程式語言一　講義

指標與記憶體模型

Hanly《C 語言詳論》6e

§ 1　名詞速查表

中文	English	一句話定義	科目	章節
指標	pointer	儲存另一個變數記憶體位址的變數	程式語言一	指標與記憶體
取址運算子	address-of operator	`&` 取得變數所在的記憶體位址	程式語言一	指標與記憶體
解參考	dereference	`*` 透過指標取得或寫入它所指位址的值	程式語言一	指標與記憶體
空指標	null pointer	不指向任何有效物件的指標，值為 `NULL`	程式語言一	指標與記憶體
野指標	wild pointer	未初始化、指向不確定位址的指標	程式語言一	指標與記憶體
懸空指標	dangling pointer	指向已釋放或已失效記憶體的指標	程式語言一	指標與記憶體
記憶體洩漏	memory leak	配置後失去所有指向、無法再釋放的堆積記憶體	程式語言一	指標與記憶體
堆疊	stack	存放區域變數與函式呼叫框，進出函式時自動配置與回收	程式語言一	指標與記憶體
堆積	heap	動態配置區，生命週期由程式設計師以 `malloc`／`free` 管理	程式語言一	指標與記憶體
動態配置	dynamic allocation	執行期向堆積要記憶體（`malloc`／`calloc`／`realloc`）	程式語言一	指標與記憶體
指標算術	pointer arithmetic	指標加減以「所指型別大小」為單位移動位址	程式語言一	指標與記憶體

§ 2 核心概念

核心概念

指標本身是一個變數，它存的內容是一個記憶體位址。宣告 `int *p;` 讀作「`p` 是指向 `int` 的指標」：`p` 的值是某個位址，`*p` 是那個位址裡存的整數。

位址與值是兩個不同層次。`&x` 是「x 在哪裡」，`x` 是「x 是多少」；對指標而言 `p` 是「指向哪裡」，`*p` 是「那裡是多少」。把這兩層混用，是初學指標最主要的錯誤來源。

型別決定兩件事，缺一不可：解參考時要讀幾個 byte（`int*` 讀 4 個、`char*` 讀 1 個），以及指標算術一步跨幾個 byte。所以指標一定要帶型別，不是隨便存個數字而已。

§ 3 主要內容

3.1 記憶體是有編號的格子

把記憶體想成一排有編號（位址）的格子，每格存 1 個 byte。變數是「被取了名字的一段格子」：`int x = 42;` 佔連續 4 個 byte，編譯器記住「x 從某位址開始」。指標就是「內容剛好是另一段格子起始編號」的變數。

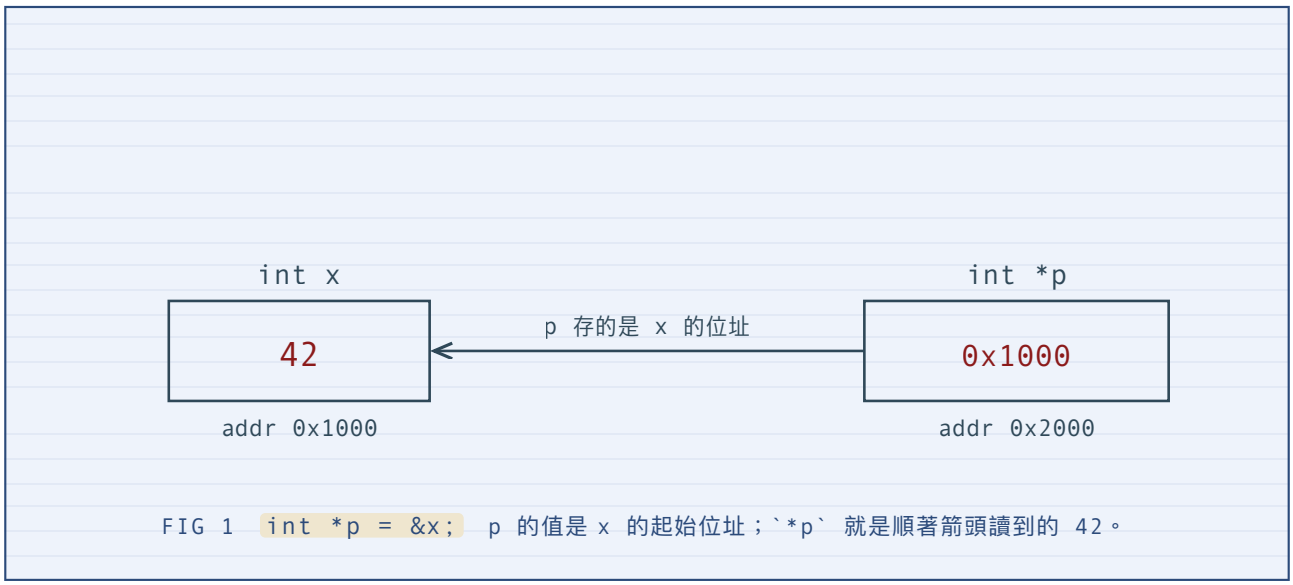

FIG 1 `int *p = &x;` `p` 的值是 `x` 的起始位址；`*p` 就是順著箭頭讀到的 `42`。

3.2 三個基本動作：取址、宣告、解參考

```c
int  x = 42;      // 一般變數
int *p = &x;      // p 指向 x：& 取出 x 的位址
printf("%d\n", *p);   // 解參考：印出 42
*p = *p + 1;      // 透過 p 改寫 x，x 變成 43
printf("%d\n", x);    // 印出 43
```

`&x` 給出位址、`*p` 順著位址取值。`*p = ...` 是「寫回」那個位址，等同改了 `x` 本人，這是指標能「從遠端改變數」的關鍵。

3.3 型別與指標算術

指標加 1，不是位址加 1 個 byte，而是加「一個所指型別的大小」。

```c
int  a = 0;  int  *pi = &a;   // pi + 1 位址 +4 (sizeof(int))
char c = 0;  char *pc = &c;   // pc + 1 位址 +1 (sizeof(char))
```

這條規則讓「指標 + 整數」剛好走到下一個同型別元素，正是陣列走訪的基礎。

3.4 指標與陣列

陣列名在大多數場合會「退化」成指向首元素的指標。因此 `a`、`&a[0]`、`a+0` 三者位址相同，且 `a[i]` 與 `*(a + i)` 完全等價。

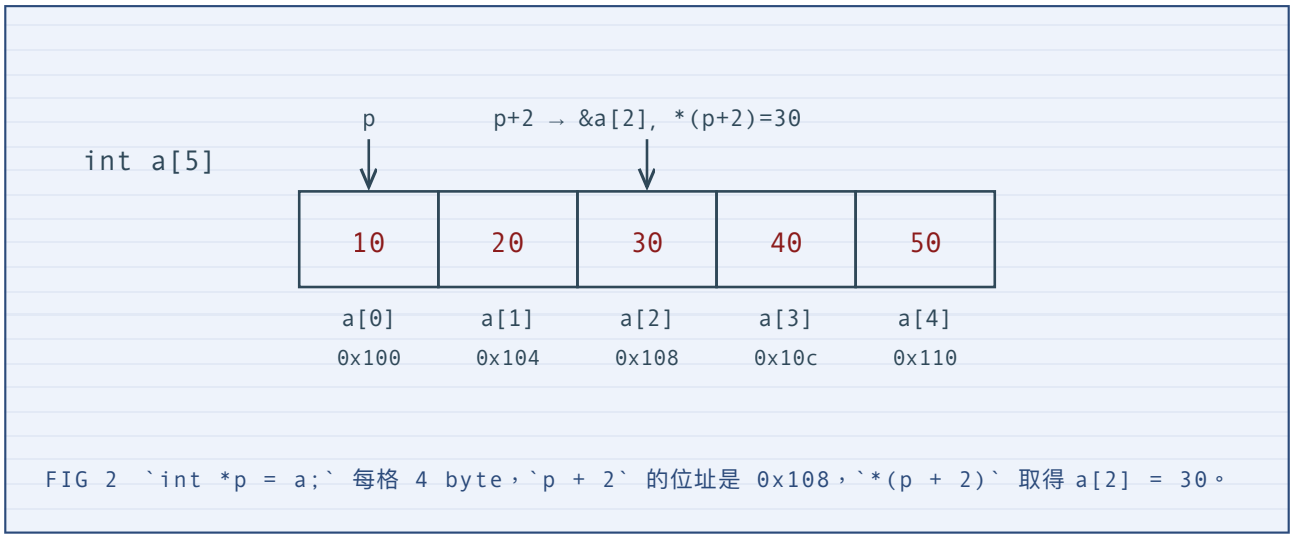

FIG 2 `int *p = a;` 每格 4 byte，`p + 2` 的位址是 0x108，`*(p + 2)` 取得 a[2] = 30。

退化也有代價：陣列一旦傳進函式，參數收到的是指標，`sizeof` 量到的是指標大小（8 byte）而非整個陣列。所以陣列傳參數通常要「另外傳長度」。

3.5 指標當函式參數：為什麼 swap 要傳位址

C 的參數傳遞是「傳值」（call by value）：函式拿到的是引數的複本，改複本動不到原本。要改呼叫端的變數，就把它的「位址」傳進去，函式再透過指標寫回。

```
void swap(int *a, int *b) {    // 收兩個位址
    int t = *a;                // 讀 a 指向的值
    *a = *b;                   // 寫回
    *b = t;
}
int x = 3, y = 5;
swap(&x, &y);                  // 傳位址，呼叫後 x=5, y=3
```

若寫成 `void swap(int a, int b)` 只交換了複本，呼叫端 `x`、`y` 不變。這也是 `scanf("%d", &x)` 要加 `&` 的同一個理由：scanf 必須拿到位址才能把讀到的值寫進 `x`。

3.6 堆疊與堆積：兩種記憶體生命週期

區域變數放在**堆疊（stack）**：進入函式自動配置、離開自動回收，生命週期跟著 `{ }`。要在函式結束後還活著、或大小到執行期才知道的記憶體，得向**堆積（heap）**動態配置，並自己負責釋放。

```
int  n = 5;
int *arr = malloc(n * sizeof(int));   // 向 heap 要 n 個 int
if (arr == NULL) { /* 配置失敗，處理錯誤 */ }
for (int i = 0; i < n; i++) arr[i] = i * 10;   // 當成陣列用
free(arr);      // 用完釋放
arr = NULL;     // 釋放後立刻設 NULL，避免懸空
```

`malloc(位元組數)` 回傳 `void*`（失敗回 `NULL`，必須檢查）；`calloc(個數, 每個大小)` 會把記憶體歸零；`realloc(舊指標, 新大小)` 調整既有區塊大小。配置與釋放一定成對。

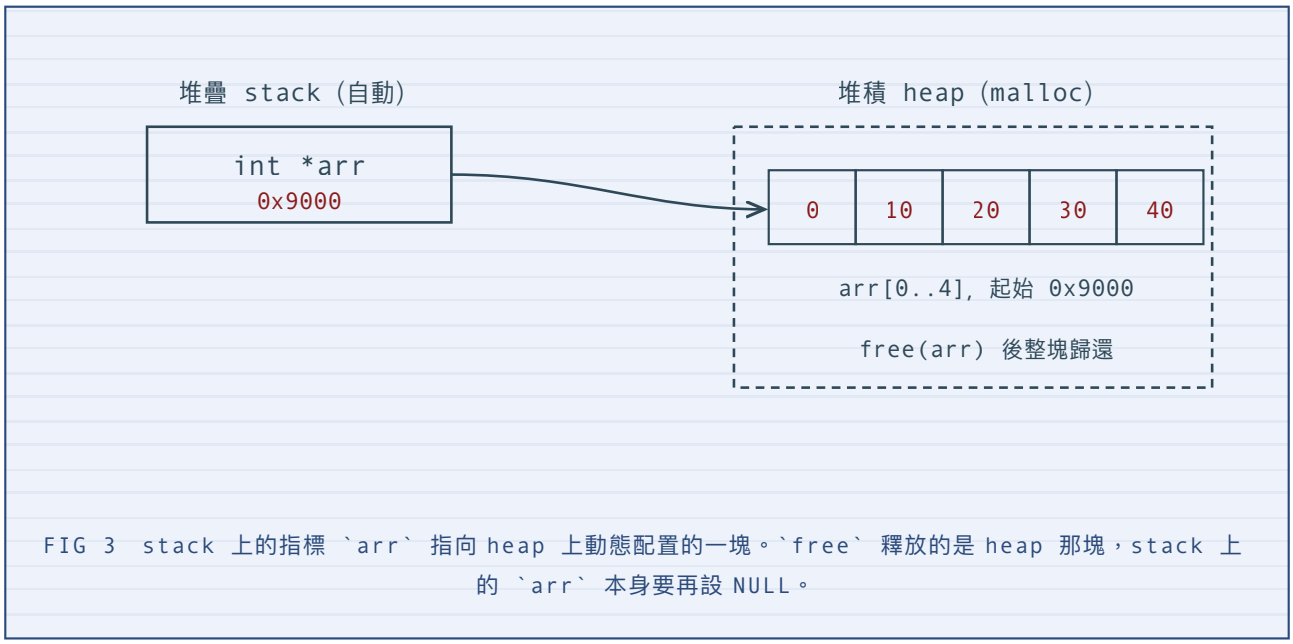

FIG 3 stack 上的指標 `arr` 指向 heap 上動態配置的一塊。`free` 釋放的是 heap 那塊，stack 上的 `arr` 本身要再設 NULL。

3.7 三種記憶體錯誤（生命週期沒對齊）

- **懸空指標—回傳區域變數位址**：函式回傳後該變數的 stack 空間已回收，拿到的位址無效。
- **懸空指標—釋放後再用（use-after-free）／重複釋放（double free）**：`free` 後該指標已失效，再讀寫或再 `free` 都是未定義行為。
- **記憶體洩漏**：`malloc` 後弄丟了唯一指向它的指標（覆寫或變數離開作用域），那塊 heap 再也 `free` 不到。

這三種都屬「未定義行為（undefined behavior, UB）」，特性是**有時剛好不崩、有時才崩**，所以不能靠「跑一次沒當就沒事」判斷。要用 `clang -fsanitize=address,undefined` 編譯執行，問題才會穩定現形。

§ 4 語法與函式速查

```c
&x              // 取 x 的位址
int *p;         // 宣告指向 int 的指標
p = &x;         // 指向 x
*p              // 解參考：p 所指位址的值
p = NULL;       // 空指標（不指向任何有效物件）

// 動態記憶體（需 #include <stdlib.h>）
void *malloc(size_t size);              // 配置，內容未初始化，失敗回 NULL
void *calloc(size_t n, size_t size);    // 配置 n 個並歸零
void *realloc(void *p, size_t newsize); // 調整既有區塊大小
void  free(void *p);                    // 釋放（之後把指標設 NULL）
```

寫法	意思
`p`	指標的值（一個位址）
`*p`	p 指向位址裡的值
`&x`	x 的位址
`p + i`	位址前移 `i * sizeof(*p)` 個 byte
`a[i]`	等同 `*(a + i)`

§ 5 常見錯誤

常見錯誤

- 未初始化指標就解參考（野指標）：宣告時先設 `NULL` 或立刻指向有效物件。
- 位址與值混用：`scanf("%d", &x)` 漏掉 `&`；或對已是值的東西再多一層 `*`。
- 回傳區域變數的位址（懸空指標）：改成回傳值、或由呼叫端提供緩衝區、或用 `malloc`。
- `free` 後繼續使用（use-after-free）或重複 `free`（double free）：釋放後立刻 `arr = NULL;`。
- 忘記 `free`（記憶體洩漏）：每個 `malloc` 都要有對應的 `free`。
- 指標算術忘了單位是 `sizeof(*p)`，誤以為是 1 個 byte。
- 陣列傳進函式後用 `sizeof` 期待整個陣列大小，實際量到的是指標大小。
- `malloc` 沒檢查回傳值是否為 `NULL` 就直接用。

§ 6 練習題

練習 1（一般題）：預測輸出

> **引導步驟**
>
> 1. 畫出 `x` 與 `p` 兩格，標上值與箭頭。
> 2. 逐行更新：`*p = 20` 改的是哪一格？
> 3. 印出時 `x` 與 `*p` 各讀哪裡？

```c
int x = 10;
int *p = &x;
*p = 20;
printf("%d %d\n", x, *p);
```

解答

輸出 `20 20`。`p` 指向 `x`，`*p = 20` 等於把 `x` 改成 20；之後 `x` 與 `*p` 讀的是同一格，都是 20。

> **易錯**
>
> · 以為 `*p = 20` 只改了 `p` 不影響 `x`；其實 `*p` 就是 `x` 本人。

練習 2（重要題）：手寫 swap

> **引導步驟**
>
> 1. 想清楚：要改呼叫端的 `x`、`y`，參數該收值還是收位址？
> 2. 函式內用暫存變數，透過 `*` 讀寫指向的值。
> 3. 呼叫時引數要加什麼？

寫一個 `swap` 函式，使呼叫 `swap(&x, &y)` 後 `x`、`y` 的值互換。

解答

```c
void swap(int *a, int *b) {
    int t = *a;
    *a = *b;
    *b = t;
}
/* 呼叫：swap(&x, &y); */
```

參數收「位址」，函式才改得到呼叫端的變數。若參數寫成 `int a, int b`，只交換了複本，原變數不變。

易錯

- 參數寫成傳值，交換無效。
- 呼叫時忘了加 `&`，把值當位址傳。

練習 3（重要題）：抓 bug

> **引導步驟**
>
> 1. `make_array` 回傳後，`local` 還活著嗎？
> 2. 它的記憶體在 stack 還是 heap？回收時機是？
> 3. 怎麼改才安全（兩種以上做法）？

下列程式為何危險？指出問題並提出修法。

```
int *make_array(void) {
    int local[3] = {1, 2, 3};
    return local;        // ?
}
```

解答

`local` 在 stack 上，函式回傳時其空間即回收，回傳的位址成為**懸空指標**，呼叫端再讀寫是未定義行為。修法：

```
int *make_array(void) {
    int *a = malloc(3 * sizeof(int));   // 改放 heap
    if (a == NULL) return NULL;
    a[0] = 1; a[1] = 2; a[2] = 3;
    return a;            // 呼叫端用完負責 free
}
```

或由呼叫端先配好陣列、把位址傳進函式填值（呼叫端持有生命週期）。

易錯

- 以為「跑起來沒當」就沒問題；UB 可能時好時壞，需 `-fsanitize=address` 才穩定現形。
- 改用 `malloc` 後忘了在呼叫端 `free`，變成洩漏。

練習 4（一般題）：指標算術

引導步驟

1. `p` 一開始指向哪一格、位址多少？
2. `p + 3` 位址前移幾個 byte？（`sizeof(int) = 4`）
3. `*(p + 3)` 讀到第幾個元素？

設 `int a[5] = {10,20,30,40,50};` 且 `a` 起始位址為 `0x200`，`int *p = a;`。求 `p + 3` 的位址與 `*(p + 3)` 的值。

解答

`p + 3` 位址 = `0x200 + 3 * 4 = 0x20c`，指向 `a[3]`；`*(p + 3) = a[3] = 40`。

易錯

- 把 `p + 3` 當成位址只加 3 個 byte，忽略要乘 `sizeof(int)`。

§ 7 自我檢核

- [ ] 能用一句話說清 `&x`、`x`、`p`、`*p` 各是什麼。
- [ ] 能畫出 `int *p = &x;` 的記憶體圖並標出箭頭方向。
- [ ] 能解釋 C 是傳值，因此 swap 為何要傳位址。
- [ ] 能分辨 stack 與 heap、自動配置與動態配置的差別。
- [ ] 能寫出 `malloc`→檢查 NULL→使用→`free`→設 NULL 的完整流程。
- [ ] 能指出懸空指標的三種成因並各提一種修法。
- [ ] 能說明為何忘記 `free` 會造成洩漏。
- [ ] 能正確做指標算術（以 `sizeof(*p)` 為單位）。
- [ ] 知道記憶體錯誤屬 UB，要用 `-fsanitize=address,undefined` 驗證。