【第四講】

陣列

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

課程大綱

- 動態配置：`new` 與 `delete`
- 陣列簡介與實作
  - C 陣列的問題
  - 設計類別來模擬陣列
  - 避免透過建構式隱性轉型
  - 物件的『複製』：複製建構式與賦值運算子
  - 大三法則 (Rule of Three)
  - 自我賦值
- **C++ STL 的字串 (`std::string`)**
- 可變動大小的陣列
  - 可變動大小陣列的優化

動態配置：new 與 delete [1]

- 自動配置的方法：
  - 所有資料型態的變數會自動在定義時配置記憶體，在結束生命週期時釋放記憶體
  - 自訂型態 (類別和結構等) 的變數 (物件)，在配置記憶體後還會自動呼叫建構式，在釋放記憶體前會自動呼叫解構式
- 動態配置的方法：
  - 在 C 語言裡，我們使用 **malloc** 與 **free** 兩種函式來配置與釋放動態記憶體。
  - 在 C++ 語言裡，我們使用 **new** 與 **delete** 兩種運算子來配置與釋放物件
    - 配置物件時會配置記憶體與呼叫建構式： `Grade *g = new Grade();`
    - 釋放物件時會釋放記憶體與呼叫解構式： `delete g;`

動態配置：new 與 delete [2]

- 動態配置一樣可以呼叫不同參數的建構式：

```
Grade *a = new Grade;     // 呼叫 Grade();[預設建構式]
Grade *b = new Grade();   // 呼叫 Grade();[預設建構式]
Grade *c = new Grade(3); // 呼叫 Grade(int);
```

- 動態配置也可以配置陣列，但此時也只能呼叫預設建構式：

```
Grade *f = new Grade[3]; // 呼叫三次 Grade();
```

  - 指標可以當做以所指向物件當做第一個元素的陣列用：

```
f[1] = 30;                // (f+1)->Set(30);
```

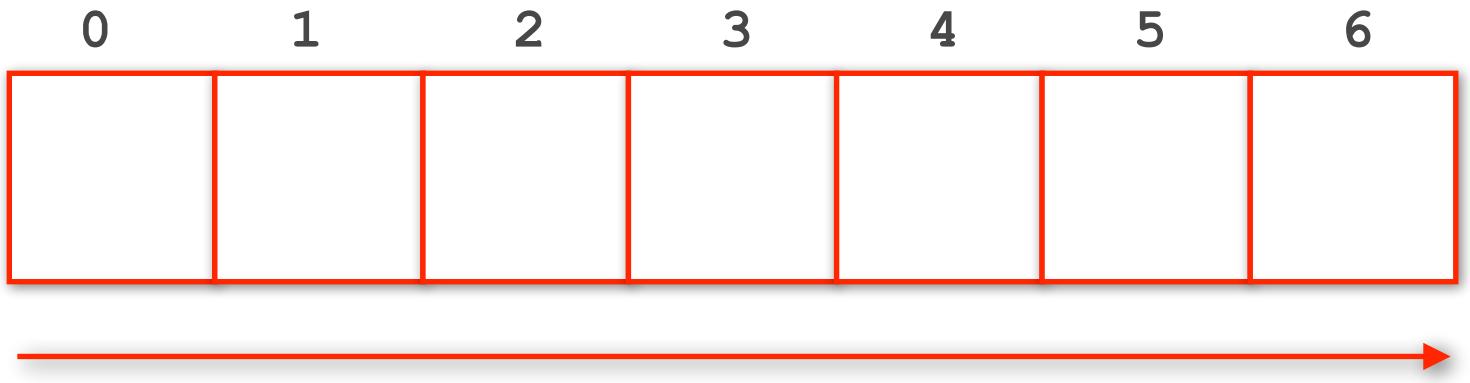
  - 釋放配置的陣列時要用 **delete[]** ：

```
delete[] f;               // 呼叫三次 ~Grade();
```

[範例] dynamic.cpp

【範例】陣列簡介與實作

- 什麼是陣列？
  - 一種將固定數量的同型元素依照線性連續儲存的資料結構，可以用特定的索引值或鍵值做『快速存取』
    - 固定數量？
    - 線性儲存 → 快速存取



【思考】為什麼陣列可以做快速存取？

C 陣列的問題

- **C / C++ 語言有內建陣列這種資料結構，但是相對而言受到了一些限制：**
  - 例如：陣列無法直接做複製：

```
int v[5] = {1, 2, 3, 4, 5};
int n[5];
n = v;          // [編譯錯誤]
```

  - 替代方案：
    1. 使用迴圈或呼叫函式做複製
    2. 使用指標去指向陣列的開頭元素後當陣列用
  - 兩種替代方案使用上都不夠直覺！
- **因此我們可以設計一個類別來模擬陣列的效果，好處是可以再加上我們需要的新功能 (例如複製)**

設計類別來模擬陣列 [1]

- 我們先想好能怎麼使用 C 陣列：
  - 我們可以怎麼使用 C 陣列？

```
int a[10];                  // 定義時指定陣列大小
a[3] = 7;                   // 指定其中一個元素的值
cout << a[3] << endl; // 取得其中一個元素的值
```

  - 我們不可以怎樣使用 C 陣列？

```
int b[];                    // [編譯錯誤] 定義時需指定陣列大小
```

設計類別來模擬陣列 [2]

- 要怎麼設計類別來模擬陣列的使用：
  - 我們可以怎麼使用這個類別？

```
IntArray a(10);                 // 建構時指定陣列大小
a.At(3) = 7;                    // 指定其中一個元素的值
cout << a.At(3) << endl;        // 取得其中一個元素的值
```

  - 我們不可以怎樣使用這個類別？

```
IntArray b;                 // [編譯錯誤] 建構時需指定陣列大小
```

  - 此外，我們還希望新增下面幾種 C 陣列沒有的用法：

```
cout << a.Size() << endl;   // 取得陣列大小
IntArray b = a;             // 初始化時複製陣列
b = a;                      // 賦值運算時複製陣列
```

```
class IntArray {
 public:
  // 產生大小為 n 的陣列
  IntArray(int n);

  // 回傳陣列大小
  int Size();

  // 存取第 i 號陣列元素
  int At(int i);
};

int main() {
  IntArray a(10);
  cout << a.Size() << endl;
  a.At(3) = 7;                    // [編譯錯誤] 怎麼辦?
  cout << a.At(3) << endl;
  IntArray b = 6;                 // 什麼意思?
  IntArray c;                     // [編譯錯誤]
  const IntArray &d = a;
  cout << d.Size() << endl;       // [編譯錯誤] 怎麼辦?
  cout << d.At(3) << endl;        // [編譯錯誤]
  return 0;
}
```

利用剛歸納的結果設計類別的公開介面

在設計階段時我們可以先不實作函式成員的定義內容

先想好要怎麼『使用』這個類別

[範例] design_1.cpp

避免透過建構式隱性轉型

- 具有單一參數的建構式預設可用來作為隱性轉型用：

```
class IntArray {
 public:
  //可以用這個建構式隱性將 int 轉型為 IntArray
  IntArray(int n);
};
```

  - 因此：

```
IntArray a = 10; 相當於 IntArray a = IntArray(10);
```

- 想避免單一參數的建構式自動作為隱性轉型用的話，要在建構式前面加上 **explicit** 修飾字：

```
class IntArray {
 public:
  explicit IntArray(int n);
};
```

沒特殊理由的話我們單一參數的建構式都要加 explicit

```
class IntArray {
 public:
 // 產生大小為 n 的陣列
  explicit IntArray(int n);
  // 回傳陣列大小
  int Size() const;
  // 存取第 i 號陣列元素
  int &At(int i);
  int At(int i) const;
};

int main() {
  IntArray a(10);
  cout << a.Size() << endl;
  a.At(3) = 7;                          // [1], [4]
  cout << a.At(3) << endl;              // [4]
  IntArray b = 6;                       // [2] [編譯錯誤]
  IntArray c;                           // [編譯錯誤]
  const IntArray &d = a;
  cout << d.Size() << endl;             // [3]
  cout << d.At(3) << endl;              // [4]
  return 0;
}
```

[2] 為什麼加 explicit ?

[3] 為什麼加上 const ?

[1] 為什麼回傳參考 (&) ?

[4] 為什麼需要兩個 At() ?

[範例] design_2.cpp

```
class IntArray {
 public:
  // 產生大小為 n 的陣列
  explicit IntArray(int n);

  // 回傳陣列大小
  int Size() const;

  // 存取第 i 號陣列元素
  int &At(int i);
  int At(int i) const;
  int &operator[](int i);
  int operator[](int i) const;
};
int main() {
  IntArray a(10);
  a[3] = 7;
  cout << a[3] << endl;
  const IntArray &d = a;
  cout << d.Size() << endl;
  cout << d[3] << endl;
  return 0;
}
```

使用運算子多載

[範例] design_3.cpp

實作 IntArray 類別

- 完成設計後，接著就是去實作這個設計好的類別
  - 需要實作所有成員函式的定義內容，在實作的過程中有必要可以幫該類別新增私有 (private) 成員
- 可以從比較直觀的成員函式開始實作：
  - **`Size()`**
    - 新增私有資料成員 int size_ , 然後想想什麼時候會改到 size_ ?
  - **`At(int)`**
    - 新增私有資料成員 int *data_, 然後想想什麼時候會改到 data_ ?
  - **`operator[](int)`**
    - 用現成的操作 (At) 去實作！

[範例] `array_1.cpp`

記憶體洩漏 (memory leak)

- 我們在 **IntArray(int)** 裡面呼叫了 **new[]** 運算子去配置陣列，所以要記得在 **~IntArray()** 裡面呼叫 **delete[]** 運算子去釋放陣列
  - 如果不釋放會怎樣？
  - 當我們動態配置記憶體時 (使用 **new**)，在正常情況下就意味著我們要動態的釋放記憶體 (必須要使用到 **delete**)
    - 通常當你寫了一個 **new** 運算就表示你需要寫一個 **delete** 運算
    - 如果在建構式裡面使用到 **new** ，通常也意味著會在解構式裡面使用到 **delete**

[範例] array_2.cpp

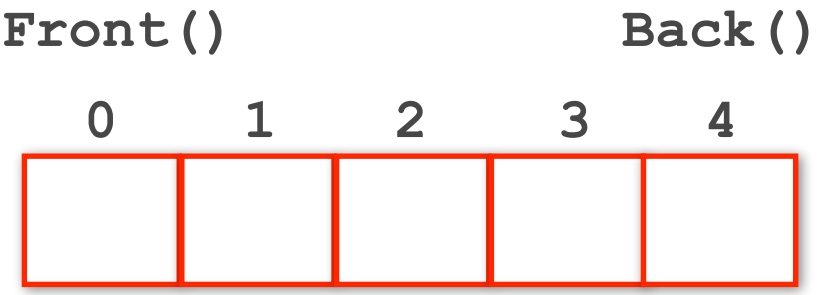
【練習】新增操作

- 在 **IntArray** 內新增下列操作：

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

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

為什麼需要兩個同名函式？



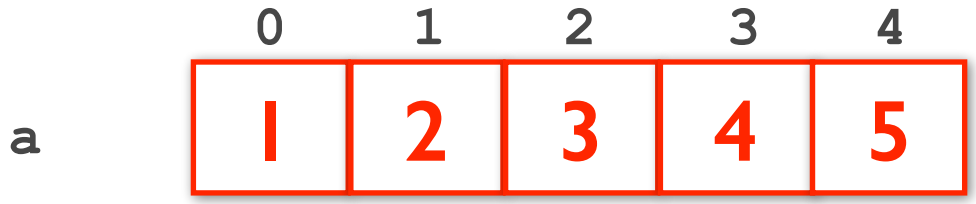
[練習] ex4A.cpp

【練習】新增操作

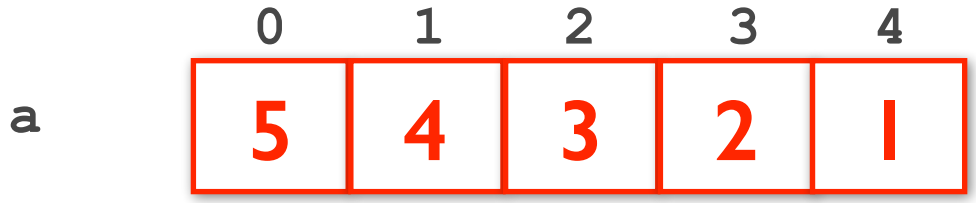
- 在 **IntArray** 內新增下列操作：

```
// 交換 IntArray 的內容
void Swap(IntArray &x);

// 反轉 IntArray 的內容
void Reverse();
```




```
a.Reverse();
```



[練習] ex4B.cpp

物件的『複製』

- 我們會在「用同型物件做初始化」與「賦值運算」時『複製』物件：

```
Grade a(30);
Grade b = a;        // 用同型物件做初始化(呼叫複製建構式)
// 上面這寫法會隱性轉成 Grade b(a);

b = a;                  // 賦值運算(呼叫賦值運算子)
// 上面這寫法會呼叫 b.operator=(a) 或 operator=(b, a)
```

  - 上面這例子會用到：
    - 複製建構式 **Grade(const Grade &)**
    - 賦值運算子 **Grade &Grade::operator=(const Grade &)**
  - 要注意到這兩個是不同的

複製建構式

- 複製建構式就是以同型物件作為參數的建構式：
  - 例如，對 Grade 類別而言, `Grade(const Grade&)` 就是他的複製建構式
  - 複製建構式的參數必定為同型物件的參考
  - 此外設計上複製建構式通常都是固定值參考

```
Grade a(30);
Grade b(a);          // 用同型物件做初始化(呼叫複製建構式)

Grade c = a;         // 用同型物件做初始化(呼叫複製建構式)
// 上面這寫法會隱性呼叫 Grade(const Grade&)
```

- 如果我們沒有替類別設計複製建構式，則會有一個預設的複製建構式。預設的複製建構式會幫你直接複製所有資料成員

[範例] copyctor.cpp

賦值運算子

- **operator=** 是賦值運算子，我們可對他作多載：

```
Grade a(30);        // 呼叫 Grade(int)
Grade b;            // 呼叫 Grade(); [預設建構式]

b = a;              // 1] 呼叫 operator=(b, a) 或者
                    // 2] 呼叫 b.operator=(a)
```

  - 一般而言，我們會使用上述的第二個方式。
    - 因為賦值運算通常會修改到類別的私有成員！
  - 賦值運算子支援串接，所以回傳值型態會是第一個運算元的參考型態，而在成員函式內其值為 ***this**：

```
Grade c;
c = b = a = 3;          // 這樣串接會呼叫哪些函式？
```

- 同樣的，當你沒幫類別設計賦值運算子多載時，預設的賦值運算子會複製所有資料成員

[範例] copyop.cpp

大三法則

- **大三法則 (Rule of Three)**：當類別定義了下列三個特殊函式的其中一個時，通常其他兩個也都要同時被定義：解構式、複製建構式與賦值運算子。
  - 例如當資料成員中有指標型態時，預設行為是：
    - **解構式**：什麼都不做。此時指標指向的記憶體並不會被釋放，可能造成記憶體洩漏 (memory leak)
    - **複製建構式**：直接複製資料成員。此時指標所指向的記憶體位址也會直接被複製過去，但不是真的複製一份指向的值 (shallow copy)
    - **賦值運算子**：直接複製資料成員。此時原本指標指向的記憶體可能還沒被釋放但指標所指向的位址已經被改變了，會造成記憶體洩漏 (memory leak)

```
IntArray a(10);
IntArray b(a);   // 呼叫了哪個函式？(複製建構式)
IntArray c = a; // 呼叫了哪個函式？(複製建構式)
c = a;          // 呼叫了哪個函式？(複製指定運算子)
c[0] = 3;       // 此時 a[0] 跟 b[0] 會變成 3 嗎 ?
```

自我賦值

- 想想下面這個程式碼會做什麼？

```
IntArray a(10);
a[3] = 7;
a = a;                          // 呼叫賦值運算子
```

  - 當發生自我賦值的時候會呼叫 `a.operator=(a);`
  - 賦值運算子一般要做的事情有兩個：
    - 清除自己原本的資料
    - 將另一個物件的資料複製過來
  - 自我賦值時因為自己跟被複製的物件指的都是同一份，所以在「清除自己原本的資料」後，資料就不見了！
  - 所以在賦值運算子中通常會檢查是否是自我賦值，如果是的話通常就不做清除跟複製：

```
if (this != &rhs) { /* 清除資料、複製資料 */ }
```

【補充】這不是最完美的做法

```
class IntArray {
 public:
  // 複製建構式
  IntArray(const IntArray &rhs);
  // 產生大小為 n 的陣列
  explicit IntArray(int n);
  // 解構式
  ~IntArray();

  // 回傳陣列大小
  int Size() const;

  // 存取第 i 個陣列元素
  int &At(int i);
  int At(int) const;
  int &operator[](int i);
  int operator[](int i) const;

  // 賦值運算子
  IntArray &operator=(const IntArray &rhs);
};
```

這些設計細節請熟記！

大三法則 (1): 複製建構式

禁止隱性轉型

大三法則 (2): 解構式

加 const 讓固定值物件可以使用

回傳參考讓他可以放在等號左邊

加 const 讓固定值物件可以使用

大三法則 (3): 賦值運算子

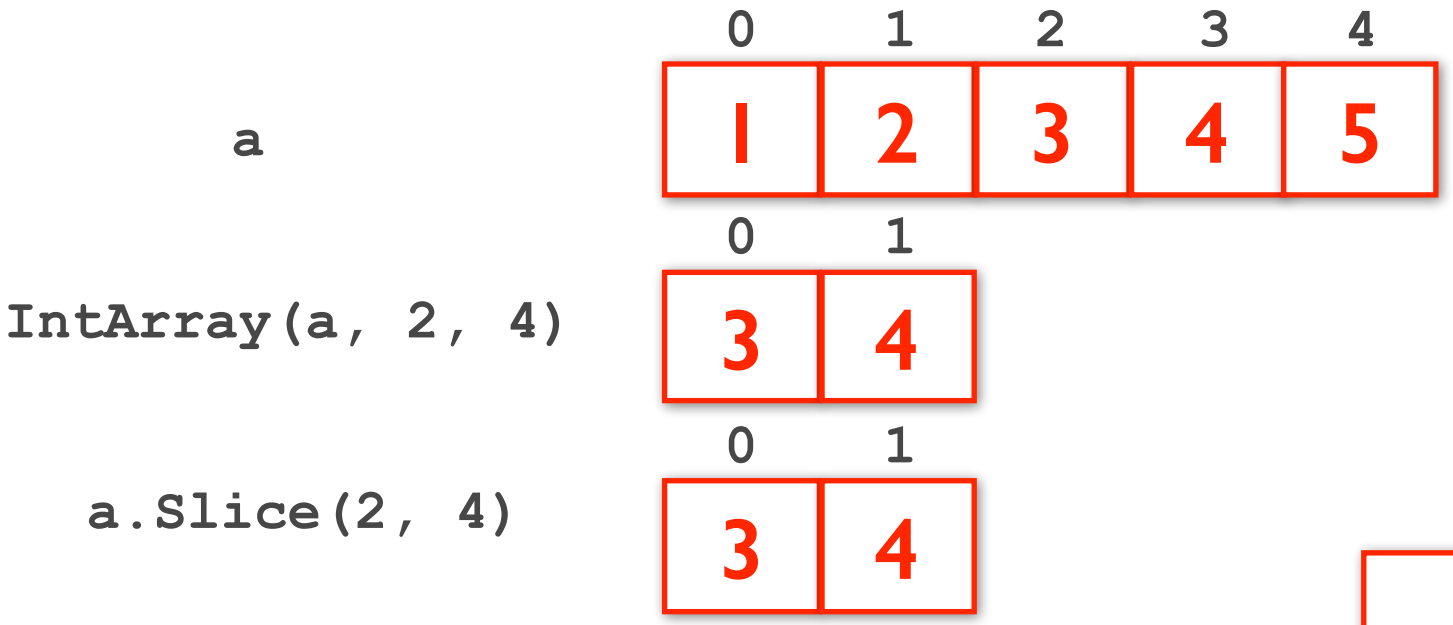
[範例] array_3.cpp

【練習】新增操作

- 在 **IntArray** 內新增下列操作：

```
// 建構式: 產生由 x[begin] 到 x[end-1] 構成的陣列
IntArray(const IntArray &x, int begin, int end);

// 回傳由第 begin 號到 end-1 號元素構成的陣列
const IntArray Slice(int begin, int end);
```



[練習] ex4C.cpp

【練習】字串類別

```
class String {
 public:
  String();
  String(const String &s);
  String(const char* s);
  ~String();
  int Size() const;
  char &At(int i);
  char At(int i) const;
  char &operator[](int i);
  char operator[](int i) const;
  String &operator=(const String &rhs);
};

const String operator+(const String &lhs,
                       const String &rhs);
```

這裡我們故意將 operator+ 作為非成員函式

[練習] ex4D.cpp

【範例】C++ STL 的字串

- 字串處理：
  - 在 C 裡面我們使用字元陣列或字元指標配合 <cstring> 內的函式去操作字串。
  - 在 C++ 內我們可以使用 <string> 內的 std::string 來表示或操作字串。
- 試著去看參考文件來大略了解 `std::string` 的用法！
  - http://www.cplusplus.com/reference/string/string/
  - 先看提供的建構式 (constructor): 可以怎麼產生字串
  - 再看提供的運算子: 可以對字串做什麼運算
  - 最後看提供的操作: 其他特殊的操作

[範例] string.cpp

【範例】可變動大小的陣列

```
class IntVector {
 public:
  IntVector();                    // 產生大小為 0 的陣列
  IntVector(const IntVector &rhs);
  explicit IntVector(int n);
  ~IntVector();

  int Size();
  // 修改陣列長度為 n
  void Resize(int n);

  int &At(int i);
  int At(int i) const;
  int &operator[](int i);
  int operator[](int i) const;
  IntVector &operator=(const IntVector &rhs);
};
```

[範例] vector_1.cpp

【練習】新增操作

- 在 IntVector 內新增下列功能：

```
// 清除內容 (大小改為零)
void Clear();

// 插入 elem 元素到陣列的最後面 (陣列大小會多一)
void PushBack(int elem);

// 刪除陣列的最後一個元素 (陣列大小會少一)
void PopBack();
```

[練習] ex4E.cpp

【練習】新增操作

- 在 IntVector 內新增下列功能：

```
// 插入 elem 元素到陣列的最前面 (陣列大小會多一)
void PushFront(int elem);

// 刪除陣列的第一個元素 (陣列大小會少一)
void PopFront();
```

[練習] ex4F.cpp

【練習】新增操作

- 在 IntVector 內新增下列功能：

```
// 插入新元素 elem 至編號 pos 的位置
// (後面的元素往後移動, 陣列大小會多一)
void Insert(int pos, int elem);

// 移除編號 pos 位置的元素
// (後面的元素往前移動，陣列大小會少一)
void Erase(int pos);
```

[練習] ex4G.cpp

【範例】可變大小陣列的優化

- 在配置記憶體時我們可以事先配置多一點，等到真的不夠用再重新配置，可以減少重新配置的次數：
  - 那要多配置多少？

```
// 配置至少可存放 n 個元素的空間
void Reserve(int n);

// 回傳已經配置的空間可以放多少個元素
int Capacity() const;
```

  - 有多少的成員函式要修改？

[範例] vector_2.cpp