資料結構

流程圖

目的：

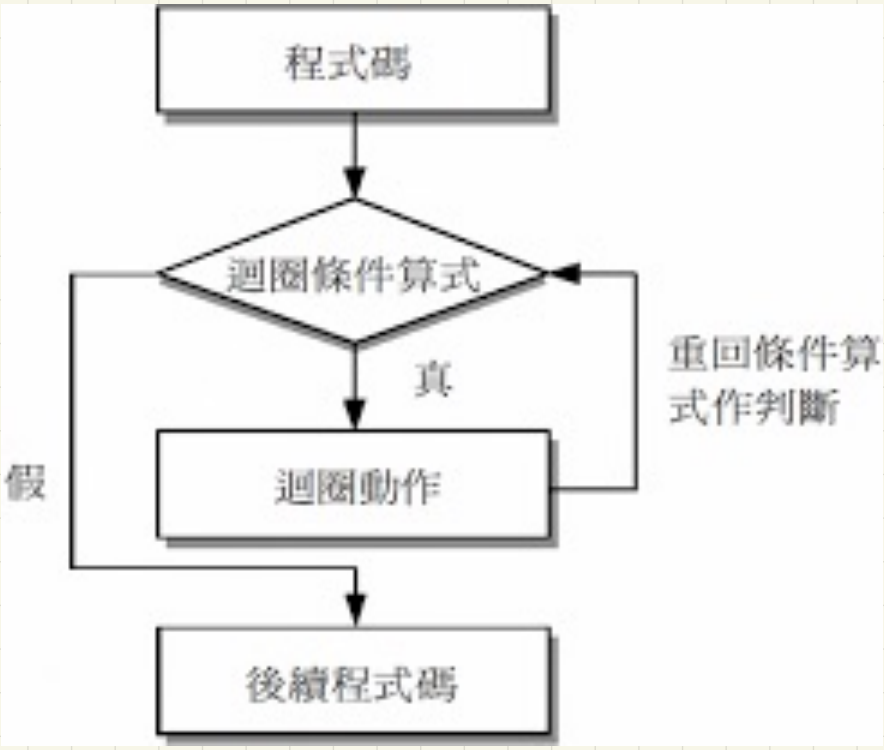
以圖像具象化想法，將繁複的文字敘述化成前顯易懂之圖表 — 視覺化。

基本符號：

起止 ⬭

流程 →

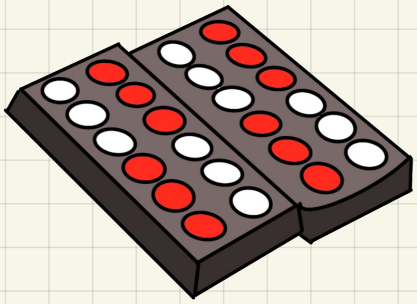
程序 □



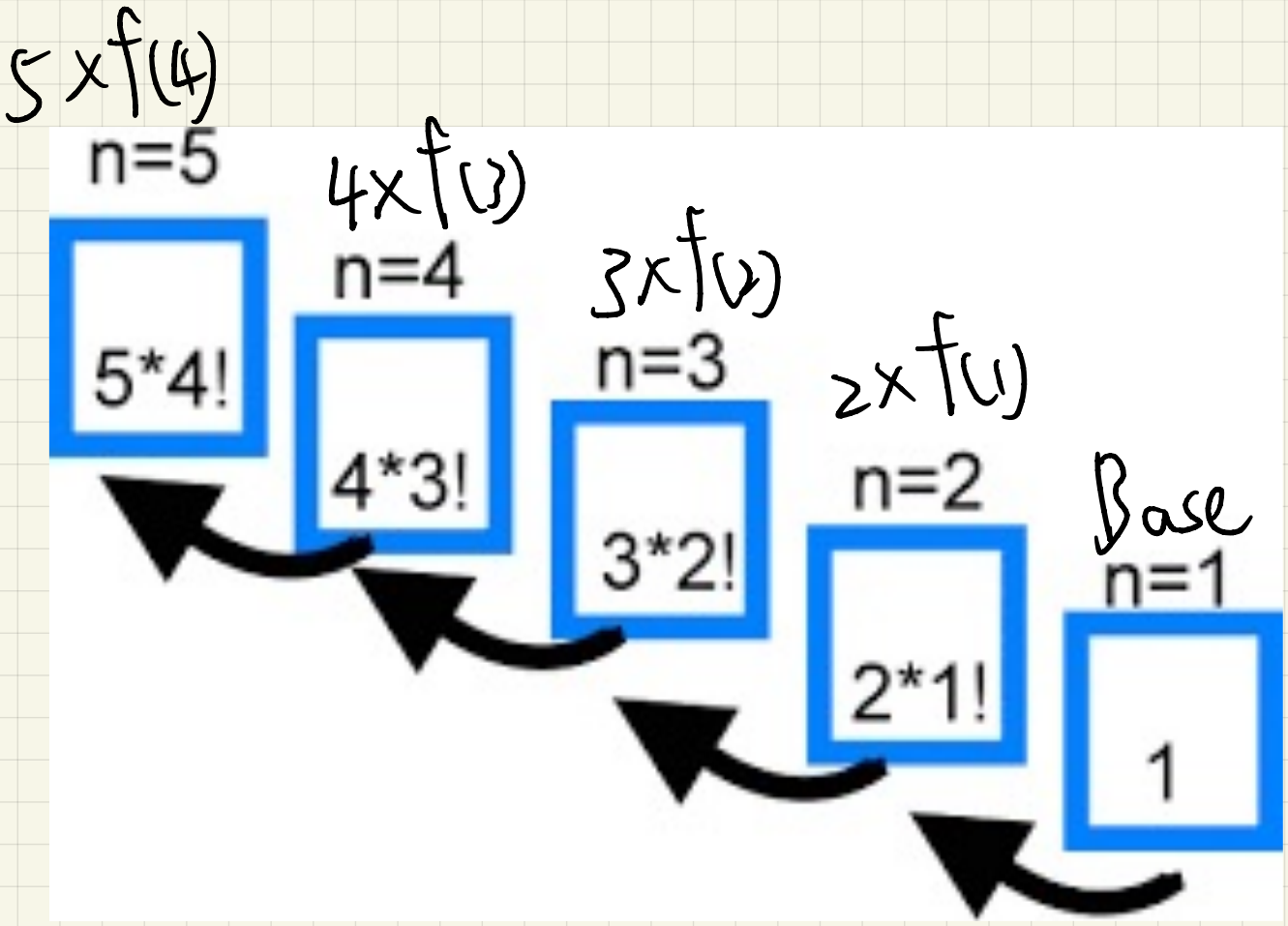
遞迴

目的：

將大問題拆成多個小問題，使程式重複使用特性能夠發揮到極致。



如果我們遇到了一個大問題，無法立即推演通式解，不妨試試看將大問題拆解成多個子問題，舉例來說我們無法得知費氏數列 f(50) 數值為何 ，但只要我們有了子問題的解決方法就有機會解出大問題 f(2) = f(1) + f(0) | f(3) = f(2) + f(1) | ..... | f(n) = f(n-1) + f(n-2) when n >= 2 and f(0) = 0 f(1) = 1 ，像是骨牌一樣。



檔案存取

目的：

在現實中程式不可能像超人一樣，一個檔案包山包海，我們會讓各個檔案各司其職，並利用我們的寫的智慧程式關聯需要的檔案。

(一)串流（或資料流）

C++ 語言提供 I/O 裝置（如：螢幕、鍵盤、記憶體、磁碟）的存取採串流方式，串流是一連串的資料，以 fstream類別將資料讀取或寫入檔案儲存，語法格式如下：

```
#include <fstream.h>                    //引用 <fstream.h> 標頭檔
fstream file;                           //宣告一個 fstream 物件
file open("Readme.txt",ios::in);        //以讀取模式開啟 Readme.txt 檔
```

(二) fstream 物件的 open 函數

fstream 物件的 open 函數中有兩個參數，一個是檔名，另一個是開啟檔案的模式參數，常用的模式參數如下：

模式參數	說明
ios::in	檔案開啟為讀取狀態
ios::out	檔案開啟為寫入狀態
ios::ate	從檔案結尾讀取及寫入資料
ios::app	將資料附加在檔案結尾
ios::trunc	如果檔案存在，就清除檔案內的資料

(三)常用的檔案處理函數

函數	說明
open(file,mode)	以 mode 模式開啟名為 file 的檔案
close()	關閉檔案
is_open()	檢查檔案是否為開啟狀態，若是則傳回 true，否則傳回 flase
eof()	判斷是否至檔案結尾
write(buffer,n)	將 buffer 陣列中 n 個字元寫入至檔案中

二、檔案讀取

常用的檔案讀取方式，如下：

read(buffer,n)	自目前位置至檔案結尾為止，讀取 n 個字元至 buffer 陣列
get(ch)	讀取一個字元至 ch 字元變數
getline(buffer,sizeof(buffer))	讀取一行至buffer陣列

後序式

目的：

人們平常使用的運算式，是將運算元放在運算子兩旁，例如 a + b / d 這樣的式子，這稱為中序（infix）表示式；然而電腦剖析運算式時，為了有效率地判斷運算的順序，可將中序表示式轉換為後序（postfix）或前序（prefix）表示式。

範例：

例如 (a + b) * (c + d)，依演算法的輸出過程如下：

元素	堆疊	輸出
(	(	-
a	(	a
+	(+	a
b	(+	ab
)	-	ab+
*	*	ab+
(	*(	ab+
c	*(	ab+c
+	*(+	ab+c
d	*(+	ab+cd
)	*	ab+cd+
-	-	ab+cd+*

資料抽象化

目的：

Abstraction 抽象化是指解決問題時，通常與引入相關事物；當描述這些事物時，我們通常僅專注與問題相關的部分，而忽略其他的細節，以免增加問題的難度或干擾解題者的思緒。

範例：

軟體工程中的抽象化
軟體工程特別強調模組化 (modularity) 概念，以便控制軟體發展時的複雜度，通常模組指的是 method 與 class，描述這些模組時，僅說明其規格而非實作的細節。

Functional Abstraction 功能抽象化
僅描述某個 method 提供哪些功能 (what the function does) ，而非此 method 如何實現這些功能 (how the function does)。

Data Abstraction 資料抽象化
描述你可以如何操作一組資料，而非如何實作這些操作與資料儲存的方式。實踐的方式通常是經由系統分析階段去產生 Abstract Data Types (ADT) 抽象資料型態。

Abstract Data Types (ADT) 抽象資料型別
是一堆資料 (data) 的集合，和一組可以在那堆資料上執行的操作 (operation)。

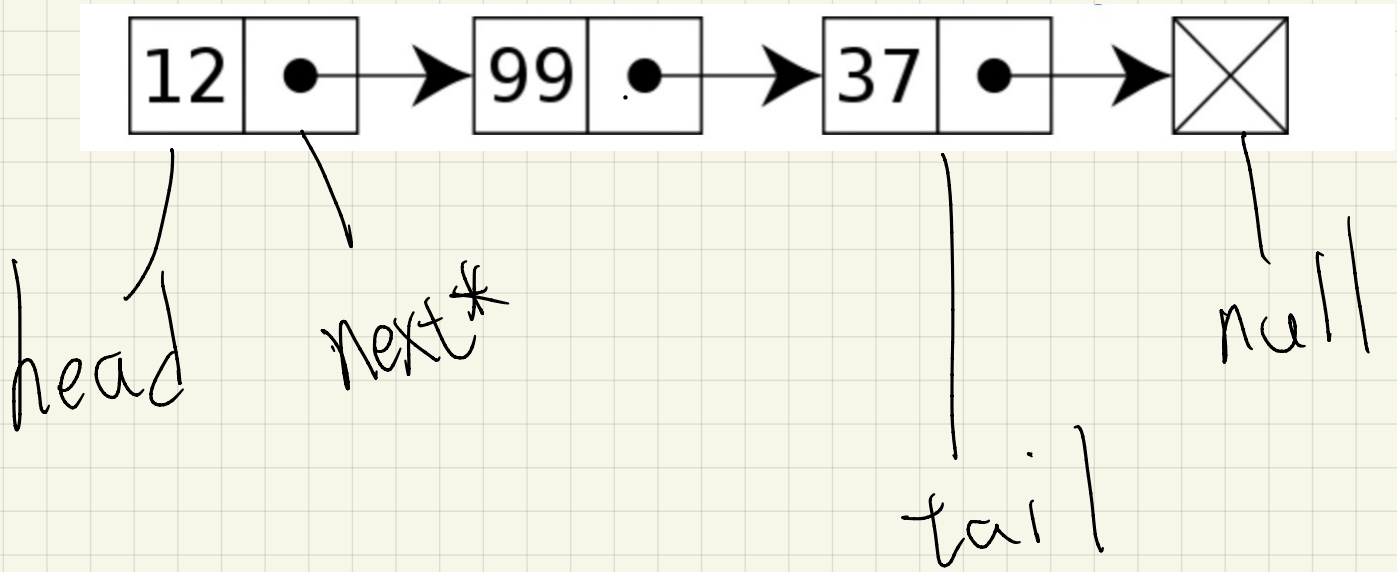
鏈結串列

目的：

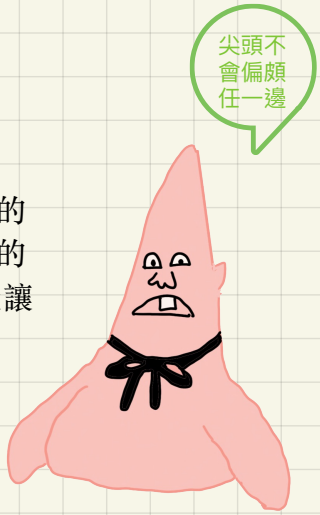
鏈結串列是一種基本線性資料集合，每一個資料元素都是獨立的物件。儲存資料的方式和一般陣列配置連續物理記憶體空間不同，而是在各節點儲存額外的指標指向下一個節點。

- 不需事先知道資料型別大小，充分利用動態記憶體管理。
- 以常數時間插入 / 刪除，不需重新配置記憶體（reallocation）。

範例：



AVL樹



目的：

「為未來鋪路，是一種奢侈。」二元樹在未經過平衡前，使用者在進行搜尋時可能會遇到效率最差、最好的狀態，運氣差到一直遇到最差的人可能會產生如右圖的尖頭表情，倘若在一開始花費時間平衡左右子樹,就能讓每位使用者都能以平均時間完成搜尋。

實現：

1. 單向右旋平衡處理LL：由於在*a的左子樹根節點的左子樹上插入節點，*a的平衡因子由1增至2，致使以*a為根的子樹失去平衡，則需進行一次右旋轉操作；
2. 單向左旋平衡處理RR：由於在*a的右子樹根節點的右子樹上插入節點，*a的平衡因子由-1變為-2，致使以*a為根的子樹失去平衡，則需進行一次左旋轉操作；
3. 雙向旋轉（先左後右）平衡處理LR：由於在*a的左子樹根節點的右子樹上插入節點，*a的平衡因子由1增至2，致使以*a為根的子樹失去平衡，則需進行兩次旋轉（先左旋後右旋）操作。
4. 雙向旋轉（先右後左）平衡處理RL：由於在*a的右子樹根節點的左子樹上插入節點，*a的平衡因子由-1變為-2，致使以*a為根的子樹失去平衡，則需進行兩次旋轉（先右旋後左旋）操作。