【第八講】

堆疊與佇列

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

★ ★ ★ ★

★ ★ ★ ★

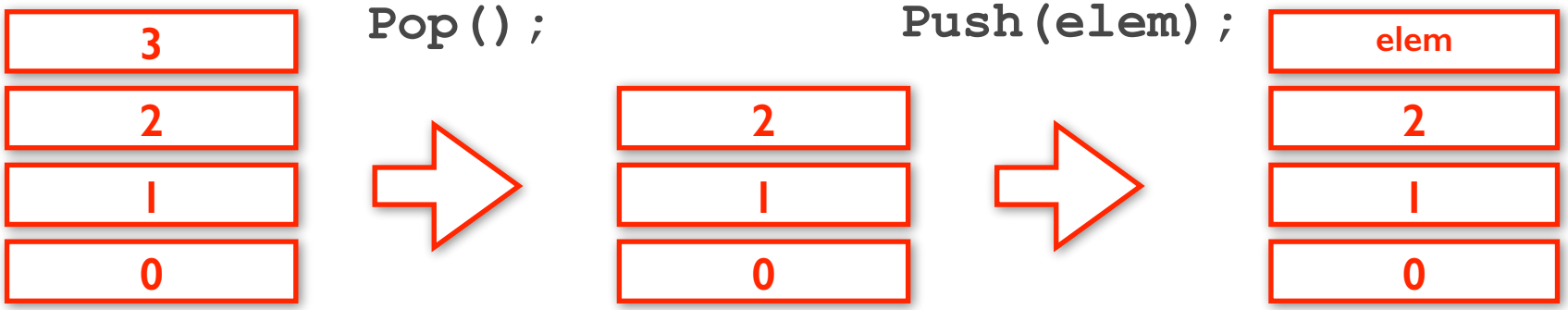
課程大綱

- 堆疊 **(stack)**
  - 使用可變大小陣列 (**std::vector**) 實作堆疊
  - 使用串列 (**std::list**) 實作堆疊
- 佇列 **(queue)**
  - 使用串列 (**std::list**) 實作堆疊
  - 使用可變大小陣列 (**std::vector**) 實作堆疊
- 自適應容器 **(adaptive container)**
- 雙邊結尾佇列 **(double-ended queue)**
- 使用 **STL** 的堆疊與佇列
- 堆疊與佇列的應用：四則運算

堆疊 (stack)

- 堆疊是一個可以從頂端新增或移除元素的資料結構
  - 可實現 **LIFO** (Last-in first-out) 的機制

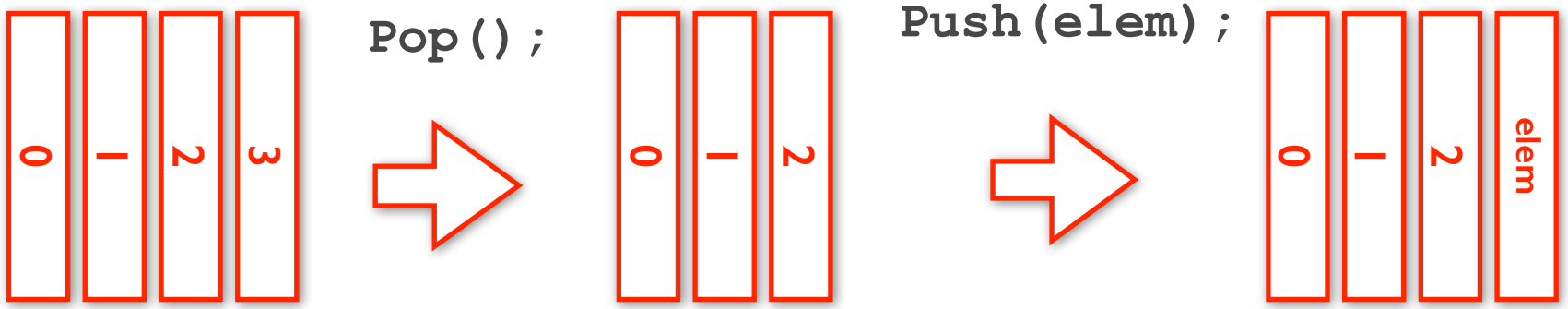
```
template <class ElemType>
class Stack {
  public:
    ElemType &Top();                          // 回傳頂端的元素
    const ElemType &Top() const;
    void Push(const ElemType &elem);// 插入元素於頂端
    void Pop();                               // 移除頂端的元素
};
```



【練習】使用陣列實作堆疊

- 練習使用 **std::vector** 去實作堆疊

```
template <class ElemType>
class Stack {
 public:
  ElemType &Top();                          // 回傳頂端的元素
  const ElemType &Top() const;
  void Push(const ElemType &elem);// 插入元素於頂端
  void Pop();                               // 移除頂端的元素
  bool Empty() const;                       // 回傳是否為空
  int Size() const;                         // 回傳堆疊大小
};
```

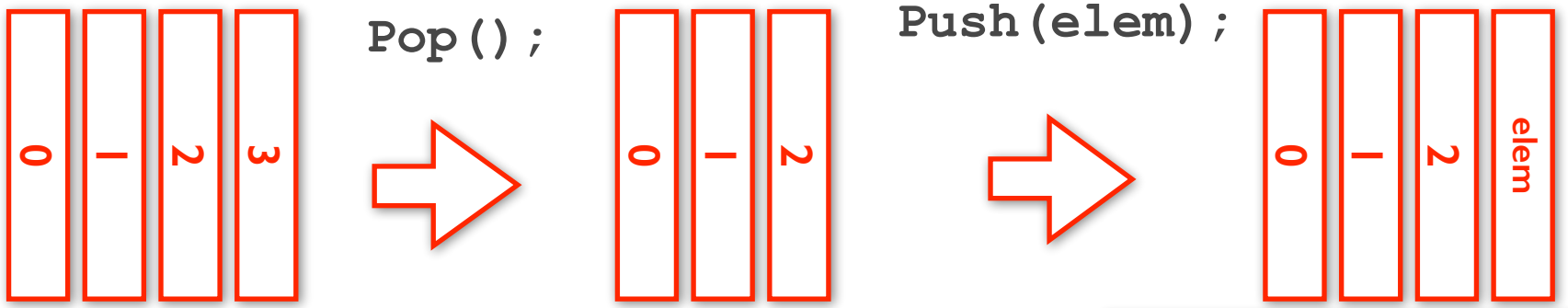


[練習] ex8A.cpp

【練習】使用串列實作堆疊

- 練習使用 **std::list** 去實作堆疊

```
template <class ElemType>
class Stack {
 public:
  ElemType &Top();                          // 回傳頂端的元素
  const ElemType &Top() const;
  void Push(const ElemType &elem);// 插入元素於頂端
  void Pop();                               // 移除頂端的元素
  bool Empty() const;                       // 回傳是否為空
  int Size() const;                         // 回傳堆疊大小
};
```

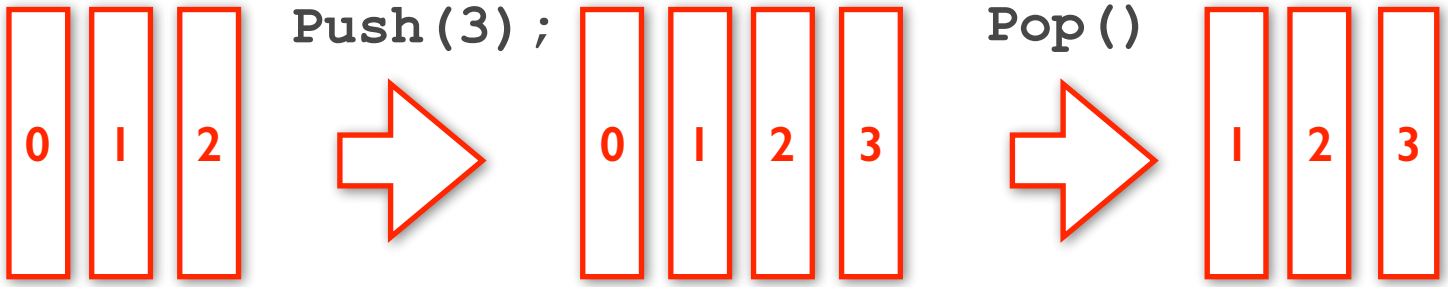


[練習] ex8B.cpp

佇列 (queue)

- 佇列是一個可以從前端移除元素和從尾端新增元素的資料結構
  - 可實現 **FIFO** (First-in first-out) 的機制

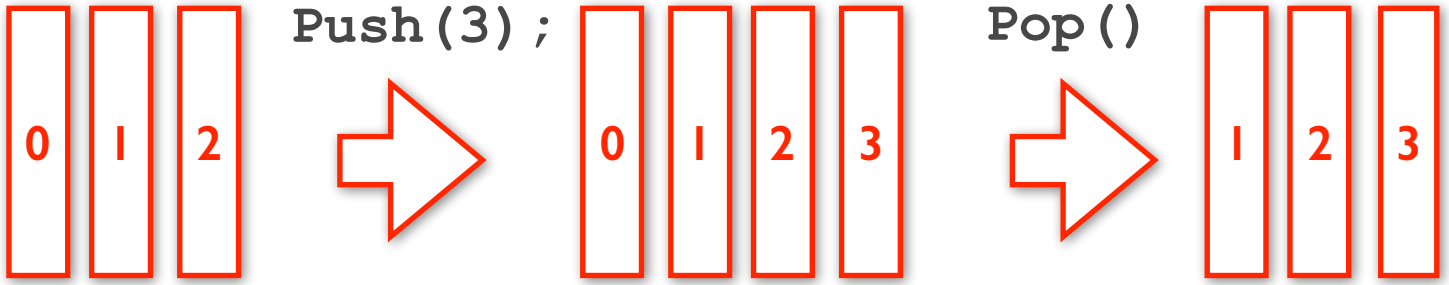
```
template <class ElemType>
class Queue {
 public:
  ElemType &Front();                          // 回傳前端的元素
  const ElemType &Front();
  void Push(const ElemType &elem);            // 新增元素於尾端
  void Pop();                                 // 移除前端的元素
  bool Empty() const;                         // 回傳是否為空
};
```



【練習】使用串列實作佇列

- 練習使用 **std::list** 去實作佇列：

```
template <class ElemType>
class Queue {
 public:
  ElemType &Front();                          // 回傳前端的元素
  const ElemType &Front() const;
  void Push(const ElemType &elem);            // 新增元素於尾端
  void Pop();                                 // 移除前端的元素
  bool Empty() const;                         // 回傳是否為空
  int Size() const;                           // 回傳佇列大小
};
```

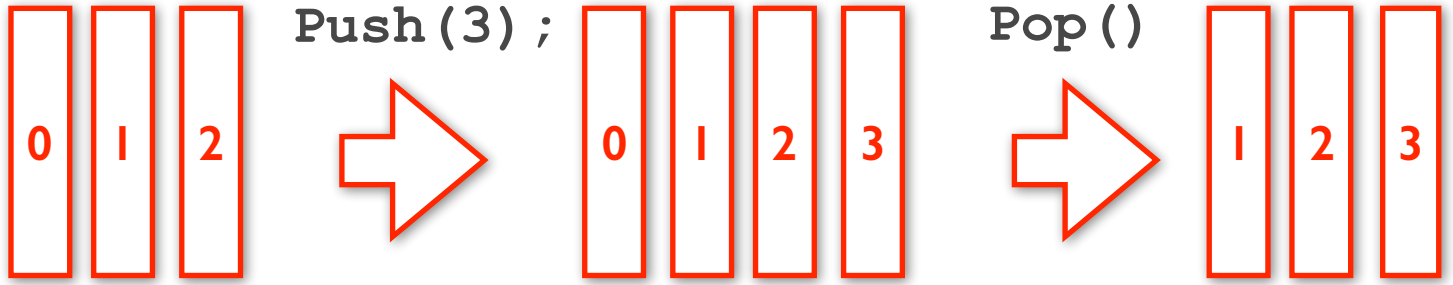


[練習] ex8C.cpp

【練習】使用陣列實作佇列

- 練習使用 **std::vector** 去實作佇列：

```
template <class ElemType>
class Queue {
 public:
  ElemType &Front();                          // 回傳前端的元素
  const ElemType &Front() const;
  void Push(const ElemType &elem);            // 新增元素於尾端
  void Pop();                                 // 移除前端的元素
  bool Empty() const;                         // 回傳是否為空
  int Size() const;                           // 回傳佇列大小
};
```



陣列適合實作佇列嗎？

[練習] ex8D.cpp

自適應容器 (adaptive container)

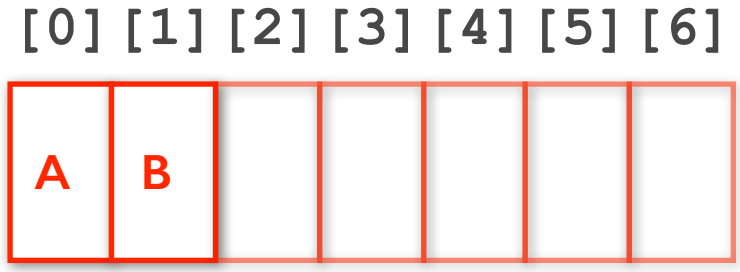
- 堆疊 (stack) 與佇列 (queue) 都是一種自適應容器 (adaptive container)。底層是由其他容器 (例如陣列或串列) 來實作的
  - 那為什麼我們不直接使用 **std::vector** 或 **std::list**？
- 堆疊 (stack) 與佇列 (queue) 定義了該容器需要支援的『操作介面』，底層的實作可以視情況作選擇，只要是該底層容器支援所需要的操作即可
  - 我們可以先試著使用堆疊或佇列來完成你的程式邏輯，但是如果你的程式邏輯牽涉到例如任意元素的存取則你會發現堆疊與佇列就不敷使用。

雙邊結尾佇列 (deque)

- 雙邊結尾佇列 **(deque: double-ended queue)** 是一種在頭尾都可以加入或刪除元素的資料結構：

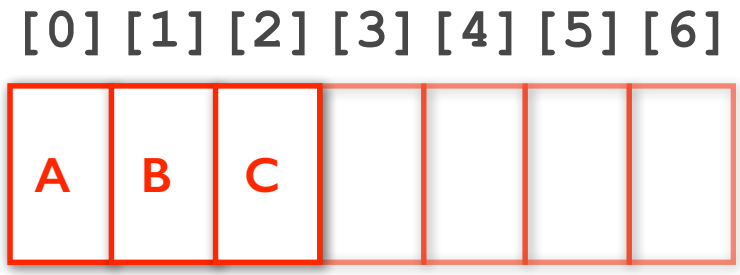
```
template <class ElemType>
class Deque {
 public:
   ElemType &Front();
   const ElemType &Front() const;
   ElemType &Back();
   const ElemType &Back() const;
   void PushFront(const ElemType &elem);
   void PopFront();
   void PushBack(const ElemType &elem);
   void PopBack();
   bool Empty() const;
   int Size() const;
};
```

雙邊結尾佇列的簡易陣列實作



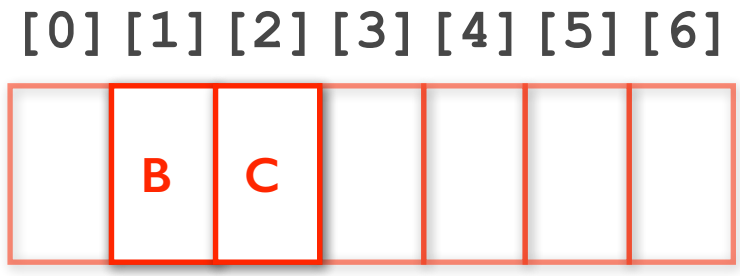

```
front_ == 0
back_  == 1
```

PushBack(C);




```
front_ == 0
back_  == 2
```

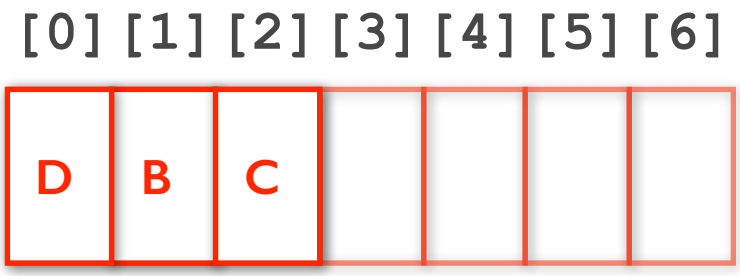
PopFront();




```
front_ == 1
back_  == 2
```

如果新增元素超過邊界大小怎麼辦？

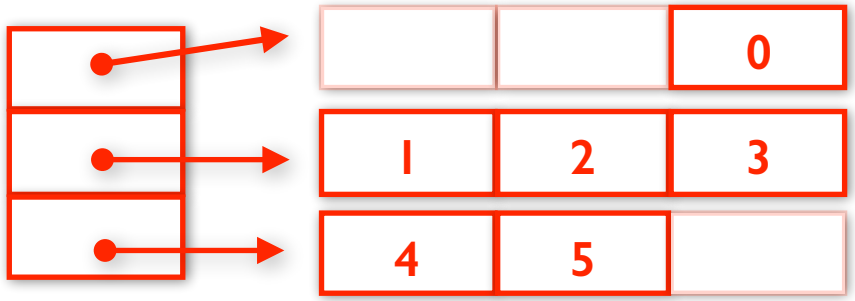
PushFront(D);




```
front_ == 0
back_  == 2
```

雙邊結尾佇列的常見實作

- 使用雙向鏈結串列 **(std::list)**
  - 隨機存取或當元素大小不大的時候效率差
- 使用可變大小陣列 **(std::vector)** 且陣列內每個元素為固定大小陣列



如果新增元素超過邊界大小怎麼辦？

- 在 **C++ STL** 中 **std::deque** 為某種特殊的實作而不跟堆疊或佇列一樣是自適應容器 **(adaptive container)**
  - **std::deque** 也支援迭代器、元素的新增刪除或改變大小等 **std::vector** 或 **std::list** 支援的操作

std::array 及 std::forward_list 是 C++11 才有的

C++ STL 內建容器

std::deque 的效率要看編譯器實作方法而定

性質	陣列 (std::array)	可變大小陣列 (std::vector)	單向鏈結串列 (std::forward_list)	雙向鏈結串列 (std::list)	雙邊結尾佇列 (std::deque)
size(), empty() (大小等資訊)	支援 (快)	支援 (快)	支援 (快)	支援 (快)	支援 (快)
at(), operator[]() (隨機存取)	支援 (快)	支援 (快)	不支援	不支援	支援 (可)
push_front(), pop_front() (在前端新增或移除)	不支援	不支援	支援 (快)	支援 (快)	支援 (可)
push_back(), pop_back() (在尾端新增或移除)	不支援	支援 (可)	不支援	支援 (快)	支援 (可)
insert(), erase() (在任意位置新增或移除)	不支援	支援 (慢)	支援 (快)	支援 (快)	支援 (可)
iterator (迭代器)	支援 (隨機)	支援 (隨機)	支援 (單向)	支援 (雙向)	支援 (隨機)

容器效率的考量與基本原則

- 當容器大小編譯時已知且不會改變時使用 **std::array**
- 當容器大小執行時才知道或會改變的情況下：
  - **std::vector** 與 **std::deque** 適用於：
    - 經常隨機存取
      - 存取多的話 **std::vector** 優於 **std::deque**
    - 元素複製成本低（例如元素大小小且複製建構式簡單等）或不常插入刪除元素
      - 新增刪除元素次數多的話 **std::deque** 優於 **std::vector**
      - 但是建構 **std::deque** 物件花的時間比 **std::vector** 高且善用 **std::vector::reserve()** 的話會大幅加速 **std::vector** 的時間
  - **std::list** 適用於
    - 不常隨機與少量循序存取
      - 就算是循序存取實務上也會有快取區域性 (locality) 較差的問題
    - 元素複製成本高且經常插入或刪除元素

std::vector | std::deque | std::list

使用 STL 的堆疊與佇列

- 堆疊 (**stack**)：
  - 在 **C++ STL** 的 **<stack>** 內有 **std::stack**
    - 可選用 **std::deque** (預設), **std::vector** 或 **std::list** 作為容器

```
stack<int> a;                    // 使用 deque<int> 實作堆疊 a
stack<int, vector<int> > b; // 使用 vector<int> 實作堆疊 b
```

- 佇列 (**queue**)：
  - 在 **C++ STL** 的 **<queue>** 內有 **std::queue**
    - 可選用 **std::deque** (預設) 或 **std::list** 作為容器

```
queue<int> a;                    // 使用 deque<int> 實作佇列 a
queue<int, list<int> > b;    // 使用 list<int> 實作佇列 b
```

C++ STL 自適應容器

是否支援的性質	堆疊 (std::stack)	佇列 (std::queue)
size(), empty() (大小等資訊)	支援	支援
at(), operator[]() (隨機存取)	不支援	不支援
push() (新增元素)	支援 (尾端)	支援 (尾端)
pop() (移除元素)	支援 (尾端)	支援 (前端)
insert(), erase() (在任意位置新增或移除)	不支援	不支援
iterator (迭代器)	不支援	不支援
可選 **std::vector** 實作	支援	不支援
可選 **std::list** 實作	支援	支援
可選 **std::deque** 實作	支援	支援

【範例】四則運算

- 我們想要寫一個可以做複雜四則運算的運算器
  - 先乘除後加減！
- 一般我們表示數學運算式的時候是使用中序 (運算子在兩個運算元中間)，現在我們改用後序輸入數學運算式 (運算子在兩個運算元後面)：

```
3 + 4 =                          3 4 + =
3 + 4 - 5 =                      3 4 + 5 - =
4 + ( 1 + 3 ) * 6 - 7 =          4 1 3 + 6 * + 7 - =
```

- 為了簡化程式，每個運算子或運算元都會以空白字元隔開。因此我們可以使用 `string` 將其一個一個讀入。

[範例] `cal.cpp`

std::stringstream 類別

- **<sstream>** 內的 **std::stringstream** 可以用來將 **string** 內容轉換為其他型態：

```
std:string str = “30”;
std::stringstream ss(str);
int v;
ss >> v;
```

  - 或者我們可以簡寫成：

```
int v;
std::stringstream(“30”) >> v;
```

產生佇列 q 和堆疊 s
無條件執行：
    讀入一個單元 e
    如果 e 是 "=":
        當 !s.empty()：
            q.push(s.top());
            s.pop();
        q.push(e);
        break;
    如果 e 是 "(":
        s.push(e);
    如果 e 是 ")":
        當 s.top() 不是 "(":
            q.push(s.top());
            s.pop();
        s.pop();
    如果 e 是 "+" 或 "-" 或 "*" 或 "/":
        當 s 不為空 且 s.top() 的優先順序大於或等於 e 時：
            q.push(s.top());
            s.pop();
        s.push(e);
    如果 e 是運算元 (以上皆非)：
        q.push(e);

4 + (1 + 3) * 6 - 7 =

q: 4 1 3 + 6 * + 7 - =

將中序轉為後序的演算法

stack：裝優先順序的運算子
queue：裝運算元

優先順序：* / 優於 + - 優於 (

【練習】四則運算

- 試著將左方的中序表示法改為右方的後序表示法進而利用前面的堆疊程式去做計算得初最後的結果：

```
3 + 4 =
3 + 4 - 5 =
4 + ( 1 + 3 ) * 6 - 7 =
```

```
3 4 + =
3 4 + 5 - =
4 1 3 + 6 * + 7 - =
```

[練習] ex8E.cpp