Modern C++

Move 語意剖析與實例觀察

山高月小 水落石出

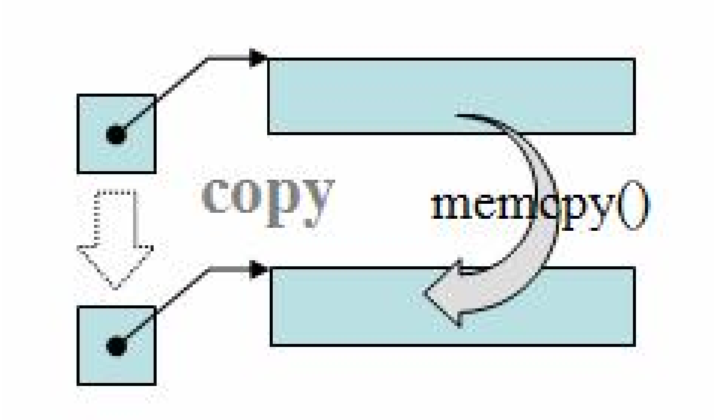
侯捷

2017/11/17   C++ 技術大會.博覽.北京

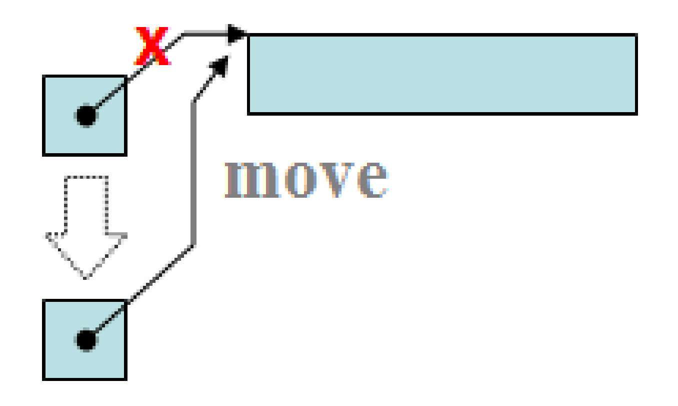
主題

- 為什麼 move？何時可以 move？
- 如何寫一個 move-aware class？
- 如何表白「我是 moveable, 如果可以請 move 我」？
- noexcept 的重要性
- 容器擴容 (grows) 時為什麼不敢調用元素的 move function which without noexcept？

深拷貝 vs. 淺拷貝 (唯有 class with resource 才需考慮)



深拷貝
deep copy



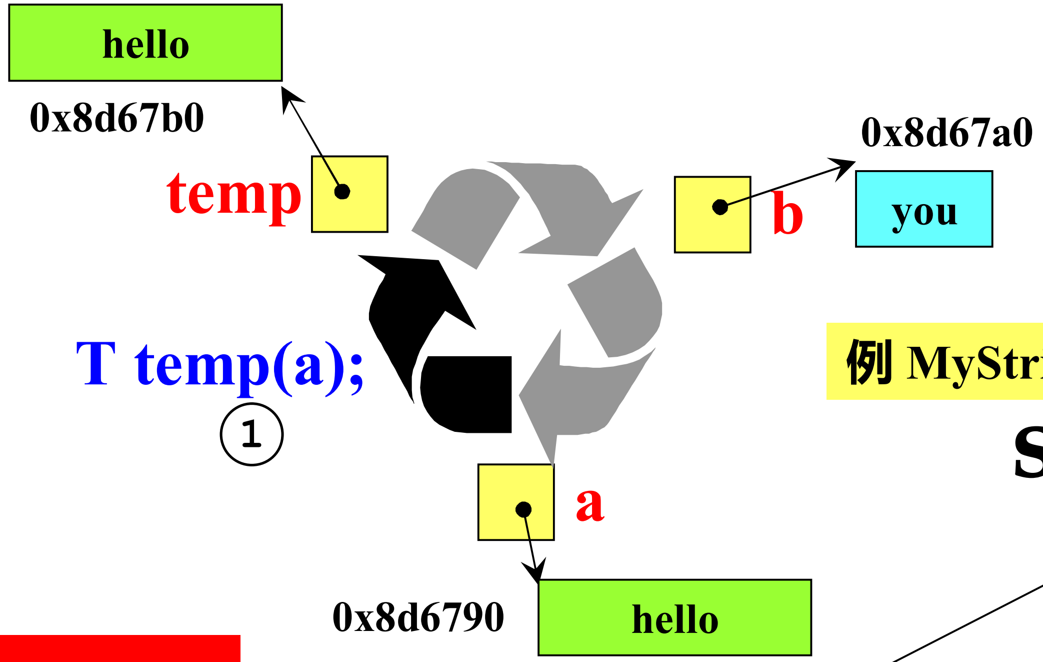
淺拷貝
shallow copy

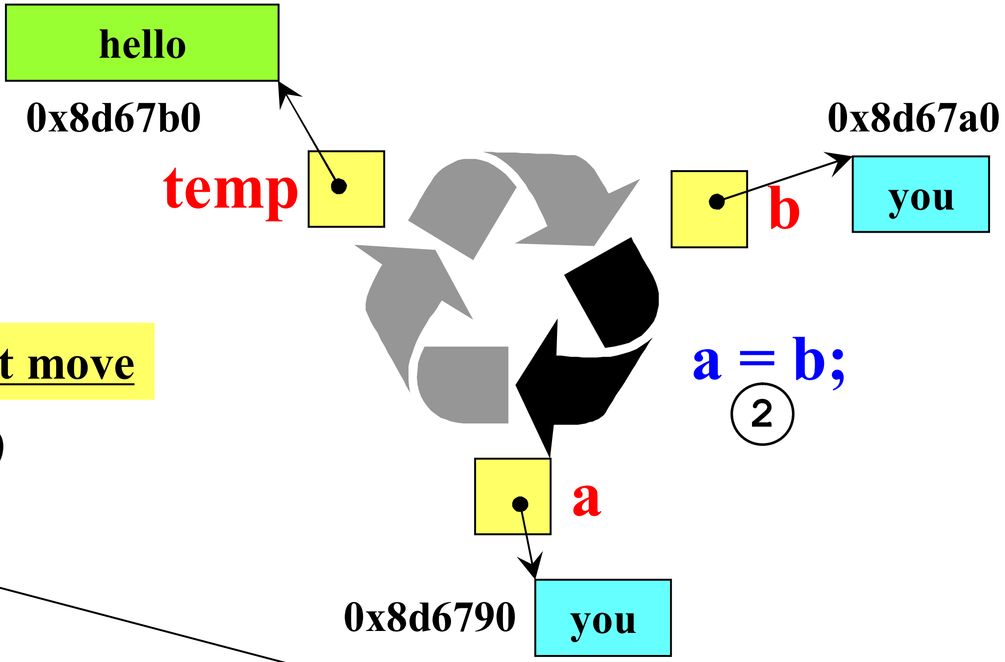
淺拷貝造成 alias，非常危險。
必須另有處理：
1, 原件放棄擁有權 or
2, 使用 reference counting 技術

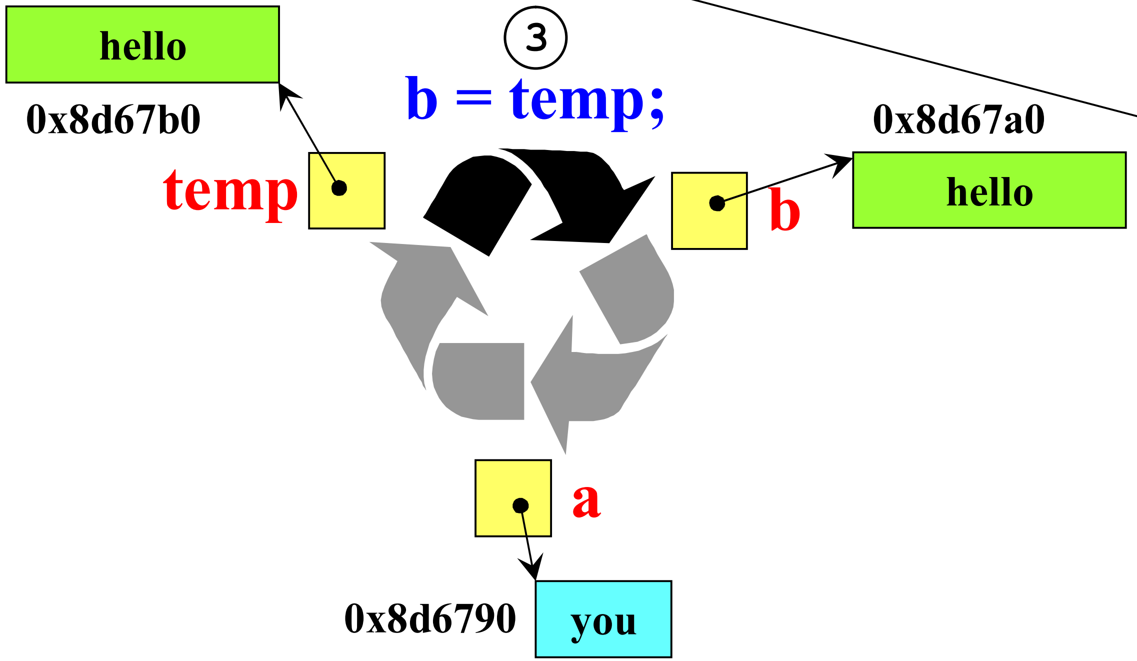
以 swap 為例 (無需深拷貝; 淺拷貝足矣)

例 MyString without move

swap



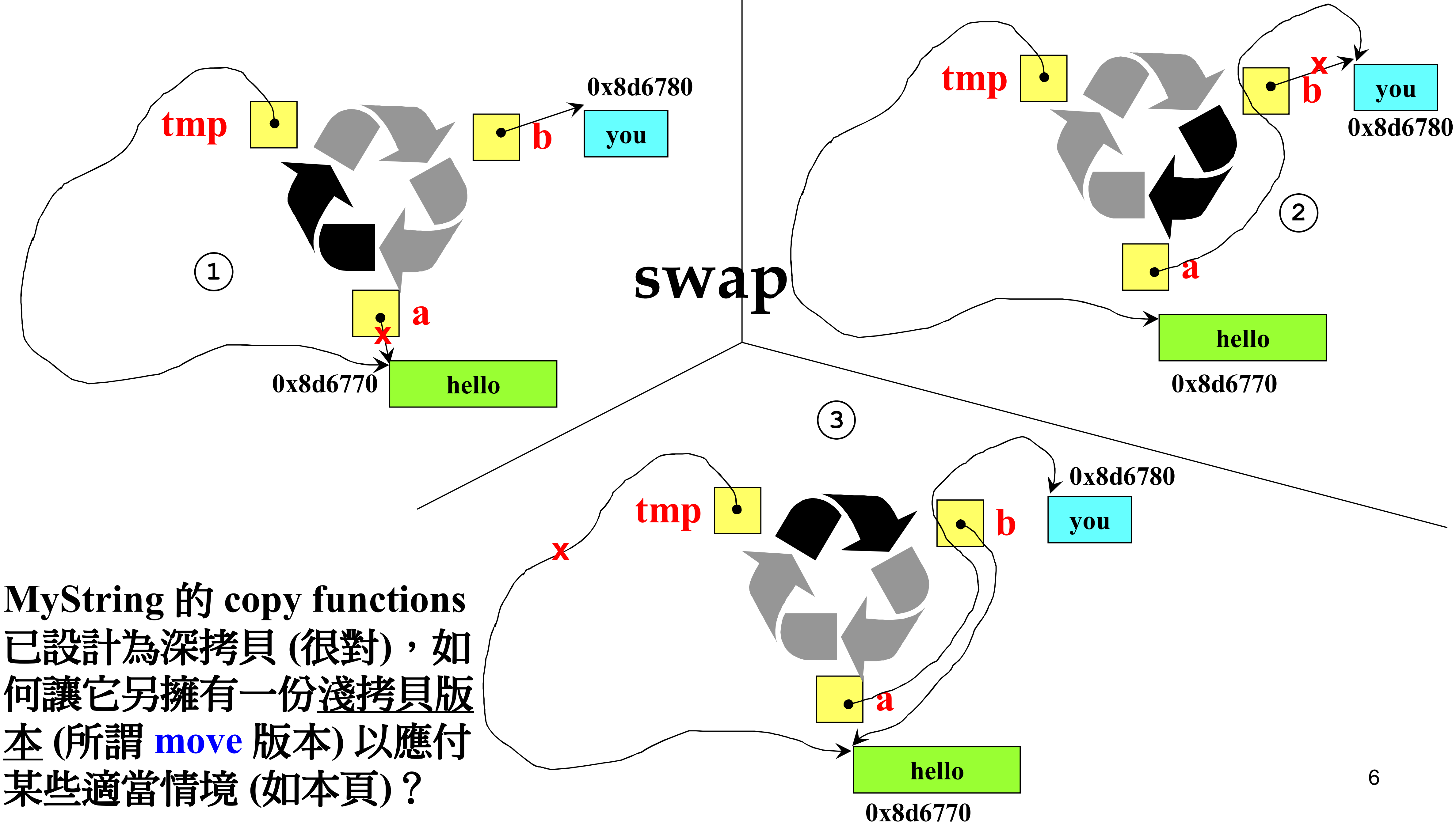




Until 2011

```
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<u>專門處理 move 的函數</u>。

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

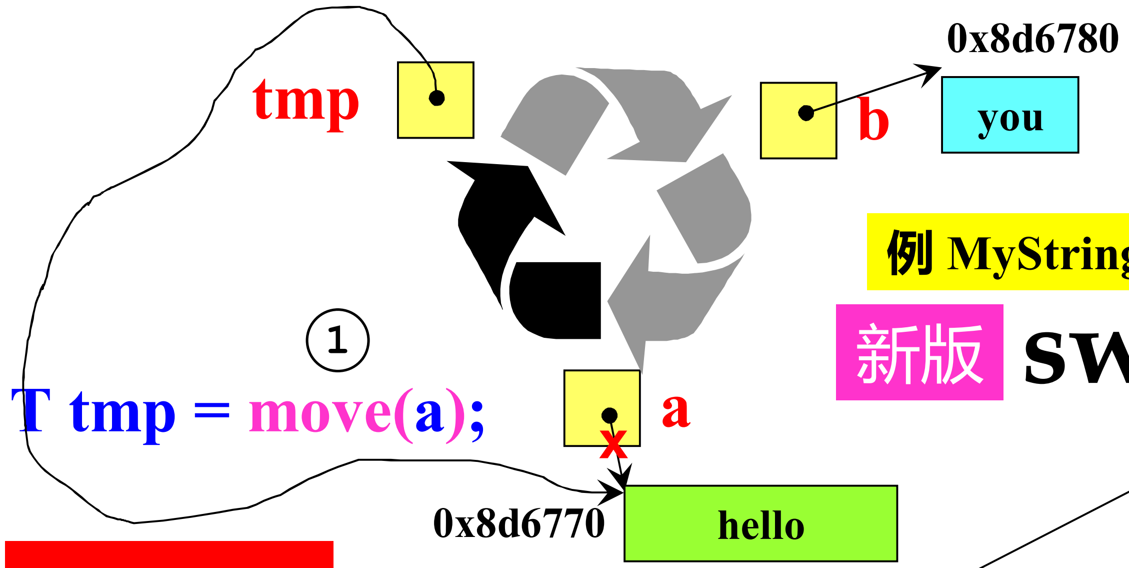
用後不打算再用

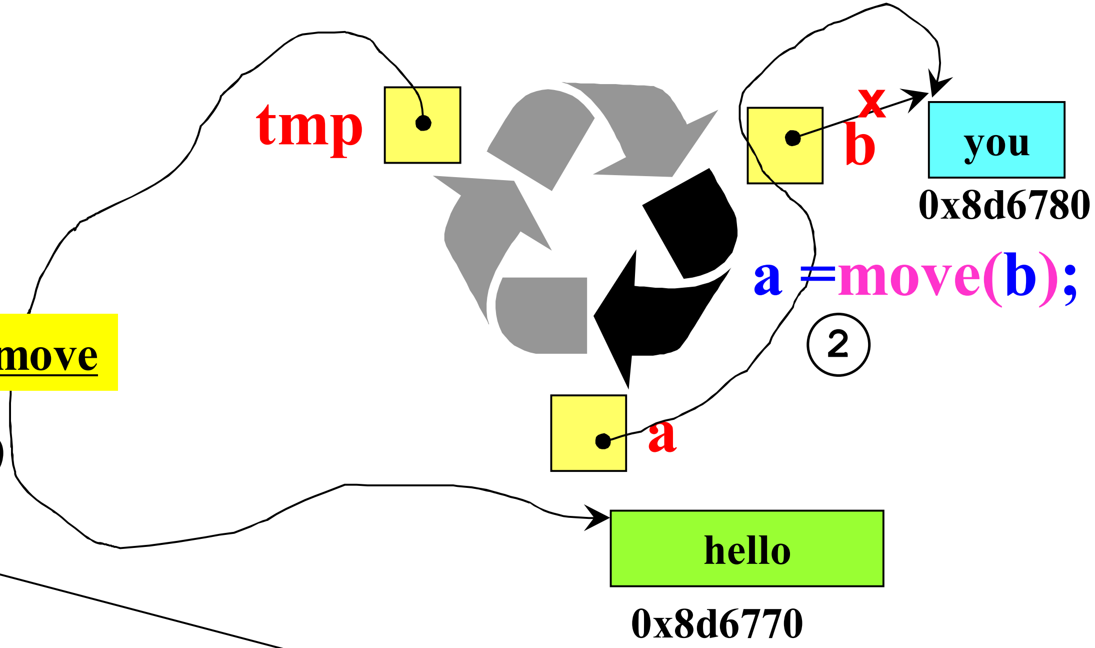
2, C++11 推出 R-value reference.

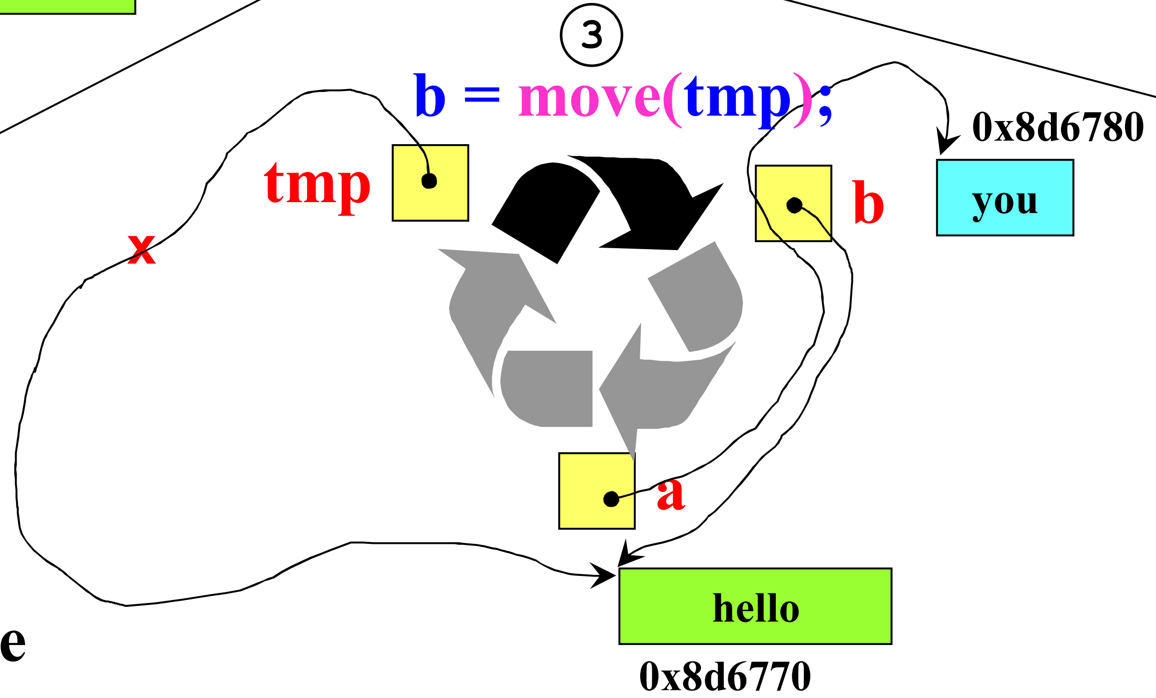
`T(T&& x);` .... move ctor
`T& operator=(T&& x);` .... move assignment

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

❶

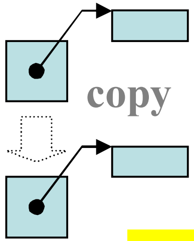
```
//copy
insert(…,&x)
```

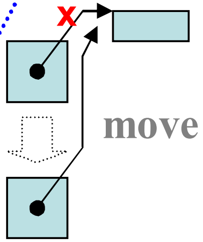
❷

```
//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() …
```





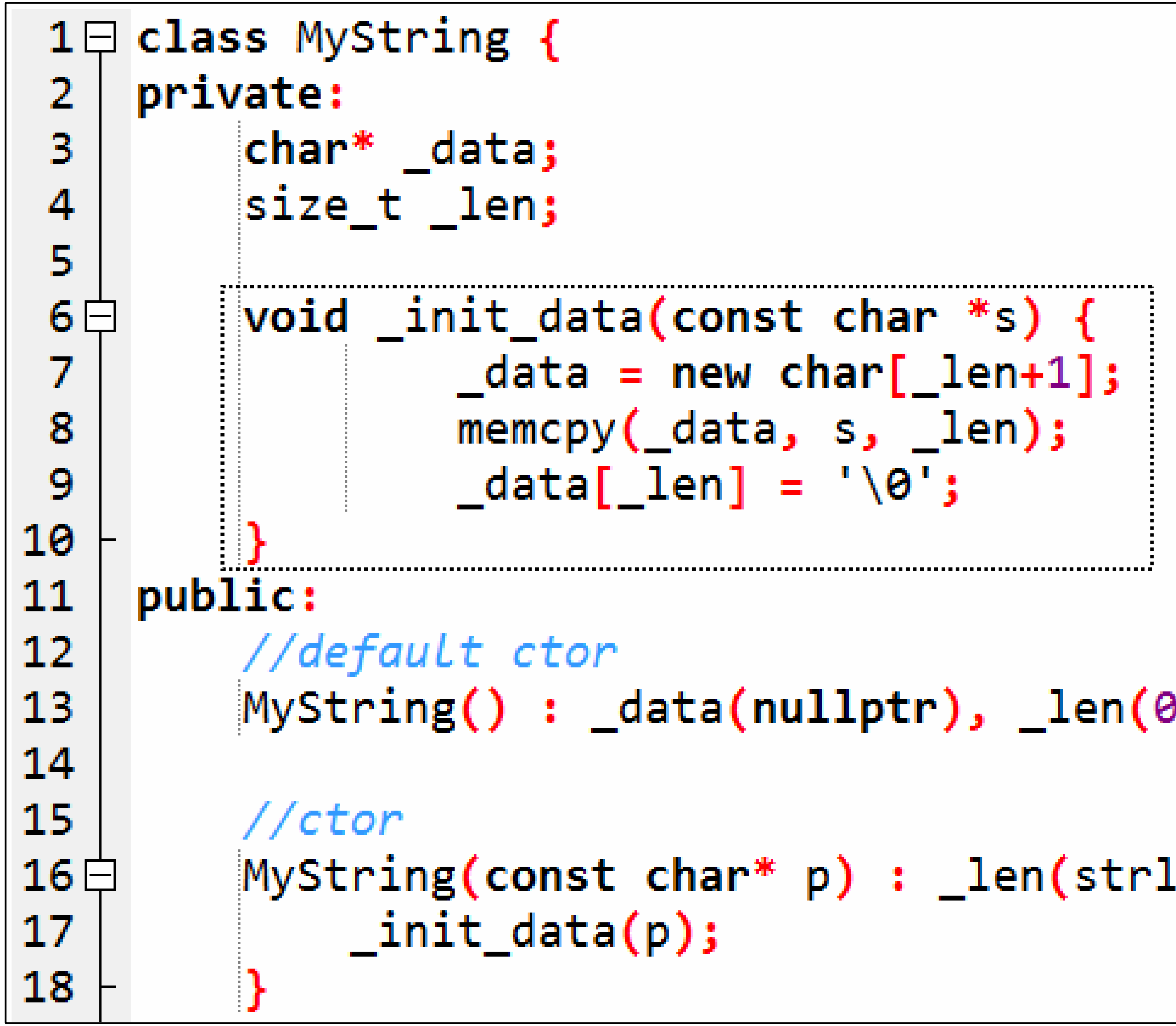
Class 的使用者

```
c.insert(iter, MyString(“hello”));

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

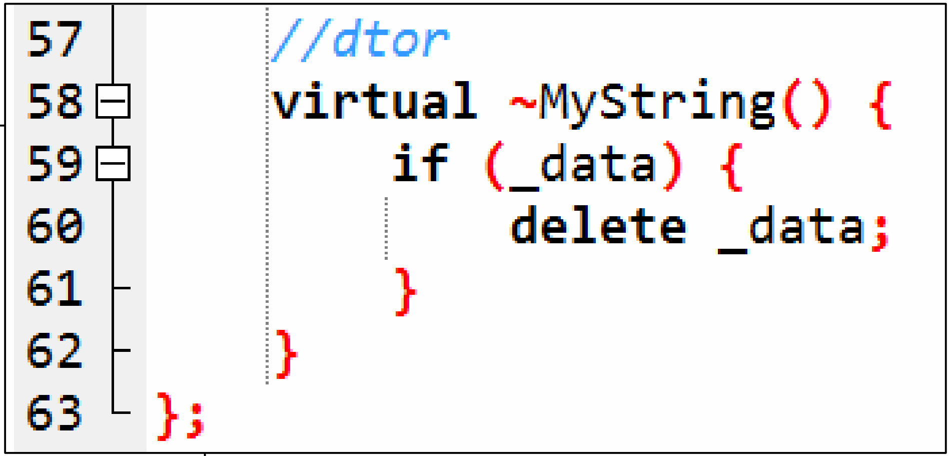
這是個天然右值

這是個人工右值(此後不能再對 str 有任何假設)

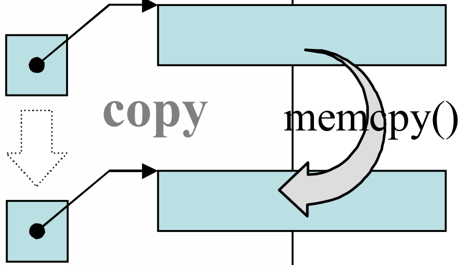
寫一個 move aware class



```cpp
class MyString {
private:
    char* _data;
    size_t _len;

    void _init_data(const char *s) {
        _data = new char[_len+1];
        memcpy(_data, s, _len);
        _data[_len] = '\0';
    }
public:
    //default ctor
    MyString() : _data(nullptr), _len(0) {  }

    //ctor
    MyString(const char* p) : _len(strlen(p)) {
        _init_data(p);
    }
```

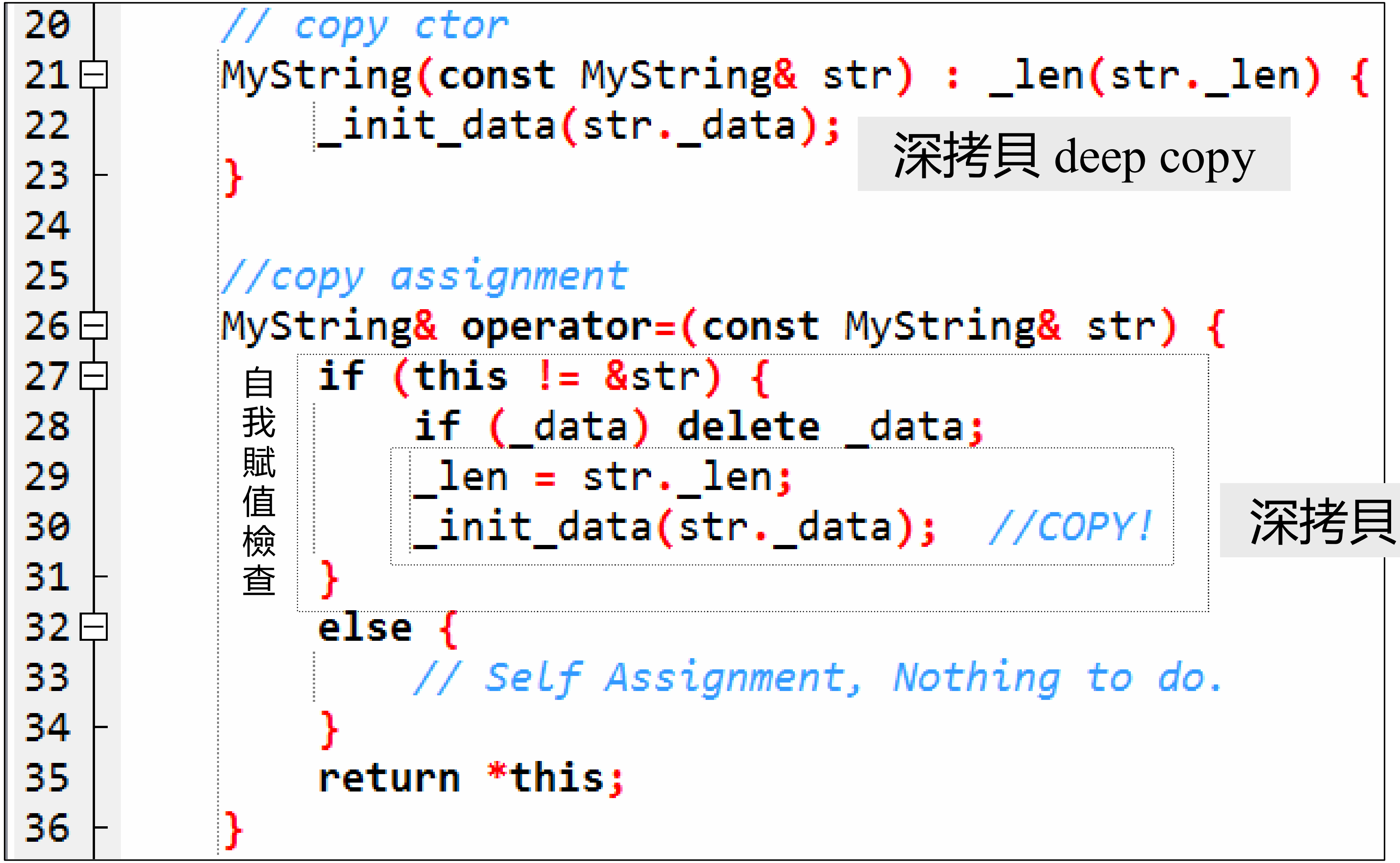


```cpp
    //dtor
    virtual ~MyString() {
        if (_data) {
            delete _data;
        }
    }
};
```



深拷貝
deep copy

寫一個 move aware class



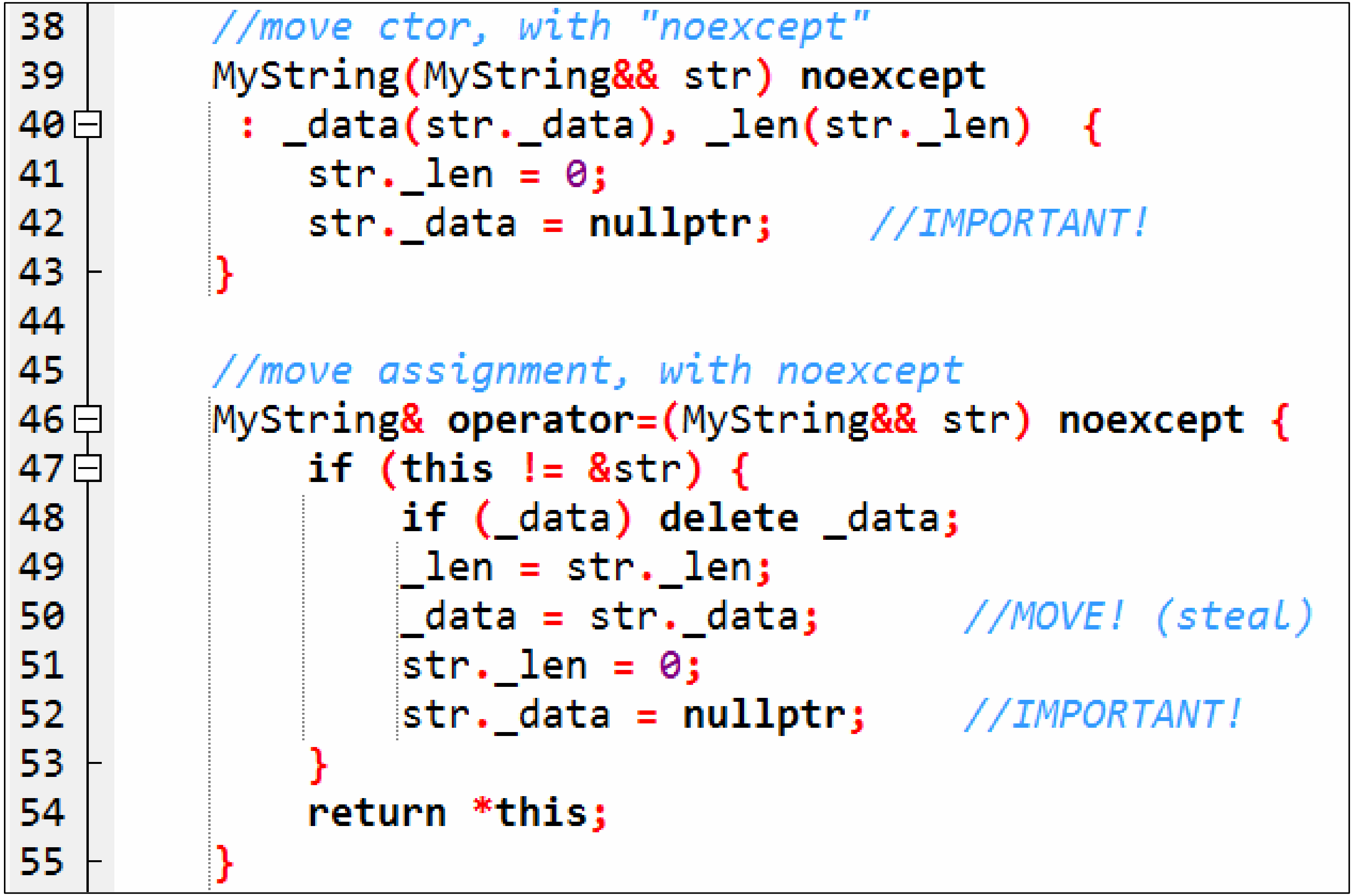
```
    // copy ctor
    MyString(const MyString& str) : _len(str._len) {
        _init_data(str._data);
    }

    //copy assignment
    MyString& operator=(const MyString& str) {
        if (this != &str) {
            if (_data) delete _data;
            _len = str._len;
            _init_data(str._data);   //COPY!
        }
        else {
            // Self Assignment, Nothing to do.
        }
        return *this;
    }
```

自我賦值檢查

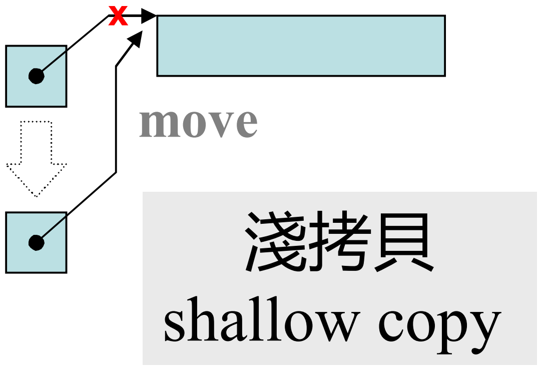
深拷貝 deep copy

深拷貝 deep copy

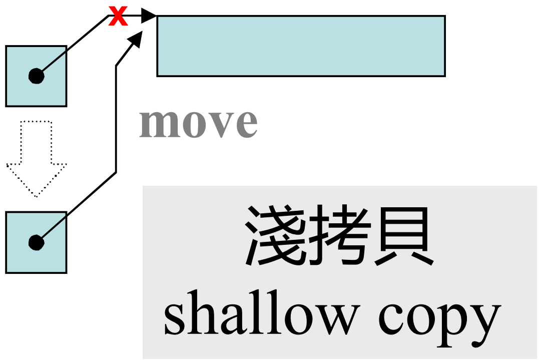
寫一個 move aware class



```
    //move ctor, with "noexcept"
    MyString(MyString&& str) noexcept
     : _data(str._data), _len(str._len)  {
        str._len = 0;
        str._data = nullptr;    //IMPORTANT!
    }

    //move assignment, with noexcept
    MyString& operator=(MyString&& str) noexcept {
        if (this != &str) {
            if (_data) delete _data;
            _len = str._len;
            _data = str._data;      //MOVE! (steal)
            str._len = 0;
            str._data = nullptr;    //IMPORTANT!
        }
        return *this;
    }
```



淺拷貝
shallow copy



淺拷貝
shallow copy

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

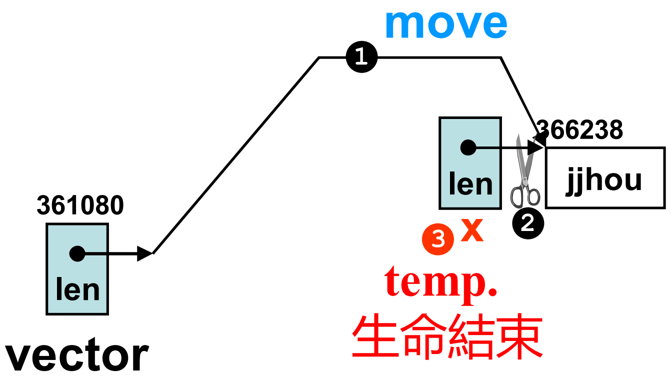
User-defined move functions 都有打斷 pointers 動作，這很重要 (避免被連坐銷毀)。default move functions 不做這動作，不妙。

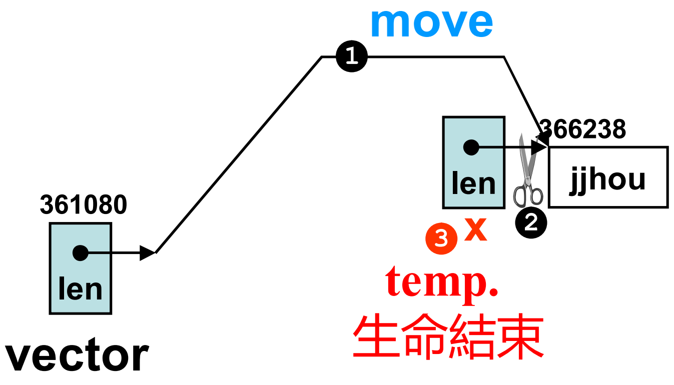
你沒寫，編譯器並不會自動給你一個 default move function。你必須明白寫出 **move** functions 否則只會擁有 default **copy** functions !!

move functions without noexcept, 圖1

move functions with noexcept, 圖1

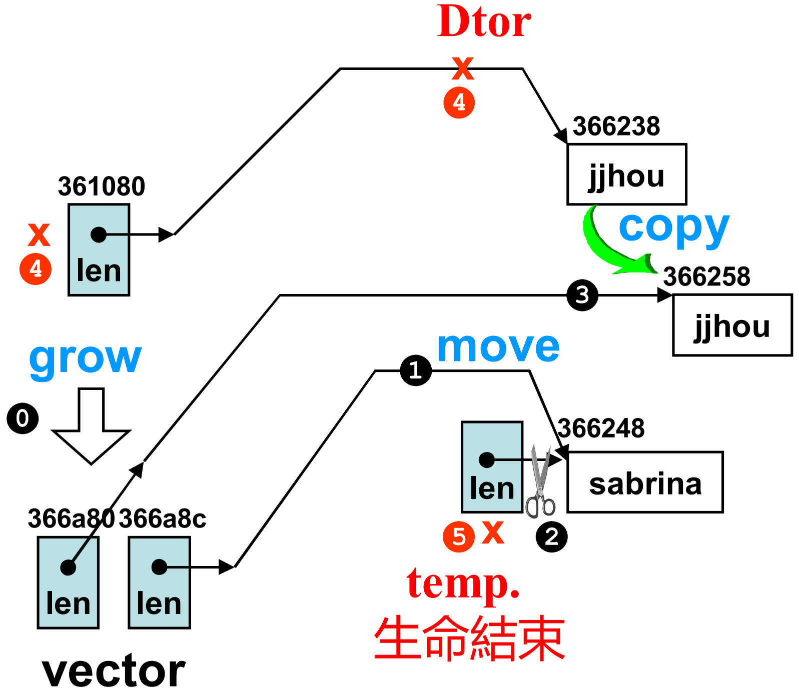
```
vector<MyString> vec;
vec.push_back(MyString("jjhou"));
vec.push_back(MyString("sabrina"));
vec.push_back(MyString("stacy"));
```

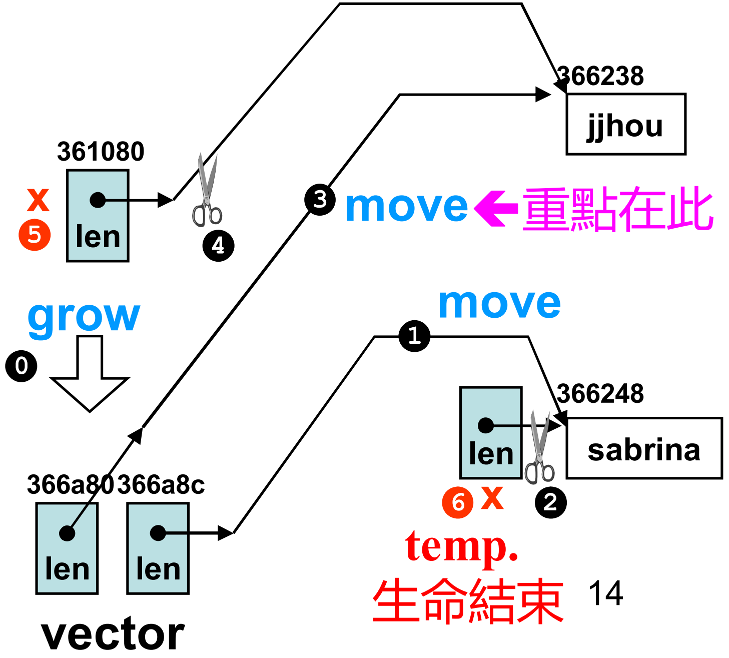




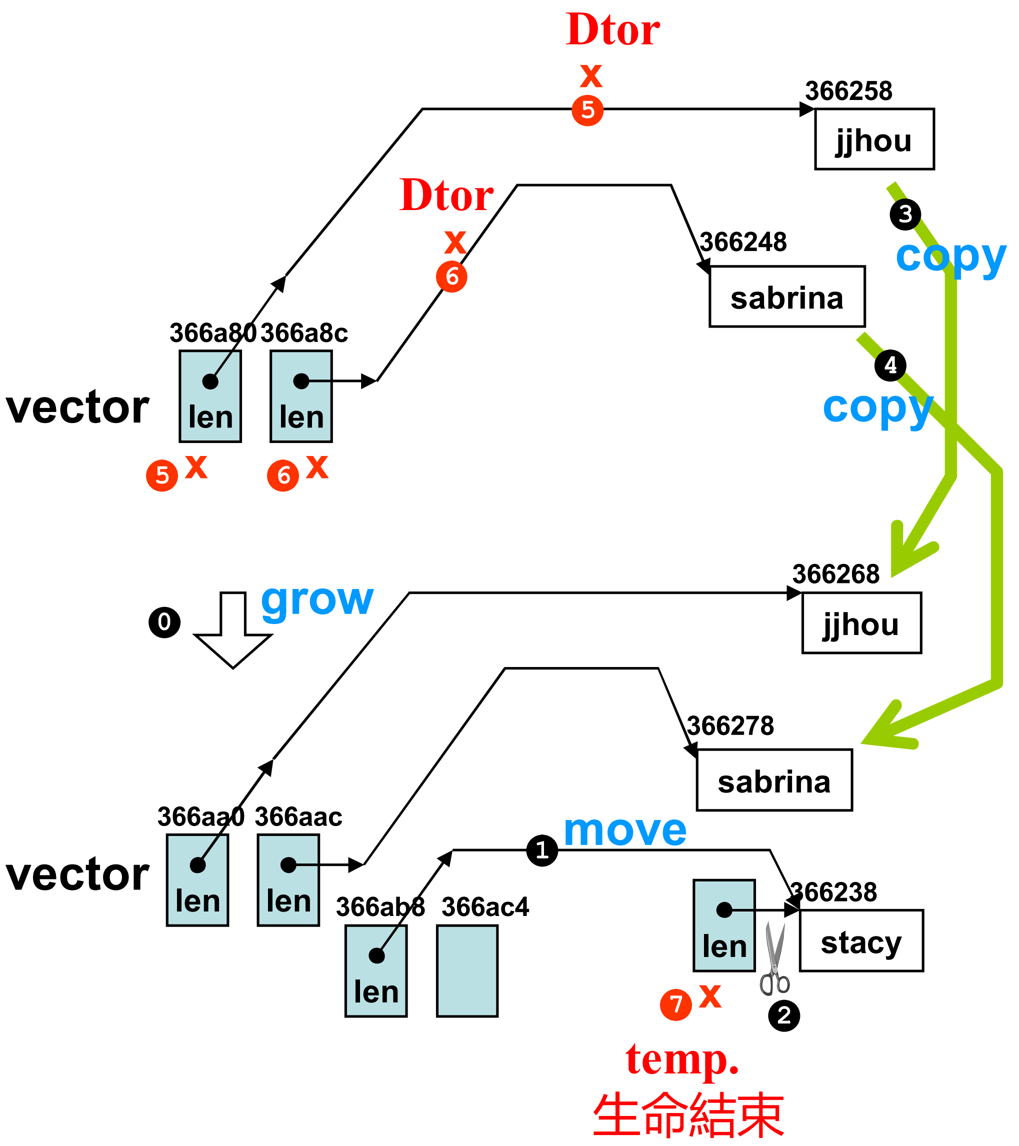
vector 擴容.
原空間清除.

擴容後的 vector
(兩倍空間)





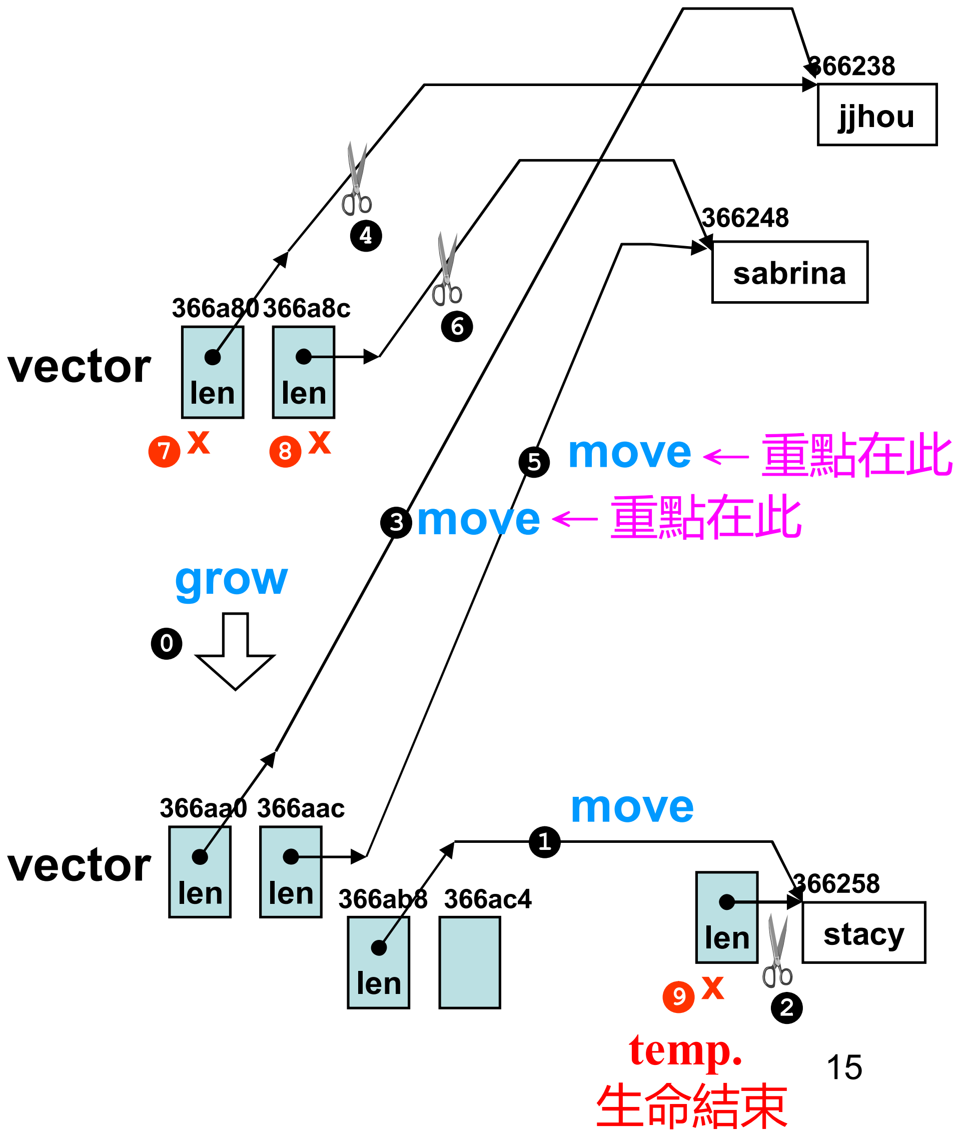
move functions without noexcept, 圖2



vector 擴容.
原空間清除.

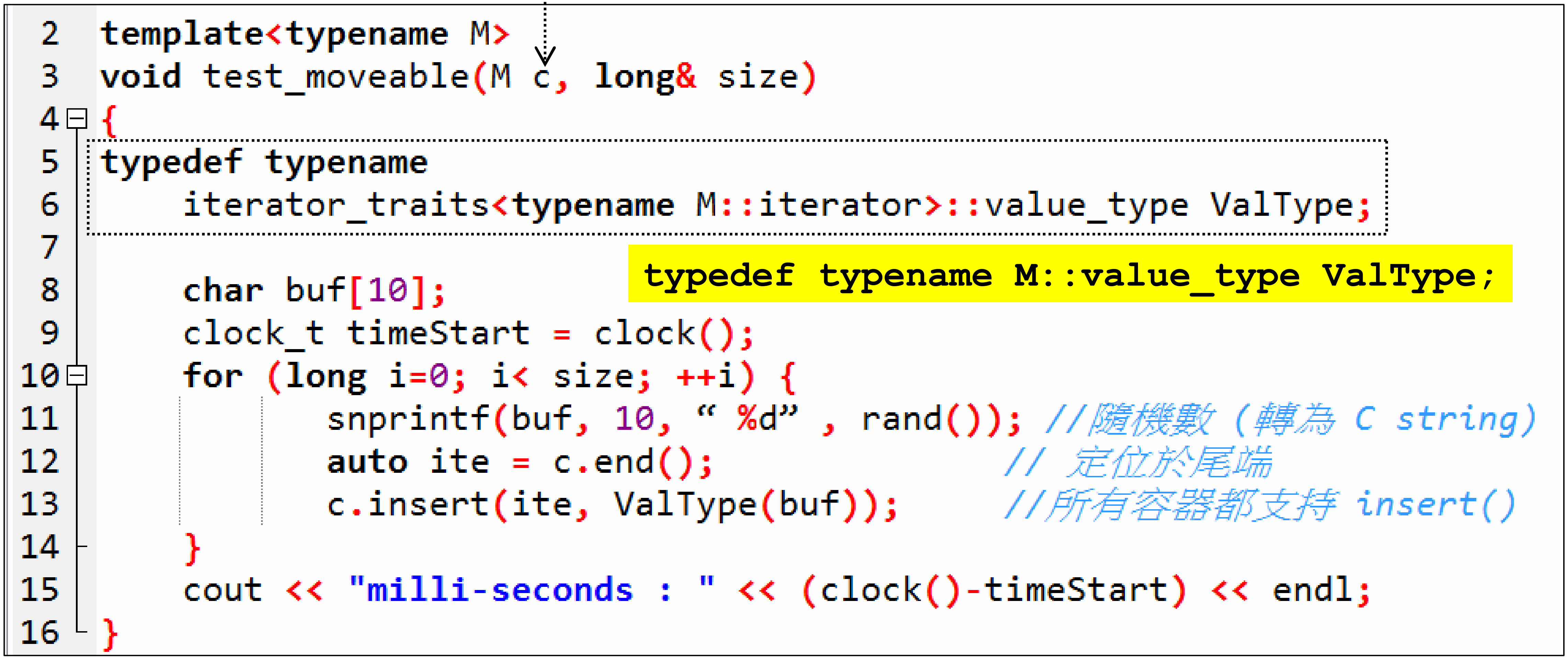
擴容後的 vector
(兩倍空間)

move functions with noexcept, 圖2



一個方便的測試函數

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



```
template<typename M>
void test_moveable(M c, long& size)
{
typedef typename
    iterator_traits<typename M::iterator>::value_type ValType;

    char buf[10];
    clock_t timeStart = clock();
    for (long i=0; i< size; ++i) {
        snprintf(buf, 10, “ %d” , rand()); //隨機數 (轉為 C string)
        auto ite = c.end();                // 定位於尾端
        c.insert(ite, ValType(buf));       //所有容器都支持 insert()
    }
    cout << "milli-seconds : " << (clock()-timeStart) << endl;
}
```

```
typedef typename M::value_type ValType;
```

noexcept 的地位

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」？

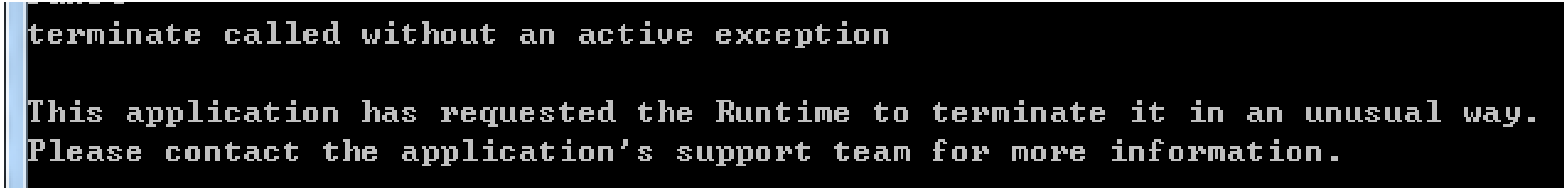
首先測試，noexcept 是否為函數簽名的一部分？
亦即 function with noexcept 和 function without noexcept 可否並存(重載)？
Ans: 不可!

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

noexcept (*expression*)

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)

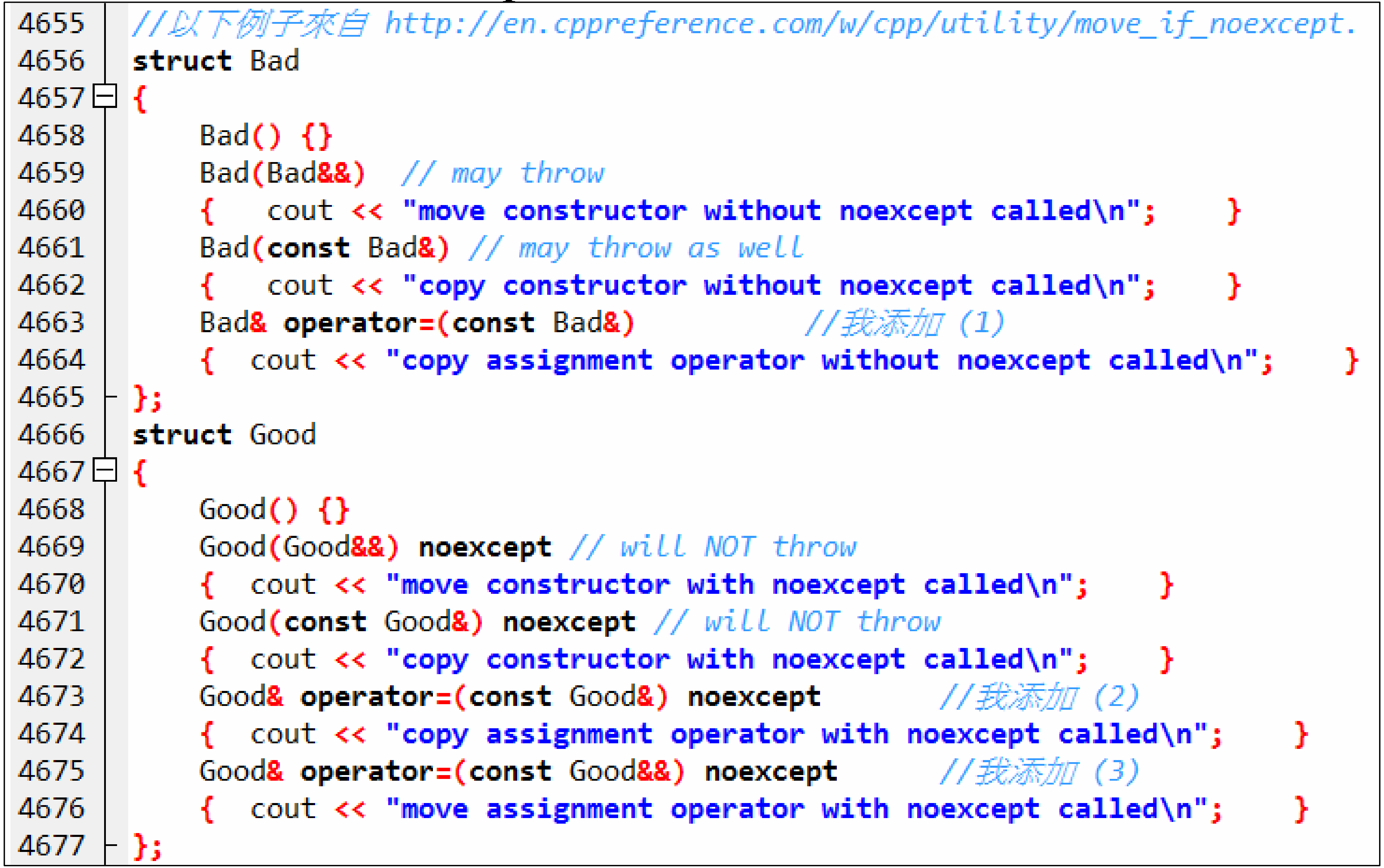
```
   cout << noexcept(func1()) << endl;  //1
   cout << noexcept(func2()) << endl;  //0
   cout << noexcept(func3()) << endl;  //1
   cout << noexcept(func4()) << endl;  //0
   cout << noexcept(func5()) << endl;  //1
   cout << noexcept(func6()) << endl;  //0
   cout << noexcept(func7()) << endl; 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

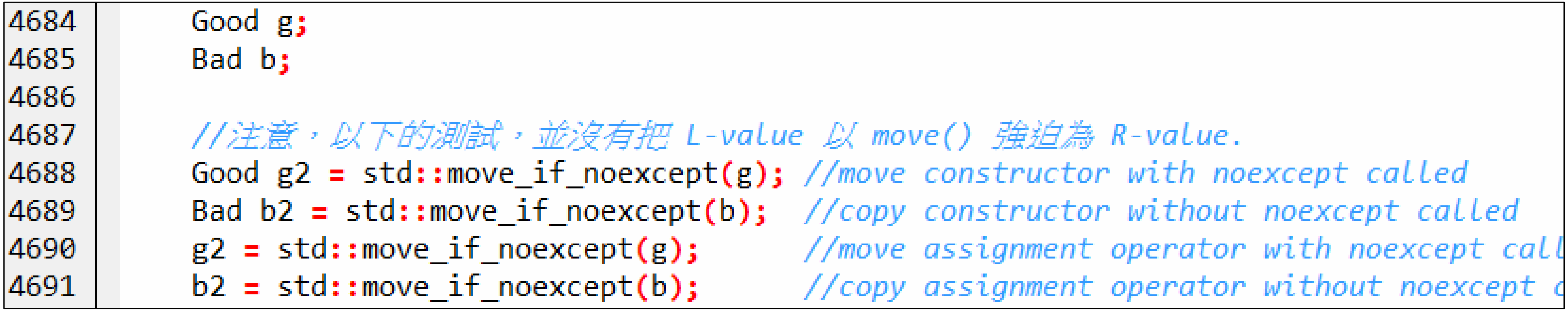
noexcept(*expression*) 返回值取決於 *expression* 所表現的函數是否 "其所聲明的 noexcept() 為 true"

move_if_noexcept



```
//以下例子來自 http://en.cppreference.com/w/cpp/utility/move_if_noexcept.
struct Bad
{
    Bad() {}
    Bad(Bad&&)  // may throw
    {   cout << "move constructor without noexcept called\n";    }
    Bad(const Bad&) // may throw as well
    {   cout << "copy constructor without noexcept called\n";    }
    Bad& operator=(const Bad&)          //我添加 (1)
    {  cout << "copy assignment operator without noexcept called\n";    }
};
struct Good
{
    Good() {}
    Good(Good&&) noexcept // will NOT throw
    {  cout << "move constructor with noexcept called\n";    }
    Good(const Good&) noexcept // will NOT throw
    {  cout << "copy constructor with noexcept called\n";    }
    Good& operator=(const Good&) noexcept       //我添加 (2)
    {  cout << "copy assignment operator with noexcept called\n";    }
    Good& operator=(const Good&&) noexcept      //我添加 (3)
    {  cout << "move assignment operator with noexcept called\n";    }
};
```

move_if_noexcept



```
    Good g;
    Bad b;

    //注意，以下的測試，並沒有把 L-value 以 move() 強迫為 R-value.
    Good g2 = std::move_if_noexcept(g); //move constructor with noexcept called
    Bad b2 = std::move_if_noexcept(b);  //copy constructor without noexcept called
    g2 = std::move_if_noexcept(g);      //move assignment operator with noexcept call
    b2 = std::move_if_noexcept(b);      //copy assignment operator without noexcept c
```

```
template<typename _Tp>
constexpr typename
conditional<__move_if_noexcept_cond<_Tp>::value, const _Tp&, _Tp&&>::type
move_if_noexcept(_Tp& __x) noexcept
{ 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_stora
    {   …      }   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
          __new_finish          挪過來 (copy or move)
              = 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++**》….。