
C++2.0 新特性
C++11/14
C++ 2.0 New Features
第一講 語言
第二講 標準庫
侯捷
Boolan 博览

Header files

C++ 2.0 新特性包括語言和標準庫兩個層面，後者以 **header files** 形式呈現。

- C++ 標準庫的 **header files** 不帶副檔名 **(.h)**，例如 **#include <vector>**
- 新式 **C header files** 不帶副名稱 **.h**，例如 **#include <cstdio>**
- 舊式 **C header files** (帶有副名稱 **.h**) 仍可用，例如 **#include <stdio.h>**

```
#include <type_traits>
#include <unordered_set>
#include <forward_list>
#include <array>
#include <tuple>
#include <regex>
#include <thread>

using namespace std;
```

many of the TR1 features that existed in namespace **std::tr1** in the previous release (like shared_ptr and regex) are now part of the standard library under the **std** namespace.

第一步, 確認支持 C++11 : macro __cplusplus

If available, you can evaluate the predefined macro __cplusplus. For C++11, the following definition holds when compiling a C++ translation unit:

#define __cplusplus 201103L

By contrast, with both C++98 and C++03, it was:

#define __cplusplus 199711L

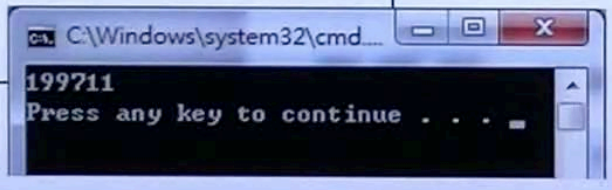
Note, however, that compiler vendors sometimes provide different values here.

...\stdbool.h

```
#ifndef __cplusplus
#define bool        _Bool
#define true        1
#define false       0
#else /* __cplusplus */
/* Supporting <stdbool.h> in C++ is a GCC extension.  */
#define _Bool       bool
#define bool        bool
#define false       false
#define true        true
#endif /* __cplusplus */
```

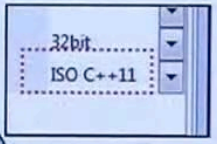
VS 2012

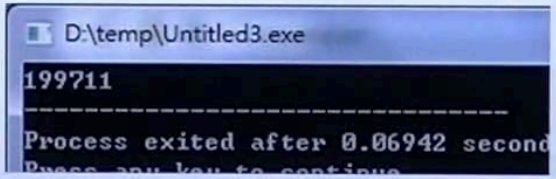
```
#include "stdafx.h"
#include <iostream>
using namespace std;

int main()
{
    cout << __cplusplus << endl;
    return 0;
}
```

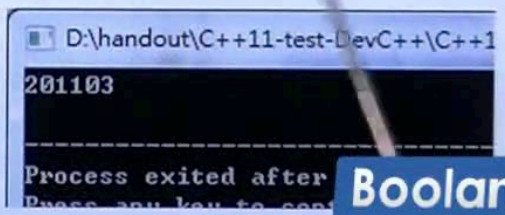


Dev-C++ 5

```
#include <iostream>

int main()
{
        std::cout << __cplusplus;
}
```







重磅出擊

語言：

- Variadic Templates
- move Semantics
- auto
- Range-base for loop
- Initializer list
- Lambdas
- …

標準庫：

- type_traits
- Unordered 容器
- forward_list
- array
- tuple
- Con-currency
- RegEx
- …

Spaces in Template Expressions

The requirement to put a space between two closing template expressions has gone:

```
vector<list<int> >;    // OK in each C++ version
vector<list<int>>;     // OK since C++11
```

G4.5.3\include\c++\profile\vector

```
  /// std::hash specialization for vector<bool>.
  template<typename _Alloc>
    struct hash<__profile::vector<bool, _Alloc>>
```

G4.5.3\include\c++\bits\stl_bvector.h

```
struct hash<_GLIBCXX_STD_D::vector<bool, _Alloc>>
```

nullptr and std::nullptr_t

C++11 lets you use nullptr instead of 0 or NULL to specify that a pointer refers to no value (which differs from having an undefined value). This new feature especially helps to avoid mistakes that occurred when a null pointer was interpreted as an integral value. For example:

```
void  f(int);
void  f(void*);
f(0);           // calls f(int)
f(NULL);        // calls f(int) if NULL is 0, ambiguous otherwise
f(nullptr);     // calls f(void*)
```

因为NULL==0所以存在二义性 指针也可以是个int的地址

nullptr is a new keyword. It automatically converts into each pointer type but not to any integral type. It has type **std::nullptr_t**, defined in **<cstddef>** (see Section 5.8.1, page 161), so you can now even overload operations for the case that a null pointer is passed. Note that std::nullptr_t counts as a fundamental data type (see Section 5.4.2, page 127).

c语言中将NULL定义为空指针，而在c++中直接定义为0，这是因为C++是强类型的，
void *是不能隐式转换成其他指针类型的。

4.9.2\include\stddef.h

```
430  #if defined(__cplusplus) && __cplusplus >= 201103L
431  #ifndef _GXX_NULLPTR_T
432  #define _GXX_NULLPTR_T
433    typedef decltype(nullptr) nullptr_t;
```

Automatic Type Deduction with auto

With C++11, you can declare a variable or an object without specifying its specific type by using **auto**. For example:

```
auto i = 42; // i has type int
double f();
auto d = f(); // d has type double
```

Using auto is especially useful where the type is a pretty long and/or complicated expression. For example:

```
vector<string> v;
...
auto pos = v.begin();          // pos has type vector<string>::iterator
auto l = [] (int x) -> bool {  // l has the type of a lambda
  ...,                         // taking an int and returning a bool
};
```

The latter is an object, representing a **lambda**.

auto keyword

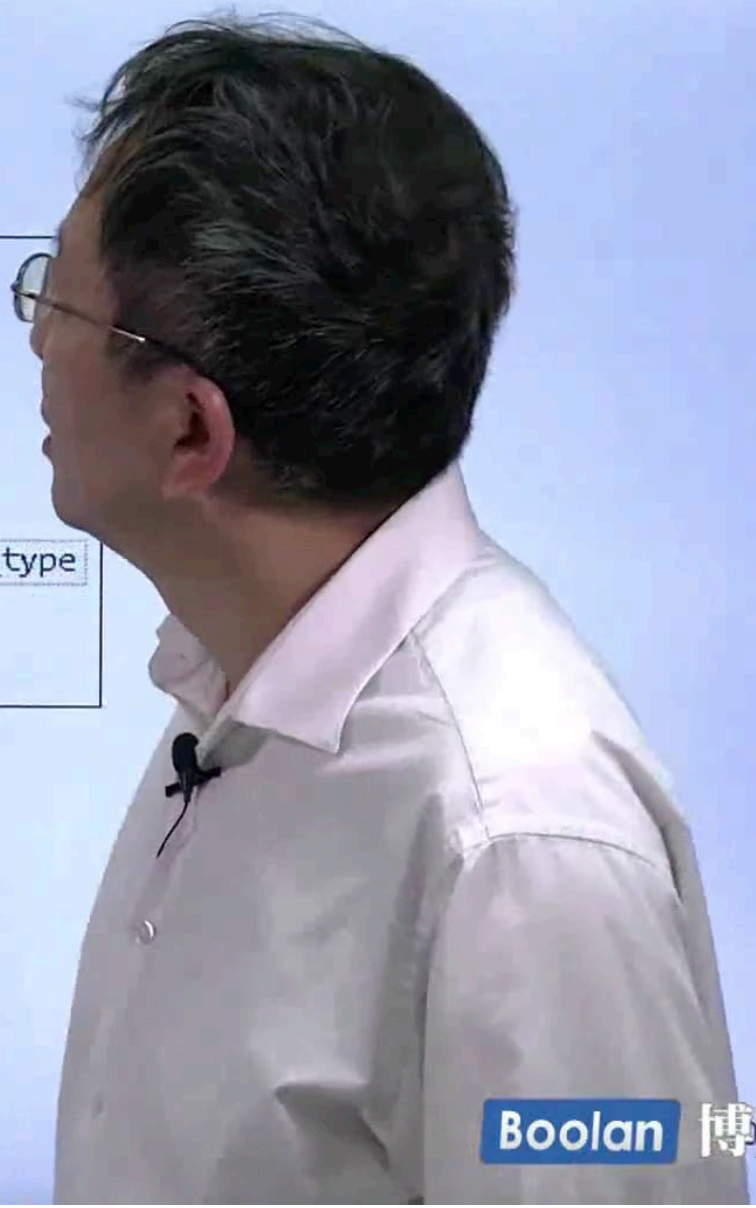
```
list<string> c;
...
list<string>::iterator ite;
ite = find(c.begin(),c.end(),target);
```

```
list<string> c;
...
auto ite = find(c.begin(),c.end(),target);
```

auto keyword

G4.5.3\include\c++\bits\stl_iterator.h

```
377  #if __cplusplus >= 201103L
378      // DR 685.
379      inline auto
380      operator-(const reverse_iterator<_IteratorL>& __x,
381                const reverse_iterator<_IteratorR>& __y)
382      -> decltype(__y.base() - __x.base())
383  #else
384      inline typename reverse_iterator<_IteratorL>::difference_type
385      operator-(const reverse_iterator<_IteratorL>& __x,
386                const reverse_iterator<_IteratorR>& __y)
387  #endif
```



Uniform Initialization

Before C++11, programmers, especially novices, could easily become confused by the question of how to initialize a variable or an object. Initialization could happen with parentheses, braces, and/or assignment operators.

For this reason, C++11 introduced the concept of **uniform initialization**, which means that for any initialization, you can use **one common syntax**. This syntax uses **braces**, so the following is possible now:

```
int values[] { 1, 2, 3};
vector<int> v { 2, 3, 5, 7, 11, 13, 17 };
vector<string> cities {
    "Berlin","New York","London","Braunschweig","Cairo","Cologne"
};
complex<double> c{4.0, 3.0}; //equivalent to c(4.0,3.0)
```

其實是利用一個事實：編譯器看到 { t1, t2… tn} 便做出一個 initializer_list<T>, 它關聯至一個 array<T,n>。調用函數 (例如 ctor) 時該 array 內的元素可被編譯器分解逐一傳給函數。但若函數參數是個 initializer_list<T>，調用者卻不能給予數個 T 參數然後以為它們會被自動轉為一個 initializer_list<T> 傳入)

這形成一個 initializer_list<string>, 背後有個 array<string,6>。調用 vectir<string> ctors 時編譯器找到了一個 vector<string> ctor 接受 initializer_list<string>。所有容器皆有如此 ctor。

這形成一個 initializer_list<double>, 背後有個 array<double,2>。調用 complex<double> ctor 時該 array 內的 2 個元素被分解傳給 ctor。complex<double> 並無任何 ctor 接受 initia

```
Rect r1 = { 3,7,20,25,&area,&print };
```

```
Rect r1(3,7,20,25);
```

```
int ia[6] = { 27,210,12,47,109,83 };
```

Initializer Lists

An **initializer list** forces so-called *value initialization*, which means that even local variables of fundamental data types, which usually have an undefined initial value, are initialized by zero (or nullptr, if it is a pointer):

```
int  i;      // i has undefined value
int  j{};    // j is initialized by 0
int* p;      // p has undefined value
int* q{};    // q is initialized by nullptr
```

Note, however, that **narrowing initializations** — those that reduce precision or where the supplied value gets modified — are not possible with braces. For example:

哎哟

```
int x1(5.3);          // OK, but OUCH: x1 becomes 5
int x2 = 5.3;         // OK, but OUCH: x2 becomes 5
int x3{5.0};          // ERROR: narrowing
int x4 = {5.3};       // ERROR: narrowing
char c1{7};           // OK: even though 7 is an int, this is not narrowing
char c2{99999};       // ERROR: narrowing (if 99999 doesn't fit into a char)
std::vector<int> v1 { 1, 2, 4, 5  };     // OK
std::vector<int> v2 { 1, 2.3, 4, 5.6 }; // ERROR: narrowing
```

在 GCC 上它們是 *[warning] narrowing conversion*

编译器调用initializer_list私有构造之前,编译器准备好array(array头)与len,传递给该构造,int类型并没有内含这个array, 于是ERROR narrowing
指针指向array,copy只是浅copy.
如今所有容器都接受任意数量的值 insert()或assign() max() min()

Initializer Lists

```
//initializer lists
int i;      //i has undefined value
int j{};    //j is initialized by 0
int* p;     //p has undefined value
int* q{};   //q is initialized by nullptr
cout << i << " " << j << ' ' << p << ' ' << q << endl;   //1  0  0x405cc

int x1(5.3);    //OK,but OUCH: x1 becomes 5
int x2=5.3;     //OK,but OUCH: x2 becomes 5
int x3{5.0};
                //[Warning] narrowing conversion of '5.0e+0' from 'double
int x4={5.3};
                //[Warning] narrowing conversion of '5.2999999999999998e+0
char c1{7};     //OK: even though 7 is an int,this is not narrowing
char c2{99999};
                //[Warning] narrowing conversion of '99999' from 'int' to 'char'
                //[Warning] overflow in implicit constant conversion [-Woverflow
cout << x1 << " " << x2 << ' ' << x3 << ' ' << x4 << ' ' << c1 << ' ' << c2 <<

vector<int> v1 {1, 2, 4, 5};        //OK
vector<int> v2 {1, 2.3, 4, 5.6 };
                //[Warning] narrowing conversion of '2.2999999999999998e+0'
                //[Warning] narrowing conversion of '5.5999999999999996e+0'
for ( auto& elem : v2 ) {
    cout << elem << ' ';  // 1 2 4 5
}
cout << endl;
```

initializer_list<>

To support the concept of initializer lists for **user-defined types**,
C++11 provides the class template **std::initializer_list<>**.
It can be used to support initializations by a list of values or in
any other place where you want to process just a list of values. For example:

```
void print (std::initializer_list<int> vals)
{
    for (auto p = vals.begin(); p != vals.end(); ++p) {   // a list of values
        std::cout << *p << "\n";
    }
}
print ({12,3,5,7,11,13,17});   // pass a list of values to print()
```

傳給 initializer_list 者，
一定必須也是個 initializer_list (or { … } 形式)

initializer_list<>

When there are constructors for both ❶ a specific number of arguments ❷ and an initializer list, the version with the initializer list is preferred.

```
class P
{
  public:
    P(int a, int b)
    {
        cout << "P(int, int), a=" << a << ", b=" << b << endl;
    }

    P(initializer_list<int> initlist)
    {
        cout << "P(initializer_list<int>), values= ";
        for (auto i : initlist)
            cout << i << ' ';
        cout << endl;
    }
};
```

complex<T> 就是這種情況

```
P p(77,5);      //P(int, int), a=77, b=5
P q{77,5};      //P(initializer_list<int>), values= 77 5
P r{77,5,42};   //P(initializer_list<int>), values= 77 5 42
P s={77,5};     //P(initializer_list<int>), values= 77 5
```

Without the constructor for the ❷ initializer list, the ❶ constructor taking two ints would be called to initialize **q** and **s**, while the initialization of **r** would be invalid.

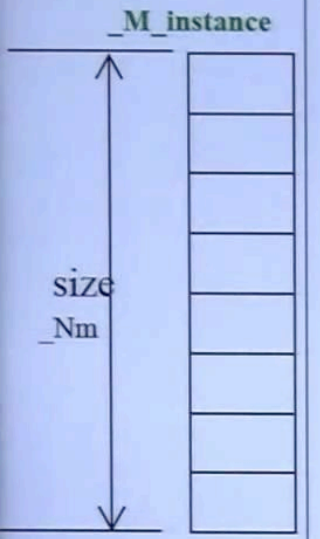
```
46  template<class _E>
47    class initializer_list
48    {
49    public:
50      typedef _E          value_type;
51      typedef const _E&   reference;
52      typedef const _E&   const_reference;
53      typedef size_t      size_type;
54      typedef const _E*   iterator;
55      typedef const _E*   const_iterator;
56
57    private:
58      iterator            _M_array;
59      size_type           _M_len;
60
61      // The compiler can call a private constructor.
62      constexpr initializer_list(const_iterator __a, size_type __l)
63      : _M_array(__a), _M_len(__l) { }
64
65    public:
66      constexpr initializer_list() noexcept
67      : _M_array(0), _M_len(0) { }
68
69      // Number of elements.
70      constexpr size_type
71      size() const noexcept { return _M_len; }
72
73      // First element.
74      constexpr const_iterator
75      begin() const noexcept { return _M_array; }
76
77      // One past the last element.
78      constexpr const_iterator
79      end() const noexcept { return begin() +
80    };
```

容器 array

TR1

array<_Tp, _Nm>

_M_instance : _Tp[_Nm]

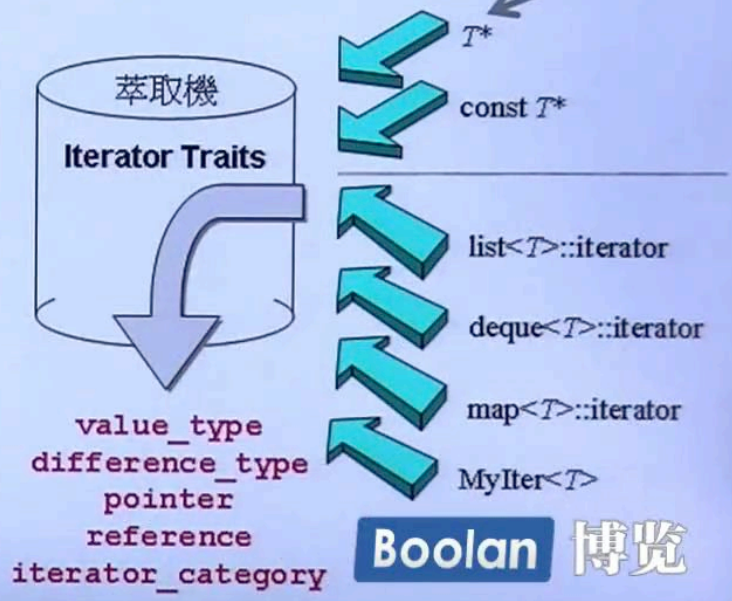



```
template<typename _Tp, std::size_t _Nm>
  struct array
  {
    typedef _Tp          value_type;
    typedef _Tp*         pointer;
    typedef value_type*  iterator;

    // Support for zero-sized arrays mandatory.
    value_type _M_instance[_Nm ? _Nm : 1];

    iterator begin()
    { return iterator(&_M_instance[0]); }

    iterator end()
    { return iterator(&_M_instance[_Nm]); }

    ...
```

其 iterator 是 native pointer,
(G2.9 vector 也是如此)

沒有ctor, 沒有dtor

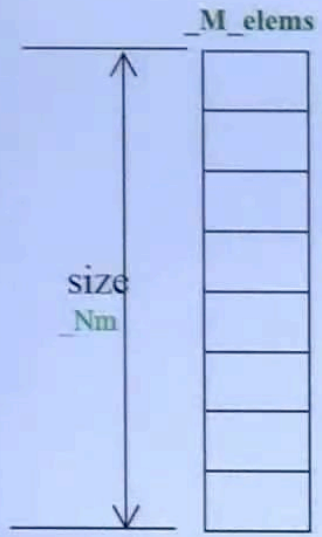
```
array<int, 10> myArray;
auto ite = myArray.begin();
   // array<int, 10>::iterator ite = …
ite += 3;
cout << *ite;
```



Boolan 博览

容器 array

G4.9




```
typename _AT_Type::_Type _M_elems;
```

```
__array_traits<_Tp, _Nm>::_Type _M_elems
```

```
_Tp _M_elems [_Nm];
```

```
template<typename _Tp, std::size_t _Nm>
  struct __array_traits
  {
    typedef _Tp _Type[_Nm];

    static constexpr _Tp&
    _S_ref(const _Type& __t, std::size_t __n) noexcept
    { return const_cast<_Tp&>(__t[__n]); }
  };
```

```
template<typename _Tp, std::size_t _Nm>
  struct array
  {
    typedef _Tp              value_type;
    typedef value_type*      pointer;
    typedef value_type&      reference;
    typedef value_type*      iterator;
    typedef std::size_t      size_type;

    // Support for zero-sized arrays mandatory.
    typedef _GLIBCXX_STD_C::__array_traits<_Tp,
    typename _AT_Type::_Type _M_elems;

    // No explicit construct/copy/destroy for aggregate type.
    iterator begin() noexcept   { return iterator(data()); }
    iterator end() noexcept     { return iterator(data() + _Nm);
    constexpr size_type size() const noexcept { return _Nm; }
    reference operator[](size_type __n) noexcept
     { return _AT_Type::_S_ref(_M_elems, __n); }
    reference at(size_type __n)  {
       if (__n >= _Nm) std::__throw_out_of_range_fmt(…
       return _AT_Type::_S_ref(_M_elems, __n);
    }
    pointer data() noexcept
    { return std::__addressof(_AT_Type::_S_ref(_M_e
    ...
  };
```

```
int a[100]
int[100]
typed
T c;
```

initializer_list<>

```
class P
{
  public:
      P(int a, int b)
      {
          cout << "P(int, int), a=" << a << ", b=" << b << endl;
      }

      P(initializer_list<int> initlist)
      {
          cout << "P(initializer_list<int>), values= ";
          for (auto i : initlist)
              cout << i << ' ';
          cout << endl;
      }
};
```

```
P p(77,5);      //P(int, int), a=77, b=5
P q{77,5};      //P(initializer_list<int>), values= 77 5
P r{77,5,42};   //P(initializer_list<int>), values= 77 5 42
P s={77,5};     //P(initializer_list<int>), values= 77 5
```

initializer_list objects are automatically constructed as if an array of elements of type T was allocated, with each of the elements in the list being copy-initialized to its corresponding element in the array, using any necessary non-narrowing implicit conversions.

The initializer_list object refers to the elements of this array without containing them: copying aninitializer_list object produces another object referring to the same underlying elements, not to new copies of them (reference semantics).

The lifetime of this temporary array is the same as the initializer_list object.

```
46
47
template<class _E>
  class initializer_list
  {
  public:
    typedef _E          value_type;
    typedef const _E&   reference;
    typedef const _E&   const_reference;
    typedef size_t      size_type;
    typedef const _E*   iterator;
    typedef const _E*   const_iterator;

  private:
    iterator            _M_array;
    size_type           _M_len;

    // The compiler can call a private constructor.
    constexpr initializer_list(const_iterator __a, size_type __l)
    : _M_array(__a), _M_len(__l) { }

  public:
    constexpr initializer_list() noexcept
    : _M_array(0), _M_len(0) { }

    // Number of elements.
    constexpr size_type
    size() const noexcept { return _M_len; }

    // First element.
    constexpr const_iterator
    begin() const noexcept { return _M_array; }

    // One past the last element.
    constexpr const_iterator
    end() const noexcept { return begin() +
};
```

initializer_list<>

vector

```
#include <initializer_list>

vector(initializer_list<value_type> __l,
       const allocator_type& __a = allocator_type())
    : _Base(__a)
{ _M_range_initialize(__l.begin(), __l.end(), random_access_iterator_tag()); }

vector& operator=(initializer_list<value_type> __l)
{ this->assign(__l.begin(), __l.end());    return *this;  }

void insert(iterator __position, initializer_list<value_type> __l)
{ this->insert(__position, __l.begin(), __l.end()); }

void assign(initializer_list<value_type> __l)
{ this->assign(__l.begin(), __l.end()); }
```

此頁顯示，如今所有容器都接受指定任意數量的值用於建構或賦值或insert()或assign(); max()和 min() 也願意接受任意參數.

D:\4.5.3\include\c++\bits\stl_set.h

```
00212: set(initializer_list<value_type> __l,
00263: operator=(initializer_list<value_type> __l)
00452: * @param list A std::initializer_list<value_type> of element
00458: insert(initializer_list<value_type> __l)
```

D:\4.5.3\include\c++\bits\stl_list.h

```
00554: list(initializer_list<value_type> __l,
00624: operator=(initializer_list<value_type> __l)
00675: assign(initializer_list<value_type> __l)
01011: insert(iterator __p, initializer_list<value_type> __l)
```

D:\4.5.3\include\c++\bits\stl_vector.h

```
00271: vector(initializer_list<value_type> __l,
00357: operator=(initializer_list<value_type> __l)
00412: assign(initializer_list<value_type> __l)
00840: insert(iterator __position, initializer_list<value_type> __l)
```

D:\4.5.3\include\c++\bits\stl_deque.h

```
00825: deque(initializer_list<value_type> __l,
00907: operator=(initializer_list<value_type> __l)
00961: assign(initializer_list<value_type> __l)
01399: insert(iterator __p, initializer_list<value_type> __l
```

D:\4.5.3\include\c++\bits\basic_string.h

```
00508: basic_string(initializer_list<_CharT> __l, const _Allc
00578: operator=(initializer_list<_CharT> __l)
00912: operator+=(initializer_list<_CharT> __l)
00978: append(initializer_list<_CharT> __l)
01114: assign(initializer_list<_CharT> __l)
01158: insert(iterator __p, initializer_list<_CharT> __l)
01617: initializer_list<_CharT> __l)
```

D:\4.5.3\include\c++\bits\forward_list.h

```
00534: forward_list(std::initializer_list<_Tp> __il,
00584: operator=(std::initializer_list<_Tp> __il)
00636: assign(std::initializer_list<_Tp> __il)
00919: insert_after(const_iterator __pos, std::initializer_list<_Tp> __il);
```

D:\4.5.3\include\c++\bits\stl_algo.h

```
04083: min(initializer_list<_Tp> __l)
04088: min(initializer_list<_Tp> __l, _Compar
04093: max(initializer_list<_Tp> __l)
04098: max(initializer_list<_Tp> __l, _Compa
04103: minmax(initializer_list<_Tp> __l)
04112: minmax(initializer_list<_Tp> __l, _Cor
```

D:\4.5.3\include\c++\valarray

```
00153: valarray(initializer_list<_Tp>);
00230: valarray& operator=(initializer_list<_Tp>);
00649: valarray<_Tp>::valarray(initializer_list<_Tp> __l)
00694: valarray<_Tp>::oper
```

D:\4.5.3\include\c++\bits\stl_map.h

```
00195: map(initializer_list<value_type> __l,
00287: operator=(initializer_list<value_type> __l)
00506: * @param list A std::initializer_list<value_type> of pairs to be
00512: insert(std::initializer_list<value_type> __list)
```

D:\4.5.3\include\c++\bits\stl_bvector.h

```
00533: vector(initializer_list<bool> __l,
00580: operator=(initializer_list<bool> __l)
00605: assign(initializer_list<bool> __l)
00799: void insert(iterator __p, initializer_list<bool> __l)
```

initializer_list<>

```
157     vector<int> v1   { 2,5,7,13,69,83,50 };
158     vector<int> v2 ( { 2,5,7,13,69,83,50 } );
159     vector<int> v3;
160     v3 = { 2,5,7,13,69,83,50 };
161     v3.insert(v3.begin()+2, { 0,1,2,3,4} );
162
163     for (auto i : v3)
164         cout << i << ' ';
165     cout << endl;   // 2 5 0 1 2 3 4 7 13 69 83 50
166
167     cout << max( { string("Ace"), string("Stacy"), string("Sabrina"), string("Barkley") } ); //Stacy
168     cout << min( { string("Ace"), string("Stacy"), string("Sabrina"), string("Barkley") } ); //Ace
169     cout << max( { 54, 16, 48, 5 } );  //54
170     cout << min( { 54, 16, 48, 5 } );  //5
```

```
vector(initializer_list<value_type> __l,
       const allocator_type& __a = allocator_type())
    : _Base(__a)
{ _M_range_initialize(__l.begin(), __l.end(),
                      random_access_iterator_tag()); }
```

```
vector& operator=(initializer_list<value_type> __l)
{ this->assign(__l.begin(), __l.end());    return *this; }
```

```
void insert(iterator __position, initializer_list<value_type> __l)
{ this->insert(__position, __l.begin(), __l.end()); }
```

```
3437    template<typename _Tp>
3438      inline _Tp
3439      min(initializer_list<_Tp> __l)
3440      { return *std::min_element(__l.begin(), __l.end()); }
```

```
3447    template<typename _Tp>
3448      inline _Tp
3449      max(initializer_list<_Tp> __l)
3450      { return *std::max_element(__l.begin(), __l.end()); }
```

explicit for ctors taking ~~more than~~ one argument

```
struct Complex
{
   int real, imag;

   Complex(int re, int im=0) : real(re), imag(im)
   { }

   Complex operator+(const Complex& x)
   { return Complex((real + x.real),
                    (imag + x.imag)); }
};
```

```
Complex c1(12,5);
Complex c2 = c1 + 5;
```

c++1.0 时 只能用于一个参数时

```
struct Complex
{
   int real, imag;

   explicit
   Complex(int re, int im=0) : real(re), imag(im)
   { }

   Complex operator+(const Complex& x)
   { return Complex((real + x.real),
                    (imag + x.imag)); }
};
```

```
Complex c1(12,5);
Complex c2 = c1 + 5;
//[Error] no match for 'operator+' (operand types are 'Complex' and
```

explicit for ctors taking more than one argument

c++2.0 可以用在多参数情况

```
class P
{
public:
  P(int a, int b) {
        cout << "P(int a, int b) \n";
  }

  P(initializer_list<int>) {
        cout << "P(initializer_list<int>) \n";
  }

  explicit P(int a, int b, int c) {
        cout << "explicit P(int a, int b, int c) \n";
  }
};

void fp(const P&) {  };
```

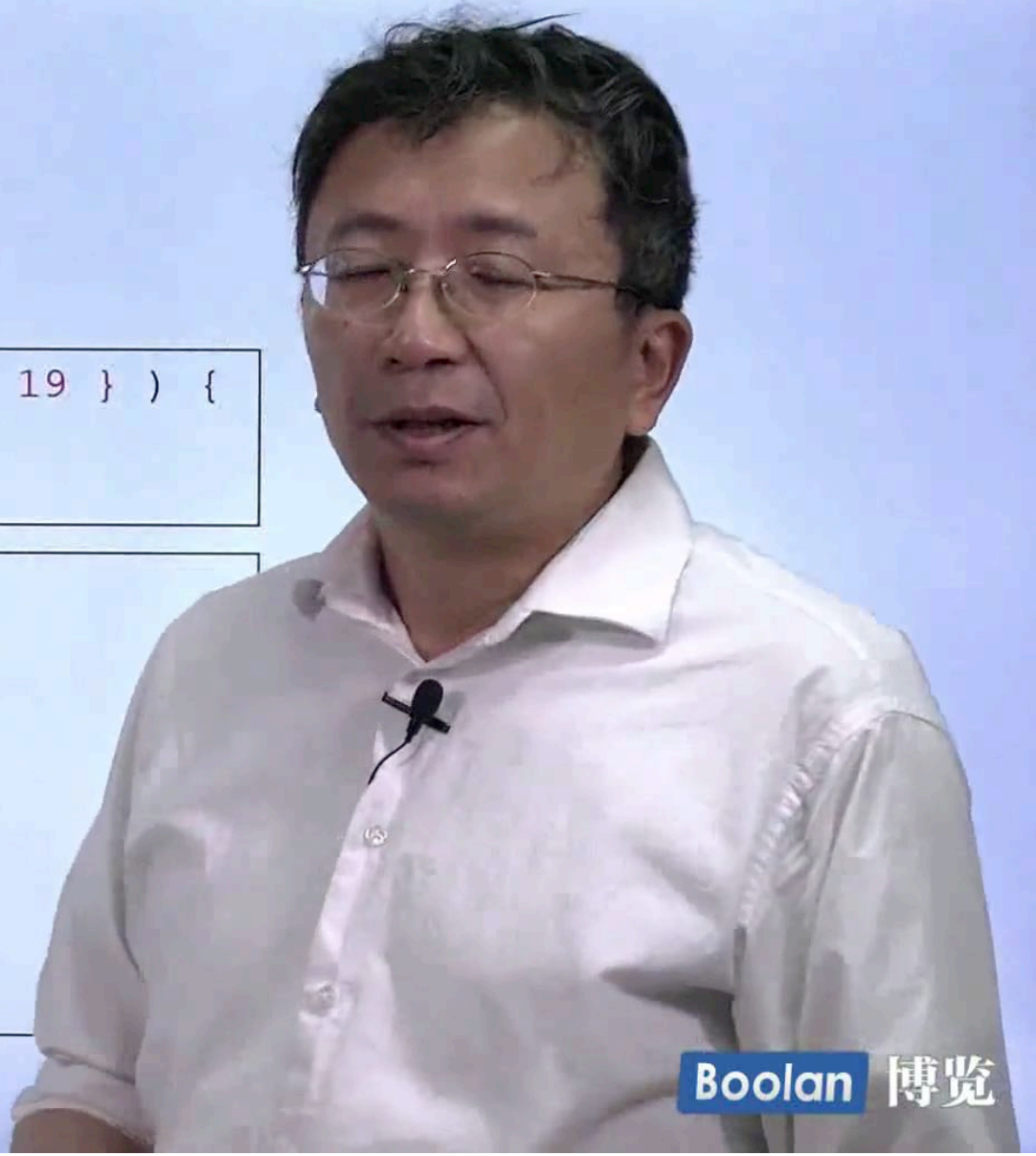
```
    //explicit ctor with multi-arguments
    P p1 (77, 5);           //P(int a, int b)
    P p2 {77, 5};           //P(initializer_list<int>)
    P p3 {77, 5, 42};       //P(initializer_list<int>)
    P p4 = {77, 5};         //P(initializer_list<int>)
//! P p5 = {77, 5, 42};     //[Error] converting to 'P' from initializer list would use explicit constructor 'P::P(
    P p6 (77,5,42);         //explicit P(int a, int b, int c)

    fp( {47,11} );          //P(initializer_list<int>)
//! fp( {47,11,3} );        //[Error] converting to 'const P' from initializer list would use explicit constructor
    fp( P{47,11} );         //P(initializer_list<int>)
    fp( P{47,11,3} );       //P(initializer_list<int>)

    P p11 {77, 5, 42, 500};     //P(initializer_list<int>)
    P p12 = {77, 5, 42, 500};   //P(initializer_list<int>)
    P p13 {10};                 //P(initializer_list<int>)
```

range-based for statement

```
for ( decl : coll ) {
    statement
}
```

```
for (int i : { 2, 3, 5, 7, 9, 13, 17, 19 } ) {
     cout << i << endl;
}
```

```
vector<double> vec;
...
for ( auto elem : vec ) {
       cout << elem << endl;
}

for ( auto& elem : vec ) {
       elem *= 3;
}
```



range-based for statement

```
for ( decl : coll ) {
    statement
}
```

```
{
  for (auto _pos=coll.begin(), _end=coll.end();  _pos!=_end; ++_pos ) {
      decl = *_pos;
      statement
  }
}
```

or

```
{
  for (auto _pos=begin(coll), _end=end(coll);  _pos!=_end; ++_pos ) {
      decl = *_pos;
      statement
  }
}
```

both begin() and end() are global

```
template <typename T>
void printElements (const T& coll)
{
    for (const auto& elem : coll) {
        cout << elem << endl;
    }
}
```

```
{
  for (auto _pos=coll.begin();  _pos != coll.end(); ++_pos ) {
      const auto& elem = *_pos;
      cout << elem << endl;
  }
}
```

range-based for statement

no explicit type conversions are possible when elements are initialized as *decl* inside the for loop. Thus, the following does not compile:

```
class C
{
   public:
      explicit C(const string& s);   // explicit(!) type conversion from strings
      ...
};

vector<string> vs;
for (const C& elem : vs)  { // ERROR, no conversion from string to C defined
   cout << elem << endl;
}
```

=default, =delete

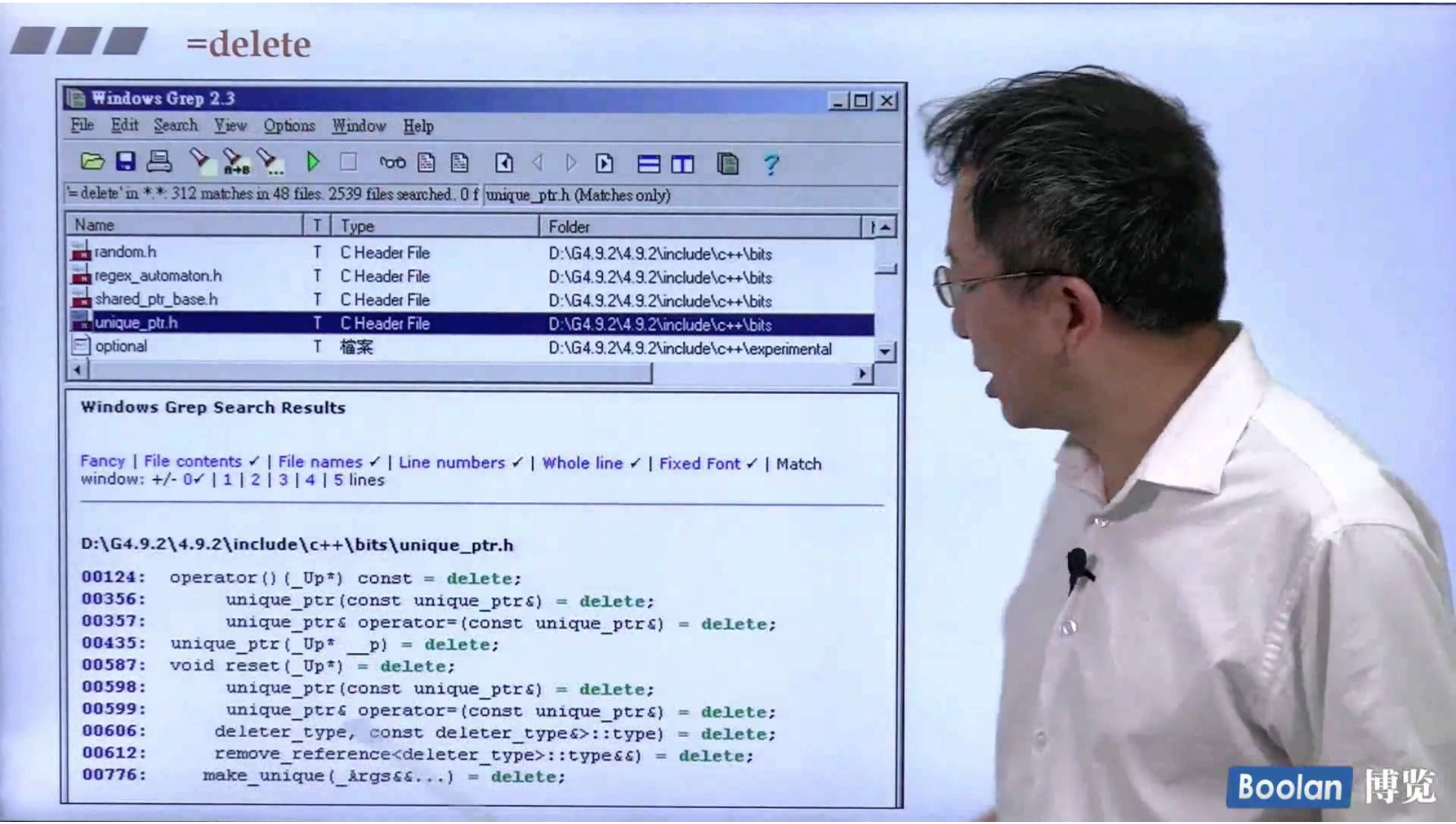
如果你自行定義了一個 ctor，那麼編譯器就不會再給你一個 default ctor。

如果你強制加上 =default，就可以重新獲得並使用 default ctor。

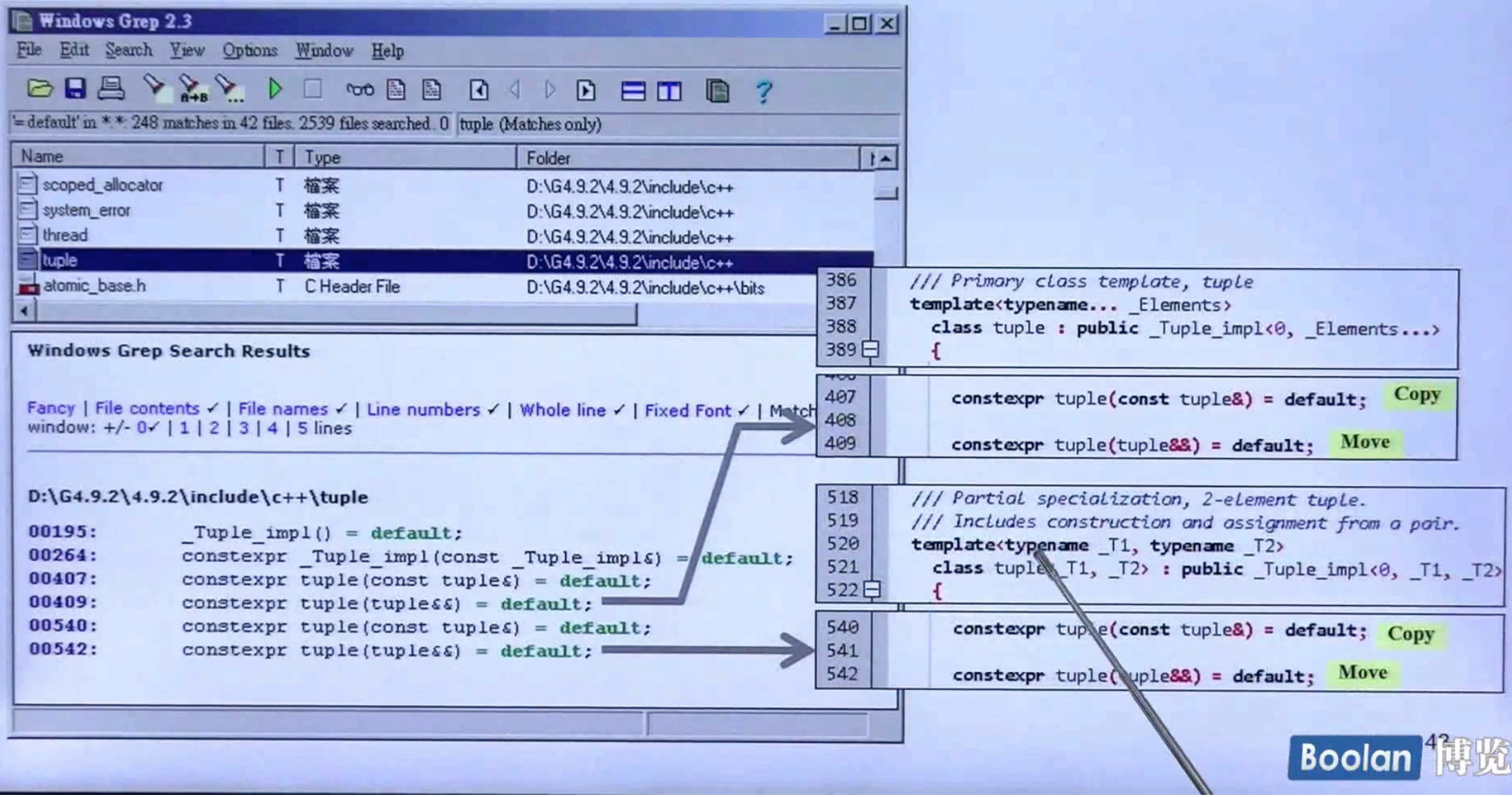
```
class Zoo
{
public:
    Zoo(int i1, int i2) : d1(i1), d2(i2) { }
    Zoo(const Zoo&) = delete;
    Zoo(Zoo&&) = default;
    Zoo& operator=(const Zoo&) = default;
    Zoo& operator=(const Zoo&&) = delete;
    virtual ~Zoo() { }
private:
    int d1, d2;
};
```

move ctor 右值引用

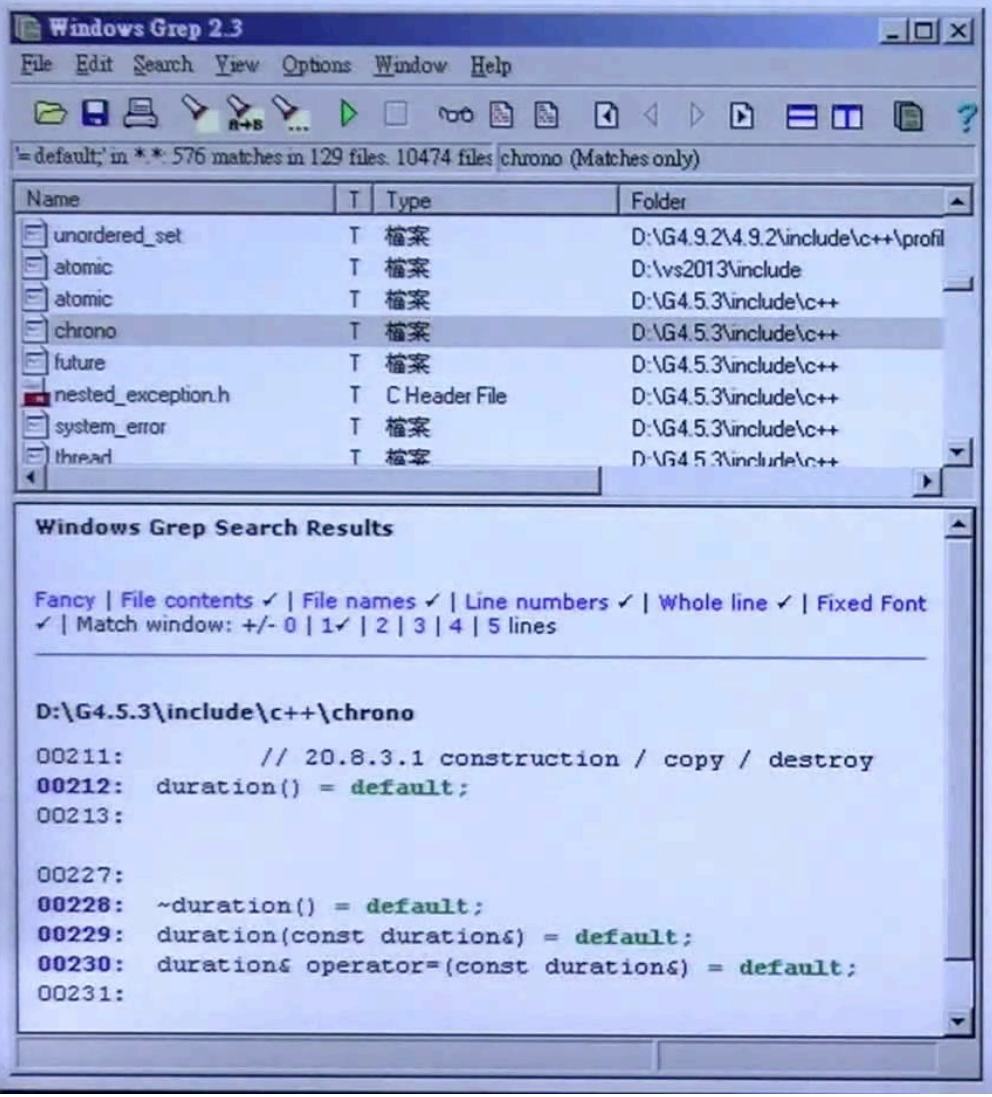
=delete



=default



=default




```
199     /// duration
200     template<typename _Rep, typename _Period>
201       struct duration
202       {
203     static_assert(!__is_duration<_Rep>::value, "rep cannot be a duration");
204     static_assert(__is_ratio<_Period>::value,
205                   "period must be a specialization of ratio");
206         static_assert(_Period::num > 0, "period must be positive");
207
208         typedef _Rep    rep;
209         typedef _Period period;
210
211         // 20.8.3.1 construction / copy / destroy
212       duration() = default;
213
```

```
227
228       ~duration() = default;
229       duration(const duration&) = default;
230       duration& operator=(const duration&) = default;
```

```
329       private:
330         rep __r;
331       };
```

=default, =delete

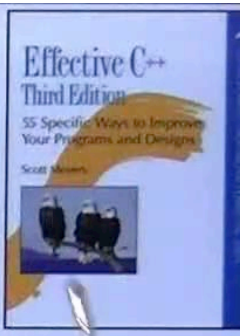
```
Foo f1(5);
Foo f2;      //如果沒有寫出 =default 版本 => [Error] no matching function for call to 'Foo::Foo()'
Foo f3(f1); //如果 copy ctor = delete;   [Error] use of deleted function 'Foo::Foo(const Foo&)')
f3 = f2;     //如果 copy assign = delete; [Error] use of deleted function 'Foo& Foo::operator=(const Foo&)')
```

```
class Foo
{
public:
   Foo(int i) : _i(i) { }
   Foo() = default;  //於是和上一個並存 (ctor 可以多個並存)

   Foo(const Foo& x) : _i(x._i) { }
   //! Foo(const Foo&) = default;  //[Error] 'Foo::Foo(const Foo&)' cannot be overloaded
   //! Foo(const Foo&) = delete;   //[Error] 'Foo::Foo(const Foo&)' cannot be overloaded

   Foo& operator=(const Foo& x) { _i = x._i; return *this; }
   //! Foo& operator=(const Foo& x) = default;  //[Error] 'Foo& Foo::operator=(const Foo&)' cannot be overloaded
   //! Foo& operator=(const Foo& x) = delete;   //[Error] 'Foo& Foo::operator=(const Foo&)' cannot be overloaded

   //!void func1() = default;   //[Error] 'void Foo::func1()' cannot be defaulted
   void func2() = delete;       //ok

   //! ~Foo() = delete;  //這會造成使用 Foo object 時出錯 => [Error] use of deleted function 'Foo::~Foo()'
   ~Foo() = default;

private:
   int _i;
};
```

copy ctor只能有一个

=default; 用於 Big-Five 之外是何意義？ ➔ 無意義, 編譯報錯.
=delete; 可用於任何函數身上. (=0 只能於 virtual 函數)

Know what functions C++ silently writes and calls



什麼時候 empty class 不再是 empty 呢？當 C++ 處理過它之後。是的，如果你自己沒聲明，編譯器就會為它聲明一個 copy ctor、一個 copy assignment operator 和一個 dtor (都是所謂 *synthesized version*)。如果你沒有聲明任何 ctor，編譯器也會為你聲明一個 default ctor。所有這些函式都是 public 且 inline。

```
class Empty { };
```

惟有當這些函式被需要（被調用），它們才會被編譯器合成。以下造成右側每個函式被編譯器合成。

```
{
Empty e1;

Empty e2(e1);

e2 = e1;
}
```

```
class Empty { };

class Empty {
public:
  Empty() { ... }
  Empty(const Empty& rhs) { ... }
  ~Empty( ) { ... }

  Empty& operator=(const Empty& rhs) { ... }
};
```

這些函式做了什麼呢？嗯，default ctor 和 dtor 主要是給編譯器一個地方用來放置「藏身幕後」的 code，像是喚起 base classes 以及 non-static members 的 ctors 和 dtors。

編譯器產出的 dtor 是 non-virtual，除非這個 class 的 base class 本身宣告有 virtual dtor。

至於 copy ctor 和 copy assignment operator，編譯器合成版只是單純將 source object 的每一個 non-static data members 拷貝到 destination object。

complex<T> 有所謂 Big-Three 嗎？

```
121   template<typename _Tp>
122     struct complex
123     {
124       /// Value typedef.
125       typedef _Tp value_type;
126
127       ///  Default constructor.  First parameter is x, second parameter is y.
128       ///  Unspecified parameters default to 0.
129       _GLIBCXX_CONSTEXPR complex(const _Tp& __r = _Tp(), const _Tp& __i = _Tp())
130       : _M_real(__r), _M_imag(__i) { }
131
132       // Lets the compiler synthesize the copy constructor
133       // complex (const complex<_Tp>&);
134       ///  Copy constructor.
135       template<typename _Up>
136         _GLIBCXX_CONSTEXPR complex(const complex<_Up>& __z)
137       : _M_real(__z.real()), _M_imag(__z.imag()) { }
```

有默認值

其實多此一舉.
連這兒都說明了!

...

沒有 operator=(const complex<...>&)
沒有 ~complex()

```
223     private:
224       _Tp _M_real;
225       _Tp _M_imag;
226     };
```

string 有所謂 Big-Three 嗎？

當然有. 而且有 Big-Five

No-Copy and Private-Copy

```
struct NoCopy {
  NoCopy() = default;     // use the synthesized default constructor
  NoCopy(const NoCopy&) = delete;     // no copy
  NoCopy &operator=(const NoCopy&) = delete;     // no assignment
  ~NoCopy() = default;     // use the synthesized destructor
  // other members
};
```

= delete 告訴編譯器不要定義它。必須出現在聲明式。適用於任何成員函數。但若用於 dtor 後果自負。

```
struct NoDtor {
  NoDtor() = default;  // use the synthesized default constructor
  ~NoDtor() = delete;  // we can't destroy objects of type NoDtor
};
NoDtor nd;                    // error: NoDtor destructor is deleted
NoDtor *p = new NoDtor();     // ok: but we can't delete p
delete p;                     // error: NoDtor destructor is deleted
```

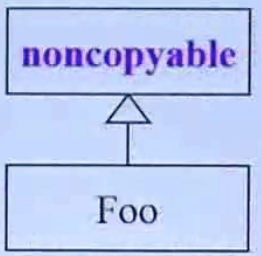
```
class PrivateCopy {
private:
  // no access specifier; following members are private by default; see § 7.2 (p.268)
  // copy control is private and so is inaccessible to ordinary user code
  PrivateCopy(const PrivateCopy&);
  PrivateCopy &operator=(const PrivateCopy&);
  // other members
public:
  PrivateCopy() = default; // use the synthesized default constructor
  ~PrivateCopy(); // users can define objects of this type but not copy them
};
```

此 class 不允許被 ordinary user code copy，但仍可被 friends 和 members copy。若欲完全禁止，不但必須把 copy controls 放到 private 內且不可定義之。

boost::noncopyable

...\boost\noncopyable.hpp :

```
namespace boost {
// Private copy constructor and copy assignment ensure classes derived from
// class noncopyable cannot be copied.
// Contributed by Dave Abrahams

namespace noncopyable_  // protection from unintended ADL
{
 class noncopyable
 {
 protected:
    noncopyable() { }
    ~noncopyable() { }
 private:  // emphasize the following members are private
    noncopyable( const noncopyable& );
    const noncopyable& operator=( const noncopyable& );
 };
}

typedef noncopyable_::noncopyable noncopyable;
} // namespace boost
```



Alias Template (template typedef)

化名

```
template <typename T>
using Vec = std::vector<T,MyAlloc<T>>;    // standard vector using own allocator
```

the term

```
Vec<int> coll;
```

is equivalent to

```
std::vector<int,MyAlloc<int>> coll;
```

It is not possible to partially or explicitly specialize an alias template.

不能对化名特化或偏特化

//使用 macro 無法達到相同效果

```
#define Vec<T> template<typename T> std::vector<T,MyAlloc<T>>
```

於是：

```
Vec<int> coll;
```

⇩

```
template<typename int> std::vector<int,MyAlloc<int>>;  //這不是我要的
```

使用 typedef 亦無法達到相同效果，因為 typedef 是不接受參數的。

我們至多寫成這樣：`typedef std::vector<int,MyAlloc<int>> Vec;` //這不是我要的

Alias Template

難道只是為了少打幾個字？

我的實戰經驗如下 ~~~

```
//[Error] variable or field 'test_moveable' declared void
//[Error] 'Container' was not declared in this scope
//[Error] 'T' was not declared in this scope
void test_moveable(Container cntr, T elem)
{
Container<T> c;

   for(long i=0; i< SIZE; ++i)
       c.insert(c.end(), T());

   output_static_data(T());
   Container<T> c1(c);
   Container<T> c2(std::move(c));
   c1.swap(c2);
}
```


```
test_moveable(list, MyString);
test_moveable(list, MyStrNoMove);
```

```
//[Error] 'Container' is not a template
template<typename Container, typename T>
void test_moveable(Container cntr, T elem)
{
Container<T> c;

   for(long i=0; i< SIZE; ++i)
       c.insert(c.end(), T());

   output_static_data(T());
   Container<T> c1(c);
   Container<T> c2(std::move(c));
   c1.swap(c2);
}
```


```
test_moveable(list(), MyString());
test_moveable(list(), MyStrNoMove());
```

Alias Template

難道只是為了少打幾個字？

我的實戰經驗如下～～～

//[Error] variable or field 'test_moveable' declared void
//[Error] 'Container' was not declared in this scope
//[Error] 'T' was not declared in this scope

```
void test_moveable(Container cntr, T elem)
{
Container<T> c;

   for(long i=0; i< SIZE; ++i)
       c.insert(c.end(), T());

   output_static_data(T());
   Container<T> c1(c);
   Container<T> c2(std::move(c));
   c1.swap(c2);
}
```

天方夜譚

```
test_moveable(list, MyString);
test_moveable(list, MyStrNoMove);
```

//[Error] expected nested-name-specifier before 'Contain

```
template<typename Container, typename T>
void test_moveable(Container cntr, T elem)
{
 typename Container<T> c;

   for(long i=0; i< SIZE; ++i)
       c.insert(c.end(), T());

   output_static_data(T());
   Container<T> c1(c);
   Container<T> c2(std::move(c));
   c1.swap(c2);
}
```

天方夜譚

```
test_moveable(list(), MyString());
test_moveable(list(), MyStrNoMove());
```

採用 function template + iterator + traits

```
template<typename T>
void output_static_data(const T& obj)
{
  cout << … // static data of obj
}
```

有沒有 template 語法能夠在模板接受一個 template 參數 Container 時, 當 Container 本身又是個 class template, 能取出 Container 的 template 參數？例如收到一個 vector<string>, 能夠取出其元素類型 string？

另向思考：既然談的是 “Container”, 其 iterator 就能夠回答 value_type 這種問題, 間接解決了上述提問. 但在沒有 iterator 和 traits 的情況下呢 ?!

```
template<typename Container>
void test_moveable(Container c)
{
typedef typename iterator_traits<typename Container::iterator>::value_type Valtype;

  for(long i=0; i< SIZE; ++i)
    c.insert(c.end(), Valtype());

  output_static_data(*(c.begin()));
  Container c1(c);
  Container c2(std::move(c));
  c1.swap(c2);
}
```

```
test_moveable(list<MyString>());
test_moveable(list<MyStrNoMove>());

test_moveable(vector<MyString>());
test_moveable(vector<MyStrNoMove>());

test_moveable(deque<MyString>());
test_moveable(deque<MyStrNoMove>());
…
//使用 RB-tree 時, 元素需提供 operator<
//使用 hashtable 時, 元素需提供 operator<, hash function
//使用 multi-容器時, 元素還需提供 operator==
```

template template parameter

```
template<typename T,
         template <class>
           class Container
        >
class XCls
{
private:
   Container<T> c;
public:
  XCls()  { //constructor
     for(long i=0; i< SIZE; ++i)
         c.insert(c.end(), T());

     output_static_data(T());
     Container<T> c1(c);
     Container<T> c2(std::move(c));
     c1.swap(c2);
  }
};
```

```
213   template<typename _Tp, typename _Alloc = std::allocator<_Tp> >
214     class vector : protected _Vector_base<_Tp, _Alloc>
215     {
```

Alias templates are never deduced by template argument deduction when deducing a template template parameter.

```
XCls<MyString, vector> c1;

//[Error] type/value mismatch at argument 2 in template parameter list for
          'template<class T, template<class> class Container> class XCls'
//[Error] expected a template of type 'template<class> class Container',
          got 'template<class _Tp, class _Alloc> class std::vector'
//[Error] invalid type in declaration before ';' token
```

当容器为模板模板参数时
容器第二个参数编译器无法自动推出来

template template parameter + alias template

```
template<typename T,
         template <class>
           class Container
        >
class XCls
{
private:
   Container<T> c;
public:
  XCls()  { //constructor
      for(long i=0; i< SIZE; ++i)
          c.insert(c.end(), T());

      output_static_data(T());
      Container<T> c1(c);
      Container<T> c2(std::move(c));
      c1.swap(c2);
  }
};
```

```
template<typename T>
using Vec = vector<T, allocator<T>>;

template<typename T>
using Lst = list<T, allocator<T>>;

template<typename T>
using Deq = deque<T, allocator<T>>;
```

不得在 function body 之內聲明

```
XCls<MyString, Vec>  c1;
XCls<MyStrNoMove, Vec> c2;

XCls<MyString, Lst>  c3;
XCls<MyStrNoMove, Lst> c4;

XCls<MyString, Deq>  c5;
XCls<MyStrNoMove, Deq> c6;
…
//使用 RB-tree 時, 元素需提供 operator<
//使用 hashtable 時, 元素需提供 operator<, hash function
//使用 multi-容器時, 元素還需提供 operator==
```

Type Alias (similar to typedef)

```
// type alias, identical to
// typedef void (*func)(int, int);
using func = void (*) (int,int);

// the name 'func' now denotes a pointer to function:
void example(int, int) {}
func fn = example;
```

函数指针

type alias等价于typedef,没有什么不同

There is no difference between a type alias declaration and typedef declaration. This declaration may appear in block scope, class scope, or namespace scope.

Alias template

```
// type alias used to hide a template parameter
template <class CharT> using mystring =
   std::basic_string<CharT,std::char_traits<CharT>>;

mystring<char> str;
```

<string> 和 <string_fwd.h> 都有以下 typedef :

```
typedef basic_string<char>   string;
```

```
// type alias can introduce a member typedef name
template<typename T>
struct Container {
   using value_type = T;
};

// which can be used in generic programming
template<typename Cntr>
void fn2(const Cntr& c)
{
   typename Cntr::value_type n;
}
```

➔ typedef T value_type;

using

`using namespace std;` `using std::count;`

- using-directives for namespaces and using-declarations for namespace members
- using-declarations for class members
- type alias and alias template declaration (since C++11)

```
using func = void (*) (int,int);
```

```
template<typename T>
struct Container {
   using value_type = T;
};
```

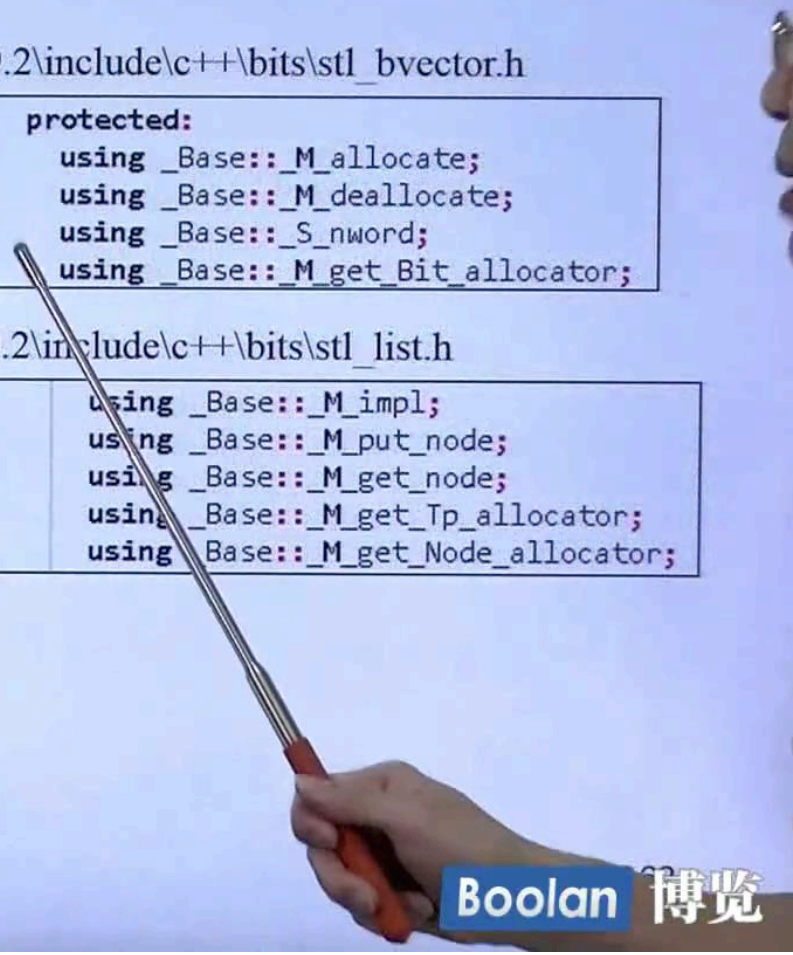
```
template <class CharT> using mystring =
   std::basic_string<CharT,std::char_traits<CharT>>;
```

…\4.9.2\include\c++\bits\stl_bvector.h

```
551   protected:
552     using _Base::_M_allocate;
553     using _Base::_M_deallocate;
554     using _Base::_S_nword;
555     using _Base::_M_get_Bit_allocator;
```

…\4.9.2\include\c++\bits\stl_list.h

```
477     using _Base::_M_impl;
478     using _Base::_M_put_node;
479     using _Base::_M_get_node;
480     using _Base::_M_get_Tp_allocator;
481     using _Base::_M_get_Node_allocator;
```



基本使用

```
namespace ns2 {
#ifdef isNs1
    using ns1::func;    /// ns1中的函数
#elif isGlobal
    using ::func; /// 全局中的函数}
```

改变访问性

```
class Base{
public:
 std::size_t size() const { return n;  }
protected:
 std::size_t n;};
class Derived : private Base {
public:
 using Base::size;
protected:
 using Base::n;};
```

上述size可以按public的权限访问，n可以按protected的权限访问

函数重载

在继承过程中，派生类可以覆盖重载函数的0个或多个实例，一旦定义了一个重载版本，那么其他的重载版本都会变为不可见。若基类的重载函数中派生类只修改一个，又要让其他的保持可见，必须要重载所有版本，十分繁琐。

```
class Derived : private Base {
    public:
        using Base::f;
        void f(int n){
            cout<<"Derived::f(int)"<<endl; }};
```

派生类中使用using声明语句指定一个名字而不指定形参列表，所以一条基类成员函数的using声明语句就可以把该函数的所有重载实例添加到派生类的作用域中。此时，派生类只需要定义其特有的函数就行了

c++2.0特性 type alias and alias template

取代typedef

对应typedef A B ，将B定义为A类型，也就是给A类型一个别名B，使用using B=A可以进行同样的操作

noexcept

void foo () **noexcept**; → void foo () **noexcept**(true);

保证这个函数一定不会丢出异常

declares that foo() won't throw. If an exception is not handled locally inside foo() — thus, if foo() throws — the program is terminated, calling **std::terminate**(), which by default calls **std::abort**().

中断

You can even specify a condition under which a function throws no exception. For example, for any type *Type*, the global swap() usually is defined as follows:

```
void swap (Type& x, Type& y) noexcept(noexcept(x.swap(y)))
{
   x.swap(y);
}
```

Here, inside noexcept(...), you can specify a Boolean condition under which no exception gets thrown: Specifying noexcept without condition is a short form of specifying noexcept(true).

Exception 是一門大學問，另課探討
swap( ) 也是一門大學問，另課探討

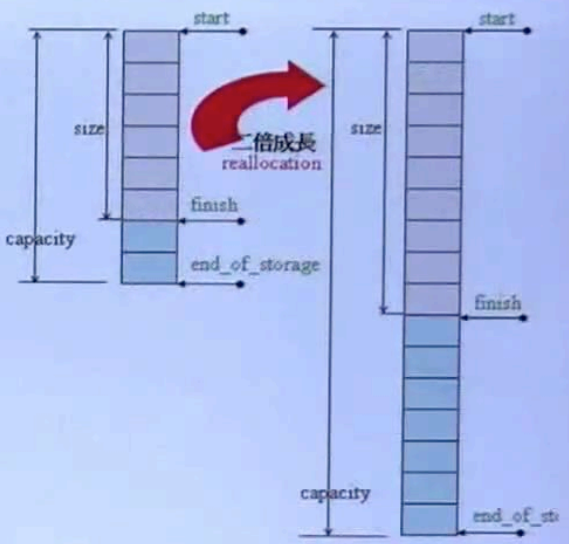
异常可传递　a调用b b抛出异常,a没处理,就会抛出异常
1.异常的回传机制:调用foo如果抛出异常，foo会接着往上层抛出异常，如果最上层没有处理，则会调用terminate函数，terminate函数内部调用abort，使程序退出
2.在使用vector deque等容器的移动构造和移动赋值的时候，如果移动构造和移动赋值没有加上noexcept，则容器增长的时候不会调用move constructor，效率就会偏低，所以后面需要加上noexcept，安心使用

noexcept

http://stackoverflow.com/questions/8001823/how-to-enforce-move-semantics-when-a-vector-grows

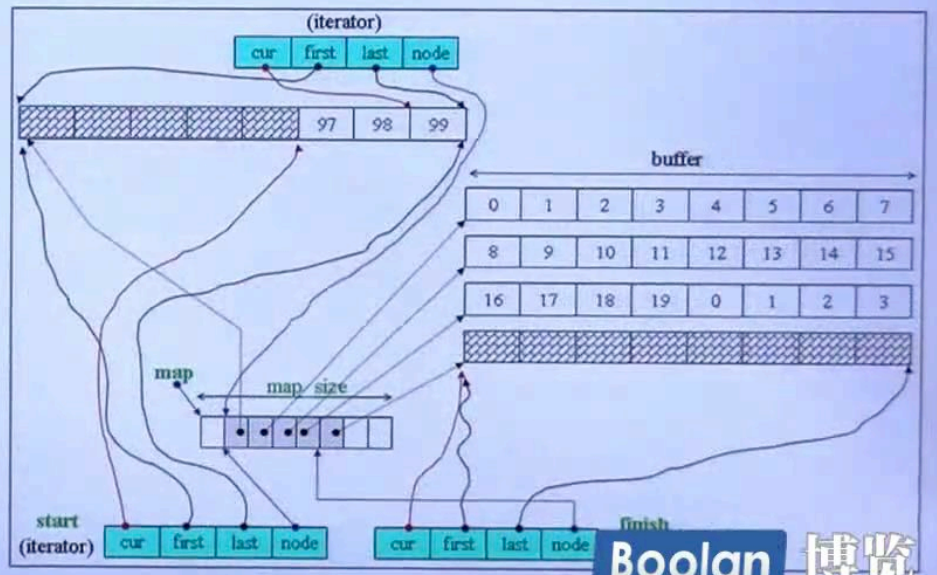
You need to inform C++ (*specifically* **std::vector**) that your *move constructor and destructor* does not throw. Then the *move constructor* will be called when the vector grows. If the *constructor* is not noexcept, std::vector can't use it, since then it can't ensure the exception guarantees demanded by the standard.

注意，growable conatiners (會發生 memory reallocation) 只有二種：vector 和 deque.




```
class MyString {
private:
   char* _data;
   size_t _len;
...
public:
   //move constructor
   MyString(MyString&& str) noexcept
      : _data(str._data), _len(str._len)  { ... }

   //move assignment
   MyString& operator=(MyString&& str) noexcept
      { ... return *this;  }
...
```

move function必须用noexcept
不然vector不敢使用它成长(搬动)



override

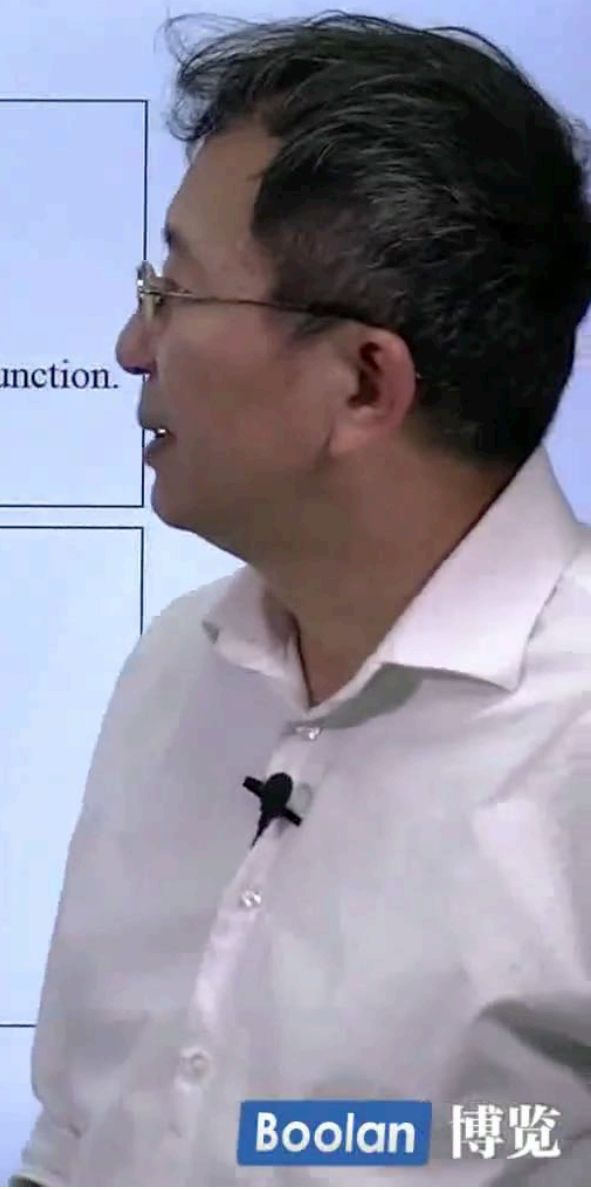
给我们对虚函数的重写做检查

```
struct Base {
   virtual void vfunc(float) {    }
};
struct Derived1 : Base {
   virtual void vfunc(int) { }
        //accidentally create a new virtual function, when one intended to override a base class function.
        //This is a common problem, particularly when a user goes to modify the base class.
};
```

如果写错了编译器会认为重新定义了一个虚函数

```
struct Derived2 : Base {
   virtual void vfunc(int) override { }
        //[Error] 'virtual void Derived2::vfunc(int)' marked override, but does not override

        //override means that the compiler will check the base class(es) to see if there is
        //a virtual function with this exact signature.
        //And if there is not, the compiler will indicate an error.

   virtual void vfunc(float) override { }
};
```

让编译器帮你识错 告诉编译器这是一个重写



final

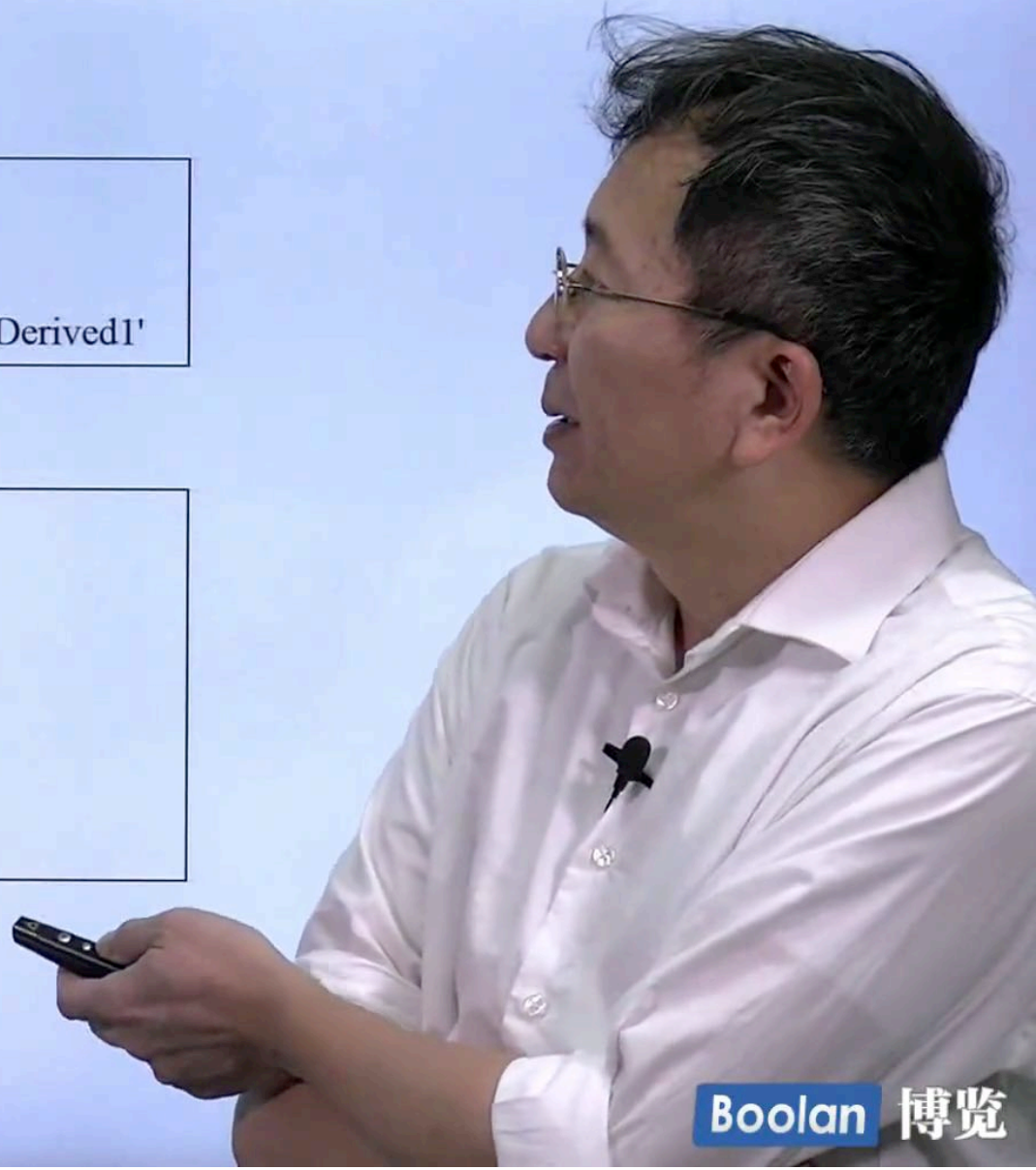
```
struct Base1 final { };
```

用于类说明不会再有子类继承

```
struct Derived1 : Base1 { };
   //[Error] cannot derive from 'final' base 'Base1' in derived type 'Derived1'
```

```
struct Base2 {
   virtual void f() final;
};

struct Derived2 : Base2 {
   void f();
      //[Error] overriding final function 'virtual void Base2::f()'
};
```

用于函数说明不会再被覆写



decltype

unevaluated context

defines a type equivalent to the type of an expression

http://en.cppreference.com/w/cpp/utility/declval

declval converts any type T to a reference type, making it possible to use member functions in decltype expressions without the need to go through constructors.

https://msdn.microsoft.com/en-us/magazine/ee336130.aspx

decltype allows the compiler to infer the return type of a function based on an arbitrary expression and makes *perfect forwarding* more generic. In past versions, for two arbitrary types T1 and T2, there was no way to deduce the type of an expression that used these two types. The decltype feature allows you *to state*, for example, an expression that has template arguments, such as sum<T1, T2>() has the type T1+T2.

– <the C++ Standard Library 2/e>

By using the new decltype keyword, you can let the compiler find out the type of an expression. This is the realization of the often requested **typeof** feature. However, the existing **typeof** implementations were inconsistent and incomplete, so C++11 introduced a new keyword. For example:

```
map<string, float> coll;
decltype(coll)::value_type elem;
```

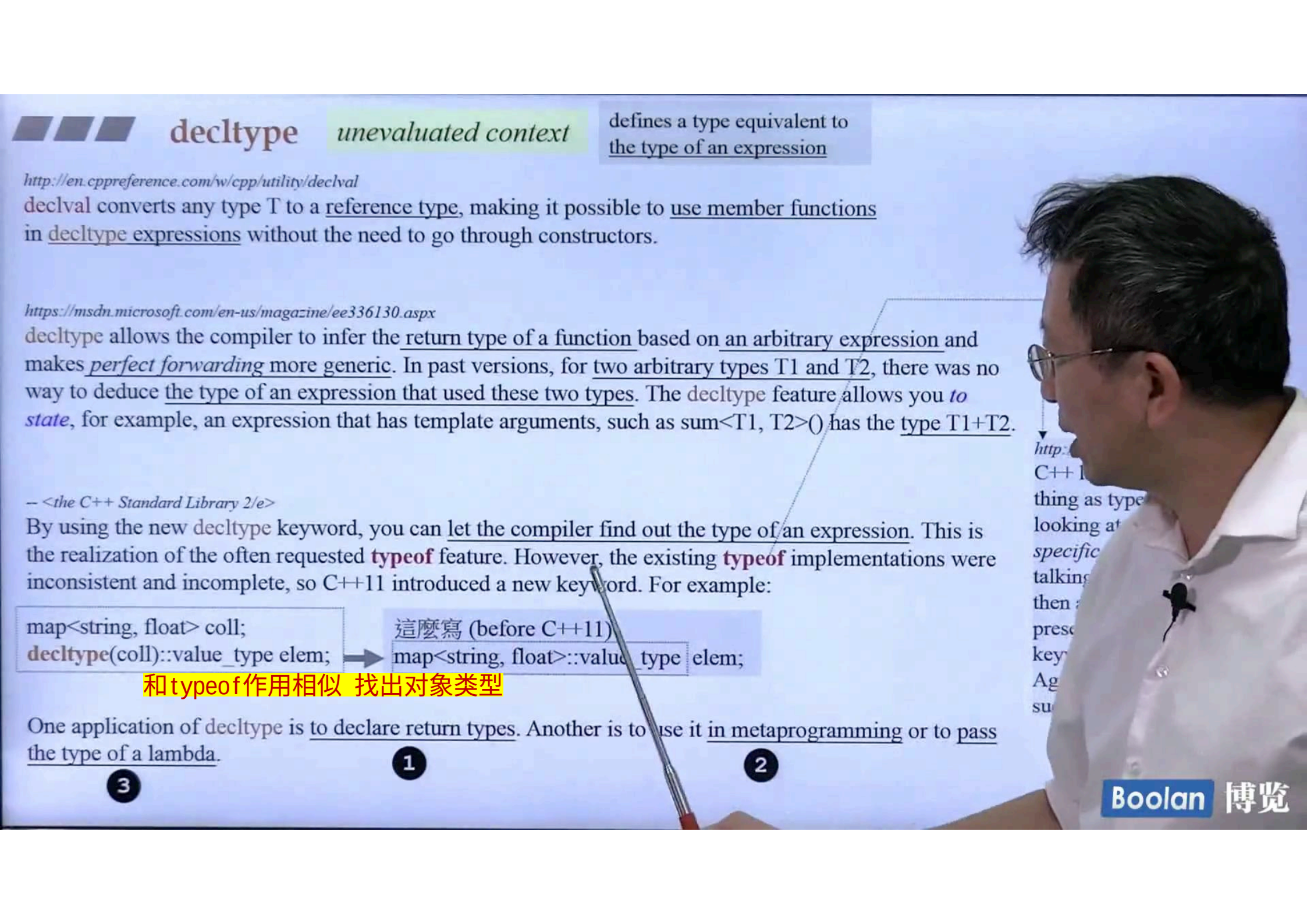
這麼寫 (before C++11)

```
map<string, float>::value_type elem;
```

和typeof作用相似 找出对象类型

One application of decltype is to declare return types. Another is to use it in metaprogramming or to pass the type of a lambda.

1 2 3



decltype

defines a type equivalent to the type of an expression

-- *<the C++ Standard Library 2/e>*

By using the new decltype keyword, you can let the compiler find out the type of an expression. This is the realization of the often requested **typeof** feature.

1

One application of decltype is to declare return types. Another is to use it in metaprogramming or to pass the type of a lambda.

2

3

decltype, used to declare return types

Sometimes, the return type of a function depends on an expression processed with the arguments.
However, something like

```
template <typename T1, typename T2>
decltype(x+y) add(T1 x, T2 y);
```

使add的返回值类型为x+y的类型

was not possible before C++11, because the return expression uses objects not introduced or in scope yet.

But with C++11, you can alternatively declare the return type of a function behind the parameter list :

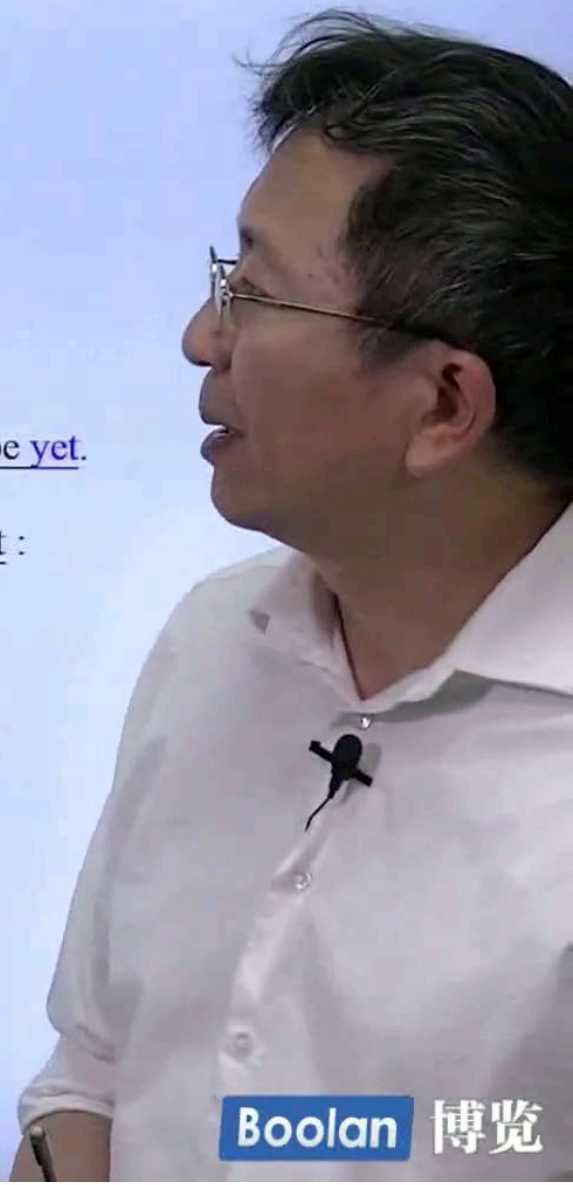
```
template <typename T1, typename T2>
auto add(T1 x, T2 y) -> decltype(x+y);
```

1

新的函数返回类型指定方式

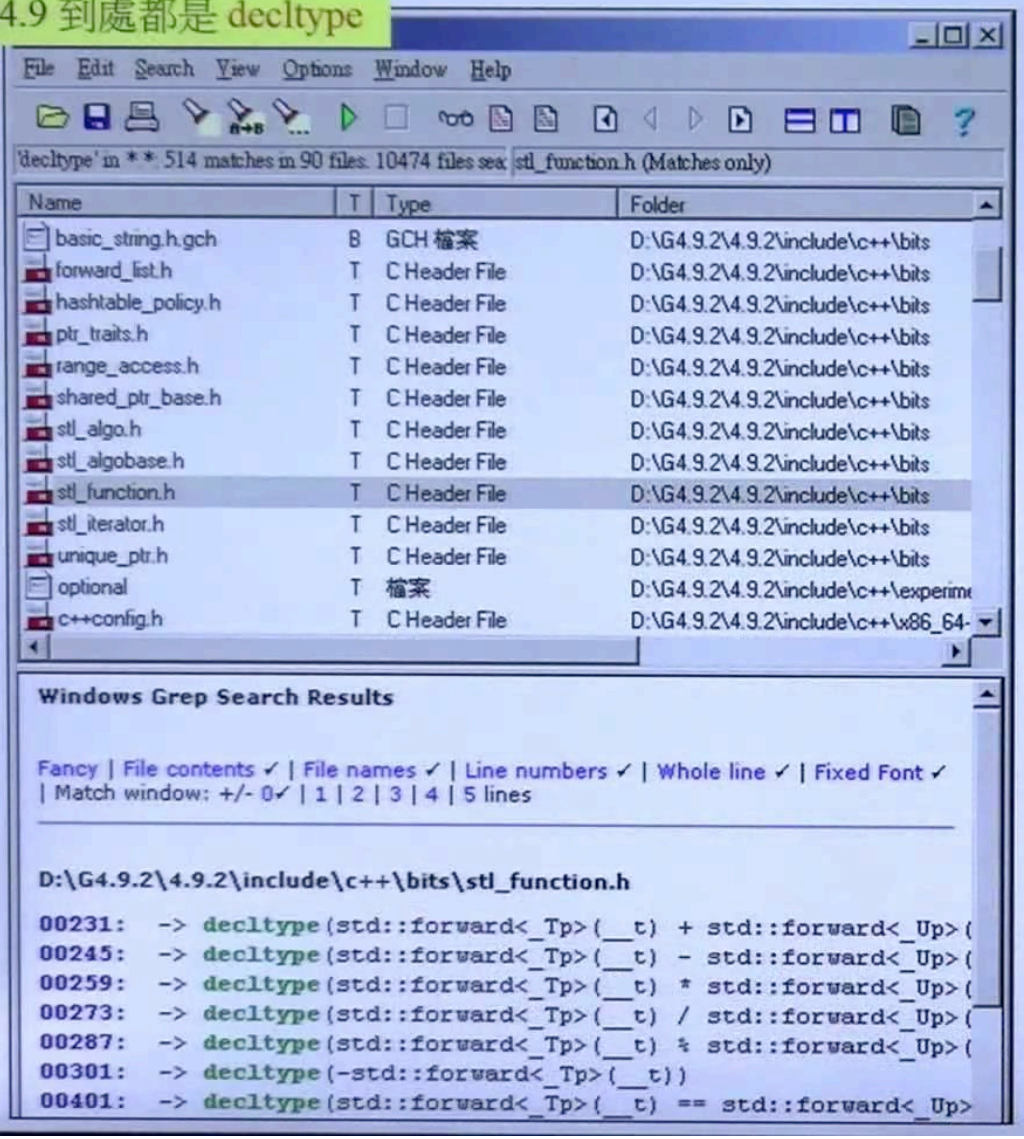
This uses the same syntax as for lambdas to declare return types.

[...] (...) mutable$_{opt}$ *throwSpec*$_{opt}$ -> *retType*$_{opt}$ {...}



decltype

G4.9 到處都是 decltype



...\4.9.2\include\c++\bits\stl_function.h

```
166   template<typename _Tp>
167     struct plus : public binary_function<_Tp, _Tp, _Tp>
168     {
169       _Tp
170       operator()(const _Tp& __x, const _Tp& __y) const
171       { return __x + __y; }
172     };
```

```
219   #if __cplusplus > 201103L
220
221   #define __cpp_lib_transparent_operators 201210
222   //#define __cpp_lib_generic_associative_lookup 201304
223
224     template<>
225       struct plus<void>
226       {
227         template <typename _Tp, typename _Up>
228       auto
229       operator()(_Tp&& __t, _Up&& __u) const
230       noexcept(noexcept(std::forward<_Tp>(__t) + std::forwa
231       -> decltype(std::forward<_Tp>(__t) + std::forward<_Up
232       { return std::forward<_Tp>(__t) + std::forward<_Up>(_
233
234         typedef __is_transparent is_transparent;
235       };
```

decltype

```
template <typename T>
void test18_decltype(T obj)
{
   map<string, float>::value_type elem1;

   map<string, float> coll;
   decltype(coll)::value_type elem2;

   //瞧我故意設計本測試函數為 template function, 接受任意參數 T obj


2  //如今有了 decltype 我可以這樣：
   typedef typename decltype(obj)::iterator iType;

     typedef typename T::iterator iType;

   decltype(obj) anotherObj(obj);
}
```

當我們手上有 type，可取其 inner typedef，沒問題.

面對 obj 取其 class type 的 inner typedef
因為如今我們有了工具 decltype

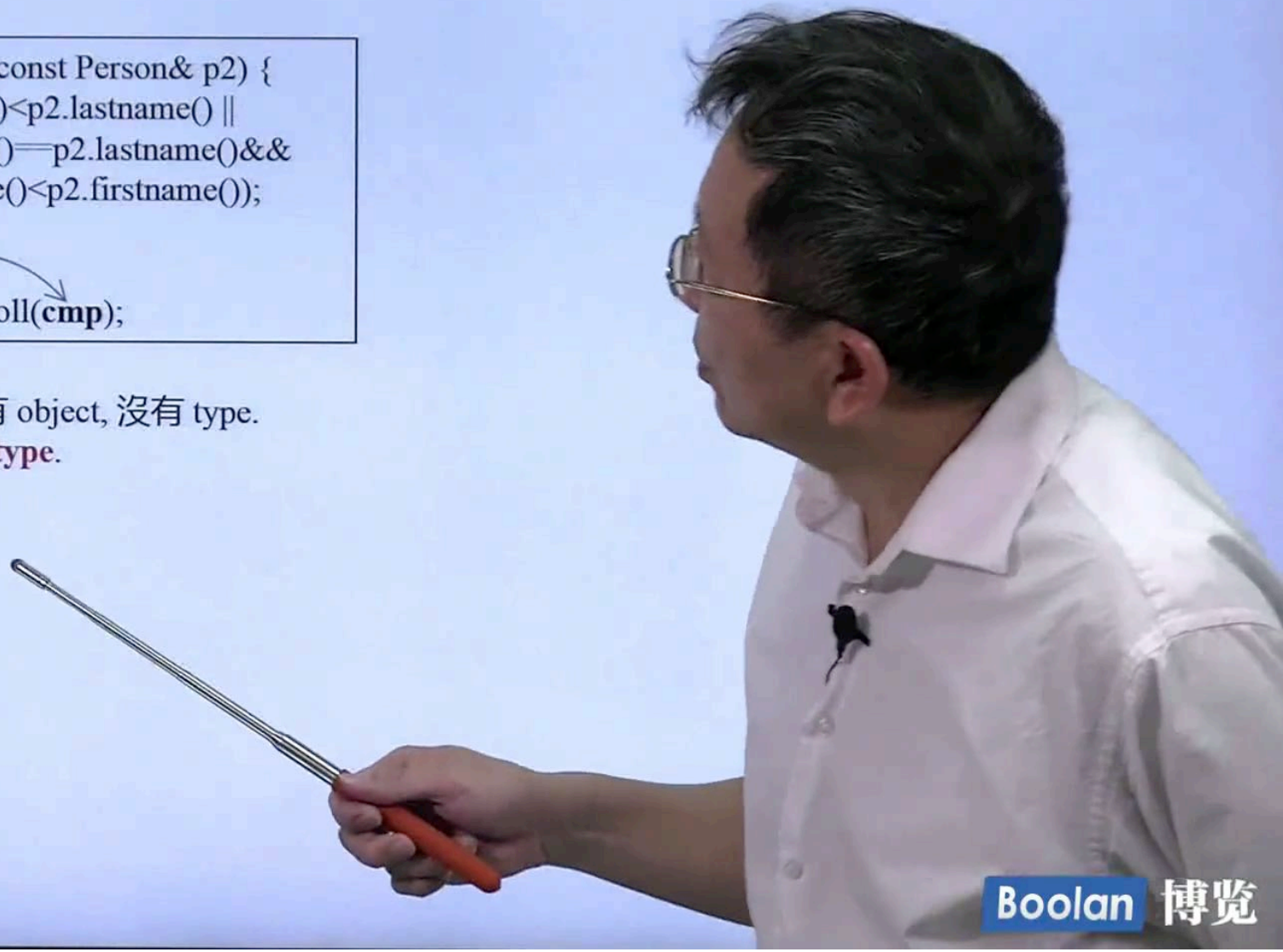
```
//! test18_decltype(complex<int>()); //編譯失敗
```

complex沒有迭代器

decltype, used to pass the type of a lambda

```
auto cmp = [] (const Person& p1, const Person& p2) {
               return p1.lastname()<p2.lastname() ||
                      (p1.lastname()==p2.lastname()&&
                       p1.firstname()<p2.firstname());
           };
...
std::set<Person,decltype(cmp)> coll(cmp);
```

面對 lambda, 我們手上往往只有 object, 沒有 type.
要獲得其 type 就得借助於 **decltype**.



decltype (expression)
推导出表达式类型

与using/typedef合用，用于定义类型
```
using nullptr_t = decltype(nullptr);
vector<int >vec;
typedef decltype(vec.begin()) vectype;
```

重用匿名类型

泛型编程中结合auto，用于追踪函数的返回值类型 (最大用处

判别规则:
如果e是一个没有带括号的标记符表达式或者类成员访问表达式，那么的decltype（e）就是e所命名的实体的类型。此外，如果e是一个被重载的函数，则会导致编译错误。 否则 ，假设e的类型是T，如果e是一个将亡值(临时对象)，那么decltype（e）为T&& 如果e是一个左值，那么decltype（e）为T&。 否则decltype（e）为T。

左值情况:
```
decltype ((i))var6 = i;     //int&
decltype (true ? i : i) var7 = i; //int&  条件表达式返回左值。
decltype (++i) var8 = i; //int&  ++i返回i的左值
decltype(arr[5]) var9 = i;//int&. []操作返回左值
decltype(*ptr)var10 = i;//int& *操作返回左值
decltype("hello")var11 = "hello"; //const char(&)[9]  字符串字面常量为左值，且为const左值。
```

Lambdas

C++11 introduced *lambdas*, allowing the definition of **inline** functionality, which can be used as a parameter or a local object. **Lambdas change the way the C++ standard library is used.**

A lambda is a definition of functionality that can be defined inside statements and expressions. Thus, you can use a lambda as an **inline function**. The minimal lambda function has no parameters and simply does something:

```
[] {
   std::cout << "hello lambda" << std::endl;
}
```

You can call it directly:

$$[...]\ (...)\ \text{mutable}_{opt}\ \ throwSpec_{opt}\ \ \text{->}\ \ retType_{opt}\ \{...\}$$

```
[] {
   std::cout << "hello lambda" << std::endl;
} 0;    // prints "hello lambda"
```

or pass it to objects to get called:

```
auto l = [] {
   std::cout << "hello lambda" << std::endl;
};
...
l0;    // prints "hello lambda"
```

[introducer](optional)mutable throwSpec->retType {}
mutable决定[]能够被改写 mutable throwSpec retType都是可选的,只要有一个存在就得写()
retType 返回类型
()放参数
[]放外面变量 passed by value or reference

Lambdas

lambda introducer

All of them are optional,
but if one of them occurs,
the parentheses for the parameters are mandatory. 强制的

$$[\ldots]\ (\ldots)\ \text{mutable}_{opt}\ \ \textit{throwSpec}_{opt}\ \ {->}\textit{retType}_{opt}\ \{\ldots\}$$

capture to access nonstatic outside objects inside the lambda. Static objects such as std::cout can be used.

you can specify a *capture* to access data of outer scope that is not passed as an argument:

objects are passed by value, but inside the function object defined by the lambda, you have write access to the passed value.

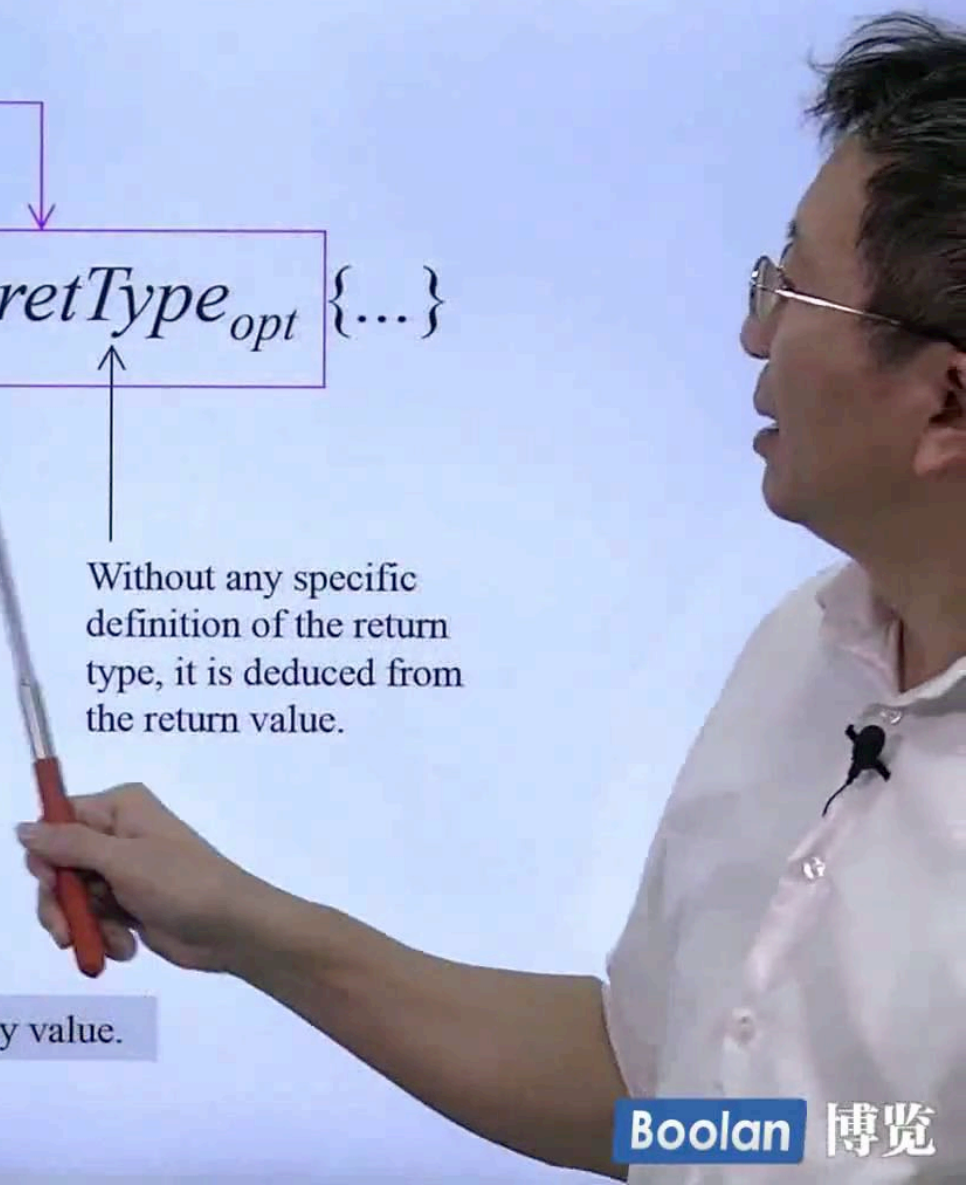
Without any specific definition of the return type, it is deduced from the return value.

- [=] means that the outer scope is passed to the lambda by value.
- [&] means that the outer scope is passed to the lambda by reference.

Ex:

```
int x=0;
int y=42;
auto qqq = [x, &y] { ... };
```

[=, &y] to pass y by reference and all other objects by value.



Lambdas

objects are passed by value, but inside the function object defined by the lambda, you have write access to the passed value.

```
int id = 0;
auto f = [id] () mutable {
    std::cout << "id: " << id << std::endl;
    ++id; // OK
};

id = 42;
f();
f();
f();
std::cout << id << std::endl;
```

如果沒寫 mutable，
難道不能 ++id; 嗎？

The type of a lambda is an anonymous function object (or functor)

```
class Functor {
  private:
    int id;     // copy of outside id
  public:
    void operator() () {
        std::cout << "id: " << id << std::endl;
        ++id;   // OK
    }
};
Functor f;
```

```
id: 0
id: 1
id: 2
42
```

Lambdas

傳值

```
int id = 0;
auto f = [id] () mutable {
    cout << "id: " << id << endl;
    ++id; // OK
};

id = 42;
f();
f();
f();
std::cout << id << std::endl;
```

```
id: 0
id: 1
id: 2
42
```

傳引用

```
int id = 0;
auto f = [&id] (int param) {
    cout << "id: " << id << endl;
    ++id; ++param; // OK
};

id = 42;
f(7);
f(7);
f(7);
std::cout << id << std::endl;
```

```
id: 42
id: 43
id: 44
45
```

```
//[Error] increment of
read-only variable 'id'
int id = 0;
auto f = [id] () {
    cout << "id: " << id << endl;
    ++id; //
};

id = 42;
f();
f();
f();
std::cout << id << std::endl;
```

```
auto f = [id] () mutable {
    cout << "id: " << id << endl;
    ++id;
    static int x =5;
    int y = 6;
    return id;
};
```

可以聲明變量
可以返回數值

Lambdas

Here is what compiler generates for lambda's:

```
int tobefound = 5;
auto lambda1 = [tobefound](int val) {return val == tobefound;};

class UnNamedLocalFunction
{
    int localVar;
public:
    UnNamedLocalFunction(int var) : localVar(var) {}
    bool operator()(int val)
    {   return val == localVar;   }
};

UnNamedLocalFunction lambda2(tobefound);

bool b1 = lambda1(5);
bool b2 = lambda2(5);
```

Lambdas

The type of a lambda is an anonymous function object (or functor) that is unique for each lambda expression. Thus, to declare objects of that type, you need templates or auto. If you need the type, you can use decltype(), which is, for example, required to pass a lambda as hash function or ordering or sorting criterion to associative or unordered containers.

```
auto cmp = [] (const Person& p1, const Person& p2) {
               return p1.lastname()<p2.lastname() ||
                      (p1.lastname()==p2.lastname()&&
                       p1.firstname()<p2.firstname());
           };
...
std::set<Person,decltype(cmp)> coll(cmp);
```

Because you need the type of the lambda for the declaration of the set, decltype must be used, which yields the type of a lambda object, such as **cmp**. Note that you also have to pass the lambda object to the constructor of coll; otherwise, coll would call the default constructor for the sorting criterion passed, and by rule lambdas have no default constructor and no assignment operator. So, for a sorting criterion, a class defining the function objects might still be more intuitive.

因此如果被調用, 編譯器報錯:
use of deleted function …

```
template <class Key,
          class Compare = less<Key>,
          class Alloc = alloc>
class set {
public:
  // typedefs:
  ...
  typedef Compare key_compare;
  typedef Compare value_compare;
private:
  typedef rb_tree<key_type, value_type,
                  identity<value_type>,
                  key_compare, Alloc> rep_type;
  rep_type t;  // red-black tree representing set
public:
  ...
  set() : t(Compare()) {}
  explicit set(const Compare& comp) : t(co
```

如果后面参数没有cmp 则调用默认构造函数
会调用lambda的默认构造函数 而lambda没有默认构造函数和赋值操作
故需要传入cmp参数

Lambdas

Function objects are a very powerful way to customize the behavior of Standard Template Library (STL) algorithms, and can encapsulate both code and data (unlike plain functions). But function objects are inconvenient to define because of the need to write entire classes. Moreover, they are not defined in the place in your source code where you're trying to use them, and the **non-locality** makes them more difficult to use. Libraries have attempted to mitigate some of the problems of **verbosity** and **non-locality**, but don't offer much help because the syntax becomes complicated and the compiler errors are not very friendly. Using function objects from libraries is also less efficient since the function objects defined as data members are not in-lined.

Lambda expressions address these problems. The following code snippet shows a lambda expression used in a program to remove integers between variables x and y from a vector of integers.

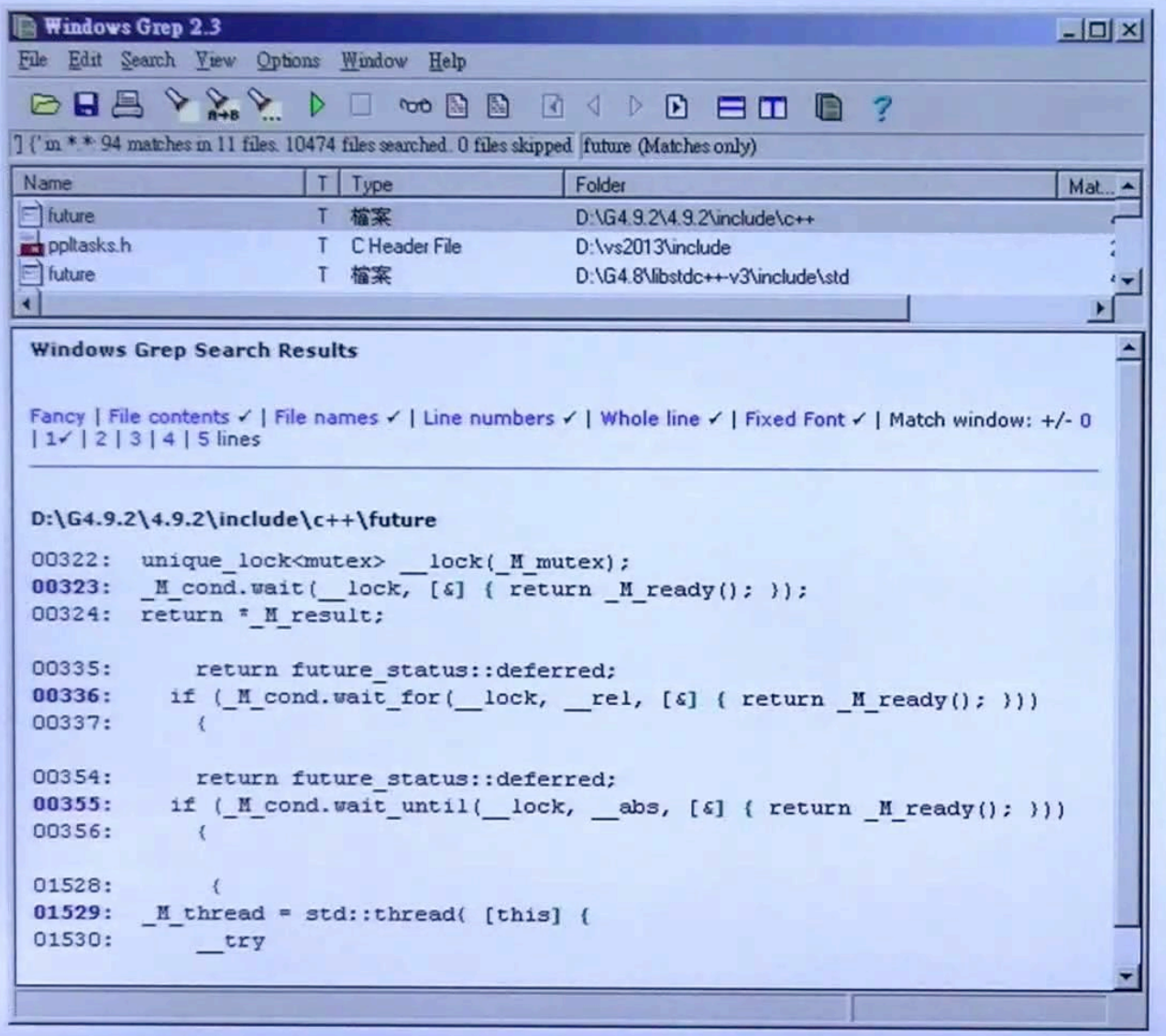
```
vector<int> vi { 5,28,50,83,70,590,245,59,24};
int x = 30;
int y = 100;
vi.erase( remove_if(vi.begin(),
                    vi.end(),
                    [x, y](int n) { return x < n && n < y; }
                   ),
          vi.end()
        );
for (auto i : vi)
    cout << i << ' ';      //5 28 590 245 24
cout << endl;
```

```
class LambdaFunctor {
public:
   LambdaFunctor(int a, int b) : m_a(a), m_b(b) { }
   bool operator()(int n) const {
       return m_a < n && n < m_b; }
private:
   int m_a;
   int m_b;
};

v.erase( remove_if(v.begin(), v.end(),
                   LambdaFunctor(x, y) ),
         v.end()
       );
```

效率

用仿函数更加安全 但非本地,用起来也麻烦，需要去看怎样使用，另外编译出错的信息也不友好，而且它们不是inline的，效率会低一些

Lambdas



第一顆震撼彈 Variadic Templates

```
839  void printX()
840  {
841  }
842
843  template <typename T, typename... Types>
844  void printX(const T& firstArg, const Types&... args)
845  {
846      cout << firstArg << endl;    // print first argument
847      printX(args...);             // call print() for remaining arguments
848  }
```

```
40   int __cdecl printf (
41           const char *format,
42           ...
43           )
44   /*
45    * stdout 'PRINT', 'F'ormatted
46    */
47   {
48           va_list arglist;
49           int buffing;
50           int retval;
51
52           va_start(arglist, format);
53
54           _ASSERTE(format != NULL);
55
56           _lock_str2(1, stdout);
57
58           buffing = _stbuf(stdout);
59
60           retval = _output(stdout,format,arglist);
61
62           _ftbuf(buffing, stdout);
63
64           _unlock_str2(1, stdout);
65
66           return(retval);
67   }
```

Variadic Templates

數量不定的模板參數

```
1 void print ()
  {
  }

2 template <typename T, typename... Types>
  void print (const T& firstArg, const Types&... args)
  {
     cout << firstArg << endl;       // print first argument
     print(args...);                 // call print() for remaining arguments
  }
```

```
print(7.5, "hello", bitset<16>(377), 42);
```

```
7.5
hello
0000000101111001
42
```

Inside variadic templates, **sizeof...(args)** yields the number of arguments

```
3 template <typename... Types>
  void print (const Types&... args)
  { /* … */ }
```

❷和❸可以並存嗎？
若可，誰較泛化？誰較特化？

… 就是一個所謂的 **pack** (包)
用於 template parameters, 就是 template parameters **pack** (模板參數包)
用於 function parameter types, 就是 function parameter types **pack** (函數參數類型包)
用於 function parameters, 就是 function parameters **pack** (函數參數包)

Variadic Templates

可以很方便地完成 recursive function call.

```
class CustomerHash {
public:
  std::size_t operator() (const Customer& c) const {
    return hash_val(c.fname, c.lname, c.no);
  }
};
```

```
template <typename T, typename... Types>
inline void hash_val (size_t& seed,
                      const T& val, const Types&... args) {
    hash_combine(seed, val);
    hash_val(seed, args...);
}

// auxiliary generic function
template <typename... Types>
inline size_t hash_val(const Types&... args) {
    size_t seed = 0;
    hash_val (seed, args...);
    return seed;
}
```

recursive

```
#include <functional>
template <typename T>
inline void hash_combine(size_t& seed, const T& val) {
    seed ^= std::hash<T>()(val) + 0x9e3779b9
      + (seed<<6) + (seed>>2);
}
```

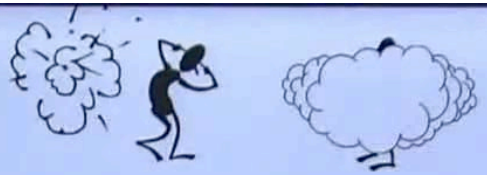
```
// auxiliary generic functions
template <typename T>
inline void hash_val(size_t& seed, const T& val) {
    hash_combine(seed, val);
}
```

last

Boolan 博览

Variadic Templates

重回核爆區



- 談的是 template
  - function template
  - class template
- 變化的是 template parameters
  - 參數個數 (variable number) ..... 利用參數個數逐一遞減的特性，實現遞歸函數調用. 使用 function template 完成.
  - 參數類型 (different type) ..... 利用 參數個數逐一遞減 導致 參數類型也逐一遞減 的特性，實現遞歸繼承或遞歸複合，以 class template 完成.

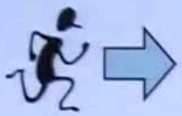
```
void func() { /* … */ }
```

```
template <typename T, typename... Types>
void func(const T& firstArg, const Types&... args)
{
    處理 firstArg
    func(args...);
}
```

以下介紹七大例

Variadic Templates 例 1

Since C++11, templates can have parameters that accept a variable number of template arguments. This ability is called **variadic templates**. For example, you can use the following to call printX() for a variable number of arguments of different types:



```
print(7.5, "hello", bitset<16>(377), 42);
```

```
7.5
hello
0000000101111001
42
```

```
void printX()
{
}

template <typename T, typename... Types>
void printX(const T& firstArg, const Types&... args)
{
    cout << firstArg << endl;       // print first argument
    printX(args...);                // call printX() for remaining arguments
}
```

Inside variadic templates, **sizeof...(args)** yields the number of arguments

❸
```
template <typename... Types>
void printX(const Types&... args)
{ /* ... */ }
```

❶和❸可以並存嗎？
若可，誰較泛化？誰較特化？
答案是❶比較特化.

特化的会被调用
所以不会冲突

If one or more arguments are passed, the function template is used, which by specifying the first argument separately allows the first argument to print and then **recursively** calls printX() for the remaining arguments. **To end the recursion**, the non-template overload of printX() is provided.

Variadic Templates 例 2

使用 variadic templates 重寫 printf()

```
//http://stackoverflow.com/questions/3634379/variadic-templates
void printf(const char *s)
{
  while (*s)
  {
    if (*s == '%' && *(++s) != '%')
        throw std::runtime_error("invalid format string: missing arguments");
    std::cout << *s++;
  }
}
```

```
template<typename T, typename... Args>
void printf(const char* s, T value, Args... args)
{
  while (*s)  {
    if (*s == '%' && *(++s) != '%')  {
      std::cout << value;
      printf(++s, args...); // call even when *s == 0 to detect extra argun
      return;
    }
    std::cout << *s++;
  }
  throw std::logic_error("extra arguments provided to printf")
}
```

```
int* pi = new int;
printf("%d %s %p %f \n",
       15,
       "This is Ace.",
       pi,
       3.141592653);
```

15 This is Ace. 0x3e4ab8 3.14159

20151101 侯捷 :
原版本是 printf(s, args...);
會把 specifier 印出, 不佳.

Variadic Templates

例 3

```
// ...\4.9.2\include\c++\bits\stl_algo.h
template<typename _ForwardIterator,
         typename _Compare>
  _ForwardIterator
  __max_element(_ForwardIterator __first,
                _ForwardIterator __last,
                _Compare __comp)
  {
    if (__first == __last) return __first;
    _ForwardIterator __result = __first;
    while (++__first != __last)
      if (__comp(__result, __first))
        __result = __first;
    return __result;
  }

template<typename _ForwardIterator>
  inline _ForwardIterator
  max_element(_ForwardIterator __first,
              _ForwardIterator __last)
  {
    return __max_element(__first, __last,
                         __iter_less_iter());
  }
```

若參數 types 皆同, 實無需動用 variadic templates, 使用 **initializer_list<T>** 足矣.

```
cout << max( { 57, 48, 60, 100, 20, 18 } ) << endl;
```

100

```
// ...\4.9.2\include\c++\bits\stl_algo.h
  template<typename _Tp>
    inline _Tp
    max(initializer_list<_Tp> __l)
    { return *max_element(__l.begin(),
                          __l.end()); }
```

```
// ...\4.9.2\include\c++\bits\predefined_oops.h
  inline _Iter_less_iter
  __iter_less_iter()
  { return _Iter_less_iter(); }
```

```
// ...\4.9.2\include\c++\bits\predefined_oops.h
  struct _Iter_less_iter
  {
    template<typename _Iterator1,
             typename _Iterator2>
      bool
      operator()(_Iterator1 __it1,
                 _Iterator2 __it2) const
      { return *__it1 < *__it2; }
  };
```

Variadic Templates 例 4

若參數 type 皆同, 實無需動用 variadic templates

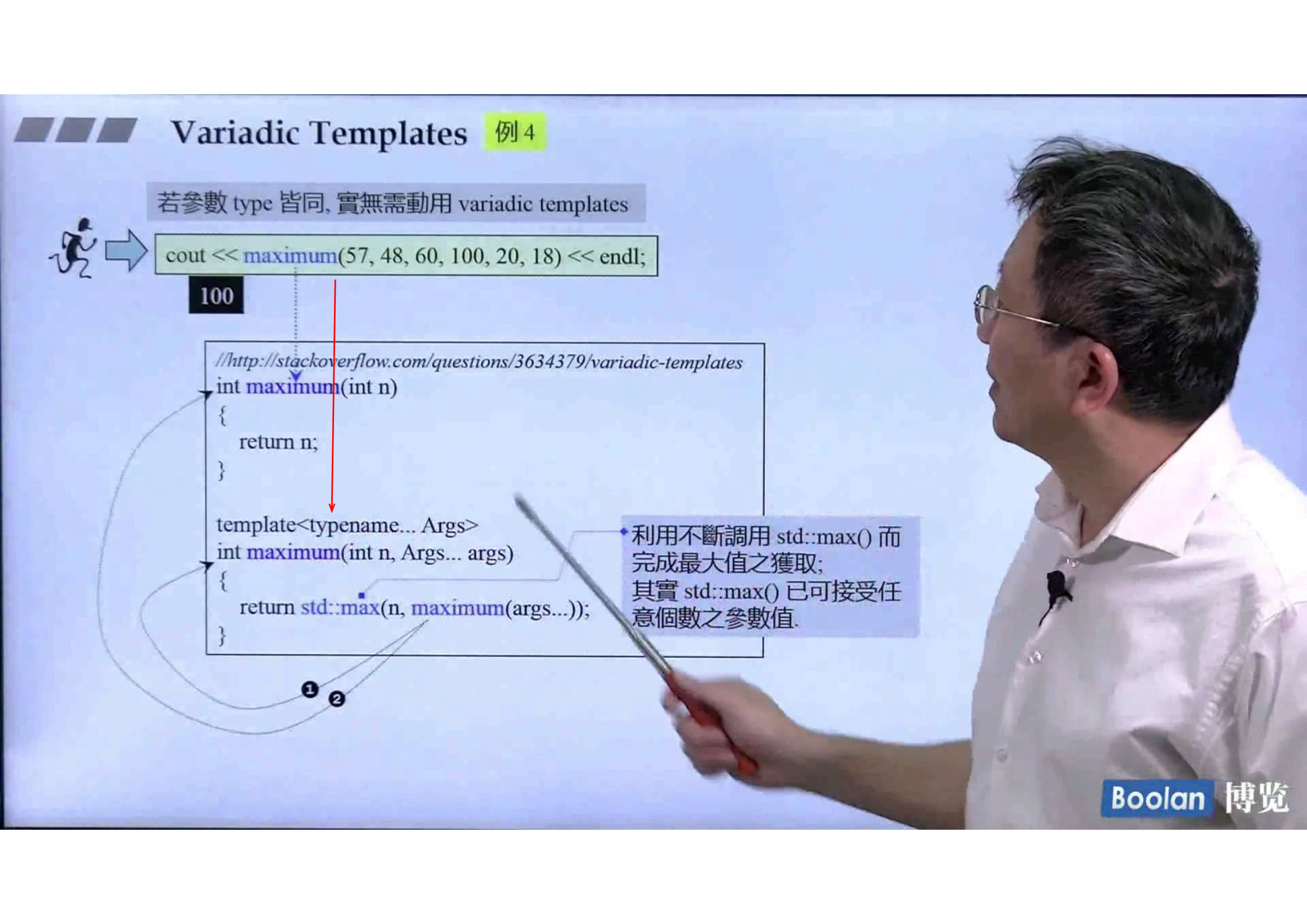
```
cout << maximum(57, 48, 60, 100, 20, 18) << endl;
```

100

```
//http://stackoverflow.com/questions/3634379/variadic-templates
int maximum(int n)
{
    return n;
}

template<typename... Args>
int maximum(int n, Args... args)
{
    return std::max(n, maximum(args...));
}
```

❶ ❷

利用不斷調用 std::max() 而完成最大值之獲取;
其實 std::max() 已可接受任意個數之參數值.



Variadic Templates 例 5

以異於一般的方式處理 first 元素和 last 元素

递归创建

```
// output operator for tuples
template <typename... Args>
ostream& operator<<(ostream& os, const tuple<Args...>& t) {
    os << "[";
    PRINT_TUPLE<0, sizeof...(Args), Args...>::print(os,t);
    return os << "]";
}
```

那就必須知道現在處理的元素的 index.
本例以 sizeof…() 獲得元素個數,
以 get<index>(t) 取出元素並將 index++,
…

```
// boost:  util/printtuple.hpp
// helper: print element with index IDX of tuple
//         with MAX elements
template <int IDX, int MAX, typename... Args>
struct PRINT_TUPLE {
    static void print (ostream& os, const tuple<Args...>& t) {
        os << get<IDX>(t) << (IDX+1==MAX ? "" : ",");
        PRINT_TUPLE<IDX+1, MAX, Args...>::print(os,t);
    }
};
```

❶

```
// partial specialization to end the recursion
template <int MAX, typename... Args>
struct PRINT_TUPLE<MAX, MAX, Args...> {
    static void print (std::ostream& os, const tuple<Args...>& t) {
    }
};
```

❷

```
cout << make_tuple(7.5, string("hello"), bitset<16>(377), 42);
//provided there is a overloaded operator<< for bitset.
```

```
[7.5,hello,0000000101111001,42]
```

头尾不同

Variadic Templates

例 6 用於遞歸繼承, recursive inheritance

```
template<typename... Values> class tuple;
template<> class tuple<> { };

template<typename Head, typename... Tail>
class tuple<Head, Tail...>
 : private tuple<Tail...>
{
   typedef tuple<Tail...> inherited;
 public:
   tuple() { }
   tuple(Head v, Tail... vtail)
     : m_head(v), inherited(vtail...) { }

  typename Head::type head() { return m_head; }
  inherited& tail() { return *this; }
protected:
  Head m_head;
};
```

遞歸調用 處理的都是參數,
使用 function template

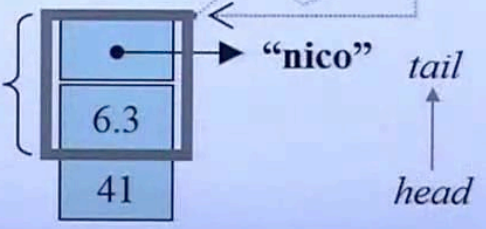
遞歸繼承 處理的是類型 (type),
使用 class template

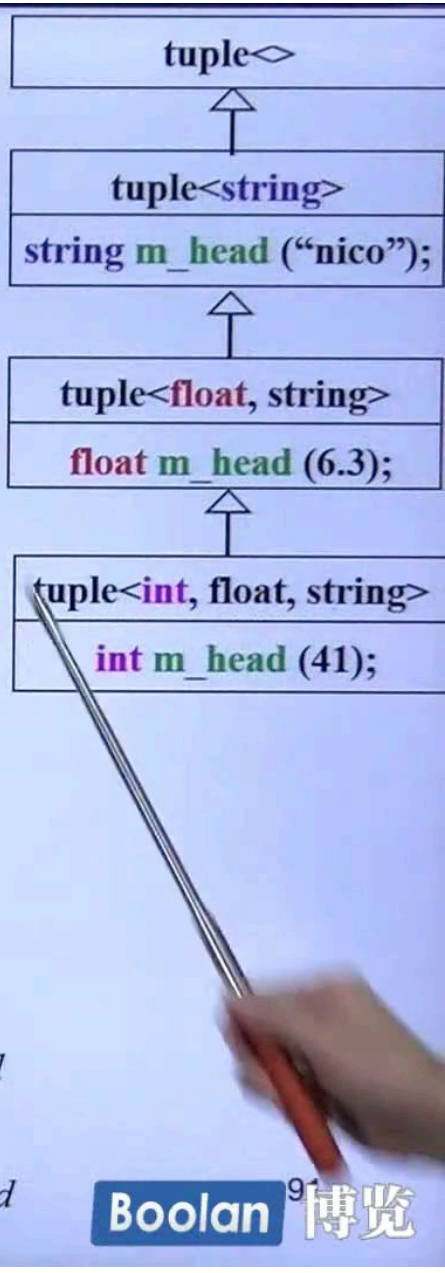
調用 base ctor 並予參數
(不要誤以為是創建 temp object)

注意這是
initialization list

return 後
轉型為 inherited,
獲得的是

例：tuple<int,float,string>
t(41, 6.3, "nico");
t.head() ➔ 獲得 41
t.tail() ➔獲得
t.tail().head()➔獲得 6.3
&(t.tail()) ➔





Variadic Templates 例 6 用於遞歸繼承, recursive inheritance

```
template<typename... Values> class tuple;
template<> class tuple<> { };
```

```
template<typename Head, typename... Tail>
class tuple<Head, Tail...>
 : private tuple<Tail...>
{
   typedef tuple<Tail...> inherited;
 public:
   tuple() { }
   tuple(Head v, Tail... vtail)
     : m_head(v), inherited(vtail...) { }

   typename Head::type head() { return m_head; }
   inherited& tail() { return *this; }
 protected:
   Head m_head;
};
```

遞歸調用 處理的
使用 function

遞歸繼承 處理的
使用 class templ

參數
temp object)

注意這是
initialization list

例：tuple<i
t(41, 6.3
t.head()
t.tail()
t.tail()
&(t.ta

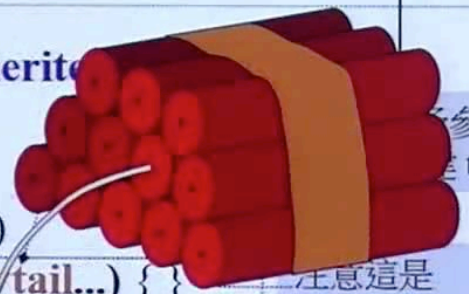
return 後
轉型為 inherited,
獲得的是

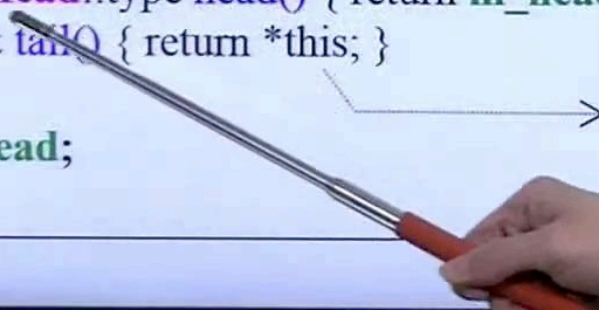
tuple<>

tuple<string>
string m_head ("nico");

tuple<float, string>
float m_head (6.3);

<int, float, string>
(41);





Variadic Templates

```
template<typename... Values> class tuple;
template<> class tuple<> { };
```

```
template<typename Head, typename... Tail>
class tuple<Head, Tail...>
 : private tuple<Tail...>
{
   typedef tuple<Tail...> inherited;
protected:
  Head m_head;
 public:
   tuple() { }
   tuple(Head v, Tail... vtail)
     : m_head(v), inherited(vtail...) { }

  auto head()->decltype(m_head) { return m_head; }
  inherited& tail() { return *this; }
};
```

如果沒移上來，
編譯器會說不認得 m_head.
這太離譜了吧!!

```
tuple<int, float, string> t(41, 6.3, "nico");

cout << sizeof(t) << endl;  //12

cout << t.head() << endl;
cout << t.tail().head() << endl;
cout << t.tail().tail().head() << endl;
```

```
41
6.3
nico
```

Variadic Templates

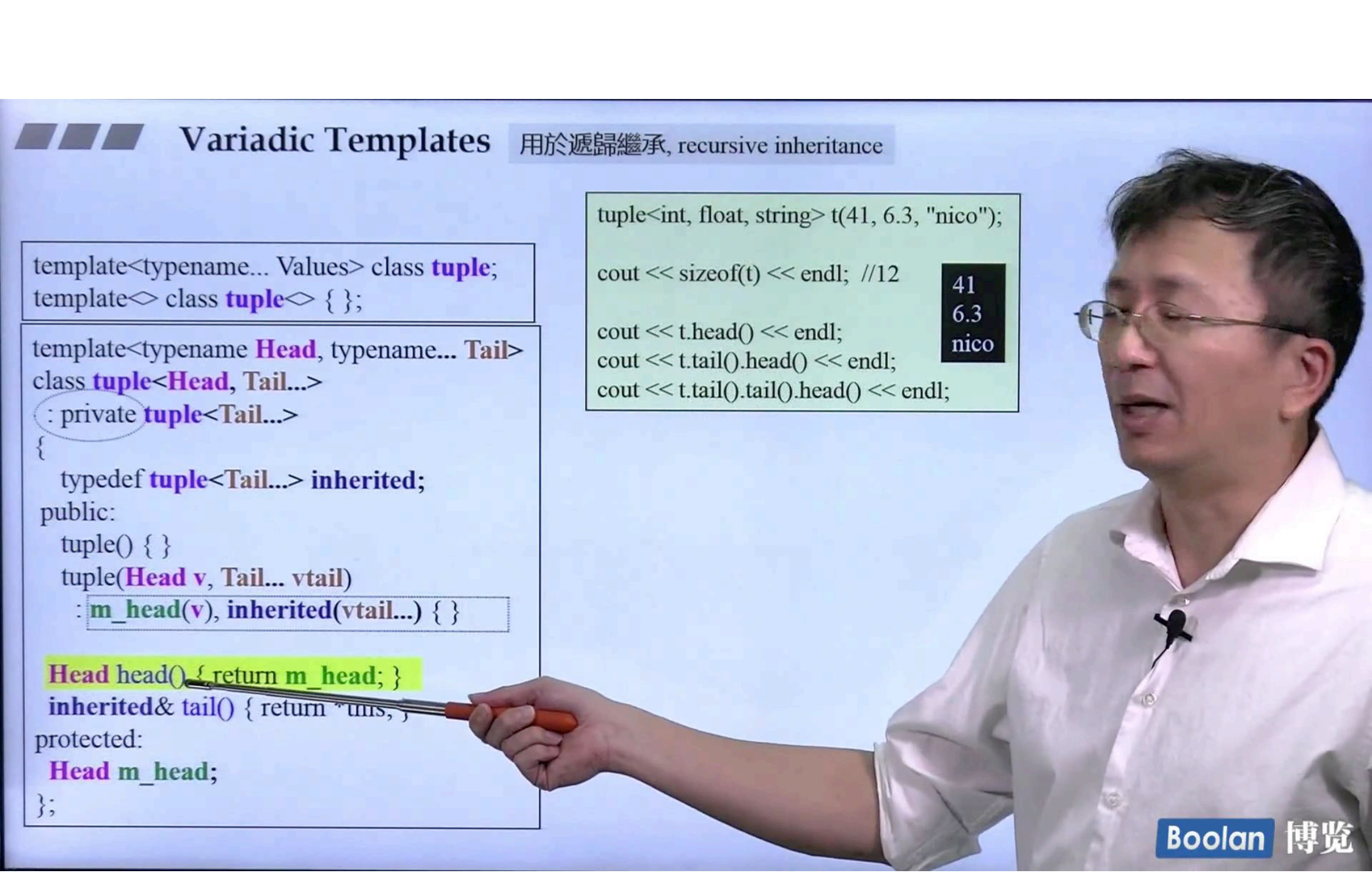
用於遞歸繼承, recursive inheritance

```
template<typename... Values> class tuple;
template<> class tuple<> { };
```

```
template<typename Head, typename... Tail>
class tuple<Head, Tail...>
 : private tuple<Tail...>
{
   typedef tuple<Tail...> inherited;
 public:
   tuple() { }
   tuple(Head v, Tail... vtail)
     : m_head(v), inherited(vtail...) { }

  Head head() { return m_head; }
  inherited& tail() { return *this; }
protected:
  Head m_head;
};
```

```
tuple<int, float, string> t(41, 6.3, "nico");

cout << sizeof(t) << endl;  //12

cout << t.head() << endl;
cout << t.tail().head() << endl;
cout << t.tail().tail().head() << endl;
```

```
41
6.3
nico
```



Variadic Templates

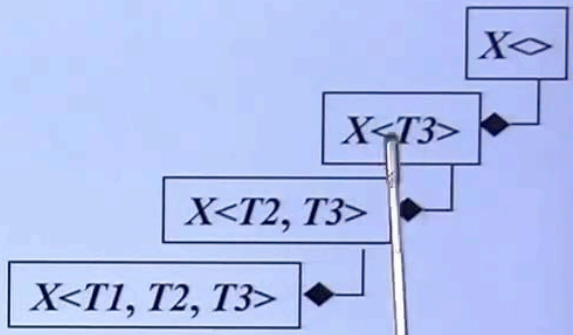
例 7 用於遞歸複合, recursive composition

```
template<typename... Values> class tup;
template<> class tup<> { };
```

遞歸調用 處理的都是參數，
使用 function template

```
template<typename Head, typename... Tail>
class tup<Head, Tail...>
{
  typedef tup<Tail...> composited;
protected:
        composited m_tail;
        Head m_head;
public:
  tup() { }
  tup(Head v, Tail... vtail)
    : m_tail(vtail...), m_head(v) { }

  Head head() { return m_head; }
  composited& tail() { return m_tail; }
};
```

遞歸複合 處理的是類型 (type)，
使用 class template




```
tup<int, float, string> it1(41, 6.3, "nico");
cout << sizeof(it1) << endl;
cout << it1.head() << endl;
cout << it1.tail().head() << endl;
cout << it1.tail().tail().head() << endl;

tup<string, complex<int>, bitset<16>, double>
      it2("Ace", complex<int>(3,8), bitset<16>(377), 3.141592653);
cout << sizeof(it2) << endl;
cout << it2.head() << endl;
cout << it2.tail().head() << endl;
cout << it2.tail().tail().head() << endl;
cout << it2.tail().tail().tail().head() << endl;
```

```
16
41
6.3
nico
40
Ace
(3,8)
0000000101111001
3.14159
```

C++ keywords

http://en.cppreference.com/w/cpp/keyword

This is a list of reserved keywords in C++. Since they are used by the language, these keywords are not available for re-definition or overloading.

alignas (since C++11)	else	requires (concepts TS)
alignof (since C++11)	enum	return
and	explicit	short
and_eq	export(1)	signed
asm	extern	sizeof
auto(1)	false	static
bitand	float	static_assert (since C++11)
bitor	for	static_cast
bool	friend	struct
break	goto	switch
case	if	template
catch	inline	this
char	int	thread_local (since C++11)
char16_t (since C++11)	long	throw
char32_t (since C++11)	mutable	true
class	namespace	try
compl	new	typedef
concept (concepts TS)	noexcept (since C++11)	typeid
const	not	typename
constexpr (since C++11)	not_eq	union
const_cast	nullptr (since C++11)	unsigned
continue	operator	using(1)
decltype (since C++11)	or	virtual
default(1)	or_eq	void
delete(1)	private	volatile
do	protected	wchar_t
double	public	while
dynamic_cast	register	xor
	reinterpret_cast	xor_eq

- (1) - meaning changed in C++11

Rvalue references

Rvalue references are a new reference type introduced in C++0x that help solve the problem of ***unnecessary copying*** and enable ***perfect forwarding***. When the right-hand side of an assignment is an rvalue, then the left-hand side object can steal resources from the right-hand side object rather than performing a separate allocation, thus enabling ***move semantics***.

C++ with its user-defined types has introduced some subtleties regarding modifiability and assignability that cause this definition to be incorrect

Lvalue : 可以出現於 operator= 左側者
Rvalue : 只能出現於 operator= 右側者

以 int 試驗：

```
int a = 9;
int b = 4;

a = b;     // ok
b = a;     // ok
a = a + b; // ok

a + b = 42;
```

← [Error] lvalue required as left operand of assignment

以 string 試驗：

```
string s1("Hello ");
string s2("World");
s1 + s2 = s2;          //竟然通過編譯
cout << "s1: " << s1 << endl;          //s1: Hello
cout << "s2: " << s2 << endl;          //s2: World
string() = "World"; //竟然可以對 temp obj 賦值
```

以 complex 試驗：

```
complex<int> c1(3,8), c2(1,0);
c1 + c2 = complex<int>(4,9);          //c1+c2 可以當做 Lvalue 嗎？
cout << "c1: " << c1 << endl;          //c1: (3,8)
cout << "c2: " << c2 << endl;          //c2: (1,0)
complex<int>() = complex<int>(4,9); //竟然可以對 temp obj 賦值
```

临时对象是一种右值
不放在等号左边

Rvalue references

Rvalue : 只能出現於 operator= 右側者

```
int foo() { return 5; }
…
int x = foo();          // ok
int* p = &foo();        //[Error]
foo() = 7;              //[Error]
```

對著 Rvalue 取其 reference, 不可以.
沒有所謂 Rvalue reference (before C++0x)

當 Rvalue 出現於 operator= (copy assignment) 的右側, 我們認為對其資源進行偷取/搬移(move) 而非拷貝 (copy) 是可以的, 是合理的。
那麼：

1, 必須有語法讓我們在調用端告訴編譯器 “這是個 Rvalue”。
2, 必須有語法讓我們在被調用端寫出一個專門處理 Rvalue 的所謂 move assignment 函數。

Rvalue references and Move Semantics

```
class MyString {
private:
   char* _data;
   ...
public:
   //copy ctor
   MyString(const MyString& str)
     : initialization list {
        ...
   }

   move aware
   //move ctor
   MyString(MyString&& str) noexcept
     : initialization list  {
        ...
   }
...
```

❶ //copy
insert(…,&x)

❷ //move
insert(…,&&x)

temp object
一定被當作 Rvalue

You need to inform C++ (*specifically* **std::vector**) that your *move constructor and destructor* does not throw. Then the *move constructor* will be called when the vector grows.

```
template<typename M>
void test_moveable(M c, long& value)
{
char buf[10];

typedef typename
        iterator_traits<typename M::iterator>::value_type Vtype;
clock_t timeStart = clock();
   for (long i=0; i< value; ++i) {
      snprintf(buf, 10, "%d", rand()); //隨機數 (轉為字符串)
      auto ite = c.end();              // 定位於尾端
      c.insert(ite, Vtype(buf));       //插入
   }
   cout << "milli-seconds : " << (clock()-timeStart) << endl;
   cout << "size()= " << c.size() << endl;
   output_static_data(*(c.begin()));  //copy/move ctors 被調用次數

M c1(c);
M c2(std::move(c1));
   c1.swap(c2);
}
```

右值

必須確保後續不再使用 c1

左值也可以
用std::move
前提是左值之后不再用

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

Rvalue references and Move Semantics

G2.9

```
157  iterator insert(iterator position, const T& x) {
158    size_type n = position - begin();
159    if (finish != end_of_storage && position == end()) {
         construct(finish, x);
         ++finish;
       }
       else
164      insert_aux(position, x);
165    return begin() + n;
166  }
```

G4.9

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

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

move aware

Rvalue references and Move Semantics

```
class MyString {
private:
    char* _data;
    ...
public:
    //copy assignment
    MyString& operator=(const MyString& str) {
        ...
        return *this;
    }
```

move aware

```
    //move assignment
    MyString& operator=(MyString&& str)
        noexcept {
        ...
        return *this;
    }
 ...
```

Perfect Forwarding

完美的转交

Perfect forwarding allows you to write a single function template that takes n arbitrary arguments and *forwards them transparently* to another arbitrary function. The nature of the argument (modifiable, const, lvalue or rvalue) is preserved in this forwarding process.

```
template <typename T1, typename T2>
void functionA(T1&& t1, T2&& t2)
{
    functionB(std::forward<T1>(t1),
              std::forward<T2>(t2));
}
```

Unperfect 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
                     //強制將 a 由 Lvalue 改為 Rvalue
forward(2);  //forward(int&&): 2,  process(int&): 2
        //Rvalue 經由 forward() 傳給另一函數卻變成 Lvalue.
        //(原因是傳遞過程中它變成了一個 named object)
forward(move(a));    //forward(int&&): 0,  process(int&): 0
                     //Rvalue 經由 forward() 傳給另一函數卻變成了 Lvalue
//!    forward(a);     //[Error] cannot bind 'int' lvalue to 'int&&'
       const int& b = 1;
//!    process(b);     //[Error] no matching function for call to 'process(const int&)'
//!    process(move(b));  //[Error] no matching function for call to
                     // 'process(std::remove_reference<const int&>::type)'
//! int& x(5);        //[Error] invalid initialization of non-const reference of
                     //type 'int&' from an rvalue of type 'int'
//! process(move(x)); //上行失敗,本行當然無庸置疑: "x" was not declared in this scope
```

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

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

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

失败

type traits

Perfect Forwarding

...\4.9.2\include\c++\bits\move.h

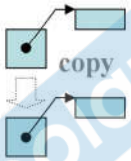
```
68   /**
69    *  @brief  Forward an lvalue.
70    *  @return The parameter cast to the specified type.
71    *
72    *  This function is used to implement "perfect forwarding".
73    */
74   template<typename _Tp>
75     constexpr _Tp&&
76     forward(typename std::remove_reference<_Tp>::type& __t) noexcept
77     { return static_cast<_Tp&&>(__t); }
78
79   /**
80    *  @brief  Forward an rvalue.
81    *  @return The parameter cast to the specified type.
82    *
83    *  This function is used to implement "perfect forwarding".
84    */
85   template<typename _Tp>
86     constexpr _Tp&&
87     forward(typename std::remove_reference<_Tp>::type&& __t) noexcept
88     {
89       static_assert(!std::is_lvalue_reference<_Tp>::value, "template argument"
90             " substituting _Tp is an lvalue reference type");
91       return static_cast<_Tp&&>(__t);
92     }
```

```
94   /**
95    *  @brief  Convert a value to an rvalue.
96    *  @param  __t  A thing of arbitrary type.
97    *  @return The parameter cast to an rvalue-reference to allow moving it.
98    */
99   template<typename _Tp>
100    constexpr typename std::remove_reference<_Tp>::type&&
101    move(_Tp&& __t) noexcept
102    { return static_cast<typename std::remove_reference<_Tp>::type&&>(__t); }
```

完美

寫一個 moveable class

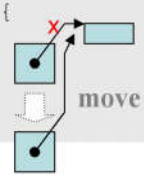
```
class MyString {
public:
    static size_t DCtor;   //累計 default-ctor 呼叫次數
    static size_t Ctor;    //累計 ctor 呼叫次數
    static size_t CCtor;   //累計 copy-ctor 呼叫次數
    static size_t CAsgn;   //累計 copy-asgn 呼叫次數
    static size_t MCtor;   //累計 move-ctor 呼叫次數
    static size_t MAsgn;   //累計 move-asgn 呼叫次數
    static size_t Dtor;    //累計 dtor 呼叫次數
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(NULL),_len(0) { ++DCtor; }

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




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

    //move ctor, with "noexcept"
    MyString(MyString&& str) noexcept
      : _data(str._data), _len(str._len)  {
        ++MCtor;
        str._len = 0;
        str._data = NULL; //避免 delete (in dtor)
    }
    //copy assignment
    MyString& operator=(const MyString& str) {
        ++CAsgn;
        if (this != &str) {
            if (_data) delete _data;
            _len = str._len;
            _init_data(str._data); //COPY!
        }
        else {
        }
        return *this;
    }
```

重要

没有这行临时对象生命结束时会
调用析构函数 使得move的内存释放



浅拷贝

寫一個 moveable class


```
//move assignment
MyString& operator=(MyString&& str) noexcept {
    ++MAsgn;
    if (this != &str) {
        if (_data) delete _data;
        _len = str._len;
        _data = str._data;   //MOVE!
        str._len = 0;
        str._data = NULL;    //避免 delete (in dtor)
    }
    return *this;
}
//dtor
virtual ~MyString() {
    ++Dtor;
    if (_data) {
        delete _data;
    }
}
bool
operator<(const MyString& rhs) const //爲了 set
{
    return std::string(this->_data)
           < std::string(rhs._data);
    //借用現成事實：string 已能比較大小.
}
```

自我賦值檢查

重要

destructor is noexcept by default.

檢查NULL

```
bool
operator==(const MyString& rhs) const //爲了 set
{
    return std::string(this->_data)
           == std::string(rhs._data);
    //借用現成事實：string 已能判斷相等.
}

char* get() const { return _data; }
};
size_t MyString::DCtor=0;
size_t MyString::Ctor=0;
size_t MyString::CCtor=0;
size_t MyString::CAsgn=0;
size_t MyString::MCtor=0;
size_t MyString::MAsgn=0;
size_t MyString::Dtor=0;
```

靜态定义

```
namespace std //必須放在 std 內
{
template<>
struct hash<MyString> { //這是爲了 unordered containers
    size_t
    operator()(const MyString& s) const noexcept
    {  return hash<string>()(string(s.get())); }
     //借用現有的 hash<string>
     // (在 ...\4.9.2\include\c++\bits\basic_string.h)
};
```

測試函數

```
test, with moveable elements
construction, milli-seconds : 8547
size()= 3000000
8MyString --
 CCtor=0 MCtor=7194303 CAsgn=0 MAsgn=
copy, milli-seconds : 3500
move copy, milli-seconds : 0
swap, milli-seconds : 0
```

```
#include <typeinfo>  //typeid()
template<typename T>
void output_static_data(const T& myStr)
{
   cout << typeid(myStr).name() << " -- " << endl;
   cout << " CCtor=" << T::CCtor
        << " MCtor=" << T::MCtor
        << " CAsgn=" << T::CAsgn
        << " MAsgn=" << T::MAsgn
        << " Dtor=" << T::Dtor
        << " Ctor=" << T::Ctor
        << " DCtor=" << T::DCtor
        << endl;
}
```

```
template<typename M, typename NM>
void test_moveable(M c1, NM c2, long& value)
{
char buf[10];

1 //測試 moveable
typedef typename iterator_traits<typename M::iterator>::value_type  V1type;
clock_t timeStart = clock();
   for(long i=0; i< value; ++i) {
      snprintf(buf, 10, "%d", rand()); //隨機數, 放進 buf (轉換爲字符串)
      auto ite = c1.end();             // 定位尾端
      c1.insert(ite, V1type(buf));    //安插於尾端 (對 RB-tree 和 HT 這只是hint)
   }
   cout << "construction, milli-seconds : " << (clock()-timeStart) << endl;
   cout << "size()= " << c1.size() << endl;
   output_static_data(*(c1.begin()));

M c11(c1);                  關於 std::move
M c12(std::move(c1));       必須確保接下來不會再用到 c1
   c11.swap(c12);

2 //測試 non-moveable
…
}
```

```
test_moveable(vector<MyString>(),
              vector<MyStrNoMove>(),
              value);
```

move后来源端作废

moveable 元素對於 vector 速度效能的影響

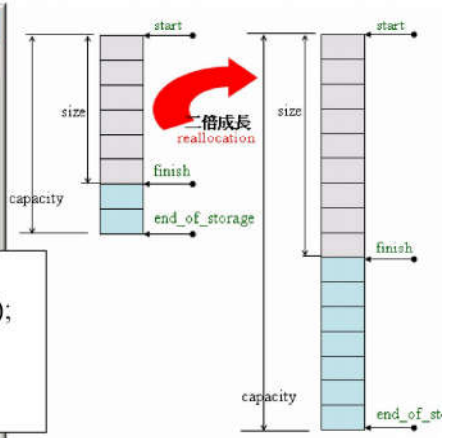
```
D:\handout\C++11-test-DevC++\Test-STL\test-stl.exe

test, with moveable elements
construction, milli-seconds : 8547
size()= 3000000
8MyString --
 CCtor=0 MCtor=7194303 CAsgn=0 MAsgn=0 Dtor=7194309 Ctor=3000006 DCtor=0
copy, milli-seconds : 3500
move copy, milli-seconds : 0
swap, milli-seconds : 0


test, with non-moveable elements
construction, milli-seconds : 14235
size()= 3000000
11MyStrNoMove --
 CCtor=7194303 MCtor=0 CAsgn=0 MAsgn=0 Dtor=7194307 Ctor=3000004 DCtor=0
copy, milli-seconds : 2468
move copy, milli-seconds : 0
swap, milli-seconds : 0
```

差別巨大

差別很大，有時候更是巨大得多

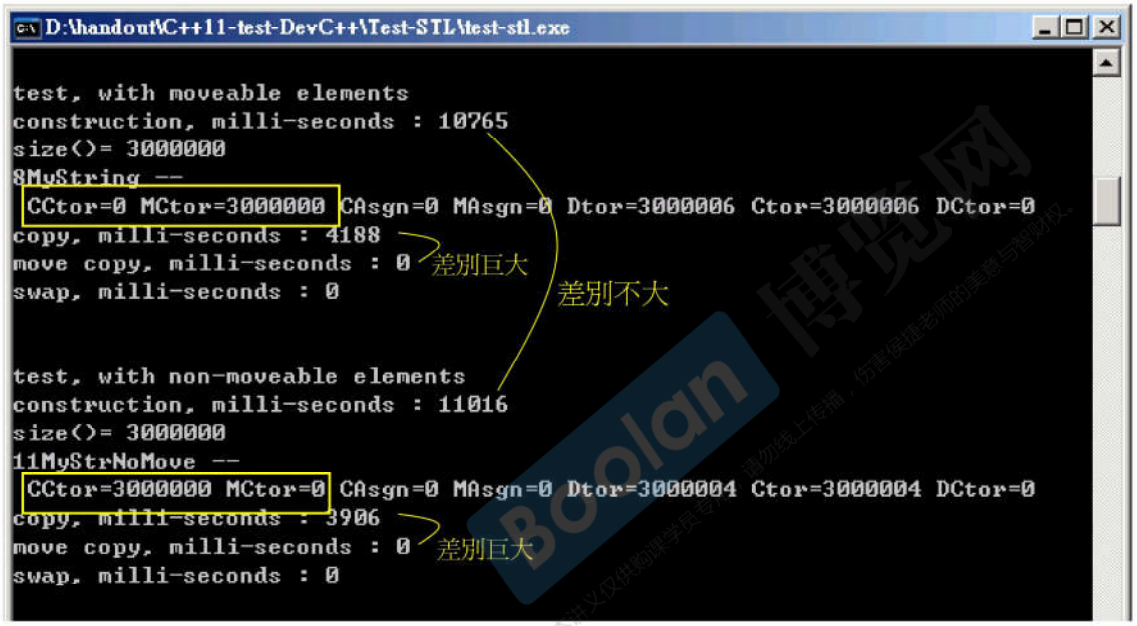
差別巨大

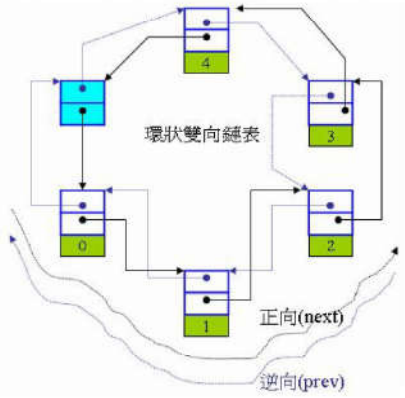
```
for(long i=0; i< value; ++i) {
    snprintf(buf, 10, "%d", rand());
    auto ite = c1.end();
    c1.insert(ite, V1type(buf));
}
```




```
M c1;
...
M c11(c1);
M c12(std::move(c1));
  c11.swap(c12);
```

moveable 元素對於 list 速度效能的影響





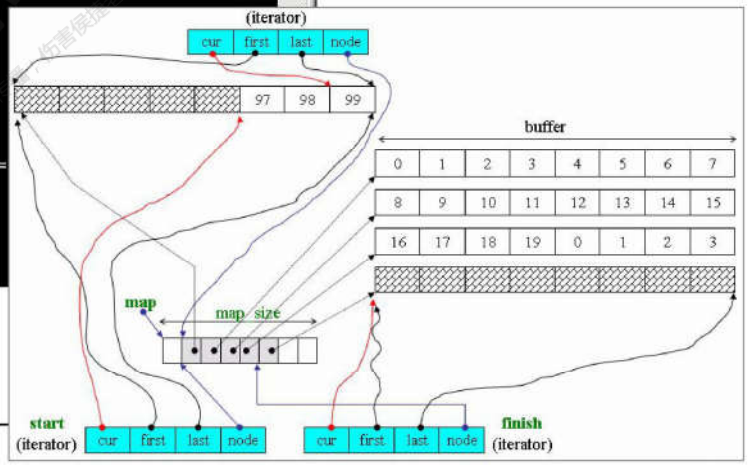
moveable 元素對於 deque 速度效能的影響

```
D:\handout\C++11-test-DevC++\Test-STL\test-stl.exe

test, with moveable elements
construction, milli-seconds : 9078
size()= 3000000
8MyString --
 CCtor=0 MCtor=3000000 CAsgn=0 MAsgn=0 Dtor=3000006 Ctor=3000006 DCtor=0
copy, milli-seconds : 3031
move copy, milli-seconds : 0
swap, milli-seconds : 0


test, with non-moveable elements
construction, milli-seconds : 9844
size()= 3000000
11MyStrNoMove --
 CCtor=3000000 MCtor=0 CAsgn=0 MAsgn=0 Dtor=3000004 Ctor=
copy, milli-seconds : 2437
move copy, milli-seconds : 0
swap, milli-seconds : 0
```

差別巨大

差別不大

差別巨大



— 侯捷 —

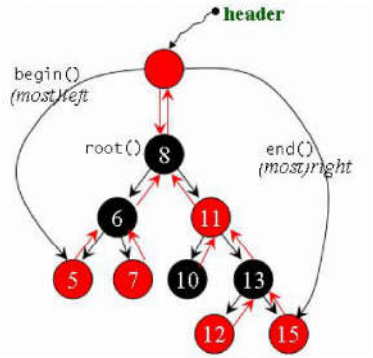
moveable 元素對於 multiset 速度效能的影響

```
D:\handout\C++11-test-DevC++\Test-STL\test-stl.exe

test, with moveable elements
construction, milli-seconds : 74125
size()= 3000000
8MyString --
 CCtor=0 MCtor=3000000 CAsgn=0 MAsgn=0 Dtor=3000006 Ctor=3000006 DCtor=0
copy, milli-seconds : 5438
move copy, milli-seconds : 0
swap, milli-seconds : 0


test, with non-moveable elements
construction, milli-seconds : 74297
size()= 3000000
11MyStrNoMove --
 CCtor=3000000 MCtor=0 CAsgn=0 MAsgn=0 Dtor=3000004 Ctor=3000004 DCtor=0
copy, milli-seconds : 4765
move copy, milli-seconds : 0
swap, milli-seconds : 0
```

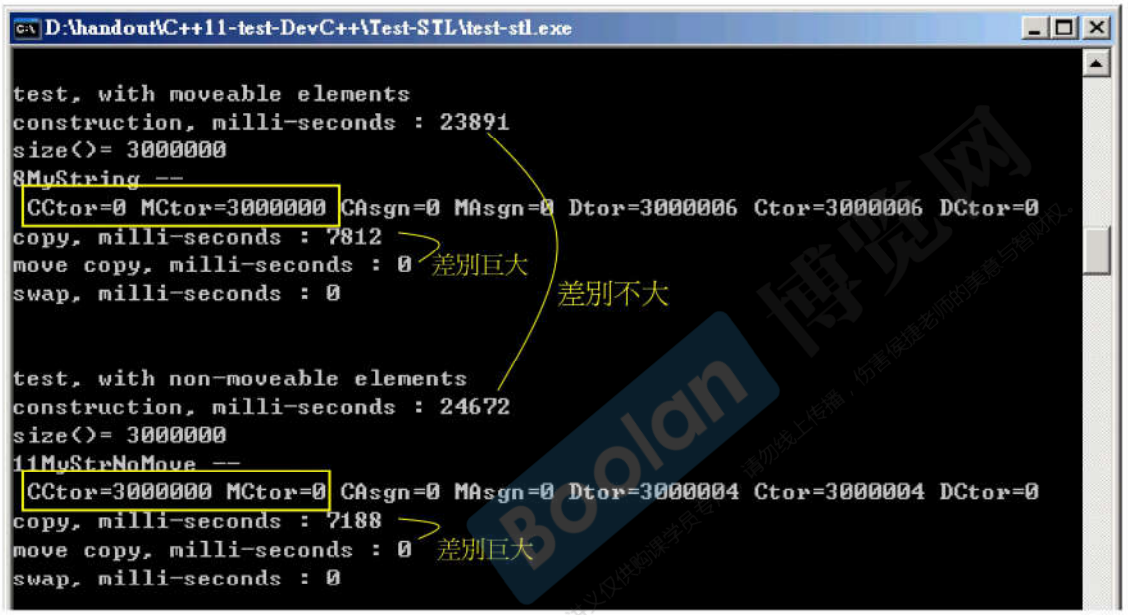
差別巨大

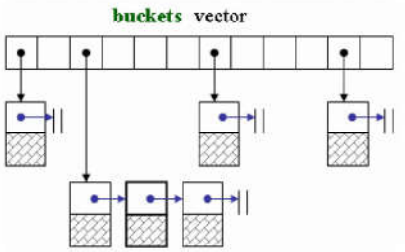
差別不大

差別巨大



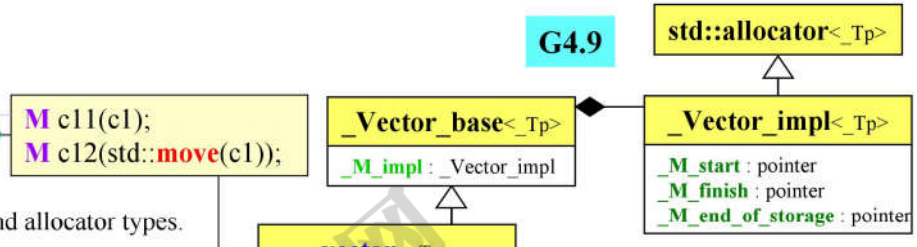
moveable 元素對於 unordered_multiset 速度效能的影響





vector 的 copy ctor

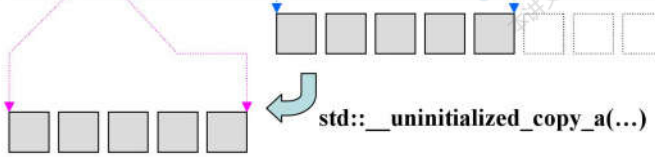
G4.9




```
M c11(c1);
M c12(std::move(c1));
```

```
/**
 *  @brief  %Vector copy constructor.
 *  @param  __x  A %vector of identical element and allocator types.
 *
 *  The newly-created %vector uses a copy of the allocation
 *  object used by @a __x.  All the elements of @a __x are copied,
 *  but any extra memory in
 *  @a __x (for fast expansion) will not be copied.
 */
vector(const vector& __x)
  : _Base(__x.size(),
          _Alloc_traits::_S_select_on_copy(__x._M_get_Tp_allocator()))
{ this->_M_impl._M_finish =
    std::__uninitialized_copy_a(__x.begin(), __x.end(),
                                this->_M_impl._M_start,
                                _M_get_Tp_allocator());
}
```

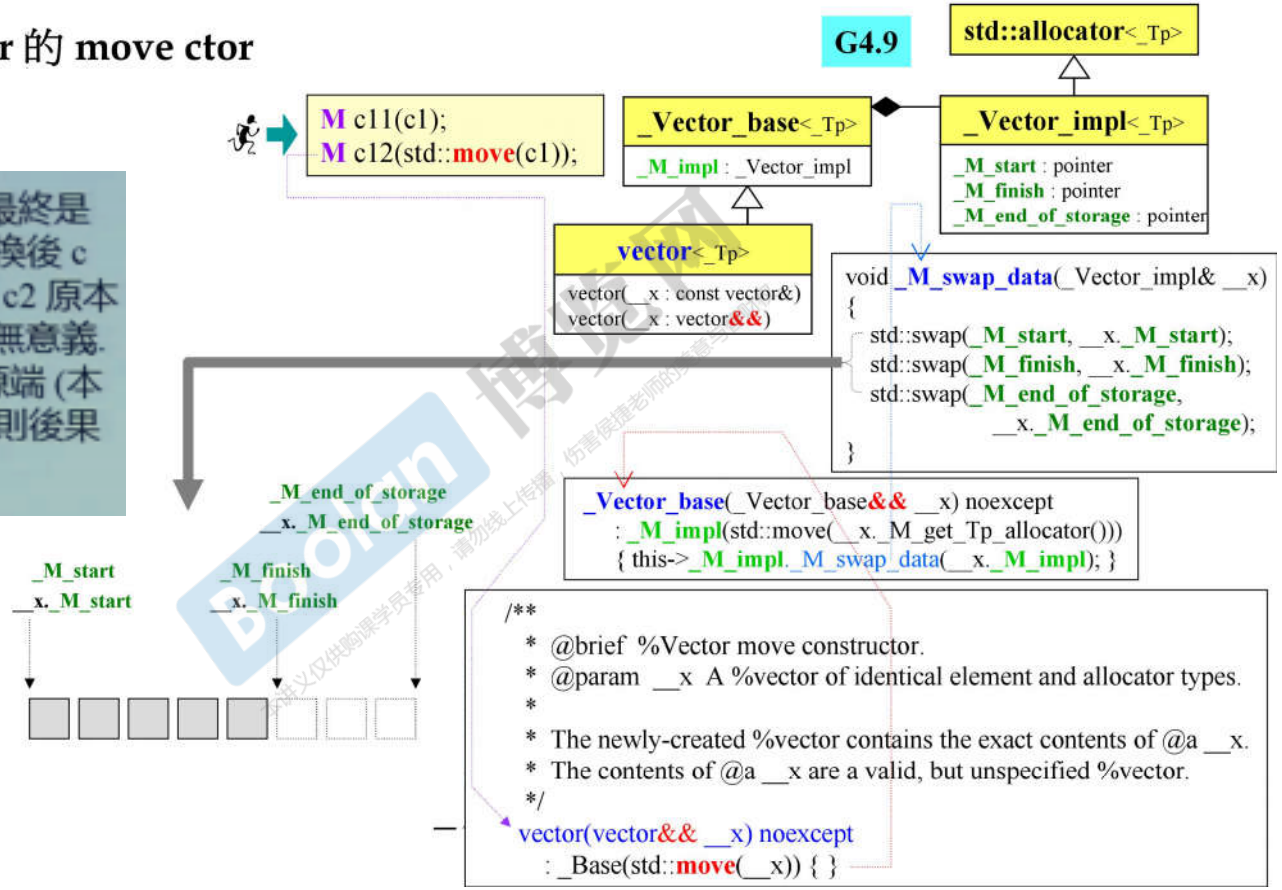
深拷贝



由此圖看出, copy 最終的確完成了 memory allocation 及 copy ctors 的調用。

vector 的 move ctor

由此圖看出, move-copy 最終是三根指針進行 swap(), 交換後 c 成了 c2 而 c2 成了 c。但 c2 原本無意義, 所以現在 c 變得無意義. 因此 move-copy 之後的源端 (本例的 c)不能再被使用, 否則後果自負。



std::string 是否 moveable？

```
579  #if __cplusplus >= 201103L
580        /**
581         *  @brief  Move assign the value of @a str to this string.
582         *  @param  __str  Source string.
583         *
584         *  The contents of @a str are moved into this string (without copying).
585         *  @a str is a valid, but unspecified string.
586         **/
587        // PR 58265, this should be noexcept.
588        basic_string&
589        operator=(basic_string&& __str)
590        {
591      // NB: DR 1204.
592      this->swap(__str);
593      return *this;
594        }
```

```
504  #if __cplusplus >= 201103L
505        /**
506         *  @brief  Move construct stri
507         *  @param  __str  Source strin
508         *
509         *  The newly-created string contains the exact contents of @a __str.
510         *  @a __str is a valid, but unspecified string.
511         **/
512        basic_string(basic_string&& __str)
513  #if _GLIBCXX_FULLY_DYNAMIC_STRING == 0
514        noexcept // FIXME C++11: should always be noexcept.
515  #endif
516        : _M_dataplus(__str._M_dataplus)
517        {
518  #if _GLIBCXX_FULLY_DYNAMIC_STRING == 0
519      __str._M_data(_S_empty_rep()._M_refdata());
520  #else
521      __str._M_data(_S_construct(size_type(), _CharT(), get_allocator()));
522  #endif
523        }
```

```
549        /**
550         *  @brief  Assign the value of @a str to this string.
551         *  @param  __str  Source string.
552         */
553        basic_string&
554        operator=(const basic_string& __str)
555        { return this->assign(__str); }
```

```
456        /**
457         *  @brief  Construct string with copy of value of @a str.
458         *  @param  __str  Source string.
459         */
460        basic_string(const basic_string& __str);
```

```
168  template<typename _CharT, typename _Traits, typename _Alloc>
169    basic_string<_CharT, _Traits, _Alloc>::
170    basic_string(const basic_string& __str)
171    : _M_dataplus(__str._M_rep()->_M_grab(_Alloc(__str.get_allocator()),
172                  __str.get_allocator()),
173          __str.get_allocator())
174    { }
```