

# C++ / Using the GCC Compiler (II)

資訊之芽語法班 2015

# 本週預備知識與主題

- week 7 的 compiler 介紹
  - 我們要自己下指令編譯，不是全部依賴 Dev:C++ 了！
- 基礎 `struct` 與函數撰寫
  - 把檔案分開會講到 `struct` 跟函數

# 為什麼要學怎麼用 compiler ? - (I)

- Dev:C++ 的 F11 到底做了什麼？
  - compiler options、linker options 是啥？



- 為什麼一個專案裡不能有很多個 `main` 函數？
- 錯誤訊息到底是什麼意思？
  - `Undefined reference to ...`
  - `Multiple definition of ...`

## 為什麼要學怎麼用 compiler ? - (II)

- 所以小遊戲那一堆設定在做什麼？
- 把「函式庫」分開來有什麼好處？
- 還記得 bmp 作業中的 bmp\_hdrl.h 嗎？那是什麼？

# 複習：使用命令列模式

- 打開命令列模式（以下方法二選一）
  - 開始 → 所有程式 → 附屬應用程式 → 命令提示字元
  - 開始 → 搜尋 cmd

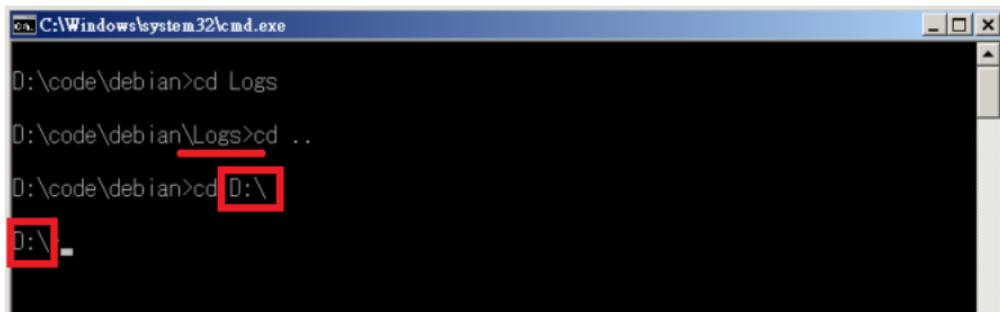
## 複習：使用命令列模式

- 操作方法：輸入文字，輸入完按 enter，他會顯示輸出
- 紅線框起來的是「當前工作目錄」，代表你現在在哪個資料夾下。任何檔案操作都是相對於此路徑。

# 複習：使用命令列模式

- 指令：`cd`，切換當前工作目錄
  - `..` 是特殊名稱，代表相對於當前工作目錄的上一層目錄

```
> cd Logs  
> cd ..  
> cd D:\
```



A screenshot of a Windows Command Prompt window titled "C:\Windows\system32\cmd.exe". The window shows the following command history:

```
C:\Windows\system32\cmd.exe  
D:\code\debian>cd Logs  
D:\code\debian\Logs>cd ..  
D:\code\debian>cd D:\  
D:\>
```

The command `cd D:\` is highlighted with a red rectangle. The output `D:\>` is also highlighted with a red rectangle.

# 複習：使用命令列模式

- 指令：`X:`，切換當前磁碟機

```
> C:  
> D:
```



A screenshot of a Windows Command Prompt window. The title bar reads "C:\Windows\system32\cmd.exe". The command line shows the user switching between drives D and C, and then switching back to drive D. The current directory is shown as "C:\Users\suhorng>D:".

```
D:\>C:  
C:\Users\suhorng>D:  
D:\>
```

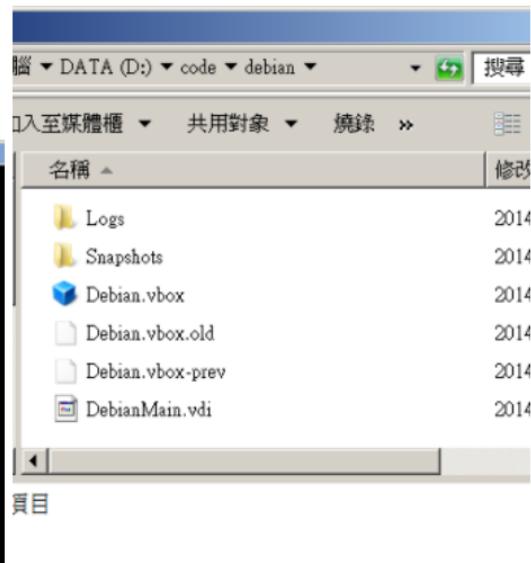
# 複習：使用命令列模式

- 指令：`dir`，列出目錄

```
> dir  
> dir D:\code\
```

C:\Windows\system32\cmd.exe

```
D:\code\debian>dir  
磁碟區 D 中的磁碟是 DATA  
磁碟區序號: 0C6A-816C  
  
D:\code\debian 的目錄  
  
2014/03/20 下午 02:31 <DIR> .  
2014/03/20 下午 02:31 <DIR> ..  
2014/03/20 下午 02:31 0,990 Debian.vbox  
2014/03/20 下午 02:31 9,843 Debian.vbox-prev  
2014/03/17 上午 12:51 9,861 Debian.vbox.old  
2014/03/20 下午 02:31 7,448,141,824 DebianMain.vdi  
2014/03/20 下午 02:12 <DIR> Logs  
2014/03/20 下午 02:31 <DIR> Snapshots  
4 個檔案 7,448,171,518 位元組  
4 個目錄 10,776,485,888 位元組可用  
  
D:\code\debian>
```



# 複習：簡單使用 compiler

- 命令：`g++ -o test.exe test.cpp`
  - `-o OUTPUT_FILE_NAME` 一整串指定輸出檔名
  - `test.cpp` 為輸入的 `.cpp` 原始檔

# 複習：簡單使用 compiler

- Q：所以 `g++ XXXX.cpp` 要打在哪裡？
  - A：就是打在剛才的黑框框裡面！
- Q：有什麼指令可以用？
  - A：可以輸入 `HELP<ENTER>` 看看有什麼指令
- Q：要怎麼看一個指令怎麼用（例如 `dir`）？
  - A：輸入 "指令" `/?`，例如 `dir /?`
  - 對於 `g++`，要輸入 `--help`  
(i.e. `g++ --help`) 來看說明

# 如何使用 Compiler

- 其他常見的 flag

- `-O1 / -O2 / -O3` : 設定程式碼優化層級
- `-Wall / -Wshadow / -Wextra` : 設定編譯器警告
- `-pedantic` : 設定編譯器嚴格程度
- `-std=c++03 / -std=c++11` : 指定 C++ 標準版本
- 範例：

```
g++ -o test.exe test.cpp -O2 -Wall -std=c++03
```

- 本週主題：`-C` 選項，只編譯不連結

# C++ 的編譯模型 (Compilation Model)

- `g++` 是 GCC compiler toolchain 中的 driver program，會呼叫真正編譯、連結的 `cc1plus.exe` 與 `ld.exe`。
- 原始碼經過編譯後會成為 object file，最後由 linker 把（可能數個）object file 連結產生執行檔

# C++ 的編譯模型 (Compilation Model)

考慮以下程式碼：

```
#include <iostream>
#define STRING "Hello"
int main(void)
{
    const char *ptr = NULL;
    ptr = STRING;

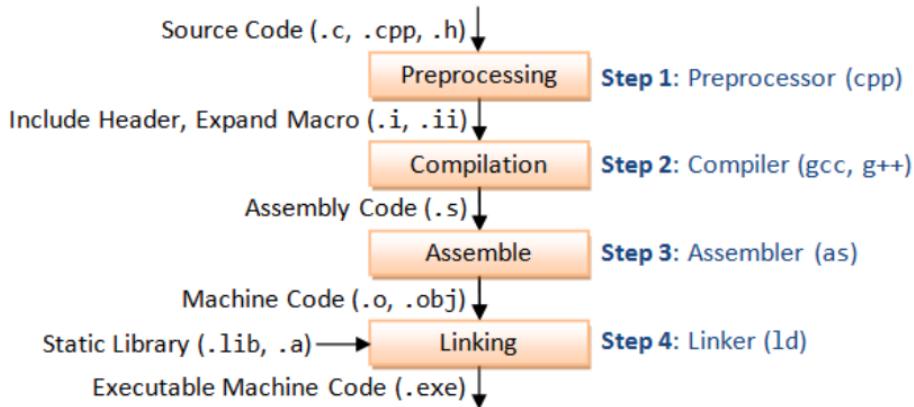
    // Print the Hello string
    std::cout << ptr << '\n';
    return 0;
}
```

```
g++ -fPIC -c hello.c
```

```
g++ hello.o -o hello
```

這些是什麼？

# C++ 的編譯模型 (Compilation Model)



# C++ 的編譯模型 (Compilation Model)

- The preprocessing step

把引入的標頭檔貼到你寫的程式碼，替換掉 define，把註解拿掉...

```
# 1 "hello.cpp"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "hello.cpp"
# 1 "/usr/include/c++/4.8/iostream" 1 3
# 36 "/usr/include/c++/4.8/iostream" 3

# 37 "/usr/include/c++/4.8/iostream" 3

...
# 2 "hello.cpp" 2

int main(){
    const char *ptr = __null;
    ptr = "Hello";

    std::cout << ptr << '\n';
    return 0;
}
```

# C++ 的編譯模型 (Compilation Model)

- The compilation step

將程式碼編譯成組合語言

```
.file  "hello.cpp"
.local _ZStL8__ioinit
.comm _ZStL8__ioinit,1,1
.section .rodata
.LC0:
.string "Hello"
.text
.globl main
.type  main, @function
main:
...
```

# C++ 的編譯模型 (Compilation Model)

- The assembly step

把組合語言轉換成機器碼

```
L .text$| 0` .data@0|.bss|0|.rdat (@0@/40@0@/158D@a0@U$|D|helloGCC: (GNU) 4.8.1
]!B

.file gtest.cpp_main .text!.data.bss.rdata$8__main _puts ..rdata$zzz.eh_frame,rdata$zzz
```

看不懂的亂碼(事實上是二進位的數字)

# C++ 的編譯模型 (Compilation Model)

- The linking step

把未定義的 symbol 連結到實作好的 library

得到執行檔 hello !

# C++ 的編譯模型 (Compilation Model)

- 動機：每次都從原始碼開始處理，很慢，很浪費時間
  - `cstdio`、`cstdlib`、`cstring`、……等內容基本不會變，不需要每次重複處理
  - 有些底層與作業系統接軌的部份沒辦法用 C/C++ 撰寫
  - 不同的程式語言之間怎麼互相連結？
- 作法：把原始碼利用 `-c` flag 編譯成 `.o` 檔 (object file)，然後最後再把一堆 `.o` 檔合成一個執行檔

```
g++ -c -o source1.o source1.cpp  
g++ -c -o source2.o source2.cpp  
...  
g++ -o app.exe source1.o source2.o ...
```

# C++ 的編譯模型 (Compilation Model)

- `.h`：在原始碼檔案裡面只需要有函數（或變數）的宣告/**定義**，不需要有它的實作
- `.cpp`：函式實作部份
- `.o`：是某種中間格式，通常存的是不完整的機器碼
  - 沒有定義的函式位址會留空，等待連結器填入。所以**連結時必須所有的函數都有實作**。
  - 會留有符號 (symbol) 的資訊，例如「`add1` 函數的定義是 `int add1(int);`」，在某某地方被使用」
- ⇒
  - 函數的使用與實作可以在不同檔案

# 編譯：宣告/原型 (prototype)

- 呼叫一個函數的時候至少要有什麼資訊？
- 使用一個變數時要知道什麼？
- forward declaration !
  - 只要型別就足夠了（為何？）

```
int add_floor(int, double);  
extern int data;
```

- 但是如以下， `bmp_t` 是什麼？

```
void output_bmp(const char* filename, bmp_t* bmp);
```

- 型別也可以有 forward declaration : `struct bmp_t;`，這樣我們就知道 `bmp_t` 是個型別，而可以有 `bmp_t*` 指標這個型別

# 連結：定義

- 要真正的實作一個函數

```
int add_floor(int a, double b) {  
    return a + static_cast<int>(b);  
}
```

- 真正的宣告變數：沒有 `extern`

```
int data = -1;
```

- 要有 `struct` 的定義

```
struct bmp_t {  
    int height;  
    int width;  
    ...  
};
```

\* 思考：linked-list 的定義中為何不可能把 `node \*next` 改成 `node next`？

# 函數：宣告與定義

- 宣告 (declaration、prototype)：沒有大括弧、以分號結尾的

```
RETURN_TYPE function_name(PARAM_TYPE1, ..., PARAM_TYPEN);
```

- 定義 (definition、implementation)：有大括弧的

```
RETURN_TYPE function_name(PARAM_TYPE1 PARAM1, ..., PARAM_TYPEN PARAMN) {  
    ... code ...  
}
```

- 粟子

```
int print_hello(int, int);  
int main() {  
    int a = print_hello(3, 4); // all you need is its type!  
}  
  
int print_hello(int a, int b) {  
    std::cout << "hello";  
    return a + b;  
}
```

# 函數：宣告與定義

- 練習：假設現在有兩個函數 even、odd，分別會遞迴地呼叫對方。請修改增加新的程式讓它可以編譯。

```
/* VVV add your code here VVV */  
  
/* ^^^ add your code here ^^^ */  
  
bool is_even(int n) {  
    if (n == 0)  
        return true;  
    else  
        return is_odd(n-1);  
}  
  
bool is_odd(int n) {  
    if (n == 0)  
        return false;  
    else  
        return is_even(n-1);  
}
```

# 變數：宣告與定義

- 宣告：加 `extern` 在型別前；定義：如往常

```
extern TYPE VARIABLE;
```

- 不可以又有 `extern` 又有初始化，這會使得 `extern` 被忽略

```
extern int data = -1; // extern會沒有效
```

- 說明：同樣地，加了 `extern` 的話編譯器就會把它當成「宣告」而非「定義」，因此會留一個 symbol 在檔案中，而不會真的把內容寫出來。
- 問題：如果在一個檔案中宣告 `extern int data;` 另一個檔案中寫 `char data = 2;` 會如何？

# 小節

- 我們可以在一個 .cpp 檔中使用在另外一個 .cpp 檔案裡的函式
  - 在其中一個檔案裡面寫函式的真正的程式
  - 在其他檔案中只要宣告函式的型別即可
- 使用 .o 檔可以避免每次都從原始碼開始處理
- 只要有型別的宣告，就可以在其他函式的宣告中**出現該型別**（在說 struct A\*）
- 要真的使用 struct 還是需要有該 struct 的完整定義（才能知道有什麼欄位、東西多大）

# header 檔慣例寫法

- 要讓在一個 cpp 中使用另外一個 cpp 中的函式，要宣告函式的型別
  - 每個 .cpp 檔都要這樣做的話，很不方便

\* 可以把相關的宣告都寫在\*\*標頭檔\*\* (header file) 裡面，每次使用時再引入標頭檔即可  
\* 標頭檔中通常放著各個函式的宣告、陣列的定義等，以及（若有的話）`extern` 的變數

- 栗子（註：這個例子並不完整）

```
// ex.h
struct output_buffer {
    int buffer_pos;
    char buffer[1024];
};
size_t print_string(output_buffer*, const char* );
size_t print_int(output_buffer*, int);
size_t print_double(output_buffer*, double);
```

# header 檔慣例寫法

- `#include <FILENAME>` / `#include "FILENAME"`
  - 慣例上 `<filename>` 放系統的標頭檔、`"filename"` 放使用者的標頭檔
  - `#include` 是前置處理器 (preprocessor) 的指令，代表把檔案複製貼上
  - 呼叫 `g++` 時可以用 `-E` 來觀看前置處理後的結果

```
// A.h
void A();

// D.cpp
#include "A.h"

// g++ -E D.cpp
# 1 "D.cpp"
# 1 "<command-line>"
# 1 "D.cpp"
# 1 "A.h" 1
void A();
# 2 "D.cpp" 2
```

# header 檔慣例寫法

- **include guard**

- 如果 `B.h` 用到 `A.h`，`C.h` 用到 `A.h`，且 `D.cpp` 同時用到 `B.h` 和 `B.h` 的話……

\* `A.h` 可能會被 `include` 兩次！即使函數宣告不會出問題，但 `struct` 不能重新定義（這跟 C++ 的 type erasure 有關）

- 用以下的慣例寫法

- `#pragma once`

- ```
#ifndef _FILENAME_H
#define _FILENAME_H

...
#endif
```

# header 檔慣例寫法

- 粟子 (Cont'd)

```
#ifndef _EX_H
#define _EX_H

// ex.h

struct output_buffer {
    int buffer_pos;
    char buffer[1024];
};

size_t print_string(output_buffer*, const char*);
size_t print_int(output_buffer*, int);
size_t print_double(output_buffer*, double);

#endif
```

# 設計函式庫與介面

- 抽象的型別 `bmp_t`
- 相關操作 `input_bmp`、`output_bmp`、`draw_line`…

```
struct bmp_t {
    // note: the BMP stores the (R, G, B) triples in B - G - R order
    int height;
    int width;
    uint8_t pixels[BMP_MAX_SIZE*3];
};

// pixel operation
void set_pixel(bmp_t *bmp, int y, int x, uint8_t r, uint8_t g, uint8_t b);
void get_pixel(bmp_t *bmp, int y, int x, uint8_t*r, uint8_t*g, uint8_t*b);

// initialize a bmp
bool create_bmp(int height, int width, bmp_t *bmp); /* @return: success */

// input and output
bool input_bmp(const char *filename, bmp_t *bmp); /* @return: success */
bool output_bmp(const char *filename, bmp_t *bmp); /* @return: success */

// Draw method
void fill_rect(bmp_t *bmp, uint8_t r, uint8_t g, uint8_t b,
               int x1, int x2, int y1, int y2);

void draw_line(bmp_t *bmp, uint8_t r, uint8_t g, uint8_t b,
               int x1, int x2, int y1, int y2);
```

# 在不同的 compilation unit 之間隱藏名字

- 在全域的函數或變數中加上 `static` 關鍵字，可以讓此函數（變數）僅於當前檔案可見。

```
static int local_data = 123; //就算不同檔案中有 `local_data` 也沒關係
```

```
static void local_helper(int* a, int* b) {  
    int c = *a;  
    *a = *b;  
    *b = c;  
}
```

- `static` 有很多個意思，寫在不同的地方有不同的效果！  
(例如靜態變數 / 靜態成員函數)

```
int get_count() {  
    static int cnt = 0; // cnt 是個靜態變數  
    cnt = cnt+1;  
    return cnt;  
}
```

# 這份投影片沒包含的部份

- Mac 的 terminal
  - Mac OS / Linux 發行版等這一類系統都是比較接近 POSIX 標準的，所以很多東西會長得很像。Mac OS 跟 BSD 的血緣比較接近，要找 Mac 的 terminal 的教學也可以看 Linux 中常見 shell 的。
- 其他作業系統的執行檔
  - e.g. Linux distributions 的 ELF 格式
- Makefile，自動化編譯過程 + 依照檔案的相依性檢查修改時間，可以不用每次都打指令

## 延伸閱讀

- [gcc-compilation-process](#)
- [GCC and Make](#)

## 延伸閱讀

- [gcc-compilation-process](#)
- [GCC and Make](#)