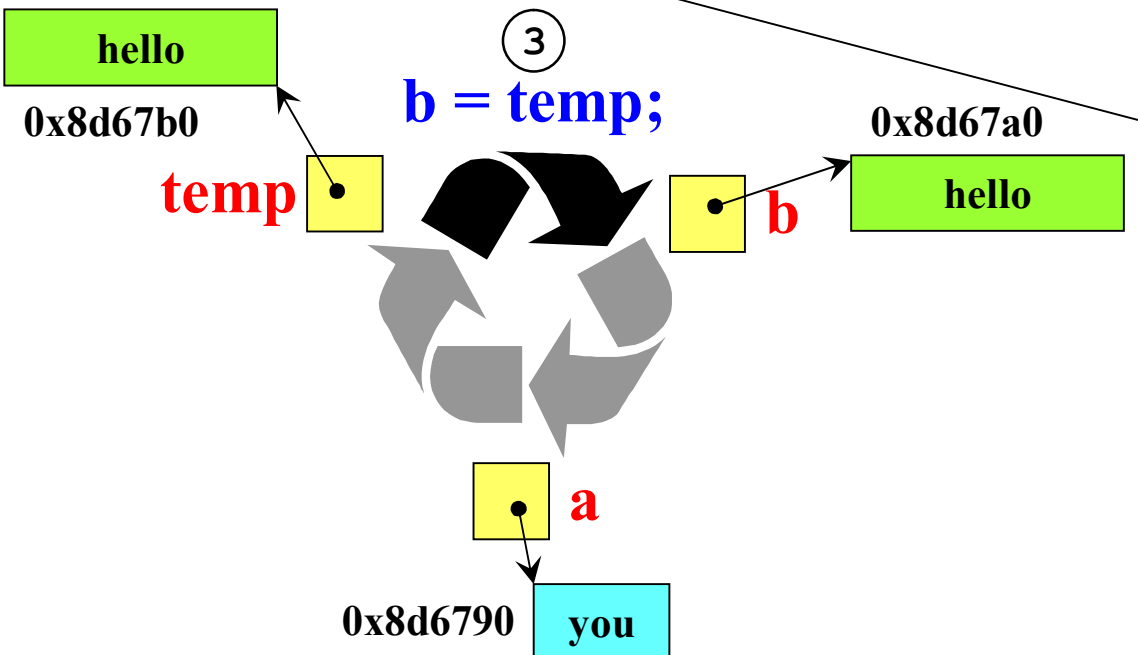


例 MyString without move

swap



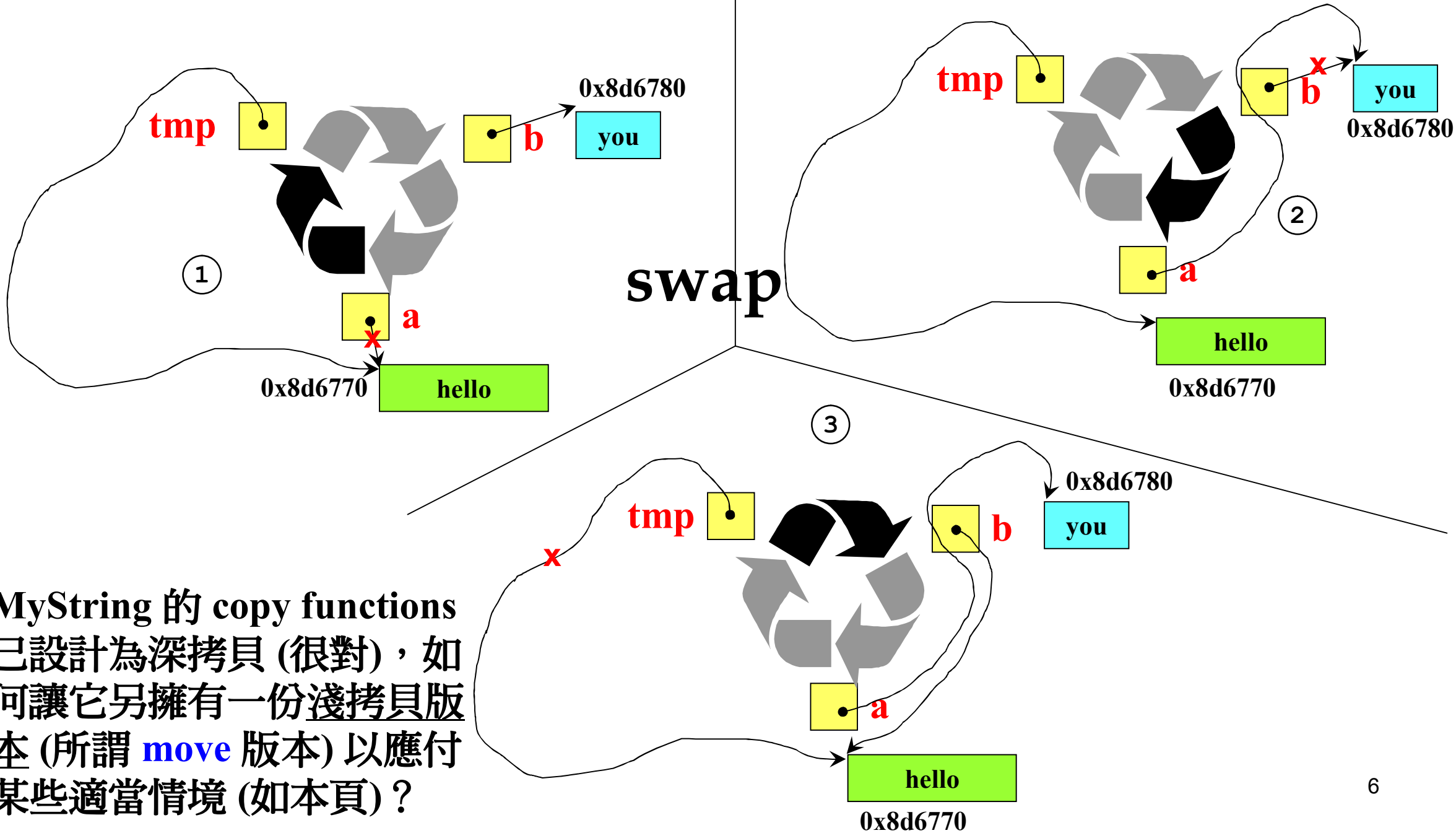
Until 2011

```

namespace std {
template<typename T>
void swap(T& a, T& b)
{
① T temp(a);
② a = b;
③ b = temp;
}
}

```

三個動作
都是 copy



MyString 的 copy functions 已設計為深拷貝 (很對)，如何讓它另擁有一份淺拷貝版本 (所謂 **move** 版本) 以應付某些適當情境 (如本頁)？

當 resource 可被「偷取、借用、移動 (move)」時...

C++ 必須讓我們能夠：

- 1, 告訴編譯器，目前待操作的是這種東西 (R-value)。
- 2, 告訴編譯器，這東西的確有設計出一套專門處理 move 的函數。

1, 右值 (R-value) 有天然和人工兩種。

→ 天然右值：temp. object, literal

→ 人工右值：**std::move**(x)

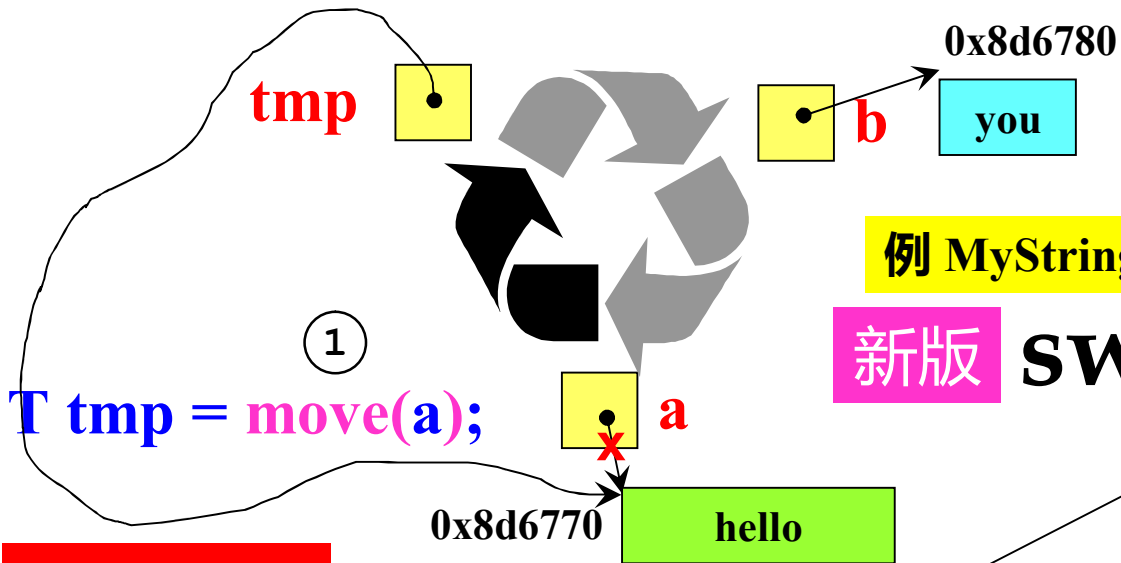
左值：可取址，有名稱
右值：不可取址，沒名稱

用後不打算再用

2, C++11 推出 R-value reference.

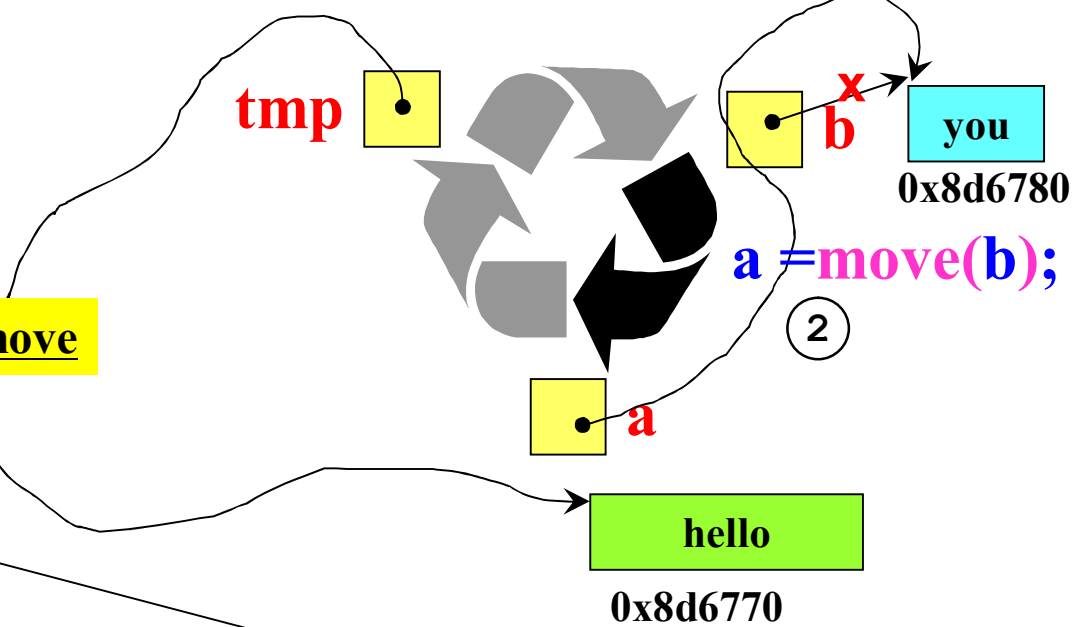
T(T&& x) ; move ctor

T& operator=(T&& x) ; move assignment



例 MyString with move

新版 swap



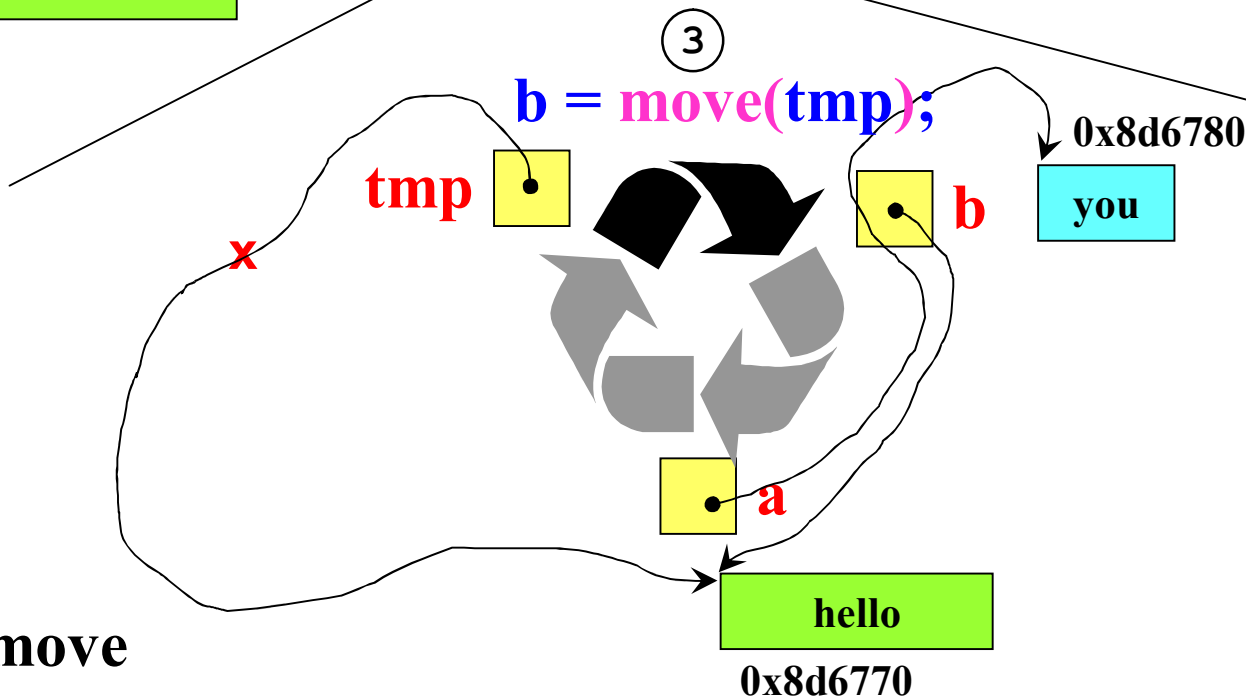
Since 2011

```

namespace std {
template<typename T>
void swap(T& a, T& b)
{
① T temp(move(a));
② a = move(b);
③ b = move(temp);
}
}

```

三個動作有可能是 move



Rvalue references 和 Move semantics

STL 容器

1

```
//copy  
insert(..., &x)
```

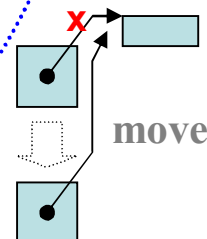
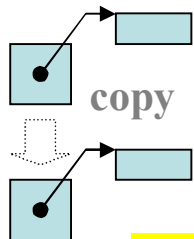


2

```
//move  
insert(..., &&x)
```

Class 的設計者

```
class MyString {  
private:  
    char* _data;  
    ...  
public:  
    //copy operations  
    MyString(const MyString& str) ...  
  
    //move operations  
    MyString(MyString&& str) ...  
  
    //dtor  
    ~MyString() ...  
};
```



Class 的使用者

```
c.insert(iter, MyString("hello"));  
  
MyString str("hello");  
c.insert(iter, std::move(str));
```

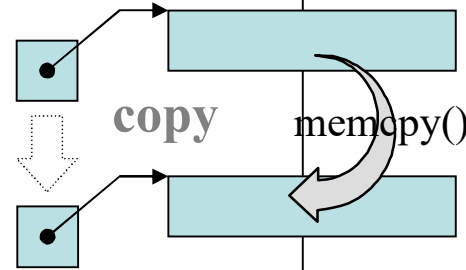
這是個天然右值

這是個人工右值
(此後不能再對 str 有任何假設)

寫一個 move aware class

```
1 class MyString {
2 private:
3     char* _data;
4     size_t _len;
5
6     void _init_data(const char *s) {
7         _data = new char[_len+1];
8         memcpy(_data, s, _len);
9         _data[_len] = '\0';
10    }
11 public:
12     //default ctor
13     MyString() : _data(nullptr), _len(0) { }
14
15     //ctor
16     MyString(const char* p) : _len(strlen(p)) {
17         _init_data(p);
18     }
```

```
57 //dtor
58 virtual ~MyString() {
59     if (_data) {
60         delete _data;
61     }
62 }
63 };
```



深拷貝
deep copy

寫一個 move aware class

```
20 // copy ctor
21 MyString(const MyString& str) : _len(str._len) {
22     _init_data(str._data);
23 }
24
25 //copy assignment
26 MyString& operator=(const MyString& str) {
27     if (this != &str) {
28         if (_data) delete _data;
29         _len = str._len;
30         _init_data(str._data); //COPY!
31     }
32     else {
33         // Self Assignment, Nothing to do.
34     }
35     return *this;
36 }
```

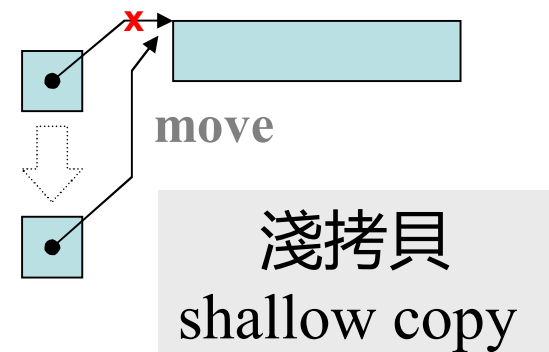
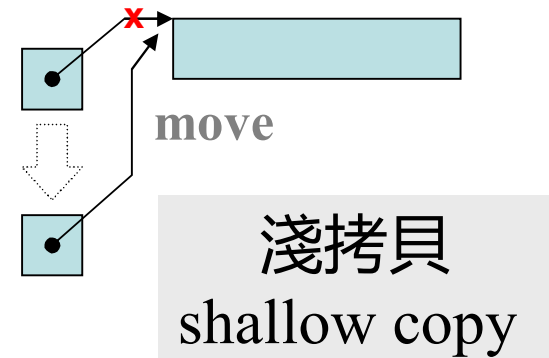
深拷貝 deep copy

自我賦值檢查

深拷貝 deep copy

寫一個 move aware class

```
38 //move ctor, with "noexcept"
39 MyString(MyString&& str) noexcept
40 : _data(str._data), _len(str._len) {
41     str._len = 0;
42     str._data = nullptr; //IMPORTANT!
43 }
44
45 //move assignment, with noexcept
46 MyString& operator=(MyString&& str) noexcept {
47     if (this != &str) {
48         if (_data) delete _data;
49         _len = str._len;
50         _data = str._data; //MOVE! (steal)
51         str._len = 0;
52         str._data = nullptr; //IMPORTANT!
53     }
54     return *this;
55 }
```



move-aware 的四種情況

MyString, move functions **without noexcept**

MyString, move functions **with noexcept**

MyString, move functions **=default**

MyString, **no** move functions

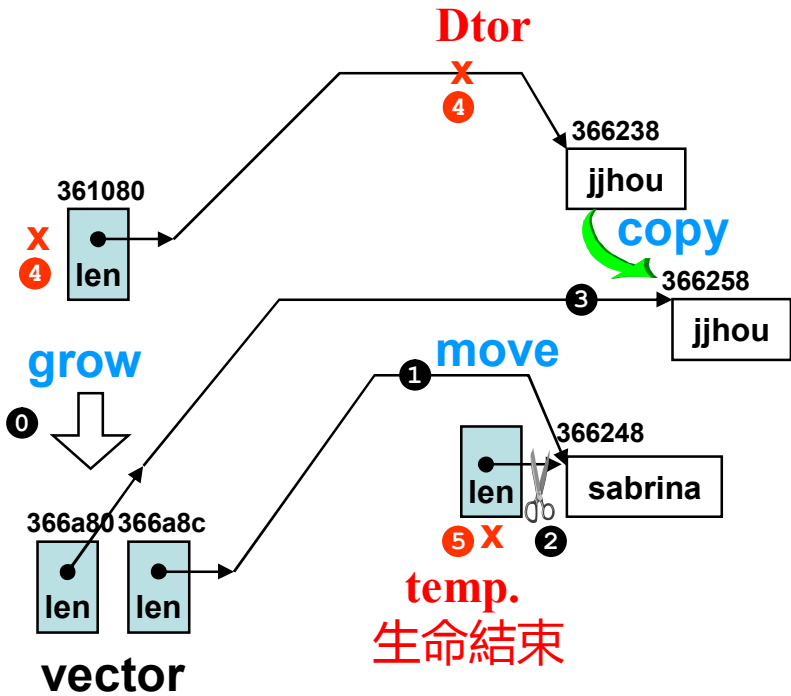
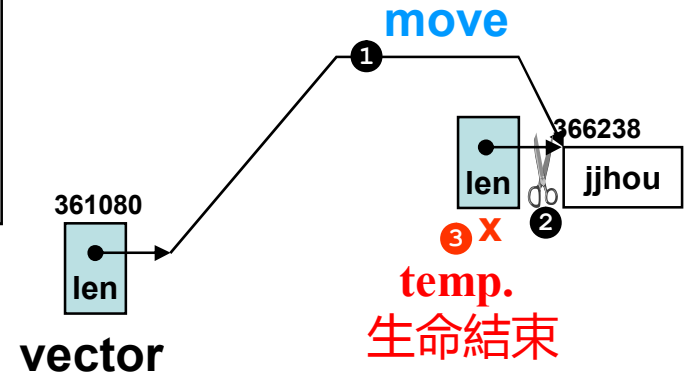
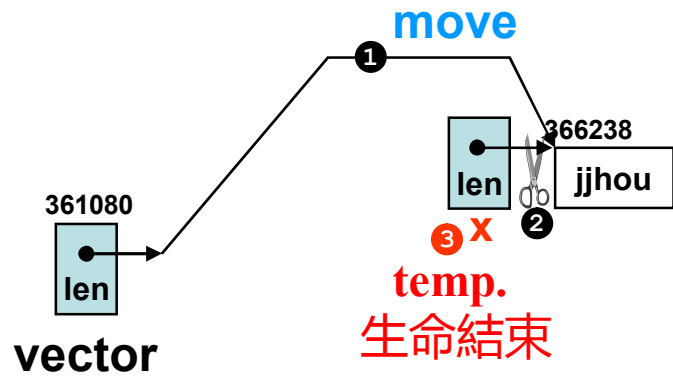
你沒寫，編譯器並不會自動給你一個 default move function。你必須明白寫出 **move functions** 否則只會擁有 default **copy functions** !!

User-defined move functions 都有打斷 pointers 動作，這很重要 (避免被連坐銷毀)。default move functions 不做這動作，不妙。

move functions without noexcept, 圖1

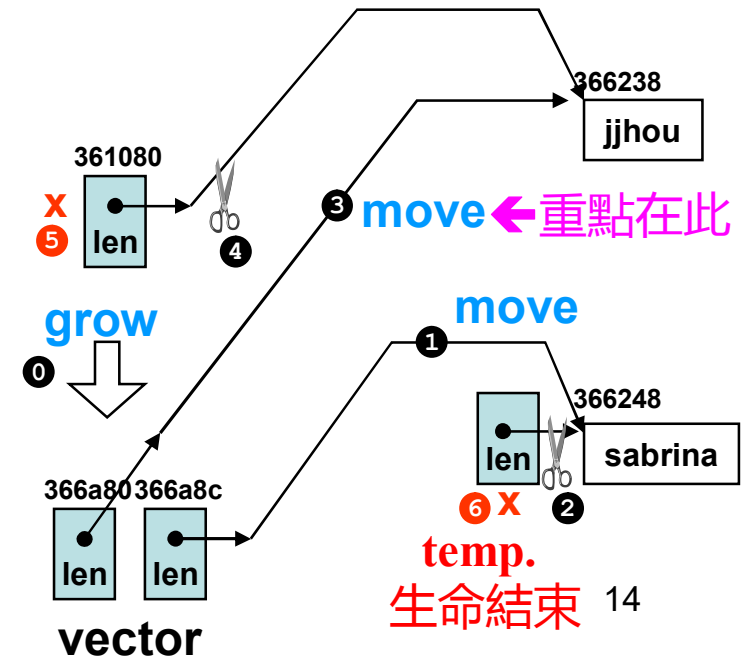
move functions with noexcept, 圖1

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



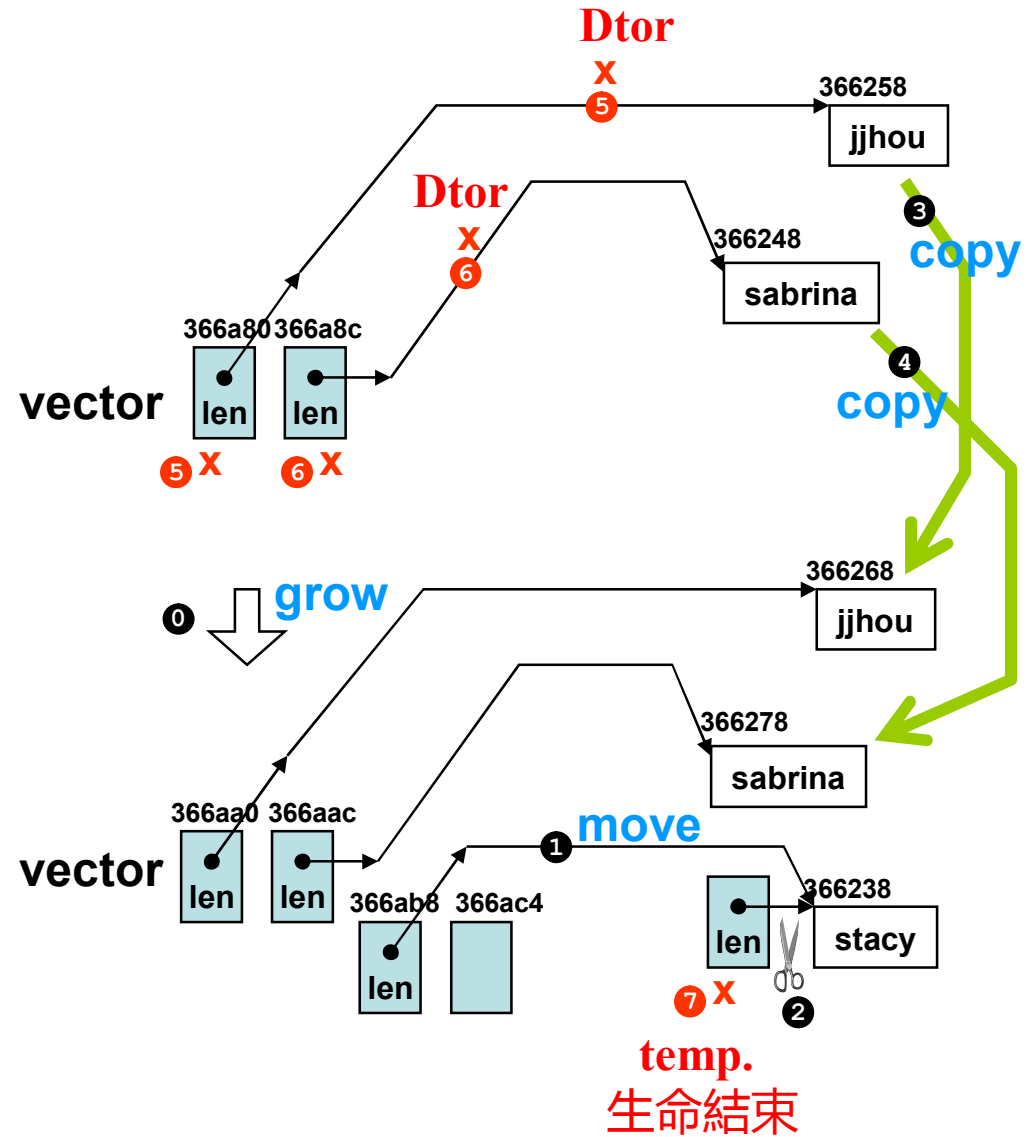
vector 擴容.
原空間清除.

擴容後的 vector
(兩倍空間)



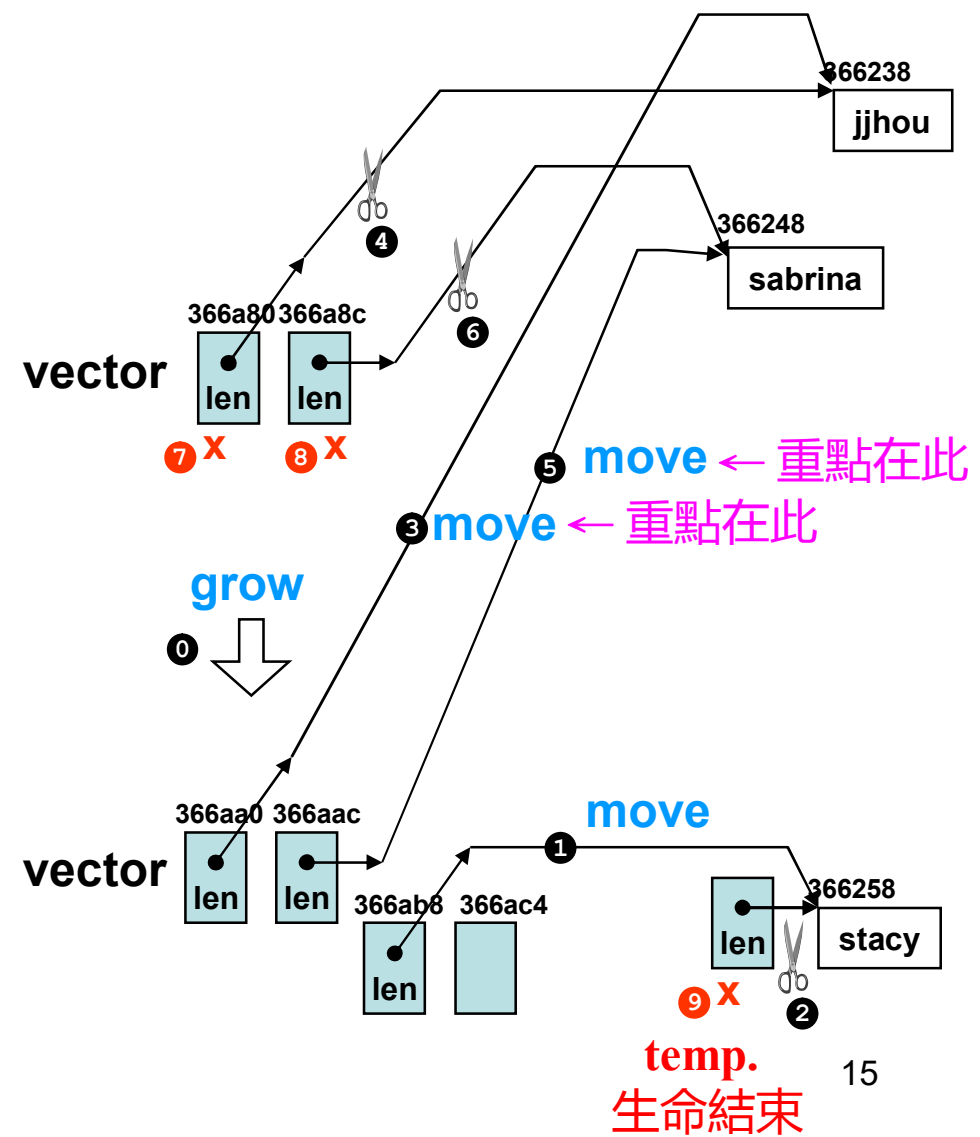
move functions without noexcept, 圖2

move functions with noexcept, 圖2



vector 擴容.
原空間清除.

擴容後的 vector
(兩倍空間)



move ← 重點在此

move ← 重點在此

temp.
生命結束

一個方便的測試函數

  `test_moveable(vector<MyString>, 30000000L);`

```
2  template<typename M>
3  void test_moveable(M c, long& size)
4  {
5      typedef typename
6          iterator_traits<typename M::iterator>::value_type ValType;
7
8      char buf[10];          typedef typename M::value_type ValType;
9      clock_t timeStart = clock();
10     for (long i=0; i< size; ++i) {
11         snprintf(buf, 10, " %d" , rand()); // 隨機數 (轉為 C string)
12         auto ite = c.end();                // 定位於尾端
13         c.insert(ite, ValType(buf));      // 所有容器都支持 insert()
14     }
15     cout << "milli-seconds : " << (clock()-timeStart) << endl;
16 }
```

為什麼 std 容器有時候“不敢”調用 move function without noexcept?

Function with noexcept 可否調用 function without noexcept ?

換言之關鍵字 **noexcept** 是否和關鍵字 **const** 一樣擁有某種「編譯期強制性」?

首先測試，**noexcept** 是否為函數簽名的一部分？

亦即 function with **noexcept** 和 function without **noexcept** 可否並存(重載)？

Ans: 不可!

Functions differing only in their exception specification cannot be overloaded (just like the return type, exception specification is **part of function type**, but **not part of the function signature**) (since C++17).

noexcept

其次測試 function with noexcept 卻(違反承諾)拋出異常會怎樣？

順利編譯。編譯器不檢查。但運行時 crash!

```
terminate called without an active exception
```

```
This application has requested the Runtime to terminate it in an unusual way.  
Please contact the application's support team for more information.
```

最後測試 function with noexcept 呼叫 function without noexcept 會怎樣？

function without noexcept 調用 function with noexcept，通過是應該的。
function with noexcept 調用 function without noexcept (而實際未拋異常)，
編譯通過，執行也 ok.

noexcept operator 執行 **compile-time check** , 如果 *expression* 被聲明 (declared) 為不拋任何異常 (not throw any exceptions) , 就返回 true.

noexcept → noexcept (true)

```
cout << noexcept(func1()) << endl; //1
cout << noexcept(func2()) << endl; //0
cout << noexcept(func3()) << endl; //1
cout << noexcept(func4()) << endl; //0
cout << noexcept(func5()) << endl; //1
cout << noexcept(func6()) << endl; //0
cout << noexcept(func7()) << endl; //1
cout << noexcept(func8()) << endl; //1
```

上述 8 個函數凡聲明 **noexcept** 者便得 1 (true) , 否則便得 0 。

```
void func9() noexcept(false) { }
//==> func9() noexcept(false) { }
void func10() noexcept(noexcept(func1())) { }
//==> func10() noexcept(true) { }
void func11() noexcept(noexcept(func2())) { }
//==> func11() noexcept(false) { }

cout << noexcept(func9()) << endl; //0
cout << noexcept(func10()) << endl; //1
cout << noexcept(func11()) << endl; //0
```

noexcept(*expression*) 返回值取決於 *expression* 所表現的函數是否 "其所聲明的 noexcept() 為 true"

move_if_noexcept

```
4655 //以下例子來自 http://en.cppreference.com/w/cpp/utility/move\_if\_noexcept.
4656 struct Bad
4657 {
4658     Bad() {}
4659     Bad(Bad&&) // may throw
4660     { cout << "move constructor without noexcept called\n"; }
4661     Bad(const Bad&) // may throw as well
4662     { cout << "copy constructor without noexcept called\n"; }
4663     Bad& operator=(const Bad&) //我添加 (1)
4664     { cout << "copy assignment operator without noexcept called\n"; }
4665 };
4666 struct Good
4667 {
4668     Good() {}
4669     Good(Good&&) noexcept // will NOT throw
4670     { cout << "move constructor with noexcept called\n"; }
4671     Good(const Good&) noexcept // will NOT throw
4672     { cout << "copy constructor with noexcept called\n"; }
4673     Good& operator=(const Good&) noexcept //我添加 (2)
4674     { cout << "copy assignment operator with noexcept called\n"; }
4675     Good& operator=(const Good&&) noexcept //我添加 (3)
4676     { cout << "move assignment operator with noexcept called\n"; }
4677 };
```


move_if_noexcept

```
4684 Good g;  
4685 Bad b;  
4686  
4687 //注意，以下的測試，並沒有把 L-value 以 move() 強迫為 R-value。  
4688 Good g2 = std::move_if_noexcept(g); //move constructor with noexcept called  
4689 Bad b2 = std::move_if_noexcept(b); //copy constructor without noexcept called  
4690 g2 = std::move_if_noexcept(g); //move assignment operator with noexcept called  
4691 b2 = std::move_if_noexcept(b); //copy assignment operator without noexcept called
```

```
00118: template<typename _Tp>  
00119: constexpr typename  
00120: conditional<__move_if_noexcept_cond<_Tp>::value, const _Tp&, _Tp&&>::type  
00121: move_if_noexcept(&_Tp& __x) noexcept  
00122: { return std::move(__x); }
```

std::vector 以什麼方法判斷 callee 為 noexcept ?

```
template<typename _Tp, typename _Alloc>
template<typename... _Args>
typename vector<_Tp, _Alloc>::iterator
vector<_Tp, _Alloc>::
emplace(const_iterator __position, _Args&&... __args)
{
    const size_type __n = __position - begin();
    if (this->_M_impl._M_finish != this->_M_impl._M_end_of_storage
        && __position == end()) 尚有空間且欲置於最末
        ...
    else 空間不足需擴容
        _M_insert_aux(begin() + (__position - cbegin()), .....
                      std::forward<_Args>(__args)...);
    return iterator(this->_M_impl._M_start + __n);
}
```

→ 下一頁

```
template<typename _Tp, typename _Alloc>
```

```
template<typename... _Args>
```

```
void
```

```
vector<_Tp, _Alloc>::
```

```
_M_insert_aux(iterator __position, _Args&&... __args)
```

```
{
```

```
if (this->_M_impl._M_finish !=
```

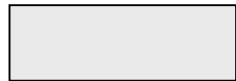
```
    this->_M_impl._M_end_of_storage
```

```
{ ... } 尚有空間
```

```
else
```

空間不足需擴容

```
{ ...
```



```
...
```

```
}
```

```
}
```

```
__try
```

```
{ ...
```

擴容後首先把原先所有元素
挪過來 (copy or move)

```
__new_finish
```

```
= std::__uninitialized_move_if_noexcept_a
```

```
(this->_M_impl._M_start, __position.base(),
```

```
    __new_start, _M_get_Tp_allocator());
```

```
++__new_finish;
```

```
__new_finish
```

然後安插新元素(s) (copy or move)

```
= std::__uninitialized_move_if_noexcept_a
```

```
(__position.base(), this->_M_impl._M_finish,
```

```
    __new_finish, _M_get_Tp_allocator());
```

```
}
```

```
__catch(...)
```

```
{ ... }
```



The End



講師簡介

侯捷，計算機技術之著、譯、評、講。同濟大學軟件學院客座教授。

專長 **Windows SDK Programming, MFC Programming, Design Patterns, Memory Management, STL Architecture, Concurrency.**

著有《**虛擬記憶體**》、《**深入淺出 MFC**》、《**STL 源碼剖析**》、
《**多型與虛擬**》、《**Windows 記憶體管理**》、《**無責任書評**》...

譯有《**C++ Primer**》、《**Effective C++**》、《**More Effective C++**》、
《**Thinking in Java**》、《**Refactoring**》、《**Refactoring to Patterns**》、
《**Multithreaded Applications in Win32**》、《**Inside Visual C++**》...