

【第六講】

串列

講師: 李根逸 (Ken-Yi Lee), E-mail: feis.tw@gmail.com



課程大綱

■ 定義型態：`typedef`

- ▶ 為型態取別名

■ 單向鏈結串列

- ▶ 建構單向鏈結串列 (`constructor`)
- ▶ 隨機存取單向鏈結串列 (`operator[]`)
- ▶ 在串列前端新增或删除元素 (`PushFront()`, `PopFront()`)
- ▶ 在串列尾端新增或删除元素 (`PushBack()`, `PopBack()`)
- ▶ 於單向鏈結串列中的任意位置插入元素 (`Insert()`)
- ▶ 删除單向鏈結串列中任意位置的元素 (`Erase()`)
- ▶ 大三法則 (解構式、複製建構式與賦值運算子)
- ▶ 有序型單向鏈結串列

■ 陣列與單向鏈結串列的效率比較

定義型態： `typedef`

- 一般我們可以像下面這樣定義變數：

```
int a;           // a 是個變數（物件） 名稱其型態為 int
```

- 相似地，我們可以用 `typedef` 來從已有的型態去定義新的型態

```
typedef int a;    // a 是個型態名稱，可以用來宣告變數
```

- 例如我們可以用 `int` 去定義 `Grade`：

```
typedef int Grade; // 以後 Grade 就是個型態名稱  
                  // 且與 int 同義
```

```
Grade b;          // 定義一個變數名為 g  
                  // 其型態為 Grade
```

【思考】這跟我們之前用類別來自訂型態有什麼不一樣？

[範例] `typedef.cpp`

為型態取別名

- **typedef** 最常見的用途是來為一個比較複雜的型態取簡單易懂的別名：

```
int c[20];           // 定義一個名為 c 的變數
int d[20];           // 定義一個名為 d 的變數

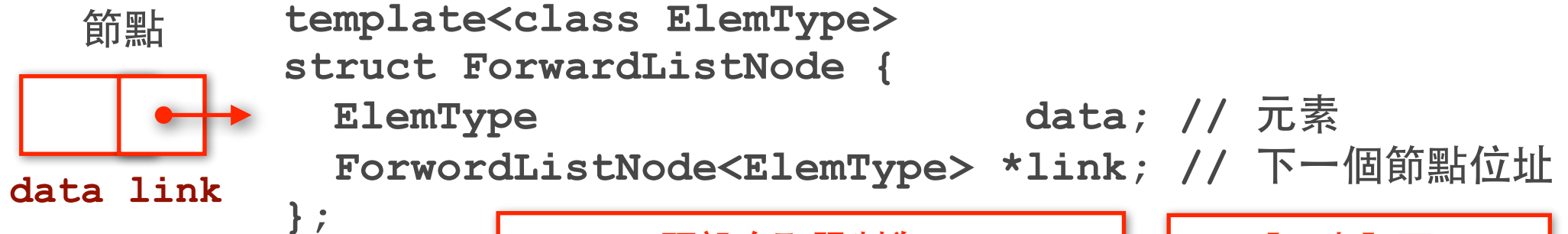
typedef int Array[20]; // 定義一個名為 Array 的型態
Array c;              // 定義一個名為 c 的變數
Array d;              // 定義一個名為 d 的變數
                     // 此時的 c, d 為 20 個 int 的陣列
c[5] = 6;

typedef int (*Cmp)(const void *, const void *);

Cmp f;                // 此時 f 是個什麼？
```

單向鏈結串列

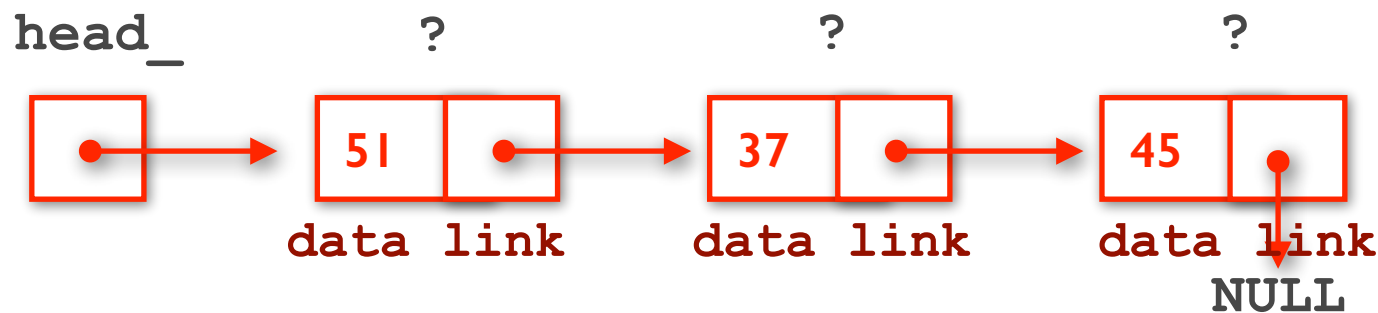
- 鏈結串列 (**linked list**) 由節點 (**node**) 構成：



struct 預設存取限制為 **public**，
class 預設存取限制為 **private**

【思考】要用 **struct** 或 **class** ？

- 單向鏈結串列本身會記得第一個節點位址 (**head_**)，而每個節點會記得下一個節點的位址，最後一個節點會指向空節點 (實作上通常是 **NULL** 位址)



可以用參考指向
下一個節點嗎？

參考可以不參考
任何東西嗎？

【範例】 建構單向鏈結串列

- 使用單向鏈結串列實作下面這個介面：

```
template<class ElemType>
class ForwardList {
    typedef ForwardListNode<ElemType> Node;

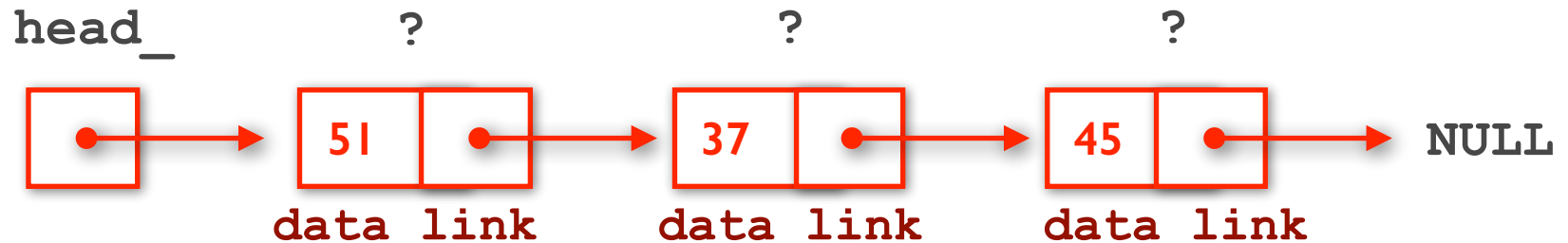
public:
    ForwardList();           // 產生大小為 0 的串列
    explicit List(int n);    // 產生大小為 n 的串列
    ~List();

    int Size() const;        // 回傳串列內的元素個數

private:
    int size_;               // 串列內的元素個數
    Node *head_;             // 指向第一個元素
};
```

[範例] list_1.cpp

存取串列的元素



```
head_->data == 51 // 第一個元素
head_->link->data == 37 // 第二個元素
head_->link->link->data == 45 // 第三個元素
```

// 存取第 `i` 個 元素：

```
head_->(重複 i-1 次 link->) data
```

【思考】可以寫成迴圈嗎？

```
head_->link->link->link == NULL // 結尾
```

【範例】隨機存取串列

- 在 `ForwardList<ElemType>` 內新增下列操作：

```
// 回傳第 i 號元素  
ElemType &At(int i);  
const ElemType &At(int i) const;  
  
ElemType &operator[](int i);  
const ElemType &operator[](int i) const;
```

`At()` 和 `operator[]()` 所花的時間會不會跟元素個數成正比？

【練習】計算串列大小

- 在 `ForwardList<ElemType>` 內新增或修改下列操作：
- ▶ 在不使用資料成員 (`size_`) 的情況下求得串列元素個數

```
int Size() const;    // 回傳串列元素個數
```

這樣 `Size()` 所花的時間會不會跟元素個數成正比？

```
bool Empty() const;  // 回傳是否為空串列
```

如果串列是空則回傳 `true`，否則回傳 `false`

【練習】存取串列頭尾

- 在 `ForwardList<ElemType>` 內新增下列操作：

```
// 回傳第一個元素  
ElemType &Front();  
const ElemType &Front() const;
```

Front() 所花的時間會不會跟元素個數成正比？

```
// 回傳最後一個元素  
ElemType &Back();  
const ElemType &Back() const;
```

Back() 所花的時間會不會跟元素個數成正比？

【思考】 有辦法讓這些操作所花的時間與元素個數無關嗎？

【範例】 在前端新增或删除元素

- 在 `ForwardList<ElemType>` 內新增下列操作：

```
// 新增元素在串列的前端 (串列大小多一)  
void PushFront(const ElemType &elem);
```

```
// 移除第一個元素 (串列大小少一)  
void PopFront();
```

跟陣列比起來如何？

【練習】在尾端新增或刪除元素

- 在 `ForwardList<ElemType>` 內新增下列操作：

```
// 新增元素在串列的尾端 (串列大小多一)  
void PushBack(const ElemType &elem);
```

```
// 移除最後一個元素 (串列大小少一)  
void PopBack();
```

跟陣列比起來如何？

【注意】當串列為空時，這兩個操作會不會出事？

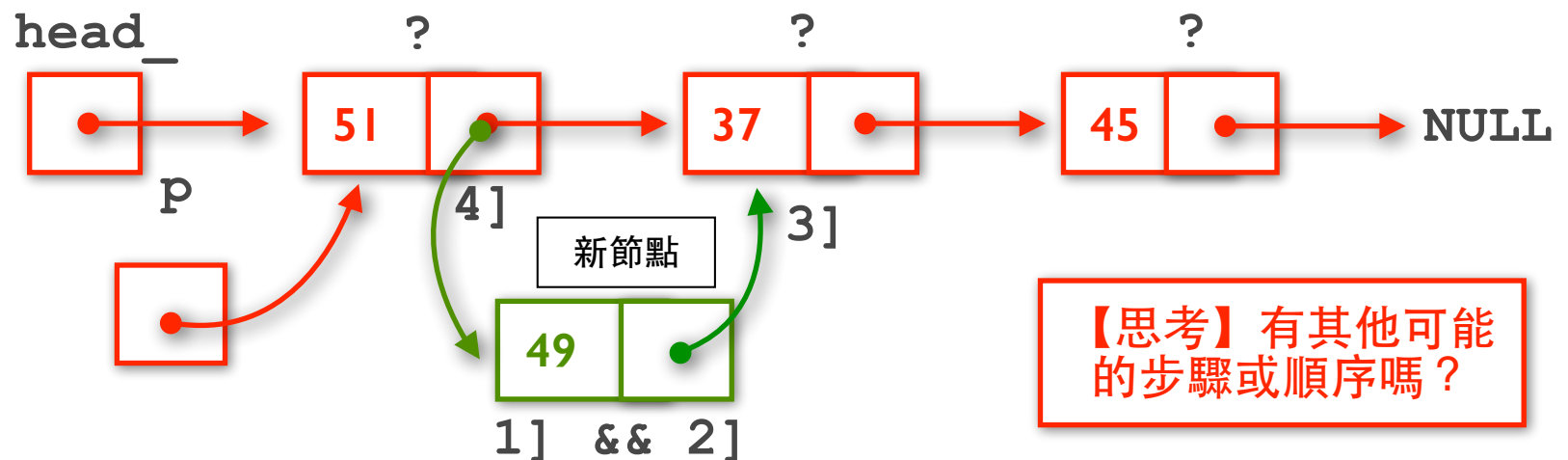
`PushBack()` 與 `PopBack()` 花的時間會不會跟元素個數成正比？

有辦法讓這些操作所花的時間與元素個數無關嗎？

[練習] `ex6C.cpp`

插入元素於串列的任意位置

- 1] 找到要插入位置的前一個節點，並將位址儲存在 p 指標
- 2] 產生『新節點』並填寫『新節點』的元素值
- 3] 設定『新節點』的下一個節點位址為 p 指向節點的下一個節點位址
- 4] 設定 p 指向節點的下一個節點位址為『新節點』的位址

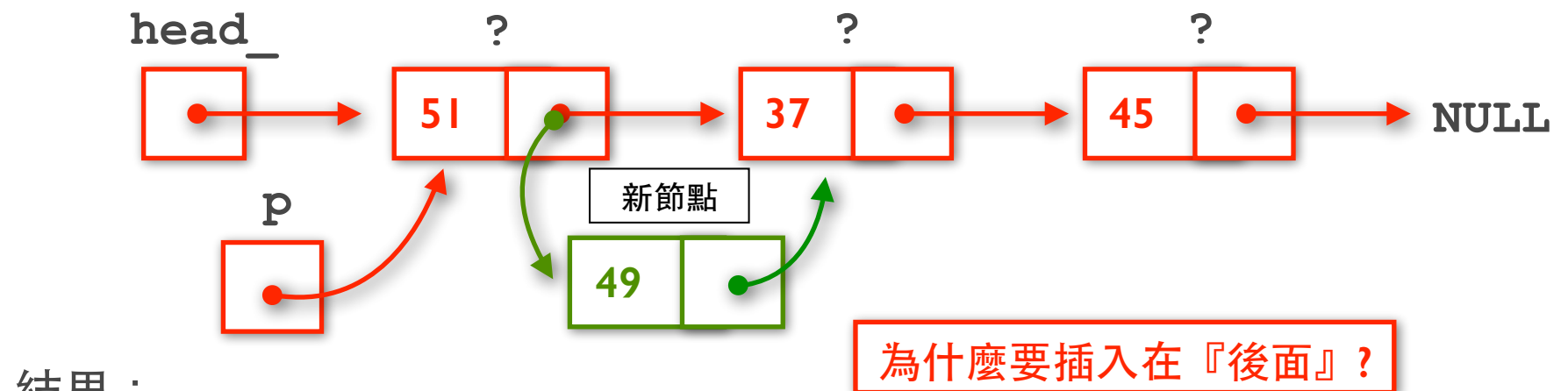


【範例】 在任意位置插入元素

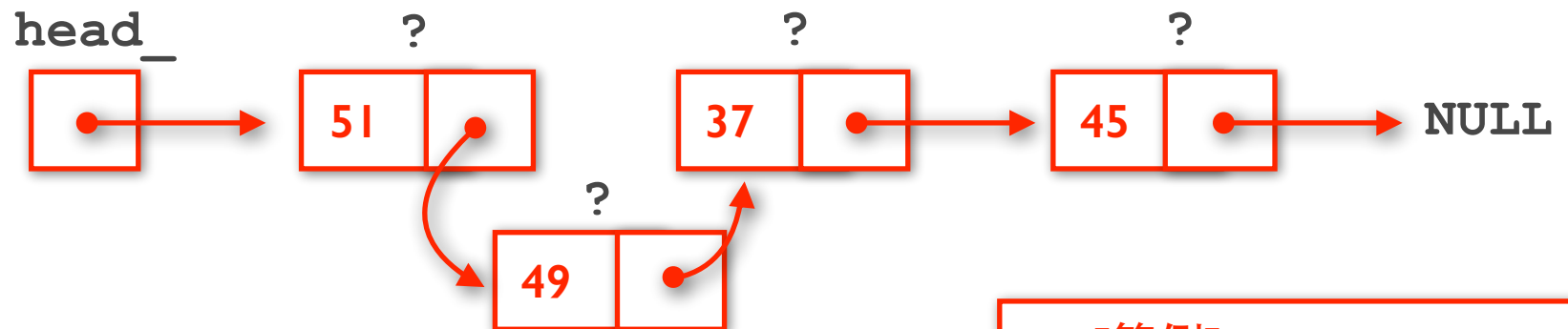
// 插入元素於 pos

```
void Insert(int pos, const ElemType &elem);
```

呼叫 Insert(1, 49):



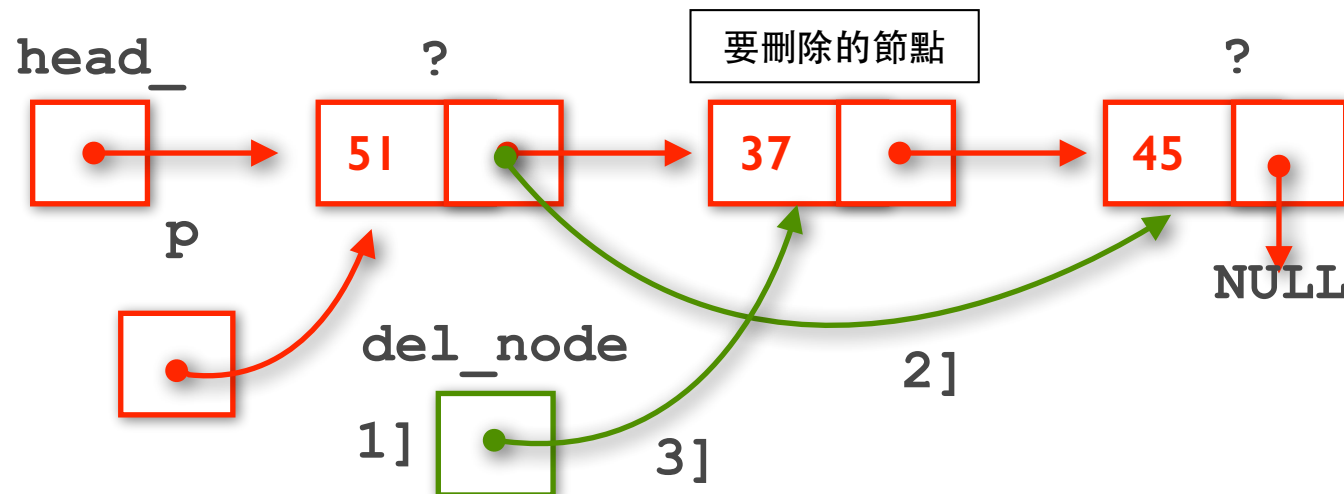
結果：



[範例] list_4.cpp

刪除串列中任意位置的元素

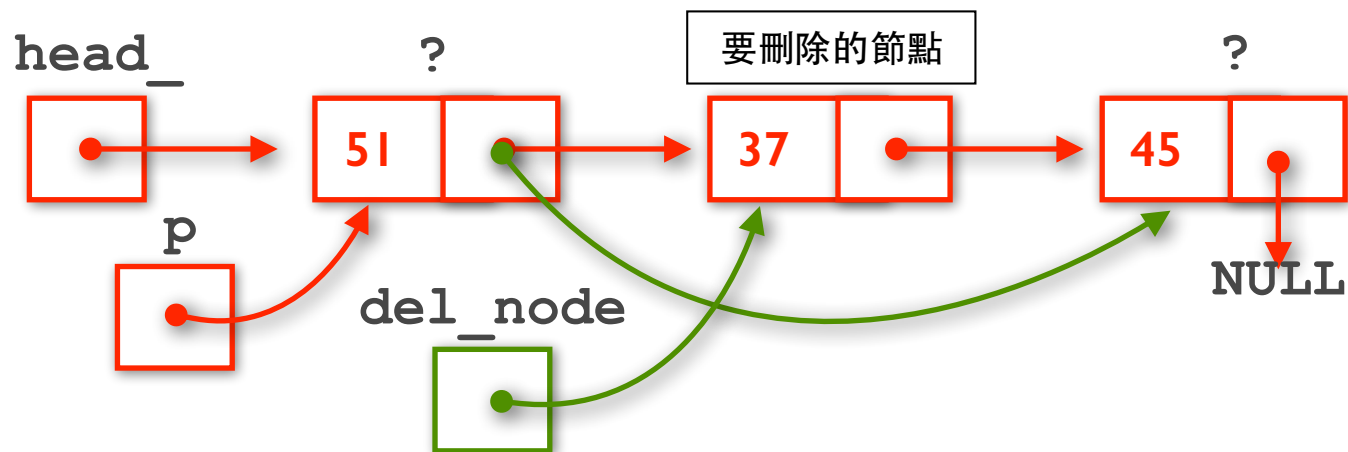
- 1] 找到要刪除節點的前一個節點，並將位址儲存在 `p` 指標。另外將要刪除的節點的位址存放在 `del_node` 指標
- 2] 設定 `p` 指向節點的下一個節點位址為 `del_node` 指向節點的下一個節點位址
- 3] 刪除 `del_node` 指向的節點



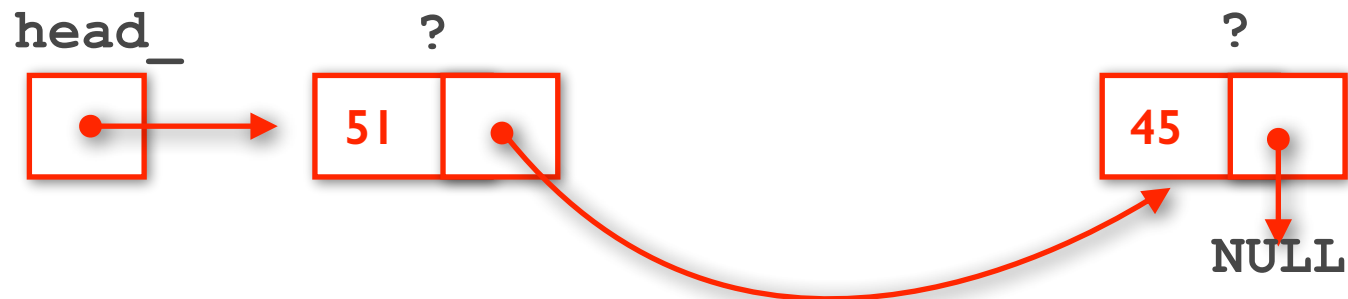
【範例】在任意位置刪除元素

```
// 刪除 pos 位置的元素  
void Erase(int pos);
```

呼叫 `Erase(1)` :



結果：



[範例] `list_5.cpp`

【練習】大三法則

- 在 `ForwardList<ElemType>` 內新增下列操作：

```
// 清除內容  
void Clear();
```

注意記憶體洩漏 (memory leak) 問題

```
// 交換兩個串列 (試著在過程中不要複製元素或產生新節點)  
void Swap(List<ElemType> &x);
```

```
// [複製建構式] 配置與複製節點  
ForwardList(const List<ElemType> &rhs);
```

```
// [賦值運算子]  
ForwardList<ElemType> &operator=(  
    const ForwardList<ElemType> &rhs);
```

可不可以用上面兩種函式與解構式構成
`operator=` ?

複製與交換法 (Copy-and-Swap)：利用複製建構式、解構式與交換函式做賦值運算
`T &operator=(const T &rhs){ T tmp(rhs); Swap(tmp); return *this;}`

【思考】用複製與交換法時，出現自我賦值會有問題嗎？

[練習] `ex6D.cpp`

【練習】 新增其他操作

- 在 `List<ElemType>` 內新增下列操作：

```
// 改變串列大小 (試著在過程中不要複製元素)  
void Resize(int n);
```

```
// 反轉串列內容 (試著在過程中不要複製元素或產生新節點)  
void Reverse();
```

盡量用修改指標的方式達成

【練習】有序型單向鏈結串列

- 有序型 (**ordered**) 串列指的是內容的資料會保證是由小排到大 (或是依照某種特定順序排列) :
 - ▶ 插入元素時要自動找到適當的位置插入來保證維持有序型的狀態 (大小順序不變)

```
template<class ElemType>
class OrderedForwardList {
public:
    OrderedForwardList();
    ~OrderedForwardList();
    int Size() const;
    ElemType &At(int i);
    const ElemType &At(int i) const;
    void Insert(const ElemType &elem);

    ElemType &operator[](int i);
    const ElemType &operator[](int i) const;
};
```

【注意】依照大三法則，我們應該還要實作複製建構式與賦值運算子，不過為了簡化練習就省略了

[練習] ex6F.cpp

陣列與串列的效率比較

操作	可變大小陣列 (Vector)	單向鏈結串列 (ForwardList)
Size, Empty (大小等資訊)	與個數無關	可與個數無關
At, operator[] (隨機存取)	與個數無關	與個數有關
PushFront, PopFront (在前端新增或移除元素)	與個數有關	與個數無關
PushBack, PopBack (在尾端新增或移除元素)	可能與個數無關	PopBack 與個數有關
Insert, Erase (在任意位置新增或移除元素)	與個數有關	與個數有關
解構式, 複製建構式與複製指定 運算子 (大三法則)	與個數有關	與個數有關

勝

勝

為什麼新增
跟刪除會與
個數有關？
要如何設計
才可以跟個
數無關？

雙向鏈結陣
列與迭代器







