif
```

在這個例子中，假定我們的專案不支援類 Unix 系統，試圖在類 Unix 系統下編譯時會引發錯誤訊息。

引入各編譯器特有的特性

#pragma 敘述開放給各個 C 編譯器，用來自訂新的巨集功能。像是我們先前提到的 #pragma once 就是一個例子。#pragma 敘述的用法在 C 編譯器不統一，得查閱各個 C 編譯器的使用手冊，這裡不多做說明。

預先定義的巨集

在 C 語言中，預先定義好數個巨集，這些訊息和程式本身的資訊相關，可用於除錯等。包括以下巨集：

- `__LINE__`：程式所在的行數
- `__FILE__`：檔案名稱
- `__DATE__`：前置處理器執行的日期
- `__TIME__`：前置處理器執行的時間
- `__STDC__`：確認某個編譯器是否有遵守 C 標準
- `__func__`：函式名稱 (C99)

我們利用這些巨集寫了一個印出錯誤訊息的巨集：

```
#define DEBUG_INFO(format, ...) { \
    fprintf(stderr, "(%s:%d) " format "\n", \
            __FILE__, __LINE__, ##__VA_ARGS__); \
}
```

在這個巨集中，第一個參數當成格式化輸出的模板，第二個以後的參數則是輸入的字串。我們利用兩個內定的巨集 `__FILE__` 和 `__LINE__` 分別印出錯誤訊息所在的檔案名稱和行數，有利於除錯。

巨集對於 C 程式的意義

由於巨集是字串代換，不會有函數呼叫的開銷。但同樣的程式碼區塊會反覆出現，使程式體積變大。過度使用巨集會使得程式難以除錯，不建議為了優化程式刻意把函式改寫成巨集。

結語

在 C 語言中，前置處理器是實用卻易被忽略的特性，許多入門教材不會深入前置處理器的使用方式。可能的原因是巨集難寫、難除錯。然而，只要不過度使用魔術語法，巨集也能成為我們撰寫 C 程式的助力。

函式庫 (LIBRARY)

函式庫和套件的差異

C 語言對於函式庫 (library) 的概念相對簡單，C 函式庫是由標頭檔 (.h) 和二進位檔 (靜態函式庫：.a 檔或 .lib 檔，動態函式庫：.so 檔或 .dylib 檔或 .dll 檔) 所組成。使用 C 函式庫時不需要原始碼，只要有二進位檔即可使用。

近年來流行的開放原始碼 (open-source-software movement) 是軟體授權的模式，這件事本身對使用 C 函式庫不是必要的。

相對來說，C 語言沒有套件 (package) 的概念。我們在類 Unix 系統上看到的套件管理程式 (如 yum 或 apt 等) 算是後設的概念，而非 C 語言本身的功能。

早期的 Windows 並不注重 C (或 C++) 套件的議題，在分享 C (或 C++) 函式庫時就沒有那麼方便，有些第三方方案，像是 Conan⁸³，企圖解決套件相關的議題；近年來 C++ 重新抬頭，微軟推出 vcpkg⁸⁴，也是另一個 C (或 C++) 套件的方案。

標準函式庫和第三方函式庫

一般 C 入門教材對於函式庫的概念僅止於 C 標準函式庫，而不注重第三方 C 函式庫的使用，但實際上我們不會每個函式庫都自己刻，而會藉由使用預先寫好的函式庫，減少重造輪子的時間，專注在我們想要實作的核心功能上。

不過這也不全然是教科書的錯，比起標準函式庫，第三方函式庫的 API 相對沒那麼穩定，也有可能會在缺乏維護下逐漸凋亡，比較不適合放在教科書中。

註：使用外部函式庫要注意授權範圍，初學者往往直接忽視這一塊就任意地使用外部函式庫。

C 函式庫的檔案格式

C 函式庫包括標頭檔和二進位檔兩個部分，標頭檔存有該套件的公開界面，包括型別、函式、巨集等項目的宣告；二進位檔則是編譯後的套件實作內容。

註：Windows 中，有一部分函式庫會額外使用.def 檔案，.def 也可視為函式庫的公開界面。

⁸³ <https://conan.io/>

⁸⁴ <https://github.com/Microsoft/vcpkg>

二進位檔又依其發布方式分為靜態函式庫 (static library) 和動態函式庫 (dynamic library) 兩種；這兩種函式庫格式會影響應用程式發布的方式，靜態函式庫會直接將程式碼包進主程式中而動態函式庫會在執行時才去呼叫。

分享 C 函式庫時可以不公開 C 程式碼，只要有標頭檔和二進位檔即可使用該套件。近年來流行的開放原始碼運動算是一種軟體授權的策略或模式，對執行函式庫本身不是必備的。

至於 Unix 或類 Unix 系統上常見的 `/usr/include` 或 `/usr/lib` 等存放標頭檔或二進位檔的位置是由系統另外定義的，而非 C (或 C++) 內建的特性。

實例：二元搜尋樹

在本文中，我們使用一個小範例來說明如何撰寫 C 函式庫。由於 C 沒有規範專案如何安排，我們按照常見的 C 函式庫的專案架構來安排我們的程式碼。

在此專案中，我們撰寫以 GNU Make 來管理編譯流程，使用 Make 的好處是不用綁定特定的 IDE，只要編輯器支援 C 就可以使用此專案。

這個範例專案位於這裡⁸⁵，這是一個二元搜尋樹的練習，但我們重點會放在如何以 C 撰寫套件，不會深入探討二元樹的實作，也不會額外講解 Makefile 的語法。

以本例來說，其中一個標頭檔如下：

```
#ifndef ALGO_BSTREE_H          /* 1 */
#define ALGO_BSTREE_H           /* 2 */

#ifndef __cplusplus             /* 3 */
    #include <stdbool.h>        /* 4 */
#endif                         /* 5 */

#ifdef __cplusplus              /* 6 */
extern "C" {                   /* 7 */
#endif                         /* 8 */

typedef struct bstree_int_t bstree_int_t; /* 9 */

bstree_int_t *
algo_bstree_int_new(void);           /* 10 */
bool algo_bstree_int_is_empty(
    bstree_int_t *self);            /* 11 */
bool algo_bstree_int_find(
    bstree_int_t *self, int value); /* 12 */
int
```

(continues on next page)

⁸⁵ <https://github.com/cwchentw/bstree-c>

(continued from previous page)

```

algo_bstree_int_min(bstree_int_t *self); /* 13 */
int
algo_bstree_int_max(bstree_int_t *self); /* 14 */
bool algo_bstree_int_insert(
    bstree_int_t *self, int value); /* 15 */
bool algo_bstree_int_delete(
    bstree_int_t *self, int value); /* 16 */
void algo_bstree_int_free(void *self); /* 17 */

#ifdef __cplusplus /* 18 */
{
    /* 19 */
    /* 20 */
}

#endif /* ALGO_BSTREE_H */ /* 21 */

```

標頭檔中放的是函式庫的宣告部分，實作則會另外放在 C 程式碼中。依照 C 語言的慣例，一般會用 *.h* 做為標頭檔的副檔名。

一開始時會用 include guard 的手法，避免重覆 include 時引發錯誤。include guard 的動作位於第 1 行、第 2 行、第 21 行。

從巨集的觀點來看，include guard 本身是一段條件編譯的敘述，當第一次引入此標頭檔時，ALGO_BSTREE_H 是未定義的，所以會讀完整個標頭檔。而在第二次後再度引入此標頭檔時，ALGO_BSTREE_H 是已定義的，這時候不會再讀一次此標頭檔，藉此避開重覆宣告的問題。

另外，如果這個函式庫要從 C++ 呼叫，也要加入一些樣板程式碼。這段動作位於第 6 行至第 8 行及第 18 行至第 20 行。

從巨集的觀點來看，使用 C++ 讀入此標頭檔時，其等效宣告如下：

```

extern "C" {

/* Declarations */

}

```

其實就是一塊 `extern "C"` 區塊。

使用純 C 讀入此標頭檔時，不會有 `extern "C"` 的部分，故可以相容於 C 程式碼。

接著，就會開始寫宣告，包括引用的外部函式庫、型別、函式定義、巨集等。在這個例子中，我們不想暴露結構的內部實作，使用了 forward declaration 的小技巧，這個動作位於第 9 行。

標頭檔和程式碼間的連動

接著，來看 *bstree.c* 的程式碼 (節錄)：

```
#include <assert.h>
#include <stdbool.h>
#include <stdlib.h>
#include <stdio.h>
#include "algo/bstree.h"
#include "bstree_internal.h"
#include "bstnode.h"

/* More code ... */
```

在我們這個例子中，重要的並不是二元樹如何實作，而是觀察檔案間的連動關係。觀察標頭檔就可以知道原始碼之間的相依關係。

在 *bstree.c* 的程式碼中，會去讀取 *algo/bstree.h*，兩者間就產生了連動。實作公開界面 (標頭檔) 時，程式碼要引入想實作的界面。

細心的讀可應該可以發現，我們將 *bstree_int_t* 及 *node_int_t* 兩個型別部分的程式碼拉出來，這是因為我們將此套件的實作拆開在 *bstree.c* 及 *bstiter.c* 兩個檔案中，為了避免重覆的程式碼，我們將共同需要的部分拉出來，放在 *bstree_internal.h*、*bstnode.h* 及 *bstnode.c* 中。

再看 *bstiter.c* 的程式碼 (節錄)：

```
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include "algo/bstree.h"
#include "bstree_internal.h"
#include "bstnode.h"
#include "algo/bstiter.h"

/* More code ... */
```

從這裡可看出，這兩個模組都有用到共同的程式碼。這些程式碼會編譯成二進位檔案，而不會隨標頭檔發布出去，所以我們沒有放在 *include* 資料夾而放在 *src* 資料夾中。

如果讀者有看 *bstiter.c* 的程式碼，可發現我們在程式碼中額外塞入一個佇列，這是為了實作迭代器所需的資料結構，由於我們沒有想要公開這個資料結構，從我們的程式碼可看出在標頭檔中完全都沒有這個佇列相關的資訊。

基本上，只要知道標頭檔和實作程式碼間的關係，用模組化的概念寫 C 程式碼並不會太困難。

至於多個檔案在編譯時要如何串連，這就牽涉到編譯器指令的操作，這也就是為什麼我們

要在先前的文章中介紹 C 編譯器的指令。一開始覺得編譯器指令太難的話，不妨先用 IDE 現有的功能來完成這一部分，待熟悉這個流程後再轉用 CMake 或 GNU Make 等跨 IDE 的專案管理程式。

延伸閱讀

本文對於 C 函式庫僅是淺白的介紹，如果想要深入學習如何設計 C 函式庫，可參考 *C Interfaces and Implementations, Addison-Wesley Professional (1996)*。這本書算老書了，但 C 的核心語法相對穩定，故仍有一定參考價值。

基本輸出入 (INPUT AND OUTPUT)

前言

電腦程式除了運算 (computing) 外，還要能和外部環境溝通，才能夠傳遞資料 (data)。這些溝通的方式統稱為輸入 (input) 和輸出 (output)。

C 語言本身不具備輸出入的功能，而是透過函式庫來補完。在 C 語言中，和基本輸出入相關的功能定義在 **stdio.h** 函式庫裡。該函式庫中的函式可依輸出入目標再細分為終端機輸出入和檔案輸出入兩部分。

學習 stdio.h 函式庫

stdio.h 函式庫內的函式分為數種情境：

- 輸入 vs. 輸出
- 檔案 vs. 終端機
- 格式化 vs. 未格式化
- 一般字元 vs. 寬字元

另外還有一些其他功能的函式：

- 移動檔案指標的位置
- 錯誤處理
- 暫存檔案

學習時，不要硬背函式名稱，要想該函式所要處理的情境。

標準串流 (Standard Streams)

C 語言定義三種終端機輸出入的通道：

- 標準輸入 (standard input)
- 標準輸出 (standard output)
- 標準錯誤 (standard error)

在這三者之中，後兩者皆為輸出，故標準串流有一種輸入及兩種輸出。為什麼會有兩種輸出呢？因為輸出時，不希望正常的輸出和錯誤訊息混在一起，故刻意區分為兩種輸出。

標準輸出和標準錯誤乍看好像是同一種輸出，但可藉著重導 (redirect) 將兩種輸出分別導向不同的檔案。像是以下的指令：

```
$ nohup ./program </dev/null >out 2>err &
```

nohup(1) 可將指令轉為背景程式，直到該程式結束為止。在這個範例指令中，我們忽略標準輸入，將標準輸出導向 *out* (文字檔)，將標準錯誤導向 *err* (文字檔)。

在 C 語言中，分別以 **stdin**、**stdout**、**stderr** 等巨集宣告表示標準串流。

使用標準串流時，不需要開立檔案，直接將文字資料寫到串流即可。C 函式庫會為程式設計者自動處理標準串流的細節。

檔案輸出入

如果要將資料從檔案讀入或寫入檔案，可使用 `fopen()` 函式。該函式的界面為 `fopen(path, mode)`。path 為檔案所在的路徑，而 mode 是檔案的模式。

使用方式如下：

```
FILE *fp = fopen("path/to/file.txt", "r");
if (!fp) {
    /* `fp` is invalid here. */
    /* Handle the error. */
}

/* Use `fp` here. */

fclose(fp);
```

fp 是 file handle，代表檔案物件。此段程式以 *r* (讀取) 模式讀入檔案。

`fopen()` 有可能會失敗，像是檔案不存在、目錄不存在、權限不足等，所以要檢查 *fp* 物件是否成功建立。當 `fopen()` 函式無法建立 *fp* 物件時，會回傳 NULL。

用完 *fp* 物件時，要用 `fclose()` 將 *fp* 物件關閉。

以下是 `fopen()` 函式的模式：

- *r* : (文字) 讀取
- *w* : (文字) 寫入
- *a* : (文字) 附加
- *r+* : (文字) 讀取，可讀寫

- w+ : (文字) 寫入，可讀寫
- a+ : (文字) 附加，可讀寫

C 語言的檔案輸出入不僅用於文字檔案，也可用於二進位檔案。處理二進位檔案的模式如下：

- rb : (二進位) 讀取
- wb : (二進位) 寫入
- ab : (二進位) 附加
- r+b : (二進位) 讀取，可讀寫
- w+b : (二進位) 寫入，可讀寫
- a+b : (二進位) 附加，可讀寫

當檔案已存在時，寫入模式會摧毀原有檔案後再建立新的檔案。這算是有風險的動作。因為程式使用者可能不自覺毀掉重要檔案。在 **C11** 後，加入安全的寫入模式 x。使用方式如下：

- wx
- w+x
- wbx
- w+bx

使用 x 模式後，`fopen()` 偵測到檔案存在時會引發錯誤而不會覆寫檔案。這算是防呆的措施。

格式化輸出的格式

以下是一個使用格式化輸出的簡短範例：

```
#include <stdio.h>

int main(void)
{
    char *greeting = "Hello";
    char *object = "World";

    printf("%s %s\n", greeting, object);

    return 0;
}
```

在這個範例中，"`%s %s\n`" 的部分即是輸出格式。該參數本身是 C 字串，但有特定的格式，做為輸出的模板。

C 語言的模板比較簡單，不具有程式語言的功能，只能指定某些轉換字元，像是 `%s`。以下是常見的轉換字元：

- `%c`：輸出單一字元
- `%s`：輸出 C 字串
- `%d` 或 `%i`：輸出帶號整數
- `%u`：輸出無號整數
- `%f` 或 `%F`：輸出浮點數
- `%g` 或 `%G`：輸出浮點數或科學記號
- `%p`：輸出指標的虛擬位址
- `%%`：輸出 `%` 本身

除了轉換字元外，所有的字元都會原封不動地輸出。

格式化輸出函式

格式化輸出在輸出時可指定字串格式，這系列函式以 `printf` 來命名：

- `printf(format, ...)`：輸出到標準輸出
- `fprintf(stream, format, ...)`：輸出到指定的串流 (stream)
- `sprintf(buffer, format, ...)`：輸出到字元陣列
- `snprintf(buffer, size, format, ...)` (**C99**)：輸出到字元陣列，最多 n 個字元

`...` 的部分代表要放入模板內的參數，其型態和數量要和模板保持一致。

`stream` 的部分可為標準串流 (標準輸出或標準錯誤) 或檔案物件 (寫入模式)。

`buffer` 為字元陣列。`C` 不會主動檢查 `buffer` 的大小是否足夠，程式設計者得自己檢查。

另外還有在 **C11** 後所引入的安全版本：

- `printf_s(format, ...)`
- `fprintf_s(stream, format, ...)`
- `sprintf_s(buffer, size, format, ...)`
- `snprintf_s(buffer, size, format, ...)`

`sprintf()` 函式在輸入的字串大於 `buffer` 時的行為是未定義的，這項不良的特性容易變成安全漏洞。呼叫該函式時要預先配置好夠大的 `buffer` 或是改用較安全的 `snprintf()` 函式。

格式化輸入函式

格式化輸入在輸入時會依據指定的字串格式來解析，這系列函式以 `scanf` 來命名：

- `scanf(format, ...)`：從標準輸入輸入
- `fscanf(stream, format, ...)`：從指定的串流輸入
- `sscanf(buffer, format, ...)`：從字串陣列輸入

同樣地，`...` 的部分代表要放入模板內的參數，其型態和數量要和模板保持一致。

另外還有在 **C11** 後所引入的安全版本：

- `scanf_s(format, ...)`
- `fscanf_s(stream, format, ...)`
- `sscanf_s(buffer, format, ...)`

格式化輸入函式並不若想像中的實用。與其把格式寫死在參數模板中，不如先用未格式化輸入函式接收文字資料，然後再以解析器 (parser) 去拆解字串。

常見的文字檔案格式，通常可以找到別的程式設計師寫好的函式庫，不太需要自行實作。

未格式化輸出函式

這些函式會輸出單一字元或字串到特定串流，但無法指定輸出格式。這系列函式以 `put` 來命名：

- `putchar(ch)`：輸出單一字元到標準輸出，等同於 `putc(ch, stdout)`
- `fputc(ch, stream)` 或 `putc(ch, stream)`：輸出單一字元到指定的串流。後者可能以巨集實作，故不適用於表達式
- `fputs(str, stream)`：輸出字串到指定的串流

未格式化輸入函式

這些函式從特定串流輸入單一字元或特定長度字串，但無法指定輸入格式。這系列函式以 `get` 來命名：

- `getchar(void)`：從標準輸入讀入單一字元，等同於 `getc(stdin)`
- `fgetc(stream)` 或 `getc(stream)`：從指定的串流讀入單一字元。後者可能以巨集實作，故不適用於表達式
- `fgets(str, size, stream)`：從指定的串流讀入指定長度的字元，碰到換行字元會停下來

`fgets()` 每次會讀入一行。其使用方式如下：

```
#define SIZE 4096;
char buffer[SIZE];

while(fgets(buffer, SIZE, fp)) {
    /* Process `buffer` here. */
}
```

在此段程式中，我們假定使用者輸入的資料的行寬不超過 4096 個字元。對於一般文字檔案來說，這樣的假設是可接受的。但有時候我們仍會碰到較長的文字檔案，這樣的寫法就行不通。

如果想要妥善地處理較長的文字資料，得用動態配置文字緩衝 (text buffer) 的方式。參考以下程式碼：

```
#include <stdio.h>                      /* 1 */
#include <stdlib.h>                       /* 2 */
#include <string.h>                        /* 3 */

#ifndef END_OF_LINE                         /* 4 */
#if __WIN32                                /* 5 */
#define END_OF_LINE  "\r\n"                  /* 6 */
#elif __unix__ || __APPLE__                  /* 7 */
#define END_OF_LINE  "\n"                   /* 8 */
#else                                       /* 9 */
    #error "Unsupported"
#endif                                         /* 11 */
#endif                                         /* 12 */

#ifndef PUTERR                               /* 13 */
#define PUTERR(format, ...) { \
    fprintf(stderr, \
        format "%s", ##__VA_ARGS__, \

```

(continues on next page)

(continued from previous page)

```

        END_OF_LINE); \
    }
#endif                                         /* 14 */
                                         /* 15 */

char * stream_read_all(FILE *fp)             /* 16 */
{
    char *buffer = NULL;                      /* 18 */
    char *more_buffer = NULL;                 /* 19 */
    char *line = NULL;                        /* 20 */
    char *more_line = NULL;                   /* 21 */

    /* Allocate memory for buffer. */          /* 22 */
    /* Arbitrary size. */
    size_t buf_sz = 1024;                     /* 23 */
    buffer = (char *) \
        malloc(buf_sz * sizeof(char));        /* 24 */
    if (!buffer) {                           /* 25 */
        PUTERR(
            "Failed to allocate "
            "a C string");                  /* 26 */
        PUTERR(
            "Check available memory");      /* 27 */
        return NULL;                         /* 28 */
    }                                         /* 29 */

    buffer[0] = '\0';                         /* 30 */

    /* Allocate memory for line. */           /* 31 */
    /* Arbitrary size. */
    size_t line_sz = 150;                     /* 32 */
    line = \
        (char *) malloc(
            line_sz * sizeof(char));        /* 33 */
    if (!line) {                           /* 34 */
        PUTERR(
            "Failed to allocate "
            "a C string");                  /* 35 */
        PUTERR(
            "Check available memory");     /* 36 */
        goto ERROR_CIO_READ_ALL;         /* 37 */
    }                                         /* 38 */

    line[0] = '\0';                          /* 39 */

/* Read file stream

```

(continues on next page)

(continued from previous page)

```

line by line. */          /* 40 */
size_t len = 0;           /* 41 */
while (fgets(line, line_sz, fp)) { /* 42 */
    size_t sz = strlen(line); /* 43 */
    if (line_sz == sz) { /* 44 */
        /* Reallocate the line. */ /* 45 */
        if ('\n' != line[sz-1]) { /* 46 */
            /* Reallocate the buffer. */ /* 47 */
            if (len + sz >= buf_sz) { /* 48 */
                while (len + sz >= buf_sz) /* 49 */
                    buf_sz <= 1; /* 50 */

                more_buffer = \
                    realloc(buffer, buf_sz); /* 51 */
                if (!more_buffer) { /* 52 */
                    goto ERROR_CIO_READ_ALL; /* 53 */
                } /* 54 */

                buffer = more_buffer; /* 55 */
            } /* 56 */
        }
    }
    strcpy(buffer+len, line); /* 57 */
    len += sz; /* 58 */
    buffer[len] = '\0'; /* 59 */
    line_sz <= 1; /* 60 */
}

more_line = \
    realloc(line, line_sz); /* 61 */
if (!more_line) { /* 62 */
    goto ERROR_CIO_READ_ALL; /* 63 */
} /* 64 */

line = more_line; /* 65 */
}
else {
    goto LOAD_LINE; /* 66 */
}
else {
LOAD_LINE:
    /* Reallocate buffer. */ /* 73 */
    if (len + sz >= buf_sz) { /* 74 */
        while (len + sz >= buf_sz) /* 75 */
            buf_sz <= 1;
    }
}

```

(continues on next page)

(continued from previous page)

```

buf_sz <= 1;           /* 76 */

more_buffer = \
    realloc(buffer, buf_sz); /* 77 */
if (!more_buffer) {      /* 78 */
    goto ERROR_CIO_READ_ALL; /* 79 */
}

buffer = more_buffer;   /* 81 */
}                       /* 82 */

/* Copy line to buffer. */ /* 83 */
strcpy(buffer + len, line); /* 84 */

len += sz;              /* 85 */
buffer[len] = '\0';     /* 86 */
}                       /* 87 */
}                       /* 88 */

free(line);             /* 89 */

return buffer;          /* 90 */

ERROR_CIO_READ_ALL: /* 91 */
if (line)               /* 92 */
    free(line);          /* 93 */

if (buffer)             /* 94 */
    free(buffer);         /* 95 */

return NULL;            /* 96 */
}                       /* 97 */

```

在此範例程式中，我們以兩個文字緩衝 buffer 和 line 來儲存文字資料。buffer 會存入整個 fp 物件的文字資料，line 則僅用於暫存單行資料。

此程式在第 23 行至第 30 行間建立 buffer 物件。其大小 1024 是任意的 (arbitrary) 數值，並沒有什麼意義。同理，在第 32 行至第 39 行間建立 line 物件。其大小 150 也是任意的值。由於 buffer 和 line 的大小可以動態調整，不用太糾結起始值的大小。

實際讀取 fp 物件的迴圈位在第 42 行至第 88 行。在此處，我們用 fgets() 函式逐行讀入文字資料。讀入資料時，要考慮單行資料的寬度是否有超出 line 的寬度。

當單行資料的寬度大於等於 line 的寬度，要視情形擴展 line 和 buffer。這段程式碼位於第 44 行到第 70 行。反之，雖然不用擴展 line，但仍可能需要擴展 buffer。這段程式碼位於第 71 行至第 87 行。

判斷是否要擴展 `line` 的依據是尾端字元是否為 '\n' (newline)，當尾端不為 newline 時，代表文字檔案的寬度較大，這時會將 `line` 和 `buffer` 皆擴展。判斷句位於第 44 行。

由於每次讀取資料時，`line` 內的資料會被清空覆寫。所以要把讀入的資料存入 `buffer` 中。存入的指令位於第 57 行及第 84 行。

`line` 只是暫存用文字緩衝。用完後會將 `line` 所占用的記憶體清除。這個指令位於第 89 行。相對來說，`buffer` 會做為回傳值，故不需要清除。回傳的指令位於第 90 行。

除了常規的指令外，也要考慮錯誤發生時提早中止程式的程式。這些指令位於第 91 行至第 96 行。

這個函式是 `cwchentw/clibs`⁸⁶ 函式庫的一部分。如果讀者需要在自己的程式中使用此函式，可以直接從 `clibs` 函式庫拷貝這段程式碼。

⁸⁶ <https://github.com/cwchentw/clibs>

前言

當讀者讀到這裡，代表這本書即將結束。但對程式人來說，學完程式語言的核心功能不代表學習的結束，反而是下一個階段的開始。當我們能夠用程式語言解決問題時，才算是脫離新手的階段，達到學以致用的狀態。

在本文中，我們介紹一些可能的學習方向，讓讀者可以繼續努力，不會因學完核心功能就停滯不前。

標準函式庫 (Standard Library)

標準函式庫會隨著程式語言的編譯器或直譯器一起發佈，可視為程式語言預設的功能。在學完程式語言的核心功能後，應該花點時間看一看標準函式庫提供什麼功能。

由於標準函式庫會隨著程式語言本身共生共榮，當標準函式庫能滿足我們的功能時，應優先考慮使用標準函式庫。反之，在標準函式庫無法滿足我們時，才會尋求第三方方案。

以 C 語言來說，目前標準函式庫的專書很少，比較新的實體書籍像是 *C in a Nutshell: the Definitive Reference, 2nd edition, O'Reilly (2015)* (有中譯本)。可能的原因是許多網站會免費提供這方面的資訊，像是 [cppreference.com⁸⁷](https://en.cppreference.com) 或是 [cplusplus.com⁸⁸](http://www.cplusplus.com) 等。

學習標準函式庫時，並不是要把函式名稱背起來，這些都是隨手可查的資訊。而是要先知道標準函式庫有什麼功能，在需要時會想得到這些功能，然後再去查標準函式庫的文件。

資料結構 (Data Structure)

資料結構是有效率地存取資料的方式。對於許多高階語言來說，基本資料結構是標準函式庫中立即可用的功能。但對 C 語言來說，除了陣列 (array) 以外，我們仍然得自行實作資料結構，或是使用第三方函式庫。此外，高階語言的標準函式庫不會提供所有的資料結構，我們仍然要有實作資料結構的能力。

由於大專院校有資料結構的課程，市面上可以看到許多資料結構的書籍。這些書籍會把基礎的資料結構重新實作一輪，給人重造輪子的感覺。但學資料結構的重點在於理解其原理，以及體驗實作的過程。我們自己做出來的資料結構，不一定會在實際上線的程式中用到。

⁸⁷ <https://en.cppreference.com/>

⁸⁸ <http://www.cplusplus.com/>

演算法 (Algorithm)

相對於資料結構是處理資料的方式，演算法則是正確且有效率地實作程式的方法。由於有些演算法要搭配特定的資料結構，我們通常會在學完資料結構後再學習演算法。

大專院校學習的演算法算是基礎演算法，這些演算法很有可能已由某個函式庫實做出來，除了在課堂以外，我們不一定要自己重新撰寫一次。這時候的學習重點除了了解各個演算法的原理之外，也要學習估算程式的效率，以及利用現有的基礎演算法撰寫更複雜的程式。

資料結構及演算法算是比較偏理論面的內容，行有餘力的話，可以試著實作一些小型應用程式，以磨練自己的能力。

命令列工具 (Console Application)

這裡的命令列工具不是我們在 C 語言教材所看到的範例程式，而是具有實用的功能，執行特定任務的命令列工具。Windows 使用者可能較少接觸命令列工具，相對來說，許多類 Unix 系統上的經典命令列工具就是以 C 語言實作。

雖然命令列工具沒有漂亮的使用者界面，由於命令列工具的輸出入相對單純，更容易專注在功能面的實作。我們可能無法馬上寫出像是 Git⁸⁹ 般經典的命令列工具，但我們可以先試著用命令列工具解決小型問題，像是結合 cURL 和匯率查詢 API 的匯率查詢工具，或是使用 SQLite 儲存 TODO 的簡易清單軟體。

圖形介面程式 (GUI Application)

如果已經熟悉基本的 C 程式，可以考慮撰寫圖形介面程式。比起命令列工具，圖形介面程式更有機會寫出適合一般使用者的應用程式。一開始可以先寫一些小型應用程式，像是採地雷之類的小遊戲，或是小時鐘等小型程式，以免因開發時程過長產生挫折感。

C 語言沒有原生的圖形介面函式庫，所以要用第三方函式庫來寫。如果只需在 Windows 上發佈可考慮用 Windows API，如果需要寫跨平台程式可考慮 GTK+⁹⁰。如果要寫電腦遊戲，除了比較簡單的遊戲可用圖形介面函式庫來寫外，通常都會搭配遊戲引擎來寫，像是 Allegro⁹¹。

⁸⁹ <https://git-scm.com/>

⁹⁰ <https://www.gtk.org/>

⁹¹ <https://liballeg.org/>

網頁應用程式 (Web Application)

網頁除了做為靜態文件外，也可以透過網頁技術製作網頁應用程式。從早期撰寫 CGI 程式做為網頁後端程式，到近年來用 C 或 C++ 等語言在網頁前端撰寫 WebAssembly 程式。C 雖然不是網頁程式最常見的選項，仍然是可用的語言。

使用 C 語言撰寫網頁程式的確比較困難，因為 C 語言在字串處理的能力比較弱。因此，我們通常會使用更易於使用的高階語言來寫網頁程式，只有在寫嵌入式軟體時，使用 C 語言寫網頁程式，做為整個系統的一部分。

嵌入式裝置 (Embedded System) 和物聯網 (Internet of Thing)

在個人電腦或伺服器上跑程式時，由於運算資源相對充裕，無法充分體會到使用 C 語言的好處。但在嵌入式裝置或物聯網上跑軟體時，由於運算資源受限，像 Java 或 C# 這類肥大的運行環境便顯得過於奢侈，這時候 C 語言便是首選。一些例子像是 Arduino⁹²。

在寫嵌入式軟體時，使用的 C 編譯器可能和個人電腦上的相異，這時候 C 語言的特性也會和先前所學的相異。像是我們之前提到的 embedded C，就是針對嵌入式裝置所設置的 C 標準。

編譯器 (Compiler) 或直譯器 (Interpreter)

除了寫應用程式外，用 C 語言實作高階語言也是一個學習的方向。實際上，有很多高階語言的官方實作品是以 C 來撰寫，像是 Perl、Python、Ruby、PHP、Lua 等。這些語言會公開其 C API，保留以 C 語言撰寫延伸模組的靈活性。

對於一般程式人來說，要寫到世界知名的通用型語言難度過高，但我們可以利用學習編譯器或直譯器時所學到的知識撰寫實用的工具，像是設定檔處理軟體、程式碼重排軟體、程式碼檢查軟體、小型領域專用語言等。

⁹² <https://www.arduino.cc/>

科學運算 (Scientific Computing)

由於電腦程式可快速運算出結果，科學家大量用來解決科學上的議題，也就是所謂的科學運算。很多書籍會使用 Python 或 R 等語言，但其實也可以用 C 或 C++ 等語言來進行運算。有時候以直接以 C 或 C++ 寫程式來運算，有時候則是用 C 或 C++ 寫其他高階語言的延伸模組，再用這些高階語言間接呼叫這些程式來做運算。

對於科學運算來說，除了撰寫程式的技能外，領域知識也相當重要。對於電腦程式本身來說，那些資料就是一些數字而已，資料所帶來的意義，還是要由科學家來判讀。

學習另一個程式語言

有些程式人會在學完 C 語言後轉而學習另一個程式語言。這件事本身無可厚非，因為主流語言能夠存活，代表該語言在某項應用上有其市場。但是，如果學了十門程式語言的語法，卻沒有試著用這些語言寫應用程式或是解決某項實務問題，還不如專注在一門語言上，並使用該語言撰寫具有實用性的程式。

比起其他的語言，C 語言有特別的優勢。如同前文所述，許多主流語言有公開的 C API，用來寫該語言的 binding。我們可以用 C、C++、Rust 等語言撰寫核心功能，並為該功能寫 C 界面，再和高階語言提供的 C API 串接成延伸模組。由此可知，即使我們平常不寫 C，還是可以用到 C 相關的知識。

附錄：在 VISUAL STUDIO 2022 中建立和編譯 C 專案

前言

Visual Studio 預設使用 Visual C++ 做為其 C 和 C++ 編譯器，而現在 Visual Studio 也支援 Clang 了。由於新版 Visual C++ 加入了現代 C 語言 (C11、C17) 的支援，故兩者皆可用來練習 C 語言。

本文說明在 Visual Studio 2022 中建立和編譯 C 專案的方式。

安裝必要的工作負載和套件

在 Visual Studio 安裝程式中選擇工作負載「使用 C++ 的桌面開發」，並選擇「適用於 Windows 的 C++ Clang 工具」：



由於 Clang 不是預設的項目，需要自己手動勾選。

建立主控台專案

開啟 Visual Studio 2022，進入主畫面：



選擇「主控台應用程式」，即為命令列程式：



這時候不會有 C 專案，因為 Visual Studio 是設計給 C++ 用的。我們可以事後修改，不用擔心。

設置專案名稱和路徑：



這樣即可建立專案。

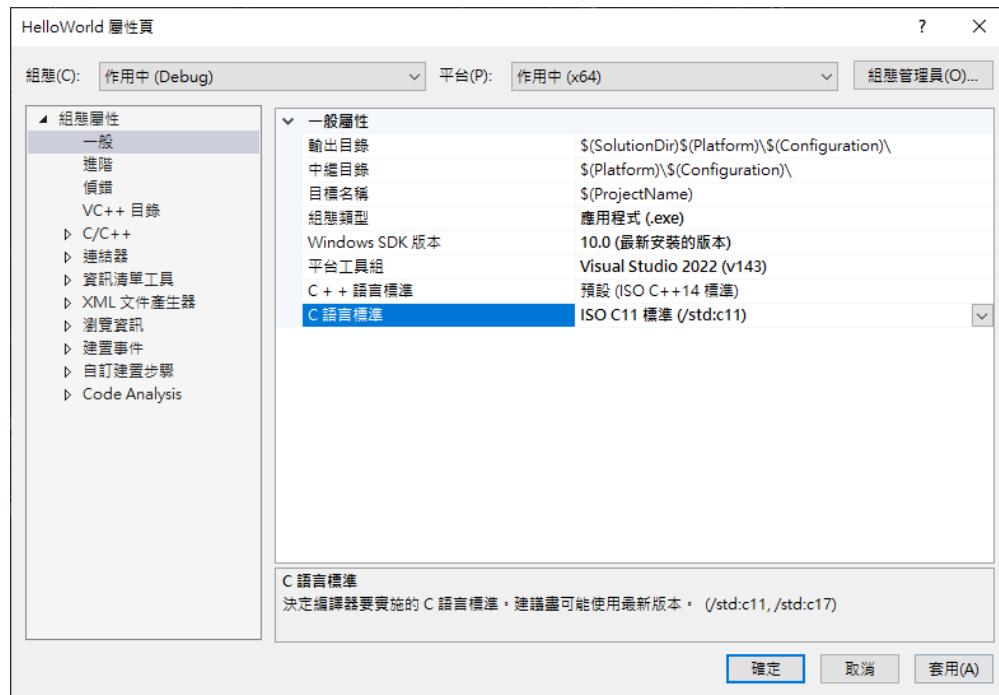
設置專案屬性

在「方案總管」中，對專案按右鍵，選擇「屬性」：



設置專案屬性

在「組態屬性」→「一般」選單中，將「C 語言標準」改為「ISO C11 標準 (/std:c11)」：



經過這樣設置後，就是用現代 C 語言的特性來撰寫 C 專案。

使用現代 C 語言的特性撰寫範例程式

這時候，可以用一個具有現代 C 語言特性的 C 程式來測試設置好的組態。將檔名按右鍵，重新命名為 *main.c*，加入以下內容：

```
#include <math.h>
#include <stdio.h>

#define LOG(x) _Generic((x), \
    long double: log10l, \
    float: log10f, \
    default: log10)(x)

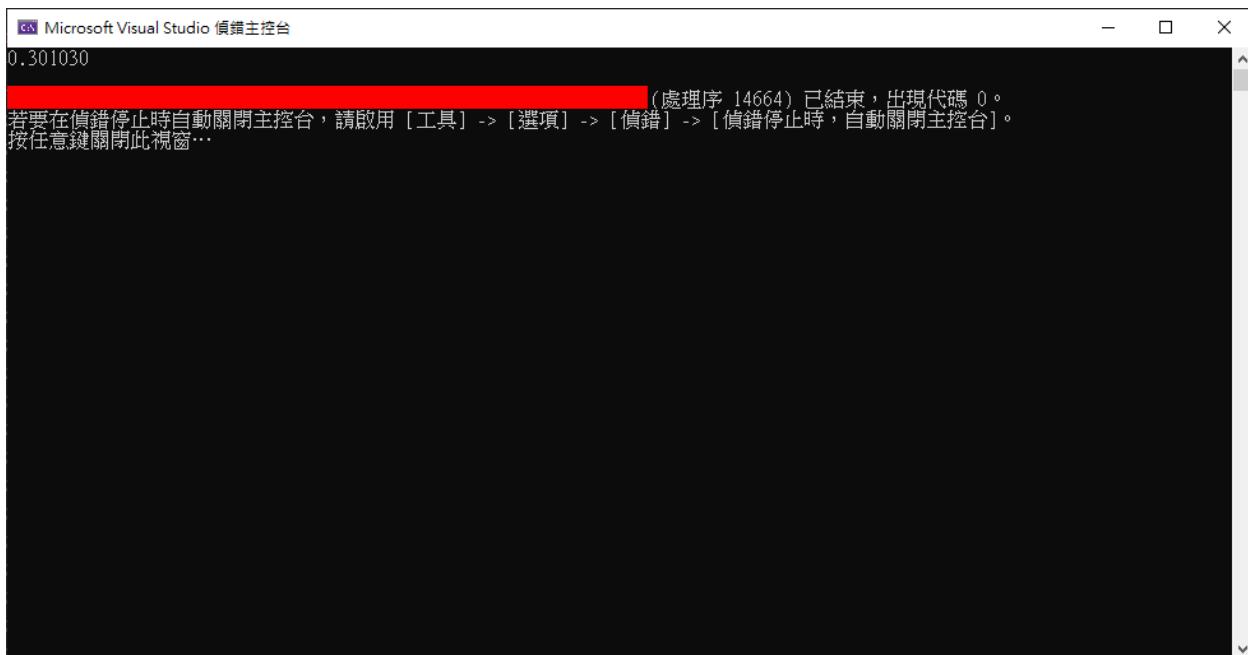
int main(void)
{
    printf("%Lf\n", LOG(2.0L));
    return 0;
}
```

由於 *_Generic* 巨集敘述是舊版的 Visual C++ 不支援的特性，剛好符合我們試用 Visual Studio 的需求。

按下上方的「本機 Windows 偵錯工具」：



跑出程式的輸出：



參考本文的流程，讀者應該可以順利地用 Visual Studio 學習現代 C 語言。

前言

C 巨集沒有函式的額外開銷，也不受到資料型態的限制。對於簡短的 C 程式碼來說，可以用巨集取代函式。使用巨集取代函式會使用程式體積變大，因為使用巨集等同於將程式碼複製貼上。但對簡短的巨集來說，這樣的影響微乎其微。

本文簡要地介紹幾個實用的 C 巨集，讀者可以直接在自己的專案中使用這些程式碼。也可以參考這些巨集，自己試著寫巨集。

附帶換行符號的終端機輸出

以下巨集約略等同於 `printf` 函式，但會自動附上換行符號，就可以少打一點程式碼：

```
#define PPUTS(format, ...) \
    fprintf(stdout, format "\n", ##__VA_ARGS__);
```

承上，將輸出改導向標準錯誤，即可寫出以下巨集：

```
#define PUTERR(format, ...) \
    fprintf(stderr, format "\n", ##__VA_ARGS__);
```

不附帶換行符號的終端機輸出

其實這個巨集等同於 `printf` 函式，設計這個巨集只是要和上一節的巨集成對：

```
#define PRINT(format, ...) \
    fprintf(stdout, format, ##__VA_ARGS__);
```

承上，將輸出改導向標準錯誤，即可寫出以下巨集：

```
#define PERROR(format, ...) \
    fprintf(stderr, format, ##__VA_ARGS__);
```

用於除錯的終端機輸出

使用 `printf` 除錯時，除了印出錯誤訊息外，追蹤錯誤發生的檔案和行數也很重要。所以，以下巨集利用 C 內建的巨集變數輸出檔案和行號：

```
#ifdef DEBUG
#define DEBUG_INFO(format, ...) \
    fprintf(stderr, "(%s:%d) " format "\n", \
            __FILE__, __LINE__, ##__VA_ARGS__);
#else
#define DEBUG_INFO(...)
#endif
```

這裡利用巨集變數 `DEBUG` 來開關此巨集。程式要上線時，關掉此變數，就不會輸出附錯訊息。

指定範圍的隨機整數

以下巨集可取得範圍在 `min` 至 `max` 間的隨機整數：

```
#define RANDINT(min, max) \
    ((min) + random() % ((max) - (min) + 1))
```

當然，巨集使用者要自行設置亂數種子。這個巨集只是用來簡化程式碼。

基本的數學運算

以下巨集可取得兩數中較大值，且不受資料型態限制：

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

同上，以下巨集可取得兩數中較小值，且不受資料型態限制：

```
#define MIN(a, b) ((a) < (b) ? (a) : (b))
```

以下巨集可取得絕對值：

```
#define ABS(n) ((n) > 0.0 ? (n) : -(n))
```

利用前述 `ABS` 巨集可檢查兩浮點數是否相等：

```
#define IS_EQUAL(a, b, epsilon) (ABS((a) - (b)) <= (epsilon))
```

這個巨集僅適用於一般的浮點數運動。對於超小浮點數需用其他演算法。這超出本書的範圍，故不討論。

以下巨集可比較兩數間的大小關係，且不受資料型態限制：

```
#define COMPARE(a, b) \
((a) > (b) ? 1 : \
(a) == (b) ? 0 : -1)
```

以下巨集可檢查特定整數是否為偶數：

```
#define IS_EVEN(n) (!((n) & 1))
```

這裡用到二進位運算的技巧。當某數為偶數時，最後一位數必為 0，和 1 交集 (bitwise and) 的結果為 0。再以 ! 進行反向布林運算即可確認該數為偶數。

同理，以下巨集可檢查特定整數是否為奇數：

```
#define IS_ODD(n) ((n) & 1)
```

以下巨集可以在不使用額外變數的前提下交換兩個整數：

```
#define SWAP(a, b) (((a) ^= (b)), ((b) ^= (a)), ((a) ^= (b)))
```

複數

雖然新版的 Visual C++ (MSVC) 支援 C11 及 C17，但為了相容現存程式碼，仍然不支援 C99 的複數型態。為了在不同 C 編譯器使用複數型態，可使用以下巨集：

```
#include <complex.h>

#if __STDC_VERSION__ >= 199901L
#define COMPLEX_T complex
#define COMPLEX_NEW(real, imag) ((real) + (imag) * I)
#else
#define _MSC_VER
#define COMPLEX_T _Dcomplex
#define COMPLEX_NEW(real, imag) ((_Dcomplex){(real), (imag)})
#endif
#error "Complex numbers are not supported"
#endif /* MSVC */
#endif /* C99 */
```

無限大、無限小、非數字

本節說明如何用巨集表示無限大、無限小、非數字等無法用實字寫出來的數字。

以下巨集可取得無限大：

```
#ifdef _MSC_VER
    #include <math.h>
    #define INF (-logf(0.0))
#else
    #define INF (1.0 / 0.0)
#endif
```

在 GCC 和 Clang 上，可以用 $1.0 / 0.0$ 來取得無限大。但在 Visual C++ 中，該程式碼會引發錯誤，故改用另一種方式來取得無限大。無限小用 $-\text{INF}$ 即可取得。

以下巨集可檢查數字是否為無限大：

```
#ifdef _MSC_VER
    #include <math.h>
    #define IS_INF(n) (!_finite(n) && (n) > 0)
#else
    #define IS_INF(n) ((n) > FLT_MAX)
#endif
```

在 GCC 和 Clang，比 FLT_MAX 大的數字即為無限大。按照數學理論，這樣的式子是說不通的。這個巨集利用電腦的特性來取巧。在 Visual C++ 中，該程式碼會引發錯誤，故改用 MSVC 特有的函式 $_finite$ 來判定是否為有限數，再取相反的結果。

同理，以下巨集可檢查數字是否為無限小：

```
#ifdef _MSC_VER
    #include <math.h>
    #define IS_NEG_INF(n) (!_finite(n) && (n) < 0)
#else
    #define IS_NEG_INF(n) (- (n) > FLT_MAX)
#endif
```

以下巨集可取得非數字：

```
#ifdef _MSC_VER
    #include <math.h>
    #define NaN (logf(0.0) - logf(0.0))
#else
    #define NaN (0.0 / 0.0)
#endif
```

在 GCC 和 Clang 上，可用 `0.0 / 0.0` 取得非數字。但在 Visual C++ 中，該程式碼會引發錯誤，故改用另一種方式來取得非數字。

以下巨集可檢查數字是否為非數字：

```
#define IS_NaN(n) \
  ((n) > (n) ? 0 : \
   (n) == (n) ? 0 : \
   (n) < (n) ? 0 : 1)
```

這裡利用非數字在電腦上的特性而得。