C 語言應用程式設計 (試讀本)

Release 1.0.0

Michelle Chen

Jul 19, 2021

目錄

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)

- 1.0.0
  - 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：

```c
#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：

```c
/* 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
```

如何建立 C 專案 (PROJECT)

前言

平常練習 C 語言時，我們可能只用單一或少數 C 原始碼檔案，只用簡單的指令或 IDE 按鈕來編譯 C 原始碼。但我們若想用 C 寫應用程式或函式庫，應該要以專案的形式管理 C 原始碼。本文介紹建立 C 專案的方式。

什麼是 C 專案？

C 語言本身沒有軟體專案的概念，C 專案是透過開發工具後設取得的。如同其他語言的專案，C 專案有特定的目標。像是一個處理 JSON 格式檔案的函式庫 json-c[1] 就是一個 C 專案。

可用的 C 專案有兩項基本要件：

- 將 C 原始碼集中在特定目錄或其子目錄中
- 使用自動編譯軟體進行自動化編譯

至於其他的項目，像是說明文件、範例程式碼、版本控制等，都是輔助性的東西。對於使用 C 專案本身來說，不是必備的。

C 專案的種類

根據專案的產出，可將 C 專案分為應用程式 (application) 和函式庫 (library) 兩大類。應用程式為可執行的電腦程式 (program)。函式庫則是預寫的函式、巨集等，給外部程式使用。

根據使用者介面又可再將應用程式分為命令列工具 (console application)、圖形界面程式 (GUI application)、網頁程式 (web application)、網頁前端程式 (web front-end application)、應用程式伺服器 (application server) 等。

註：C 和 C++ 程式碼可編譯成 WebAssembly 後，在網頁前端執行。

函式庫則可分為靜態函式庫 (static library) 和動態函式庫 (dynamic library 或 shared library) 兩種。兩者使用的方式相異。

靜態函式庫會在編譯時將函式庫的程式碼「貼」到主程式中，所以主程式發佈時不需要攜帶靜態函式庫。但主程式的體積會變大，而且更新時要直接替換主程式的執行檔。

[1] https://github.com/json-c/json-c

相對來說，動態函式庫會在執行期才和主程式進行連結。主程式發佈時需要另外攜帶動態函式庫。但使用動態函式庫的應用程式的體積較小，而且更新時可以只更新動態函式庫，不用替換主程式。

C 專案不一定僅限單一產出形式。我們可以把專案的核心功能包成函式庫，再自己寫不同界面的應用程式呼叫此函式庫，就可以讓多種形式的應用程式共享相同的核心功能。

C 語言原始碼

C 語言的原始碼分為標頭檔 (header) 和程式碼 (source) 兩部分。標頭檔為 C 程式的宣告，程式碼則是 C 程式的實作。

我們偶爾可以在網路上看到有些開發者分享號稱 header-only 的 C 專案，但卻把實作寫在標頭檔內。雖然這是合法的 (valid)，但不建議把實作寫在標頭檔內。因為這樣做有可能會造成重覆引入的問題。

按照 C 語言的慣例，不論在那個系統上，標頭檔的副檔名皆為 *.h* ，而程式碼的副檔名則為 *.c* 。C 語言不會像 Java 般刻意限制檔案名稱，所以檔案名稱只要簡明易懂即可。

C 專案的架構

C 語言沒有內定的專案架構，可參考一些現有的 C 專案架構來建立自己的 C 專案。

如果檔案數量很少，直接用扁平式的專案架構也無妨。這時候，所有的標頭檔、程式碼、自動編譯設定檔等，都會放置在專案的根目錄下。

以下用 `tree(1)` 展示一個假想的扁平專案：

```
$ cd path/to/project
$ tree
.
├── Makefile
├── list.c
├── list.h
└── test_list.c
```

如果檔案比較多，就會建議用巢狀式的專案架構。這時候、標頭檔、程式碼、測試程式、範例程式等，會各自放在不同的子目錄。以下是常見的子目錄名稱：

- *src* ：存放原始碼和內部標頭檔
- *include* ：存放公開標頭檔
- *dist* ：存放專案產出
- *bin* ：當專案產出為應用程式時，可用 *bin* 取代 *dist*

- *test* ：存放測試程式
- *example* ：存放範例程式
- *doc* ：存放說明文件

我們同樣用 `tree(1)` 來展示一個假想的巢狀專案：

```
$ cd path/to/project
$ tree -L 1
.
├── Makefile
├── dist
├── include
├── src
└── test
```

大型專案則會用子目錄進一步細分 C 原始碼，像是 GNOME 的 glib[2] 和 gtk[3] 都採用這樣的方式來設置專案架構。

C 專案的輔助文件

除了標頭檔和程式碼外，C 專案還會有一些輔助文件。雖然這些文件不是原始碼的一部分，但也對了解專案有所幫助。以下是常見的輔助文件：

- *README* 或 *README.md* ：讀我檔案
- *INSTALL* ：安裝說明
- *LICENSE* ：專案的授權文件
- *HISTORY* ：專案的版本釋出記錄
- *.gitignore* ：讓 Git 忽略某些形態的目錄或檔案
- *.travis.yml* ：Travis CI 的設定檔
- *Makefile* ：`make(1)` 或 `gmake(1)` 預設的設定檔
- *configure* ：Autotools 的命令稿
- *CMakeLists.txt* ：`cmake(1)` 預設的設定檔

這些文件會放在專案的根目錄。

README 是專案的介紹文件。目的是讓接觸此專案的程式設計者對專案有初步的認識。由於 *README* 對於專案有著推廣的作用，應該儘量用簡明的文字來撰寫。

[2] https://github.com/GNOME/glib
[3] https://github.com/GNOME/gtk

許多專案會刻意使用 *README.md* 為讀我檔案的名稱，並用 Markdown 格式來書寫讀我檔案。因為 GitHub、GitLab 等程式碼專案管理網站會自動解析該文件，根據該文件生成漂亮的說明文件。

INSTALL 是專案的安裝說明文件。該文件會記載專案的相依性、編譯工具、編譯指令等。理想的 *INSTALL* 文件應該讓專案使用者可以只閱讀該文件就足以順利編譯和安裝專案程式。

LICENSE 是專案的授權文件。由於 GitHub、GitLab 等網站會解析該文件，並在頁面上自動顯示專案的授權，授權文件最好保持此名稱。

現在的程式碼專案大多會用版本控制系統 (version control system) 管理。Git 是近年來最知名的版本控制軟體。*.gitignore* 用來控制不應存入專案內的檔案。C.gitignore[4] 是 GitHub 給 C 專案的忽略清單，若有使用 Git 管理專案，可以參考一下。

CI (continuous integration) 是自動化編譯及測試的一環。簡單地說，就是在雲端自動編譯及測試程式碼專案後，確認程式碼專案可正常運作，CI 服務會以電子郵件等方式通知專案擁有者。常見的 CI 有 Travis CI 等。

C 專案的標頭檔和原始碼只是靜態檔案，要自動化編譯得使用外部工具來管理。常見的工具有 Make、Autotools、CMake 等。詳見下一節的說明。

自動編譯軟體 (Build Automation System)

除了 C 原始碼外，C 專案另一個重要的部分即為自動編譯軟體的設定檔。透過這類軟體，可以省下許多指令輸入，快速地編譯 C 專案。本節介紹常見的自動編譯軟體。

Make

Make 是自動化編譯軟體的濫觴，在 Unix 及類 Unix 系統上相當流行，在 Windows 上也可見 Make 的移植品。由於 Make 是 POSIX 標準的一部分 (參考這裡[5])，大部分類 Unix 系統上都有某種 Make 的實作品。

使用 Make 的好處在於不綁定任何 IDE，在命令列環境下也可以工作。但使用 Make 需要自行撰寫設定檔 (通常稱為 *Makefile*)，上手比較困難。

要注意 Make 不是單一的軟體，而有數種實作品。不同 Make 實作品的 *Makefile* 語法會有一些差異。常見的 Make 實作品有 POSIX Make 和 GNU Make 等。目前主流的系統都有 GNU Make 可用，所以不用刻意守在 POSIX Make 的語法。

當專案使用者所在的系統和專案的設定檔相異時，專案使用者得手動編輯 *Makefile* 。中大型專案的 *Makefile* 往往也相當複雜，要手動修改相當不方便。於是，出現 Autotools 這種可偵測系統狀態、自動生成 *Makefile* 的軟體。詳見下一小節。

[4] https://github.com/github/gitignore/blob/master/C.gitignore
[5] https://pubs.opengroup.org/onlinepubs/009695399/utilities/make.html

Autotools

要維護跨平台的 *Makefile* 相容不容易。讓專案使用者自行修改 *Makefile* 也不是很方便的做法。因此，出現 Autotools 這類自動生成 *Makefile* 的軟體。解決了原本 *Makefile* 在異質平台間的的相容性議題。

Autotools 在 C 專案中以 *configure* 這隻命令稿存在。要編譯 C 專案時，先呼叫 *configure* 命令稿：

```
$ ./configure
```

這時候 *configure* 會自動偵測系統的狀態，生成相對應的 *Makefile* 。若宿主系統不符合編譯此專案的需求時，*configure* 會中止並吐出提示訊息。此外，呼叫 *configure* 時，可額外加入參數，用來改變 *configure* 的行為。

然後用 `make(1)` 編譯專案：

```
$ make
```

如果要把編譯好的 C 程式安裝到系統目錄，可使用以下指令：

```
$ sudo make install
```

在 Unix 或類 Unix 系統上拿到上游 (upstream) 原始碼專案時，通常都是使用這三個步驟來編譯及安裝程式。

但 Autotools 僅限於 Unix 或類 Unix 系統上才能使用，Windows 則無法使用以 Autotools 管理的 C 專案。所以才會出現 MSYS2 這類開發工具，用來解決在 Windows 系統上使用 Autotools 的議題。

但這樣處理算是繞一圈，畢竟 Windows 缺乏原生的 Autotools 支援。因此，出現 CMake 這類跨平台的專案管理軟體。請見下一小節。

CMake

CMake 和 Autotools 概念相似，但支援更廣。不僅在 Unix 或類 Unix 系統上可生成 *Makefile* ，在 Windows 上也可生成給 Visual Studio 使用的 MSBuild 設定檔。此外，CMake 還支援一些較少見的專案設定檔格式，像是 Code::Blocks 的設定檔。這裡[6]列出 CMake 所支援的專案設定檔格式。

CMake 在近幾年受到商業公司的注目。像是 CLions 一開始就採用 CMake 管理專案。甚至 Visual Studio 開始接受以 CMake 為基礎的專案 (參考這裡[7])。微軟主導的 C 和 C++ 套件管理軟體 vcpkg[8] 也以 CMake 管理套件。所以 CMake 的確是值得注意的自動編譯系統。

[6] https://cmake.org/cmake/help/latest/manual/cmake-generators.7.html
[7] https://docs.microsoft.com/en-us/cpp/build/cmake-projects-in-visual-studio
[8] https://docs.microsoft.com/en-us/cpp/build/vcpkg

Ninja

註：ninja 在日文中是忍者的意思。

前面數小節所述的自動編譯系統算是有點歷史的軟體。相對來說，Ninja[9] 則是較新的自動編譯系統。

Ninja 著重於速度而非易寫性，故不建議直接用 Ninja 來管理 C 專案。甚至 Ninja 官方團隊也列出可以生成 Ninja 專案設定檔的開發工具[10]。前一小節介紹的 CMake 就可以生成 Ninja 專案設定檔，所以不需要刻意去學 Ninja 的語法。

Bazel

Bazel[11] 原本是 Google 內部使用的自動編譯系統的一部分，後來成為開放原始碼專案。Bazel 可以用來管理 C、C++、Objective-C、Java、Android 等類型的專案。Bazel 主要是針對大型專案而設計，小型專案無法充分感受 Bazel 的益處。

C 專案產生器 (C Project Generator)

C 語言並沒有內定的專案產生器，一些 C 和 C++ IDE 自動建立 C 專案的功能只是輔助，而不是必需的。正規的方法是慢慢手動建目錄、C 原始檔、自動編譯設定檔、輔助文件等。因為 C 沒有制式的專案架構，只要專案架構合理即可。

網路上有一些 C 專案產生器，像是筆者自己也寫了一個[12]。但這些 C 專案產生器只是把原本手動建置專案的過程用電腦程式自動化而已，真正管理專案的還是 Make、CMake 等自動編譯系統。所以這些 C 專案產生器沒有成為市場主流。

結語

在本文中，我們介紹了 C 專案的概念及建立方式。在後續的章節中，本書會展示以 CMake 為基礎的應用程式專案和函式庫專案。

[9] https://ninja-build.org/
[10] https://github.com/ninja-build/ninja/wiki/List-of-generators-producing-ninja-build-files
[11] https://bazel.build/
[12] https://github.com/cwchentw/mkg

前言

雖然 C 語言沒有支援物件導向程式的語法，但我們可以在一些真實世界的專案看到具有物件導向思維的 C 程式碼，一些知名的例子像是 Linux 核心和 GTK+ 等。

然而，C 語言的教材甚少直接探討這個議題，大部分教材的態度是在教完 C 核心語法後就轉往 C++，像是 *C 程式設計藝術 (全華圖書)* 的前半部是 C 語言，後半部就塞入 C++ 相關的內容來增加版面。這些教材隱含著一個思想：C 只能寫命令式程式，要寫物件導向程式就要轉用 C++。

那麼，我們為什麼要用 C 語言寫物件導向程式呢？

物件導向是什麼？

物件導向是一種程式設計的範式 (paradigm)，理論上是抽象的程式設計思維，但卻會受到程式語言特性的影響，從每個程式語言的語法就可以看到不同程度的物件特性。

對物件導向來說，最基本的特性是資料 (data) 和行為 (behavior) 連動所帶來的狀態 (state) 改變；再進一步就是封裝 (encapsulation)、組合 (composition)、繼承 (inheritance)、多型 (polymorphism) 等特性；一些相對次要的特性包括建構子 (constructor)、運算子重載 (operator overloading) 等。如果仔細觀察不同語言，就會發現不同語言對上述特性的支援度不同。

用 C 語言撰寫物件的方式

對於 C 語言來說，如果要創造新的複合型別，就是用結構 (struct) 將該型別的屬性 (fields) 包起來。以下我們以二維空間的點 (point) 為例，來建立一個類別和相關的方法，這是一個常見的例子：

```
#include <stdlib.h>

/* Declare point_t class. */
typedef struct point {
    double x;
    double y;
} point_t;

/* The constructor of point_t. */
```

(continues on next page)

(continued from previous page)

```
point_t* point_new(double x, double y)
{
    point_t* self = \
        (point_t*) malloc(sizeof(point_t));

    self->x = x;
    self->y = y;

    return self;
}

/* The getter of x. */
double point_x(point_t* self)
{
    return self->x;
}

/* The setter of x. */
void point_set_x(point_t* self, double x)
{
    self->x = x;
}

/* The getter of y. */
double point_y(point_t* self)
{
    return self->y;
}

/* The setter of y. */
void point_set_y(point_t* self, double y)
{
    self->y = y;
}

/* The destructor of point_t. */
void point_delete(point_t* self)
{
    if (!self)
        return;

    free(self);
}
```

在我們這個例子中，point_t 類別內的屬性和方法已經有基本的連動，但除此之外，就什麼都沒有；我們沒用 opaque pointer 將物件封裝，也沒有實作其他的物件導向特性。

point_t 算不算一個物件呢？

我們延續 point_t 類別的例子，來看外部程式如何使用 point_t 物件：

```
// Excerpt.

int main(void)
{
    /* Create a point_t object. */
    point_t* pt = point_new(0, 0);

    /* Access x and y. */
    printf("(%.2f, %.2f)\n", point_x(pt), point_y(pt));

    /* Mutate x and y. */
    point_set_x(pt, 3);
    point_set_y(pt, 4);

    /* Access x and y again. */
    printf("(%.2f, %.2f)\n", point_x(pt), point_y(pt));

    /* Free the object. */
    point_delete(pt);

    /* Return the program status. */
    return 0;
}
```

在我們這個例子中，除了手動將 point_t 物件帶進函式的語法有點 verbose 以外，這個例子就是基本的物件屬性存取。這樣的程式碼表面上看起來不太物件導向，但物件和函式已有基本的連動了。

物件導向 C 的思維

由於 C 語言沒有內建的物件導向語法，撰寫物件導向程式時都要自己撰寫額外的樣板程式碼去模擬一些物件導向的特性。這些做法並沒有真正的標準，都是程式設計者對某項物件導向特性解構後重新用 C 語言去實作。

像是有開發者以 C 語言巨集寫了一整套的輕量級物件系統 (見 lw_oopc[13])，我們可以參考該物件系統的寫法，甚至也可以直接將該物件系統拿來用，但這些物件導向的語法就不適合放在 C 語言的標準裡。

由於 C 的物件導向程式沒有標準的做法，我們看到某一套 C 物件系統的實作時，要去思考該寫法背後的思維，為什麼要這樣寫？解決了什麼語法特性？而不僅是直接硬背下來。

[13] https://github.com/Akagi201/lw_oopc

程式設計討論區上其實也不乏相關的討論 (像這裡[14])，由一些相關的討論，可以看出不同開發者對這個議題的態度不同。有一派開發者會說「對，我們可以這樣做」，然後就會開始討論一些用 C 模擬物件導向的技巧；另一派則會直接說「為什麼不直接用 C++」。

Stroustrup 博士在做 cfront (C++ 轉 C 的轉譯器) 時應該也想過類似的問題，當時的答案就是做出一個保留 C 特性的新語言，也就是廣人熟知的 C++。

為什麼要寫物件導向 C？

那麼，我們什麼時候會用 C 語言寫物件導向程式呢？通常是在維護現有程式碼和需要使用 C 而非 C++ 或其他語言時。

程式設計初心者會以為追逐新興語言很重要，但是，我們並不會因為 Rust 這類新興編譯語言出現後就把整個軟體專案翻掉再來一次，通常會有更強烈的理由才這麼做；在網路上有時會看到用 Rust 重寫某個類 Unix 系統指令或工具的社群專案，往往就是得到一個功能不齊的次級品，而對資訊界沒有實質的貢獻。

有時候我們的目標環境不允許我們用 C++ 或其他比較肥大的語言，只能使用 Bash、C、Lua 等相對節省運算資源的工具，這時候用一些物件導向的手法整理程式碼的確會有所幫助。

撰寫物件導向 C 的注意事項

C 語言無法獲得完整的物件導向特性，但撰寫一些基於物件的特性的程式碼倒是沒有問題。像是我們可以透過 opaque pointer 和 static function 很容易就獲得具有封裝概念的物件，這種輕量級物件對於整理程式碼相當有幫助。

相較起來，要在 C 語言中撰寫具有多型特性的程式就比較費工，需要撰寫較多的樣板程式碼來實踐該特性。至於繼承則是無法取得的特性，只能用組合的方法來模擬。C 語言畢竟是程序性的 (procedural) 程式語言，當我們需要撰寫大量樣板程式碼來滿足某些語法特性時，或許我們誤用了工具。

結語

除了實用的觀點，用 C 撰寫物件導向程式也是很好的頭腦體操。由於 C 沒有內建的物件導向語法，我們可以重新思考到底我們平日所熟悉的物件導向特性想要達成什麼目的，我們需要用什麼 C 語言特性來滿足這些需求；藉由這個解構再組合的過程，我們對物件導向又有進一步的了解。

[14] https://www.quora.com/Why-dont-most-C-programming-books-teach-object-oriented-skills-in-C-but-gear-toward-C-soon-after-b

當然，物件導向只是撰寫程式的過程中所用到的一些特性，不是最後的產品，我們也不需要一直沉醉在這樣的頭腦體操中；我們已經有 C++ (或 Java 或 C#) 了，如果能直接用現有的工具就能解決問題，何必重造輪子呢？