李松錡

Follow 303 Followers About

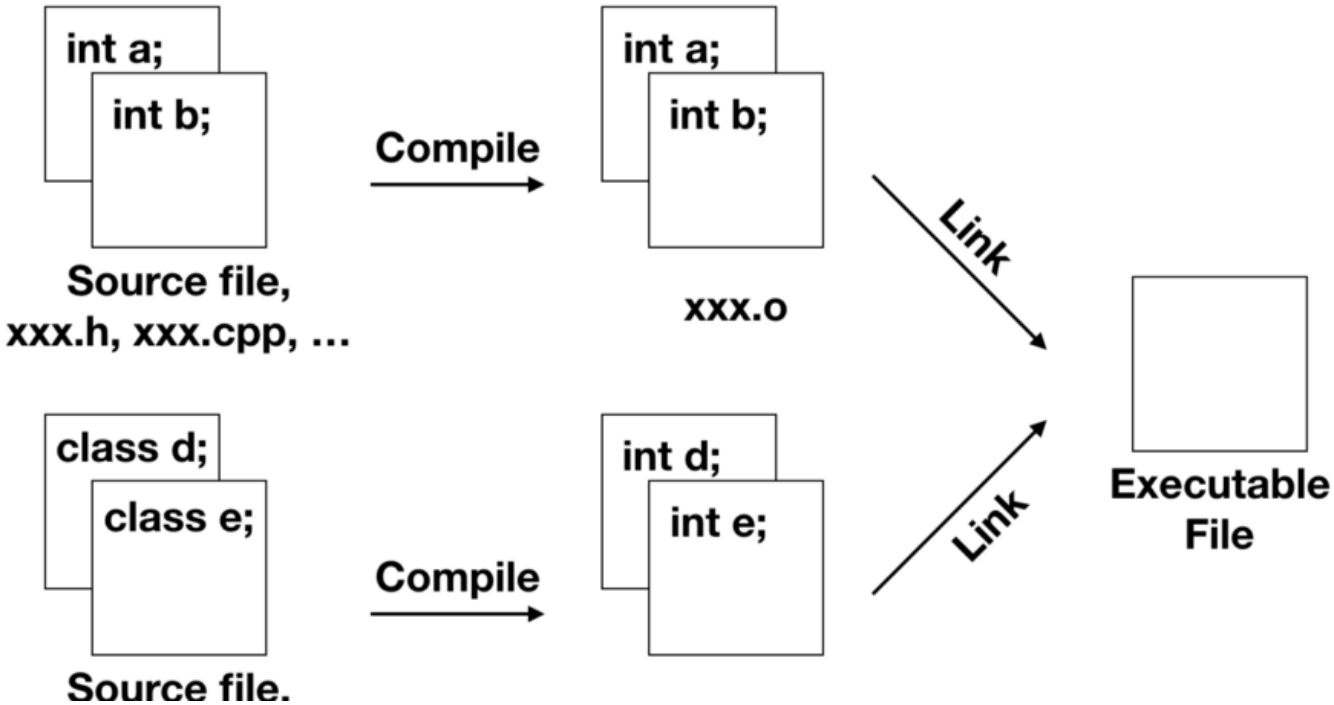
C/C++ 中的 static, extern 的變數

李松錡 Feb 3, 2018 · 11 min read

以前在大學了時候計程學的是 C++，但因為課程長度的關係，所以有很多比較複雜的觀念並沒有上到或是沒有弄得很清楚，最近因為在改一個 C++ 的 open source 的專案，所以就遇到了這個問題，於是去翻了一些資料，順手就把學到的東西記錄下來。

完整的 Compile 過程

首先如果你以前跟我ㄧ樣，都是在 windows 上用 codeblocks 或 dev c++ 這類的 IDE，那肯定對 C/C++ 的執行檔是怎麼出來的會很沒有概念，因此這邊我們不討論這些 IDE，我們下面會用 gcc/g++，了解 IDE 到底在編譯成執行檔的過程中經過哪些步驟。首先一般的編譯過程其實是長下面這個樣子的



我們人類用高階語言撰寫了各種 source file，也就是各種的 .h 和 .cpp 檔，然後每個獨立的 .h 和 .cpp 檔都應該要可以各自被 compile 成 .o 檔而不報錯，也就是說下列的問題可以被檢查出來:

- 語法錯誤
- 不存在或沒宣告過的變數，如果有變數是透過 include 而來的，在檢查過程中也會去參考其他有 include 到的檔案該變數是否被宣告過
- 變數型態錯誤

我們直接使用以下範例來解釋:

main.cpp:

```
#include <iostream>
#include "module1.h"
using namespace std;

int main() {
    greeting();
    return 0;
}
```

module1.h

```
#ifndef MODULE_1_H
#define MODULE_1_H

void greeting();

#endif
```

module1.cpp

```
#include <iostream>
#include "module1.h"
using namespace std;
```

從上面例子我們可以看到 greeting() 這個 function 在 module1.h 檔案裡面只有 function signature, 但沒有實作的部分，但是我們可以使用 g++ 針對 main.cpp 先進行 compile:

```
g++ -c main.cpp
```

此時是可以成功產出一個 mian.o 這個中間過程檔的。同樣的，我們一樣可以對 module1.cpp 進行 compile:

```
g++ -c module1.cpp
```

此時也可以產出一個中間過程檔 module1.o。最後我們只要把兩個 .o 檔 Link 在一起，就可以產出一個能運行的 exectuable file:

```
g++ -o main main.o module1.o
```

從上面這個例子我們也可以看出，有一些東西即使沒把實作寫出來，每個獨立的 .cpp 檔都應該要可以被 compile 成 .o 檔，因為在 coimplie 成 .o 檔的過程中要的只有定義而已。從這裡我們也可以了解一件事，就是在 Link 的過程中兩個 .o 檔才有了互動，此時也會有更進一步的檢查。而這些繁瑣的步驟，就是 IDE 幫我們處理掉的部分。

在對 compile 有所了解以後，我們就可以來探討更複雜的問題，extern 和 static 的用途是什麼。

extern

現在我們知道每個 .cpp 檔都可以獨自被 compile，假如今天我們有一個變數要在多個檔案之間共用，該怎麼處理呢? 這就是 extern 的作用。extern 告訴 compiler 這個變數

main.cpp:

```
#include <iostream>
#include "module1.h"
using namespace std;

int main() {
    greeting();
    cout << "In main, a = " << a << endl;
    a = 0;
    cout << "In main, a = " << a << endl;
    return 0;
}
```

module1.h

```
#ifndef MODULE_1_H
#define MODULE_1_H

extern int a;
void greeting();

#endif
```

module1.cpp

```
#include <iostream>
#include "module1.h"
using namespace std;

int a = 1;
void greeting() {
    cout << "Hello World!" << endl;
    cout << "In greeting, a = " << a << endl;
}
```

我們增加了一個變數 a，在 module1.h 檔中有 extern int a，我們很清楚的告訴了會 include module1.h 的人，這個模組裡面有一個 int a 變數可以用，但是這個變數的宣告並不在此，而是有某個 include 了 module1.h 的人會去做宣告，這個工作在這裡我

```
Hello World!
In greeting, a = 1
In main, a = 1
In main, a = 0
Hello World!
In greeting, a = 0
```

同時我們可以看到在 main 裡面看到的 a 跟在 module1.cpp 裡面看到的 a 是同一個，因此在 main 裡面修改 a 的值，module1 裡面拿到的 a 也就改變了。

static

看完了 extern，接下來就要來解釋 static 了。相比之下 extern 算是很好理解的了，static 則比較混亂一點。static 之所以混亂，是因為他出現在不同地方，他的意義就會不同，也就是說 static 會被 overload，但每個單獨的定義其實也都很好了解。在 C 裡面因為沒有 class，所以 static 只會有兩種定義，而在 C++ 中因為多了 class，所以會再多兩種定義。static 的意義就是 “被修飾的東西，會從程式一開始執行就存在，且不會因為離開 scope 就消失，會一直存在到程式結束”。 static 出現在哪裏及用什麼定義如下:

- static 出現在 variable 之前，且該 variable 宣告在某個 function 之中 (C/C++)
- static 出現在 variable 之前，且該 variable 並不是宣告在某個 function 中 (C/C++)
- static 出現在 class 的 member variable 之前 (C++ only)
- static 出現在 class 的 member function 之前 (C++ only)

大致也就是這四種定義，因此看到或想有剛好的應用情境時，可以根據上面的列表來區分。

static 出現在 **variable** 之前，且該 **variable** 宣告在某個 **function** 之中

這個算是非常好理解的，一般我們寫 funcion 時，裡面宣告的變數在 funcion 結束之時也會跟著消失。但如果我們在某個變數之前加上 static，該變數就不會因為 function 結束而消失。最經典的例子大概就是要計算這個 function 被呼叫幾次:

```
        ++counter;
        cout << "Greeting function has been called "
             << counter << "times" << endl;
    }
```

每次呼叫 greeting 時，counter 就會加一，且 greeting 結束時 counter 還存在，因此可以用於計算 greeting 到底被呼叫幾次。

static 出現在 variable 之前，且該 variable 並不是宣告在某個 function 中

在我們解釋 extern 的範例中，我們會遇到變數在不同檔案中要共用，只要 include 某個檔案以後，就可以使用其中的變數。但這會導致一個比較麻煩的問題，假設今天有兩個檔案 a.cpp 和 b.cpp (應該不需要用 Alice.cpp 跟 Bob.cpp 來解釋吧 XD)，各自有各自的 .h 檔。a.cpp include 了 b.h，但 a.cpp 跟 b.cpp 裡面各自宣告了一個在 function 外 的 bool debug 的變數，分別是用來啟動或關閉 a 和 b 的 debug 功能。兩個 compile 後的 .o 檔在 link 的過程中就會因為名字相同而產生衝突的錯誤。對不存在在 funciton 外的變數來說，基本上都是 global variable，static 的用意就是要讓這樣的 global 只限定在該檔案內，而不是整個程式中。因此，如果我們把 a.cpp 和 b.cpp 中的 debug 在宣告時前面都加上 static，這樣 compiler 在處理前期替換掉的名字就會不同，因此 link 的過程中也不會有名字衝突的問題。

因為覺得上面只用這個例子解釋並不好理解，所以我補上一個例子。首先我們有:

- main.cpp
- a.h 和 a.cpp
- b.h 和 b.cpp

main.cpp

```
#include "a.h"
#include "b.h"

int main() {
    show_debug_in_a();
    show_debug_in_b();
    return 0;
}
```

```
#ifndef _A_H_
#define _A_H_

void show_debug_in_a();

#endif
```

a.cpp

```
#include <iostream>
using namespace std;

static bool debug = true;

void show_debug_in_a() {
    cout << debug << endl;
}
```

b.h

```
#ifndef _B_H_
#define _B_H_

void show_debug_in_b();

#endif
```

b.cpp

```
#include <iostream>
using namespace std;

static bool debug = false;

void show_debug_in_b() {
    cout << debug << endl;
}
```

```
$ g++ -c main.cpp // 產出 main.o
$ g++ -c a.cpp // 產出 a.o
$ g++ -c b.cpp // 產出 b.o
$ g++ -o main main.o a.o b.o // 把 main.o, a.o, b.o link 成 main
```

其中 main.cpp include a.h 和 b.h，當中所有的實作以及 debug 這個 static 的變數宣告都寫在 a.cpp 和 b.cpp 中，如此一來 a和 b 中的 debug 就不是同一個，也不會互相污染到。為什麼要這麼麻煩不直接 include a.cpp 和 b.cpp 呢？這是因為如果 include 某個檔案後，裡面所有的東西對於 include 的人就通通都看得到了，也因此如果直接 include a.cpp 和 b.cpp，對於 main.cpp 來說，他就看到有兩個地方都宣告了 debug 這個變數，所以 compile 的時候直接就會被檢查到而失敗。

那你可能會問，咦，我只能 include .h 檔，又沒有 include .cpp 檔，那 main.cpp 不就還是沒看到兩個 cpp 裡的 debug，那為什麼還要用 static 呢? 這是因為如果沒寫 static，則在 -c 這個 compile 的過程中產生的 .o 檔裡面會帶有這個檔案轉化後的資訊，而所有不在 function 等等 scope 的變數通通會被視為是 global variable。也因此，即便 main.cpp 在 compile 的最終只看到 show_debug_in_a 和 show_debug_in_b 兩個變數，但在把 main.o, a.o 和 b.o link 的過程中，compiler 看到 a.o 和 b.o 裡面都有一個 debug 的變數，就會報錯了。

static 出現在 class 的 member variable 之前

static 出現在 class 的 member variable 的意思是該 variable 並不屬於某個 instance，他屬於這個 class，所有以此 class 生成出來的 instance 都共用這個 variable。最經典的範例就是用來計數這個 class 總共生成了多少個 instance。

static 出現在 class 的 member function 之前

static 出現在 member function 的意思是該 function 並不屬於某個 instance，他也屬於這個 class，所有以此 class 生成出來的 instance 都共用這個 function。也因此，即便我們沒有產生 instance 出來，我們也隨時可以取用這個 function。

C Programming