

C 語言常見誤解

綠色代表對程式設計師是好消息，紅色代表對實作是好消息
(這篇只講好消息！不過不太適合初學者讀.....)

主要標準依據：[C99+TC1/2/3 草稿 N1256](#)

作者：[f0v0n;0\(f0v0n;0\(a\)900;|\)](#)，[\\$anyungYang\(|;|+|e\\$0n.ββ\\$@|+.cc\)](#)

[版權聲明](#)

[前言](#)

[指標 \(一\) 表示法和轉型](#)

[指標 \(二\) 空指標與 NULL](#)

[指標 \(三\) 指標運算](#)

[整數 \(一\) 表示法和位元運算](#)

[整數 \(二\) 字面常量](#)

[運算順序 \(一\) 最佳化](#)

[運算順序 \(二\) 表達式](#)

[結構與聯合](#)

[字元 \(一\) 表示法與位元組](#)

[字元 \(二\) 編碼](#)

[浮點數](#)

[main 函式](#)

[編譯器的檢查](#)

[勘誤與重大更動](#)

[感謝](#)

版權聲明

本篇文章採 [Creative Commons Attribution-ShareAlike 3.0 Unported License](#) (以下稱創用授權) 授權，但可能有部份內容未經原作者授權，乃以教學、研究等正當目的，透過合理使用直接重製於文章中。本文作者並不保證此合理使用絕對有效，因此在行使創用授權所賦予的權力前，請審慎評估此主張是否合理，並自行承擔因侵犯著作權所造成的損失。本文經原作者授權，得以標示姓名而引用之作品如下：

- 圖片  [LuMaxArt FS Collection Orange0147](#) 來自 [lumaxart.com](#) 採用 CC BY-SA 授權

前言

世界上不是只有程式語言標準；機器碼有標準，ABI 有標準，系統函式有標準，編譯器也有文件可以參考。然而很多寫 C 的人了解太多實作資訊，誤把特定實作當成 C 語言標準的一部分，

寫出許多不可攜 (non-portable) 的程式碼而不自知。不可攜不代表一定要換一台電腦或是改用嵌入式系統才會錯誤，只要換編譯器或是改變參數，都是一個新的實作；有很多寫 C 的人常提及開太多最佳化選項容易導致程式錯誤，其實常常只是程式碼本身不可攜罷了 (除非該最佳化故意違反標準)。同理，可攜程式碼也能一字不動的從 32 位元系統移植至 64 位元系統上 (或是稍早之前，從 16 位元系統移植至 32 位元系統)。

這篇文章企圖列出寫 C 語言一段時間後還是很難知道的陷阱。除了標準文件的考證外，也盡量舉出實際上會發生錯誤的實作 (機器、編譯器或某些參數等等)，以免淪為理論空談。問題清單主要從台灣 BBS 站 PTT 的 C/C++ 版實地考察蒐集而成。除標準委員會公佈的文件外，主要考證依據還包括 comp.std.c 討論區。

這篇文章並不是給初學者讀的，而是針對寫 C 一段時間的人而寫，因此不會有初學才有的問題。標準艱澀難懂，但有興趣的人可下載官方草稿 [N1256](#) 搭配答案一起看，其中 2.3p4 代表 2.3 第 4 段；本文語言標準的超連結全連往[非官方 N1256 網頁版本](#) (雖然無法由段落定位很可惜，但還是比一個 PDF 檔好)。

本篇文章收錄誤解問答的準則與精神：

- 可攜性為最高原則。「誤解」代表寫出不可攜的程式碼而不自知。特定平台上不可攜的程式碼可能有明確的定義，但不在討論範圍內。
- 本文認為「以為某語法正確」比「以為某語法錯誤」嚴重，因此
 - 不提保證可攜的炫技寫法。
 - 介紹安全寫法時，以實務上好記好理解為原則。當規則過於複雜時，寧可介紹更保守但保證安全的簡單作法。
- 只收錄長年寫 C 的人可能有的誤解，不收錄初學 C 時才容易犯的錯誤。
- 不收錄 C++ 為主的問答 (因為文章已經太長了)，不過可以適時提 C++ 的狀況。
- 優先收錄編譯器不用發出任何警告，而且有真實例子的誤解。

歡迎大家提供值得寫進文章的內容，或是指出某些問答違反以上準則或精神。文章冗長，難免有疏漏謬誤之處，還請大家不吝指教。

指標 (一) 表示法和轉型

- 指標不就是記憶體的位置嗎？不同型態的指標只有在位移 (指標運算) 時不一樣，實際上連函式指標都只是記憶體中的位置不是嗎？
並不是所有機器都使用「平面記憶體模型」 (flat memory model)，可以把記憶體當成一個巨大、連續的表格看待。除了標準 (參考 [N1256 6.2.5p27](#)) 規定的幾種狀況外，不同型態的指標可以長得完全不一樣。最後，函式指標 (function pointer) 是另一世界的東西，可以跟物件指標完全沒關係。並不是所有機器都把物件和程式碼都擺在同一大塊記憶體裡面。

如果你測試發現一樣，那只是你測試的實作用同一種方法表示所有指標，請參考你作業系統或機器的文件確認你的發現。世界上的確有實作用不同格式表示不同型態的指標，不可攜的程式在那些機器上可能會出現嚴重的錯誤。 (參考 [clc FAQ 5.17](#) 看實際例子)

- 所有指標都能轉型成 `void*` 或是 `char*` 不是嗎？
標準不保證函式指標也可以轉型成 `void*` 或 `char*`。(參考 [clc FAQ 4.13](#) 和 [N1256 6.3.2.3p1](#))
- 指標可以轉型成 `int` 或是 `long` 不是嗎？
就算不考慮函式指標，還是不保證成功，而且轉過去的數字可能也無法直接讀。實際的例子像是 AMD64 指令集下 Linux/Mac OS/Solaris/FreeBSD 等等 (採用 LP64 資料模型) 裡面的 `int` 或是 Microsoft Windows 的 X64 版本 (採用 LLP64 資料模型) 裡面的 `int` 和 `long` 都存不下 64 位元的指標。可攜程式碼不需要因為指標大小變了改寫。
(參考標準 [N1256 6.3.2.3p6](#))
- 把指標 (直接) 轉型成 `intptr_t` 或 `uintptr_t` 總可以了吧？
同上。就算不考慮函式指標，還是不保證成功。
- 那到底要怎麼寫才能轉型成整數存起來？
先轉型成 `void*` (注意函式指標可能不能轉) 然後再轉型成 `intptr_t` 或 `uintptr_t`.

```

■ intptr_t i = (intptr_t)(void*)p; // 假設 p 的型態是 int*

```

 要用時先轉回 `void*` 再轉回原來指標。

```

■ int* pi = (int*)(void*)i;

```

 這是萬無一失，絕對可攜的作法。唯一的問題是實作可能沒有提供 `intptr_t`。在某些機器上指標也不用數字表示。(參考 [N1256 7.18.1.4p1](#) 和 [clc FAQ 5.17](#) 看實際例子)
- 如果兩個指標比較 (`==` 運算子) 相等，(成功) 轉型成整數型態後比較也會相等嗎？
沒有保證，因此用此種方法來做 `hash` 之類的不可攜。可轉型回原本的指標型態再比較是否相等。(參考 [comp.std.c](#) 的討論)
- 假如指標成功轉型成另一個型態的指標，應該就可以用轉型後的指標去存取物件吧？
否，轉型成功不代表可以用來存取。標準對於什麼型態的指標能存取什麼型態的物件有嚴格規定 (type-based aliasing rules)。例如 `int` 物件不能被指向 `short` 的指標存取，即使指標轉型成功也不行。除了標準規定的例外外，編譯器可以假設不同型態的存取指向不同物件，做超乎想像的最佳化，把運算順序隨意調動；如下列程式碼：

```

■ int i = 0;
■ short *s = (short*) &i;
■ i = 0;
■ *s = 1;

```

 在實際的機器上，最後 `i` 的值可能還是 0。如果想違反標準，請了解如何關掉相關的最佳化。(參考 [N1256 6.5p7](#), [Linus Torvalds 的抱怨](#)，[gcc 應該可用 `-fno-strict-aliasing` 關掉](#) 和 [N1520: Fixing the rules for type-based aliasing](#))
- 假設 `p` 是一個指標，那麼 `printf("%p", p)` 就該印出 `0x7fff12345678` 之類的東西不是嗎？

printf 系列函式的 %p 只能印指向 void 的指標，所以必須先轉型成 void* (注意函式指標可能不能轉)。只要 p 不是函式指標就可以這樣寫：

```
printf("%p", (void*)p);
```

注意即使轉型成功，指標要怎麼印是實作決定，不一定要印成十六進位，也不一定要加上 0x。(參考 [N1256 7.19.6.1p8](#))

- 我作業系統的文件說 %p 其實可以當作 %#x 或是 %lx。這是不是代表其實可以用 %#x 或 %lx 印出？
查作業系統文件值得鼓勵，但如同問題所描述，這頂多只有在該作業系統上才對會。印指標還是要用 %p 或是用上述方法轉成 intptr_t 再印出來。

指標 (二) 空指標與 NULL

- NULL 是空指標嗎？型態是什麼？
NULL 不一定是指標，但一定是空指標常數 (兩者概念不同)。空指標常數轉型成指標後保證可以拿到正確的空指標，但自己不一定是指標。C99 (N1256) 中，空指標常數代表計算結果為零的整數常數表達式，如 0 或 0ULL，或是前述表達式轉型成 void*，如 (void*)(0U)。

C++03 中空指標常數只能是計算結果為零的整數常數表達式，如 0，連指標都不是，在多載時可能會選了錯誤的函式。C++0x 引入 nullptr 企圖解決這問題。(可參考較容易閱讀的 [More C++ Idioms/nullptr](#))
- 下列程式碼一定可以拿到空指標嗎？

```
int a = 0;  
int *p = (int*)a;
```


不保證。只有常數表達式才保證會轉型成空指標，執行期間才得到的零不保證可以成功轉型成空指標。一個可能的原因是¹，不是所有機器的空指標都是零 (見此題目)，如果要允許執行期間的空指標轉換，程式很可能需要在執行期間特別檢查數字是不是零。如果是常數，編譯器可以在編譯期間就轉成適當的空指標。(參考 [clc FAQ 5.18](#))
- 所以 printf("%p", NULL) 嚴格符合標準嗎？
否。這樣才嚴格符合標準：

```
printf("%p", (void*)NULL);
```


注意 NULL 不是空指標，但 NULL 轉型成指標後一定是空指標。
- 如果每次用 NULL 都先轉型成適當的型態 (如寫 (int*)NULL) 是不是就沒有陷阱了？
是 (雖然如果非常清楚標準的話，其實不用每個地方都寫)。這個轉換對 C++ 尤其重要，可以避免很多問題，見上面 [NULL 是空指標嗎？](#)。
- 為什麼這麼麻煩？空指標一定是零不是嗎？為什麼不能直接把 NULL 定成空指標？

¹這是本文作者亂猜的，沒有任何依據。

不是所有機器的空指標 (的表示法) 都是零，而且不是所有指標都長得一樣 (見「指標表示法」)，所以很難定義一個萬用空指標。C99 (N1256) 中規定空指標跟空指標常數比較會相等，也可以把空指標常數寫入一個指標 (編譯器會在編譯期間做適當的轉換)。(參考 [clc FAQ 5.17](#) 看實際機器)

- 既然空指標不一定是零，所以寫 `NULL` 比寫 `0` 好不是嗎？因為可以應付不同的機器。這無助於可攜性，比較像是寫作風格和美觀的問題 (可能有助於除錯)。C99 (N1256) 中常數零一定是合法的空指標常數，一定可以成功轉型成空指標，也一定可以和其他指標比較。

- 假設全域變數寫

```
int * p;
```

既然空指標不一定是零，`p` 會初始化成什麼呢？

C99 保證會初始化成空指標 (即使在空指標不是零)。(參考 [N1256 6.7.8p10](#))

指標 (三) 指標運算

- 假如有兩個指標型態都是 `int*`，可以相減得知他們相差多遠嗎？一般來說不行。兩個指標必須指向相同陣列的元素²，或是該陣列最後一個元素超過一格的位置，才能相減比較。陣列太大也有可能出狀況，請看下一個問題。(另外參考這個問題和 [clc FAQ 5.17](#) 看實際例子)

- 指標相減得到的距離是什麼型態？

是有號整數型態 `ptrdiff_t`。注意如果因為陣列太大導致這個型態無法表示之間的距離，則相減視為嚴重的錯誤。請務必檢查 `PTRDIFF_MAX` 和 `PTRDIFF_MIN` 確保此型態滿足程式所需。

- 我知道陣列和指標不同，但既然陣列在很多地方都會自動轉型成指標，如果把 `int a[100]` 的型態理解成 `int * const` (不能改內容的指標) 會有什麼陷阱呢？一個經典的例子是如果第一個檔案寫

```
int a[100];
```

第二個檔案寫

```
extern int * const a;
```

則在很多實作中，第二個檔案內存取 `a[0]` 時，編譯器會把第一個檔案 `a[0]` 的內容硬當作 `int` 指標 `p` 再把 `*p` 當作計算結果，進而發生嚴重的錯誤。在這例子中 C99 (N1256) 並不要求編譯器發出任何錯誤訊息。

- 如果一個結構是

```
struct {  
    int heap[1024];  
    int stack[1];  
};
```

²在做指標運算時，指向非陣列物件的指標當作指向大小為一的陣列的指標。

```
};
```

那可以用 `.stack[-1]` 去存取 `heap` 的內容嗎？

不行。首先要考慮填充的問題，`heap` 和 `stack` 中間不一定連續。就算確定沒有填充，陣列存取等價於先做指標運算後再解參照，而指標運算只能在同一陣列內部或超出最後一個元素一格的位置之間移動。也就是說，符合標準的編譯器可以額外加上陣列範圍檢查。

- 如果一個陣列 `a` 型態為 `int[5][5]` 可以用 `a[1][10]` 去存取嗎？陣列不是連續的嗎？這樣存取不保證成功。雖然陣列是連續的，看起來也可以存取，但是不行。陣列存取等價於指標運算加上解參照，在這例子中等價於 `*(*(a+1)+10)` 這個表達式。但指標運算只能在同一陣列內部或超出最後一個元素一格的位置之間移動。`+10` 違反了這個限制。（請一同參考指標相減的規則。）標準允許實作主動檢查指標是否超過範圍，甚至用完全不同的方法表示陣列。（參考 [N1256 Annex J.2p1](#)³ 和 [clc FAQ 5.17](#) 看實際例子）

整數（一）表示法和位元運算

- 有號整數表示法中的位元分三種：正負號、值和填充。正負號和值的格式可以是二補數（補碼）、一補數（反碼）或正負號加上大小（原碼）三種格式其中之一。無號整數表示法則少了正負號位元。（參考 [N1256 6.2.6.2p1~p2](#)、以及 [C Defect Report #069](#)）
- 並不是所有的位元組合都能表示合理的數字，存取某些位元組合在某些機器上可能會造成嚴重錯誤，此種組合稱作陷阱表示法（trap representation）。除非使用位元運算或是違反標準其他規定（如溢位），一般的運算不可能產生陷阱表示法。標準明確允許實作自行決定在以下兩種狀況下是否是陷阱表示法：
 - 型態為有號整數且正負號及值位元為特定組合時（三種格式各有一特殊組合）。
 - 填充位元為某些組合時。（參考 [N1256 註腳⁴ 44, 45](#)）
- 位元運算會忽略填充位元，因此（等級不輸給 `unsigned int` 的）無號整數可安心使用。⁵⁶為了最大可攜性，位元運算不應該用在有號整數上。
- `uintN_t` 和 `intN_t` 保證沒有填充位元，`intN_t` 一定是二補數，而且 `intN_t` 不可能有陷阱表示法，堪稱是最安全的整數型態。實作可能不提供這些型態，但一旦提供就要保證這些好性質。（參考標準 [N1256 7.18.1.1p1~p3](#)）

- `int` 剛好 32 位元不是嗎？
不一定。整數的寬度（正負號和值位元的數量）沒有上界，只要能表示標準規定的數字範圍即可。更何況除了寬度之外，可能還有其他填充位元。同理 `short` 也不一定是 16 位元，`long long` 也不一定是 64 位元。想要固定寬度請使用 `int32_t`。

³標準的 Annex J 沒有約束力，但仍值得參考。

⁴註腳並無約束力，但可作參考。

⁵雖然標準有模糊的地方，但這是比較合理的解讀；目前標準委員會也沒發現任何因填充位元造成錯誤的實作。（參考 [C99 Rationale rev5.10 6.2.6.2](#) 以及 [comp.std.c](#) 上的討論）

⁶要求等級不輸給 `unsigned int` 是為了防止可能的整數自動轉型。（參考 [N1256 6.3.1.1p2](#)）

- 那 `int` 至少 32 位元吧？
也不一定。因為 `int` 只保證能存下 $-2^{15}+1$ (-32767) 到 $2^{15}-1$ (32767) 之間的整數，16 位元已經足夠。`int_least32_t` 和 `int_fast32_t` 可以保證存下至少 $-2^{31}+1$ 到 $2^{31}-1$ 之間的整數（由於不一定是沒有陷阱表示法的二補數，所以保證範圍的下限不是 -2^{31} 而是 $-2^{31}+1$ ）。
- 我想要有一個 300 乘 300 的 `double` 陣列，`malloc(300*300*sizeof(double))` 有什麼問題？
`300*300` 可能超出 `INT_MAX`，而且 `300*300*sizeof(double)` 可能超出 `SIZE_MAX` 或 `INT_MAX`（看實作決定轉型成哪個型態）。實際上比較危險的狀況是有可能 `malloc` 實際上只給了一塊很小的記憶體，但程式卻當作一塊很大的記憶體使用，造成可能的緩衝區溢位漏洞。為了要安全可攜可以做兩件事情：第一個是盡量從頭到尾維持型態 `size_t` 或範圍更大的無號整數型態，所以 `sizeof` 擺前面（順序很重要），而且中途所有數字都是等級在 `unsigned int` 以上的無號整數（如常量尾巴加上 `u`）；第二個要保證運算結束後不會超過 `size_t` 的範圍。一個可能寫法如下：⁷

```

if (SIZE_MAX / 300u / 300u < sizeof(double)) {
    p = NULL;
} else {
    p = malloc(sizeof(double)*300u*300u) ;
}

```

（參考只寫一半的 [clc FAQ 7.16](#), [clang 的檢查](#)）
- 到底要怎麼用 `int` 才不會超出範圍？！
`int` 保證可以存下 $-2^{15}+1$ 到 $2^{15}-1$ 之間的整數。更一般的寫法是使用 `INT_MAX` 和 `INT_MIN` 得知真正的範圍。其他整數型態都可用類似的方法得到範圍。
- 假如 `sizeof(int)` 為 4 或是確定 `int` 佔 32 位元，是不是代表 `int` 剛好可以儲存 -2^{31} 到 $2^{31}-1$ 之間的整數？
不一定。首先一個位元組不一定是 8 位元（見此問題）。即使是，`int` 表示法中不一定每個位元都會用來表示值（這種位元稱作填充位元）。退萬步言，即使寬度（不含填充位元）剛好為 32 位元，`int` 的格式可能也不是二補數，所以不一定是從 -2^{31} 開始算。再退萬步言，縱使用二補數，特定組合可能是陷阱表示法，所以可能還是無法表示 -2^{31} 。用 `int32_t` 可以避開以上所有問題，滿足所有需求，除了 `sizeof(int32_t)` 不一定是 4。⁸
- 假如 `i` 的型態是整數。能不能用 `memset(&i, 0, sizeof(i))` 歸零？
標準委員會已決定向廣大程式碼妥協。注意只有 0 有特赦條款保證。（參考 [C Defect Report #263](#) 看標準如何妥協，以及 [N1256 6.2.6.2p5](#)）

⁷暫時找不出漏洞了，歡迎大家來找碴！

⁸`intN_t` 的大小一定是 N 位元（因為沒有填充位元），但一位元組不一定等於 8 位元，因此 `sizeof(int32_t)` 不一定是 4。

- 假設 `a` 和 `b` 兩個變數有相同的整數型態，`a^=b`；`b^=a`；`a^=b`；是否可讓兩數交換？
不保證。因為 `a^b` 可能會產生陷阱表示法。（參考陷阱表示法）
- 該不會 `|` `^` `&` `~` 四種位元運算都可能產生陷阱表示法（trap representation）？
沒錯。例如在無號整數上都有可能產生陷阱表示法（其他某些型態也有可能）。（參考陷阱表示法）
- 那應該如何安全的使用位元運算？
使用等級不輸給 `unsigned int` 的無號整數可高枕無憂。⁹
- 假設 `a` 的型態是 `unsigned int` 且寬度恰為 32, `a<<32` 結果會是 0 嗎？
不保證（參考 N1256 6.5.7p3）。實際上有些處理器結果會是 `a` 而不是 0, 因為在那些機器上 `a << b` 實際上是 `a << (b % 32)`。（參考 KennyTM 舉的現實例子）
- 要如何區辨是 little-endianness 還是 big-endianness？
世界上有機器兩者都不是，理論上也不可能有（簡單的）可攜寫法可以判斷，請仔細考慮是否真的需要判斷機器怎麼存數字。網路傳資料時請用系統提供之轉換函式。（參考 middle-endian 和 bi-endian）

整數（二）字面常量

- 80000U 一定是 `unsigned int` 嗎？
不一定。`unsigned int` 可能只能表示到 65535. 不能表示的情況下，會變成 `unsigned long`.
 - 0x33FF 是無號整數嗎？
正好相反，一定是有號整數。因為 `int` 一定可以表示此數字。
 - 好吧，那 0xFFFFFFFF 一定是有號整數嗎？
不一定。
 - 假設 `int` 能表示的上限是 2147483647, 2147483648 會轉成 `unsigned int` 嗎？
否。
-
- 尾巴有 `u` 或 `U` → 無號串列
尾巴沒有 `u` 或 `U`, 且為十進位 → 有號串列
尾巴沒有 `u` 或 `U`, 且為八或十六進位 → 通吃串列
 - 尾巴有 `l` 或 `L` → 等級二起跳
尾巴有 `ll` 或 `LL` → 等級三起跳

⁹當然熟知標準的話，用什麼型態都能避開陷阱。使用等級不輸給 `unsigned int` 的無號整數只是一個簡便的作法。

	一	一	二	二	三	三	四	魔王
有號串列	int		long		long long		其他延伸無號整數型態 (不一定有)	掛掉
無號串列		unsigned int		unsigned long		unsigned long long	其他延伸有號整數型態 (不一定有)	掛掉
通吃串列	int	unsigned int	long	unsigned long	long long	unsigned long long	其他延伸整數型態 (不一定有)	掛掉

- 規則：在正確的串列中，找符合等級要求且可以表示該數字的型態中，最左邊的那個。C++03 沒有等級三和等級四，C++1x 直接從 C99 抄過來。(參考 [N1256 6.4.4.1p6](#), C++03 2.13.1p2 和 C++1x N3242 2.14.2p3)

運算順序 (一) 最佳化

- ```
int a++;
int b++;
```

編譯後的組語或機器碼中，a 一定會先被更動嗎？

不一定。除了少數例外外，機器實際的運作方式可以和抽象機器完全不同。編譯器很可能為了最佳化更動運算順序，甚至讓 ( 編譯器覺得 ) 不相干的運算交錯進行。( 參考 [N1256 5.1.2.3p9](#) )
- 可是我聽說分號那裏有循序同步點 ( sequential points )，代表所有副作用都要完結？C99 (N1256) 中用循序同步點用來規定抽象機器的行為<sup>10</sup>，所以是抽象機器上的副作用要完結，而不是真實機器。編譯器不用一直維持真實機器和抽象機器之間的對應。( 參考 [N1256 5.1.2.3p9](#) )
- ```
int a;
a = b / c;
fprintf (stderr, "done\n");
```

看到訊息的同時代表 a 已經更新完成了嗎？

不一定。因為 fprintf 存取不到 a 所以編譯器可以自由的改變運算順序。(參考 [N1256 5.1.2.3p9](#))
- (承上) 那如果改寫

```
int a;
```

¹⁰新的 C 標準用了新的方法規定抽象機器的運算順序。

```
    a = b / c;
    fprintf (stderr, "done: %d\n", a);
```

看到訊息的同時代表 `a` 的值已經更新完成了嗎？

還是不一定，因為 `fprintf` 還是存取不到 `a` 本身。例如 `a` 的新值可能從暫存器直接傳給 `fprintf` 而沒有先存起來。（參考 [N1256 5.1.2.3p9](#)）

- 既然用 `printf` 除錯可能會失敗，是不是用除錯器就沒事？
否。改變順序通常是編譯器做的事情，所以不管用什麼工具除錯，運算順序已和原始碼中表面的順序不同。值得一提的是，不同的最佳化可能會導致運算順序不同，因此錯誤的程式不一定檢查得出來。只有正確的程式才保證不受實作或是最佳化影響。
- 如果我檢查編譯後的程式，確定運算順序跟我想的一樣，那代表實際執行時也一樣嗎？
否。例如許多現代處理器為了加速，不會按照順序執行程式，可以同時執行很多指令，也都會用很多層快取（`cache`）。真實的記憶體存取並不好預測，即使看了組語或機器碼也很難說準。（見 [Wikipedia: Out-of-order execution](#)）
- 如果我的程式要處理不同步的 `signal`，該怎麼阻止編譯器改變運算順序？
非常困難。除了標準少數的規定外，編譯器都能任意改變運算順序或讓運算交錯運行；`signal` 很有可能在變數存取到一半時進來。唯一可能可以解套的方法是使用 `volatile sig_atomic_t` 等型態，但實作還是有很多自由；請務必參考實作文件確認 `volatile` 是你想要的意思。不同實作可能會另外提供自己的方法阻止運算順序被改變。新標準 C1X 可能會提供更多控制機制。（參考 [glibc 文件 24.4.7 Atomic Data Access and Signal Handling](#) 看實際上會錯誤的例子，[Wikipedia: Memory barrier](#) 討論如何防止運算順序被改變，[gcc 文件 6.40 When is a Volatile Object Accessed?](#)，[N1256 5.1.2.3p9](#)，[6.7.3p6](#)，[7.14.1.1p5](#)）

運算順序（二）表達式

-
- 以下討論的是抽象機器的行為。實作不用一直保持抽象機器和實際機器之間的對應。

-
- ```
int i = f() + g();
```

  
哪個函式會先被呼叫呢？  
都有可能。（參考 [N1256 6.5p3](#)）

- ```
int i = i++;
```


結果是原來的值還是後來的值呢？
這是嚴重的錯誤，編譯器不受任何限制，包括格式化你的硬碟。因為在沒有循序同步點（`sequential points`）¹¹保護的情況下更改 `i` 兩次。（參考 [N1256 6.5p2](#)）

¹¹這是 C99 (N1256) 的術語。C99 用循序同步點（`sequential points`）來規範抽象機器的副作用執行順序。如果兩個副作用之間有一個同步點隔開，則保證前面的副作用會先完成，後面的副作用才開始。新的 C 標準 C1X 改用別的方法描述運算順序。

- `a[i] = i++;`
 結果會存到那一格。
 這是嚴重的錯誤，任何事情都有可能發生。`a[i]` 中讀取 `i` 的原值，並不是為了計算 `i` 的新值，違反 C99 (N1256) 的規定。(參考 N1256 6.5p2)
- `f(i++, i++)` 會怎麼呼叫？
 這是嚴重的錯誤，什麼事情都有可能發生。(參考 N1256 6.5p2)

結構與聯合

- 獨立宣告的 `int` 和 `struct` 內的 `int` 可以有不同對齊要求嗎？
 有可能不同。對齊要求不一定在所有地方都一樣。(參考 C Defect Report #074)
- `struct {int a; int b;}` 和 `int[2]` 是不是一樣大呢？`a` 和 `b` 在記憶體中是連續的嗎？
 都不一定。`a` 和 `b` 中間或是 `b` 之後都可能有空白。標準沒有規定 `struct` 一定要用所謂「最有效率」的擺法。(參考 N1256 6.7.2.1p13,15 及 C Defect Report #074)
- `struct {int a;}` 和 `int` 對整要求一樣嗎？
 不保證。例如實作可能可以讓 `struct {int a;}` 有更嚴格的對齊要求。(參考 C Defect Report #074)
- `union {int a;}` 或 `struct {int a;}` 跟 `int` 一樣大嗎？
 不一定。`union` 和 `struct` 尾巴都可以多塞空白。(參考 N1256 6.7.2.1p15)

- 以下用法會有問題嗎？

```

struct string {
    size_t length;
    char data[0];
};
struct string* str =
    (struct string*) malloc(sizeof(struct string) + 100);
str->length = 100;
```

首先，不管是 C99 (N1256) 或 C++03 都不可以宣告大小為 0 的陣列，以上的用法僅在特定的編譯器上可行。然而 C99 (N1256) 特別允許 `struct` 的最後一個成員是大小未知的陣列 (incomplete array type)，提供一個可攜安全的寫法如下：

```

struct string {
    size_t length;
    char data[];
};
```

C99 保證當一個 `struct string` 指標指向一個更大的空間 (以上例來說，`str` 指向一個比 `sizeof(struct string)` 還要多 100 位元組的空間) 時，此結構尾端的特殊

陣列可以被視為「讓此結構的大小不超過可用空間的最大陣列」，以上例來說 `data` 可以被視為有 100 個元素的 `char` 陣列。但除此之外並不保證這個結構的擺法與寫死 `data` 大小的結構是相同的：

```
struct string2 {
    size_t length;
    char data[100]; // 這個結構成員的位置和 struct string 不一定相同
};
```

(參考 [gcc 文件 6.17 Arrays of Length Zero](#) 以及 [N1256 6.7.2.1p18](#))

- ```
struct t1 { int m; } a;
struct t2 { int m; } *b;
b = (struct t2*) &a;
```

可以如此用 `b` 去存取 `a` 的內容嗎？

不保證。標籤 `t1` 和 `t2` 不同，所以兩個結構型態既不相同也不兼容。就算內容看起來一樣也不是。實際上最嚴重的問題是，編譯器可以假設 `b` 指到的物件和 `a` 不同而任意改變運算順序，做超乎想像的最佳化，例如

```
b->m = 0;
a.m = 1;
```

最後 `a.m` 不保證是 1. ( 見此問題，[N1256 6.2.7p1](#) )


- ( 承上 ) 可是我記得標準說如果擺在 `union` 裡面，使用而且開頭的部份是相通的，則可以自由換型態存取啊？所以應該只有一種實作方法？  
這是一條特赦條款，有很多使用條件，要全部符合才保證可攜。沒有把握請不要用。不同結構即使有相同的成員，還是可以有不同的擺法，但在這裡不是重點，畢竟幾乎所有已知的編譯器都會用同樣的擺法。這條規則實際上的效果是阻止編譯器假設不同型態的存取 ( 如上題的 `a` 和 `b` ) 代表不同物件，進而做超乎想像的最佳化，把計算順序任意更動。合適的 `union` 宣告一定要放在惡搞用的程式碼前面，因為傳統上 C 盡量讓編譯器只需要掃過程式碼一次。( 參考此問題，[N1256 6.5.2.3p5](#), [New C Standard 6.5.2.3](#) 的註解，[C Defect Report #236](#) 和 [#257](#) )

- 假如有一全域變數寫

```
union {
 char c;
 int i;
} u;
```

其中 `u` 的記憶體會全部清空嗎？

否。首先通常全域變數沒有指定初始值時，值會清成零或是變成空的，但不代表底下的記憶體 ( 表示法 ) 變成零 ( 如空指標不一定是零 )。 `union` 還有一個陷阱，就是只有第一個成員的值保證會清成零或是變成空的；因為不同型態的「零」可能底下有不同的表示法，記憶體 ( 表示法 ) 全部清成零不一定有意義。( 參考此問答，[N1256 6.7.8p10](#) )

-  下面的程式碼合法嗎？

```
union {
 int i;
 short s;
} u;
int *pi = &u.i;
short *ps = &u.s;
*pi = 0;
*ps = 1;
```

-  一團混亂的 union: [C Defect Report #283](#) 允許 type punning.
-  爭議超大的 [C Defect Report #236](#) : Mak 提出的 [N1111](#) ( 應該是未來標準改版方向 ) , [2005 年 4 月會議紀錄](#) , [2005 年 9 月會議紀錄](#) , [2006 年 3 月會議紀錄](#)。

## 字元 ( 一 ) 表示法與位元組

- char 到底是有號還是無號？  
標準並未規定 char 有號或無號。事實上 char、signed char 以及 unsigned char 是三個不同的型態。雖然標準規定 char 的表示方法、值範圍及行為必需和 signed char 或是 unsigned char 其中之一相同 ( 哪一個相同則交由實作決定 ) , 但依然是三個不同的型態。 ( 參考 [N1256 註腳 35](#) )
- 一個 char 一定是 8 個位元嗎？  
不是。C 語言標準裡面的 char 不一定是 8 個位元，但至少要 8 個位元才符合標準。  
( 參考 [N1256 5.2.4.2.1p1](#) 或是 [Paul Dixon 的說明](#) )
- 難道有機器一個 char 不是 8 個位元？  
有！至少世界上有機器是 16 位元或是 32 位元。 ( 參考 [Jack Klein 舉的例子](#) )
- 如果一個 char 是 16 位元，這代表一個 char 佔兩個位元組 ( bytes ) 嗎？  
不是！在 C 語言標準裡面 char 和位元組幾乎是同義詞。如果某實作決定讓 char 佔了 128 個位元，代表虛擬機器上的位元組就是 128 個位元。除了特例 bit field 外，char 是最終記憶體單位，其大小 ( sizeof ) 永遠是 1 不會改變。標準另外制定了型態 wchar\_t 處理寬字元。 ( 參考 [clc FAQ 8.10](#) )
- ```
char c;  
while ((c=getchar())!=EOF) { /* ... */ }
```


上述程式碼有什麼問題？
getchar 的傳回值是 int, 最明顯的問題是 char 可能裝不下 EOF. (請繼續閱讀下一

題) (參考只講一半的 [clc FAQ 12.1](#))

- 那改成這樣就沒問題了嗎？

```
int c;
while ((c=getchar())!=EOF) { /* ... */ }
```

否，尤其當 `int` 只佔一個位元組時。這是 C 標準的罩門，其他類似函式 (如 `fgetc`) 也有同樣的問題。原因跟上一題相反，就是 `int` 可能裝不下所有字元外加 `EOF`。似乎沒有什麼簡單的可攜寫法適用所有平台，但可以檢查 `INT_MAX` 和 `UCHAR_MAX` 的大小預先察覺問題。(參考 [comp.std.c 上的討論1 討論2 討論3](#), [C Defect Report 171](#) 和 [comp.unix.programmer 提及的真實例子](#))

- `EOF` 一定是 -1 嗎？

否。但是 `EOF` 一定是負的。建議直接跟 `EOF` 比較而不要去管正負。¹² (參考 [N1256 7.19.1p3](#))

字元 (二) 編碼

- C 語言用的編碼都是用 ASCII 吧？

不一定。C 語言標準並沒有規定字元的編碼要是 ASCII。C99 (N1256) 只有列舉一些基本字元，並說他們的編碼一定要可以塞進一個字元。(參考 [N1256 5.2.1](#))

- 難道實際上有電腦不是用 ASCII？

有的。例如有些古早的電腦用 ISO 646，然後有些電腦用 EBCDIC。所以 C 語言標準才制定 `trigraph`。(參考比較好讀的 [Wikipedia: Digraphs and trigraphs](#))

- 'a' 到 'z' 或 'A' 到 'Z' 是連續的嗎？

不保證。實際例子請參考 EBCDIC 編碼。

- 如果用大五碼 (Big-5) 代表標準字元集中的編碼和 ASCII 相同嗎？

嚴格來說不保證。大五碼 (收錄於 CNS11643 的版本) 只有規定中文的雙位元組編碼，沒有明確規定其他部份。(參考 [中華民國政府對於大五碼 \(Big5-2003 \) 的說明](#))

浮點數

- 浮點數的格式是什麼？

沒有強制規定。可以檢查 `__STDC_IEC_559__` 是否有定義，有的話理論上要遵照 IEC 60559:1989 標準 (和 ANSI/IEEE 754-1985 相同)。相較於整數，浮點數在現今機器中仍相當混亂，使用時請格外小心。(參考 [N1256 Annex F](#))

- 浮點數滿足 IEC 60559 標準 (有定義 `__STDC_IEC_559__`) 是什麼意思？

`float` 和 `double` 與 IEC 60559 定義的單精倍浮點數和雙精倍浮點數吻合 (`long`

¹²例如寬字元用的 `WEOF` 就不一定是負的。記正負號恐怕只是浪費腦力。

`double` 沒有強制規定要變成特定哪一種)。(參考 [N1256 Annex F.2](#))

- 假設 `a, b, c, d` 型態都是 `float`. 以下三行

```
a = b;  
a += c;  
a += d;
```

和這一行

```
a = b + c + d;
```

結果一樣嗎？

不保證。例如計算 `b + c + d` 的途中可能會使用精準度更高的浮點數，和每次都要儲存到 `a` 而喪失額外精準度的作法可能不一樣。(參考 [N1256 5.2.4.2.2p8](#))

- (承上) 那編譯器允許把那三行最佳化成最後一行嗎？

理論上不行，但很有可能你用的編譯器不遵守標準。在某些機器中，使用機器內建的浮點數指令，並讓數字停留在專屬暫存器上可能最快；若要完全符合標準，則計算時要故意把額外的精準度拿掉，可能速度會慢很多。(參考 [N1256 5.2.4.2.2p8](#), [comp.std.c](#) 上的討論)

- 假設浮點數滿足 IEC 60559 標準，`a` 的型態是浮點數，`a==a` 的結果永遠是真嗎？

否。如果 `a` 是 NaN 結果為假。



- 假設浮點數滿足 IEC 60559 標準，下面這兩行

```
if (a < b) f();  
else g();
```

和這兩行

```
if (a >= b) g();  
else f();
```

等價嗎？

否。主要還是 NaN 的問題。如果程式需要很多浮點數運算，建議閱讀更多關於 NaN 的資料。

- 如何在 `[0,1]` 之間隨機選一個實數？

不，你不行。`0` 到 `1` 之間的實數有不可數無窮多個，比所有整數都還多，更別提一般來說只有有限精確位的浮點數。即使你滿足於有限位數，浮點數通常越靠近零時絕對誤差就越小，能表示的數字並不是 `[0,1]` 之間等間距分佈。

- 如何把浮點數轉成最接近的整數？

C99 (N1256) 可以使用 `lround` 系列。c9c FAQ 說可以使用以下這行：

```
int xi = (int)(x < 0 ? x - 0.5 : x + 0.5);
```

但本文作者覺得 `lround` 系列可靠許多。([c9c FAQ 14.6](#), [N1256 7.12.9.7](#))

main 函式

- main 函式可以遞迴嗎？
C++03 明確規定不行 (參考 C++03 3.6.1p3 及 5.2.2p9) C 沒有明講，所以應該可以。
- main 傳回值可以寫 void 嗎？
這並非標準保證的格式。C 允許實作自己增加新的形式，但世界上有實作會壞掉，請務必再三確認實作的文件。C++ 要求傳回值是 int, 所以一定不可能。寫標準保證的形式絕對沒問題。(參考 [clc FAQ 11.14b](#) 看實際上會壞掉的例子，[N1256 5.1.2.2.1p1](#) 以及 C++03 3.6.1p2)
- main 可以不寫傳回值也不用 exit 嗎？
C99 (N1256) 和 C++03 可以，但為了最大可攜性，請加上 `return 0;` 或是 `return EXIT_SUCCESS;` 因為以前的標準 (C90) 採用不同的規定。手動加上傳回值，不管在哪个版本的標準都是可行的。([N1256 5.1.2.2.3](#), C++03 3.6.1p5)

編譯器的檢查

- 編譯器沒有出現警告，就應該沒事情吧？如果我有寫錯，不是應該要警告我？
編譯器是因為好心才提醒你。即使沒有提醒你，你的程式還是可能有無法預期的行為。某些不可攜的程式碼¹³標準甚至允許編譯器隨便惡搞，包括產生一個引爆核彈的程式。如要寫不可攜的程式碼，務必參考實作文件，確認在該平台上沒有問題。
- 我還有加上 `-Wall -Wextra -Wxxx -Wyyy -Wzzz` (gcc) 也沒有出現警告啊？
同上一個問題。即使所有警告參數全開，目前作者也還沒發現任何可以保證安全的實作。
- 我還有用 Purity, Valgrind, CCured 等等軟體，都沒有發現問題。
雖然有檢查比沒檢查好，但上述程式沒有一個能保證你的程式是安全的。如果這些檢查工具夠用，這篇文章也不用寫了！

勘誤與重大更動

- 2011/9/14
嘗試以 Lisp Machine 為基礎為某些問題提供更多真實反例。
- 2011/9/13
之前版本的「等級比 unsigned int 高」應該寫成「等級不輸給 unsigned int」。相關內容已全部改寫。(所幸這錯誤沒有推薦不安全的東西。)
- 2011/9/9

¹³嚴格來說，這裡是指「未定義行為」(代表語言標準對實作沒有任何要求)。當然語言標準不是唯一的標準；如果其他標準有規範，實作必須同時滿足所有標準。

增加兩個關於全域變數初始化的問答。

- 2011/9/7
`a[a[0]] = 0;` 和 `node->next->next = node;` (實作循環串列時) 兩用法雖然在 C99 (N1256) 可能不可攜, 但由於此寫法相當合理, 沒有任何已知實作會錯誤, 而且新標準 (C1X) 可能會允許, 所以暫時刪除。(參考 [N1256 6.5p2](#) 及 [comp.std.c 討論](#))
- 2011/9/6
之前推薦適合位元運算的整數型態時, 忽略了標準另一個陷阱 (整數型態自動轉型), 現在已經修正。陷阱是如果原本用的無號型態能表示的範圍太小, 有可能被自動轉成 `int` 而成為有號整數型態。(參考 [N1256 6.3.1.1p2](#))
此外原本還有另一個問答提及 $(a^b)^b$ 不安全, 由於牽涉標準模糊地帶, 因此先暫時刪除。(參考 [comp.lang.c 討論](#) 和 [comp.std.c 討論](#))

感謝



感謝許許多多朋友的支持與意見。