目錄

Google C++ Style Guide 繁體中文版

本書翻譯了 Google 官方訂定的 Coding Style (程式碼風格) 標準。

如果程式碼都能夠遵照同一份標準來撰寫的話，就能夠大大地增加程式的可讀性。 Google 所訂定的 C++ 程式碼風格非常明確，也非常仔細。 我認爲是一個很不錯的標準。

不過可惜的是，Google 官方只有提供英文版的說明。爲了讓不善英文，並使用中文的人能夠理解文件的內容，特別撰寫本書供大家閱讀。

版本

目前此文的版本爲：release-20170126-001

對應的原文版本

請注意原本的 Google C++ Style Guide 也會與時俱進，英文本的內容會不斷修改。 因此這裡標註出此書翻譯時對應的英文版原件：

- 目前本書內容對應到的版本爲 Google 官方 Repository 2016/12/31 Commit 的版本

需要支援

有些部分我個人覺得翻譯仍不太準確，而且翻得很爛。 例如「背景」那章的「風格指引的目標」下的內容，普遍來說翻譯應該都不太好。 因此我需要大家提供意見幫我改進那邊的翻譯。 基本上就對照中英文看一下，然後開一個 issue 讓我知道一下怎麼修改會比較好。

一些撰寫時的規則

- 所有文句除標題外，一律以句號結尾。
- 句號之後若還有其他語句在同一段，一律加上一個半形空白。
- 英文單字與中文字詞的銜接處一律加上一個半形空白。

英文專有名詞的處理

這點我想了很久，有些程式中的關鍵字像是 class 或者 function 之類的，到底該使用中文的翻譯名詞呢？還是直接打英文名詞？

使用中文專有名詞的好處是，對於直接從中文教材學習的人來說，很快就可以進入狀況。 不過有可能我翻譯的名詞與這些人當初所學的不同，這樣或許也沒有好處。 而且對於本來就學英文的人來說，一下子看到中文名詞可能很難進入狀況。 事實上，我個人平常是中英文夾雜

使用的。 遇到專有名詞就切換成英文，一般解釋時就用中文。 但是無論如何，我必須要訂出一個統一的標準貫穿本書，以方便讀者閱讀。

我想來想去，覺得這本書主要的目還是在於翻譯。因此除了程式碼的部分外，盡量少用英文名詞較好。 所以我最後的方針是：

> 除非是非得使用英文的情況，不然程式的專有名詞一律翻譯成中文。 少見名詞第一次出現時，在旁標明英文。 常用的名詞則直接使用中文翻譯。

爲了方便只知道英文名詞的人能夠查閱看不懂翻譯的名詞，我在附錄列了一張中英專有名詞的對照表。 希望多少有點幫助。

回報與修正

如果覺得哪裡翻譯有誤，或者覺得語句不通順，歡迎到 Github repository 的 issue 提出問題。 如果熟悉 Git 的話，也歡迎在 Github 上 fork 這個 repository 自行修改，並開啓 pull request 請求合併。

相關連結

- 本書的 Github Repository：https://github.com/SLMT/google-cpp-style-guide-zh-tw
- 本書的 GitBook 網址：https://www.gitbook.com/book/slmt/google-cpp-style-guide-zh-tw
- Google 官方 C++ Style 說明文件 (英文)：https://google-styleguide.googlecode.com/svn/trunk/cppguide.html

背景

C++ 是許多 Google 開源專案 (Open Source Project) 中常用的程式語言。 如同許多 C++ 程式設計師所知，C++ 具有許多強大的特性，但是這也帶來許多複雜性，使得程式碼容易產生各種錯誤 (Bug)，並且也難以閱讀及維護。

本指南的目標在於藉由說明那些東西該寫、哪些不該寫，來控制 C++ 程式碼的複雜度。 指南中提及的規則可以讓程式碼易於維護，同時也讓程式設計師能夠好好發揮 C++ 帶來的強大生產力。

Style (風格) 也稱作 Readability (可讀性)，可以說是一種管理 C++ 程式碼的約定。 事實上，用 Style 這個字眼可能不太精確。 因爲這類約定並非只有著墨在程式碼檔內的排版與格式。

幾乎所有 Google 開發的開源專案皆遵守著本指南。

注意：這份指引並非教你如何寫 C++，我們假設了讀者已經非常熟悉這個程式語言。

風格指引的目標

爲什麼會有這份文件？

這邊有幾個核心目標是我們相信這份文件該提供的。 這些目標就是構成這些規定的主要原因。 藉由提出這想法，我們希望能夠攤開來討論，並且讓大家理解爲什麼會有這些規則，以及某些特別的規定是如何訂定出來的。 如果你能了解每個規則是爲了甚麼目標而制定，那大家應該也能更容易了解某些規定被丟棄時，有哪些必要的考量以及反論。

這份文件目前我們認爲的目標有以下幾點：

規則需要具有足夠的份量

一條風格規則所帶來的好處必須大到足夠合理來要求我們的工程師記住它。 它的好處是以沒有這份指南的情況下撰寫出的程式碼爲準來衡量的。 有些規則可能在一些極端不好的程式碼中具有些微的好處，但是這種程式碼一般人不太可能會寫得出來的話，我們就不會寫進這份指引。 這個原則解釋了大多數我們沒有撰寫的規則，而不是我們會有寫出的規則。 舉例來說，`goto` 違反了下面很多條原則，但是基本上很少人會使用它，因此這份風格指引就不討論它。

並非爲了撰寫者，而是爲了讀者來優化

我們預計我們的程式碼庫 (以及大多數提交到上面的各個組件) 會持續維護很長的一段時間。 因此大多數的時間會花在讀懂它，而非撰寫它。 我們明確地選擇優化我們平凡工程師在於讀、維護以及偵錯程式碼的體驗，而非撰寫的便利性。 「爲讀者留下足跡」正是一種在這個原則之下常見的觀點：如果在一段程式碼中正在發生一些驚喜或者不尋常的事情 (例如：轉移一個指標的所有權)，在這裡留下一些文字上的提示將會非常有用。 (`std::unique_ptr` 明確地在呼叫處展示了所有權轉移)。

與現有的程式碼保持一致性

在我們程式庫中使用一致的風格讓我們可以專注在其他 (更重要) 的問題。 一致性同時也允許了自動化：那些自動幫你調整格式或者 `#include` 位置的工具只在你的程式碼如它預期般地一致時才能正常運作。 在很多狀況下，那些標記爲「保持一致性」的規則其實是肇因於「不要想太多，挑一個就好」。 在那些情形允許彈性的潛在好處其實大於大家在那邊爭論所花費的時間。

適當時與盡可能大多數的 **C++** 社群保持一致性

跟其他組織使用 C++ 的方式保持一致所帶來的價值與在我們自己的程式庫中保持一致性的理由是相同的。 如果一個 C++ 標準的特性可以解決一個問題，或者一個寫法被大眾所接受，那就可以當成使用它的理由。 然而，有時候這些特性與寫法可能具有某些缺陷，或者並不是我們的程式庫所需要的。 在這類情況下，其實更適合限制或者禁止這些特性。 在某些情形，如果我們無法感受到特別的好處，或者沒有足夠的價值，比起那些定義在 C++ 標準上的函式庫，我們更喜歡從頭到尾自己撰寫或者直接使用一個第三方的函式庫。

避免令人驚訝或者危險的結構

C++ 有著一些比看起來更令人訝異或者危險的特性。 這份風格指引內的部分限制就是爲了要避免掉入這種陷阱裡。 風格指引對於放棄那些限制具有很高的標準，因爲放棄那些規則很有可能會對程式的正確性造成很大的危機。

避免那些讓我們的平凡 **C++** 程式設計師認爲詭異或者難以維護的結構

有些 C++ 的特性可能一般來說不太適合被使用，因爲它們可能常造成一些額外的複雜性。 在一些廣泛被使用的程式碼中可能會適合使用特別的語言結構，因爲複雜實作所帶來的好處會被廣泛使用這點放大，而且了解這些複雜東西的代價不需要在寫新一塊程式碼時再度付出一次。 如果有疑問的話，可以向專案領導詢問是否可以拋棄這類的規則。 這對我們的程式庫特別重要，因爲程式碼的管理團隊與團隊內部的成員會隨著時間改變：儘管現在正在使用某段程式碼的每個人都了解它，經過了幾年後也不能保證仍會是如此。

在我們的規模下要非常小心

對於一個超過 1 億行以及有著數千位工程師管理的程式庫來說，一個工程師的一些錯誤或者過於簡化的程式碼可能會造成嚴重的後果。 舉例來說，要特別注意不能汙染全域命名空間：如果大家都把東西放在全域命名空間的話，對於這種具有數億行程式碼的程式庫來說可能會很難避免命名衝突，並使得程式庫難以維護。

承認必要的優化

雖然有些效能上的優化可能會牴觸這份文件的一些原則，但是對於必要的情況來說還是可接受的。

這份文件的用意在於最大化地提供指引與適當的限制。 一如往常，我們的盡量使規則符合常識或看起來順眼。 我們特別參考了整個 Google C++ 社群建立起的傳統，而非單獨考量你個人的喜好或者你的團隊。 當你看到不尋常的結構時，請善用你的智慧並保持懷疑：沒有限制它們並不代表你可以忽略它們。 此時請你自行判斷，如果你不太確定，請不要猶豫，立即詢問你的專案領導來取得意見。

標頭檔 (Header Files)

一般來說，每一個 `.cc` 檔都應該要有一個對應的 `.h` 檔。 不過也有一些常見的例外，像是單元測試 (Unit Test) 跟一些只包含著 `main()` 的小型 `.cc` 檔就不需要有。

正確地使用標頭檔可以對可讀性、程式碼大小與效能帶來巨大的影響。

以下這些原則會引領你克服標頭檔中各式各樣的陷阱。

自給自足標頭檔 (Self-contained Headers)

> 標頭檔應該要自給自足 (self-contained)，而且副檔名必須是 `.h`。 專門用來插入文件的非標頭檔，則應該要使用 `.inc` 作爲副檔名，並且應該盡少使用。

所有標頭檔都應該要自給自足。 換句話說，使用者或者重構工具 (Refractoring Tool) 並不需要依賴任何額外的條件才能夠插入標頭檔。 更精確地說，標頭檔應該要包含 標頭檔保護，而且應該要自己插入所有需要的其他標頭檔。

建議將模板與行內函式的定義放在同樣的檔案作爲宣告。 所有使用到這些東西的 `.cc` 檔都應該要載入這些結構，不然在某些建置設定下會造成程式無法連結。 如果將定義與宣告分別在不同檔案，載入宣告的話也應該要能夠同時載入其定義。 不要將這些定義移至額外的 `-inl.h` 檔中。 這種作法在過去很常見，但是從現在開始不允許這麼做。

有一個例外是，函式模板的顯式實體化 (explicitly instantiated) 或者該模板是一個類別的私有成員的話，可以只定義在實體化該模板的 `.cc` 檔中。

在某些極少數的狀況下，標頭檔可以不用是自給自足的。 這些特殊的標頭檔通常是用來載入程式碼到做一些不尋常的位置，例如載入到另一個檔案的中間。 他們可以不使用 標頭檔保護，而且可能沒有載入他們所需的檔案。 這種類型的檔案應該使用 `.inc` 作爲副檔名。 盡量別使用這種檔案，可能的話還是盡量用自給自足的標頭檔。

`#define` 保護 (The `#define` Guard)

> 所有的標頭檔應該要包含 `#define` 保護，以防止多重載入。 其名稱的格式爲 `<專案名稱>_<路徑>_<檔名>_H_`。

爲了保證名稱的獨特性，應該要遵照該檔案在專案中的完整路徑來定義。 例如，一個在專案 foo 之中 `foo/src/bar/baz.h` 位置下的檔案，其保護應該要這樣寫：

```
#ifndef FOO_BAR_BAZ_H_
#define FOO_BAR_BAZ_H_

...

#endif  // FOO_BAR_BAZ_H_
```

前向宣告 (Forward Declarations)

> 盡可能地避免使用前向宣告。 只要 `#include` 你需要的標頭檔就好。

定義

類別、函式與模板的前向宣告指的就是在沒有定義其內容的情況下，預先宣告名稱的程式碼。

優點

- 前向宣告可以節省編譯時間。 `#include` 會使得編譯器處理時必須要開啓更多檔案而且處理更多資料。
- 前向宣告可以避免不必要的重新編譯。 `#include` 有可能造成你的標頭檔有點修改，其他相關的程式碼就得需要被重新編譯。

缺點

- 前向宣告可以用來隱藏某些依賴關係，可能會造成使用者的程式碼在標頭檔修改後略過了必要的重新編譯過程。
- 一個前向宣告可能會因爲其函式庫內部的修改而損壞。 函式與模板的前向宣告會造成撰寫標頭檔的人無法更改 API。 舉例來說，增加函式接受的參數，給模板增加一個預設數值，或者轉移到一個新的名稱空間。
- 前向宣告 `std::` 名稱空間內的符號時常產生一些未定義的行爲。
- 有時候很難界定到底是否該在程式碼中使用前向宣告，或者全部使用 `#include`。 有時候替換掉 `#include` 可能會大幅改變程式碼的意義：

```
// b.h:
struct B {};
struct D : B {};

// good_user.cc:
#include "b.h"
void f(B*);
void f(void*);
void test(D* x) { f(x); } // calls f(B*)
```

如果上面的程式碼中，將 `#include` 替換成 `B` 跟 `D` 的前向宣告的話，`test()` 就會呼叫 `f(void*)`。

- 從一個標頭檔前向宣告多個符號比單純地 `#include` 更難在發生錯誤時除錯。
- 爲了使用前向宣告而重構程式碼（像是把把物件成員換成指標），可能會造成程式變慢或者更加複雜

抉擇

- 盡量避免前向宣告其他專案中的實體
- 當使用一個宣告在標頭檔內的函式時，總是 `#include` 那個標頭檔
- 當你要使用類別模板時，盡量使用 `#include`

請參考「`#include` 時的名稱與順序」來參考何時應該插入標頭檔

譯註

老實說我也是第一次看到「前向宣告」這個詞，因此我花了點時間研究這到底是什麼東西。我稍微閱讀了維基百科上的資料之後，在此寫下我對這個這個詞的理解。

事實上，前向宣告這個詞簡單來說就是「在定義之前先宣告」。 相信應該不少人有宣告過函式吧？像是：

```
int sum(int, int);
```

這裏你可以看到我們並沒有寫說 `sum` 這個函式代表著什麼意思，只說他接受兩個整數參數，回傳一個整數值，而實際上 `sum` 的內容則定義在別的地方。 這個目的就是在告訴編譯器：「有個叫做 `sum` 的函式存在，等下使用時，請把 `sum` 這個符號當成一個接受兩個整數參數，回傳一個整數的函式處理。」

其實這就是一個「前向宣告」，也就是預先告知這個東西的存在，然後再另外定義內容。

前向宣告除了函式之外，也可以用在類別與模板上。 假設現在其他檔案中定義了 `class A`，而你的檔案需要用到它，你可以單純地宣告：

```
class A;
```

這樣就可以不用使用 `#include` 來載入相關的標頭檔。 但是這個使用有個限制，就是程式碼中使用到 `class A` 的地方，都只能使用指標或參考，而不能直接使用像是 `A a` 這樣的變數。 使用類別的前向宣告要注意的事情相對比較多，所以上面的建議才會提到大部份的情況還是直接 `#include` 比較好。

行內函式 (Inline Functions)

> 只有在函式程式碼少於或等於 10 行時才將它宣告爲行內函式

定義

透過宣告函式爲行內函式，可以讓編譯器直接在呼叫該函式的地方展開函式，而不是遵照一般的函式呼叫機制編譯。

優點

將函式宣告爲行內函式可以產生更有效率的目的碼 (object code)，因爲行內函式比起一般的函式小很多。 你可以盡量將存取函式 (accessor) 、修改函式 (mutator) 以及一些極短但對效能有巨大影響的函式行內化。

缺點

過度使用行內函式可能會造成程式變慢。 依照函式長度的不同，行內化可能會增加或減少程式碼的大小。 行內化一個很小的存取函式通常可以減少程式碼的長度，不過行內化一個很大的函式可能會巨幅地增加長度。 現在的處理器因爲指令快取 (instruction cache) 的關係，處理較短的程式碼通常會更快。

抉擇

一個適當的規則是不要將 10 行以上的函式行內化。 其中要特別注意解構函式 (destructors)，解構函式常常比你所看到的還要長。 因爲解構函式還會隱性地另外呼叫成員以及基底類別 (base class) 的解構函式。

另一個有用的規則：一般來說將一個有迴圈或者 `switch` 的函式行內化對效能並沒有幫助 (除非大多數的情況下這個迴圈或 `switch` 都不會被執行到)。

要特別注意的是，就算將一個函式宣告爲行內函式，編譯器也不一定會照做。 例如：虛擬函式 (virtual function) 或遞迴函式 (recursive function) 常常不會被行內化。 因爲遞迴函式一般來說不該是行內函式。 至於將虛擬函式寫成行內函式的理由，通常只是爲了要方便將函式的定義放在類別內而已 (例如類別的存取函式或修改函式)。

譯註

這邊提供一個行內函式的範例，`inline` 關鍵字是將函式行內化的關鍵：

```
inline int sum(int a, int b) {
    return a + b;
}
```

`#include` 時的名稱與順序

> 利用右列順序 `#include` 標頭檔，避免隱藏的依賴關係：直接相關的標頭檔、C 函式庫、C++ 函式庫、其他函式庫 `.h` 檔、你的專案的 `.h` 檔。

所有標頭檔的路徑應該都要以專案的程式碼目錄爲起點，並且不要使用 UNIX 資料夾簡稱，像是 `.` (現在) 跟 `..` (上一個目錄)。 舉例來說，`google-awesome-project/src/base/logging.h` 檔案應該要被這樣載入：

```
#include "base/logging.h"
```

假設現在有 `dir/foo.cc` 或 `dir/foo_test.cc` 檔案，其目標在於實作或測試 `dir2/foo2.h` 檔內的東西，那麼 `#include` 的順序應該這樣寫：

1. `dir2/foo2.h`
2. C 系統檔
3. C++ 系統檔
4. 其他函式庫 `.h` 檔
5. 你的專案的 `.h` 檔

依照這個順序，如果 `dir2/foo2.h` 遺漏了任何必要的 `#include`，那麼在建置 `dir/foo.cc` 或 `dir/foo_test.cc` 的時候就會中斷。 因此這確保了建置會先在這些檔案中斷，而不是在其他無辜的地方發生。

`dir/foo.cc` 與 `dir2/foo2.h` 通常會放在同一個資料夾下，像是 `base/basictypes_test.cc` 跟 `base/basictypes.h`，但是有時候也有可能會分開放。

每個區塊中的檔案應該要依照字母順序排列。要注意一些比較老的專案中可能沒有遵照這個規則，這些都應該要等方便的時候修改過來。

你應該要 `#include` 所有包含你使用到任何符號的標頭檔 (除非你使用了前向宣告)。如果你使用了 `bar.h` 中的符號，別期待你 `#include` 了 `foo.h` 之後，`foo.h` 裡面會包含著 `bar.h`，此時你應該也要 `#include` `bar.h`，除非 `foo.h` 有明顯地展現出它提供了 `bar.h` 中的符號。另外，已經在相關的標頭檔中 `#include` 過的東西，可以不用在 `cc` 檔中 `#include` (像是 `foo.cc` 可以依賴在 `foo.h` 上)。

舉個範例，`google-awesome-project/src/foo/internal/fooserver.cc` 檔中的 `#include` 可能長這樣：

```
#include "foo/server/fooserver.h"

#include <sys/types.h>
#include <unistd.h>
#include <hash_map>
#include <vector>

#include "base/basictypes.h"
#include "base/commandlineflags.h"
#include "foo/server/bar.h"
```

特例

有時候，只能在某些系統中使用的程式碼需要有條件地 `#include`，這種就可以程式碼就可以放在所有 `#include` 之後。當然，盡可能地讓這種程式碼越少且影響範圍越小越好。例子：

```
#include "foo/public/fooserver.h"

#include "base/port.h"  // For LANG_CXX11.

#ifdef LANG_CXX11
#include <initializer_list>
#endif  // LANG_CXX11
```

作用域 (Scoping)

名稱空間 (Namespace)

> 除了某些特殊情況外，程式碼應當放在名稱空間中。 名稱空間應該要有個跟專案名稱有關、獨一無二的名字，最好也把路徑考慮進去。 不要使用 using 指示詞 (using-directive)，像是 `using namespace foo`。 不要使用行內名稱空間 (inline namespace)。 關於未被命名的名稱空間，請參考「無名名稱空間與靜態變數」一章。

定義

名稱空間將全域區分爲多塊分開、各自命名的作用域，這可以有效避免在整個作用域中遇到名稱相同的狀況。

優點

名稱空間提供了一種避免在大型程式中名稱相衝的同時，也能讓大部分程式碼使用較短名稱的方法。

例如，兩個不同的專案在全域都具有 `Foo` 這個類別，這個符號可能會在編譯或執行時發生衝突。 如果這些專案能把他們的程式碼分別放在不同的名稱空間下，那麼 `project1::Foo` 跟 `project2::Foo` 就會被視爲不同的符號，也不會有衝突的問題，而且在各自的專案中還能夠繼續使用 `foo` 這個名字同時不需加上前綴名稱。

行內名稱空間 (inline namespace) 會自動把它們的名稱放進作用域中。 例如，請參考以下程式碼：

```
namespace X {
inline namespace Y {
  void foo();
} // namespace Y
} // namespace X
```

這會產生 `X::Y::foo()` 等同於 `X::foo()` 的效果。 當初行內名稱空間的主要目標就是用來處理不同版本的 ABI (Application Binary Interface, 應用二進位介面) 的兼容問題。

缺點

名稱空間可能容易造成混淆，因爲它在類別存在的情況下，又額外提供了一種切分命名作用域的方式。

行內名稱空間可能也會造成混淆，因爲它的作用域有時並非如它當初宣告的那樣。 只有在某些需要分類不同版本的情況下才可能會有幫助。

在某些程式碼中，可能會需要使用完整的名稱來指出所需的名稱空間。 對於深層的名稱空間來說，這樣的寫法可能會造成一些混亂。

抉擇

名稱空間應該如下列般使用：

- 遵從「命名」的「名稱空間命名」一章的規則。
- 如同範例中在名稱空間的尾端加上註解。
- 名稱空間必須將整個原始碼檔中，`#include` 、gflags 定義和前向宣告其他類別之後的內容包裹起來。

```
// 在某個 .h 檔中
namespace mynamespace {

// 所有的定義都在名稱空間的作用域內
// 注意沒有使用縮排
class MyClass {
 public:
  ...
  void Foo();
};

}  // namespace mynamespace
```

```
// 在某個 .cc 檔中
namespace mynamespace {

// 函式的定義也放在名稱空間的作用域內
void MyClass::Foo() {
  ...
}

}  // namespace mynamespace
```

一般的的 `.cc` 檔中可能會有更多複雜的細節，包含其他名稱空間的類別等等。

```
#include "a.h"

DEFINE_bool(someflag, false, "dummy flag");

class C;  // 前向宣告全域名稱空間下的類別 C
namespace a { class A; }  // 前向宣告 a::A.

namespace b {

... b 的程式碼 ...        // Code goes against the left margin.

}  // namespace b
```

- 別在 `std` 名稱空間下宣告任何東西，甚至是標準函式庫的前向宣告。 宣告 `std` 名稱空間下的作法沒有一個統一的標準，並不是一個跨平台的作法。 如果想要宣告標準函式庫內的實體，請直接 `#include` 那些標頭檔。
- 你不該爲了要讓某個名稱空間中的名稱皆可直接呼叫而使用「using 指示詞 (using-directive)」

```
// 禁用 -- 這會汙染你的名稱空間
using namespace foo;
```

- 別在標頭檔中的名稱空間中使用名稱空間別名 (Namespace Alias)，除非是在明確被標示爲只給專案內部使用的名稱空間。 因爲在一個被載入的標頭檔內的任何東西，都會被當作該檔的公共 API 對待。

```
// 在 `.cc` 檔中簡化存取某個常用名稱的動作
namespace fbz = ::foo::bar::baz;
```

```
// 在 `.h` 檔中簡化存取某個常用名稱的動作
namespace librarian {
// 以下別名可以被任何載入這個標頭檔的檔案使用 (在 librarian 名稱空間內)
// 別名在同一個專案中應該要有一致性的命名方式
namespace pd_s = ::pipeline_diagnostics::sidetable;

inline void my_inline_function() {
  // 僅在這個函式內作用的別名
  namespace fbz = ::foo::bar::baz;
  ...
}
}  // namespace librarian
```

- 別使用行內名稱空間

譯註

就我所知，行內名稱空間是非常鮮見的東西，至少從我學習 C/C++ 到現在都沒有在一個實際的專案中看過。 這邊稍微介紹一下這個功能。

首先看下面這段範例：

```
namespace A {
namespace B {
void fun() {
    // code
}
}
}
```

一般來說，上面這段程式碼想要呼叫 `fun` 這個函式的話，就要使用 `A::B::fun()` 這個名稱。 但是如果此時引入了行內關鍵字，在 B 前面加上 `inline` ，變成這樣：

```
namespace A {
inline namespace B {
void fun() {
    // code
}
}
}
```

這個時候你使用 `A::fun()` 這個名稱的話，就可以呼叫到 `fun` 這個函式。當然，使用 `A::B::fun()` 也同樣可以呼叫得到 `fun`。有興趣的話可以自己試試看。

運作的原理很簡單，其實就是程式在編譯時將所有 `A::fun()` 的名稱代換爲 `A::B::fun()` 而已。

那這個技巧有甚麼用？既然我想讓其他人直接呼叫到 `fun`，那爲什麼還要特別放一個 `namespace B`？

這個技巧最常用到的地方，就像上面 Google 文件中所提到的，版本控管。

如果今天我是在寫函式庫，此時我寫了兩種不同版本的 `fun`，可是有些人的程式已經連結到了舊版的 `fun`，那要怎麼樣在不重新編譯那些程式的狀況下保持連結？這個時候可以這樣做：

假設原本的函式庫長這樣：

```
namespace SomeLib {
inline namespace v1 {
void fun() {
    // code
}
}
}
```

因此原本編譯的程式裡，呼叫到 `SomeLib::fun` 的地方都會連結到 `SomeLib::v1::fun`。而此時我只要改成：

```
namespace SomeLib {
namespace v1 {
void fun() {
    // code
}
}
inline namespace v2 {
void fun() {
    // code
}
}
}
```

這樣原本連結到 `SomeLib::v1::fun` 的程式仍然可以運作，而且新的程式呼叫 `SomeLib::fun` 時，也可以如我預期地使用到新的版本。 因此行內名稱空間才常用於函式庫的版本控管。

無名名稱空間與靜態變數

> 當你遇到一些放在 `.cc` 檔內，但不會被其他檔案參考到的定義，請將這些東西放在一個無名名稱空間中，或者將他們全部宣告爲 `static`。 不要在 `.h` 檔內使用任何前述結構。

定義

無名名稱空間可以對所有宣告給予內部連結性 (Internal Linkage)。 函數與變數則可以透過宣告爲 `static` 來給予內部連結性。 這代表你宣告的東西不能被其他檔案內的東西給存取。 如果其他檔案中宣告了同樣名稱的東西，那麼這兩者會被視爲獨立的個體。

抉擇

我們鼓勵在 `.cc` 檔中對那些不需要被其他檔案參考的程式碼使用內部連結性，但不要在 `.h` 檔中使用。

無名名稱空間的格式與一般名稱空間相同。 結尾註解的部分，名稱空間的名稱留白即可：

```
namespace {
...
} // namespace
```

非成員、靜態成員、全域函式

> 盡量將非成員函式放在名稱空間中，少用完全的全域函式。 盡量用名稱空間來組織函式，而不是用類別。 類別的靜態成員一般來說應該要與類別的實例或者靜態資料有關。

優點

非成員及靜態成員函式在某些狀況下很有用。 將非成員函式放在名稱空間中可以避免汙染全域的名稱空間。

缺點

非成員及靜態成員函式也許作爲某個類別中的成員會更合理，特別是這些函式會存取一些外部資源或者有高度相關時。

抉擇

有時候定義一個不受類別實例限制的函式很有用，甚至某些狀況下是必要的。 這樣的函式可以是非成員或者靜態成員函式。 非成員函式不應該依賴在某個外部變數上，而且應該放在某個名稱空間內。 不要特別爲了一群函式建立一個沒有分享任何靜態變數的類別，如果要組織起來的話，請使用名稱空間。 例如，在 `myproject/foo_bar.h` 標頭檔內，請寫：

```
namespace myproject {
namespace foo_bar {
void Function1();
void Function2();
}
}
```

而不要寫：

```
namespace myproject {
class FooBar {
 public:
  static void Function1();
  static void Function2();
};
}
```

如果你要定義一個非成員函式，而且這個函式只會用在某個 `.cc` 檔中，請利用內部連結性來限制它的作用域。

區域變數

> 將函式的變數盡可能地放在最小的作用域內，並在宣告變數的同時初始化

C++ 允許你在函式內的任何地方宣告變數。 我們鼓勵你盡可能地將變數宣告在越局部的 (local) 作用域越好，並且最好越靠近它第一次被使用的地方。 這讓讀者會更容易找到變數的宣告處以及了解它的型別是甚麼。 特別要注意初始化與宣告不該分開，例如：

```
int i;
i = f();      // 不好 -- 初始化與宣告分離
```

```
int j = g();  // 良好 -- 宣告同時初始化
```

```
vector<int> v;
v.push_back(1);  // 最好使用括號初始化法 (brace initialization)
v.push_back(2);
```

```
vector<int> v = {1, 2};  // 良好 -- v 一開始就初始化好
```

`if` 、 `while` 與 `for` 陳述句需要的變數一般來說應該要被宣告並限制在所屬的作用域內，例如：

```
while (const char* p = strchr(str, '/')) str = p + 1;
```

有一個要注意的特例：如果該變數是物件，那該物件的建構函式每次進入這個作用域時就會被呼叫一次，解構函式也會每次離開時都會被呼叫到。

```
// 低效率的寫法
for (int i = 0; i < 1000000; ++i) {
  Foo f;  // 我的建構函式與解構函式會各被呼叫 1000000 次
  f.DoSomething(i);
}
```

此時將這個變數宣告在迴圈外面，程式執行時會比較有效率：

```
Foo f;  // 我的建構函式與解構函式只會各被呼叫一次
for (int i = 0; i < 1000000; ++i) {
  f.DoSomething(i);
}
```

靜態與全域變數

不允許使用有著靜態生存期 (static storage duration) 的類別變數：這類變數之間建構、解構的順序並不固定，常常造成許多難以找出的錯誤。 然而，如果他們是 `constexpr` (常數表達式) 的話就可以用：因爲他們不會被動態初始化或解構。

有著靜態生存期的物件，包含全域變數、靜態變數、靜態類別成員變數和函式靜態變數，必須要是 POD (Plain Old Data)。 POD 只能包含整數、字元、浮點數、指標、陣列或者結構。

C++ 只有部分規範類別靜態變數在建構與解構時的執行順序，甚至每次建置之後的呼叫順序也有可能不同，這會造成一些難以找出的錯誤。 因此除了全域的類別變數外，我們也禁止用函式的結果來初始化靜態的 POD 變數，除非那個函式並不是依賴於其他全域資訊上 (像是 `getenv()` 或 `getpid()`)。 (這項禁令不適用於函式作用域內的靜態變數，因爲這類變數的初始化順序已經有嚴格定義過了，而且只會在程式執行到宣告位置的時候才會開始初始化)

同樣地，無論程式從 `main()` 返回而終止，或是因爲呼叫 `exit()` 終止，全域與靜態變數都會在程式結束的時候被摧毀。 解構函式被呼叫的順序被定義爲建構函式的呼叫順序的反向。 因爲建構函式的順序並不一定，因此解構函式也沒有固定順序。 例如，程式結束時有個靜態變數被摧毀了，但是程式碼還在跑，此時 (也許在其他線程中) 存取這個變數就會發生錯誤。 或是某個字串的解構函式會在另一個含有該字串的變數的解構函式之前執行。

一種緩解解構問題的方法是使用 `quick_exit()` 而不是 `exit()` 來終止程式。 差別在於， `quick_exit()` 不會呼叫解構函式，也不會呼叫那些利用 `atexit()` 註冊的 handler。 如果你有個需要在 `quick_exit()` 執行的 handler，你可以在 `at_quick_exit()` 註冊它。 (如果你有個 handler 需要在 `exit()` 與 `quick_exit()` 都執行，那你必須兩邊都要註冊。)

因此我們只允許靜態變數包含 POD 資料。 這條規則完全禁止使用 `vector` (請使用 C 的陣列)，或者 `string` (請使用 `const char []`)。

如果你需要一個類別的靜態或全域變數，請考慮從 `main()` 或者 `pthread_once()` 初始化一個指標 (永遠不會被釋放)。 注意這個指標必須要是原始指標 (raw pointer)，而不是智慧指標 (smart pointer)，因爲智慧指標的解構函式有我們極力避免的解構順序問題。

譯註

POD (Plain Old Data) 簡單來說指的就是不含明確定義的建構函式、解構函式與虛擬成員函式的類別。

附錄 **A**：常用專有名詞中英對照表

下表以英文名詞按照字母順序排序

英文 **(English)**	繁體中文 **(Traditional Chinese)**
Alias	別名
Array	陣列
Base Class	基底類別
Bug	錯誤
Class	類別
Compile	編譯
Compiler	編譯器
Constructor	建構函式
Declaration	宣告
Define, Definition	定義
Dependency	依賴關係
Destructor	解構函式
Directory	資料夾、目錄
Entity	實體
External Linkage	外部連結性
Forward Declaration	前向宣告
Function	函式
Global Scope	全域
Handler	(尚無翻譯)
Header, Header File	標頭檔
Inline Function	行內函式
Instance	實例
Internal Linkage	內部連結性
Input	輸入值
Library	函式庫
Link	連結

Linker	連結器
Local	局部的
Loop	迴圈
Namespace	名稱空間
Object Code	目的碼
Object File	目的碼檔
Output	輸出值
Parameter	參數
Pointer	指標
Private	私有的
Private Member	私有成員
Public	公用的
Public Member	公用成員
Raw Pointer	原始指標
Reference	參考
Smart Pointer	智慧指標
Statement	陳述句
Static Data Member	靜態資料成員
Static Member Function	靜態成員函式
Struct	結構
Symbol	符號
Template	模板
Thread	線程
Type	型別
Unnamed Namespaces	無名名稱空間
Virtual Function	虛擬函式