程式語言一　講義

函式

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
函式	function	把一段邏輯包起來、取名、重複使用	程式語言一	函式
參數	parameter	函式定義時的形式變數，如 `int add(int a, int b)` 的 a、b	程式語言一	函式
引數	argument	呼叫時實際傳進去的值，如 `add(3, 4)` 的 3、4	程式語言一	函式
回傳值	return value	函式用 `return` 傳回呼叫處的結果	程式語言一	函式
傳值	call by value	參數收到的是引數的**複本**，改複本動不到原本	程式語言一	函式
函式原型	prototype	先宣告函式（型態、名稱、參數），定義放後面	程式語言一	函式
遞迴	recursion	函式呼叫自己	程式語言一	函式
基底條件	base case	遞迴停止的條件；沒有它＝無窮遞迴	程式語言一	函式
呼叫堆疊	call stack	函式呼叫一層層堆疊的記憶體區	程式語言一	函式

§ 2 核心概念

核心概念

函式把重複的邏輯包起來、取個名字重用：寫一次、用很多次，修改只改一處；也把大問題拆成小問題（`is_prime(n)` 比一堆迴圈直覺）。`printf`、`scanf` 就是函式。語法：

```
回傳型態 函式名(參數列表) { ...; return 回傳值; }
int add(int a, int b) { return a + b; }
```

呼叫 `add(3, 4)` 的流程：① 暫停 `main` ② **把引數複製給參數**（3→a、4→b，傳值）③ 執行函式主體 ④ `return` 把結果傳回 ⑤ `main` 繼續。**傳值**：函式拿到的是複本，改它動不到呼叫端的變數（要改呼叫端得傳位址，見 [[指標與記憶體]]）。

§ 3 主要內容

3.1 定義、呼叫、void、無參數

```c
int add(int a, int b) { return a + b; }    // 回傳 int
void greet(int times) {                     // 不回傳值用 void
    for (int i = 0; i < times; i++) printf("Hi\n");
}
void print_line(void) { printf("----\n"); } // 不收參數寫 (void)
```

`void` =「做事但不算值」。回傳值用 `return 值;`；`void` 函式可寫 `return;`（不帶值）或省略。

3.2 函式原型（prototype）

函式可以**先宣告、後定義**。若定義放在 `main` 後面，要先在前面放原型，否則編譯器不認得：

```c
int add(int a, int b);    // 原型（宣告，記得分號）
int main(void){ printf("%d\n", add(3,4)); return 0; }
int add(int a, int b){ return a + b; }    // 定義
```

3.3 區域變數的作用域

函式裡宣告的變數是**區域的**：只在該函式內存在、函式結束就消失，不同函式的同名變數互不相干。

3.4 遞迴：函式呼叫自己

```
void countdown(int n) {
    if (n <= 0) { printf("發射\n"); return; }   // ① 基底條件：停！
    printf("%d\n", n);
    countdown(n - 1);                            // ② 遞迴步驟：問題變小
}
```

兩個必備要素：① **基底條件**（什麼時候停）② **遞迴步驟**（把問題縮小、每次更靠近基底）。**忘了基底條件＝無窮遞迴＝Stack Overflow 崩潰**，這是遞迴最常見的錯。

3.5 階乘：迴圈版 vs 遞迴版

```
int fact(int n){ if (n <= 1) return 1; return n * fact(n-1); }   // 遞迴版，直接翻譯數學定義
```

3.6 Call Stack：fact(4) 怎麼算

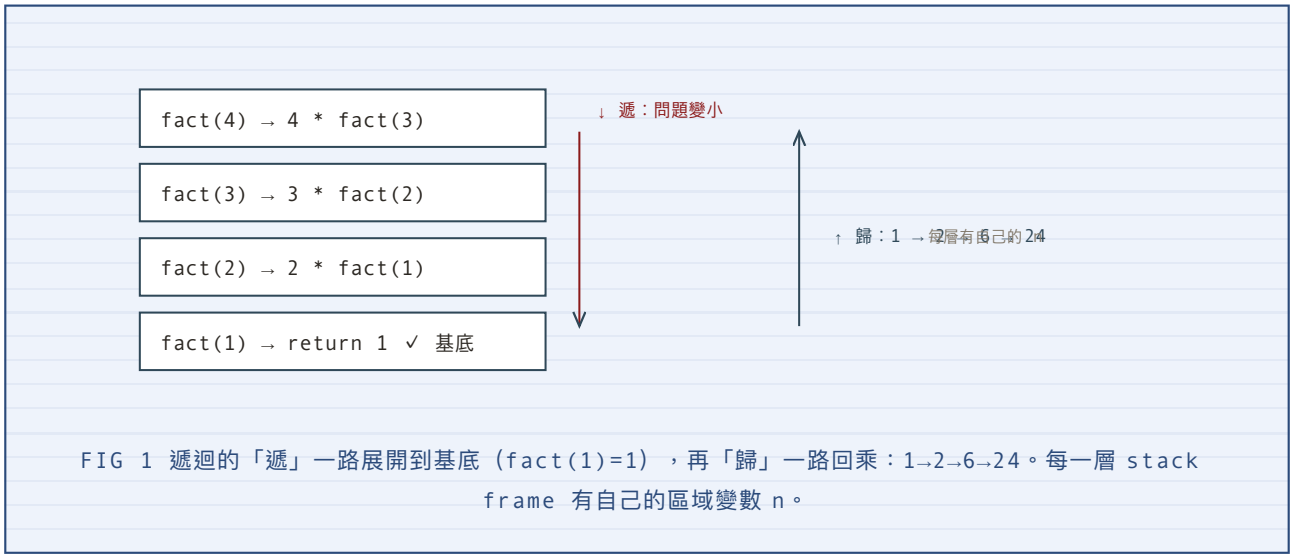

FIG 1 遞迴的「遞」一路展開到基底（fact(1)=1），再「歸」一路回乘：1→2→6→24。每一層 stack frame 有自己的區域變數 n。

3.7 費氏數列與重複計算

```
int fib(int n){ if (n <= 2) return 1; return fib(n-1) + fib(n-2); }
```

`fib(6) = fib(5)+fib(4) = 5+3 = 8`（序列 1,1,2,3,5,8）。缺點：純遞迴會**重複計算**同一個 `fib`，n 大時很慢（之後可用迴圈或記憶化改善）。

§ 4 語法與函式速查

```
回傳型態 名稱(參數列表) { ... return 值; }
int add(int a, int b){ return a+b; }     // 定義
int add(int a, int b);                   // 原型（宣告，要分號）
add(3, 4);                               // 呼叫（引數複製給參數）
void f(void){ ... }                      // 不回傳、不收參數
// 遞迴：基底條件 + 遞迴步驟（問題變小）
```

§ 5 常見錯誤

常見錯誤

- 函式有回傳型態卻忘了 `return`（或路徑沒都回傳）。
- 以為傳值能改呼叫端的變數（不行，那是複本；要改得傳位址）。
- 遞迴忘了基底條件 → 無窮遞迴 → Stack Overflow。
- 遞迴步驟沒讓問題變小（永遠到不了基底）。
- 函式原型漏分號，或定義放 main 後面又沒寫原型。
- `void` 函式卻 `return 一個值`；非 void 卻不回傳。

§ 6 練習題

練習 1（一般題）：遞迴追蹤

`sum(5)` 回傳多少？

```
int sum(int n){ if (n <= 0) return 0; return n + sum(n - 1); }
```

> **引導步驟**
>
> 1. 展開：`sum(5)=5+sum(4)=5+4+sum(3)=…`
> 2. 到基底 `sum(0)=0` 停。

解答

`15`。`5+4+3+2+1+0 = 15`（就是 1+2+⋯+5）。

練習 2（一般題）：傳值

下列印什麼？為什麼 `main` 的 `x` 沒變？

```
void inc(int v){ v = v + 1; }
int main(void){ int x = 5; inc(x); printf("%d\n", x); return 0; }
```

> **引導步驟**
>
> 1. `inc` 收到的 `v` 是 `x` 的複本還是本人？

解答

印 `5`。傳值：`v` 是 `x` 的複本，`v=v+1` 只改複本，`main` 的 `x` 不變。要改呼叫端得傳位址 `void inc(int *v){ *v += 1; }` 並 `inc(&x)`。

練習 3（重要題）：抓 bug

下列遞迴為何會 crash（Stack Overflow）？怎麼修？

```
int fact(int n){ return n * fact(n - 1); }
```

> **引導步驟**
>
> 1. 遞迴的兩個必備要素是什麼？這裡缺了哪個？
> 2. n 會不會停下來？

解答

缺**基底條件**：永遠呼叫自己、問題沒在某處停，無窮遞迴把 call stack 撐爆（Stack Overflow）。修：

```
int fact(int n){ if (n <= 1) return 1; return n * fact(n - 1); }
```

> **易錯**
>
> · 有遞迴步驟但忘了基底條件。

§ 7 自我檢核

- ☐ 會定義/呼叫函式，分得清參數與引數、回傳型態與 void。
- ☐ 懂呼叫流程是「傳值」：參數是引數的複本，改不到呼叫端。
- ☐ 會用函式原型（先宣告後定義），知道要分號。
- ☐ 知道區域變數的作用域只在該函式內。
- ☐ 懂遞迴的兩要素：基底條件 + 讓問題變小的遞迴步驟。
- ☐ 知道忘了基底條件會 Stack Overflow。
- ☐ 能追蹤 call stack（遞展開、歸回收），算 fact、sum、fib。