資料結構演算法

鏈結串列 Linked list

鏈結串列 Linked List

什麼是串列？

什麼是串列 (List)？

- 串列：
  - 有順序排序的資料
  - 任選兩個元素都會有前後關係

- 對串列資料執行的動作有：
  - 新增
  - 刪除
  - 取值
  - 修改
  - 計算資料數量

最常處理串列資料的資料結構為：

陣列

對陣列值行動作(1) – 新增

```
int list[10];
```

要在 20 與 11 之間加上數字 50 呢？

索引值	0	1	2	3	4	5	6	7	8
元素資料	10	20	11	33	123	61	1	39	40

後面的元素都要往後移

索引值	0	1	**2**	3	4	5	6	7	8	9
元素資料	10	20	**50**	11	33	123	61	1	39	40

對陣列值行動作(2) – 刪除

```
int list[10];
```

要移除 list[2] 內的元素值呢？

索引值	0	1	2	3	4	5	6	7	8
元素資料	10	20	11	33	123	61	1	39	40

後面的元素都要往前移

索引值	0	1	2	3	4	5	6	7
元素資料	10	20	33	123	61	1	39	40

對陣列值行動作(3) – 取值

```
int list[10];
```

要取得 list[2] 內的元素值呢？

索引值	0	1	2	3	4	5	6	7	8
元素資料	10	20	11	33	123	61	1	39	40

list[2] → 11

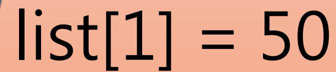
陣列裡面每個元素都有自己的編號(索引值)，可以透過編號存取原宿資料。

對陣列值行動作(4) – 修改

```
int list[10];
```

將索引值 2 內的元素值改為 50

索引值	0	1	2	3	4	5	6	7	8
元素資料	10	20	**11**	33	123	61	1	39	40



list[1] = 50

索引值	0	1	2	3	4	5	6	7	8
元素資料	10	20	**50**	33	123	61	1	39	40

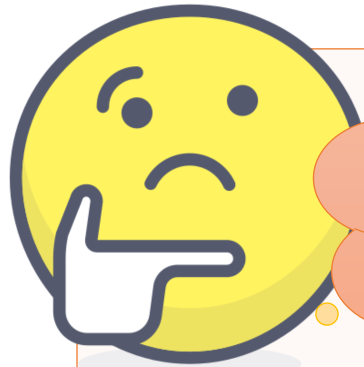
對陣列值行動作(5) –計算資料數量

```
int list[10];
```

索引值	0	1	2	3	4	5	6	7	8
元素資料	10	20	11	33	123	61	1	39	40

sizeof(list) / sizeof(list[0])

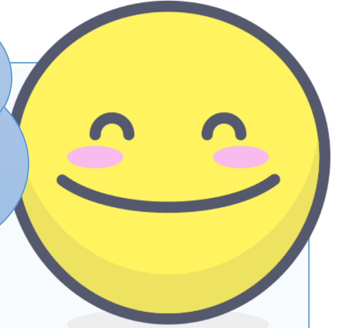
使用陣列處理串列資料



更動的索引位置後面的全部元素都要搬移，工程浩大！

- 新增
- 刪除

還有，陣列需使用太大的連續空間也是個問題！



都可以透過索引值在 O(1) 時間內完成

- 取值
- 修改
- 計算資料數量

鏈節串列（Linked List）

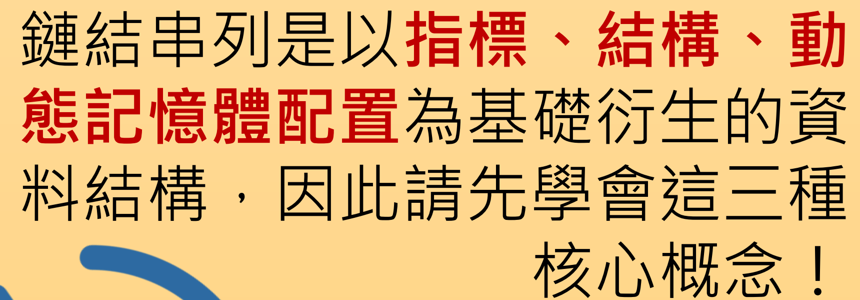
改善陣列處理新增/刪除的問題

改善陣列使用大空間的問題

什麼是
鏈結串列？

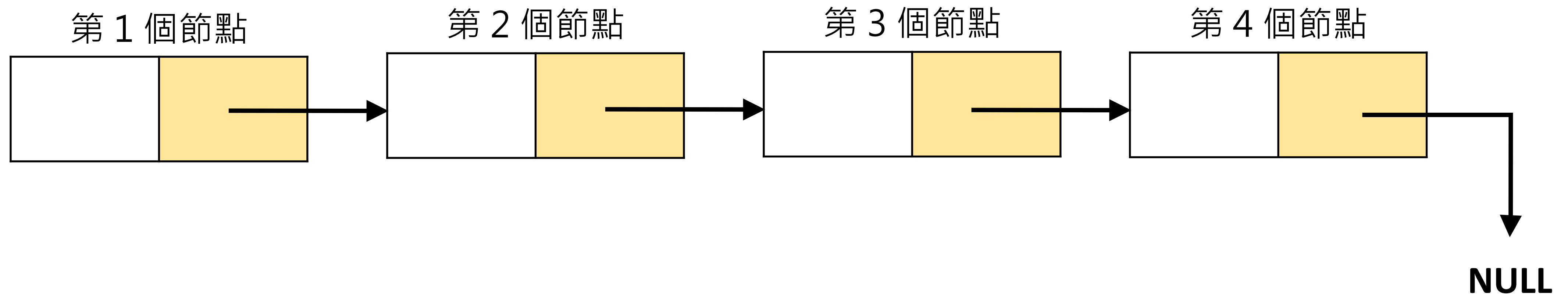
學習鏈結串列前，請先學會

- 指標
  - int *ptr = &a
- 結構
  - struct newType {};
- 動態記憶體配置
  - malloc(sizeof(型態) * 數量)



鏈結串列 (Linked List)

- 將多個在記憶體中**不連續的資料節點(node)**透過**鏈結(指標)**方式串接起來的資料結構。



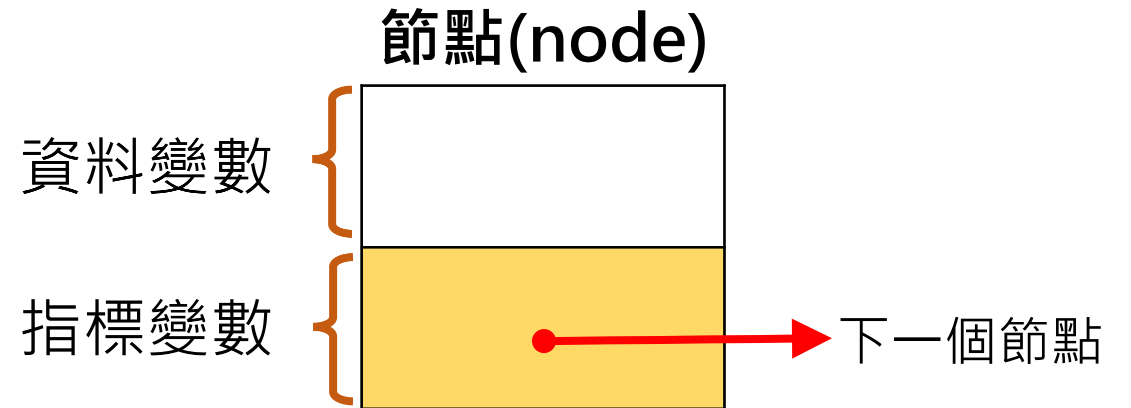
鏈結串列的基本單位：節點(node)

- 一筆資料為一個節點(node)

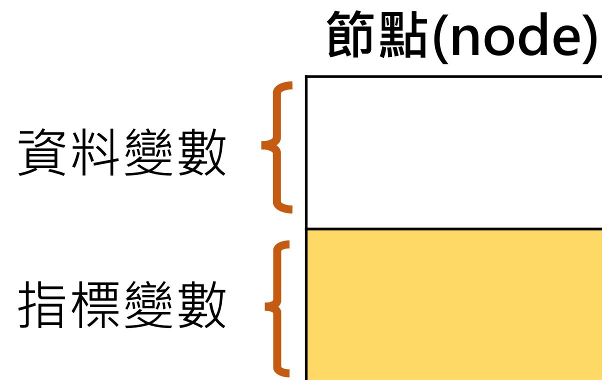
- 節點(node)是一個**結構**型態的資料

- 每個節點透過指標串連

- 結構內**至少**包含兩個元素：
  - **資料**變數
  - 儲存下一個節點的**指標**變數



節點結構

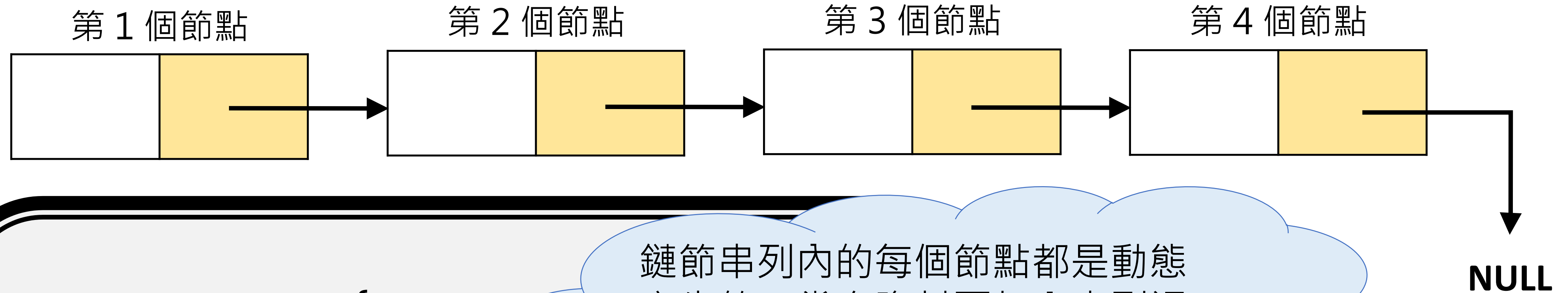



```
typedef struct {
  int data;
  struct node *next;
} node;
```

data 是存放資料的變數，會根據要儲存的資料型態而異，若一筆資料內有多個欄位項目，可在結構內定義多個變數

節點內必定有一個**自身結構型態的指標變數**，用來記錄**下一個節點的位址**，若不存在下一個節點，就將此指標設為 **NULL**

動態產生節點



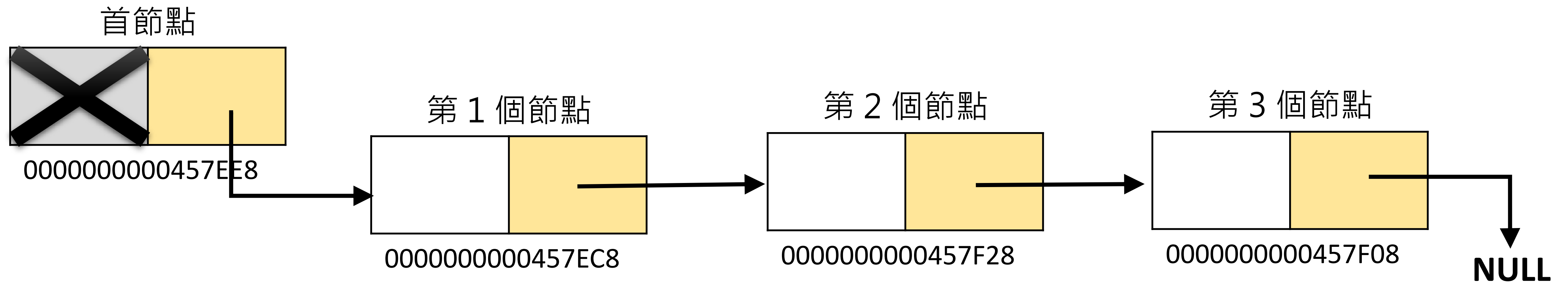
鏈節串列內的每個節點都是動態產生的，當有資料要加入串列裡面時，就透過 malloc() 動態取得存放新節點的記憶體空間。

```c
typedef struct {
  int data;
  struct node *next;
} node;

node *newNode;
newNode = (node *)malloc(sizeof(node));
newNode->next = NULL;
```

節點位址

- 各個節點的位址是不相鄰的（使用陣列處理時，每個元素必定是相鄰的）



操作
鏈結串列

建立串列 Create List

- 建立一個空的鏈結串列
  ⇒ 建立首節點
- **首節點**：
  - 首節點內不存資料
  - 只會紀錄串列第一個資料的節點位址

```c
#include <stdio.h>
typedef struct {
  int data;
  struct node *next;
} node;

int main()
{
  node *head;
  head = (node *) malloc(sizeof(node));
  head->next = NULL;
  return 0;
}
```

1 步驟1：使用 malloc() 動態產生頭節點

2 步驟2：將頭節點的下一個節點指標位址初始化為 NULL

新增節點 Insert Node

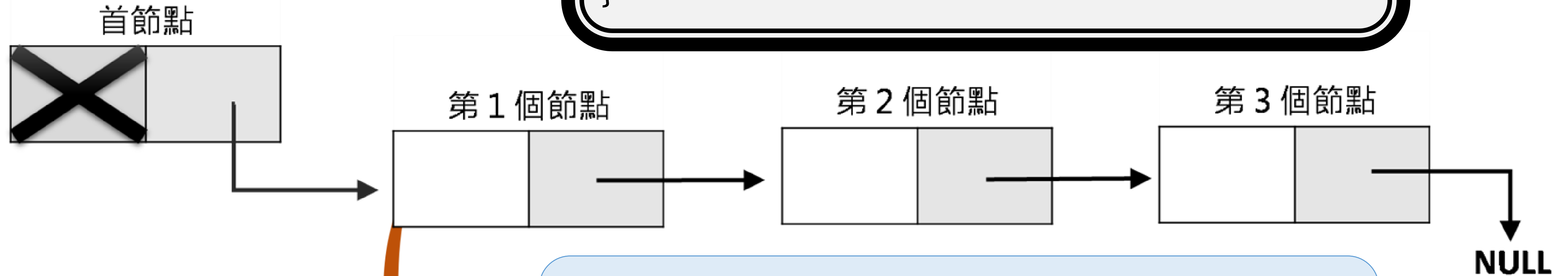
三種新增節點的位置：

- **前端**新增節點：新增在串列最前面
- **中間**新增節點：新增在指定位置（非串列第一個也非串列最後一個）
- **末端**新增節點：新增在串列最後面

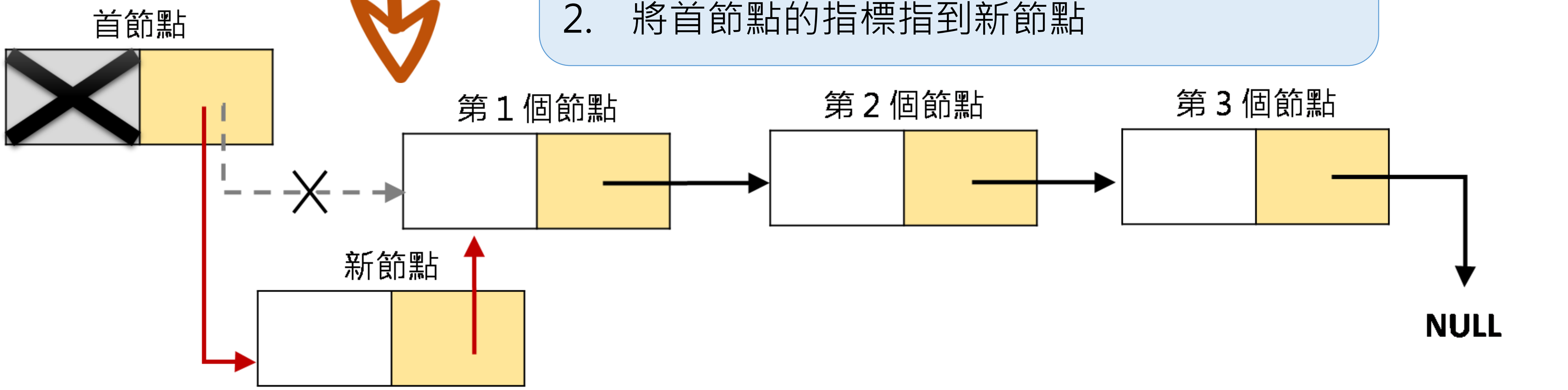
新增節點 Insert Node

前端新增節點

```c
void insertNodeFront(node *head, node *newNode)
{
  newNode->next = head->next;
  head->next = newNode;
}
```



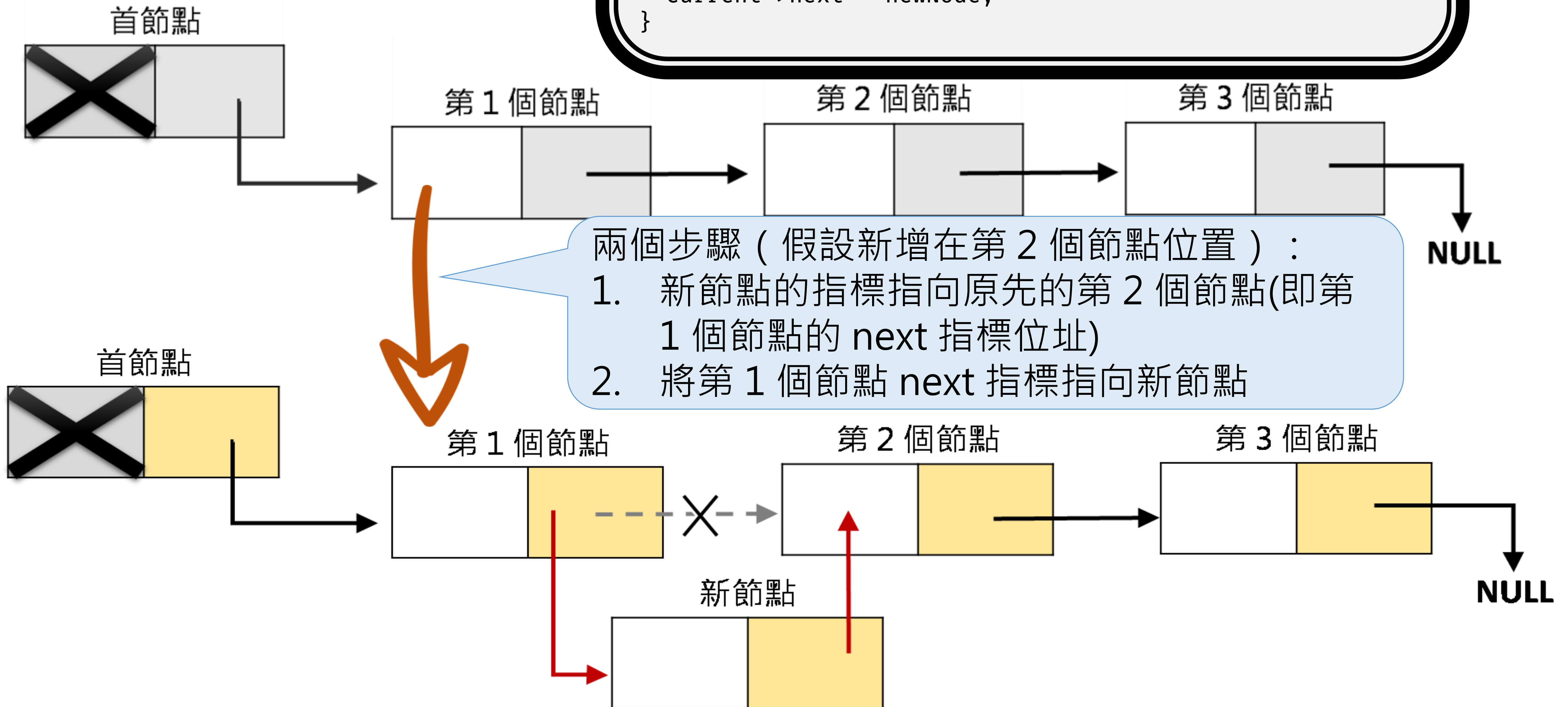
兩個步驟：
1. 新節點的指標指向原本的第 1 個節點
2. 將首節點的指標指到新節點



新增節點 Insert Node

中間新增節點

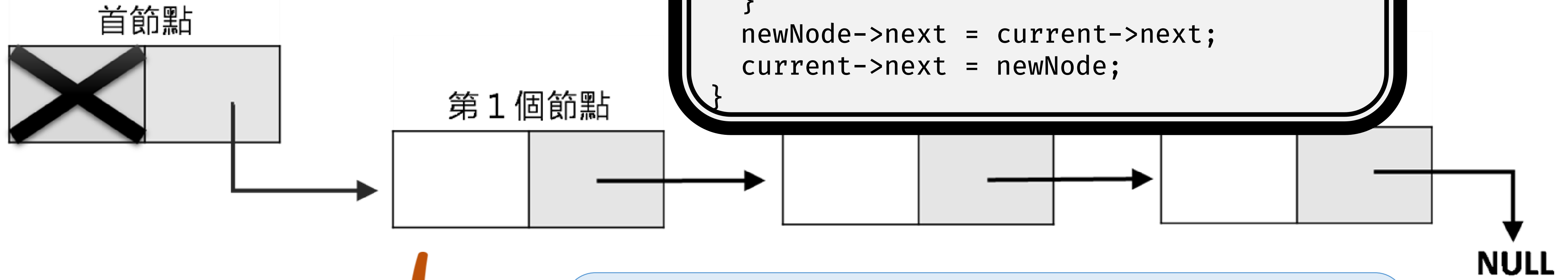
```c
void insertNodeIndex(int index, node *head, node *newNode)
{
  int count = 0;
  node *current = head;
  while(count<index && current->next != NULL){
    current = current->next;
    count = count + 1;
  }
  newNode->next = current->next;
  current->next = newNode;
}
```



新增節點 Insert Node

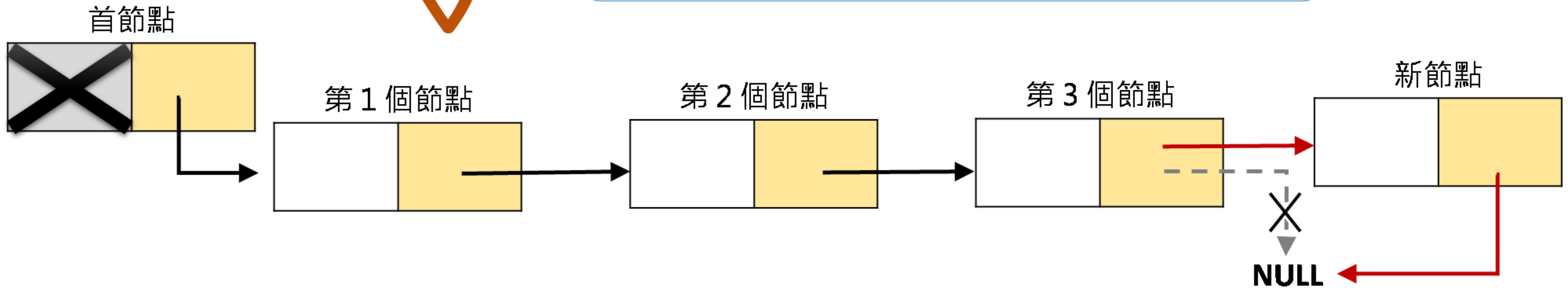
末端新增節點

```
void insertNodeEnd(node *head, node *newNode)
{
  node *current = head;
  while(current->next != NULL) {
    current = current->next;
  }
  newNode->next = current->next;
  current->next = newNode;
}
```



兩個步驟：

1. 原本最後的一個節點的指標指向新節點
2. 新節點的指標指向 NULL



新增節點 Insert Node

前端新增節點

```
void insertNodeFront(node *head, node *newNode){
  newNode->next = head->next;
  head->next = newNode;
}
```

末端新增節點

```
void insertNodeEnd(node *head, node *newNode){
  node *current = head;
  while(current->next != NULL) {
    current = current->next;
  }
  newNode->next = current->next;
  current->next = newNode;
}
```

中間新增節點

```
void insertNodeIndex(int index, node *head, node *newNode){
  int count = 0;
  node *current = head;
  while(count<index && current->next != NULL){
    current = current->next;
    count = count + 1;
  }
  newNode->next = current->next;
  current->next = newNode;
}
```

刪除節點 Delete Node

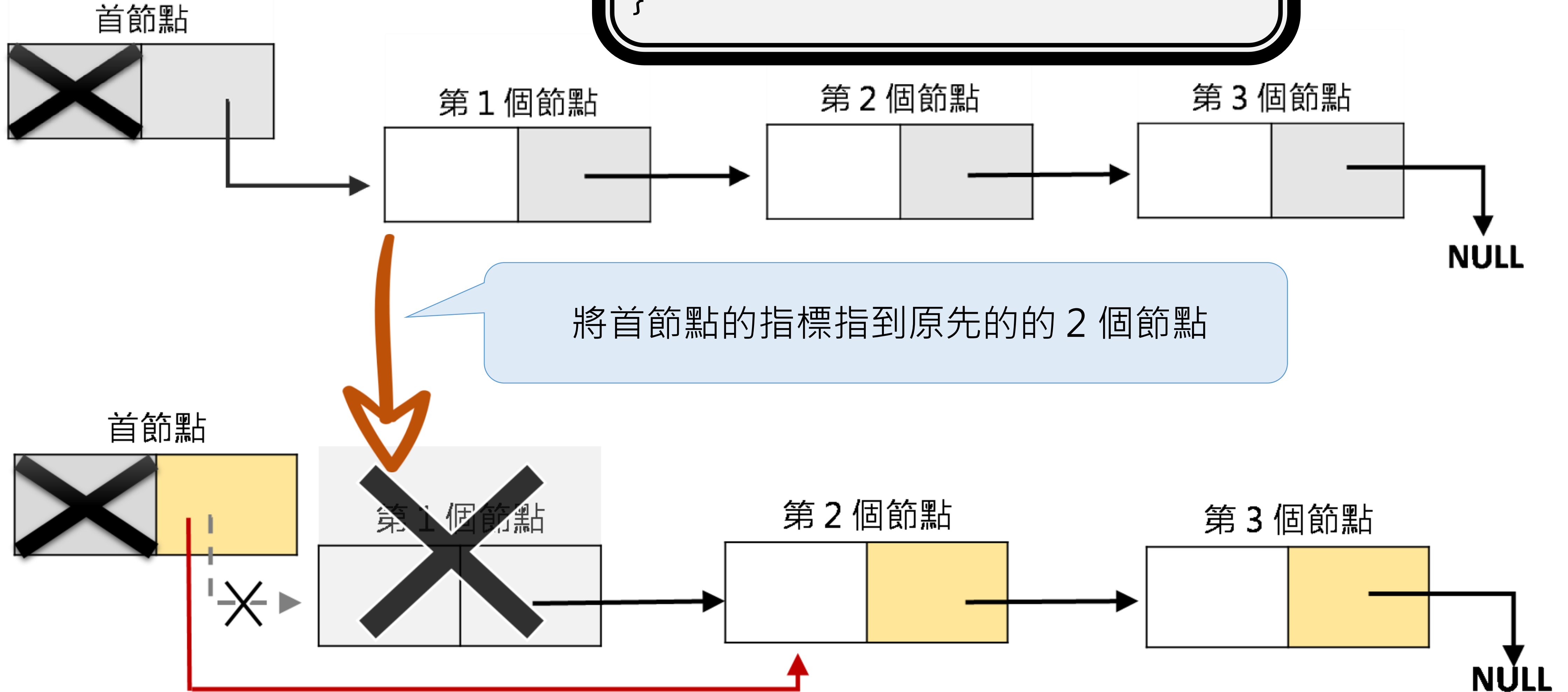
三種刪除節點的位置：

- 刪除**前端**節點：刪除串列最前面的那個節點
- 刪除**中間**節點：刪除指定位置的節點（非串列第一個也非串列最後一個）
- 刪除**末端**節點：刪除串列最後面的那個節點

刪除節點 Delete Node

刪除前端節點

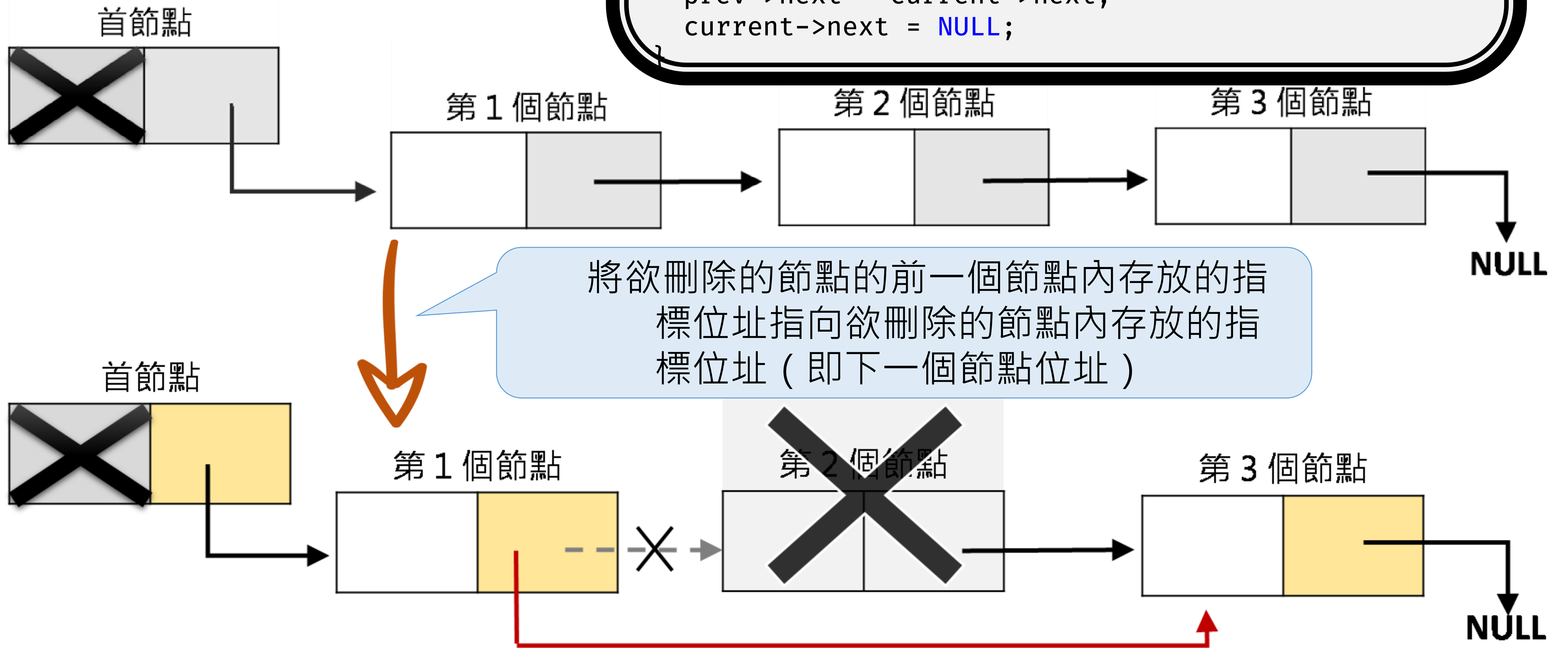
```c
void deleteNodeFront(node *head)
{
   node * current = head->next;
   head->next = current->next;
}
```



刪除節點 Delete Node

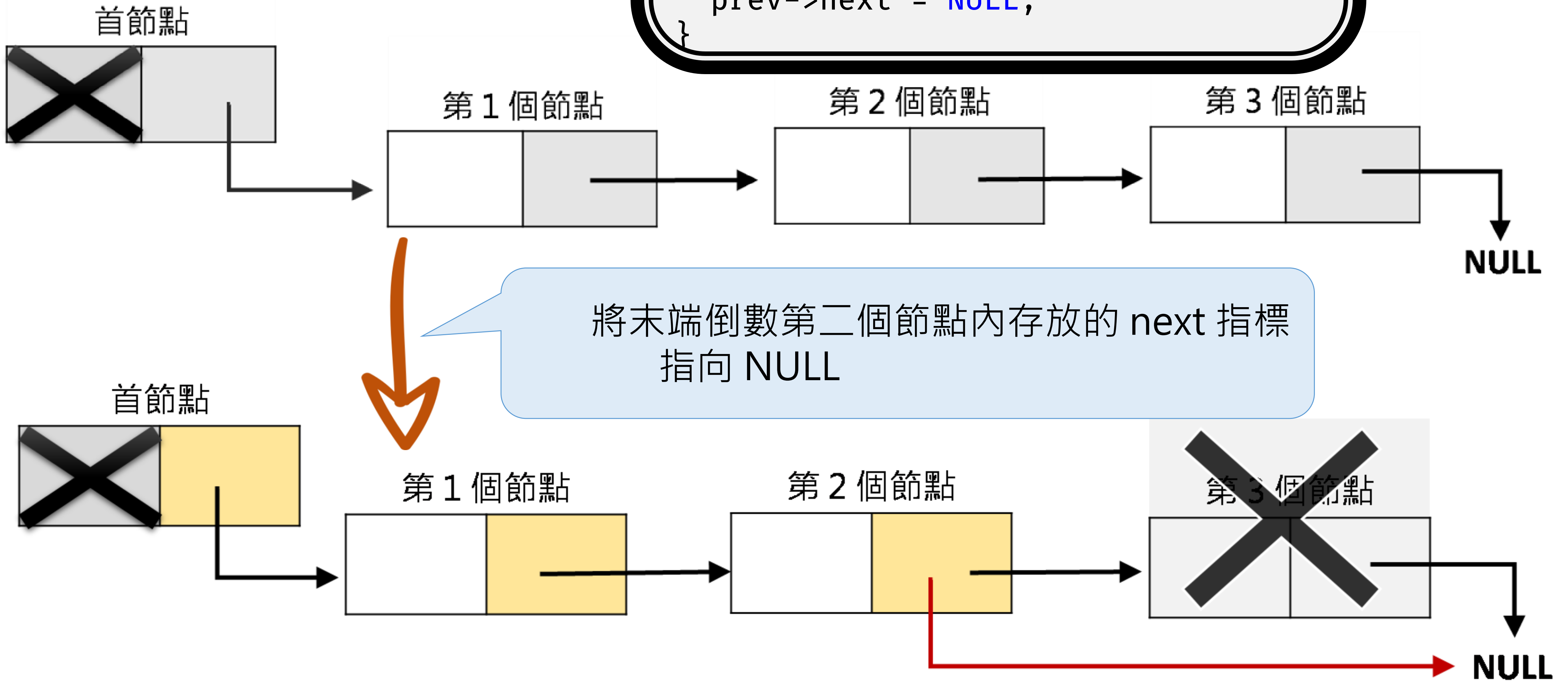
刪除中間節點

```
void deleteNodeIndex(int index, node *head)
{
  int count = 0;
  node *prev;
  node *current = head;
  while(count < index && current->next != NULL) {
    prev = current;
    current = current->next;
    count++;
  }
  prev->next = current->next;
  current->next = NULL;
}
```



刪除末端節點

```
void deleteNodeEnd(node *head)
{
  node *current = head;
  node *prev;
  while(current->next != NULL) {
    prev = current;
    current = current->next;
  }
  prev->next = NULL;
}
```



刪除節點 Delete Node

刪除前端節點

```
void deleteNodeFront(node *head)
{
  current = head->next;
  head->next = current->next;
}
```

刪除末端節點

```
void deleteNodeEnd(node *head){
{
  node *prev;
  node *current = head;
  while(current->next != NULL) {
    prev = current;
    current = current->next;
  }
  prev->next = NULL;
}
```

刪除中間節點

```
void deleteNodeIndex(int index, node *head){
{
  int count = 0;
  node *prev;
  node *current = head;
  while(count < index && current->next !=
NULL) {
    prev = current;
    current = current->next;
    count++;
  }
  prev->next = current->next;
  current->next = NULL;
}
```

刪除節點 Delete Node

- 前面所提的刪除方式都是將節點從鏈節串列中移除，但節點並沒有被刪除，此種方式稱為邏輯性刪除。

- 若要實際刪除該節點，需使用 free() 將透過 malloc 動態配置的節點空間釋出

使用 free() 將節點空間釋出

刪除前端節點

```
void deleteNodeFront(node *head){
  node *current = head->next;
  head->next = current->next;
  free(current);
}
```

刪除末端節點

```
void deleteNodeEnd(node *head){
  node *current = head;
  while(current->next != NULL) {
    prev = current;
    current = current->next;
  }
  prev->next = NULL;
  free(current);
}
```

刪除中間節點

```
void deleteNodeIndex(int index, node *head){
  int count = 0;
  node *current = head;
  while(count < index && current->next != NULL) {
    prev = current;
    current = current->next;
    count++;
  }
  prev->next = current->next;
  current->next = NULL;
  free(current);
}
```

走訪節點 Traversal

```c
void traversalNode(node *head)
{
  node *current = head->next;

  while(current != NULL) {
    current = current->next;
  }
}
```

1. 先透過首節點取得第一個節點
2. 判斷是否已經走訪到串列的末端節點了
3. 移動到下一個節點

尋找節點 Find Node

```
int findNode(int num, node *head)
{
  node *current = head->next;

  while(current != NULL) {

    if (current->num == num) {
      printf("find num %d in %d\n",current->num,index);
       return 1;
    }

    current = current->next;
  }
  return 0;
}
```

尋找節點 **Find Node**

‖

走訪節點 **Traversal + 條件判斷**

判斷目前的節點是否符合要找尋的值

尋得串列長度 Length

```
int getLength(node *head){
  int count = 0;

  node *current = head->next;

  while(current != NULL) {

   count++;

   current = current->next;
  }
  return count;
}
```

記數運算

尋找節點 **Find Node**

走訪節點 **Traversal + 記數運算**

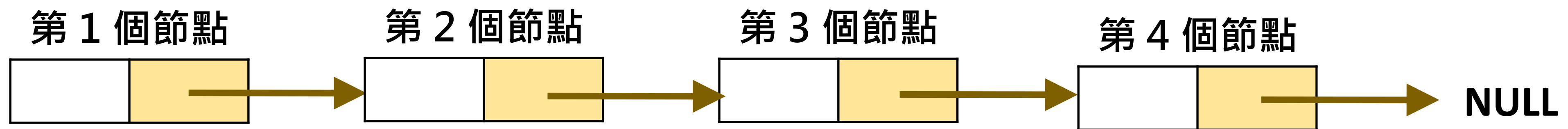
延伸的概念

概念1: 鏈節串列的種類

- 根據**節點的串接方式**，鏈節串列有三種種類：
    - **單向**鏈結串列：每個節點只有**一個指標**，指向**一個方向**
    - **雙向**鏈結串列：每個節點有**二個指標**，指向**二個方向**
    - **環狀**鏈結串列：**最後一個節點**的指標不會指到 NULL，會**指向第一個節點**

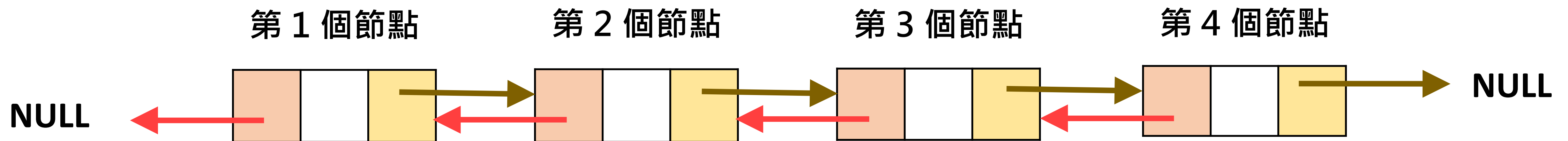
單向鏈結串列

- 每個節點只有**一個指標**，指向**一個方向**
- 缺點：串列走訪時只能順著一個方向，若只是要取前一個節點的值，仍需要將整個串列走訪一次。



雙向鏈結串列

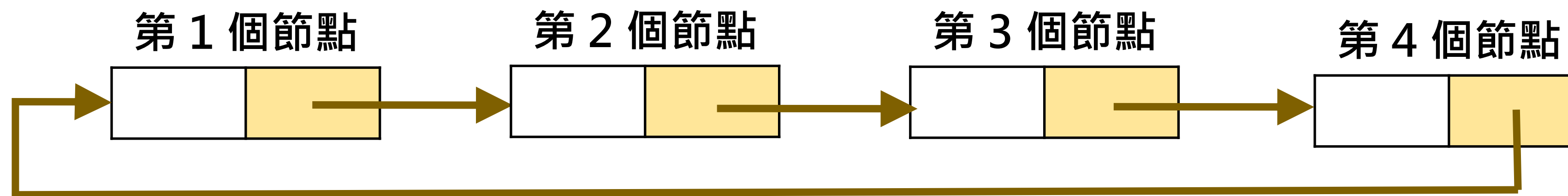
- 每個節點有**二個指標**，指向**二個方向**
- 優點：可依照需求，選擇順時針或逆時針的走訪串列。
- 缺點：每個節點需要多提供一個指標空間，以空間換取時間加速。

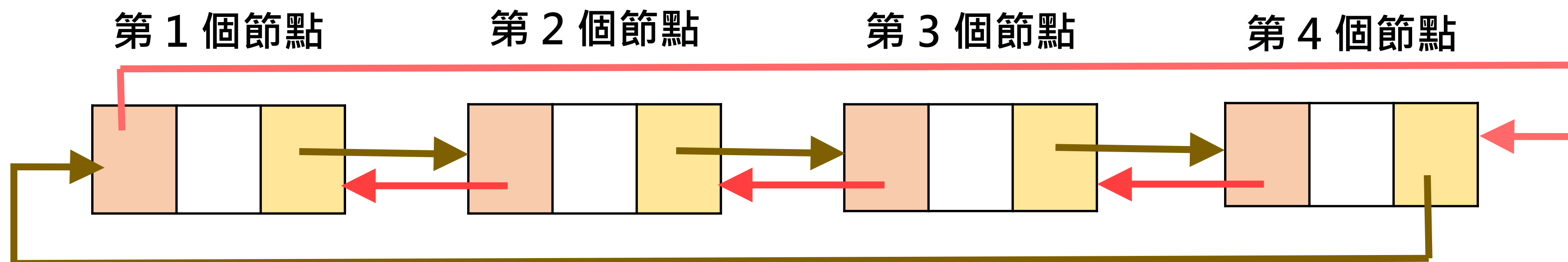


概念1: 鏈節串列的種類

環狀鏈節串列

- **最後一個節點**的指標不會指到 NULL，會**指向第一個節點**
- 可以是單向環狀鏈節串列，也可以是雙向環狀鏈節串列
- 優點：從任何一個節點開始，都可以走訪整個串列。





概念2: 陣列 V.S. 鏈節串列

	陣列	鏈節串列
記憶體配置	連續空間	不連續的空間
記憶體空間	資料欄位	資料欄位 + 指標欄位
空間配置	靜態固定空間	動態宣告
元素存取速度	快	慢
新增刪除節點	速度慢，O(n)	速度快，O(1)
共用	不可共用	可共用
可靠性	高	低 (指標設定錯誤，資料就不見)
適用時機	•需要快速存取指定元素 •資料量以知，不會動態增減	•不需要快速存取指定元素 •需要頻繁新增/刪除資料 •資料量需動態增減，增減幅度無法預期