

程式人



用十分鐘瞭解

如何避免寫出悲劇的 C 語言

陳鍾誠

2016 年 3 月 2 日

今天早上

- 三個學生拿了一個 C 語言
程式問我！

讓我想起了

- 那些我曾經親手犯下的
C 語言悲劇

雖然

- 我並不是甚麼 C 語言神人，
或者嵌入式系統專家

但是你知道的

- 只要犯過的錯誤夠多
就可以開課教別人

教甚麼呢？

教大家怎麼樣

- 避免犯下同樣的錯！

這就是所謂

- 久病成良醫的道理了！

首先

- 讓我們看看，今天早上的那個悲劇！

是關於一個向量相加的程式

那個程式長這樣

```
double[] add(double a[], double b[]) {  
    double c[];  
    int m = sizeof(a);  
    int n = sizeof(b);  
    if (m == n) {  
        for (i=0; i<n; i++) {  
            c[i] = a[i] + b[i];  
        }  
    } else {  
        printf("error");  
    }  
    return c;  
}  
  
int main() {  
    double c[]=add([1,2,3,4], [4,3,2,1]);  
    printf("c="+c);  
}
```

您有注意到

- 上面這個程式裡到底有幾個錯嗎？

讓我們再看一遍

```
double[] add(double a[], double b[]) {  
    double c[];  
    int m = sizeof(a);  
    int n = sizeof(b);  
    if (m == n) {  
        for (i=0; i<n; i++) {  
            c[i] = a[i] + b[i];  
        }  
    } else {  
        printf("error");  
    }  
    return c;  
}  
  
int main() {  
    double c[]=add([1,2,3,4], [4,3,2,1]);  
    printf("c="+c);  
}
```

看起來

- 好像還不錯！

但是

- 這是採用 javascript 的標準
- 而不是採用 C 的！

讓我們對照一下

```
function add(a, b) {  
  var c=[];  
  if (a.length == b.length) {  
    for (var i=0; i<a.length; i++) {  
      c[i] = a[i] + b[i];  
    }  
  } else {  
    console.log("error");  
  }  
  return c;  
}
```

```
var c=add([1,2,3,4], [4,3,2,1]);  
console.log("c=", c);
```

```
D:\Dropbox\cccw\db\c\code>node shit1.js  
c= [ 5, 5, 5, 5 ]
```

JavaScript 版

```
double[] add(double a[], double b[]) {  
  double c[];  
  int m = sizeof(a);  
  int n = sizeof(b);  
  if (m == n) {  
    for (i=0; i<n; i++) {  
      c[i] = a[i] + b[i];  
    }  
  } else {  
    printf("error");  
  }  
  return c;  
}
```

```
int main() {  
  double c[]=add([1,2,3,4], [4,3,2,1]);  
  printf("c="+c);  
}
```

```
D:\Dropbox\cccw\db\c\code>gcc shit1.c -o shit1  
shit1.c:1:7: error: expected identifier or '<' b  
double[] add(double a[], double b[]) {
```

C 版

看出錯誤了嗎？

讓我替你把錯誤挑出來！

```
double[] add(double a[], double b[]) {  
    double c[];  
    int m = sizeof(a);  
    int n = sizeof(b);  
    if (m == n) {  
        for (i=0; i<n; i++) {  
            c[i] = a[i] + b[i];  
        }  
    } else {  
        printf("error");  
    }  
    return c;  
}  
  
int main() {  
    double c[]=add([1,2,3,4], [4,3,2,1]);  
    printf("c="+c);  
}
```

錯誤 1: 陣列沒有給大小

錯誤 2: sizeof 不能取出參數陣列大小

錯誤 3: 初始化陣列必須用大括號才對

錯誤 4: 陣列無法直接轉為字串或印出

其實、問題還不只這些

```
double[] add(double a[], double b[]) {  
    double c[];  
    int m = sizeof(a);  
    int n = sizeof(b);  
    if (m == n) {  
        for (i=0; i<n; i++) {  
            c[i] = a[i] + b[i];  
        }  
    } else {  
        printf("error");  
    }  
    return c;  
}  
  
int main() {  
    double c[]=add([1,2,3,4], [4,3,2,1]);  
    printf("c="+c);  
}
```

錯誤 5: 應該傳回指標，非陣列

錯誤 6: i 沒有宣告

錯誤 7: 接收也應該用指標，非陣列。

於是

- 在經過一連串的
修改、編譯、修改、編譯
之後

我們終於把程式改成這個樣子

```
#include <stdio.h>
#include <stdlib.h>

double* add(double *a, double *b, int n) {
    double *c=malloc(sizeof(double)*n);
    for (int i=0; i<n; i++) c[i] = a[i] + b[i];
    return c;
}

void print(double *a, int n) {
    for (int i=0; i<n; i++) printf("%6.2f ", a[i]);
}

int main() {
    double a[]={1,2,3,4};
    double b[]={4,3,2,1};
    double *c=add(a,b,4);
    print(c, 4);
}
```

而且、可以正確編譯了！

```
D:\Dropbox\cccwd\db\c\code>gcc shit1_correct.c -std=c99 -o shit1_correct  
D:\Dropbox\cccwd\db\c\code>shit1_correct  
5.00 5.00 5.00 5.00
```

但是、這樣的 C 程式

- 算是好的 C 語言程式嗎？

我想、應該不算！

為甚麼呢？

原因是、裡面有使用 malloc

```
#include <stdio.h>
#include <stdlib.h>

double* add(double *a, double *b, int n) {
    double *c=malloc(sizeof(double)*n);
    for (int i=0; i<n; i++) c[i] = a[i] + b[i];
    return c;
}

void print(double *a, int n) {
    for (int i=0; i<n; i++) printf("%6.2f ", a[i]);
}

int main() {
    double a[]={1,2,3,4};
    double b[]={4,3,2,1};
    double *c=add(a,b,4);
    print(c, 4);
}
```

使用了 malloc

但是

- 不使用 malloc ，那怎麼分配陣列空間呢？

而且

- 使用 malloc 有甚麼缺點呢？

關於這點

- 其實必須看應用而定！

如果你希望執行速度快

- 很快、而且非常快！
- 那麼或許應該避免使用
malloc，或者盡可能少用。

為甚麼呢？

- 因為 malloc 分配多了，容易造成記憶體縫隙過多
- 這會讓分配速度變慢，而且容易造成 malloc 失敗的情況

於是、每次 malloc

- 你都應該檢查是否分配成功，否則就要進行錯誤處理！

如果您仔細觀察

- C 語言的標準函式庫的設計
- 會發現像字串複製 `strcpy(a, b)` 這樣的函數，是沒有用到 `malloc` 的。

這種設計

- 讓你可以採用《無動態分配》的
《淨式》呼叫該函數。

筆者註：原本我採用《靜態》一詞代表《無動態分配》的函數，但是有網友提到這會和原本 C 語言裡的 **static** 靜態一詞混淆，為了避免這種混淆，所以我只好自己發明《淨式》這個詞，雖然感覺怪怪的，但至少比較不容易混淆。

以下就是《淨式》字串複製 的一個範例

```
#include <stdio.h>
#include <string.h>
int main() {
    char a[]="hello!", b[100];
    strcpy(b, a); ← 字串複製
    printf("b=%s\n", b);
}
```

筆者註：有網友提及，`strcpy` 若來源參數 `a` 的長度超過 `b` 的大小 100，可能會造成 **buffer overflow** 覆蓋掉 `b` 後面的記憶體之問題，這種問題也常常是駭客用來入侵系統的一種技巧！

為了消除 **buffer overflow** 的問題，可以改用 `strncpy(b, a, 100)` 的寫法來避開此問題！

這種《淨式》程式

- 速度可以很快、非常快！
- 而且不會有記憶體分配失敗的問題！

如果採用這種策略

- 也就是支持《淨式》的方法

那麼上述《向量相加程式》 就應該修改如下

```
#include <stdio.h>
#include <stdlib.h>

void add(double *c, double *a, double *b, int n) {
    for (int i=0; i<n; i++) c[i] = a[i] + b[i];
}

void print(double *a, int n) {
    for (int i=0; i<n; i++) printf("%6.2f ", a[i]);
}

int main() {
    double a[]={1,2,3,4};
    double b[]={4,3,2,1};
    double c[4];
    add(c,a,b,4);
    print(c,4);
}
```

可《淨式》呼叫的方式

《淨式》呼叫

當然

- 並不是所有程式都需要支援《淨式》呼叫的方式。
- 但這樣做除了速度快之外，還可以增加穩定性！

我依稀記得

- 美國太空總署和國防部，有規定一些 C 語言程式禁止使用 malloc，寧可事先分配大一點，而不希望在執行時期才出現記憶體不足的問題，我想就是這個原因！

在那樣嚴格的規定裏

- 不只 malloc 不能用，任何會造成記憶體分配的函數，像是 calloc, realloc, 以及 C++ 的 new 等等功能都不能用。
- 包含字串函數的 strdup 也是被禁止的。

當然

- 並非每個專案都是如此
但是使用 C 語言應該要知道
《淨式化程式》的可能性
因為這是寫出超快且穩定 C 程式的
一種方法

而且

- 當您把函數寫成《淨式》的時候
- 並不代表該程式絕對不能用 malloc 這種《動態分配》
- 也可以改在上層才進行動態分配

這樣就可以使用《對稱式的分配與釋放》寫法，避免寫出沒有正確釋放記憶體的程序。

舉例如下

```
#include <stdio.h>
#include <stdlib.h>

void add(double *c, double *a, double *b, int n) {
    for (int i=0; i<n; i++) c[i] = a[i] + b[i];
}

void print(double *a, int n) {
    for (int i=0; i<n; i++) printf("%6.2f ", a[i]);
}

int main() {
    double a[]={1,2,3,4};
    double b[]={4,3,2,1};
    double *c = malloc(sizeof(a));
    int len = sizeof(a)/sizeof(double);
    add(c, a, b, len);
    print(c, len);
    free(c);
}
```

淨式函數

在上層成對
分配與釋放

這種成對分配與釋放的寫法

- 比起之前的在函數內分配之寫法要好得多，比較不容易產生 bug

或許您還記得上面那個第一版程式 那版程式就有這個 bug

```
#include <stdio.h>
#include <stdlib.h>

double* add(double *a, double *b, int n) {
    double *c=malloc(sizeof(double)*n);
    for (int i=0; i<n; i++) c[i] = a[i] + b[i];
    return c;
}

void print(double *a, int n) {
    for (int i=0; i<n; i++) printf("%6.2f ", a[i]);
}

int main() {
    double a[]={1,2,3,4};
    double b[]={4,3,2,1};
    double *c=add(a,b,4);
    print(c, 4);
}
```

使用了 malloc

但是卻沒有
用 free 釋放

所以淨式函數
並不是不准用動態分配

- 只是把是否要做動態分配的
決策權，交給呼叫者自行決
定並處理而已！

有很多公司

- 會規定 malloc 和 free 一定要成對出現，否則就算是《違反規定的麻煩製造者》。
- 這是因為 C 語言的記憶體漏洞 (memory leak) 實在是非常可怕的緣故。

而且、要避免軟體出問題

- 還得在每次分配後，檢查分配是否成功。
- 如果失敗就要進行錯誤處理，或者回報錯誤。

像是這樣

```
#include <stdio.h>
#include <stdlib.h>

void add(double *c, double *a, double *b, int n) {
    for (int i=0; i<n; i++) c[i] = a[i] + b[i];
}

void print(double *a, int n) {
    for (int i=0; i<n; i++) printf("%6.2f ", a[i]);
}

int main() {
    double a[]={1,2,3,4};
    double b[]={4,3,2,1};
    double *c = malloc(sizeof(a));
    if (c==NULL) { printf("malloc fail"); exit(1); }
    int len = sizeof(a)/sizeof(double);
    add(c, a, b, len);
    print(c, len);
    free(c);
    return 0;
}
```

malloc 失敗時
要進行錯誤處理

另外、學習 C 語言的時候

- 一定要徹底理解《指標》
的意義

包含

- 元素指標、結構指標、函數指標
- 字串、陣列與指標個關係
- 如何用指標進行記憶體映射輸出入

以下讓我們

- 針對上述這些進階主題
分別舉例介紹

首先介紹函數指標

```
#include <stdio.h>
#include <math.h>

#define dx 0.001

double df(double (*f)(double), double x) {
    double dy = f(x+dx) - f(x);
    return dy/dx;
}

int main() {
    printf("df(sin(x), pi/4) = %f\n", df(sin, 3.14159/4));
}
```

f 是一個函數指標

把 sin 傳進去 df 給 f

呼叫傳進來的 f 函數

以上 df 函數乃是計算 f 函數在 x 點之斜率的《數值微分函數》

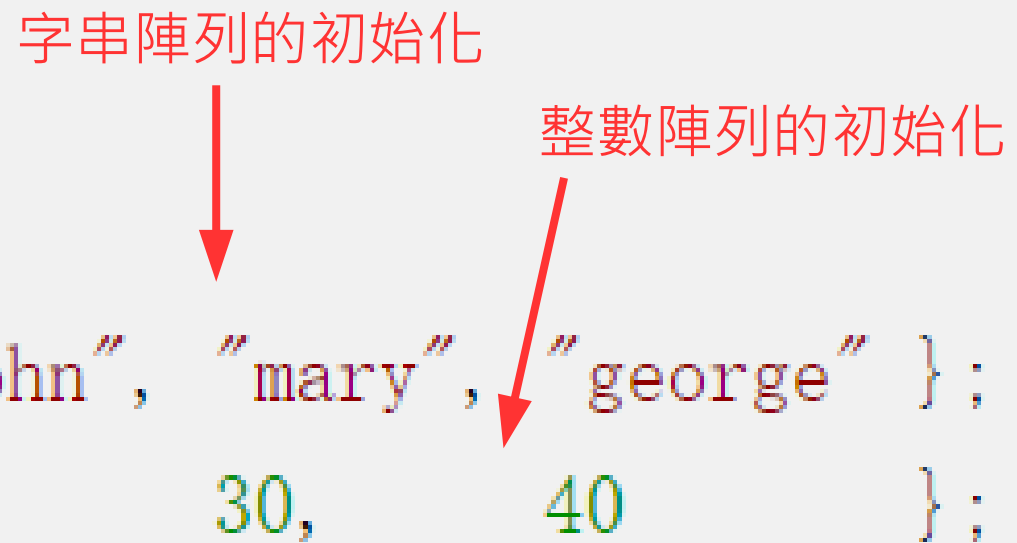
接著介紹字串與陣列的初始化

```
#include <stdio.h>
#include <string.h>
```

字串陣列的初始化

整數陣列的初始化


```
int size = 3;
char *name[] = { "john", "mary", "george" };
int age[] = { 20, 30, 40 };
```



然後介紹結構陣列的初始化

```
typedef struct {  
    char *name;  
    int  age;      結構陣列的初始化  
} People;
```

```
People peoples[] = {  
    { .name="john", .age=20},  
    { .name="mary", .age=30},  
    { .name="george", .age=40}  
};
```



以及整個結構的回傳

```
typedef struct {  
    double r, i;  
} Complex;  
  
Complex add(Complex *c1, Complex *c2) {  
    Complex c;  
    c.r = c1->r+c2->r;  
    c.i = c1->i+c2->i;  
    return c;  
}
```

回傳型態為複數 **Complex**

回傳結構變數 **c**

```
int main() {  
    Complex o1={ .r=1.0, .i=2.0 };  
    Complex o2={ .r=2.0, .i=1.0 };  
  
    print("o1", &o1);  
    print("o2", &o2);  
  
    Complex add12 = add(&o1, &o2);  
    Complex sub12 = sub(&o1, &o2);  
    Complex mul12 = mul(&o1, &o2);  
}
```

回傳後丟給 **add12**

然後是 C 語言最特殊的

- 透過指標進行《記憶體映射輸出入》
的這種特異功能！

以下是記憶體映射輸出入的範例

```
#define BYTE unsigned char
#define UINT16 unsigned short
#define BOOL unsigned char
#define SEG7_REG (*(volatile BYTE*) 0xFFFFF00)
#define KEY_REG1 (*(volatile BYTE*) 0xFFFFF01)
#define KEY_REG2 (*(volatile BYTE*) 0xFFFFF02)
#define KEY (KEY_REG2 << 8 | KEY_REG1)
```

```
// 鍵盤驅動程式
char keyboard_getkey() {
    UNIT16 key = KEY;
    for (int i=0; i<16; i++) {
        UNIT 16 mask = 0x0001 << i;
        if (key & mask !=0)
            return keymap[i];
    }
    return 0;
}
```

```
// 七段顯示器驅動程式
void seg7_show(char c) {
    SEG7_REG = map7seg[c-'0'];
}
```

```
BOOL keyboard_ishit() {
    return (KEY != 0)
}
```

當您理解了這些之後

- 我想您對 C 語言和其他語言之間的區別，應該就會有一個清楚的認識！


而這些特性

- 也正是 C 語言之所以成為
《系統軟體與嵌入式系統》
之主力語言的原因。

最後、在這篇文章刊出後

- 不少人給了《激烈的建議與批評》

像是這個

 陳先生，這種十分鐘教學，會害人。

C 的養成，沒有速成。

malloc 沒有與 free 配對問題，已經有先進提出來了。

而原程式裡本來有的「檢查」的觀念，在改寫的程式裏，都省去。

這樣的程式碼，只會讓學這套的學生，在公司裏，除錯會除不完、被釘釘不完！

讚 · 回覆 · 5小時 · 已編輯



陳鍾誠 謝謝！等我有空會補上這部分的說明！

讚 · 回覆 · 2小時

對於這些建議

- 我能理解的都已經在文中用
《筆者註：...》的方式進行了修改
與補強。
- 但仍然有一些是我無法理解的。

像是這個我就無法理解

舉例了「記憶體輸映射出入」卻沒有提到其危險性

還有這個雖然能理解

- 但是卻沒辦法舉例說明

在很多時候也要處理這種情形 (在 `stack boundary` 後面設一個寫入/讀取會觸發 `mmu exception` 的記憶體區塊，然後接 `handler` 來擴大 `stack` 或是例外處理)

關於這些問題

- 就留給各位讀者自己去思考了

希望這份投影片

- 能夠讓您更瞭解 C 語言！

這就是

- 我們今天的十分鐘系列！

希望您會喜歡

- 這次的十分鐘系列

我們下回見囉！

Bye bye !