Schedule Review

W	Date	Lecture	Homework
1	09/11, 09/15	Lec01: Course Information, Environment Introduction, and OOP Concept	Homework 00 (Fri.)
2	09/18, 09/22	Lec01: OOP Concept / Lec02: Class Introduction	
3	09/25, **09/29**	Lec02: Class Introduction, and Essential STL Introduction / No class due to Moon Festival	Homework 01 (Mon.)
4	10/02, 10/06	Lec02: Class Introduction, and Essential STL Introduction	
5	**10/09**, 10/13	No class due to the bridge holiday of Nation day / Lec03: Encapsulation	Homework 02 (Mon.)
6	10/16, 10/20	Lec03: Encapsulation / Lec04: Inheritance	
7	10/23, 10/27	Lec04: Inheritance	Homework 03 (Mon.)
8	10/30, 11/03	Lec04: Inheritance	
9	**11/06**, **11/10**	**Physical Hand-Written Midterm** / **Physical Computer-based Midterm**	Midterm
10	11/13, 11/17	Lec05: Polymorphism	
11	11/20, 11/24	Lec05: Polymorphism	Homework 05 (Mon.)
12	11/27, 12/01	Lec05: Polymorphism	
13	12/04, 12/08	Lec06: Composition & Interface	Homework 06 (Mon.)
14	12/11, 12/15	Lec06: Composition & Interface	
15	12/18, 12/22	Lec07: Efficiency + Dependency Injection	Homework 07 (Mon.)
16	12/25, 12/29	**Physical Hand-Written Final**, Flexible time	
17	**01/01**, 01/05	No class / **Physical Computer-based Final**	Final
18	01/08, 01/12	**No class here**	

Object Oriented Programming

Lec02: Essential STL

Sun Chin-Yu（孫勤昱）

cysun@ntut.edu.tw

2023/09/22

大綱

- 什麼是 STL
- 實用 STL
  - `std::string`
  - `std::vector`
  - `std::shared_ptr`

什麼是 STL

什麼是 STL（1/3）

- Standard Template Library [1]
  - 中文翻譯為標準樣（模）板函式庫、泛型函式庫
  - 惠普實驗室開發，１９８８年成為國際標準，正式成為Ｃ＋＋函式庫之重要成員
    - Alexander Stepanov

Alexander Alexandrovich Stepanov（俄語：Алекса́ндр Алекса́ндрович Степа́нов；1950 年 11 月 16 日出生於莫斯科）是一位俄裔美國電腦程式設計師，最著名的是標準程式設計人員 的主要設計者者，[1]他於 1992 年左右在HP 實驗室工作期間開始開發該軟體。早些時候，他曾在貝爾實驗室工作，與Andrew Koenig關係密切，並試圖說服Bjarne Stroustrup在 C++ 中引入Ada泛型之類的東西。[2]他被譽為概念概念的提出者。[3] [4]

他（與Paul McJones）是《程式設計元素》[5]的作者，該書源自Stepanov 在AdAdobe Systems（受僱期間）教授的「程式設計基礎」課程[6]。他也與 Daniel E. Rose 合著了《從數學到通用編程》一書。[7]

他於 2016 年 1 月從A9.com退休。[8]

什麼是 STL（2/3）

- 它提供了多種模板類和函數，用於方便地操作資料結構和演算法
  - 容器（Containers）：用來儲存資料的特殊結構，例如常見的vector, list, set, map等
  - 演算法（Algorithms）：如排序、搜索、和資料操作等相關演算法
  - 迭代器（Iterators）：用於訪問容器中的元素，例如begin(), end()等
  - 分配器（Allocators）：用於定義如何分配和釋放記憶體的物件
  - 配接器（Adapters）：提供一種方式，使得我們可以在現有的容器（如 vector ）上，用不同的方式操作這些容器
  - 函式對象（Functors）：又稱為仿函數，使得演算法的行為可以根據需要進行自定義（Override）
- C++的內建函式，方便我們撰寫code

什麼是 STL（3/3）

- 你可以用這些網站來取得 STL 的資源，認識一些有趣的 STL 來幫助你解決問題
  - https://cplusplus.com/reference/
  - https://en.cppreference.com/w/

TAIPEI TECH 國立臺北科技大學 National Taipei University of Technology

實用 STL
(std::string 與 std::vector)

Information Security LAB

字串（std::string）

- 用字串來儲存文字
- 透過字串，我們可以對文字進行處理
  - 例如取得字串長度、型態轉換、拼接

```cpp
#include <iostream>
#include <string>

int main(){
    std::string str = "Hello World"; // 宣告字串並賦值為 Hello World
    std::cout << str << std::endl;   // 印出字串
}
```

取得字串（std::string）長度

- 使用 `length()` 函數來取得字串長度

```cpp
#include <iostream>
#include <string>

int main(){
    std::string str = "Hello World";    // 宣告字串並賦值為 Hello World
    size_t length = str.length();       // 取得字串長度
    std::cout << length << std::endl;   // 印出字串長度（11）
}
```

字串（std::string）拼接

- 我們可以使用運算子 “+” 來對任意兩個字串進行拼接

```cpp
#include <string>

int main(){
    std::string a = "Hello";
    std::string b = "World!";
    std::string c = a + b; // "HelloWorld!"
}
```

`std::vector`

- 中文：**向量** Vector
  - 不是指數學上的向量，而是指一個可以動態調整大小的陣列（array）
  - 動態記憶體陣列
    - `std::vector` 使用動態記憶體陣列，使陣列長度是可變化的
    - 比起之前需要分配記憶體來創建陣列（固定大小）Ex. int a[5];
- **基本上可以取代所有的陣列**
  - **安全：**使用 std::vector 通常比使用原始陣列更安全，因為不太可能超出其邊界。另外，它自動管理記憶體，減少了記憶體洩露的可能性。
  - **容易操作：** std::vector 提供了一系列的成員函式，如 push_back(), pop_back(), size(), capacity(), 和 resize()，這使得操作相對容易。

std::vector

- 如何使用它?
- std::vector<**int**> 主要泛指這個動態記憶體陣列的型態
- 注意，std::vector使用大括號來存放元素
  - 非中括號 [ ]

```
1  #include <vector>
2
3  int main(){
4      std::vector<int> vec = {1, 2, 3, 4, 5};
5  }
```

std::vector

- 如果我們有一個元素 `6` 想要接至 `std::vector` 的尾端，
- 我們可以使用 `push_back` 來完成。

```cpp
#include <vector>

int main(){
    std::vector<int> vec = {1, 2, 3, 4, 5};
    vec.push_back(6); // [1, 2, 3, 4, 5, 6]
}
```

std::vector

- 取得陣列長度：使用 `size()` 來取得。

```cpp
#include <vector>

int main(){
    std::vector<int> vec = {1, 2, 3, 4, 5};
    vec.push_back(6);      // [1, 2, 3, 4, 5, 6]
    int size = vec.size(); // size = 6
}
```

實用 STL
(std::shared_ptr)

為什麼我們會需要 shared pointer?

- 以 C++ 來說，指標的釋放會是一個很大的議題，沒做好會導致 memory leak。

為什麼我們會需要 shared pointer?

- 記憶體沒有管理好
  - Memory Leak帶來的危害（實際課程案例）
    - 下學期的物件導向程式設計實習，如果沒有處理好 Memory Leak 的問題
    - 助教的電腦會藍屏，因為記憶體過度被占用導致電腦卡到無法處理其他程式
  - C/C++常發生的Double free跟Use-After-Free的安全漏洞
    - 安全程式設計
  - 70%的Security Bugs來自於記憶體管理不當（微軟案例）[1]

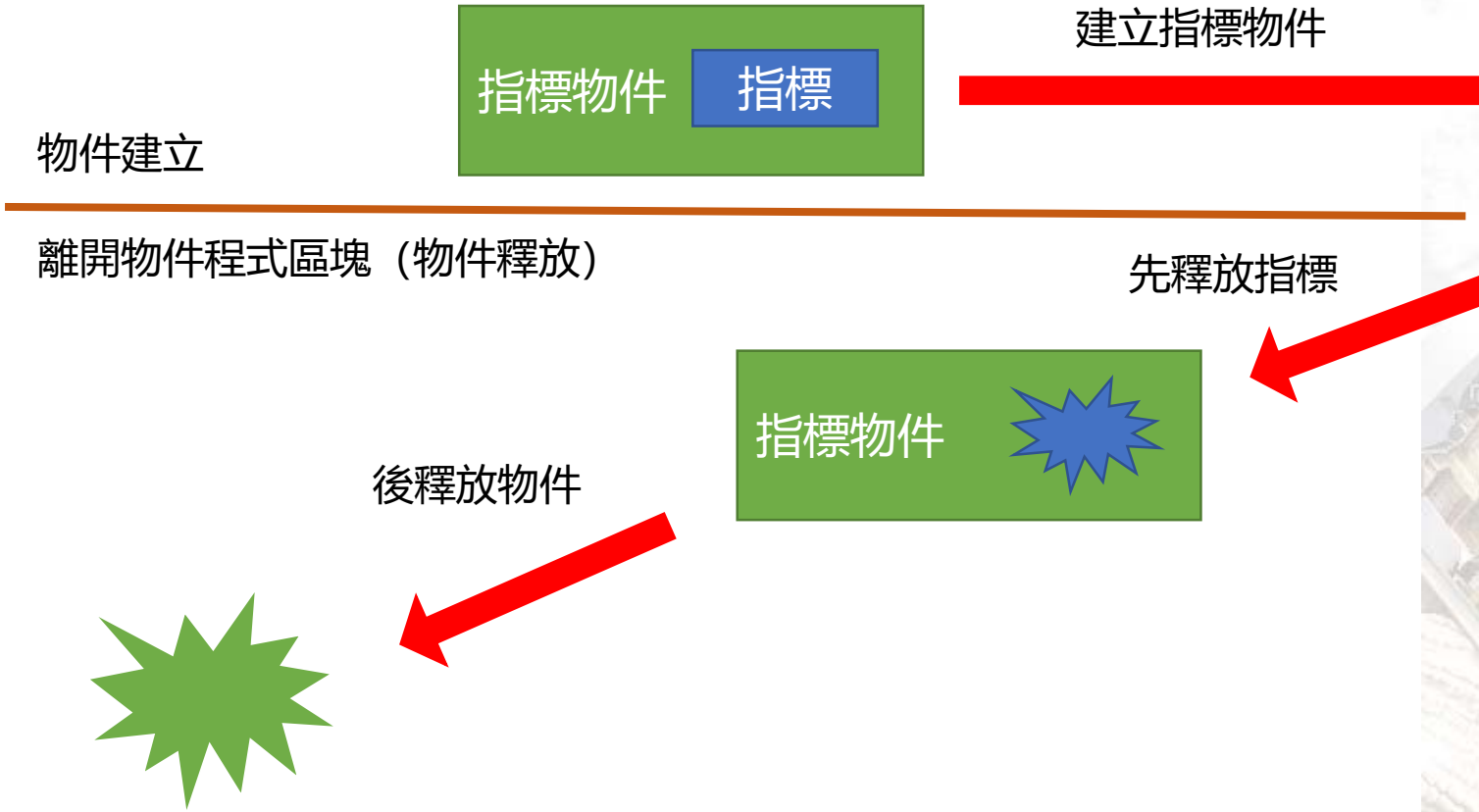
[1] https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/

std::shared_ptr

- 以物件的方式來處理指標
- 由於物件會隨著生命週期被釋放，因此以物件包裹的指標來說，我們可以仰賴生命週期來使用物件的方式將指標釋放，不用手動釋放或卡在 Memory Leak 地獄

```
1  template <typename T>
2  class SmartPtr {
3  public:
4      // 建構子，用於建立指標物件
5      SmartPtr(T *ptr = nullptr) { m_Ptr = ptr; }
6
7      // 解構子，用於釋放指標
8      ~SmartPtr() { delete m_Ptr; }
9
10     T GetValue() const { return *m_Ptr; }
11     void SetValue(T value) { *m_Ptr = value; }
12
13 private:
14     T *m_Ptr;
15 };
16
```

std::shared_ptr




```cpp
template <typename T>
class SmartPtr {
public:
    // 建構子，用於建立指標物件
    SmartPtr(T *ptr = nullptr) { m_Ptr = ptr; }

    // 解構子，用於釋放指標
    ~SmartPtr() { delete m_Ptr; }

    T GetValue() const { return *m_Ptr; }
    void SetValue(T value) { *m_Ptr = value; }

private:
    T *m_Ptr;
};
```

為什麼我們會需要 shared pointer?

- 用這一個方式可以不用手動釋放記憶體空間
- 將所有的指標視同物件使用
- 使記憶體管理更加方便

Thanks!