面向對象 深度解析
OO Inside Out (12 hours)

補充講義

侯捷

2024/04/22-23, Intel 上海 (9:30-12:00; 13:00-16:30)
2024/04/25-26, Intel 北京 (9:30-12:00; 13:00-16:30)

```
 7  class MyString {
 8  private:
 9      char* _data;
```

```
37      //move ctor, with "noexcept"
38      MyString(MyString&& str) noexcept :
39          _data(str._data), _len(str._len)  {
40          ++MCtor;
41          str._len = 0;
42          str._data = nullptr;    //重要
43      }
```

```
56      //move assignment, with noexcept
57      MyString& operator=(MyString&& str) noex
58          ++MAsgn;
59          if (this != &str) {
60              if (_data) delete[] _data;
61              _len = str._len;
62              _data = str._data;  //MOVE!
63              str._len = 0;
64              str._data = nullptr; //重要
65          }
66          return *this;
67      }
```

move aware

```
983     iterator
        insert(const_iterator __position, const value_type& __x);
```

move aware

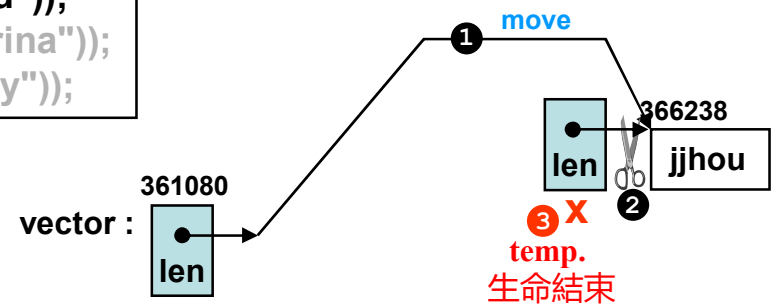
```
1014    iterator
        insert(const_iterator __position, value_type&& __x)
1015    { return emplace(__position, std::move(__x)); }
```

```
 1  #include ".\myString.h"
 2  #include <ctime>    //clock(), clock_t
 3  #include <cstring>  //strlen()
 4  #include <cstdio>   //snprintf()
 5  using namespace std;
 6
 7  enum RV { Rvalue, Lvalue };
 8
 9  template<typename Container>
10  void test_moveable(Container& cntr, long times, RV option)
11  {
12  typedef typename
13          iterator_traits<typename \
14          Container::iterator>::value_type ElemType;
15  typedef typename Container::value_type ElemType2;
16  //二述兩個 types 相同
17
18  char buf[10];
19
20      clock_t timeStart = clock();
21      for (long i=0; i< times; ++i) {
22          snprintf(buf, 10, "%d", rand());  //隨機數 (轉為字
23          auto itr = cntr.end();    // 定位於尾端
24          if (Rvalue == option)
25              cntr.insert(itr, ElemType(buf)); //所有容器都支
26          else { // (Lvalue == option)
27              ElemType elem(buf);
28              cntr.insert(itr, elem); //所有容器都支持 insert
29          }
30      }
31      cout << "milli-seconds : " << (clock()-timeStart) << en
```

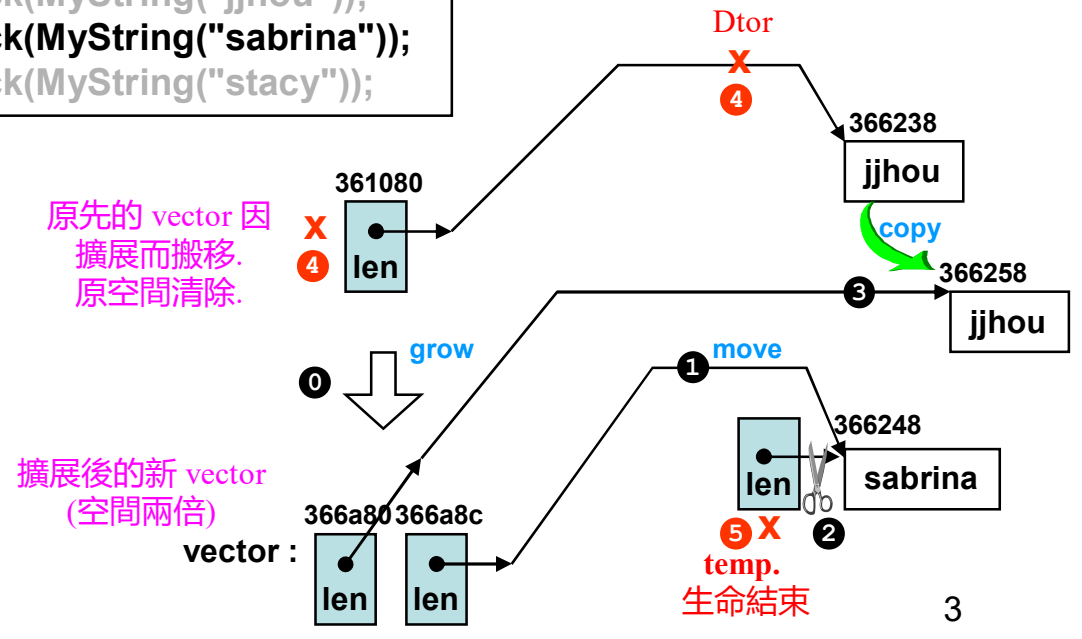
move-aware MyString, move functions without noexcept, 圖1/2

```
class MyString {
public:
  MyString(MyString&& str) noexcept
    { ... }
  MyString& operator=(MyString&& str) noexcept
    { … }
  virtual ~MyString() noexcept
    { … }
  …
};
```

```
vector<MyString> vec;

vec.push_back(MyString("jjhou"));
vec.push_back(MyString("sabrina"));
vec.push_back(MyString("stacy"));
```



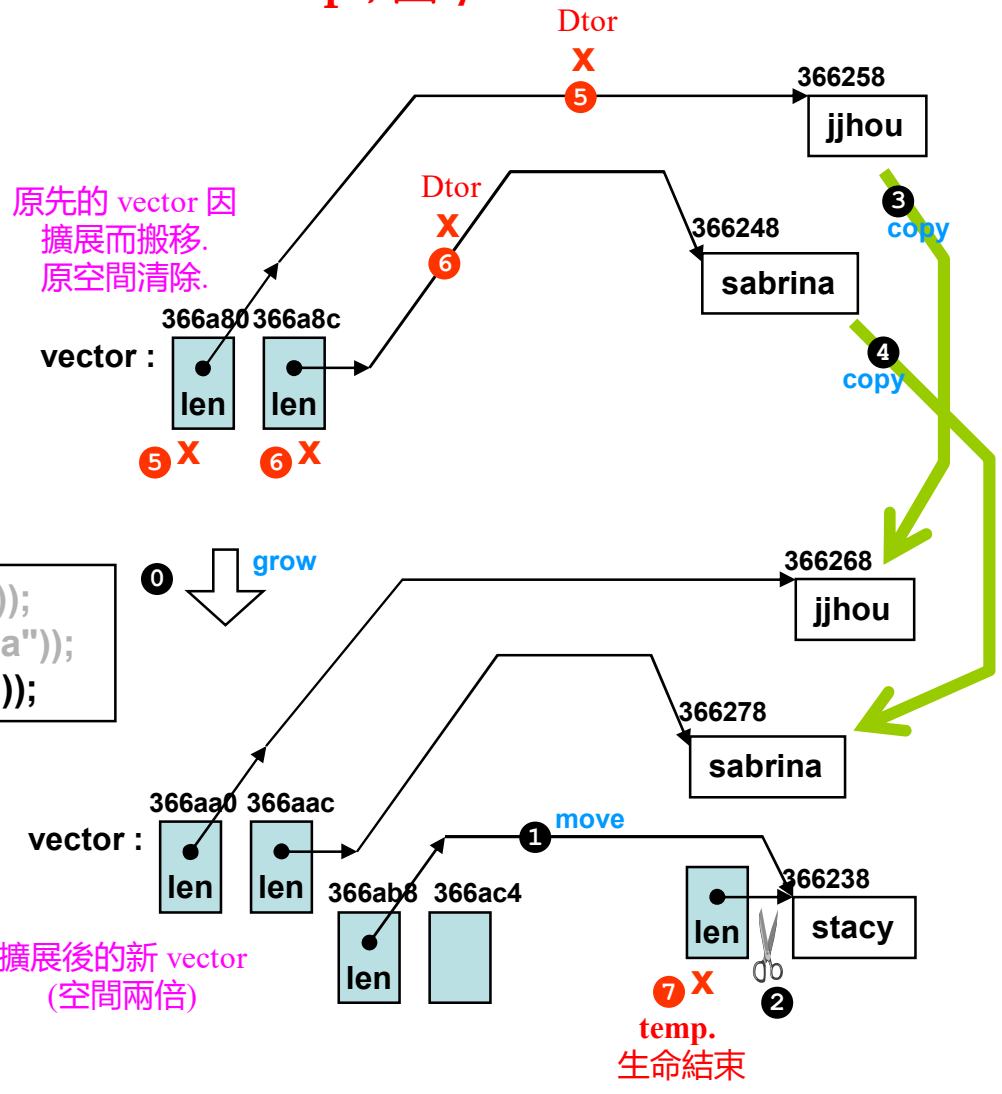

```
vec.push_back(MyString("jjhou"));
vec.push_back(MyString("sabrina"));
vec.push_back(MyString("stacy"));
```



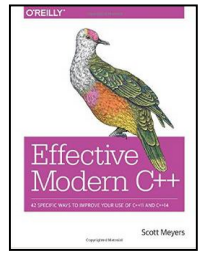
move-aware MyString, move functions without noexcept, 圖2/2

```
class MyString {
public:
  MyString(MyString&& str) noexcept
    { ... }
  MyString& operator=(MyString&& str) noexcept
    { … }
  virtual ~MyString() noexcept
    { … }
  …
};
```

```
vec.push_back(MyString("jjhou"));
vec.push_back(MyString("sabrina"));
vec.push_back(MyString("stacy"));
```

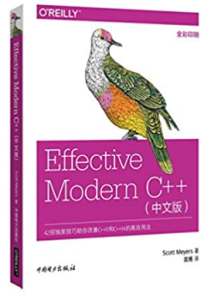


Item 24: Distinguish universal references from rvalue references.




```
void f(Widget&& param);              // 右值引用

Widget&& var1 = Widget();            // 右值引用

auto&& var2 = var1;                  // 非右值引用

template<typename T>
void f(std::vector<T>&& param);      // 右值引用
```

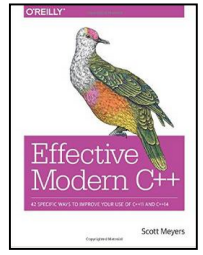
```
template<typename T>
void f(T&& param);                   // 非右值引用
```



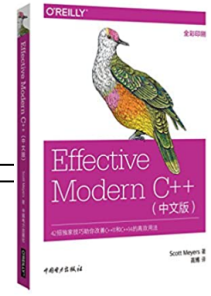
实际上，“T&&”有两种不同的含义。其中一种含义，理所当然，是右值引用。正如期望，它们仅仅会绑定到右值，而其主要的存在理由（raison d'être），在于识别出可移对象。

“T&&”的另一种含义，则表示其既可以是右值引用，亦可以是左值引用，二者居一。带有这种含义的引用在代码中形如右值引用（即T&&），但它们可以像左值引用一样运作（即T&）。这种双重特性使之既可以绑定到右值（如右值引用），也可以绑定到左值（如左值引用）。犹有进者，它们也可以绑定到const对象或非const对象，以及volatile对象或非volatile对象，甚至绑定到那些既带有const又带有volatile饰词的对象。它们几乎可以绑定到万事万物。这种拥有史无前例的灵活性的引用值得拥有一个独特的名字。我称之为万能引用（universal reference）。[注1]

Item 24: Distinguish universal references from rvalue references.




```
void f(Widget&& param);        // 不涉及型别推导；
                               // param是个右值引用

Widget&& var1 = Widget();      // 不涉及型别推导
                               // var1是个右值引用
```




```
template<typename T>
void f(T&& param);             // param是个万能引用

Widget w;
f(w);                          // 左值被传递给f；
                               // param的型别是Widget&（即一个左值引用）

f(std::move(w));               // 右值被传递给f；
                               // param的型别是Widget&&（即一个右值引用）
```

若要使一个引用成为万能引用，其涉及型别推导是必要条件，但还不是充分条件。引用声明的形式也必须正确无误，并且该形式被限定得很死：必须得正好形如“T&&”

即使是一个const饰词的存在，也足以褫夺一个引用成为万能引用的资格：

```
template<typename T>
void f(const T&& param);       // param是个右值引用
```

Imperfect Forwarding

```
int a = 0;
process(a);          //process(int&): 0
                     //變數被視為 Lvalue 處理
process(1);          //process(int&&): 1
                     //temp. object 被視為 Rvalue 處理
process(move(a));    //process(int&&): 0
                     //move() 強制將 a 由 Lvalue 改為 Rvalue
forwarding(2);  //forwarding(int&&): 2,  process(int&): 2
   //Rvalue 經由 forwarding() 傳給另一函數卻變成 Lvalue.
   //(原因是傳遞過程中它成了一個 named object)
forwarding(move(a));    //forwarding(int&&): 0,  process(int&): 0
                     //Rvalue 經由 forwarding() 傳給另一函數卻變成了 Lvalue
//!  forwarding(a);    //[Error] cannot bind 'int' lvalue to 'int&&'
```

good

good

good

```
void process(int& i) {
   cout << "process(int&): " << i << endl;
}
```

```
void process(int&& i) {
   cout << "process(int&&): " << i << endl;
}
```

```
void forwarding(int&& i) {
   cout << "forwarding(int&&): " << i << " ";
   process(i);
}
```

i 是變數, 是 Lvalue, 所以**一定**調用左值版本, **這不完美**.

```
void forwarding2(int&& i) {
   cout << "forwarding2(int&&): " << i << " ";
   process( std::forward(i));
}
```

↑使用 std::forward() 將如何?
不, 根本不能這麼用, 編譯報錯.

函數的 Lvalue refernece 版本 可接受 Rvalue
(此所以前例 vector::insert(const T&) 可接受 MyString()).
函數的 Rvalue reference 版本 不接受 Lvalue
(此所以上例 forewarding(int&&) 不接受 int a.

Perfect Forwarding; Universal reference; std::forward

```
int a = 0;
process(a);          //process(int&): 0
                     //變數被視為 Lvalue 處理
process(1);          //process(int&&): 1
                     //temp. object 被視為 Rvalue 處理
process(move(a));    //process(int&&): 0
                     //move() 強制將 a 由 Lvalue 改為 Rvalue
forwarding(2);  //forwarding<T>(T&&): 2,  process(int&&): 2
   //Rvalue 經由 forwarding() 傳給另一函數仍保持 Rvalue.
forwarding(move(a)); //forwarding<T>(T&&): 0,  process(int&&): 0
                     //Rvalue 經由 forwarding() 傳給另一函數仍保持 Rvalue
forwarding(a);       //forwarding<T>(T&&): 0, process(int&): 0
                     //Lvalue 經由 forwarding() 傳給另一函數仍保持 Lvalue
```

good

```
void process(int& i) {
   cout << "process(int&): " << i << endl;
}
```

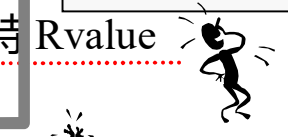
good

```
void process(int&& i) {
   cout << "process(int&&): " << i << endl;
}
```

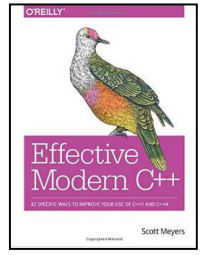
good

universal reference

```
template<typename T>
void forwarding(T&& val) {
   cout << "forwarding<T>(T&&):" << val << " ";
   process(std::forward<T>(val));
}
```



Item 25: Use std::move on rvalue references, std::forward on universal references.



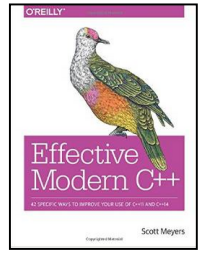

```
class Widget {
public:
  template<typename T>
  void setName(T&& newName)                    // newName是个
  { name = std::forward<T>(newName); }         // 万能引用

  ...
};
```

简而言之，当转发右值引用给其他函数时，应当对其实施向右值的无条件强制型别转换（通过std::move），因为它们一定绑定到右值；而当转发万能引用时，应当对其实施向右值的有条件强制型别转换（通过std::forward），因为它们不一定绑定到右值。

Things to Remember

- Apply `std::move` to rvalue references and `std::forward` to universal references the last time each is used.
- Do the same thing for rvalue references and universal references being returned from functions that return by value.
- Never apply `std::move` or `std::forward` to local objects if they would otherwise be eligible for the return value optimization.

Item 25: Use std::move on rvalue references, std::forward on universal references.




```
class Widget {
public:
  Widget(Widget&& rhs)                // rhs is rvalue reference
  : name(std::move(rhs.name)),
    p(std::move(rhs.p))
    { … }
  …

private:
  std::string name;
  std::shared_ptr<SomeDataStructure> p;
};
```

```
class Widget {
public:
  template<typename T>
  void setName(T&& newName)           // universal reference
  { name = std::move(newName); }      // compiles, but is
  …                                   // bad, bad, bad!

private:
  std::string name;
  std::shared_ptr<SomeDataStructure> p;
};
```

```
class Widget {
public:
  template<typename T>
  void setName(T&& newName)           // newName is
  { name = std::forward<T>(newName); }    // universal reference

  …
};
```

Object Delegation in C++: Object Delegation means using the object of another class as a class member of another class. It is known as object delegation. Below are some properties of the delegation:

- Delegation can be an alternative to inheritance, but in an inheritance, there is an is-a relationship, but in the delegation, there is no inheritance relationship between the classes.
- The Delegation allows us to use the properties of the particular class that is required in the class.
- Delegation can be viewed as a relationship between objects where one object forwards a certain method calls to another object, called its **delegate**.
- The primary advantage of delegation is run-time flexibility – the delegate can easily be changed during run-time.
- But unlike inheritance, delegation is not directly supported by most popular object-oriented languages, and it doesn't facilitate dynamic polymorphism.
  促進

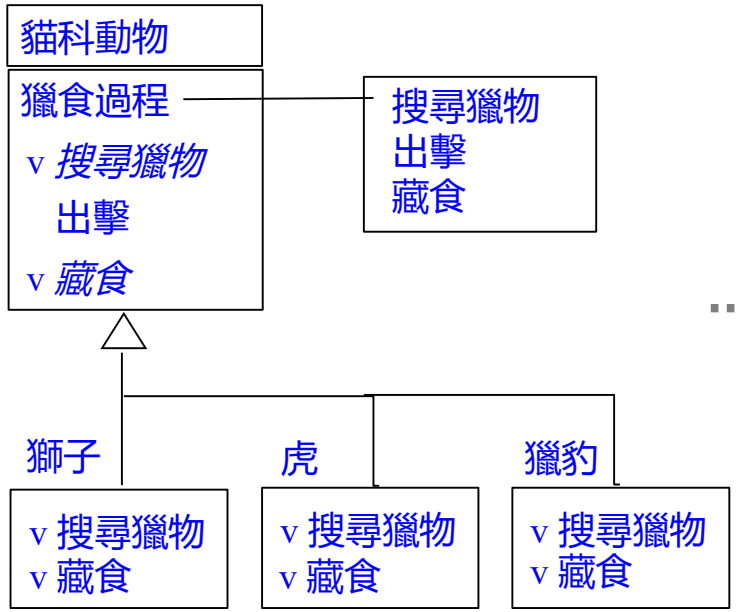
```cpp
// C++ program to illustrate the
// Object Delegation
#include <iostream>
using namespace std;
class First {
public:
    void print() { cout << "The Delegate"; }
};
class Second {
    // Creating instance of the class
    First ob;

public:
    void print() { ob.print(); }
};

// Driver Code
int main()
{
    Second ob1;
    ob1.print();
    return 0;
}
```

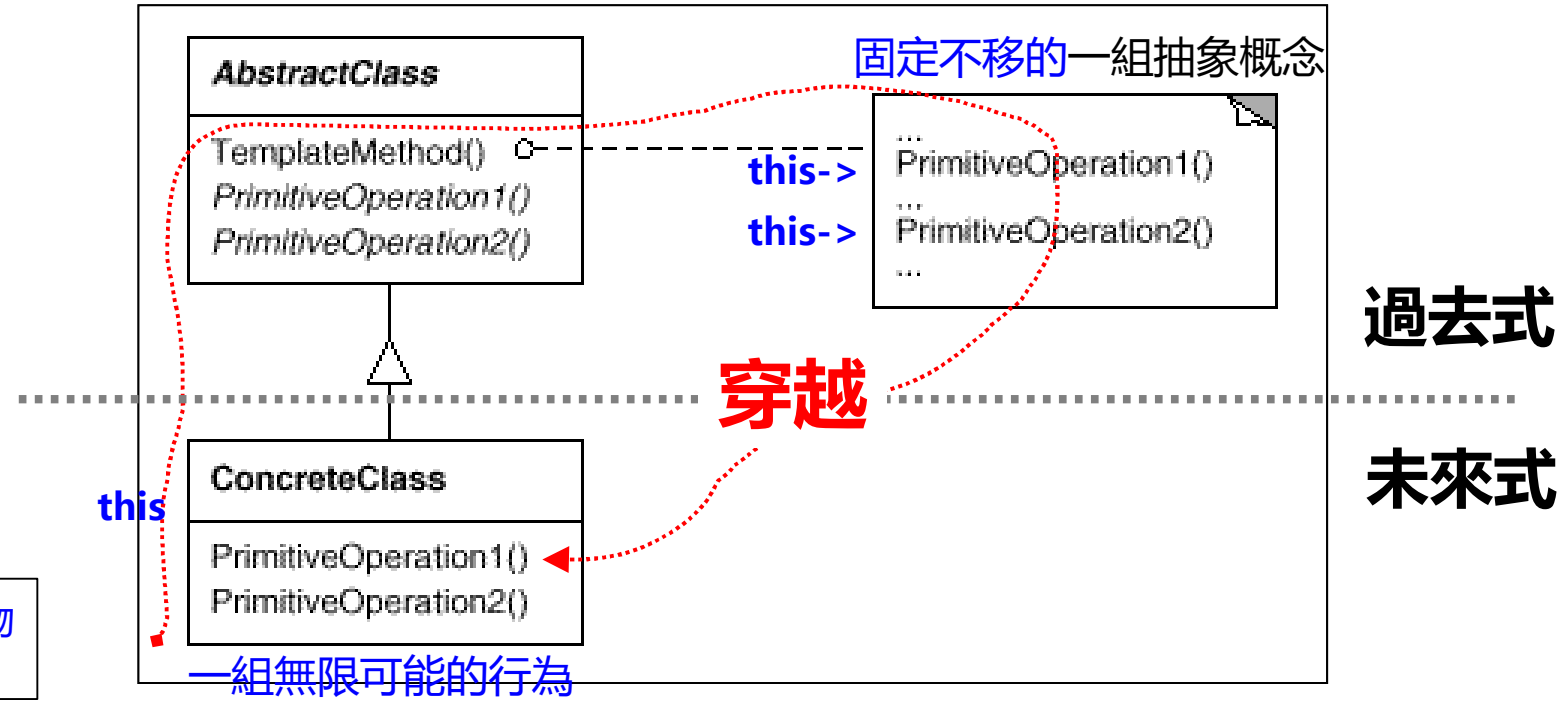
運行結果:

```
The Delegate
```

When to use what?
Here are some examples of when inheritance or delegation are being used:

- Assume class is called **B** and the derived/delegated to class is called **A**.
- If users want to express a relationship (is-a), then use inheritance.
- Users want to be able to pass the class to an existing API expecting **A**'s, then use inheritance.
- Users want to enhance **A**, but **A** is final and can no further be sub-classed than use composition and delegation.



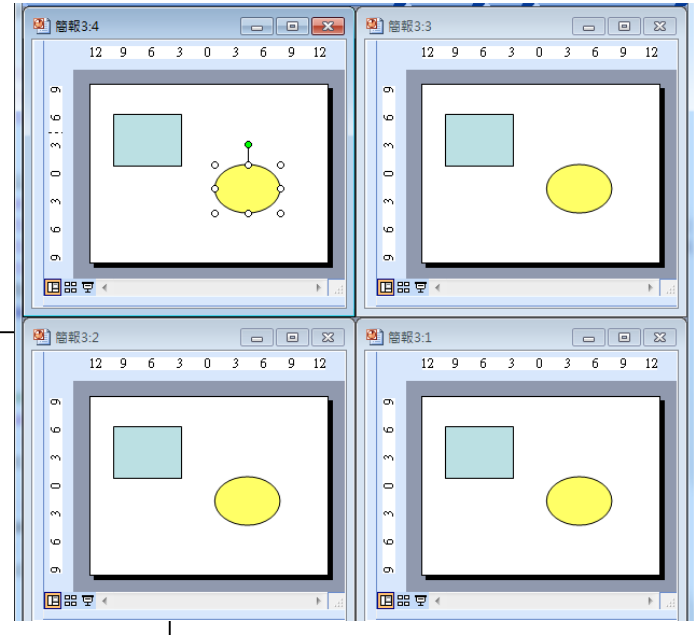


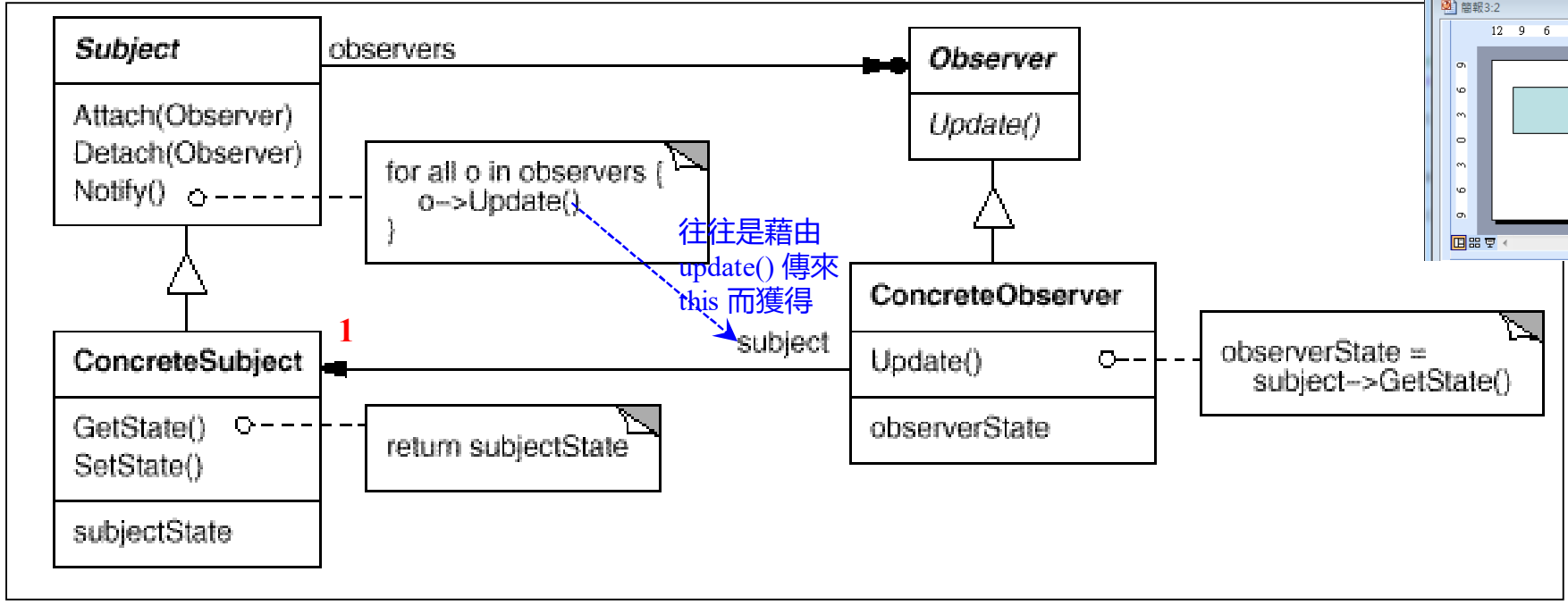

```
獅子  a;
a.獵食過程(&a);

虎 b;
b.獵食過程(&b);
//----------------------------------------
std::list<貓科動物*> myList;
myList.insert(&a);
myList.insert(&b);
…
a loop to iterate myList and call 獵食過程() of each elements
```

16. Observer

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

在 objects 之間定義 “一對多” 的依存關係，使得當有個 object 改變了它自身的 state，其所有依存者都會被通知並自動被更新。





The End