

你所不知道的C語言：技巧篇

Copyright (憲C) 2017, 2019 宅色夫

直播錄影

你覺得自己懂C語言嗎？

-  拜託，我超強，工作這麼多年，一定很懂C。
不過我看GitHub裡頭的程式就不懂了
-  學過但是從來沒有學會過
-  每次看到C程式的奇技淫巧，都嚇死寶寶了

從矩陣操作談起

□ 最初的轉置矩陣實作: [impl.c](#)

```
void naive_transpose(int *src, int *dst, int w, int h)
{
    for (int x = 0; x < w; x++)
        for (int y = 0; y < h; y++)
            *(dst + x * h + y) = *(src + y * w + x);
}
```

使用方式如下:

```
int *src = (int *) malloc(sizeof(int) * TEST_W * TEST_H);
int *out2 = (int *) malloc(sizeof(int) * TEST_W * TEST_H);
naive_transpose(src, out2, TEST_W, TEST_H);
```

這有什麼問題呢？

- 不同的轉置矩陣操作，例如 `naïve` (這詞彙源自法文，我們引申為天真無邪的版本，不是“native”），或者針對 SIMD 指令集 SSE, AVX, AVX2 等強化的實作，都需要重複撰寫相當類似的程式碼，即可透過一致的介面來存取，這樣效能分析和正確性驗證的程式碼就能共用
 - 矩陣的內部資料表達機制「一覽無遺」，違反封裝的原則，而且不同版本的矩陣運算往往伴隨著特製的資料欄位，但上述程式碼無法反映或區隔
- 需要更好的封裝，這樣才能夠處理不同的內部資料表示法 (data structure) 和演算法 (algorithms)，對外提供一致的介面: [matrix_oo](#)

```
typedef struct matrix_impl Matrix;
struct matrix_impl {
    float values[4][4];
```

```

/* operations */
bool (*equal)(const Matrix, const Matrix);
Matrix (*mul)(const Matrix, const Matrix);
};

static Matrix mul(const Matrix a, const Matrix b) {
    Matrix matrix = { .values = {
        { 0, 0, 0, 0 }, { 0, 0, 0, 0 },
        { 0, 0, 0, 0 }, { 0, 0, 0, 0 },
        },
    };
    for (int i = 0; i < 4; i++)
        for (int j = 0; j < 4; j++)
            for (int k = 0; k < 4; k++)
                matrix.values[i][j] += a.values[i][k] * b.values[k][j];
    return matrix;
}

int main() {
    Matrix m = {
        .equal = equal,
        .mul = mul,
        .values = { ... },
    };

    Matrix o = { .mul = mul, };
    o = o.mul(m, n);
}

```

這樣的好處是：

- 一旦建立矩陣操作的實例 (instance)，比方說這邊的 `Matrix m`，可以很容易指定特定的方法 (method)，藉由 **designated initializers**，在物件初始化的時候，就指定對應的實作，日後要變更 (**polymorphism**) 也很方便

- 延伸閱讀: [Why does C++11 not support designated initializer list as C99?](#)
- 再說一次，從 C99 (含) 以後，C 和 C++ 的發展就走向截然不同的路線。[Incompatibilities Between ISO C and ISO C++](#)
- 省下指標操作，除了 function pointer 的使用外，在程式中沒有直接出現，對於數值表達較為友善
 - 善用 Compound Literals for Structure Assignment
- 克服命名空間 (namespace) 衝突的問題，注意上述 `equal` 和 `mul` 函式實作時都標註 `static`，所以只要特定實作存在獨立的 C 原始程式檔案中，就不會跟其他 compilation unit 定義的符號 (symbol) 有所影響，最終僅有 [API gateway](#) 需要公開揭露



Prefer to return a value rather than modifying pointers

This encourages immutability, cultivates pure functions, and makes things simpler and easier to understand. It also improves safety by eliminating the possibility of a NULL argument.



👎 unnecessary mutation (probably), and unsafe

```
void drink_mix(Drink * const drink, Ingredient const ingr) {
    assert(drink);
    color_blend(&(drink->color), ingr.color);
    drink->alcohol += ingr.alcohol;
}
```



👍 immutability rocks, pure and safe functions everywhere

```
Drink drink_mix(Drink const drink, Ingredient const ingr) {
    return (Drink) {
        .color = color_blend(drink.color, ingr.color),
        .alcohol = drink.alcohol + ingr.alcohol
    };
}
```

```
};  
}
```

不過仍是不夠好，因為：

1. Matrix 物件的 `values` 欄位仍需要公開，我們就無法隱藏實作時期究竟用 `float`, `double`, 甚至是其他自訂的資料表達方式 (如 [Fixed-point arithmetic](#))
2. 天底下的矩陣運算當然不是只有 4×4 ，現有的程式碼缺乏彈性 (需要透過 `malloc` 來配置空間)
3. 如果物件配置的時候，沒有透過 [designated initializers](#) 指定對應的方法，那麼後續執行 `m.mul()` 就注定會失敗
4. 如果 Matrix 物件本身已初始化，以乘法來說，我們期待 `matrixA * matrixB`，對應程式碼為 `matO = matA.mul(matB)`，但在上述程式碼中，我們必須寫為 `Matrix o = m.mul(m, n)`，後者較不直覺
5. 延續 2.，如果初始化時配置記憶體，那要如何確保釋放物件時，相關的記憶體也會跟著釋放呢？若沒有充分處理，就會遇到 [memory leaks](#)
6. 初始化 Matrix 的各欄、各列的數值很不直覺，應該設計對應的巨集以化簡
7. 考慮到之後不同的矩陣運算可能會用 [plugin 的形式](#) 載入到系統，現行封裝和 RTTI 不足
 - 要考慮的議題非常多，可見 [Beautiful Native Libraries](#)

延伸閱讀：

- [Fun with C99 Syntax](#)
- [An object-oriented paradigm in the C programming language](#)
- [以 C 語言實做 Javascript 的 prototype 特性](#)
- [更多矩陣運算: matrix.h, matrix.c](#)
- [“Object-oriented design patterns in the kernel” 中文翻譯](#)

- Part 1, Part 2

明確初始化特定結構的成員

- C99 紿予我們頗多便利，比方說：

```
const char *lookup[] = {
    [0] = "Zero",
    [1] = "One",
    [4] = "Four"
};
assert(!strcasecmp(lookup[0], "ZERO"));
```

也可變化如下：

```
enum cities { Taipei, Tainan, Taichung, };
int zipcode[] = {
    [Taipei] = 100,
    [Tainan] = 700,
    [Taichung] = 400,
};
```

- Initializing a heap-allocated structure in C

追蹤物件配置的記憶體

前述矩陣操作的程式，我們期望能導入下方這樣自動的處理方式：

```
struct matrix { size_t rows, cols; int **data; };

struct matrix *matrix_new(size_t rows, size_t cols) {
    struct matrix *m = ncalloc(sizeof(*m), NULL);
    m->rows = rows; m->cols = cols;
    m->data = ncalloc(rows * sizeof(*m->data), m);
    for (size_t i = 0; i < rows; i++)
        m->data[i] = nalloc(cols * sizeof(**m->data), m->data);
    return m;
}

void matrix_delete(struct matrix *m) { nfree(m); }
```

其中 `nalloc` 和 `nfree` 是我們預期的自動管理機制，對應的實作可見 `nalloc`

複製字串可用 `strdup` 函式：

```
char *strdup(const char *s);
```

`strdup` 函式會呼叫 `malloc` 來配置足夠長度的記憶體，當然，你需要適時呼叫 `free` 以釋放資源。

[heap]

`strdupa` 函式透過 `alloca` 函式來配置記憶體，後者存在 [stack]，而非 heap，當函式返回時，整個 stack 空間就會自動釋放，不需要呼叫 `free`。

```
char *strdupa(const char *s);
```

- `alloca` function is not in POSIX.1.

- `alloca()` function is machine- and compiler-dependent. * For certain applications, its use can improve efficiency compared to the use of `malloc(3)` plus `free(3)`.
- In certain cases, it can also simplify memory deallocation in applications that use `longjmp(3)` or `siglongjmp(3)`. Otherwise, its use is discouraged.
- `strdupa()` and `strndupa()` are GNU extensions.

- `alloca()` 在不同軟硬體平台的落差可能很大，在 Linux man-page 特別強調以下：

RETURN VALUE

The `alloca()` function returns a pointer to the beginning of the allocated space. If the allocation causes stack overflow, program behaviour is **undefined**.

- Dynamic Strings in C

延伸閱讀：

- Handling out-of-memory conditions in C

Smart Pointer

- 在 C++11 的 STL，針對使用需求的不同，提供了三種不同的 Smart Pointer，分別是：
 - `unique_ptr`
確保一份資源（被配置出來的記憶體空間）只會被一個 `unique_ptr` 物件管理的 smart pointer；當 `unique_ptr` 物件消失時，就會自動釋放資源。
 - `shared_ptr`
可以有多個 `shared_ptr` 共用一份資源的 smart pointer，內部會記錄這份資源被使用的次數

(reference counter) , 只要還有 shared_ptr 物件的存在、資源就不會釋放；只有當所有使用這份資源的 shared_ptr 物件都消失的時候，資源才會被自動釋放。

- weak_ptr

搭配 shared_ptr 使用的 smart pointer，和 shared_ptr 的不同點在於 weak_ptr 不會影響資源被使用的次數，也就是說的 weak_ptr 存在與否不代表資源會不會被釋放掉，

這些 smart pointer 都是 template class 的形式，所以適用範圍很廣泛；他們都是被定義在 `<memory>` 標頭檔、在 std 這個 namespace 下。

- 延伸閱讀: [C++ 智慧型指標（Smart Pointer）：自動管理與回收記憶體](#)

- [Implementing smart pointers for C](#)

- 原理：利用 GCC extension: [attribute cleanup](#)

```
#define autofree \
    __attribute__((cleanup(free_stack))) \
    __attribute__ ((always_inline)) \
inline void free_stack(void *ptr) { free(*((void **) ptr)); }
```

- 接著就可以這樣用：

```
int main(void) {
    autofree int *i = malloc(sizeof(int));
    *i = 1;
    return *i;
}
```

- Smart pointers for the (GNU) C: Allocating a smart array and printing its contents before destruction:

```
#include <stdio.h>
#include <csptr/smarter_ptr.h>
#include <csptr/array.h>

// @param ptr points to the current element
void print_int(void *ptr, void *meta) { printf("%d\n", *(int *) ptr); }

int main(void) {
    // Destructors for array types are run on every
    // element of the array before destruction.
    smart int *ints = unique_ptr(int[5],
                                  {5, 4, 3, 2, 1},
                                  print_int);

    /* Smart arrays are length-aware */
    for (size_t i = 0; i < array_length(ints); ++i)
        ints[i] = i + 1;

    return 0;
}
```

- GCC 的 C 語言前端只能在 variable attribute 指定 `__attribute__((cleanup))`。但函式回傳的 unbound 物件及函式參數屬性皆無支援 `__attribute__((cleanup))`。
- 缺乏上述兩特性，就無法做出「函式的回傳物件無須特別處理，即可自動 free」，及「傳入函式的物件，不做特別處理（如 move），就會自動 free」
- C++ 的 smart pointer 實際上就是用物件的 deallocator 會在 out-of-scope 時自動被呼叫的特性實作。見 `unique_ptr`。若 C 有實作以上兩者功能，其實也可做出完整的 unique pointer。

C99 Variable Length Arrays

- Visual C++ 目前不支援可變長度陣列
- 使用案例:

```
void f(int m, int C[m][m]) {
    double v1[m];
    ...
#pragma omp parallel firstprivate(C, v1)
    ...
}
```

- Randy Meyers (chair of J11, the ANSI C committee) 的文章 [The New C:Why Variable Length Arrays?](#), 副標題是 “C meets Fortran, at long last.”
- 一個特例是 [Arrays of Length Zero](#), GNU C 支援, 在 Linux 核心出現多次
 - C90 和 C99 語意不同

延伸閱讀: [Zero size arrays in C](#)

字串和數值轉換

- [Integer to string conversion](#)
- [qprintf](#): sprintf accelerator for GCC and Clang

GCC 支援 Plan 9 C Extension

- gcc 編譯選項 `-fplan9-extensions` 可支援 Plan 9 C Compilers 特有功能
- 「繼承」比你想像中簡單

```
typedef struct S {
    int i;
} S;

typedef struct T {
    S;           // <- "inheritance"
} T;

void bar(S *s) { }

void foo(T *t) {
    bar(t);    // <- call with implicit conversion to "base class"
    bar(&t->S); // <- explicit access to "base class"
}
```

- 若要在寫出 gcc/clang 中都支援的版本，可考慮改用 `-fms-extensions` 編譯選項。見 [GCC Unnamed Fields](#)

GCC transparent union

- C 語言實作繼承也可善用 [transparent union](#)
- 以上的繼承範例，在呼叫 base class 時得用 `&t->S` 或 type cast `(S*)t`。但若用 transparent union，即可透過更漂亮的語法來實作：

```
typedef union TPtr TPtr;

union TPtr {
```

```

    S *S;
    T *T;
} __attribute__((__transparent_union__));
}

void foo(TPtr t) {
    t.S->s_element;
    t.T->t_element;
}

T* t;
foo(t); // T * can be passed in as TPtr without explicit casting

```

- 這個特性也可用來實作polymorphism

```

typedef enum GenericType GenericType;
typedef struct A A;
typedef struct B B;

enum GenericType {
    TYPE_A = 0,
    TYPE_B,
};

struct A {
    GenericType type;
    ...
};

struct B {
    GenericType type;
    ...
};

union GenericPtr {

```

```
GenericType *type;
A *A;
B *B;
} __attribute__((__transparent_union__));
void foo (GenericPtr ptr) {
    switch (*ptr.type) {
    case TYPE_A:
        ptr.A->a_elements;
        break;
    case TYPE_B:
        ptr.B->b_elements;
        break;
    default:
        assert(false);
    }
}

A *a;
B *b;
foo(a);
foo(b);
```

計算時間不只在意精準度，還要知道特性

- 參照 [時間處理與 time 函式使用](#) 和 [计算机系统中的时间](#)
- [High Resolution Timers](#)

goto 使用的藝術

- 有時不用 goto 會寫出更可怕的程式碼

- 參照 [Computed goto for efficient dispatch tables](#)
 - Doing less per iteration
 - Branch prediction
- 延伸閱讀: [goto 和流程控制](#)

高階的 C 語言「開發框架」

[Cello](#) 在 C 語言的基礎上，提供以下進階特徵：

- Generic Data Structures
- Polymorphic Functions
- Interfaces / Type Classes
- Constructors / Destructors
- Optional Garbage Collection
- Exceptions
- Reflection

可寫出以下風格的 C 程式：

```
/* Stack objects are created using "$" */
var i0 = $(Int, 5);
var i1 = $(Int, 3);
var i2 = $(Int, 4);

/* Heap objects are created using "new" */
var items = new(Array, Int, i0, i1, i2);

/* Collections can be looped over */
```

```
foreach (item in items) {  
    print("Object %$ is of type %$\n",  
          item, type_of(item));  
}
```

善用 GNU extension 的 `typeof`

`typeof` 允許我們傳入一個變數，代表的會是該變數的型態。舉例來說：

```
int a;  
typeof(a) b = 10; // equals to "int b = 10;"  
  
char s[6] = "Hello";  
char *ch;  
typeof(ch) k = s; // equals to "char *k = s;"
```

`typeof` 大多用在定義巨集上，因為在巨集裏面我們沒辦法知道參數的型態，在需要宣告相同型態的變數時，`typeof` 會是一個很好的幫手。

以 `max` 巨集為例：

```
#define max(a, b)      \  
{ (typeof(a) _a = a;   \  
  typeof(b) _b = b;   \  
  _a > _b ? _a : _b; ) \  
}
```

至於為什麼我們需要將 `max` 的巨集寫成這樣的形式呢？為何不可簡單寫為 `#define max(a,b) (a > b ? a : b)` 呢？

這樣的寫法會導致 **double evaluation** 的問題，顧名思義就是會有某些東西被執行 (evaluate) 過兩次。

試想如下情況：

```
#define max(a, b) (a > b ? a : b)

void doOneTime() { printf("called doOneTime!\n"); }
int f1() { doOneTime(); return 0; }
int f2() { doOneTime(); return 1; }

int result = max(f1(), f2());
```

實際執行後，我們會發現程式輸出竟有3次 `doOneTime` 函式，但在 `max` 的使用，我們只期待會呼叫 2 次？

這是因為在巨集展開後，原本 `max(f1(), f2())` 會被改成這樣的形式

```
int result = (f1() > f2() ? f1() : f2());
```

為了解決這個問題，我們必須在巨集中先用變數把可能傳入的函式回傳值儲存下來，之後判斷就不要再使用一開始傳入的函式，而是用後來宣告的回傳值變數。

解釋以下巨集的原理

```
#define container_of(ptr, type, member) \
    __extension__({ \
        const __typeof__(((type *) 0)->member) *__pmember = (ptr); \
        (type *) ((char *) __pmember - offsetof(type, member)); \
    })
```

一步步拆解。首先看到的是 `__extension__`，是一個修飾字，用來防止 gcc 編譯器產生警告。

什麼情況下，我們會想編譯器產生的警告？在編譯階段，編譯器可能會提醒我們，程式使用到非 ANSI C 標準的語句，我們開發的程式在現在的編譯器可能可以過，但是用其他的編譯器可能就不會過了。

在這邊，出問題的地方應該是在開頭的 `({})` (braced-group within expression)，實際編譯過後我們可以看到這樣的警告訊息

```
warning: ISO C forbids braced-groups within expressions [-Wpedantic]
```

這是 gcc 一個 extension，透過這種寫法可以讓我們的 macro 更安全。

如果不想要看見這個警告，可在最前面加上 `__extension__` 修飾字。

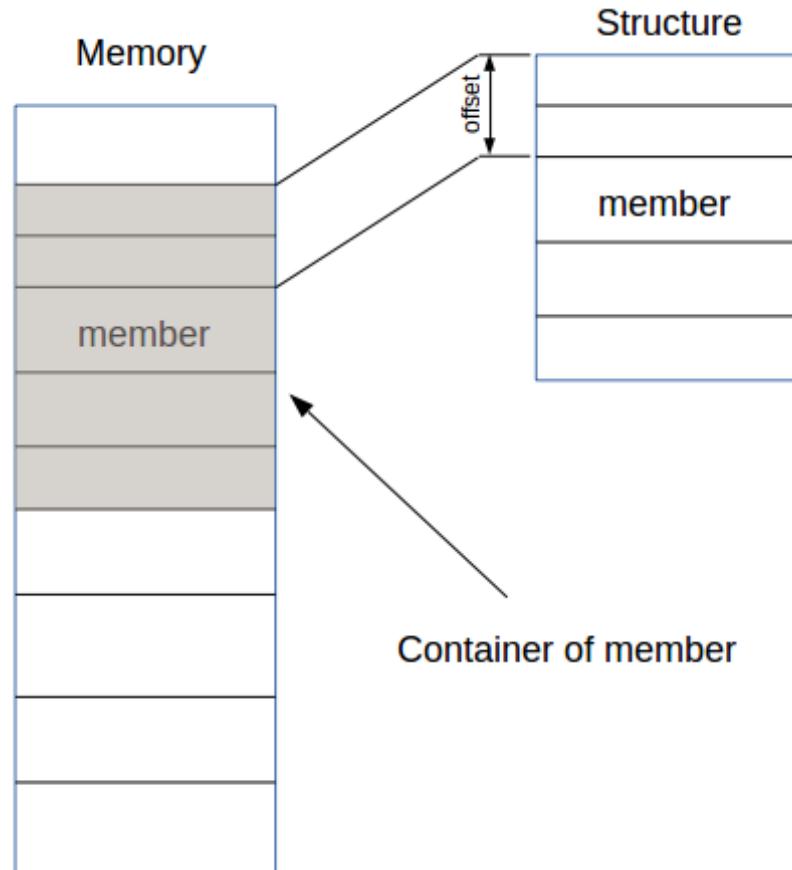
再來的話看到 `__typeof__((type *) 0)->member) * __pmember` 這段

可以看到我們用 `((type*) 0)->member` 的型態再宣告一個新的指標 `__pmember`。

`(type*) 0` 代表的是一個被轉型成 `type` 型態的 struct，這裡我們不用理會是否為一個合法的位址，所以可以直接寫 `0` 就好。再來我們讓他指向 `member`，因此 `__typeof__((type*) 0)->member)` 代表的便是 `member` 的資料型態。

再往下看到 `(type *) ((char *) __pmember - offsetof(type, member));` 這段比較好解釋，其實就只是把 `member` 的位址扣除 `member` 在整個 struct 裡面的偏移後，得到整個 struct 的開頭位址而已。

以圖像的方式會長這樣



充分掌握執行時期的機制

- [oomalloc](#): A library meant for testing application behavior in out-of-memory conditions with the use of LD_PRELOAD trick
- 延伸閱讀:
 - [動態連結器](#)

- 連結器和執行檔資訊
 - 執行階段程式庫 (CRT)
-

參考資訊

- [Obscure C](#)
 - [10 C99 tricks](#)
 - [C Tips and Tricks](#)
 - [C 语言有什么奇技淫巧？](#)
 - [C 語言的奇技淫巧](#)
-

Published on  HackMD

 27049

 7

