

C/C++語言 觀念題目整理



作者簡歷

學 經 歷： 國立科技大學 機械相關

國立科技大學 機械控制組

修課狀況： 大學(C++、LabVIEW、ASM、C#)

碩士(Python、Git、MCU)

作者撰寫此文件時 24 歲，在高中就讀機械科時發現自己對於傳統機械工程如：機構設計、機械材料等等...興趣缺缺。因而在大學時選擇就讀自動化工程，並在大二時擔任社長職務；專題部分則剛好參與了老師的計劃案，主要內容為使用 C#作為開發工具進行工具機的熱誤差補償，後來認識到自己實力不足且高階工作通常需要碩士學歷的緣故萌生了進修的念頭，又因大學成績不佳(系上 PR 50.45)；選擇用考試的方式入學並進入了機電整合所就讀，研究所題目主要是建立一套機器人系統，包含了機構設計、實驗平台以及機器人控制等等...研究中主要使用的工具則有 Solidworks(機構設計與平台設計)、Python(控制介面與通訊)、MicroPython(機器人控制)以及 Git(程式備份與版本控制)。

前言

今年碩士剛畢業的我，找工作的過程可以說是跌跌撞撞，即使在碩士期間就已經確定自己會朝韌體工程師的方向找工作，卻因個人能力的不足以及兵役問題總是無法找到理想的職缺；面試過四間公司以及拜讀網路上各路大神的經驗分享後，讓我對於自己的目標有了更清晰的藍圖，也更加明白了自己想走的路究竟需要哪些技能；因此萌生了在當兵前整理這些資訊的想法。

對於韌體工程師來說，C 語言可說是技能樹上必點的技能，而作者本身在求學過程中未曾接觸過該領域知識，因而在畢業後花了不少時間研究該領域，過程中主要參考了、Dcard 論壇網友分享的資源 ChatGPT 以及 Youtube 頻道 Feis Studio，本文件主要以 C 語言之問題為主，偶爾會提到 C++ 相關問題；倘若想做 C 語言的練習，可使用線上 C 語言編譯器 OnlineGDB(可參閱補充各式題目提供之網址)。除此之外，作者預計會撰寫其它韌體相關知識整理，包含了嵌入式系統、作業系統、資料結構等等…撰寫此文件除了希望幫助自己在找韌體工程師職缺時更加得心應手外也期盼能鍛鍊自己在往後撰寫相關文件的能力；順便填補當兵前的空閒時間，期盼此文件能讓閱讀的各位有所收穫也期盼給予碩班實驗室的後輩們一些幫助，並在往後求職的路上一帆風順。

2023/10/28 撰

誌謝

本文件獻給所有正在找軟體工程師職缺的夥伴，而在這裡我必須得先感謝我的家人們願意給我大量的資源讓我能夠先完成碩士學位，並在等待兵役的這段期間無條件支持在家自修的開銷，同時也感謝碩士班的學長們；在我剛找工作感到迷茫時為我解答了不少疑惑，也留下了寶貴的面試經驗供我參考。

最後想感謝網路上願意分享面試經驗以及整理面試題目的所有人，有了他們的分享，才得以讓我完成這些文件的撰寫。



目錄

誌謝	iii
範例目錄	vii
圖目錄	ix
第一章 指標	1
1.1 指標[1].....	1
1.1.1 基礎指標判讀	1
1.1.2 指標與其他關鍵字混用	2
1.1.3 指標之用法[2].....	2
1.2 傳值	3
1.3 傳址	4
1.4 傳參考.....	5
第二章 變數範圍和生命週期[1].....	6
2.1 local 變數	6
2.2 static 變數	6
2.3 global 變數.....	6
2.3.1 static 變數 v.s global 變數	6
2.3.2 記憶體的配置	8
第三章 const 與 volatile[1].....	11
3.1 const.....	11
3.1.1 const v.s #define.....	11
3.2 volatile.....	13
3.2.1 const 和 volatile 合用	13
第四章 預處理器與編譯器指令[1].....	16

4.1 巨集	17
4.2 #error 指令	18
4.3 引入防護和條件編譯	19
4.3.1 引入防護	19
4.4 inline	21
第五章 邏輯運算子[1]	22
5.1 C 語言中的邏輯運算子	22
第六章 記憶體與字串複製[1]	25
6.1 記憶體複製(Memory Copy)	25
6.2 字串複製(String Copy)	25
第七章 延伸性資料型態[1]	27
7.1 結構	27
7.2 重新定義型態名稱	28
7.3 共用結構	29
7.4 列舉	30
7.5 總結	30
第八章 資料存儲方式	31
8.1 Little – Endian	31
8.2 Big – Endian	32
8.3 C 語言判斷位元組順序[5]	33
第九章 未定義行為	34
9.1 未定義行為[6]	34
第十章 補充	37
10.1 運算子優先權	37

10.2 各式題目	38
參考文獻	48



範例目錄

範例 1.1 指標判讀	1
範例 1.2 指標與其他關鍵字混用之判讀	2
範例 1.3 傳值(Call by value)	3
範例 1.4 傳址(Call by address)	4
範例 1.5 傳參考(Call by reference)	5
範例 2.1 配置練習	9
範例 2.2 static 練習	9
範例 3.1 const 範例	12
範例 3.2 #define 範例	12
範例 4.1 巨集範例	17
範例 4.2 巨集錯誤示範	17
範例 4.3 #error 範例	18
範例 4.4 引入防護範例	19
範例 4.5 條件編譯的其它應用	20
範例 5.1 邏輯運算子的基本運算	22
範例 5.2 Mask 方法作 bitwise 操作	23
範例 5.3 位元左移練習	24
範例 6.1 記憶體複製與字串複製練習	26
範例 7.1 struct 練習	27
範例 7.2 typedef 練習	28
範例 7.3 union 練習	29
範例 7.4 enum 練習	30

範例 8.1 判斷位元組順序	33
----------------------	----



圖目錄

圖 2.1 Process 在記憶體中的配置圖 [重要]	8
圖 8.1 Little – Endian 示意圖[5].....	31
圖 8.2 Big – Endian 示意圖[5].....	32
圖 9.1 未定義行為編譯器警告訊息.....	34



第一章 指標

1.1 指標[1]

指標(pointer)：一個指向某個儲存位址的變數，語法為

<code>int *ptr = &var;</code>	(1.1)
-----------------------------------	-------

其中

1. &：取變數位址
2. *：表示為指標變數

也可用於函數變為函式指標(function pointer)，語法為：

<code>void (*fptr)(type_a, type_b) = &fucn;</code>	(1.2)
--	-------

1.1.1 基礎指標判讀

指標之判讀原則為「由右至左」，例如：

範例 1.1 指標判讀

<code>int *a;</code> // 一個指向整數的指標
<code>int **a;</code> // 一個指向指標的指標，它指向的指標是指向一個整型數
<code>int a[10];</code> // 一個有 10 個整數型的陣列
<code>int *a[10];</code> // 一個有 10 個指標的陣列，該指標是指向一個整數型的
<code>int (*a)[10];</code> // 一個指向有 10 個整數型陣列的指標
<code>int (*a)(int);</code> // 一個指向函數的指標，該函數有一個整數型參數並返回一個整數
<code>int (*a[10])(int);</code> // 一個有 10 個指標的陣列，該指標指向一個函數，該函數有一個整數型參數並返回一個整數

1.1.2 指標與其他關鍵字混用

一樣由右邊讀至左邊，例如

範例 1.2 指標與其他關鍵字混用之判讀

```
const int * foo; // 一個 pointer，指向 const int 變數。  
int const * foo; // 一個 pointer，指向 const int 變數。  
int * const foo; // 一個 const pointer，指向 int 變數。  
int const * const foo; // 一個 const pointer，指向 const int 變數。
```

關鍵字 volatile 等等判讀方式亦相同(有關 volatile 之相關介紹請參閱第五章)。

1.1.3 指標之用法[2]

A. 一個指向整數的指標 (A pointer to an integer)

```
int *a;
```

B. 一個指向指標的指標，它指向的指標是指向一個整型數(A pointer to a pointer to an integer)

```
int **a;
```

C. 一個有 10 個指標的陣列，該指標是指向一個整數型的(An array of 10 pointers to integers)

```
int a[10]; //該指標僅有指向 a[0]位置
```

D. 一個指向有 10 個整數型陣列的指標 (A pointer to an array of 10 integers)

```
int (*a)[10];
```

- E. 一個指向函數的指標，該函數有一個整數型參數並返回一個整數 (A pointer to a function that takes an integer as an argument and returns an integer)

```
int (*a)(int);
```

- F. 一個有 10 個指標的陣列，該指標指向一個函數，該函數有一個整數型參數並返回一個整數 (An array of ten pointers to functions that take an integer argument and return an integer)

```
int (*a[10])(int);
```

1.2 傳值

傳值(Call by value)是最常見的函式寫法，呼叫者和被呼叫者的變數各自佔有記憶體，將參數複製再傳給函式。

範例 1.3 傳值(Call by value)

```
void changeValue(int x) {  
    x = 10;  
}  
  
int main() {  
    int num = 5;  
    changeValue(num);  
    printf("num = %d\n", num); // 這裡會輸出 "num = 5"，因為參數按值  
    // 傳遞，函數中的修改不影響原始值  
    return 0;  
}
```

1.3 傳址

C 語言之父 K&R 曾表示 C 語言僅有傳值作法[3]，而會有傳址(Call by address)之說出現僅僅是為了方便教學，其指的是對指標變數進行操作的傳值，具體的執行效果則和傳參考(Call by reference)相同。

範例 1.4 傳址(Call by address)

```
void changeValue(int *x) {  
    *x = 10;  
}  
  
int main() {  
    int num = 5;  
    changeValue(&num);  
    printf("num = %d\n"); // 這裡會輸出 "num = 10"，因為參數按地址  
                           傳遞，函數中的修改影響原始值  
    return 0;  
}
```


1.4 傳參考

傳參考(Call by reference)是指呼叫者的變數使用相同的記憶體位址，因此在被呼叫函式中改變變數時，變動結果會保留。僅 C++ 才有該寫法，寫法為

<code>type func(type &var)[1]</code>	(1.3)
--	-------

範例 1.5 傳參考(Call by reference)

```
void changeValue(int &x) {  
    x = 10;  
}  
  
int main() {  
    int num = 5;  
    changeValue(num); // 使用引用傳遞參數  
    std::cout << "num = " << num << std::endl; // 這裡會輸出 "num = 10",  
    因為參數按引用傳遞，函數中的修改影響原始值  
    return 0;  
}
```

第二章 變數範圍和生命週期[1]

本章節主要介紹三種變數類型，local、static 以及 global，並針對各個變數進行範圍(Scope)以及生命週期(Life time)作講解。

[註] 關鍵字: static

2.1 local 變數

local 變數之生命週期僅存在函式內，存放位置則在 **stack** 或 **heap** 記憶體中。範圍則僅限於宣告它們的函式，而不能在該函式外宣告。

2.2 static 變數

static 變數之生命週期和程式一樣長，而範圍(Scope)則維持不變，即在宣告之函式之外是無法存取 static 變數的。

2.3 global 變數

global 變數為所有區段皆可使用的變數，其生命週期與程式一樣長。

2.3.1 static 變數 v.s global 變數

除了變數的範圍不同外，static 變數僅有宣告的函式可以使用，而 global 變數則可使用 **extern** 關鍵字修飾，即可在其他檔案以.h 標頭檔方式使用該變數(也就是 **internal linkage** 和 **external linkage** 的不同)。

2.3.1.1 內部連結與外部連結

內部連結(Internal Linkage):

具有內部連結的變數或函數僅在其所在的源檔案中可見；使用 `static` 關鍵字聲明，例如 `static int x;` 或 `static void myFunction();`。這樣的變數或函數僅在宣告它們的源檔案中可訪問。這種類型的變數或函數通常用於實現模組內的私有數據或函數，以防止其他源檔案中的名稱衝突。

外部連結(External Linkage):

具有外部連結的變數或函數可以在多個源檔案中共享和訪問。大多數全域變數和函數都具有外部連結。當宣告一個全域變數或函數時，它將具有外部連結，除非使用 `static` 關鍵字將其設置為內部連結。外部連結的變數或函數可以在不同的源檔案之間共享，並且可以在連結階段（將多個源檔案組合為一個可執行文件的階段）中連接在一起。

2.3.1.2 extern 關鍵字

在 C 語言中，`extern` 關鍵字通常不用於修飾全域變數（`global variables`），因為全域變數本身已經具有外部連結（`external linkage`），這意味著它們可以在不同的源檔案之間共享和訪問；`extern` 關鍵字通常用於聲明變數，以表示該變數是在其他地方定義的，並且您打算在當前源檔案中使用它。

2.3.2 記憶體的配置

1. Stack: 存放函數的參數、區域變數等，由空間配置系統自行產生與回收。
(稱做 Stack 的原因是其配置遵守(Last-In, First-Out, LIFO)。
2. Heap: 一般由程式設計師分配是放，執行時才會知道配置大小，如 malloc/new 和 free/delete。([註 1]其資料結構不是 DS 中的 heap 而是 link-list)
3. Global: 包含 BSS(未初始化的靜態變數)，data section(全域變數、靜態變數)和 text/code(常數字元)

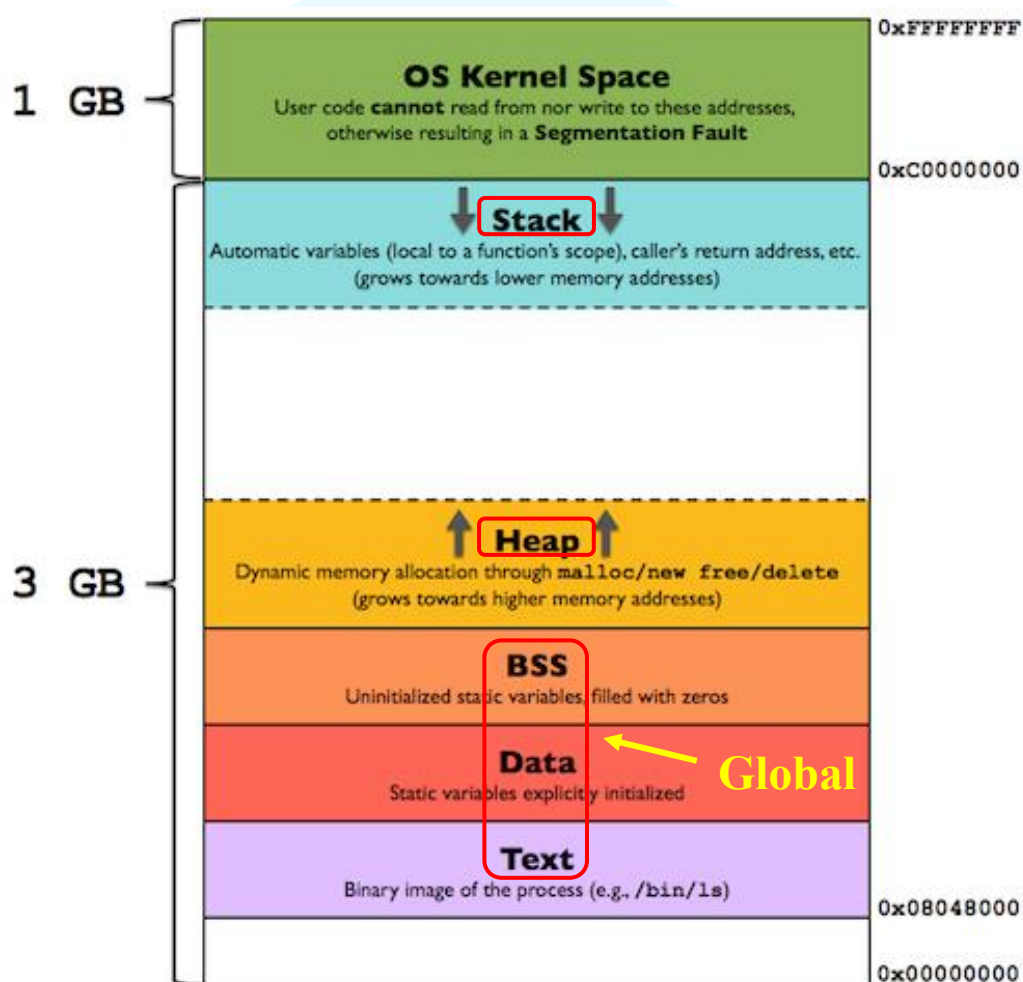


圖 2.1 Process 在記憶體中的配置圖 [重要]

範例 2.1 配置練習

```
int a=0;    //global 初始化區

char *p1;   //global 未初始化區

int main(){

    int b;           // stack

    char s[]="abc";  // stack

    char *p2;        // stack

    char *p3="123456"; // 123456\0 在常量區，p3 在 stack。

    static int c=0;   // global (static) 初始化區

    p1 = (char*)malloc(10);

    p2 = (char*)malloc(20); //分配得來得 10 和 20 位元組的區域在 heap

    strcpy(p1,"123456");

    //123456\0 在常量區，編譯器可能會將它與 p3 中的 123456\0 優化成一個地方。

}
```

範例 2.2 static 練習

```
static int num_a;

// 專屬於整個檔案的全域變數，其他檔案不能存取

void func (int num_b) { // stack 區

    int num_c; // stack 區

    static int num_d;

    //scope 不變，只能在函數 func 內呼叫，但 lifetime 是整支程式執行的時間。

}
```

補充:

在 C 語言中，static 的三種作用[2]：

1. 隱藏功能，利用這一特性可以在不同的檔案中定義同名函式和同名變數，而不必擔心命名衝突。
2. 保持變數內容的持久，儲存在靜態資料區的變數會在程式剛開始執行時就完成初始化，也是唯一的一次初始化。
3. 預設初始化為 0，其實全域性變數也具備這一屬性，因為全域性變數也儲存在靜態資料區。在靜態資料區，記憶體中所有的位元組預設值都是 0x00。

[註 1]:

在資料結構(Data Structure)中，heap 是指一種特定的數據結構，通常用於實現優先佇列(Priority Queue)。這種 heap 是一種二叉數據結構，但此處的”heap”指的是一個動態分配記憶體的區域，通常用於存儲在運行時配置的數據，透過 malloc 或 new 語句分配記憶體，這種 heap 是一個記憶體區域，通常為堆疊(Stack)的對立面。

第三章 const 與 volatile[1]

3.1 const

const 代表了"read-only"，通常表示只可讀取不可寫入的變數，常用來宣告常數。使用 const 有以下好處：

1. 提升程式碼可讀性
2. 使編譯器保護那些不希望被改變的參數
3. 給優化器一些附加的資訊

3.1.1 const v.s #define

1. 編譯器處理方式：define 在預處理階段展開；const 在編譯階段使用。
2. 類型和安全檢查：const 會在編譯階段會執行類型檢查，define 則不會。
3. 存儲方式：define 直接展開不會分配記憶體，const 會在記憶體中分配。

const 常數是真正的變數，具有數據類型，需要分配內存，且在運行時具有類型檢查。#define 宏定義只是文本替換，沒有數據類型，不需要分配內存，且在編譯時進行替換，沒有類型檢查。使用 const 常數通常更安全，因為它們提供了類型安全性和更好的調整支持。使用 #define 宏定義通常更靈活，但容易引發錯誤。

範例 3.1 const 範例

```
const int maxCount = 100; // 宣告一個整數常量  
const double pi = 3.14159265; // 宣告一個雙精度浮點數常量
```

範例 3.2 #define 範例

```
#define MAX_COUNT 100 // 定義一個符號常量  
#define PI 3.14159265 // 定義一個宏定義
```

補充: [2]

1. const int a;
2. int const a;
3. const int *a;
4. int * const a;
5. int const * a const;

答案:

1. a 是一個常數型整數
2. 同(1)
3. 一個指向常數型整數的指標(整數無法修改，但指標可以)
4. 一個指向整數的常數型指標(指標指向的整數可以修改，但指標是
不可以修改的)
5. 一個指向常數型整數的常數型指標(指標指向的整數不可以修改，同時
指標本身也是不可以修改的)

3.2 volatile

由於嵌入式系統常處理 I/O、中斷、即時操作系統 (Real-Time Operating System, RTOS) 相關的問題，因此在嵌入式系統開發中 volatile 尤為重要。被 volatile 修飾的變數代表它可能會被不預期的更新，因此告知編譯器不對它涉及的地方做最佳化，並在每次操作它的時候都讀取該變數實體位址上最新的值，而不是讀取暫存器的值。

volatile 常見的應用：

1. 修飾中斷處理程式中 (ISR) 中可能被修改的全域變數。
2. 修飾多執行緒 (multi-threaded) 的全域變數。
3. 設備的硬體暫存器 (如狀態暫存器)

3.2.1 const 和 volatile 合用

<code>extern const volatile unsigned int rt_clock;</code>	(3.1)
---	---------

式 5.1 是在 RTOS kernel 常見的一種宣告：rt_clock 通常是指系統時鐘，它經常被時鐘中斷進行更新。所以它是 volatile。因此在用的時候，要讓編譯器每次從記憶體裡面取值。而 rt_clock 通常只有一個寫者（時鐘中斷），其他地方對其的使用通常都是唯讀的。所以將其聲明為 const，表示這裏不應該修改這個變數。所以 volatile 和 const 是兩個不矛盾的東西，並且一個物件同時具備這兩種屬性也是有實際意義的。

補充:

1. 一個參數可以同時是 const 也是 volatile 嗎？解釋為什麼。

2. 一個指標可以是 volatile 嗎？解釋為什麼。

3. 下面的函數有什麼錯誤：

```
int square(volatile int *ptr){  
    return *ptr * *ptr;  
}
```

解答：

1. 是(可參考 3.2.1)，舉例說明像是"read only 的狀態暫存器"。它是 volatile，因為它可能會被非預期的改變；它是 const，因為程式不應該試圖修改它。
2. 是的，儘管這並不常見。一個例子是中斷服務函式修改一個指向 buffer 的指標時。
3. 這段程式碼的目的是用來返指標*ptr 指向值的平方，但是，由於 *ptr 指向一個 volatile 型參數，編譯器將產生類似下面的程式碼：

```
int square(volatile int *ptr){  
    int a, b;  
    a = *ptr;  
    b = *ptr;  
    return a * b;  
}
```

因為*ptr 的值可能會被不預期的改變，因此 a 和 b 可能是不同的。所以，這段程式碼可能返回不是你所期望的平方值！正確寫法的程式碼如下：

```
int square(volatile int *ptr){  
    int a;  
    a = *ptr;  
    return a*a;  
}
```



第四章 預處理器與編譯器指令[1]

預處理器(Preprocessor)是 C 語言編譯過程中的一個階段，主要負責處理與原始碼文件相關的指令，包括：

文件包含 (#include):

#include 指令用於將其他源文件的內容插入到當前源文件中，使得程式可以模組化組織。

巨集定義 (#define):

#define 用於定義巨集（宏），這是一種文本替換機制，讓程式中的標識符替換為指定的內容。

巨集展開 (#undef):

#undef 用於取消先前使用 **#define** 定義的巨集，允許重新定義或取消巨集的效果。

條件編譯 (#if, #ifdef, #ifndef, #else, #elif, #endif):

條件編譯允許根據條件判斷來選擇性地編譯部分代碼，這是通過 **#if**、**#ifdef**、**#ifndef**、**#else**、**#elif** 和 **#endif** 等指令實現的。

錯誤指示 (#error):

#error 用於在預處理階段生成一條錯誤消息，強制中斷編譯過程。

預處理器還可以包括其他指令，如**#pragma** 用於向編譯器發送特定的指令，以及**#line** 用於修改行號和文件名的指令。總的來說，預處理器的作用是在實際編譯之前進行一些文本處理和替換，以影響最終的編譯結果。

此外，編譯器還有一個 **inline** 指令，用於提示編譯器將特定函數視為內聯函數，這表示在編譯時將函數的內容插入到呼叫該函數的地方，而不是通過正常的函數呼叫機制。這樣可以提高一些函數呼叫的效率，但也可能增加代碼大小。

4.1 巨集

巨集(macro)是在前置處理器(Preprocessor)執行前處理，將要替換的程式碼展開作文字替換。其語法範例如範例 4.1，需注意的是巨集使用時須把參數用括號括起來，不然容易發生如範例 4.2 之錯誤，當 `SUM(2, 5)*10` 時，因為沒有先括弧先乘除後加減，會得出結果為 52 的錯誤。

範例 4.1 巨集範例

```
#define PI 3.1415926    //常數巨集  
  
#define A(x) x          //函數巨集  
  
#define MIN(A, B) ((A) <= (B) ? (A) : (B)) //A 是否小於或等於 B，  
倘若 是則 MIN = A，否則 MIN = B
```

範例 4.2 巨集錯誤示範

```
#define SUM(a,b) a+b
```

4.2 #error 指令

#error 就是生成編譯錯誤的訊息，然後會停止編譯，可以用在檢查程式是否是照自己所預想的執行。其語法格式如範例 4.3 所示：

範例 4.3 #error 範例

```
/* 避免重複引入 */  
#ifndef MYHEADER  
#define MYHEADER  
...  
#endif
```

4.3 引入防護和條件編譯

4.3.1 引入防護

引入防護 (Include guard) 是一種條件編譯，用於防範 `#include` 指令重複引入的問題。

範例 4.4 引入防護範例

```
/* 避免重複引入 */  
#ifndef MYHEADER  
#define MYHEADER  
...  
#endif
```

如範例 4.4 所示，第一次引入時會定義巨集"MYHEADER"，再次引入時判斷 `#ifndef` 測試失敗，因此編譯器會直接跳到 `#endif`，由此避免了重複引用。另有非標準的指令 `#pragma once` 提供相同效果，但由於可攜性不如上例，因此大多時候還是上面提到的用法為主。

條件編譯還有一些其它應用，如範例 4.5 所示。

範例 4.5 條件編譯的其它應用

```
/* 若前處理器已經 define MYHEADER，就編譯 part A，否則編譯 part B。 */  
  
#ifndef MYHEADER  
  
#define MYHEADER  
  
    // part A  
  
#else  
  
    // part B  
  
#endif  
  
/* DEBUG flag */  
  
#ifndef DEBUG  
  
    print("device_open(%p) ", file);  
  
#endif
```

補充: [2]

型態	大小(bytes)
short	2
int	4
long	4
float	4
double	8
long double	16

4.4 inline

inline 可以將修飾的函式設為行內函式，即像巨集 (macro) 一樣將該函式展開編譯，用來加速執行速度。

inline 和 #define 的差別在於：

1. inline 函數只對參數進行一次計算，避免了部分巨集易產生的錯誤。
2. inline 函數的參數類型被檢查，並進行必要的型態轉換。
3. 巨集定義盡量不使用於複雜的函數
4. 用 inline 後編譯器不一定會實作，僅為建議。

補充:

inline、macro、function 三者差異。

macro	inline	function
preprocessor	compiler	compiler
不需定型態	需定型態	需定型態

macro: 在編譯之前，前處理器就將該程式替換至各個呼叫區塊

inline: compiler 可決定是否展開 inline

function: 透過執行時期函式呼叫(push, pop)，記憶體中只有一份實體，省空間耗時

第五章 邏輯運算子[1]

5.1 C 語言中的邏輯運算子

邏輯運算子(Bitwise operator)在 C 中的語法分別如下：

1. AND (&) // bit 值皆為 True 才為 True
2. OR (|) // bit 值僅需其中一個為 True 便為 True
3. NOT (!) // True 變 False，False 變 True
4. XOR (^) // bit 值不一樣則為 True
5. complement (~) // 位元反轉運算符號，詳見[註 2]
6. shift (<<, >>) // 位元左右移動運算符號，詳見[註 3]

bitwise 的操作常與 "0x" 這種 16 進位表示法，方便轉換操作。

範例 5.1 邏輯運算子的基本運算

```
unsigned long num_a = 0x00001111;  
unsigned long num_b = 0x00000202;  
unsigned long num_c;  
  
num_c = num_a & (~num_b);  
num_c = num_c | num_b;  
  
printf("%lx", num_c); // 00001313
```

範例 5.2 Mask 方法作 bitwise 操作

```
a = a | 7    // 最右側 3 位設為 1，其餘不變。  
a = a & (~7) // 最右側 3 位設為 0，其餘不變。  
a = a ^ 7    // 最右側 3 位執行 NOT operator，其餘不變。
```

[註 2]:

- 位元反轉運算符號 (Bitwise NOT, \sim)。當應用在一個二進位數字上時，它會將每個位元 (0 或 1) 反轉，即 0 變為 1，1 變為 0。例如，如果 x 是一個二進位數字 1100，則 $\sim x$ 會變成 0011。
- 位元反轉運算通常用於掩碼操作，數字位元的反轉，或位元的狀態切換。

[註 3]:

- \ll 和 \gg 是位元左移和右移運算符，分別代表位元左移和位元右移。
 \ll 用於將一個二進位數字的所有位元向左移動指定的位數。例如， $x \ll 2$ 會將 x 向左移動 2 位。
- \gg 用於將一個二進位數字的所有位元向右移動指定的位數。例如， $x \gg 1$ 會將 x 向右移動 1 位。
- 位元左移和右移通常用於實現二進位位元的位元操作，例如對整數的位元操作、位元掩碼操作等。

這些位元操作通常用於處理二進位數字，例如位元掩碼、位元操控、位元計數等。它們在系統編程和低級位元處理中非常有用。

範例 5.3 位元左移練習

```
#include <stdio.h>

int main() {
    int x = 5; // 二進位表示：0000 0101

    int result = x << 2; // 將 x 向左移動 2 位

    // 二進位結果：0001 0100，對應十進位為 20
    printf("Result: %d\n", result);

    return 0;
}
```

第六章 記憶體與字串複製[1]

6.1 記憶體複製(Memory Copy)

```
void *memcpy( void *dest, const void *src, size_t count );    ( 6.1 )
```

在 C 語言中可使用"memcpy()"複製任何類型的資料，如式 9.1。且不處理字串結束'\0'的情況，當 *src 長度大於 *dest 時會出現緩衝區溢位(Buffer Overflow)，但編譯時不會錯誤。

6.2 字串複製(String Copy)

```
void *strcpy( void *dest, const void *src);    ( 6.2 )
```

若要在 C 語言中進行字串複製，可使用"strcpy()"語法；strcpy() 只能用於字串複製，不需要指定長度，因為會自動偵測以 '\0' 為結尾，當 *src 長度大於 *dest 時會 buffer overflow (*dest 將沒有 \0)。

範例 6.1 記憶體複製與字串複製練習

```
#include <string.h>
#include <stdio.h>
int main (){
    const char *str1 = "abc\0def";
    char str2[16] = {0};
    char str3[16] = [1];

    strcpy(str2, str1);
    memcpy(str3, str1, sizeof(str3)); // 8
    printf("str2 = %s\n", str2);      // str2 = abc
    printf("str3 = %c\n", str3[5]); // str3 = e
    return 0;
}
```

第七章 延伸性資料型態[1]

本章節主要介紹 C 語言中之延伸性資料型態，其中包含了結構(struct)、重新定義型態名稱(typedef)、共用結構(union)以及共用結構(enum)，其中結構、重新定義型態名稱以及共用結構為 C 與 C++ 語言皆具備的，列舉則僅存在於 C++ 中。

7.1 結構

結構(Struct)是使用者自定的型態，包含數個不同資料型態的變數，將不同的資料型態關聯在一起，使他們的關聯更直覺。其使用方法如範例 7.1 所示。

範例 7.1 struct 練習

```
struct [structName] {  
    char name[16];  
    int age;  
    struct [structName] *ptr;  
    // 不能含有自己，但可以有自己型別的指標。  
};  
  
int main () {  
    struct [structName] person1 = {"Amy", 20 }; // 初始化  
    person1.age = 21; // 操作  
}
```

7.2 重新定義型態名稱

重新定義型態名稱(`typedef`)的功能為保留字可以位資料型態建立別名，使程式更容易閱讀和理解。如範例 7.2 所示。

範例 7.2 `typedef` 練習

```
#include <stdio.h>

typedef struct PERSON {
    char name[16];
    int age;
    struct PERSON *ptr;
} PERSON;

int main() {
    PERSON person1 = {"Amy", 20, NULL}; // 初始化時需要給 ptr 一個初值
    person1.age = 21;

    printf("Name: %s\n", person1.name);
    printf("Age: %d\n", person1.age);

    return 0;
}
```


7.3 共用結構

由於電腦架構早期記憶體空間比較不足，因此需要使用共用結構讓各變數共用一塊記憶體，共用結構(union) 所需的記憶體空間大小由最大的成員變數決定，例如範例 7.3 的共用結構大小為 8 位元組 (upper bound double)。

範例 7.3 union 練習

```
union data{  
    char c;  
    int num;  
    double fnum;  
};  
union data a, b;
```

7.4 列舉

列舉(enum) 是一種常數定義方式，可以提升可讀性，列舉裡的識別字會以 int 的型態，從指定的值開始遞增排列（預設為 0）。其使用方式如範例 7.4 所示。

範例 7.4 enum 練習

```
enum week_type { WED, THU, FRI, SAT };  
week_type week = WED;  
if (week == WED)  
    cout << week << endl; // 0
```

7.5 總結

struct 是每個成員變數都配置一段空間，union 則是共用一段記憶體空間。另外，union 需注意記憶體內的排列方式，如 little-endian 方法排列，int 會放在 double 的 byte 3~0 的位置，從而改變 double 讀取時的值。

第八章 資料存儲方式

在電腦科學中，Little Endian 和 Big Endian 是指多位元組資料類型在記憶體中的儲存方式。這兩種方式區別在於數值的低位元組（Least Significant Byte，LSB）和高位元組（Most Significant Byte，MSB）的儲存順序。

8.1 Little – Endian

定義：

在 Little Endian 系統中，數值的最低有效位（LSB）存放在最小的記憶體位址，而最高有效位（MSB）存放在較大的記憶體位址。

舉例：

十六進制數值 0x12345678 在記憶體中的儲存順序為 0x78 0x56 0x34 0x12，如圖 8.1 所示。

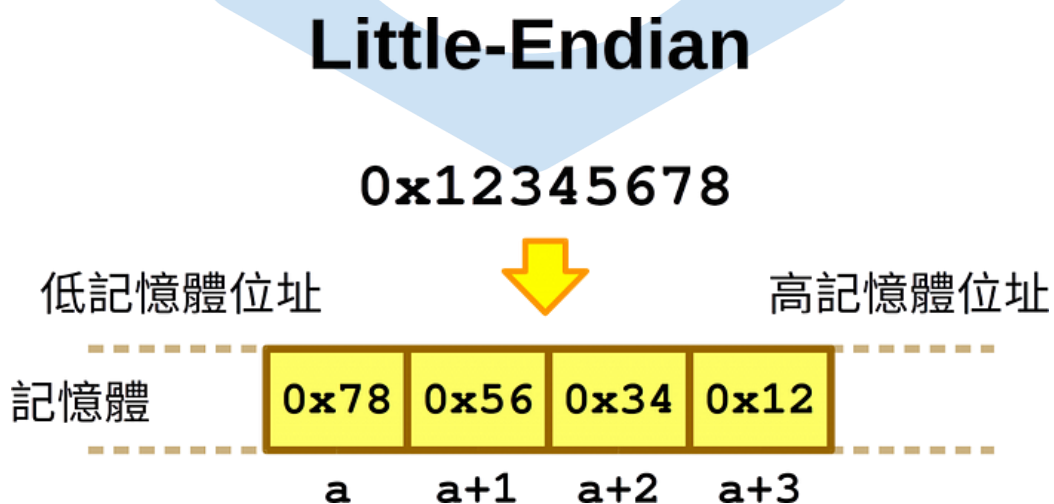


圖 8.1 Little – Endian 示意圖[5]

8.2 Big – Endian

定義：

在 Big-Endian 系統中，數值的最高有效位（MSB）存放在最小的記憶體位址，而最低有效位（LSB）存放在較大的記憶體位址。

舉例：

十六進制數值 0x12345678 在記憶體中的儲存順序為 0x12 0x34 0x56 0x78，如圖 8.2 所示。



圖 8.2 Big – Endian 示意圖[5]

8.3 C 語言判斷位元組順序[5]

如果我們的程式會使用 union 這類的低階存取方式處理記憶體中的資料，位元組順序就會是非常關鍵的問題，如果不確定自己的機器是屬於那一種位元組順序，可以使用範例 8.1 來判斷。

範例 8.1 判斷位元組順序

```
#include <stdio.h>
int main() {
    typedef union {
        unsigned int i;
        unsigned char c[4];
    } EndianTest;
    EndianTest t;
    t.i = 0x12345678;
    if (t.c[0] == 0x12 && t.c[1] == 0x34 && t.c[2] == 0x56 && t.c[3] ==
0x78){
        printf("Big Endian!!");
    }else if (t.c[0] == 0x78 && t.c[1] == 0x56 && t.c[2] == 0x34 && t.c[3]
== 0x12){
        printf("Little Endian!!");
    }else{
        printf("Other Endian!!");
    }
    return 0;
}
```

第九章 未定義行為

9.1 未定義行為[6]

在解釋未定義行為(Undefined behavior)前，我們先補充兩個常見的語法

1. `i++`：先用 `i`，再將 `i + 1`
2. `++i`：先用 `i + 1`，再用 `i`

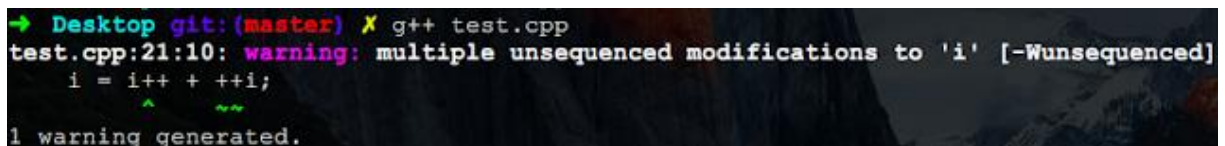
我們可以假設出現以下題目

```
int i = 10  
i = i++ + ++i;
```

在這個問題中多數人會說標準答案為

```
i = i++ + ++i;  
i = 10 + ++i;  
i = 10 + 12;  
i = 22
```

但實際上這個答案卻是有爭議的，如圖 9.1 所示，編譯器會發出警告訊息



```
→ Desktop git:(master) X g++ test.cpp  
test.cpp:21:10: warning: multiple unsequenced modifications to 'i' [-Wunsequenced]  
    i = i++ + ++i;  
        ^  ~~~  
1 warning generated.
```

圖 9.1 未定義行為編譯器警告訊息

為此，我們可以寫一段程式碼來檢測看看。

```
#include <stdio.h>

int main(int argc, char **argv) {
    int i = 10;
    i = i++ + ++i;
    printf("The answer : %dn", i);
    system("pause");
    return 0;
}
```

而上述程式在 Viual C++ 6.0 的編譯器(Compiler)執行結果為：

```
The answer : 23
```

此時讀者們應該會認為這便是正確答案了，很遺憾的是，答案是否定的。這可能會讓各位讀者感到困惑；如果真的要給這段程式碼一個答案，那應該會是未定義行為(Undefined behavior)，為何這個例子中會出現未定義行為呢？

問題就出在於，C 語言沒有規定 `i++` 或 `++i` 的加 1 動作到底是在一個句子裡的哪個時刻執行，因此，不同編譯器如果在不同的位置 +1，就有可能會有不同的結果，我們假設把上面的式子拆成幾個動作來看

1. `i++` 產生一份暫時物件（供參考用，否則+1 後值就改變了）
2. `i++` 的 `i` 遞增 1
3. `++i` 的 `i` 遞增 1

4. $(i++)$ 加上 $(++i)$

5. 將右邊運算的結果指派給 i

直覺上 $i++$ 似乎會在 $i++$ 被參考或產生暫時物件後進行加 1，但是這只是直覺上是如此，也有可能是在更後面甚至是在 $i =$ 右值 指定完之後才執行，答案就在於要看編譯器如何實作，因此不同的編譯器、編譯參數可能會有不同的答案就是如此，而這樣的程式的行為就叫做**未定義行為(Undefined behavior)**。



第十章 補充

10.1 運算子優先權

運算子優先權 (C 語言)

Precedence Table

	運算子 Operator	說明 Description	結合順序 Associativity
1	() [] -> . ++ --	Function call Array subscripting Element selection (of struct or union) through pointer Element selection (of struct or union) by reference increment/decrement (suffix) (註1)	左至右
2	! ~ ++ -- + - * & (type) sizeof	logic NOT bitwise NOT increment/decrement (prefix) unary plus and minus Indirection (dereference, right value) Address-of (left value) type cast Size-of	右至左
3	* / %	Arithmetic multiplication, division, and remainder	左至右
4	+ -	Arithmetic addition, subtraction	左至右
5	<< >>	Bitwise left shift, right shift	左至右
6	< <= > >=	Relational less than, not greater than Relational greater than, not less than	左至右
7	== !=	Relational equal, not equal	左至右
8	&	Bitwise AND	左至右
9	^	Bitwise XOR	左至右
10		Bitwise OR	左至右
11	&&	Logical AND	左至右
12		Logical OR	左至右
13	?:	Ternary conditional	右至左
14	= += -= *= /= %= &= ^= = <<= >>=	Direct assignment Assignment by sum, difference Assignment by product, quotient, remainder Assignment by bitwise AND, XOR, OR Assignment by bitwise left shift, right shift	右至左
15	,	Comma	左至右

10.2 各式題目

線上 C/C++ 編譯器: <https://www.onlinegdb.com/>

Bitwise 運算

Setting a bit

```
number |= 1UL << n;
```

Clearing a bit

```
number &= ~(1UL << n);
```

Inverse a bit

```
number ^= 1UL << n;
```

Checking a bit

```
bit = (number >> n) & 1U;
```

Write one line expression to check if a integer is power of 2

```
(x - 1) & x == 0
```

字串

反轉字串

```
void revstr(char *str1){
    char temp;
    int len = strlen(str1); // use strlen() to get the length of str string

    for (int i = 0; i < len / 2; i++){
        temp = str1[i];
        str1[i] = str1[len - i - 1];
        str1[len - i - 1] = temp;
    }
}
```

ToUpper

```
bool isUpper(int ch) {
    return (ch >= 'A' && ch <= 'Z');
}
int toUpper(int ch) {
    if (isUpper(ch))
        return ch;
    else
        return (ch + 'A' - 'a');
}
```

其他

最大公因數

```
int gcd(int x, int y){  
    if ( y == 0){ //餘 0，除數 x 即為最大公因數  
        return x;  
    }  
    else  
        return gcd(y, x % y); // 前一步驟的餘數為被除數，除數為除數
```

Decimal to Binary

```
int find(int decimal_number){  
    if (decimal_number == 0)  
        return 0;  
    else  
        return (decimal_number % 2 + 10 *  
            find(decimal_number / 2));
```

從兩個數字中找出最大的一個而不使用判斷描述

```
int max(int a, int b){  
    return (a + b + abs(a - b)) / 2;
```

給定一個 8-bit 整數，看是 1 的最高位元在第幾位？

```
int num3 = 125; // 01111101
int tmp;
int count = 0;
for (int i = 0; i <= 7; i++){
    tmp = (num3 << i) & (i << 7)
    if (tmp >> 7 == 1){
        printf("%d\n", 8 - i);
        break;
    }
}
```

```
int num33 = 125 // 01111101
for (int i = 7; i >= 0; i --){
    if (num 33 >> i) & (1)){
        printf("%d\n", i);
        break;
    }
}
```

給一個 unsigned short，換算成 16 進制後四個值是不是一樣，是回傳 1 否則 0

```
void check_16bit(unsigned short x){  
    // 0xFFFF return 1, 0xAAAB return 0  
  
    int check = 0;  
  
    check = !((x & 0xf) ^ ((x >> 4) & 0xf) ^ ((x >> 8) & 0xf) ^ ((x >> 12) & 0xf))  
  
    printf("%04x: %d\n", x, check);  
}
```

找出前 N 個元素的最大值

```
void findBiggest(struct Node* head, int range){  
    struct Node* temp = head;  
    int max_value = -INT16_MAX;  
    for (int i = 0; i < range; i++){  
        if (temp == NULL)  
            break;  
  
        if (temp->data > max_value)  
            max_value = temp->data;  
  
        temp = temp->next;  
    }  
  
    printf("\nMax value in the first %d elements is %d\n", range, max_value);  
}
```

找出前半部的元素最大值

```
void findBiggestInFront(struct Node **head_ref){
    struct Node *slow = *head_ref, *fast = *head_ref;
    int max_value = slow -> data;

    while(fast != NULL && fast -> next != NULL){
        if(slow -> data > max_value)
            max_value = slow -> data;
        slow = slow -> next;
        fast = fast -> next -> next;
    }
    printf("Max value in the first half is %d.\n", max_value);
}
```

取 Linked List 中間的 Node

```
void findMiddle(struct Node* head) {
    struct Node *fast = head, *slow = head;
    while(fast != NULL && fast -> next != NULL) {
        fast = fast -> next -> next;
        slow = slow -> next;
    }
    printf( "%d\n" , slow -> data );
}
```

判斷兩個 Linked List 是否相連

```
struct Node* get_intersection_node(struct Node *headA, struct Node *headB){  
    if (!headA || !headB)  
        return NULL;  
    struct Node *a = headA, *b = headB;  
    while(a != b){  
        a = a ? a -> next : headB;  
        b = b ? b -> next : headA;  
    }  
    return b;  
}
```


菱形問題

```
#include <stdio.h>

int main() {
    int n, space;
    printf("Enter the number of rows: ");
    scanf("%d", &n);

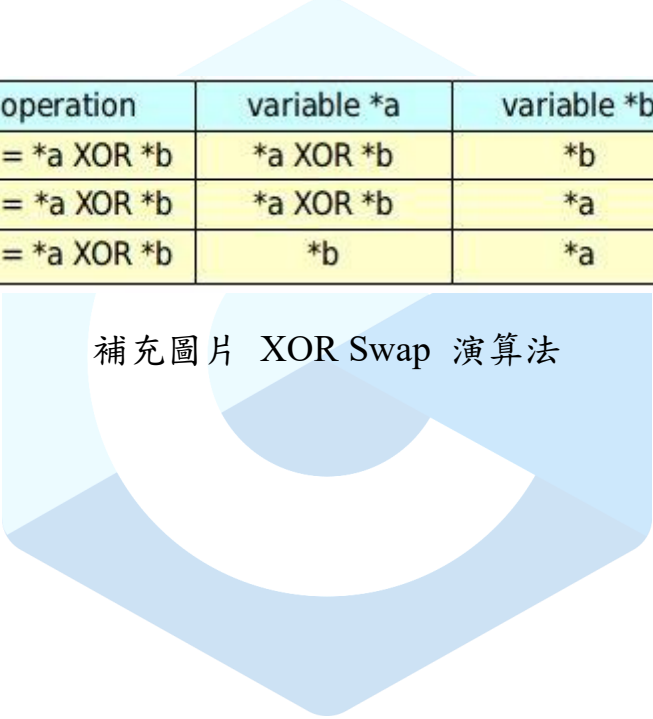
    space = n - 1;
    // 上半部分
    for (int i = 1; i <= n; i++) {
        // 打印空格
        for (int j = 1; j <= space; j++) {
            printf(" ");
        }
        // 打印星號
        for (int j = 1; j <= 2 * i - 1; j++) {
            printf("*");
        }
        printf("\n");
        space--;
    }

    space = 1;
    // 下半部分
    for (int i = 1; i <= n - 1; i++) {
        // 打印空格
        for (int j = 1; j <= space; j++) {
            printf(" ");
        }
    }
```

```

// 打印星號
for (int j = 1; j <= 2 * (n - i) - 1; j++) {
    printf("*");
}
printf("\n");
space++;
}
return 0;
}

```



operation	variable *a	variable *b
*a = *a XOR *b	*a XOR *b	*b
*b = *a XOR *b	*a XOR *b	*a
*a = *a XOR *b	*b	*a

補充圖片 XOR Swap 演算法

LeetCode

- ▶ Two Sum
- ▶ Longest Substring Without Repeating Characters
- ▶ Longest Palindromic Substring
- ▶ Container With Most Water
- ▶ Valid Parentheses
- ▶ Move Zeroes
- ▶ Merge Intervals



參考文獻

- [1] Opengate. "C/C++ — 常見 C 語言觀念題目總整理 (適合考試和面試) ." [Online] Available: <https://www.mropengate.com/2017/08/cc-c.html>.
- [2] Chienyu. " 韌體工程師的 0x10 個問題 ." [Online] Available: <https://hackmd.io/@Chienyu/S1loEqCuo>.
- [3] 郭晉魁. "C 語言有沒有 call by reference(or call by address) ?" [Online] Available: <http://eportfolio.lib.ksu.edu.tw/~T093000170/blog?node=000000119>.
- [4] 黃圳柏. "ADRIAN'S BLOG - [C 語言]兩變數內容值互換技巧." [Online] Available: <http://adrianhuang.blogspot.com/2011/08/c.html>.
- [5] S. Liu. "Medium - [C] 判斷 Big-Endian or Little-Endian." [Online] Available: <https://samuel830209.medium.com/c-%E5%88%A4%E6%96%B7big-endian-or-little-endian-d90da2a95086>.
- [6] Victor. "程式設計遇上小提琴-萬惡的未定義行為." [Online] Available: <https://blog.ez2learn.com/2008/09/27/evil-undefined-behavior/>.