C++2.0 新特性

C++11/14

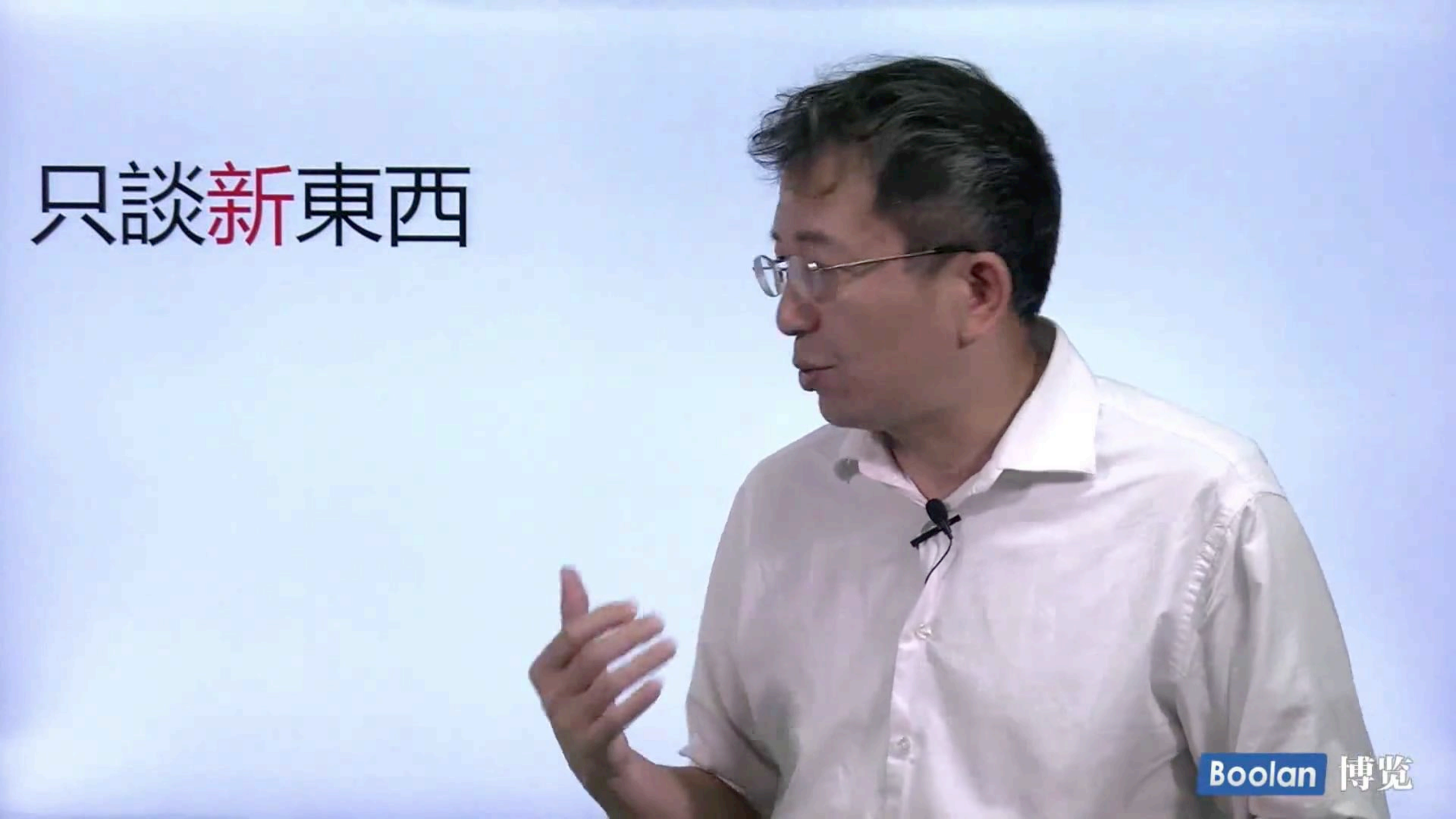
C++ 2.0 New Features

第一講 語言

第二講 標準庫

侯捷

Boolan 博览


只談新東西
Boolan 博览

你應具備的基礎

- C++ 語法, 語意

Boolan 博览

我們的目標

較全面地認識 C++ 2.0 新特性，
並從實例獲得體驗。

Boolan 博览

C++ Standard 之演化

- C++ 98 (1.0)
- C++ 03 (TR1, Technical Report 1)
- C++ 11 (2.0)
- C++ 14

Boolan 博览

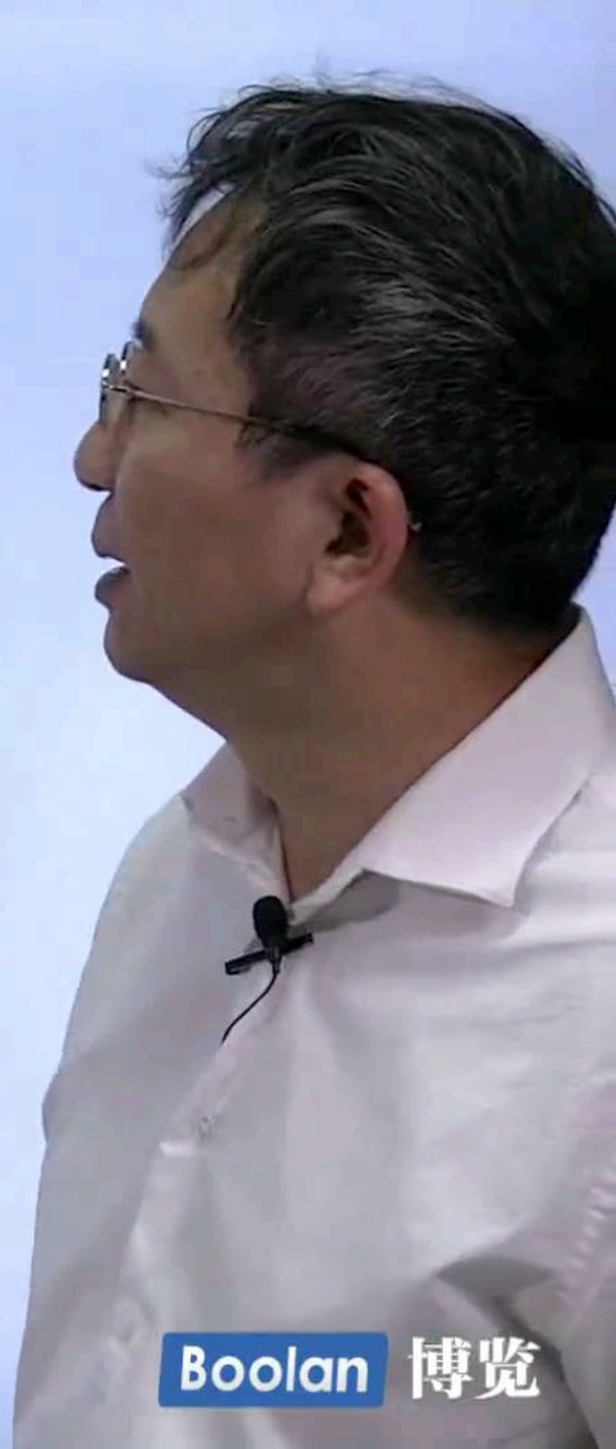
Header files

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>**

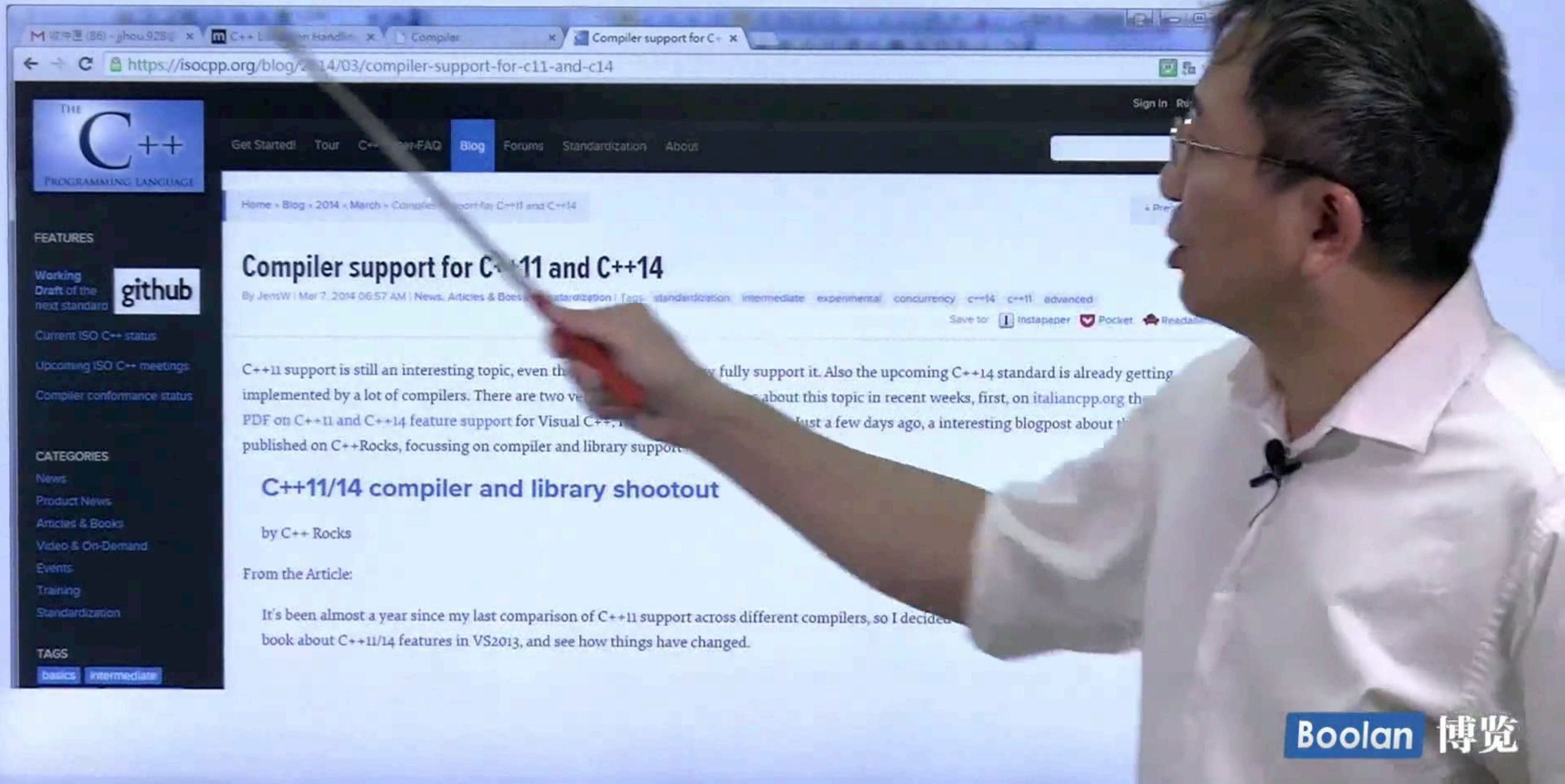
```
#include <type_traits>
#include <unordered_set>
#include <forward_list>
#include <array>
#include <tuple>
#include <regex>
#include <thread>

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.



Boolan 博览

了解你的編譯器對 C++2.0 的支持度



Boolan 博览

了解你的編譯器對 C++2.0 的支持度

Stdcxx Wiki Login

C++0xCompilerSupport

FrontPage | RecentChanges | FindPage | HelpContents | C++0xCompilerSupport

Immutable Page Info More Actions:

The following table lists C++0x features and their support in popular compilers.

Status Of C++ 0x Language Features in Compilers

C++ 0x FEATURE	PAPER(S)	HP aCC	EDG eccp	GCC	Intel C++	MSVC	IBM XLC++
alignas	N2341			4.8			
alignof	N2341			4.5			
Atomic operations	N2427			4.4	13.0	11.0	
auto	v0.9: N1984, v1.0: N2546		4.1 (v0.9)	4.4(v1.0)	11.0(v0.9)	10.0(v0.9)	11.1 (V1.0)
C99 preprocessor	N1653			4.3	11.1		10.1
Concepts [removed]	N2617			ConceptGcc			
constexpr	N2235			4.6	13.0		12.1
decltype	v1.0: N2343, v1.1: N3276		4.1 (v1.0)	4.3(v1.0)	11.0(v1.0)	10.0(v1.0), 11.0 (v1.1)	11.1 (V1.0)
Defaulted And Deleted Functions	N2346		4.1	4.4	12.0		
Delegating Constructors	N1986			4.7		11.0 nov'12	11.1
Explicit conversion operators	N2437			4.5	13.0	11.0 nov'12	12.1
Extended friend Declarations	N1791		4.1	4.7	11.0	10.0***	V1R11,11.1
extern template	N1987	3, 5, 6		3.3	9	6.0	V1R11,11.1

Boolan 博览

了解你的編譯器對 C++2.0 的支持度

cpprocks.com/c11-compiler-support-shootout-visual-studio-gcc-clang-intel/

C++11 compiler support shootout: Visual Studio, GCC, Clang, Intel

March 14, 2013 Tags: C++11, clang, g++, Intel C++, vs2012

Category: Uncategorized 9 Comments

It's been more than half a year since my last comparison of the C++11 support across different compilers. This time I'd like to see how different compilers stack up based on the documentation for the pre-release versions of these compilers.

The next release of GCC is 4.8 and the upcoming version of Clang is 3.3. If you use Visual Studio 2012, you can install an experimental CTP update released in November 2012 to get additional C++11 features.

I've also thrown in v.13.0 of Intel's C++ compiler out of curiosity, although it isn't pre-release and there were a few features I couldn't find information about. I didn't find any information about the upcoming version of this compiler.

Feature	VS2012 Nov CTP	g++ 4.8	C
auto	Yes	Yes	Y
decltype	Yes	Yes	Y
Rvalue references and move semantics	Yes	Yes	Ye
Lambda expressions	Yes	Yes	Yes
nullptr	Yes	Yes	Yes
static_assert	Yes	Yes	Yes
Range based for loop	Yes	Yes	Yes
Trailing return type in functions	Yes	Yes	Ye
extern templates	Yes	Yes	Y
>> for nested templates	Yes	Yes	
Local and unnamed types as template arguments	Yes		

Boolan 博览

了解你的編譯器對 C++2.0 的支持度

(Dev-C++ 5.11, MinGW with GCC 4.9.2)

C++11 - [C++11.dev] - Dev-C++ 5.11

File Edit Search View Project Execute Tools AStyle Window Help

TDM-GCC 4.9.2 32-bit Release

(globals)

Project | Classes | Debug | Test-C++11.cpp

- C++11
  - Test-C++11.cpp
  - widget-t.h
  - widget.h

```
2195
2196
2197    //從語法上試用各式各樣的 allocators
2198    cout << sizeof(__gnu_cxx::malloc_allocator<int>) << endl;   //1. 大小 1 者其實為 0, fields 都是
2199    cout << sizeof(__gnu_cxx::__pool_alloc<int>) << endl;       //1
2200    cout << sizeof(__gnu_cxx::__mt_alloc<int>) << endl;         //1
2201    cout << sizeof(__gnu_cxx::bitmap_allocator<int>) << endl;   //1
2202    cout << sizeof(__gnu_cxx::array_allocator<int>) << endl;    //8 ==> 因為它有一個 ptr 指向 array 和
2203    cout << sizeof(__gnu_cxx::debug_allocator<std::allocator<double>>) << endl; //8
2204 //1 cout << sizeof(__gnu_cxx::throw_allocator<int>) << endl;    //只有 throw_allocator_base, throw_alloc
2205
2206    //搭配容器
2207    list <int, __gnu_cxx::malloc_allocator<int>> list_malloc;
2208    deque <int, __gnu_cxx::debug_allocator<std::allocator<int>>> deque_debug;
2209    vector<int, __gnu_cxx::__pool_alloc<int>> vector_pool;
2210
2211
2212    //測試 cookie
2213    cout << "sizeof(int)=" << sizeof(int) << endl;          //4
2214    cout << "sizeof(double)=" << sizeof(double) << endl;    //8
2215
2216    cookie_test(std::allocator<int>(), 1);                  //相距 10h (表示有 cookie)
2217    cookie_test(__gnu_cxx::malloc_allocator<int>(), 1);     //相距 10h (表示有 cookie)
```

Compiler | Resources | Compile Log | Debug | Find Re | Close

Abort Compilation

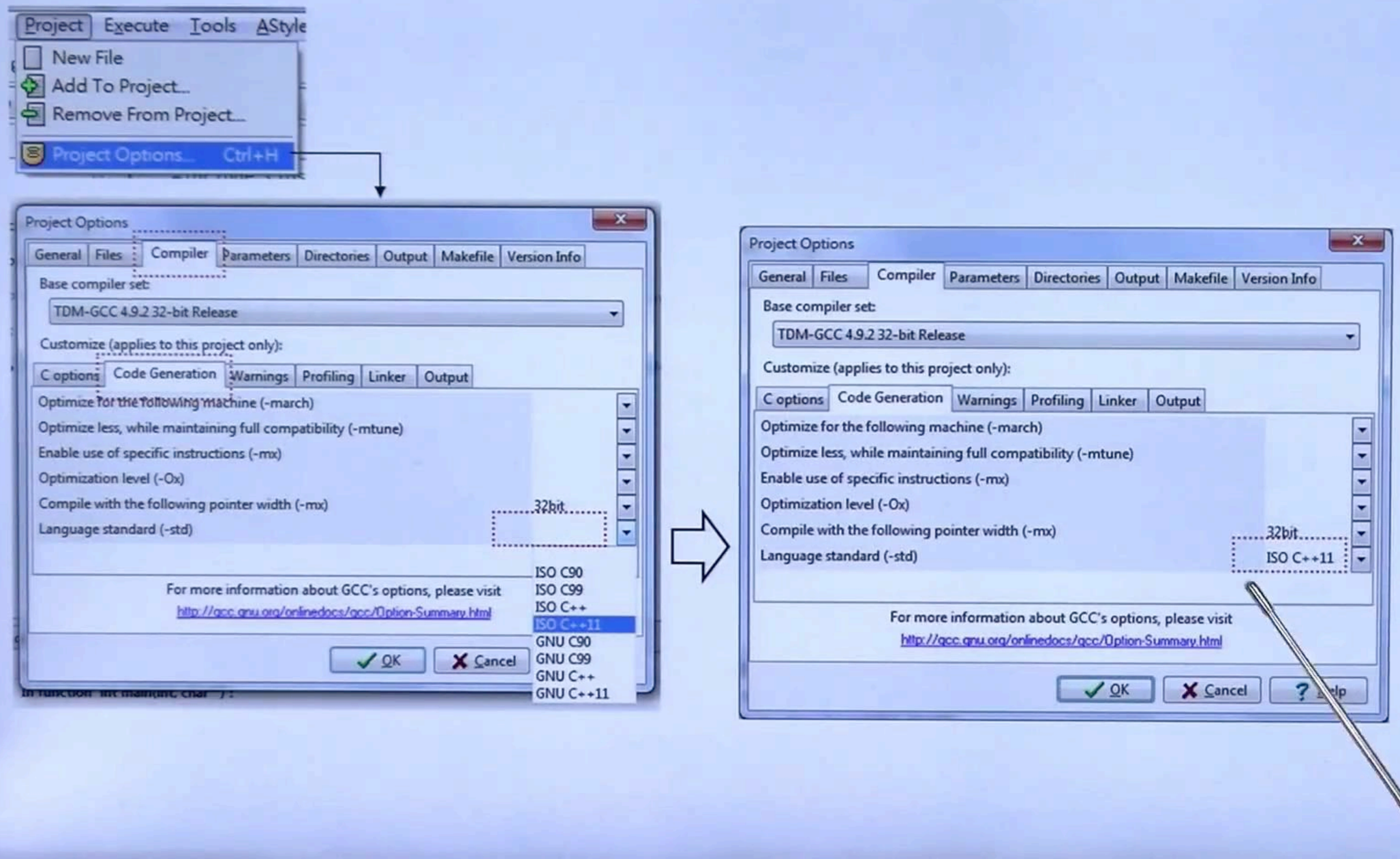
Shorten compiler paths

```
g++.exe Test-C++11.o -o C++11.                    Files/Dev-Cpp/MinGW64/lib32" -L"C:/Progr

Compilation results...
--------
- Errors: 0
- Warnings: 0
- Output Filename: D:\handout\C++11-test-
- Output Size: 2.10485744476318 MiB
- Compilation Time: 7.05s
```

Line: 2161 Col: 1 Sel: 0 Lines: 2989 Length:105275 Insert Done parsing 511 files in 1.281 seconds (398.91 files per second)

Boolan 博览

DevC++ 上的ISO C++11開關



Boolan 博览

C++11 FAQ, from Stroustrup

www.stroustrup.com/C++11FAQ.html

Morgan Stanley | Columbia University | Texas A&M University

home | C++ | FAQ | technical FAQ | C++11 FAQ | publications | TC++PL | Tour++ | Programming | D&E | bio | interviews | applications | glossary | compilers

C++11 - the new ISO C++ standard

This document is written by and maintained by Bjarne Stroustrup. Constructive comments, corrections, references, and suggestions are of course most welcome. Currently, I'm working to improve completeness and clean up the referen

Translations:

- Russian
- Chinese
- Japanese
- Czech. Warning: I have been told that this is a machine translation.
- Korean

I have contributed to the new, unified, isocpp.org C++ FAQ maintained by The C++ Foundation of which I am a director. The maintenance of this FAQ is likely to become increasingly sporatic.

C++11 is the ISO C++ standard ratified in 2011. The previous standard is often referred to as C++98 or C++03. The differences between C++98 and C++03 are so few and so technical that they ought not concern users.

A late working paper is available. This is close to the final draft international standard formally accepted by a national vote in August 2011.

Before its official ratification, we called the upcoming standard C++0x. I have not had the time to update the nam and anyway I like the name C++0x :-). The name "C++0x" is a relict of the days where I and others hoped for (i.e., C++0B == C++11).

All official documents relating to C++11/C++0x can be found at the ISO C++ committee's website. The official 22 WG21.

Caveat: This FAQ will be under construction for quite a while. Comments, questions, references, corrections, and su

Purpose

The purpose of this C++11 FAQ is

- To give an overview of the new facilities (language features and standard libraries) offered by C++11 in addition to what is provided by the standard.
- To give an idea of the aims of the ISO C++ standards effort.
- To present a user's view of the new facilities
- To provide references to allow for a more in depth study of features.
- To name many of the individuals who contributed (mostly as authors of the reports they wrote for the committee). The standard is not written by a facele

Please note that the purpose of this FAQ is not to provide comprehensive discussion of individual features or a detailed explanation of how to use them. The aim is to independently of how complex a feature is. Details can often be found in the references.

Lists of questions

Here are some high-level questions

- What do you think of C++11?
- When will C++0x be a formal standard?

Boolan 博览

重要網頁, CPlusPlus.com

www.cplusplus.com

cplusplus.com

Search: Go

Not logged

home register log in

C++

- Information
- Tutorials
- Reference
- Articles
- Forum

Welcome to cplusplus.com

© The C++ Resources Network

Information

General information about the C++ programming language, including non-technical documents and descriptions:

- Description of the C++ language
- History of the C++ language
- F.A.Q., Frequently Asked Questions

Tutorials

Learn the C++ language from its basics up to its mo advanced features.

- C++ Language: Collection of tutorials coverin the features of this versatile and powerful language. Including detailed explanations of pointers, functions, classes and templates, among others...
- more...

Reference

Description of the most important classes, functions and objects of the Standard Language Library, with descriptive fully-functional short programs as examples:

- C library: The popular C library, is also part of the of C++ language library.
- IOStream library. The standard C++ library for Input/Output operations.
- String library. Library defining the string class.
- Standard containers. Vectors, lists, maps, sets...
- more...

Articles

User-contributed articles, organized into different categories

You can co own articles!

Forum

Message boards where members can exchange knowledge and comments. Ordered by topics:

C++

Search this website:

Boolan 博览

重要網頁, CppReference.com

en.cppreference.com/w/

cppreference.com

Create account | Search

Page | Discussion | View | View source | History

C++ is 30 years old
October 14, 1985 was the release date of the first commercial compiler and the first edition of The C++ Programming Language

C++ reference
C++98, C++03, C++11, C++14

ASCII chart
Compiler support
Language
Preprocessor
Keywords
Operator precedence
Escape sequences
Fundamental types
Headers
Library concepts
Utilities library
Type support
Dynamic memory management
Error handling
Program utilities
Date and time
bitset
Function objects
pair — tuple (C++11)
integer_sequence (C++14)

Strings library
basic_string
Null-terminated byte strings
Null-terminated mult te strings
Null-terminated wide ngs
Containers library
array (C++11)
vector — deque
list — forward_list (C
set — multiset
map — multimap
unordered_set (C++11)
unordered_multiset (C++
unordered_map (C++11)
unordered_multimap (C++11)
stack — queue — priority_queue
Algorithms library
Iterators library
Numerics library
Common mathematical functions
Complex numbers
Pseudo-random number generation

Input/output library
basic_streambuf
basic_filebuf
basic_stringbuf
ios_base
basic_ios
basic_istream
basic_ostream
basic_iostream
basic_ifstream
basic_ofstream
asic_fstream
ic_istringstream
ostringstream
tringstream
lators
Lo rary
Reg library (C
Atomi ry (C
Thread

Technical specifications
Standard library extensions (library fundamentals TS)
optional — any — basic_string_view
polymorphic allocators — search
Standard library extensions v2 (library fundamentals TS v2)
propagate_const — not_fn — observer_ptr
source_location — ostream_joiner
detection idiom — uniform container erasure

Boolan 博览

重要網頁, gcc.gnu.org

https://gcc.gnu.org/onlinedocs/gcc-4.6.3/libstdc++/api/a00464.html

libstdc++

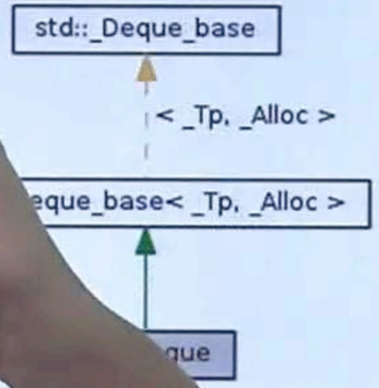
- std::ctype< char >
- std::ctype< wchar_t >
- std::ctype_base
- std::ctype_byname
- std::ctype_byname< char >
- std::decay
- std::decimal::decimal128
- std::decimal::decimal32
- std::decimal::decimal64
- std::default_delete
- std::default_delete< _Tp[]>
- std::defer_lock_t
- std::deque
- std::discard_block_engine
- std::discrete_distribution
- std::discrete_distribution::param_typ
- std::divides
- std::domain_error
- std::enable_if
- std::enable_shared_from_this
- std::equal_to
- std::error_category
- std::error_code
- std::error_condition

std::deque Class Reference

Sequences

Public Types | Public Mem
Protected Types | Protected Mem
Static Protected Mem
Prot

Inheritance diagram for std::deque:



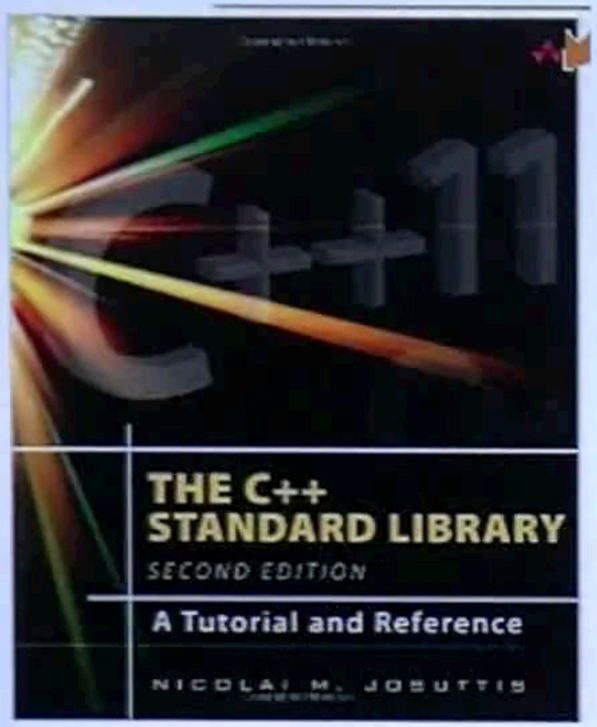
List of all members.

Public Types

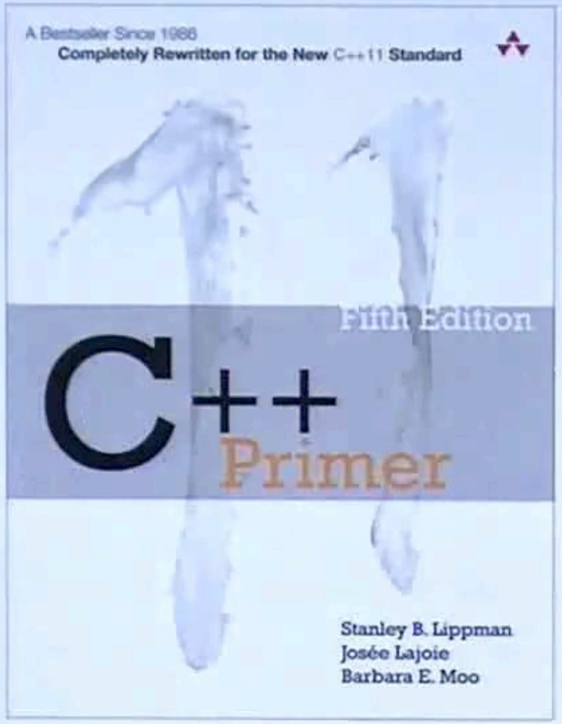
- typedef _Alloc **allocator_type**
- typedef _Base::const_iterator **const_iterator**
- typedef _Tp_alloc_type::const_pointer **const_pointer**
- typedef _Tp_alloc_type::const_reference **const_reference**
- typedef std::reverse_iterator < const_iterator > **const_reverse_iterator**
- typedef ptrdiff_t **difference_type**
- typedef _Base::iterator **iterator**
- typedef _Tp_alloc_type::pointer **pointer**
- typedef _Tp_alloc_type::reference **reference**

Boolan 博览

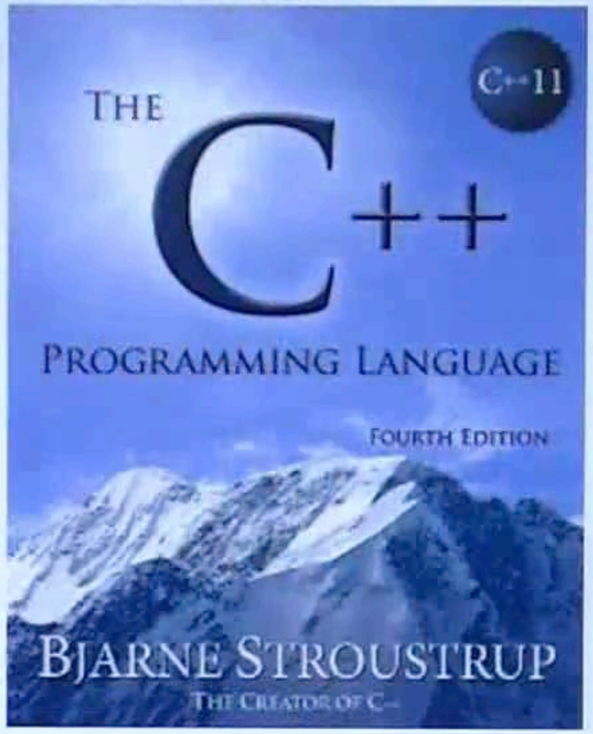
Bibliography



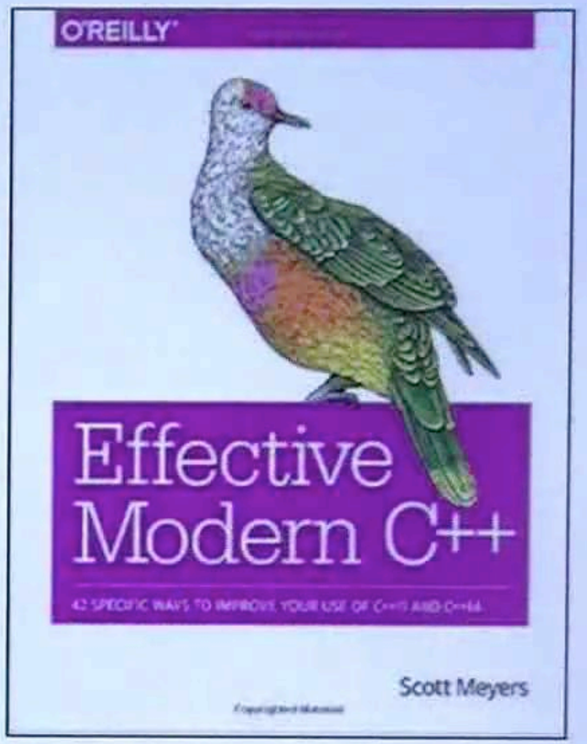
Apr 9, 2012



Aug 16, 2012



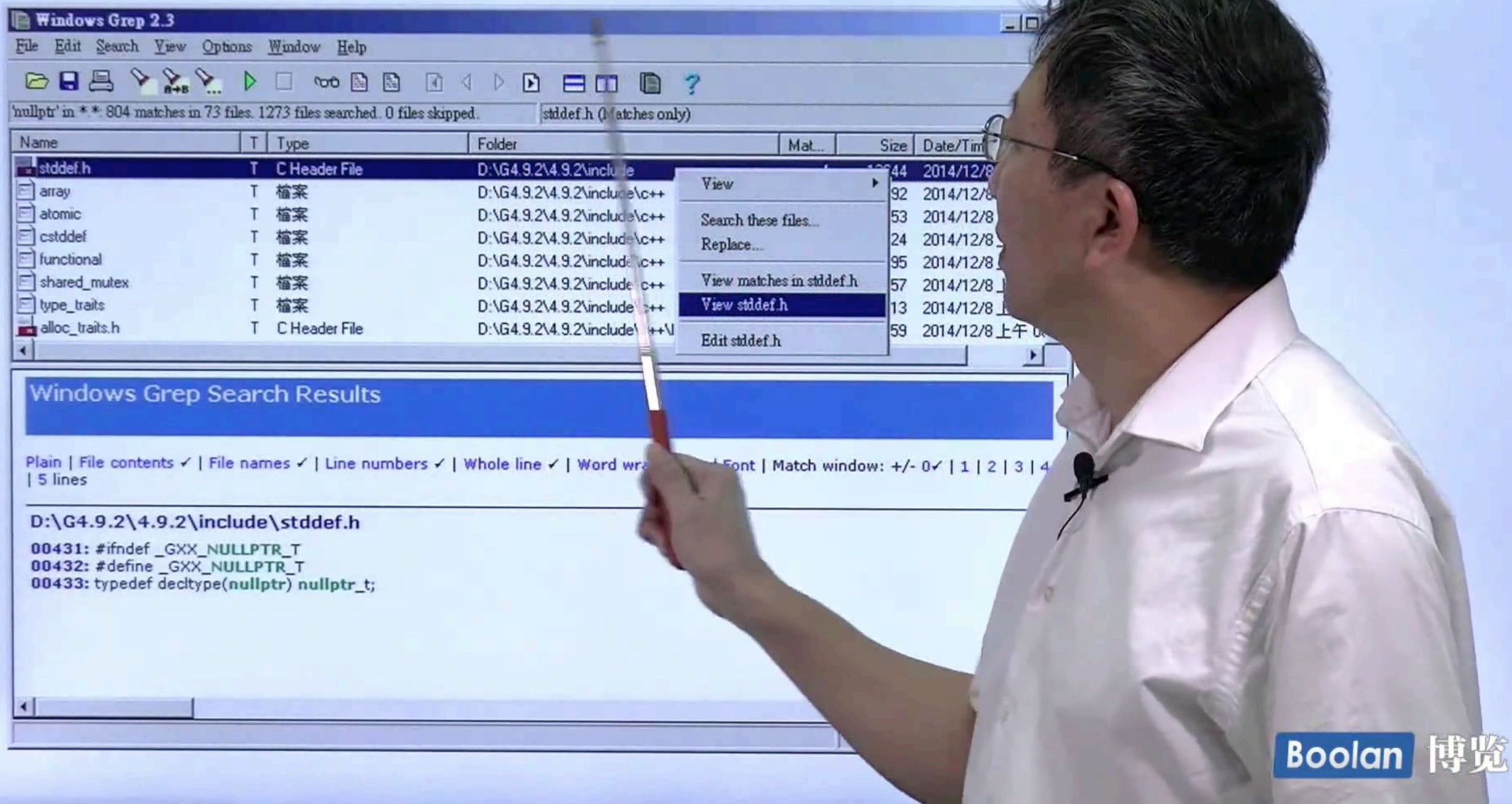
May 20, 2013



Nov. 2014

Boolan 博览

選一個順手的全文檢索工具



Boolan 博览

選一個順手的全文檢索工具

Windows Grep 2.3

File Edit Search View Options Window Help

'nullptr' in *.*: 804 matches in 73 files. 1273 files searched. 0 files skipped.

stddef.h (Whole file)

Name	T	Type	Folder	Mat...	Size	Date/Time
stddef.h	T	C Header File	D:\G4.9.2\4.9.2\include	4	13644	2014/12/8 上午 0
array	T	檔案	D:\G4.9.2\4.9.2\include\c++	1	9692	2014/12/8 上午 0
atomic	T	檔案	D:\G4.9.2\4.9.2\include\c++	2	34353	2014/12/8 上午 0
cstddef	T	檔案	D:\G4.9.2\4.9.2\include\c++	1	1824	2014/12/8 上午 0
functional	T	檔案	D:\G4.9.2\4.9.2\include\c++	6	80095	2014/12/8 上午 0
shared_mutex	T	檔案	D:\G4.9.2\4.9.2\include\c++	3	9657	2014/12/8 上午 0
type_traits	T	檔案	D:\G4.9.2\4.9.2\include\c++	3	64213	2014/12/8 上午 0
alloc_traits.h	T	C Header File	D:\G4.9.2\4.9.2\include\c++\bits	1	16559	2014/12/8 上午 0

```
00424: long long __max_align_ll __attribute__((__aligned__(__alignof__(long long))));
00425: long double __max_align_ld __attribute__((__aligned__(__alignof__(long double))));
00426: } max_align_t;
00427: #endif
00428: #endif /* C11 or C++11. */
00429:
00430: #if defined(__cplusplus) && __cplusplus >= 201103L
00431: #ifndef _GXX_NULLPTR_T
00432: #define _GXX_NULLPTR_T
00433: typedef decltype(nullptr) nullptr_t;
00434: #endif
00435: #endif /* C++11. */
00436:
00437: #endif /* _STDDEF_H was defined this time */
00438:
00439: #endif /* !_STDDEF_H && !_STDDEF_H_ && !_ANSI_STDDEF_H && !__STDDEF_H__
00440: || __need_XXX was not defined before */
```

Boolan 博览

你將獲得的代碼

Test-C++2.0.cpp

其內涵蓋本課程對 C++ 2.0 New Features 的測試

Boolan 博览

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;
}
```

C:\Windows\system32\cmd...

```
199711
Press any key to continue . . .
```

Dev-C++ 5

```
#include <iostream>

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

32bit

ISO C++11

D:\temp\Untitled3.exe

```
199711
--------------------------------
Process exited after 0.06942 second
```

D:\handout\C++11-test-DevC++\C++1

```
201103
--------------------------------
Process exited after
```

Boolan 博览

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
- …

Boolan 博览

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  }
```

Boolan 博览

Variadic Templates

數量不定的模板參數

```
❶ void print ()
{
}

❷ 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

```
❸ 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);
  }
};
```

```
#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

```
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

Boolan 博览

Variadic Templates

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

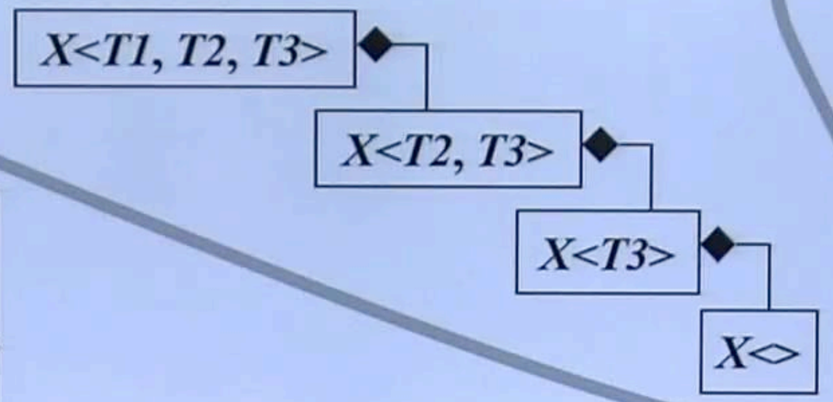
```
template<typename Head, typename... Tail>
class tuple<Head, Tail...>
 : private tuple<Tail...>
{
  typedef tuple<Tail...> inherited;
 public:
  tuple() { }
  tuple(Head v, Tail... vtail)
   : m_head(v), inherited(vtail...) { }

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

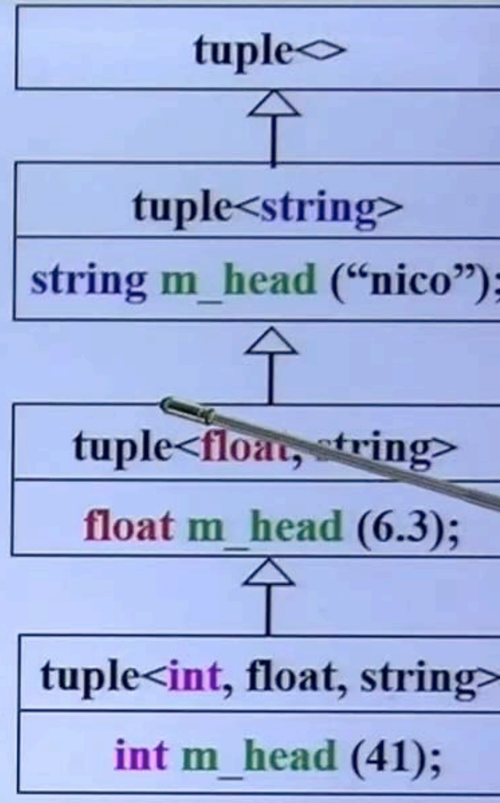
可以很方便地完成 recursive inheritance.

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

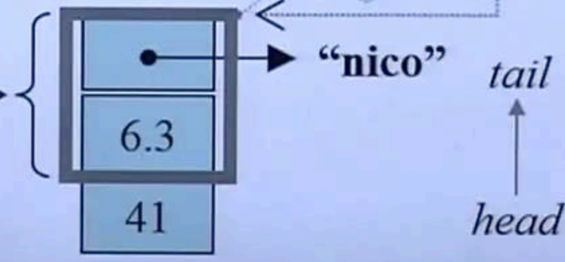
注意這是
initialization list

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()) ➔



Boolan 博览

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>>
```

Boolan 博览

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*)
```

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).

4.9.2\include\stddef.h

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

Boolan 博览

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**.

Boolan 博览

auto keyword

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

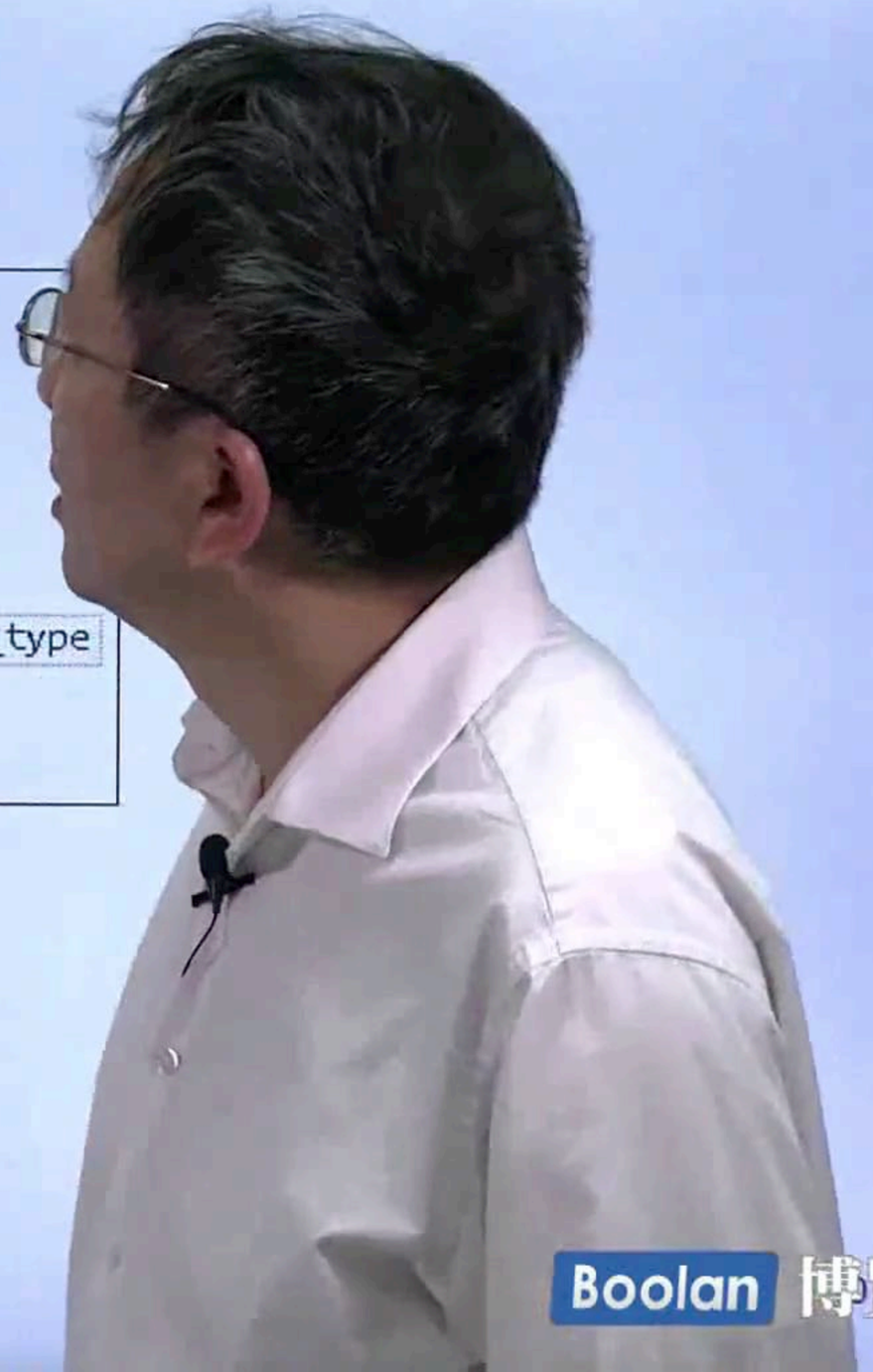
```
list<string> c;
...
auto ite = find(c.begin(),c.end(),target);
```

Boolan 博览

auto keyword

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

```
#if __cplusplus >= 201103L
    // DR 685.
    inline auto
    operator-(const reverse_iterator<_IteratorL>& __x,
              const reverse_iterator<_IteratorR>& __y)
    -> decltype(__y.base() - __x.base())
#else
    inline typename reverse_iterator<_IteratorL>::difference_type
    operator-(const reverse_iterator<_IteratorL>& __x,
              const reverse_iterator<_IteratorR>& __y)
#endif
```



Boolan 博览

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:

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)

```
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)
```

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。

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

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

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

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

Boolan 博览

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*

Boolan 博览

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  0x405ce

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                    narrowing]
int x4={5.3};
                 //[Warning] narrowing conversion of '5.2999999999999998e+0                   de { } [-Wnarrowin
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' f                win
                 //[Warning] narrowing conversion of '5.5999999999999996e+0'                  win
for ( auto& elem : v2 ) {
      cout << elem << ' ';  // 1 2 4 5
}
cout << endl;
```

Boolan 博览

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)

Boolan 博览

initializer_list<>

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

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

    ❷ 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.

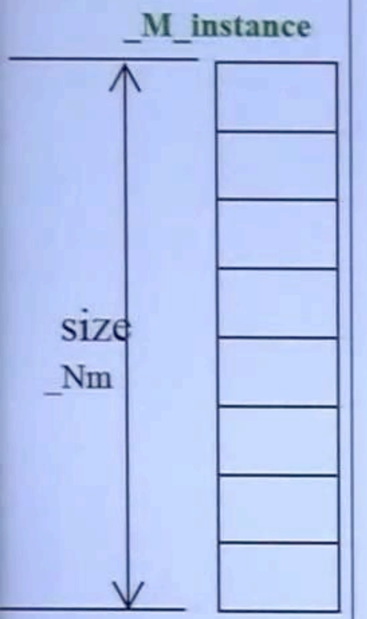
```
46  template<class _E>
47    class initializer_list
48    {
49    public:
50      typedef _E          value_type;
51      typedef const _E&       reference;
52      typedef const _E&       const_reference;
53      typedef size_t          size_type;
54      typedef const _E*       iterator;
55      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    };
```

Boolan 博览

容器 array

TR1

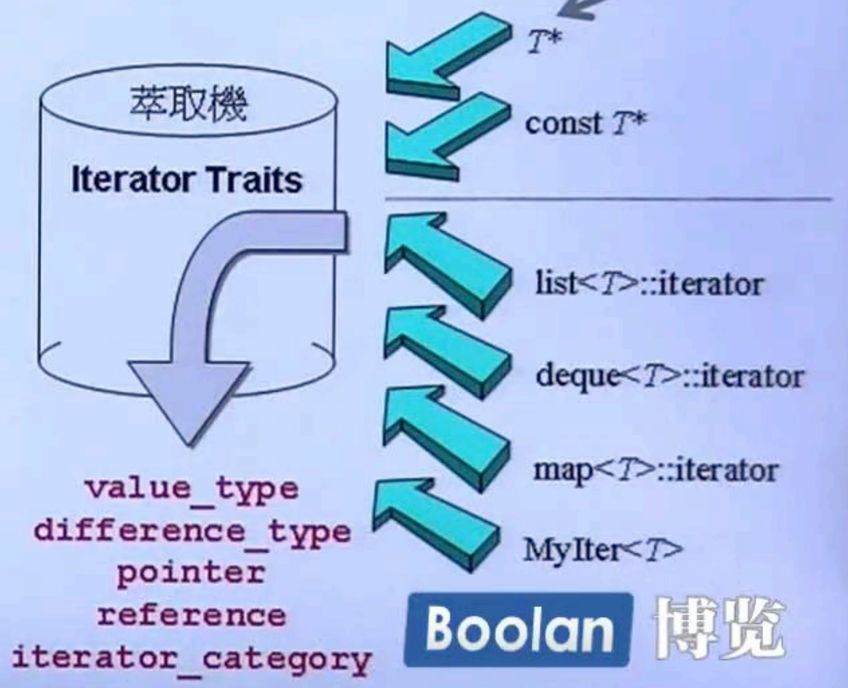
array<_Tp, _Nm>
_M_instance : _Tp[_Nm]




```
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)

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

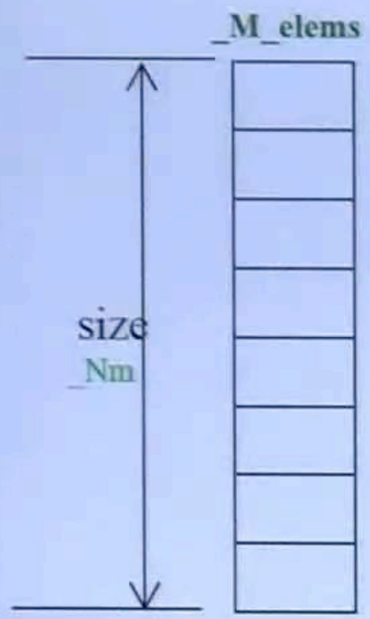


没有ctor，没有dtor

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;
```

Boolan 博览

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  template<class _E>
47    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() + _M_len; }
      };
```

Boolan 博览

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++\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)
```

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>::opera
```

Boolan 博览

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()); }
```

```
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
```

```
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()); }
```

Boolan 博览

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;
```

```
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
```

Boolan 博览

explicit for ctors taking more than one argument

```
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>)
```

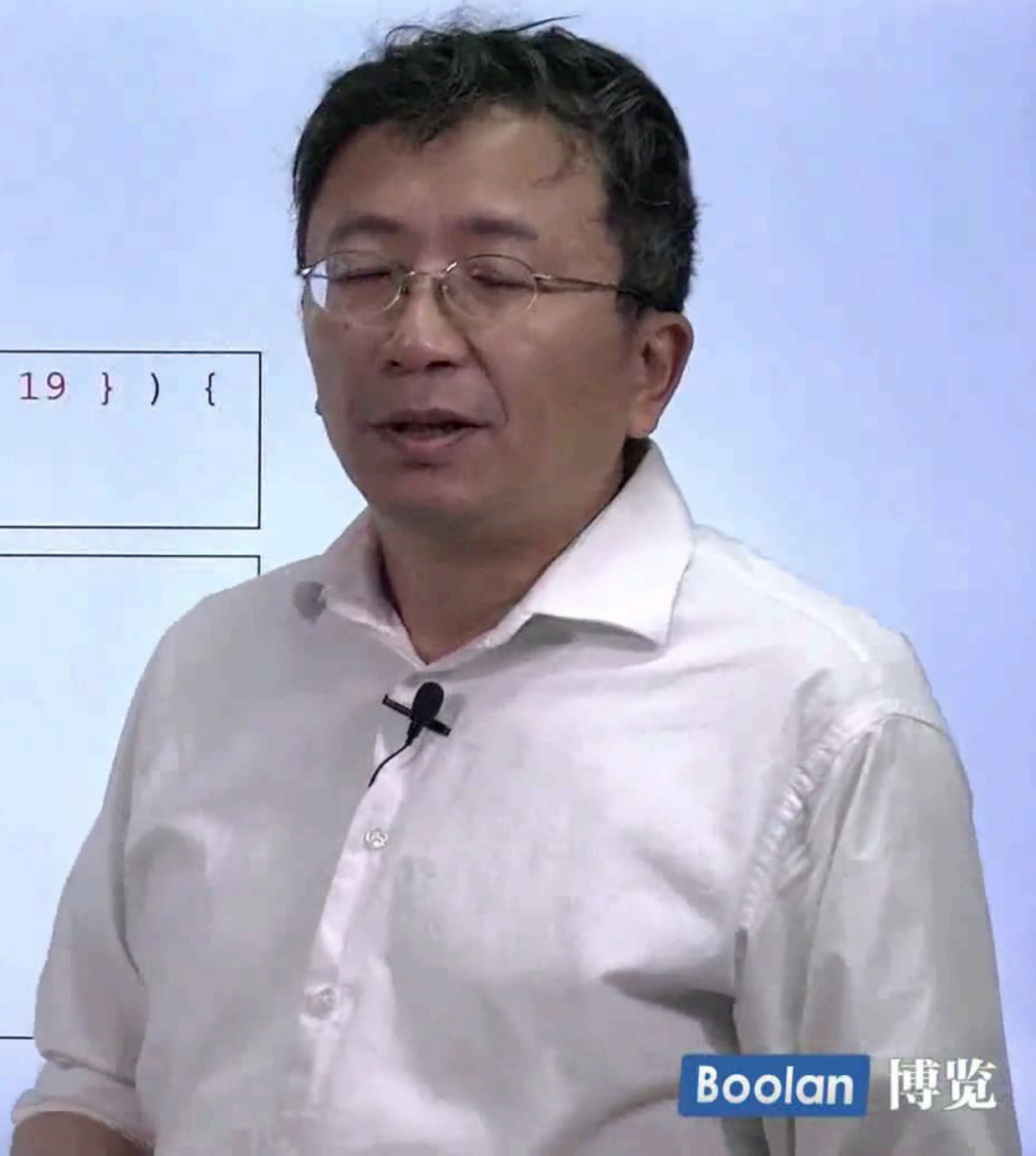
Boolan 博览

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;
}
```



Boolan 博览

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;
    }
}
```

Boolan 博览

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;
}
```

Boolan 博览

=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;
};
```

Boolan 博览

=delete

Windows Grep 2.3

File Edit Search View Options Window Help

'= delete' in *.*: 312 matches in 48 files. 2539 files searched. 0 f | unique_ptr.h (Matches only)

Name	T	Type	Folder
random.h	T	C Header File	D:\G4.9.2\4.9.2\include\c++\bits
regex_automaton.h	T	C Header File	D:\G4.9.2\4.9.2\include\c++\bits
shared_ptr_base.h	T	C Header File	D:\G4.9.2\4.9.2\include\c++\bits
unique_ptr.h	T	C Header File	D:\G4.9.2\4.9.2\include\c++\bits
optional	T	檔案	D:\G4.9.2\4.9.2\include\c++\experimental

Windows Grep Search Results

Fancy | File contents ✓ | File names ✓ | Line numbers ✓ | Whole line ✓ | Fixed Font ✓ | Match window: +/- 0✓ | 1 | 2 | 3 | 4 | 5 lines

D:\G4.9.2\4.9.2\include\c++\bits\unique_ptr.h

```
00124:  operator()(_Up*) const = delete;
00356:      unique_ptr(const unique_ptr&) = delete;
00357:      unique_ptr& operator=(const unique_ptr&) = delete;
00435:  unique_ptr(_Up* __p) = delete;
00587:  void reset(_Up*) = delete;
00598:      unique_ptr(const unique_ptr&) = delete;
00599:      unique_ptr& operator=(const unique_ptr&) = delete;
00606:     deleter_type, const deleter_type&>::type) = delete;
00612:     remove_reference<deleter_type>::type&&) = delete;
00776:    make_unique(_Args&&...) = delete;
```

Boolan 博览

=default

Windows Grep 2.3

File Edit Search View Options Window Help

'=default' in *.*: 248 matches in 42 files. 2539 files searched. 0 | tuple (Matches only)

Name	T	Type	Folder
scoped_allocator	T	檔案	D:\G4.9.2\4.9.2\include\c++
system_error	T	檔案	D:\G4.9.2\4.9.2\include\c++
thread	T	檔案	D:\G4.9.2\4.9.2\include\c++
tuple	T	檔案	D:\G4.9.2\4.9.2\include\c++
atomic_base.h	T	C Header File	D:\G4.9.2\4.9.2\include\c++\bits

Windows Grep Search Results

Fancy | File contents ✓ | File names ✓ | Line numbers ✓ | Whole line ✓ | Fixed Font ✓ | Match
window: +/- 0✓ | 1 | 2 | 3 | 4 | 5 lines

D:\G4.9.2\4.9.2\include\c++\tuple

```
00195:      _Tuple_impl() = default;
00264:      constexpr _Tuple_impl(const _Tuple_impl&) = default;
00407:      constexpr tuple(const tuple&) = default;
00409:      constexpr tuple(tuple&&) = default;
00540:      constexpr tuple(const tuple&) = default;
00542:      constexpr tuple(tuple&&) = default;
```

```
386  /// Primary class template, tuple
387  template<typename... _Elements>
388    class tuple : public _Tuple_impl<0, _Elements...>
389    {
```

```
407      constexpr tuple(const tuple&) = default;
408
409      constexpr tuple(tuple&&) = default;
```

Copy

Move

```
518  /// Partial specialization, 2-element tuple.
519  /// Includes construction and assignment from a pair.
520  template<typename _T1, typename _T2>
521    class tuple<_T1, _T2> : public _Tuple_impl<0, _T1, _T2>
522    {
```

```
540      constexpr tuple(const tuple&) = default;
541
542      constexpr tuple(tuple&&) = default;
```

Copy

Move

Boolan 博览

=default

Windows Grep 2.3

File Edit Search View Options Window Help

'= default;' in *.*: 576 matches in 129 files. 10474 files chrono (Matches only)

Name	T	Type	Folder
unordered_set	T	檔案	D:\G4.9.2\4.9.2\include\c++\profil
atomic	T	檔案	D:\vs2013\include
atomic	T	檔案	D:\G4.5.3\include\c++
chrono	T	檔案	D:\G4.5.3\include\c++
future	T	檔案	D:\G4.5.3\include\c++
nested_exception.h	T	C Header File	D:\G4.5.3\include\c++
system_error	T	檔案	D:\G4.5.3\include\c++
thread	T	檔案	D:\G4.5.3\include\c++

Windows Grep Search Results

Fancy | File contents ✓ | File names ✓ | Line numbers ✓ | Whole line ✓ | Fixed Font ✓ | Match window: +/- 0 | 1✓ | 2 | 3 | 4 | 5 lines

D:\G4.5.3\include\c++\chrono

```
00211:          // 20.8.3.1 construction / copy / destroy
00212:  duration() = default;
00213:

00227:
00228:  ~duration() = default;
00229:  duration(const duration&) = default;
00230:  duration& operator=(const duration&) = default;
00231:
```

```
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       };
```


Boolan 博览

=default, =delete

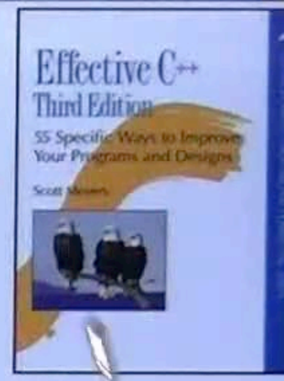
```
Foo f1(5);
Foo f2;      //如果沒有寫出 =default 版本 => [Error] no matching function for call to 'Foo::Foo()'
Foo f3(f1); //如果 copy ctor = delete;   [Error] use of deleted function 'Foo::Foo(const Foo&)')
f3 = f2;     //如果 copy assign = delete; [Error] use of deleted function 'Foo& Foo::operator=(const Foo&)')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

```
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;
};
```

=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)

Boolan 博览

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 { };
```

→

```
class Empty { };

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

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

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;
}
```

這些函式做了什麼呢？呃，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。

Boolan 博览

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    };
```

Boolan 博览

string 有所謂 Big-Three 嗎？

當然有. 而且有 Big-Five

Boolan 博览

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
};
```

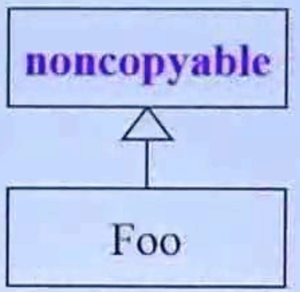
此 class 不允許被 ordinary user code copy，但仍可被 friends 和 members copy。若欲完全禁止，不但必須把 copy controls 放到 private 內且不可定義之。

Boolan 博览

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
```



Boolan 博览

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;
```

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

is equivalent to

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

//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

Boolan 博览

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] '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);
}
```

天方夜譚

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

Boolan 博览

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());
```

Boolan 博览

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==
```

Boolan 博览

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
```


Boolan 博览

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==
```

Boolan 博览

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;
```

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;

Boolan 博览

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;
```

Boolan 博览

noexcept

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

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**().

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

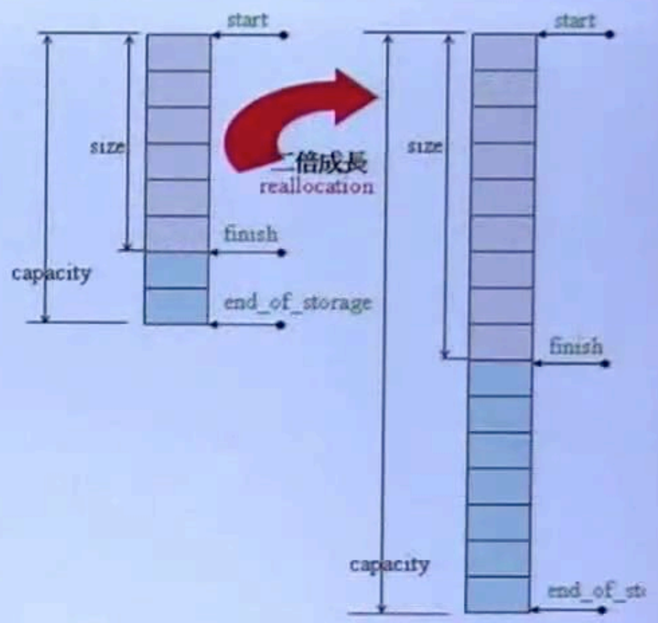
Boolan 博览

noexcept

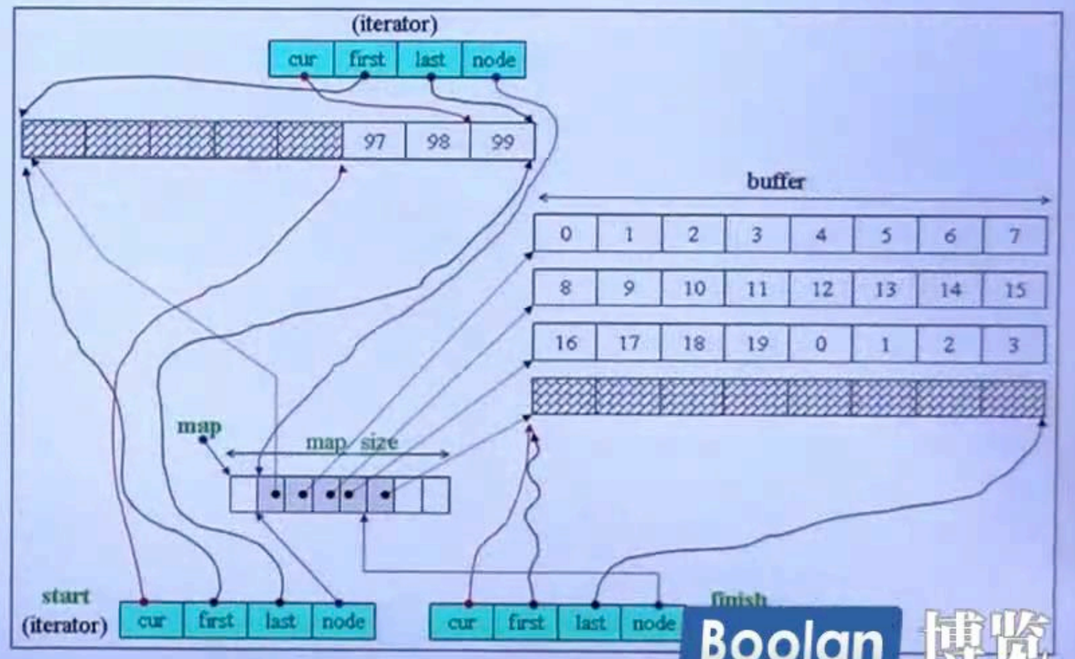
http://stackoverflow.com/questions/8001823/how-to-enforce-move-semantics-when-a-vector-grows

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;  }
…
```

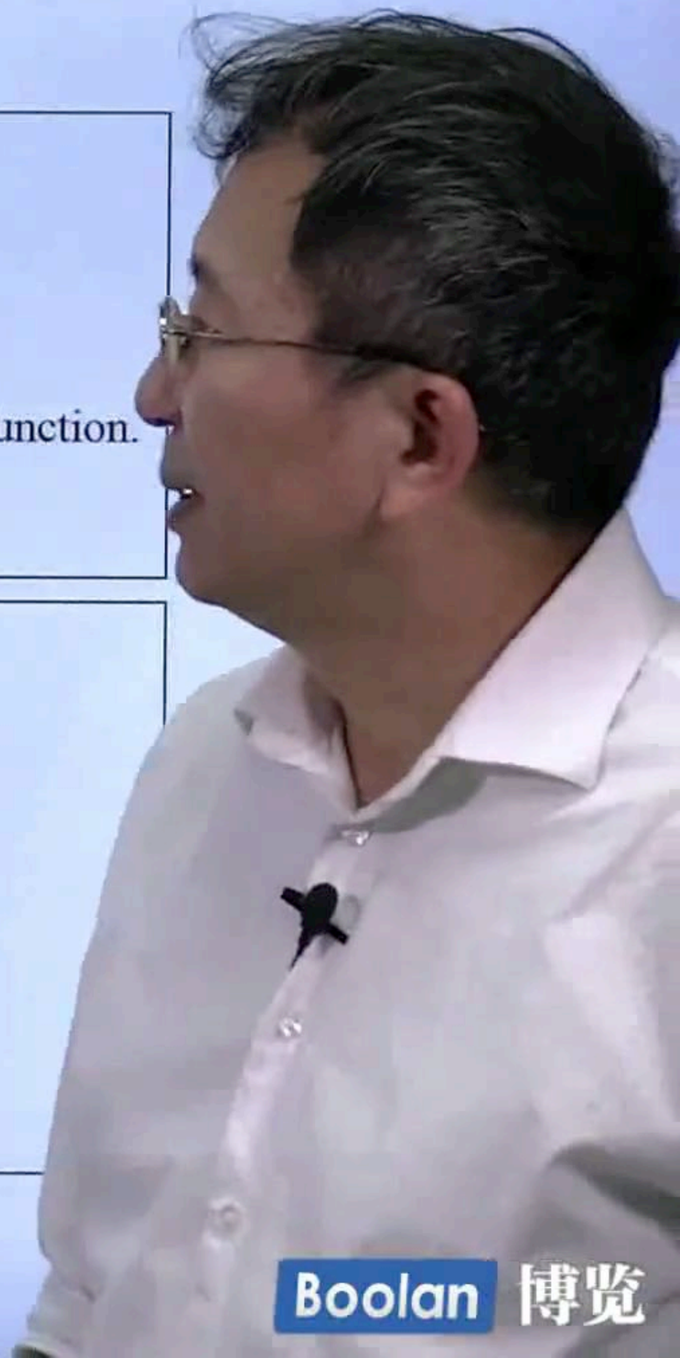


Boolan 博览

override

```
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.
};
```

```
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 { }
};
```

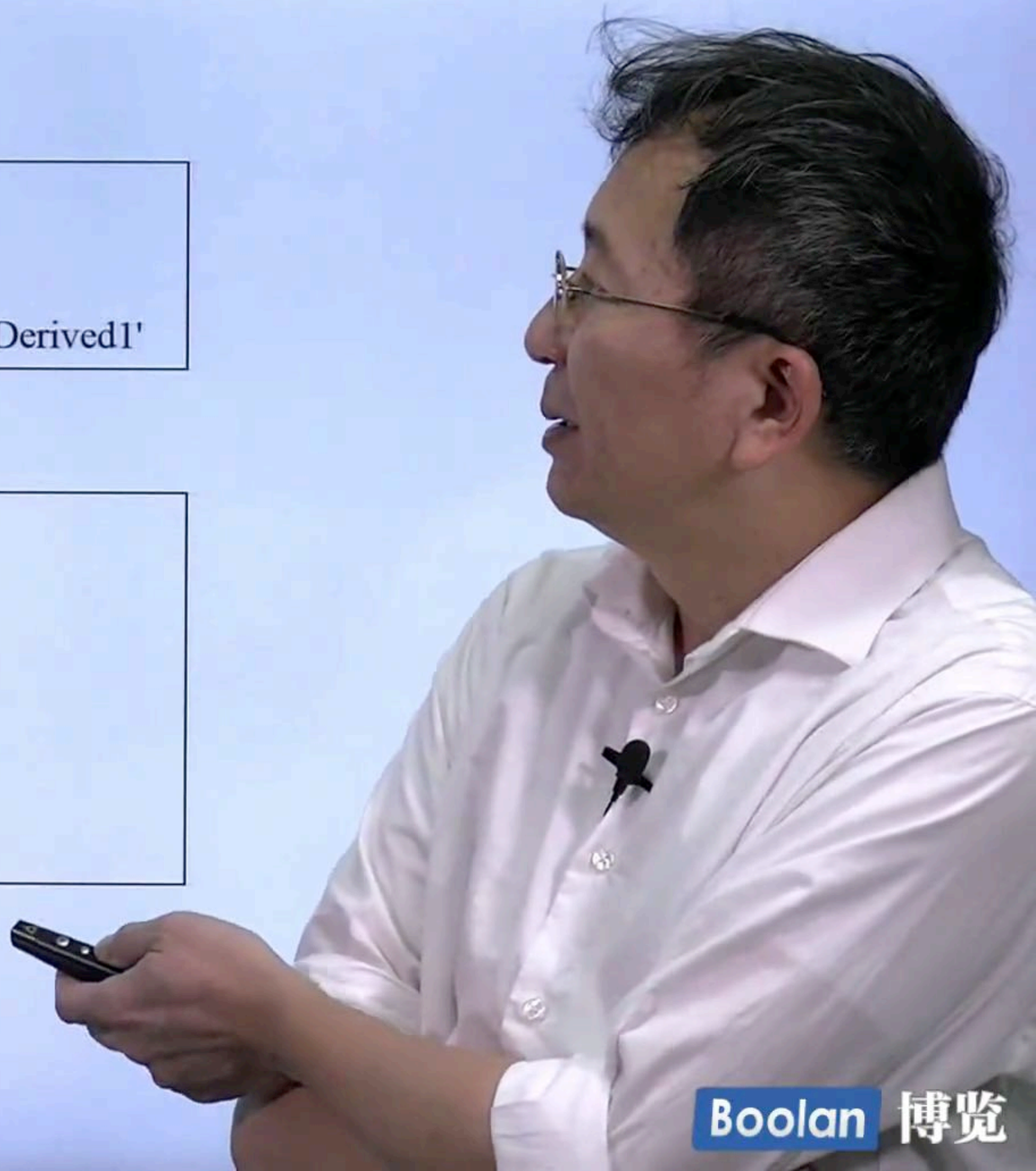


Boolan 博览

final

```
struct Base1 final { };

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()'
};
```



Boolan 博览

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;
```

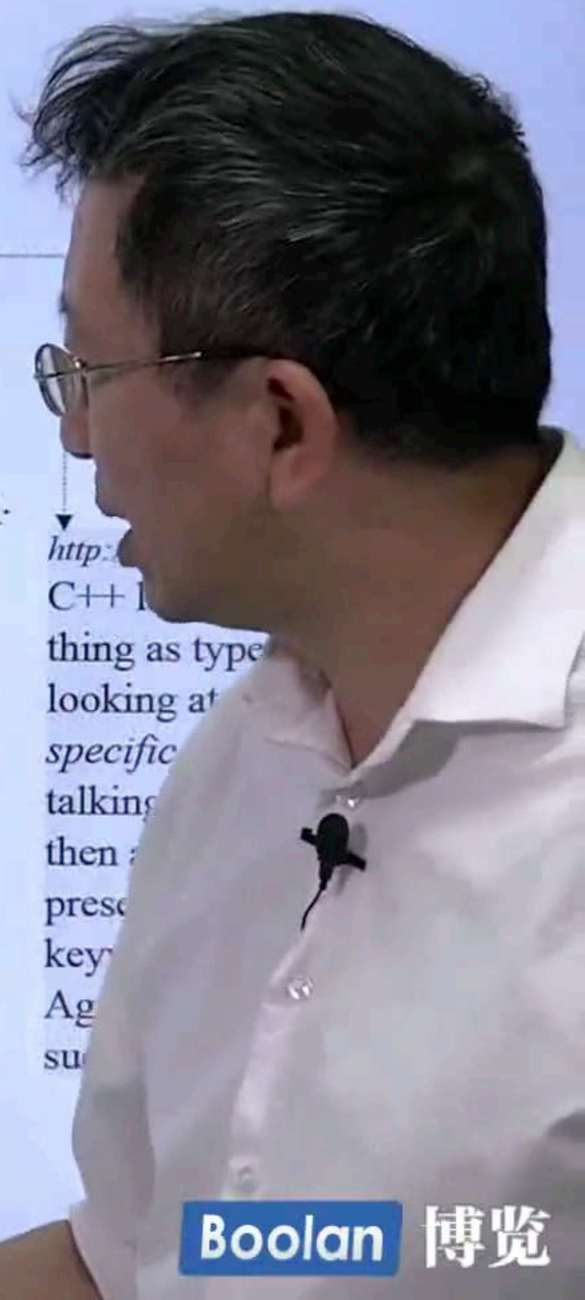
→

這麼寫 (before C++11)

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

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



Boolan 博览

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

Boolan 博览

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);
```

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

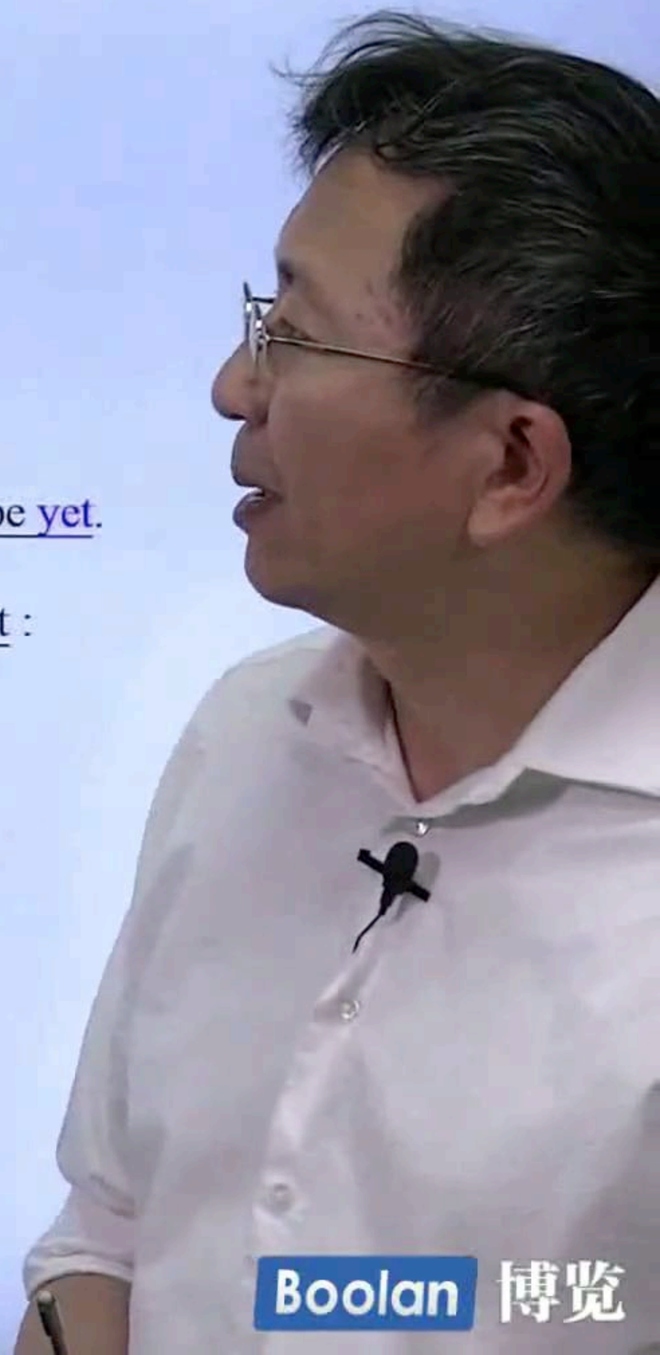
But with C++11, you can alternatively declare the return type of a function behind the parameter list :

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

1

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

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



Boolan 博览

decltype

G4.9 到處都是 decltype

File Edit Search View Options Window Help

'decltype' in * * 514 matches in 90 files 10474 files sea stl_function.h (Matches only)

Name	T	Type	Folder
basic_string.h.gch	B	GCH 檔案	D:\G4.9.2\4.9.2\include\c++\bits
forward_list.h	T	C Header File	D:\G4.9.2\4.9.2\include\c++\bits
hashtable_policy.h	T	C Header File	D:\G4.9.2\4.9.2\include\c++\bits
ptr_traits.h	T	C Header File	D:\G4.9.2\4.9.2\include\c++\bits
range_access.h	T	C Header File	D:\G4.9.2\4.9.2\include\c++\bits
shared_ptr_base.h	T	C Header File	D:\G4.9.2\4.9.2\include\c++\bits
stl_algo.h	T	C Header File	D:\G4.9.2\4.9.2\include\c++\bits
stl_algobase.h	T	C Header File	D:\G4.9.2\4.9.2\include\c++\bits
stl_function.h	T	C Header File	D:\G4.9.2\4.9.2\include\c++\bits
stl_iterator.h	T	C Header File	D:\G4.9.2\4.9.2\include\c++\bits
unique_ptr.h	T	C Header File	D:\G4.9.2\4.9.2\include\c++\bits
optional	T	檔案	D:\G4.9.2\4.9.2\include\c++\experime
c++config.h	T	C Header File	D:\G4.9.2\4.9.2\include\c++\x86_64-

Windows Grep Search Results

Fancy | File contents ✓ | File names ✓ | Line numbers ✓ | Whole line ✓ | Fixed Font ✓ | Match window: +/- 0✓ | 1 | 2 | 3 | 4 | 5 lines

D:\G4.9.2\4.9.2\include\c++\bits\stl_function.h

```
00231:  -> decltype(std::forward<_Tp>(__t) + std::forward<_Up>(
00245:  -> decltype(std::forward<_Tp>(__t) - std::forward<_Up>(
00259:  -> decltype(std::forward<_Tp>(__t) * std::forward<_Up>(
00273:  -> decltype(std::forward<_Tp>(__t) / std::forward<_Up>(
00287:  -> decltype(std::forward<_Tp>(__t) % std::forward<_Up>(
00301:  -> decltype(-std::forward<_Tp>(__t))
00401:  -> decltype(std::forward<_Tp>(__t) == std::forward<_Up>
```

...\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      };
```


Boolan 博览

decltype

in metaprogramming：模板间的运用

```
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

    //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

只要加::，就需要typename帮助编译器确认T::iterator确实是类型，因为编译到此，不确定obj是什么

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

复数类没有迭代器

Boolan 博览

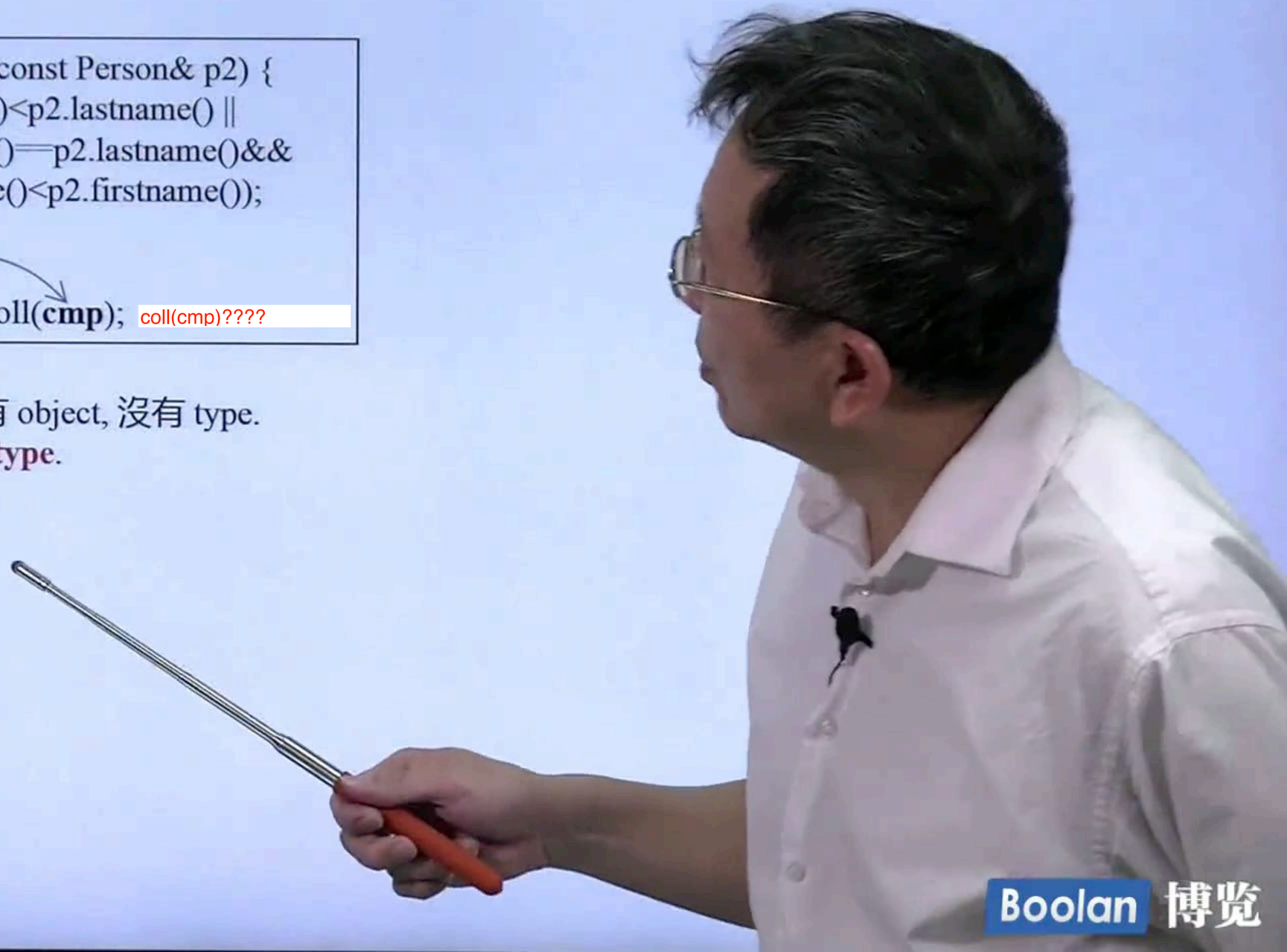
decltype, used to pass the type of a lambda

```
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);
```

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**.

lambda是匿名函数



Boolan 博览

14. lambdas

Lambdas

允许定义出inline，lambda好像inline函数：在某个地方立即写出函数定义

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.**

如：自定义比较大小的方法，即：仿函数/函数对象
lambdas是个函数对象,作用像函数，只是写法不同

A lambda is a definition of functionality that can be defined inside statements and expressions. 哪里使用，就哪里声明调用
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;
}
```

函数/类型，需要被调用

You can call it directly:

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

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

非临时对象，而是直接调用

or pass it to objects to get called:

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

多数这样用，l有类型

Boolan 博览

Lambdas

lambda introducer
导入器：取lambda外部的变量

option: 可选的

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

强制的

三个option有一个，小括号必须写
否则，参数存在就写，不存在可省略

小括号

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

参数

[ ]里数据可否被改写

可否抛出异常

lambda返回类型

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

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.

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

- [=] 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] { ... };
```

pass by value & reference

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

Boolan 博览

Lambdas

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

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

```
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？

没写mutable就不能++，所以左右不完全等同

id编译时就确定值

编译器编译

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

函数对象/functor: 此class重载了（）

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

Boolan 博览

Lambdas

```
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
```

```
int id = 0;
auto f = [id] () mutable {
    cout << "id: " << id << endl;
    static int x =5;
    int y = 6;
    return id;
};
```

可以聲明變量
可以返回數值

```
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，外界內部互相影响

```
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;
```

Boolan 博览

Lambdas

Here is what compiler generates for lambda's:

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

相当于

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

Functor

```
UnNamedLocalFunction lambda2(tobefound);

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

匿名对象：UnNameLocalFunction (5)
对象（）：调用operator()

Boolan 博览

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.

编译器知道且是独一无二的

传给模板，模板可推导出模板参数

```
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);
```

coll(cmp)，set的初值如何给，看set的ctor

set背后是rb tree

unordered_set背后是hash

```
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
```

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.

比较函数写成class比lambdas更直观

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

如果写成：
std::set<Person, decltype(cmp)> coll();但set():t(Compare()){}找不到lambdas的ctor

Boolan 博览

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;
```

inline函数，效率高，但c++中谈效率，还是得看algorithm

```
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()
        );
```


Boolan 博览

Lambdas

Windows Grep 2.3

File Edit Search View Options Window Help

] {'m *.*: 94 matches in 11 files. 10474 files searched. 0 files skipped | future (Matches only)

Name	T	Type	Folder	Mat...
future	T	檔案	D:\G4.9.2\4.9.2\include\c++	
ppltasks.h	T	C Header File	D:\vs2013\include	
future	T	檔案	D:\G4.8\libstdc++-v3\include\std	

Windows Grep Search Results

Fancy | File contents ✓ | File names ✓ | Line numbers ✓ | Whole line ✓ | Fixed Font ✓ | Match window: +/- 0 | 1✓ | 2 | 3 | 4 | 5 lines

D:\G4.9.2\4.9.2\include\c++\future

```
00322:  unique_lock<mutex> __lock(_M_mutex);
00323:  _M_cond.wait(__lock, [&] { return _M_ready(); });
00324:  return *_M_result;

00335:      return future_status::deferred;
00336:    if (_M_cond.wait_for(__lock, __rel, [&] { return _M_ready(); }))
00337:      {

00354:      return future_status::deferred;
00355:    if (_M_cond.wait_until(__lock, __abs, [&] { return _M_ready(); }))
00356:      {

01528:       {
01529:  _M_thread = std::thread( [this] {
01530:      __try
```

Boolan 博览

15. Variadic Templates 1

Variadic Templates

重回核爆區

- 談的是 template
  - function template
  - class template

    member template//不重要
- 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.

    variadic template本身不是递归，但利用其特性可实现递归
  - 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

Boolan 博览

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:

bitset<>是容器

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

```
7.5
hello
0000000101111001
42
```

此处模板函数是对的，但具体调用时，因传入参数不被cout认识，编译就会出错

overload

```
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
}
```

没有任何参数，所以也不需要模板

❷ ❶

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

虽然感觉ambig，但模板有特化

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.

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.

Boolan 博览

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++;
  }
}
```

输出控制符应该和输出字符个数一致，否则printf()会报错，这里也模拟了报错

stl有标准的异常可使用

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

```
15 This is Ace. 0x3e4ab8 3.14159
```

```
template<typename T, typename... Args>
void printf(const char* s, T value, Args... args)
{
  while (*s)  { 输出控制符没被使用，可否直接取输出字符，直接扔给cout ？
    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")
}
```

❶

❷

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

Boolan 博览

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;
```

```
// ...\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()); }
```

__l.begin(),其实就是array头部指针

```
// ...\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; }
};
```

按<方式进行比较

Boolan 博览

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. 即：例3

Boolan 博览

使用类模板对象分解类模板参数

Variadic Templates 例 5

以異於一般的方式處理 first 元素和 last 元素

```
// 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++,
… tuple成员函数

```
// 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);
    }
};
```

递归的创建

make_tuple()是方便创建tuple的函数，实际应该tuple<4个类型> obj

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

[7.5,hello,0000000101111001,42]

```
// 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)  {
    }
};
```

partial specialization????

Boolan 博览

Variadic Templates 例 6

用於遞歸繼承, recursive inheritance

前面是函数模板递归调用和类模板递归创建

```
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;
};
```

private：因为41和6.3等不能说谁是谁

因为在参数列表的位置
調用 base ctor 並予參數
(不要誤以為是創建 temp object)

注意這是
initialization list

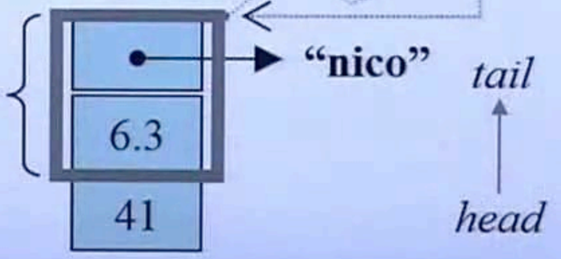
Head::type就是Head类型???
写typename更严谨

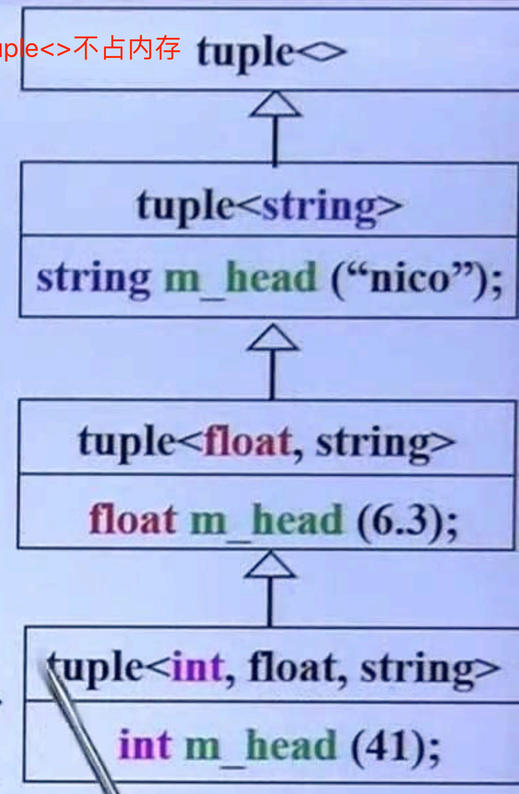
return 後
轉型為 inherited,
獲得的是

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

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()) ➔





Boolan 博览

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...> inherit
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;
};
```

Head::type编译不过，int/float/string回答不出

注意這是 initialization list

return 後 轉型為 inherited, 獲得的是

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)

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

tuple<>

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

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

<int, float, string>
(41);

Boolan 博览

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
```

Boolan 博览

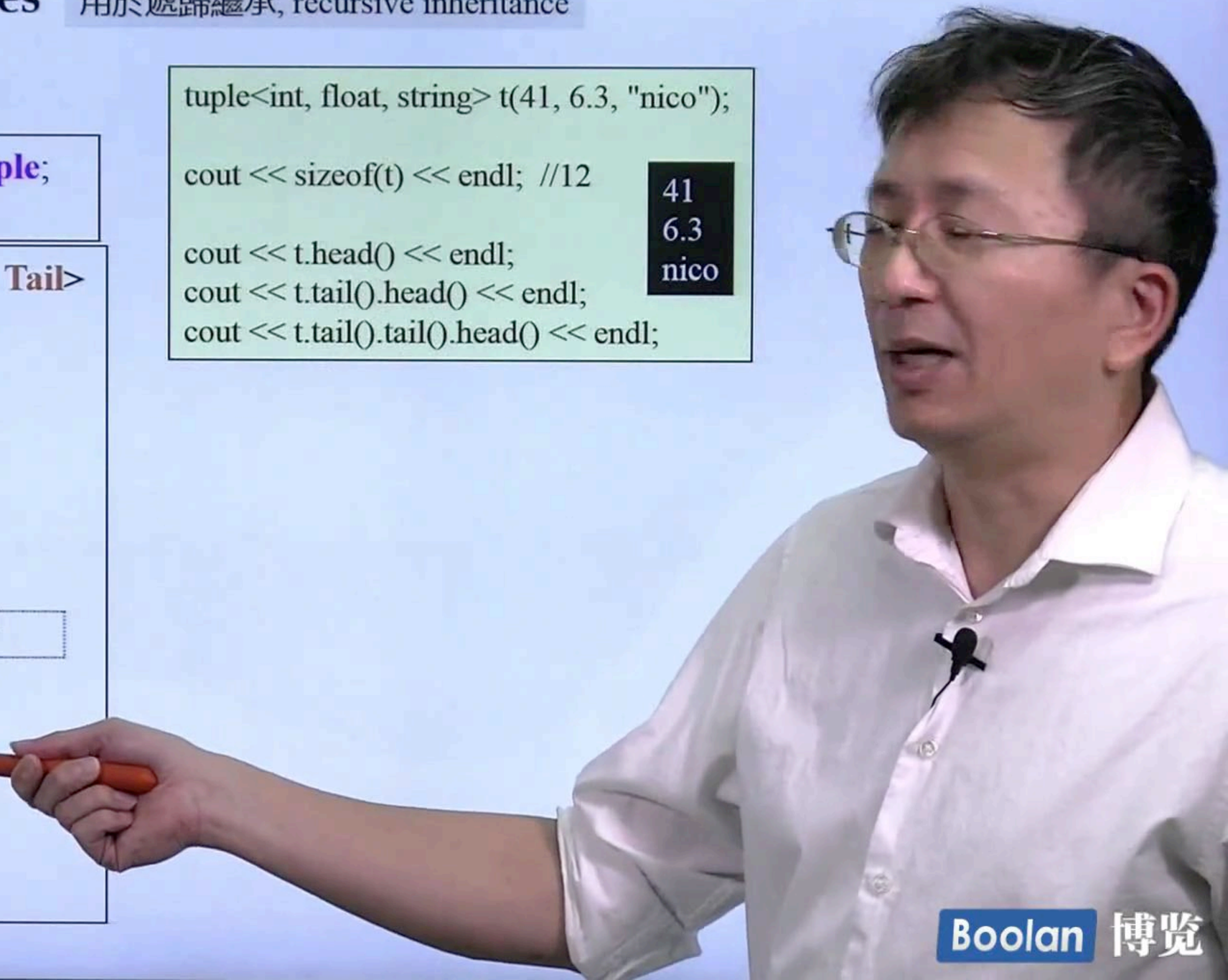
Variadic Templates 用於遞歸繼承, recursive inheritance

```
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
```



Boolan 博览

Variadic Templates 例 7 用於遞歸複合, recursive composition

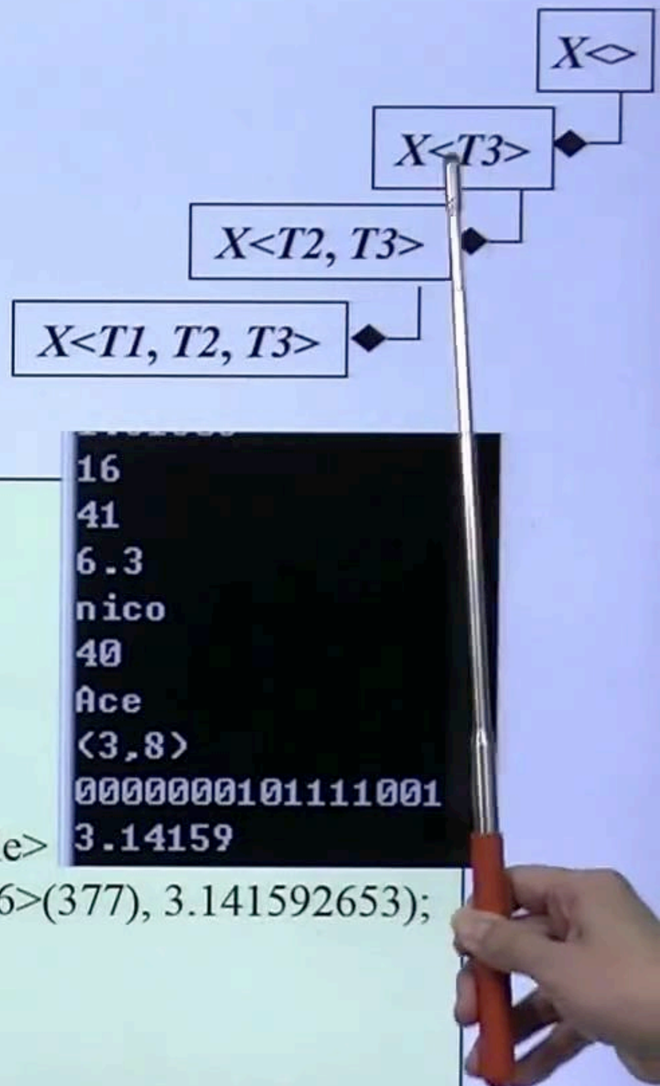
```
template<typename... Values> class tup;
template<> class tup<> { };
```

遞歸調用 處理的都是參數，
使用 function template

遞歸複合 處理的是類型 (type)，
使用 class 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; }
};
```

```
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;
```



Boolan 博览

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

Boolan 博览

The End

Boolan 博览

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

標準庫 源代碼分佈, ~~Visual C++~~

files layout

…\include

名稱
- cliext
- CodeAnalysis
- CodeCoverage
- cvt
- Manifest
- msclr
- sys
- thr
- agents
- agile
- algorithm
- allocators
- ammintrin
- amp
- amp_graphics
- amp_math
- amp_short_vectors
- amprt
- arm_neon
- armintr
- array
- assert
- atomic
- bitset
- cassert
- ccomplex
- cctype
- cerrno
- cfenv
- cfloat
- chrono
- cinttypes
- ciso646

名稱
- stack
- stdarg
- stdbool
- stddef
- stdexcept
- stdexcpt
- stdint
- stdio
- stdlib
- streambuf
- string
- string
- strstream
- swprintf.inl
- system_error
- tchar
- thread
- time
- time.inl
- tmmintrin
- tuple
- type_traits
- typeindex
- typeinfo
- typeinfo
- unordered_map
- unordered_set
- use_ansi
- utility
- vadefs
- valarray
- varargs
- vcclr
- vccorlib
- vector

…\include\cliext

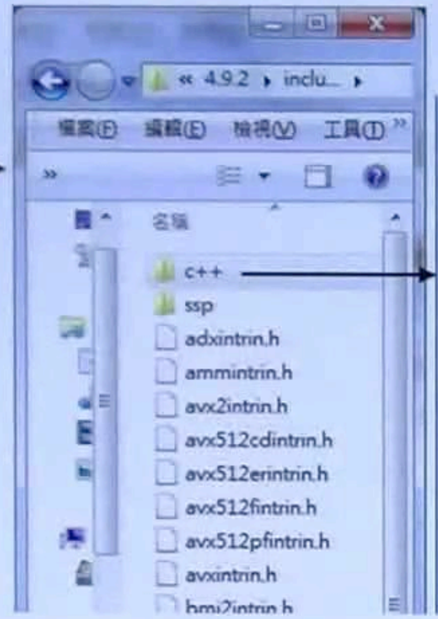
名稱
- adapter
- algorithm
- deque
- functional
- hash_map
- hash_set
- iterator
- list
- map
- memory
- numeric
- queue
- set
- stack
- utility
- vector
- xhash
- xtree
- xutility

Boolan 博览

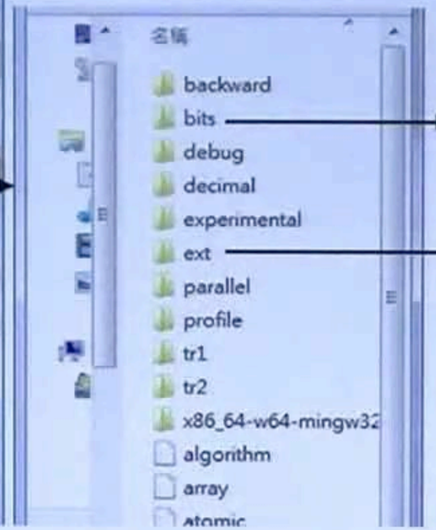
標準庫 源代碼分佈, GNU C++

files layout

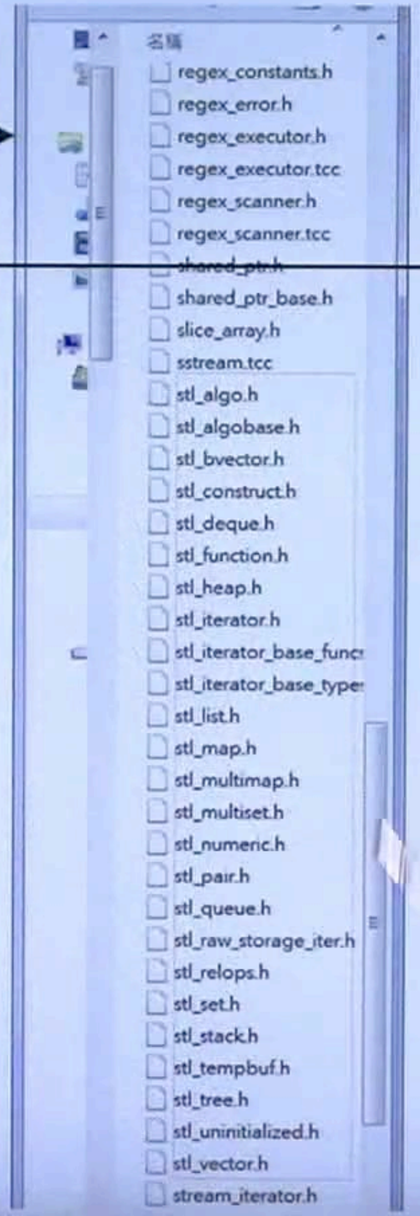
…\4.9.2\include



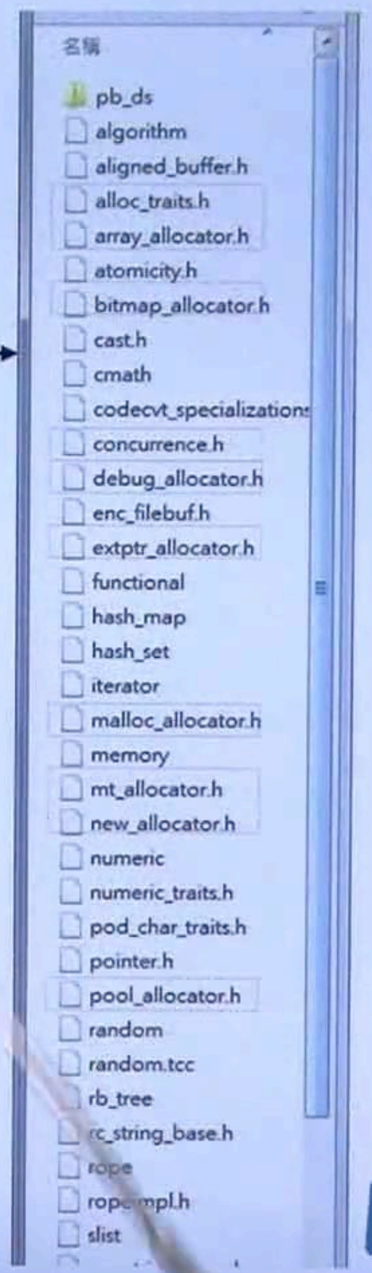
…\include\c++



…\include\c++\bits



…\include\c++\ext



Boolan 博览

Rvalue references

rvalue ref关系到move，而move对容器效率有改善
rvalue ref是语言部分，但其延伸和stl有关

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***.

生成临时对象放入容器，就是一次copy动作
copy动作可能非必要，rvalue ref就是为了解决非必要copy
容器自身可能默默做copy动作，eg>vector成长

当赋值右手边是一个右值时

而无需赋值左手边重新分配内存，要发生在指针上

c++自定义类让修改/赋值使人感觉大吃一惊，导致定义的不确定，定义指对int的理解

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

Lvalue：变量
Rvalue：不能放在左边的，右值没名称，无法赋值

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

string和complex推翻了int对rvalue的认知

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

a+b不能再被赋值，a+b是右值

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
```

临时对象是右值，这里是stl作者写的，没遵循int对右值的理解

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 ?可以
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
```

Boolan 博览

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)

函数返回值是右值，右值不可取地址
现在2.0新语法rvalue ref：可取出rvalue的ref

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。

Boolan 博览

Rvalue references and Move Semantics

容器可偷且对象让偷

❶ //copy
insert(…,&x)

❷ //move 2.0新语法: &&x
insert(…,&&x)

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

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

copy 指针深copy

打叉：右值被偷后不能再使用

move

如vector会向后推，推时可能会发生copy(除非放到末尾)
若发生copy，会调arg2类的ctor

temp object
一定被當作 Rvalue
编译器

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);
}
```

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

Vtype是MyString

临时对象是右值，要有一种语法去通知insert函数可steal临时对象中的数据，vector.insert()有两个
如果Vtype(buf)是rvalue ref，调2，以前调1

insert(Lvalue)且确定此Lvalue不再使用，用std::move()使左值变为Rvalue ref

必須確保後續不再使用 c1

Boolan 博览

Rvalue references and Move Semantics

文本
文本

如下是vector源码

G2.9 c++1.0版本

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

G4.9 c++2.0版本

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

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

move aware

Boolan 博览

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;
    }
...
```

1, Vtype(buf)
2, insert(..., &&x)
3, move ctor
1到3过程(forward转交)遗失了重要信息，不完美
左边：copy和move在ctor和assignment都有新版本

Boolan 博览

Perfect Forwarding

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));
}
```

Boolan 博览

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
```

unperfect forwarding

type traits

```
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);
}
```

此forward的设计，不支持左边三种调用
问题同前面的123调用过程遗失的信息

Boolan 博览

Perfect Forwarding

stl::forward做法

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

```
/**
 *  @brief  Forward an lvalue.
 *  @return The parameter cast to the specified type.
 *
 *  This function is used to implement "perfect forwarding".
 */
template<typename _Tp>
  constexpr _Tp&&
  forward(typename std::remove_reference<_Tp>::type& __t) noexcept
  { return static_cast<_Tp&&>(__t); }

/**
 *  @brief  Forward an rvalue.
 *  @return The parameter cast to the specified type.
 *
 *  This function is used to implement "perfect forwarding".
 */
template<typename _Tp>
  constexpr _Tp&&
  forward(typename std::remove_reference<_Tp>::type&& __t) noexcept
  {
    static_assert(!std::is_lvalue_reference<_Tp>::value, "template argument"
          " substituting _Tp is an lvalue reference type");
    return static_cast<_Tp&&>(__t);
  }
```

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

std::move

看代码时，会先调用std::forward再调用std::move

最终希望设计一个class，声明可被steal，这样对容器效率有提高

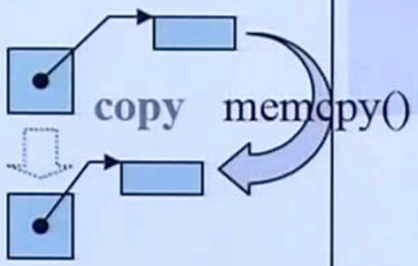
细节枯燥无味

Boolan 博览

寫一個 move aware class

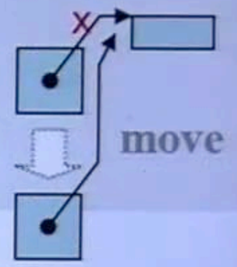
有move意识的class的对象，作为容器元素
若容器需要copy时，看到元素有move意识，就会用此move取代copy

```
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;  有指针成员，必写大三
    size_t _len;
    void _init_data(const char *s) {
        _data = new char[_len+1];
        memcpy(_data, s, _len);
        _data[_len] = '\0';
    }
public:
    //default constructor
    MyString() : _data(NULL),_len(0) { ++DCtor; }

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




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

    //move constructor, with "noexcept"
    MyString(MyString&& str) noexcept
      : _data(str._data),_len(str._len){
        ++MCtor;
        str._len = 0; 右值str被认为不再可用
        str._data = NULL; //重要
    } str._data = NULL;就是打叉，否则离开此行，会调用临时对象的析构

    //copy assignment
    MyString& operator=(const MyString& str) {
        ++CAsgn;
自  if (this != &str) {
我      if (_data) delete _data;
賦      _len = str._len;
值      _init_data(str._data); //COPY!
檢  }
查  else {
    }
    return *this;
    }
```

删除动作应该交给dtor
搬移之後棄而不用，但也别删除.



Boolan 博览

寫一個 move aware 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; //重要
    }
    return *this;
}
//dtor
virtual ~MyString() {
    ++Dtor;
    if (_data) {
        delete _data;
    }
}
bool //hash set/hash map有新名字
operator<(const MyString& rhs) const //為了 set
{
    return
        string(this->data)<string(rhs._data);
        //借用現成事實：std::string 能夠比較大小.
}
```

自我賦值檢查

x

move

destructor is **noexcept** by default.

用关联式容器(set/map/hash set/hash map)测试move aware class，都会检测元素大小，所以写了operator<和operator==

```
    bool
    operator==(const MyString& rhs) const //為了 set
    {
        return
            string(this->_data)==string(rhs._data);
            //借用現成事實：string 能夠判斷相等與否.
    }

    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;
```

定义并初始化静态成员

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

set背后是hash table，hash table需要hash funtion，这里定义hash function

Boolan 博览

move aware 測試

```
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 (轉換為字符串)
      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);
M c12(std::move(c1));
   c11.swap(c12);

2 //測試 non-moveable
...
}
```

關於 std::move

必須確保接下來不會再用到 c1

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

Boolan 博览

move aware 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
```

7194303>3000000? 因为vector会成长

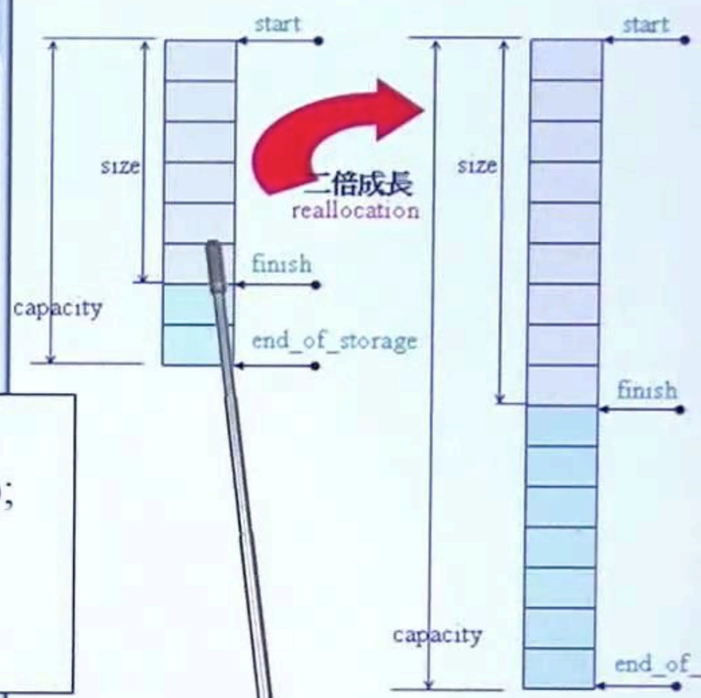
差別巨大

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);
```



Boolan 博览

move aware 元素對於 list 速度效能的影響

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

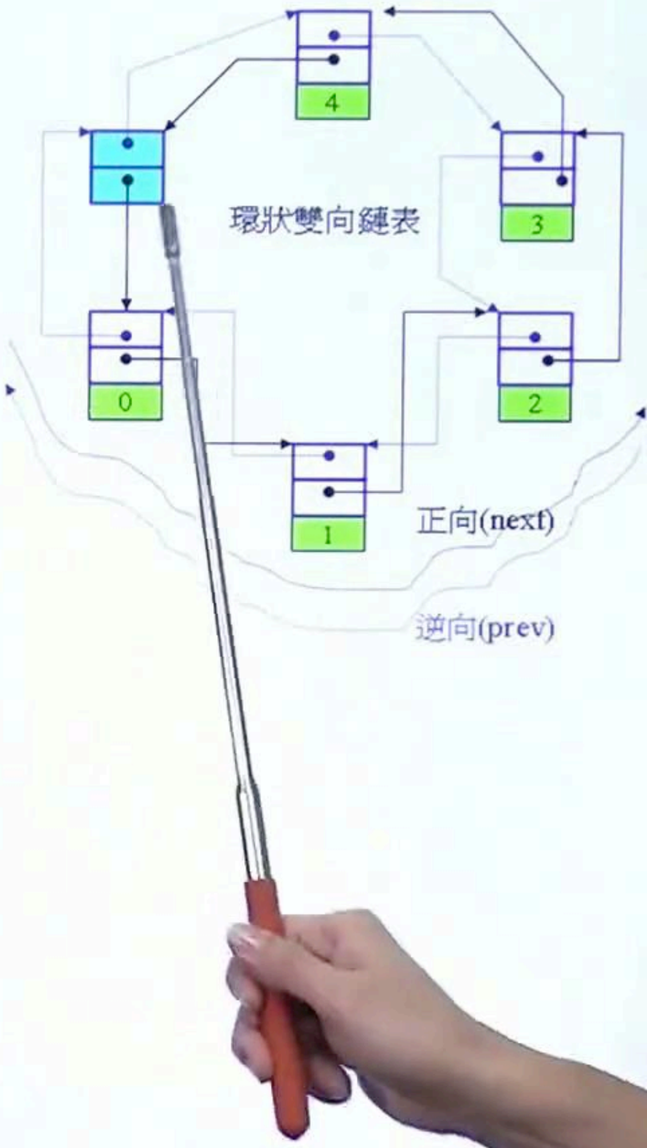
```
test, with moveable elements
construction, milli-seconds : 10765
size()= 3000000
8MyString --
 CCtor=0 MCtor=3000000 CAsgn=0 MAsgn=0 Dtor=3000006 Ctor=3000006 DCtor=0
copy, milli-seconds : 4188
move copy, milli-seconds : 0
swap, milli-seconds : 0


test, with non-moveable elements
construction, milli-seconds : 11016
size()= 3000000
11MyStrNoMove --
 CCtor=3000000 MCtor=0 CAsgn=0 MAsgn=0 Dtor=3000004 Ctor=3000004 DCtor=0
copy, milli-seconds : 3906
move copy, milli-seconds : 0
swap, milli-seconds : 0
```

差別巨大

差別不大

差別巨大

list每个节点都独立



Boolan 博览

move aware 元素對於 deque 速度效能的影響

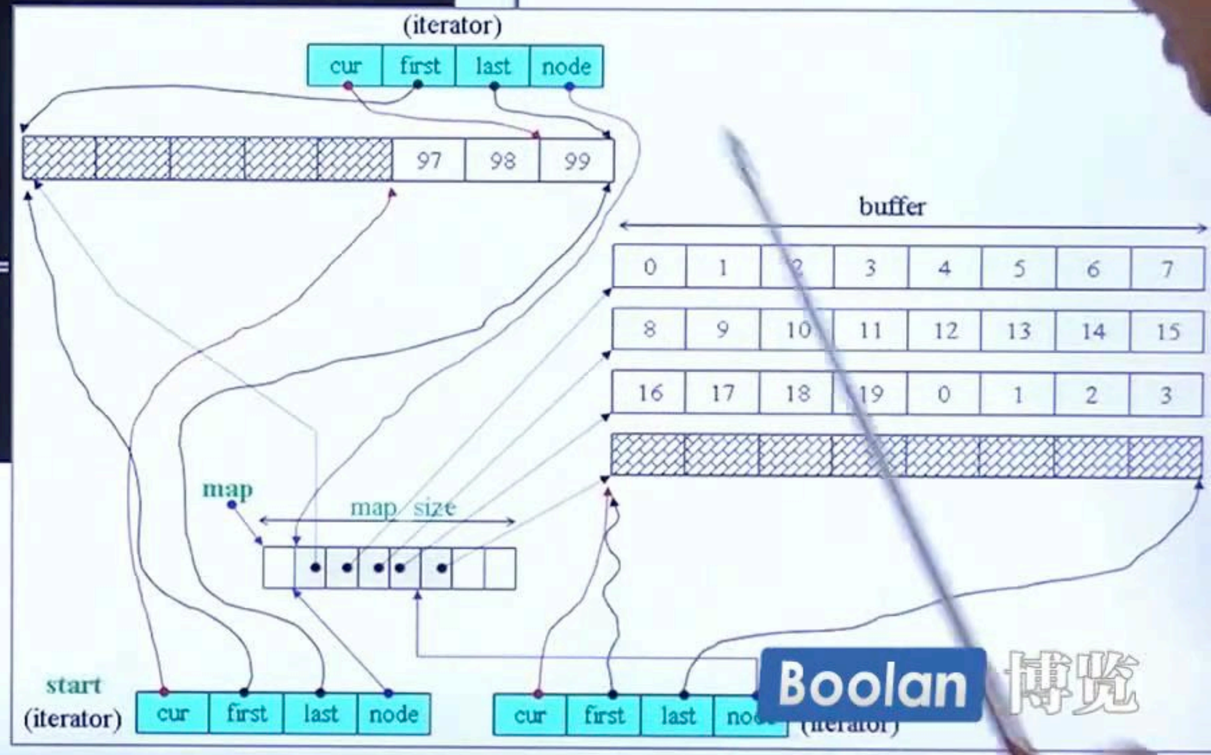
```
D:\handout\C++11-test-DevC++\Test-STL\test-stl.exe

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

deque分段连续的，测试是insert到尾部，
若插在2位置，deque会看离头尾哪个近，
就”推（copy）“哪部分



Boolan 博览

move aware 元素對於 multiset 速度效能的影響

multiset背后是rb树

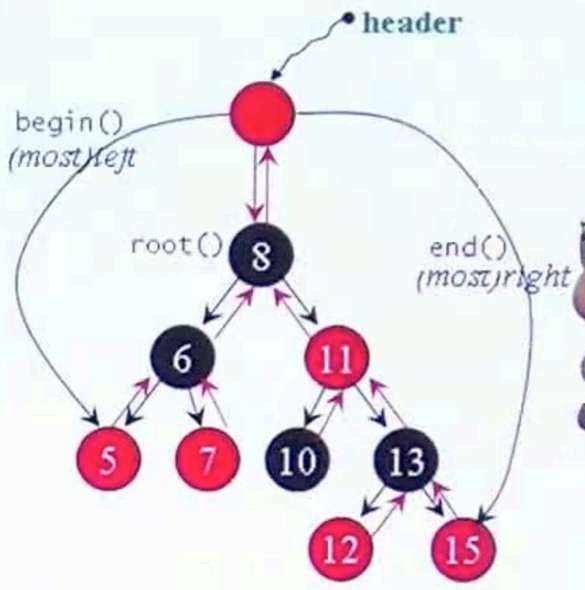
D:\handout\C++11-test-DevC++\Test-STL\test-stl.exe

```
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
```

差别巨大

差别不大

差别巨大



Boolan 博览

move aware 元素對於 unordered_multiset 速度效能的影響

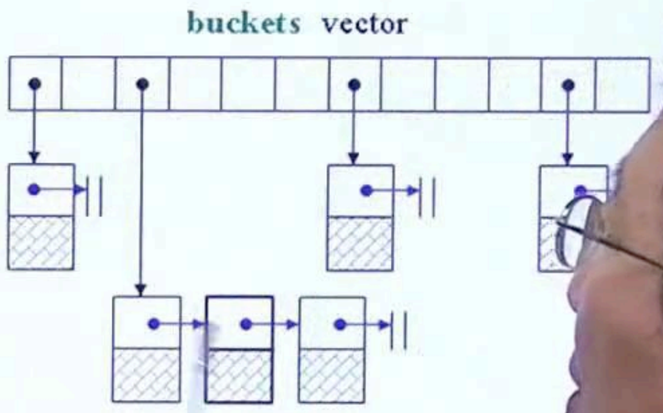
unordered_multiset背后是hash table

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

test, with moveable elements
construction, milli-seconds : 23891
size()= 3000000
8MyString --
 CCtor=0 MCtor=3000000 CAsgn=0 MAsgn=0 Dtor=3000006 Ctor=3000006 DCtor=0
copy, milli-seconds : 7812
move copy, milli-seconds : 0
swap, milli-seconds : 0


test, with non-moveable elements
construction, milli-seconds : 24672
size()= 3000000
11MyStrNoMove --
 CCtor=3000000 MCtor=0 CAsgn=0 MAsgn=0 Dtor=3000004 Ctor=3000004 DCtor=0
copy, milli-seconds : 7188
move copy, milli-seconds : 0
swap, milli-seconds : 0
```

差别巨大

差别不大

差别巨大



看起来，元素是否具有move只对vector有影响，包括deque
只要容器是以节点形式存在，那move/copy差不多

Boolan 博览

vector 自身的 copy ctor

G4.9

```
M c1(c);
M c2(std::move(c));
```

```
/**
 * @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.
 */
1 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());
  }
```

std::allocator<_Tp>

_Vector_base<_Tp>
- _M_impl : _Vector_impl

_Vector_impl<_Tp>
- _M_start : pointer
- _M_finish : pointer
- _M_end_of_storage : pointer

vector<_Tp>
- vector(__x : const vector&)
- vector(__x : vector&&)

std::__uninitialized_copy_a(...)

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

把x的头尾逐个copy到目的端，目的端需分配足够空间

Boolan 博览

vector 的 move ctor

G4.9

```
M c1(c);
M c2(std::move(c));
```

```
std::allocator<_Tp>
```

```
_Vector_base<_Tp>
  _M_impl : _Vector_impl
```

```
_Vector_impl<_Tp>
  _M_start : pointer
  _M_finish : pointer
  _M_end_of_storage : pointer
```

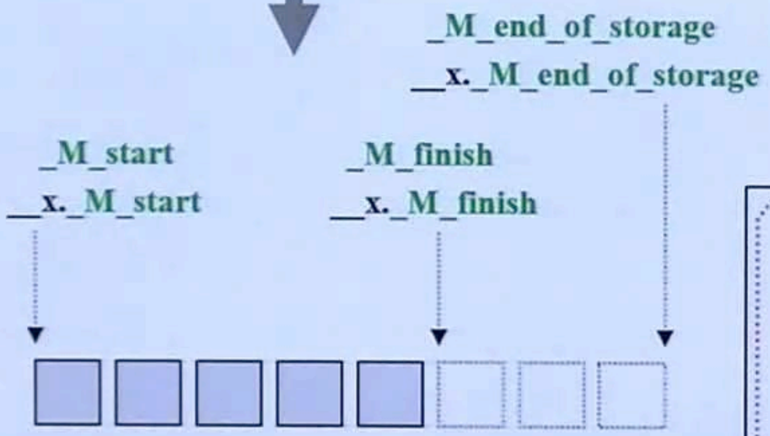
```
vector<_Tp>
  vector(__x : const vector&)
  vector(__x : vector&&)
```

```
void _M_swap_data(_Vector_impl& __x)
{
  std::swap(_M_start, __x._M_start);
  std::swap(_M_finish, __x._M_finish);
  std::swap(_M_end_of_storage,
            __x._M_end_of_storage);
}
```

只是交换了指针
因为右值，所以放心的偷

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。

```
_Vector_base(_Vector_base&& __x) noexcept
  : _M_impl(std::move(__x._M_get_Tp_allocator()))
  { this->_M_impl._M_swap_data(__x._M_impl); }
```



一个vector是由三个指针建立起来的

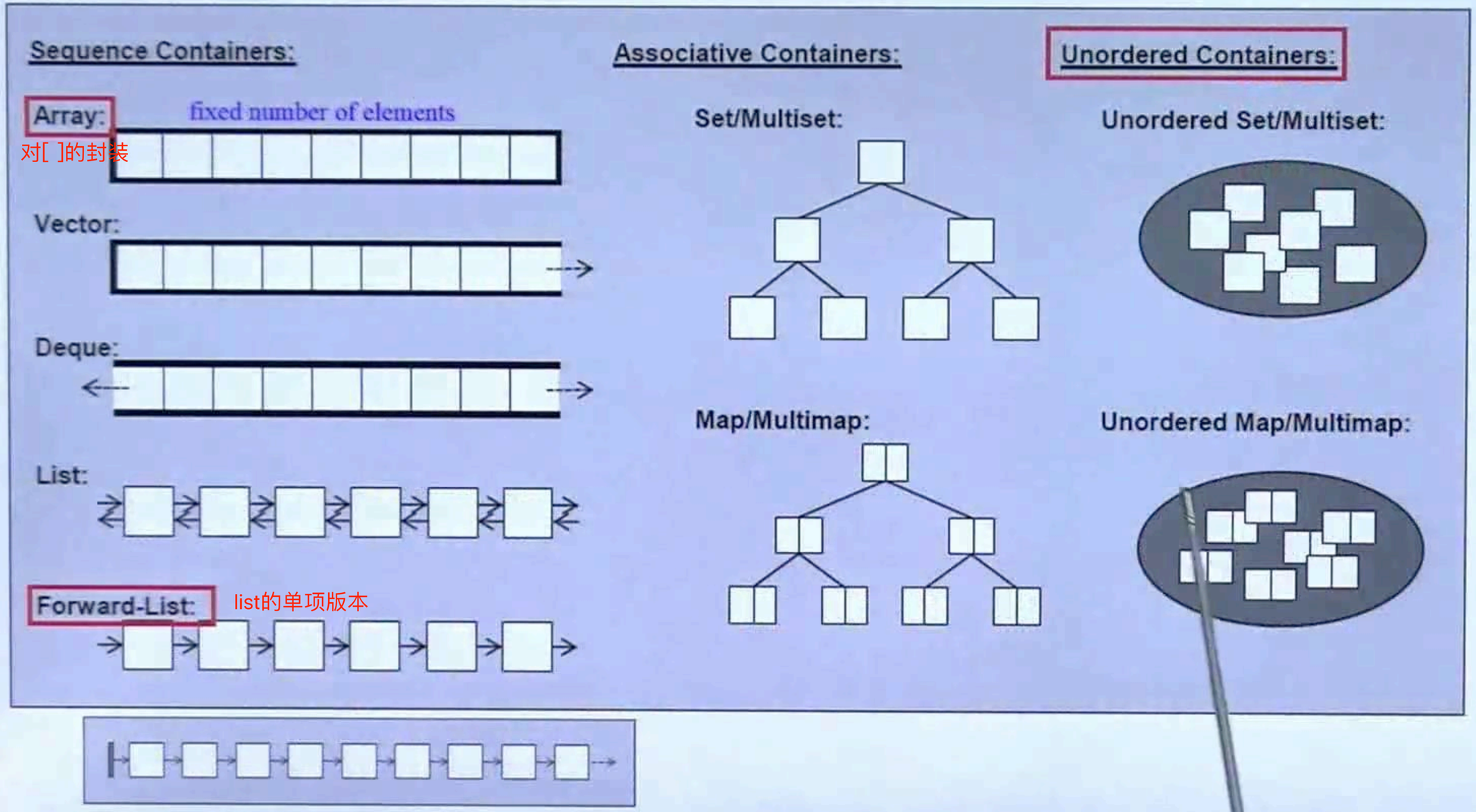
```
/**
 * @brief  %Vector move constructor.
 * @param  __x  A %vector of identical element and allocator types.
 *
 * The newly-created %vector contains the exact contents of @a __x.
 * The contents of @a __x are a valid, but unspecified %vector.
 */
vector(vector&&  __x) noexcept
  : _Base(std::move(__x)) { }
```

2

Boolan 博览

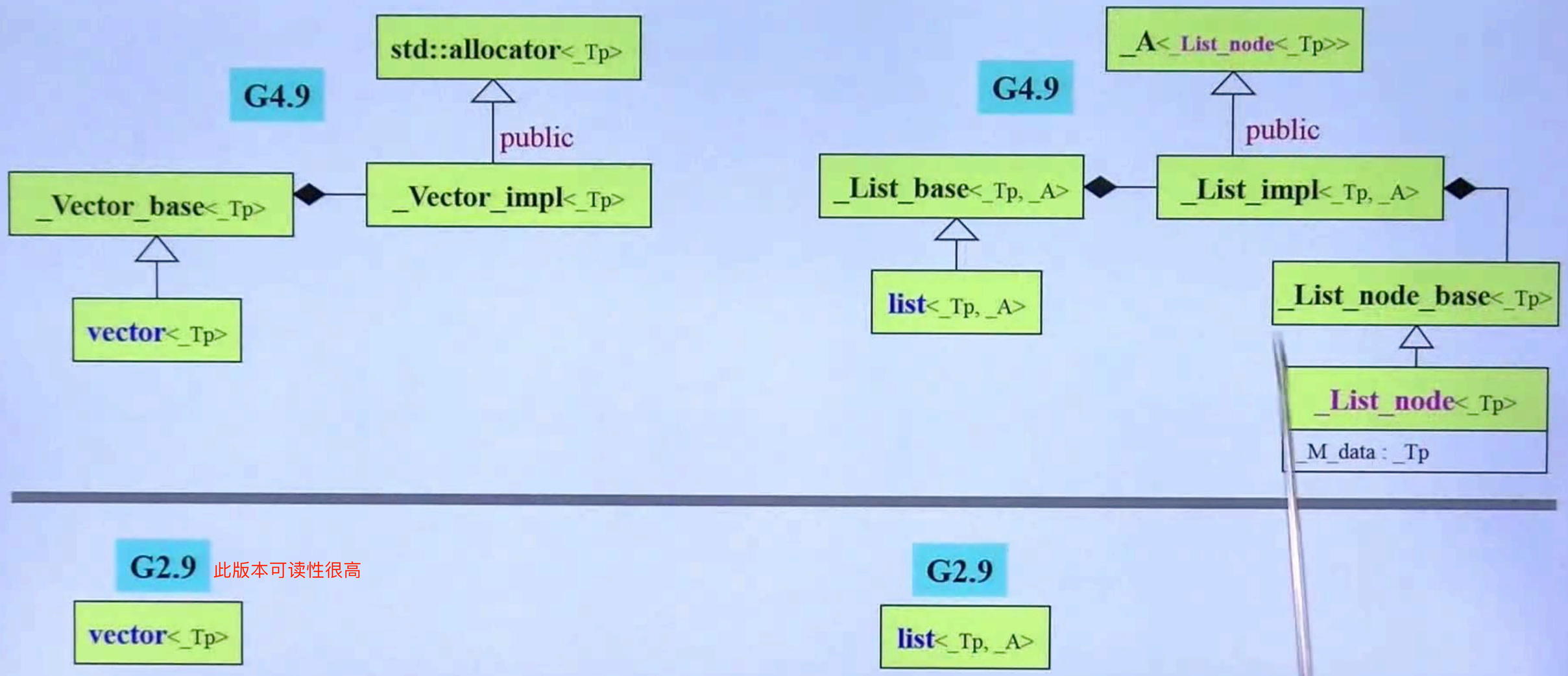
容器 – 結構與分類

红色新添加



Boolan 博览

舊與新的比較 — 關於實現手法



G2.9 此版本可读性很高

vector<_Tp>

G2.9

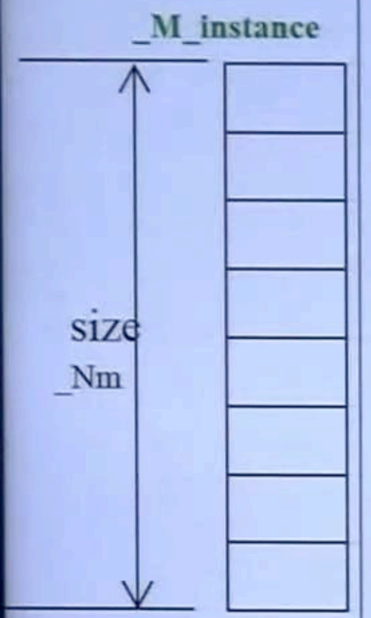
list<_Tp, _A>

Boolan 博览

容器 array

array<_Tp, _Nm>
_M_instance : _Tp[_Nm]

TR1




```
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,

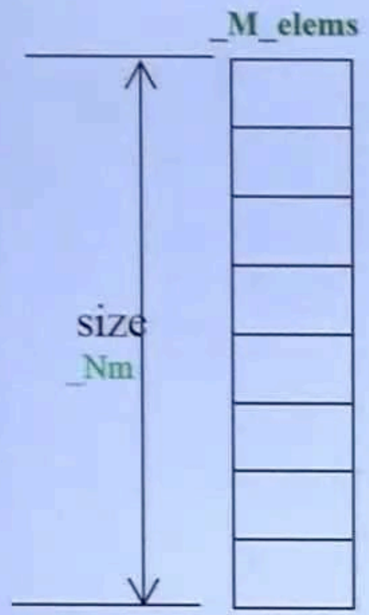
没有ctor，没有dtor 因为数组也没有

```
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
  {
    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, _Nm> _AT_Type;
    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_elems, 0)); }
    ...
};
```

```
int a[100];          //OK
int[100] b;          //fail
typedef int T[100];
T c;                 //OK
```

类型 变量

本行同 T c

没有邊檢

有邊檢

```
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]); }
  };
```

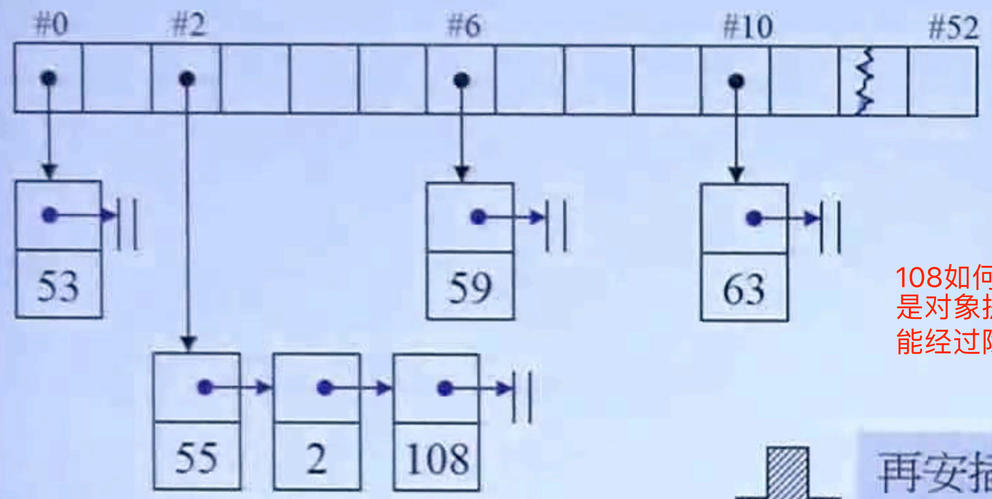
Boolan 博览

容器 hashtable

hash table/哈希表/散列表

模糊

unordered container背后是hash table
hash table在1.0就有，基于此有set & map
1.0也有hash set & hash map
hash table如下的实现版本叫separate chaining最好
separate chaining : 每个篮子里的指针指向单项(gnu c)/双向链表(vc)
篮子个数会成长，是vector实现的
元素个数多，则篮子个数会成长，否则每个篮子数据太多，查找慢
经验：元素个数多于篮子个数时，则篮子个数就增加2倍左右，每个元素位于哪个篮子需重新计算，叫rehashing
108/篮子个数=a，则108放到a篮子

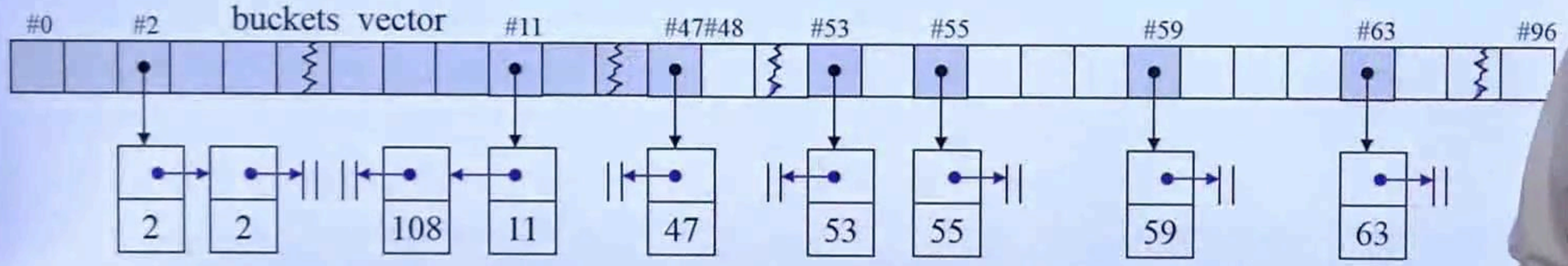


Separate Chaining

雖然 list 是線性搜尋時間，如果 list 夠小，搜尋速度仍然很快

108如何来?
是对象提供的函数计算得来，此函数对不同对象得到的值不同，否则会落在一个篮子（当然不同对象也可能经过除法计算而落在一个篮子），所以必须告诉hash table一个hash func去计算hash code

再安插 48 個元素，使總量達到 **54** 個，超過當時的 buckets vector 大小 **53**，於是 **rehashing**



Boolan 博览

unordered 容器

4个unordered container的声明

G4.9

默认hash func，Hash是模板参数，所以hash<T>是函数模板类型

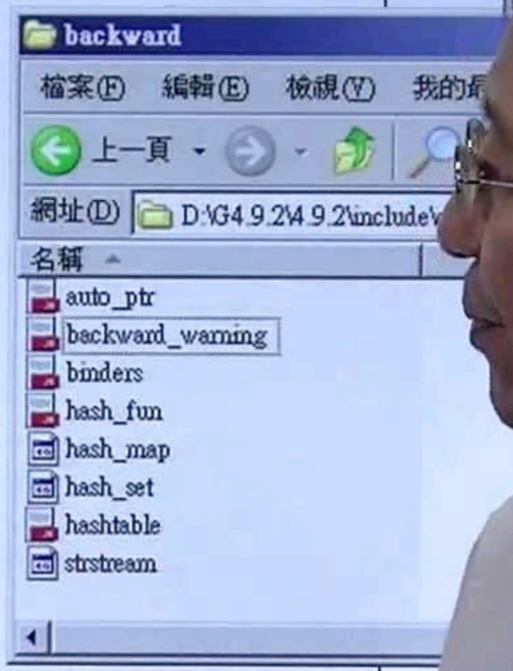
Before C++11

- hash_set
- hash_multiset
- hash_map
- hash_multimap

Since C++11

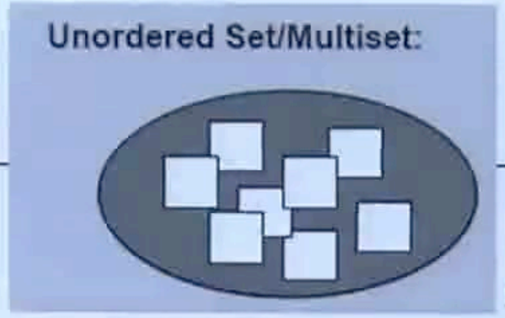
- unordered_set
- unordered_multiset
- unordered_map
- unordered_multimap

```
template <typename T,
          typename Hash = hash<T>,
          typename EqPred = equal_to<T>,
          typename Allocator = allocator<T>>
class unordered_set;

template <typename T,
          typename Hash = hash<T>,
          typename EqPred = equal_to<T>,
          typename Allocator = allocator<T>>
class unordered_multiset;

template <typename Key, typename T,
          typename Hash = hash<T>,
          typename EqPred = equal_to<T>,
          typename Allocator = allocator<pair<const Key, T>>>
class unordered_map;

template <typename Key, typename T,
          typename Hash = hash<T>,
          typename EqPred = equal_to<T>,
          typename Allocator = allocator<pair<const Key, T>>>
class unordered_multimap;
```

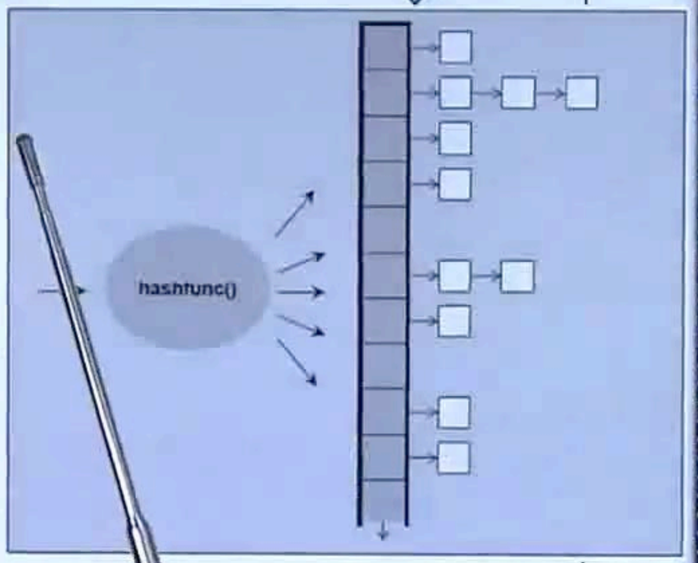


Boolan 博览

使用容器 unordered_set




```
namespace jj15
{
void test_unordered_set(long& value)
{
    cout << "\ntest_unordered_set()......... \n";

unordered_set<string> c;
char buf[10];

clock_t timeStart = clock();
    for(long i=0; i< value; ++i)
    {
        try {
            snprintf(buf, 10, "%d", rand());
            c.insert(string(buf));
        }
        catch(exception& p) {
            cout << "i=" << i << " " << p.what()
            abort();
        }
    }
    cout << "milli-seconds : " << (clock()-timeStart) << endl;  //
    cout << "unordered_set.size()= " << c.size() << endl;
    cout << "unordered_set.max_size()= " << c.max_size()
    cout << "unordered_set.bucket_count()= " << c.bucket_
    cout << "unordered_set.load_factor()= " << c.load_fact
    cout << "unordered_set.max_load_factor()= " << c.max_loa
    cout << "unordered_set.max_bucket_count()= " << c.max_bu
    for (unsigned i=0; i< 20; ++i) {
        cout << "bucket #" << i << " has " << c.bucket_size(i) << " elements.
    }
```

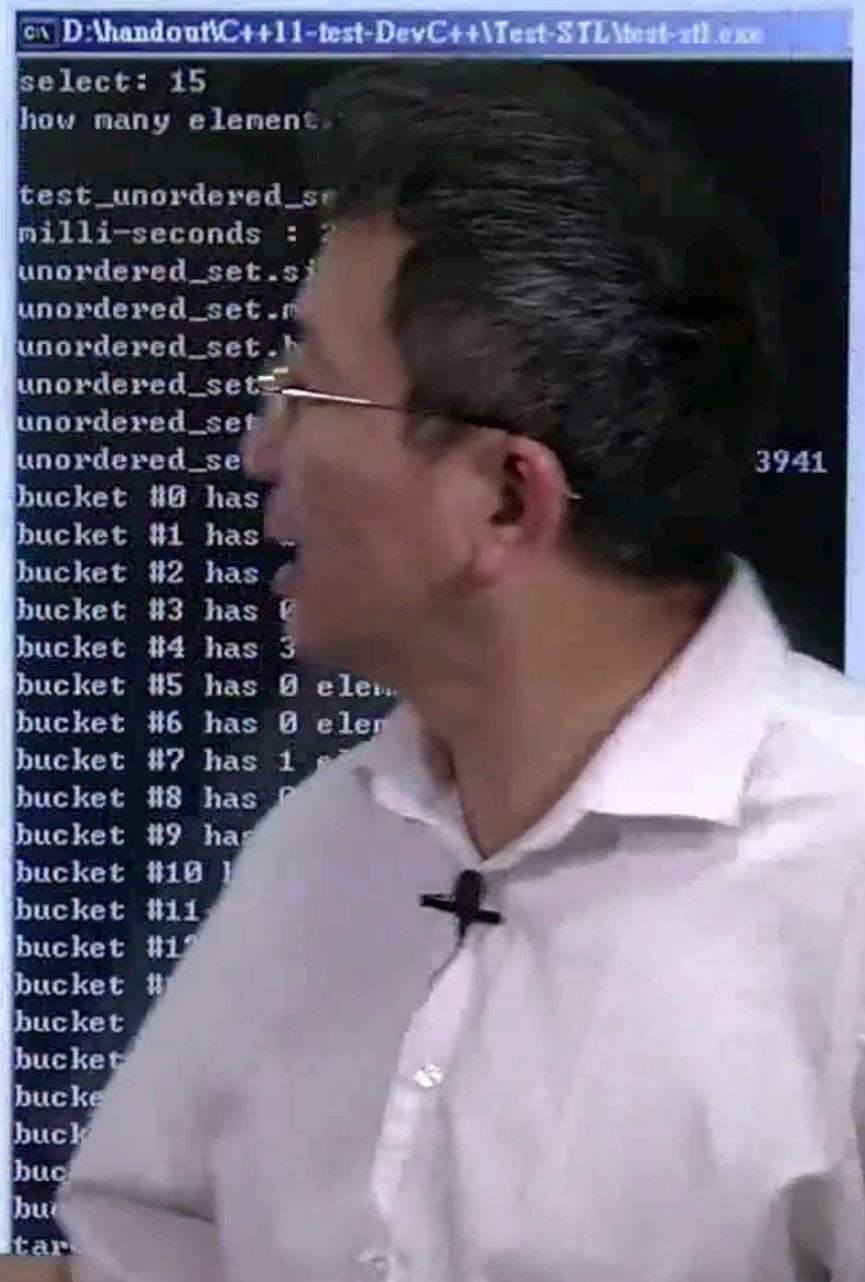



```
D:\handout\C++11-test-DevC++\Test-STL\test-stl.exe
select: 15
how many elements: 10
test_unordered_set()
milli-seconds : 2891
unordered_set.size()
unordered_set.max_s
unordered_set.bucke
unordered_set.load
unordered_set.max
unordered_set.ma
bucket #0 has 1 e
bucket #1 has 1 e
bucket #2 has 1 e
bucket #3 has 0 e
bucket #4 has 3 el
bucket #5 has 0 elem
bucket #6 has 0 element
bucket #7 has 1 elem
bucket #8 has 0 el
bucket #9 has 0
bucket #10 has
bucket #11 has
bucket #12 ha
bucket #13 h
bucket #14
bucket #15
bucket #1
bucket #
bucket #
bucket #
```

Boolan 博览

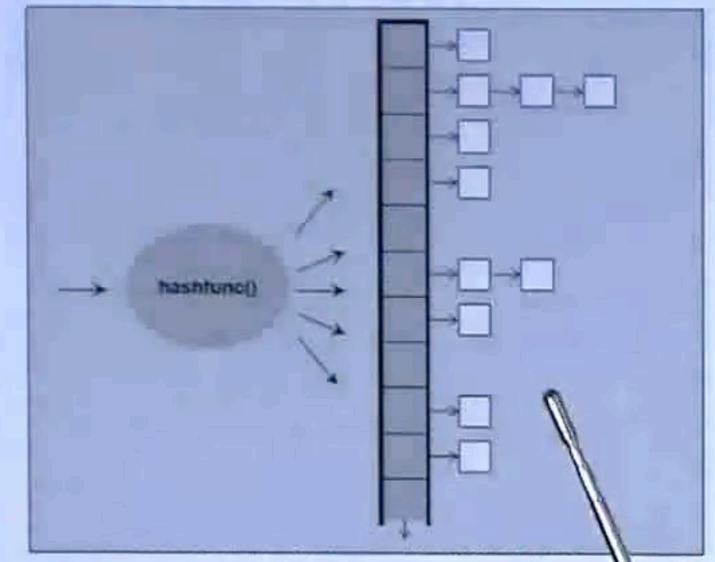
使用容器 unordered_set

```
770
771 string target = get_a_target_string();
772     {
773     timeStart = clock();
774 auto pItem = ::find(c.begin(), c.end(), target);    //比 c.find(...) 慢很多
775     cout << "::find(), milli-seconds : " << (clock()-timeStart) << endl;
776     if (pItem != c.end())
777         cout << "found, " << *pItem << endl;
778     else
779         cout << "not found! " << endl;
780     }
781
782     {
783     timeStart = clock();
784 auto pItem = c.find(target);          //比 ::find(...) 快很多
785     cout << "c.find(), milli-seconds : " << (clock()-timeStart) << endl;
786     if (pItem != c.end())
787         cout << "found, " << *pItem << endl;
788     else
789         cout << "not found! " << endl;
790     }
791 }
792 }
```



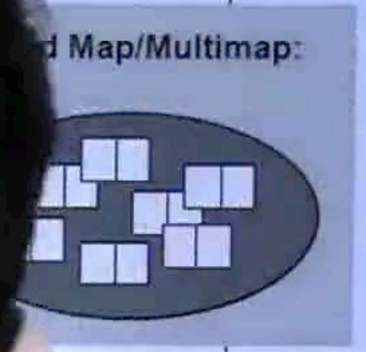
Boolan 博览

使用容器 unordered_map




```
D:\handout\C++11-test-DevC++\Test-STL\test-stl.e
select: 16
how many elements: 1000000

test_unordered_map().........
milli-seconds : 3797
unordered_map.size()= 1000000
unordered_map.max_size()= 35791394
target (0~32767): 23456
c.find(), milli-seconds : 0
found, value=12218
```

```
801 namespace jj16
802 {
803 void test_unordered_map(long& value)
804 {
805     cout << "\ntest_unordered_map()..
806
807 unordered_map<long, string> c;
808 char buf[10];
809
810 clock_t timeStart = clock();
811     for(long i=0; i< value; ++i)
812     {
813         try {
814             snprintf(buf, 10, "%d",
815             c[i] = string(buf);
816         }
817         catch(exception& p) {
818             cout << "i=" << i << " " <
819             abort();
820         }
821     }
822     cout << "milli-seconds : " << (clo
823     cout << "unordered_map.size()= "
824     cout << "unordered_map.max_size
825
826
827 long target = get_a_target_long(
828     timeStart = clock();
829 //! auto pItem = find(c.begin(
830 auto pItem = c.find(target);
831
832     cout << "c.find(), mill
833         Item != c.end())
834         "found, val
835
836
837 }
838 }
```



Boolan 博览

unordered 容器, 關於 hash functions

G4.9 基本类型的hash function

```
void* pi = (void*)(new int(100));
    cout << hash<int>()(123) << endl;                    //123
    cout << hash<long>()(123L) << endl;                  //123
    cout << hash<string>()(string("Ace")) << endl;       //1765813650
    cout << hash<const char*>()("Ace") << endl;          //5075396
    cout << hash<char>()('A') << endl;                   //65
    cout << hash<float>()(3.141592653) << endl;          //1294047319
    cout << hash<double>()(3.141592653) << endl;         //3456236477
    cout << hash<void*>()(pi) << endl;                   //4079456
```

创建临时函数对象，并调用

hash code够乱就行

0到127的ascii

g2.9没实现

如何给自定义类型去计算出他的很乱的hash code?
所有的数据/结构最终都是基本类型组成的，知道基本类型的hash code结算方式，再组合一下即可
可以在自定义类型中直接使用上述hash func，因为stl有一些hash func

2.0以前，不知道是gnu c的哪版，stl没提供一个hash func可以处理string
若放入108整数、则它的hash code就是108、应该很自然、推断：整形类型的值就是其hash code

Boolan 博览

hash-function, hash-code G2.9

泛化

```
template <class Key> struct hash { };
```

template<>

特化

```
__STL_TEMPLATE_NULL struct hash<char> {
  size_t operator()(char x) const { return x; }
};
__STL_TEMPLATE_NULL struct hash<short> {
  size_t operator()(short x) const { return x; }
};
__STL_TEMPLATE_NULL struct hash<unsigned short> {
  size_t operator()(unsigned short x) const { return x; }
};
__STL_TEMPLATE_NULL struct hash<int> {
  size_t operator()(int x) const { return x; }
};
__STL_TEMPLATE_NULL struct hash<unsigned int> {
  size_t operator()(unsigned int x) const { return x; }
};
__STL_TEMPLATE_NULL struct hash<long> {
  size_t operator()(long x) const { return x; }
};
__STL_TEMPLATE_NULL struct hash<unsigned long> {
  size_t operator()(unsigned long x) const { return x; }
};
```

```
hash<Foo>();
```

Foo的hash func也要有operator()

```
int i= hash<int>()(32);
```

Boolan 博览

hash-function, hash-code G2.9

```
inline size_t __stl_hash_string(const char* s)
{
  unsigned long h = 0;
  for ( ; *s; ++s)
    h = 5*h + *s;
    够乱就行
  return size_t(h);
}
__STL_TEMPLATE_NULL struct hash<char*>
{
  size_t operator()(const char* s) const { return __stl_hash_string(s); }
};
__STL_TEMPLATE_NULL struct hash<const char*>
{
  size_t operator()(const char* s) const { return __stl_hash_string(s); }
};
```

本例若 s 指向 "abc"，
計算得 h 為
5 * (5 * 'a' + 'b') + 'c'

注意，G2.9 沒有提供
hash<std::string>
G4.9有提供，在
...\4.9.2\include\c++\bits\basic_string.h

hash function 的目的，就是希望根據元素值算出一個 hash code（一個可進行 modulus 運算的值），使得元素經 hash code 映射之後能夠「夠雜夠亂夠隨機」地被置於 hashtable 內。愈是雜亂，愈不容易發生碰撞。

Boolan 博览

hash-function, hash-code G2.9

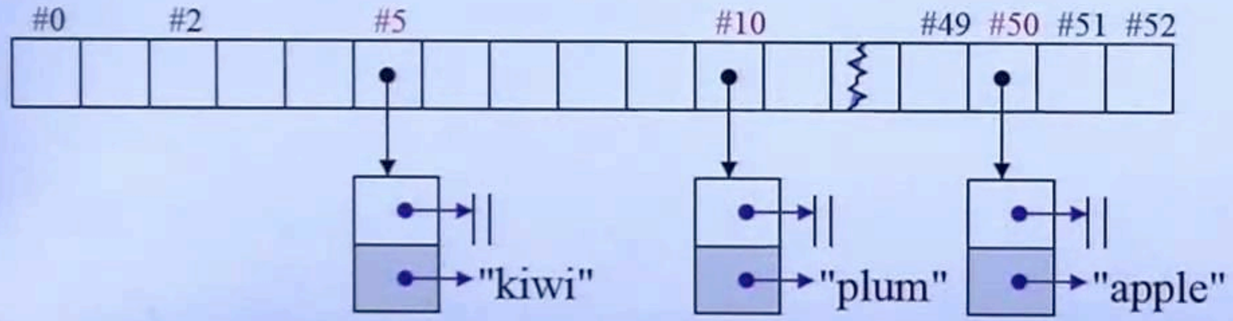
```
size_t hcode1= hash<char*>()("kiwi");
size_t hcode2= hash<char*>()("plum");
size_t hcode3= hash<char*>()("apple");
```

```
hash<char*> hashFuncforCstr;
hashFuncforCstr("kiwi");
hashFuncforCstr("plum");
hashFuncforCstr("apple");
```

hash code of "**kiwi**" = 5 * (5 * (5 * 'k' + 'i') + 'w') + 'i'
= 5 * (5 * (5 * 107 + 105) + 119) + 105
= 16700 % 53 = 5

hash code of "**plum**" = 5 * (5 * (5 * 'p' + 'l') + 'u') + 'm'
= 5 * (5 * (5 * 112 + 108) + 117) + 109
= 17394 % 53 = 10

hash code of "**apple**" = 5 * (5 * (5 * (5 * 'a' + 'p') + 'p') + 'l')+'e'
= 5 * (5 * (5 * (5 * 97 + 112) + 112) + 108)+101
= 78066 % 53 = 50




```
inline size_t
__stl_hash_string(const char* s)
{
  unsigned long h = 0;
  for ( ; *s; ++s)
    h = 5*h + *s;
  return size_t(h);
}
```

Boolan 博览

g4.9的hash func设计

hash-function, hash-code G4.9

```
49  template<typename _Result, typename _Arg>
50    struct __hash_base
51    {
52      typedef _Result      result_type;
53      typedef _Arg       argument_type;
54    };
55
56  /// Primary class template hash.
57  template<typename _Tp>
58    struct hash;
59
60  /// Partial specializations for pointer types.
61  template<typename _Tp>
62    struct hash<_Tp*> : public __hash_base<size_t, _Tp*>
63    {
64      size_t
65      operator()(_Tp* __p) const noexcept
66      { return reinterpret_cast<size_t>(__p); }
67    };
```

相當於 adaptable functor 所需要的 **unary_function<T>**

早期版本的空hash

```
template <class Arg, class Result>
struct unary_function {
    typedef Arg argument_type;
    typedef Result result_type;
};
```

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

```
69    // Explicit specializations for integer types.
70  #define _Cxx_hashtable_define_trivial_hash(_Tp)      \
71    template<>                                         \
72      struct hash<_Tp> : public __hash_base<size_t
73      {
74        size_t
75        operator()(_Tp __val) const noexcept
76        { return static_cast<size_t>(__val); }
77      };
78
79    /// Explicit specialization for bool.
80    _Cxx_hashtable_define_trivial_hash(bool)
81
82    /// Explicit specialization for char.
83    _Cxx_hashtable_define_trivial_hash(char)
84
85    /// Explicit specialization for signed char.
86    _Cxx_hashtable_define_trivial_hash(signed char)
87
88    /// Explicit specialization for unsigned char.
89    _Cxx_hashtable_define_trivial_hash(unsigned char)
90
91    /// Explicit specialization for wchar_t.
92    _Cxx_hashtable_define_trivial_hash(wchar_t)
93
94    /// Explicit specialization for char16_t.
95    _Cxx_hashtable_define_trivial_hash(char16_t
```

Boolan 博览

hash-function, hash-code

G4.9 ...\4.9.2\include\c++\bits\functional_hash.h

```
97   /// Explicit specialization for char32_t.
98   _Cxx_hashtable_define_trivial_hash(char32_t)
99
100  /// Explicit specialization for short.
101  _Cxx_hashtable_define_trivial_hash(short)
102
103  /// Explicit specialization for int.
104  _Cxx_hashtable_define_trivial_hash(int)
105
106  /// Explicit specialization for long.
107  _Cxx_hashtable_define_trivial_hash(long)
108
109  /// Explicit specialization for long long.
110  _Cxx_hashtable_define_trivial_hash(long long)
111
112  /// Explicit specialization for unsigned short.
113  _Cxx_hashtable_define_trivial_hash(unsigned short)
114
115  /// Explicit specialization for unsigned int.
116  _Cxx_hashtable_define_trivial_hash(unsigned int)
117
118  /// Explicit specialization for unsigned long.
119  _Cxx_hashtable_define_trivial_hash(unsigned long)
120
121  /// Explicit specialization for unsigned long long.
122  _Cxx_hashtable_define_trivial_hash(unsigned long long)
123
124 #undef _Cxx_hashtable_define_trivial_hash
```

```
126  struct _Hash_impl
127  {
128    static size_t
129    hash(const void* __ptr, size_t __clength,
130     size_t __seed = static_cast<size_t>(0xc70f6907UL))
131    { return _Hash_bytes(__ptr, __clength, __seed); }
132
133    template<typename _Tp>
134      static size_t
135      hash(const _Tp& __val)
136      { return hash(&__val, sizeof(__val)); }
137
138    template<typename _Tp>
139      static size_t
140      __hash_combine(const _Tp& __val, size_t __hash)
141      { return hash(&__val, sizeof(__val), __hash); }
142  };
143
144  struct _Fnv_hash_impl
145  {
146    static size_t
147    hash(const void* __ptr, size_t __clength,
148     size_t __seed = static_cast<size_t>(2166136261UL))
149    { return _Fnv_hash_bytes(__ptr, __clength, __seed); }
150
151    template<typename _Tp>
152      static size_t
153      hash(const _Tp& __val)
154      { return hash(&__val, sizeof(__val)); }
155
156    template<typename _Tp>
157      static size_t
158      __hash_combine(const _Tp& __val,
159      { return hash(&__val, sizeof(__va
160  };
```

見...\4.9.2\include\c++\bits\hash_bytes.h
但卻無定義, 只有聲明.

上下對比
惟此不同

Boolan 博览

hash-function, hash-code G4.9

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

```
162   /// Specialization for float.
163   template<>
164     struct hash<float> : public __hash_base<size_t, float>
165     {
166       size_t
167       operator()(float __val) const noexcept
168       {
169     // 0 and -0 both hash to zero.
170     return __val != 0.0f ? std::_Hash_impl::hash(__val) : 0;
171       }
172     };
173
174   /// Specialization for double.
175   template<>
176     struct hash<double> : public __hash_base<size_t, double>
177     {
178       size_t
179       operator()(double __val) const noexcept
180       {
181     // 0 and -0 both hash to zero.
182     return __val != 0.0 ? std::_Hash_impl::hash(__val) : 0;
183       }
184     };
185
186   /// Specialization for long double.
187   template<>
188     struct hash<long double>
189     : public __hash_base<size_t, long double>
190     {
191       _GLIBCXX_PURE size_t
192       operator()(long double __val) const noexcept;
193     };
```

```
195   // @} group hashes
196
197   // Hint about performance                    If not fast t
198   // containers will cache
199   // Default behavior is to
200   // otherwise.
201   template<typename _Hash>
202     struct __is_fast_hash : public std::true_type
203     { };
204
205   template<>
206     struct __is_fast_hash<hash<long double>> : public
207     { };
```

Boolan 博览

hash-function, hash-code G4.9

string应该自己实现hash func

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

```
/// std::hash specialization for string.
template<>
  struct hash<string>
  : public __hash_base<size_t, string>
  {
    size_t
    operator()(const string& __s) const noexcept
    { return std::_Hash_impl::hash(__s.data(), __s.length()); }
  };
```

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

```
126   struct _Hash_impl
127   {
128     static size_t
129     hash(const void* __ptr, size_t __clength,
130      size_t __seed = static_cast<size_t>(0xc70f6907UL))
131     { return _Hash_bytes(__ptr, __clength, __seed); }
132
133     template<typename _Tp>
134       static size_t
135       hash(const _Tp& __val)
136       { return hash(&__val, sizeof(__val)); }
137
138     template<typename _Tp>
139       static size_t
140       __hash_combine(const _Tp& __val, size_t __hash)
141       { return hash(&__val, sizeof(__val), __hash); }
142   };
```

Boolan 博览

hash-function, hash-code G4.9

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

```
#pragma GCC system_header

#include <bits/c++config.h>

namespace std
{
_GLIBCXX_BEGIN_NAMESPACE_VERSION

  // Hash function implementation for the nontrivial specialization.
  // All of them are based on a primitive that hashes a pointer to a
  // byte array. The actual hash algorithm is not guaranteed to stay
  // the same from release to release -- it may be updated or tuned to
  // improve hash quality or speed.
  size_t
  _Hash_bytes(const void* __ptr, size_t __len, size_t __seed);

  // A similar hash primitive, using the FNV hash algorithm. This
  // algorithm is guaranteed to stay the same from release to release.
  // (although it might not produce the same values on different
  // machines.)
  size_t
  _Fnv_hash_bytes(const void* __ptr, size_t __len, size_t __seed);

_GLIBCXX_END_NAMESPACE_VERSION
} // namespace
```

無定義,
只有聲明.

Boolan 博览

一個萬用的 Hash Function

自定义类型该怎么写hash func?
把自定义类型分解成若干基本类型，使用varadic template逐个处理，最后用黄金分割比例做整合

這裡該怎麼做呢？

```
#include <functional>
class Customer {
...
};

class CustomerHash
{
public:
    std::size_t operator()(const Customer& c) const {
        return ...
    }
};

unordered_set<Customer,CustomerHash> custset;
```

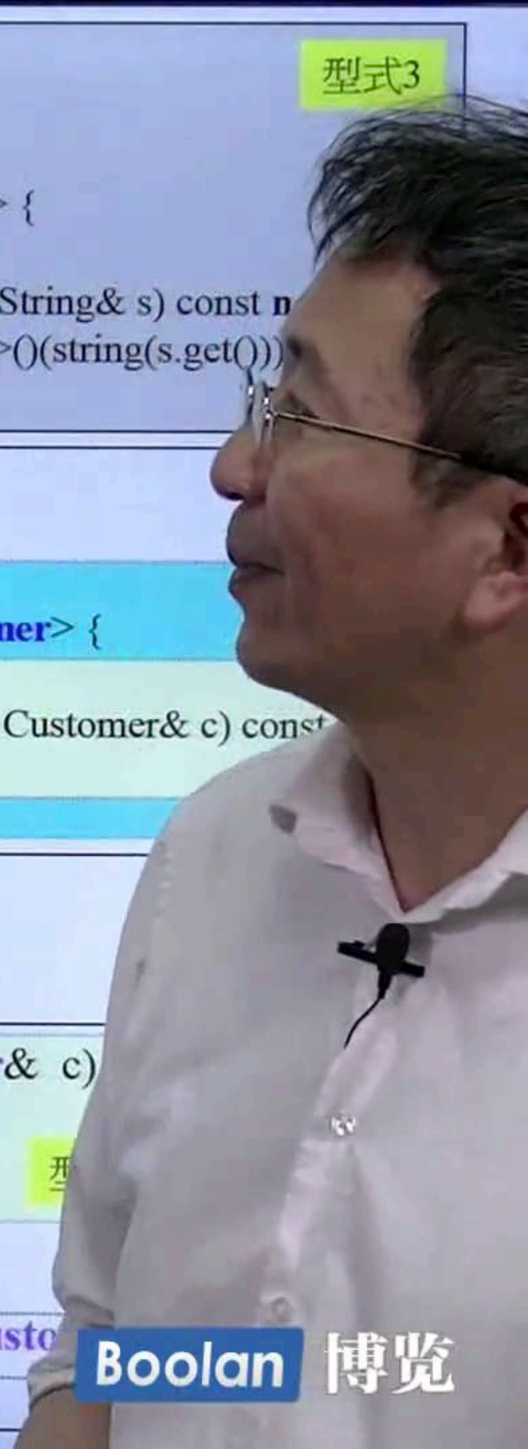
型式1

```
size_t customer_hash_func(const Customer& c)
    return ...
};

unordered_set<Customer, size_t(*)(const Custo
custset(20,customer_hash_func);
```

```
namespace std
{
template<>
struct hash<MyString> {
    size_t
    operator()(const MyString& s) const n
    { return hash<string>()(string(s.get()))
};
```

型式3

```
namespace std
{
template<>
struct hash<Customer> {
    size_t
    operator()(const Customer& c) const
    { return XXX; }
};
```



Boolan 博览

一個萬用的 Hash Function

```
//a naive approach : simply add all hash values for
//those attributes that are relevant for the hash function.
class CustomerHash {
public:
    std::size_t operator() (const Customer& c) const {
        return std::hash<std::string>()(c.fname) +
               std::hash<std::string>()(c.lname) +
               std::hash<long>()(c.no);
    }
};
```

太過天真 ✖

using **variadic templates** allows calling hash_val() with an arbitrary number of elements of any type to process a hash value out of all these values

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

```
#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

```
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...);
}
```

recursive

逐一取 val
改變 seed
(pass by reference)

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

seed 最終就被視為 hash value

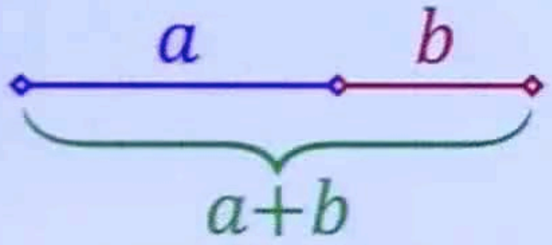
Boolan 博览

一個萬用的 Hash Function

黃金比例，又稱黃金比，是一種數學上的比例關係。黃金分割具有嚴格的比例性、藝術性、和諧性，蘊藏著豐富的美學價值。應用時一般取0.618或1.618 ，就像圓周率在應用時取3.14一樣。黃金分割早存在於大自然中，呈現於不少動物和植物外觀。現今很多工業產品、電子產品、建築物或藝術品均普遍應用黃金分割，呈現其功能性與美觀性。

常用希臘字母 φ 表示黃金比值，用代數式表達就是：

$$\frac{a+b}{a} = \frac{a}{b} \equiv \varphi$$



$a+b$ 比 a 等于 a 比 b

0x9e3779b9 / 0x7FFFFFFF = 0.618033.....
http://en.wikipedia.org/wiki/Golden_ratio

List of numbers · Irrational and suspected irrational numbers $\gamma \cdot \zeta(3) \cdot \sqrt{2} \cdot \sqrt{3} \cdot \sqrt{5} \cdot \varphi \cdot \rho \cdot \delta_S \cdot e \cdot \pi \cdot \delta$	
Binary	1.1001111000110111011...
Decimal	1.6180339887498948482...
Hexadecimal	1.9E3779B97F4A7C15F39...
Continued fraction	$1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\ddots}}}}$
Algebraic form	$\frac{1+\sqrt{5}}{2}$
Infinite series	$\frac{13}{8}+\sum_{n=0}^{\infty}\frac{(-1)^{(n+1)}(2n+1)!}{(n+2)!n!4^{(2n+3)}}$

Boolan 博览

一個萬用的 Hash Function

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

//auxiliary generic functions to create a hash value using a seed
template <typename T>
inline void hash_val (size_t& seed, const T& val)
{
    hash_combine(seed, val);
}

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;
}

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

Boolan 博览

tuple, 用例

tuple ： 指定任意模板参数，tuple可递归继承出很多类

```
cout << "string, sizeof = " << sizeof(string) << endl;      //4
cout << "double, sizeof = " << sizeof(double) << endl;      //8
cout << "float, sizeof = " << sizeof(float) << endl;        //4
cout << "int, sizeof = " << sizeof(int) << endl;            //4
cout << "complex<double>, sizeof = "
     << sizeof(complex<double>) << endl;                    //16
//tuples
// create a four-element tuple
// - elements are initialized with default value (0 for fundamental types)
tuple<string, int, int, complex<double>> t;
cout << "sizeof = " << sizeof(t) << endl;      //32, why not 28?

// create and initialize a tuple explicitly
tuple<int, float, string> t1(41, 6.3, "nico");
cout << "tuple<int, float, string>, sizeof = " << sizeof(t1) << endl;  //12
// iterate over elements:
cout << "t1: " << get<0>(t1) << ' ' << get<1>(t1) << ' ' << get<2>(t1) << e

// create tuple with make_tuple()
auto t2 = make_tuple(22, 44, "stacy");

// assign second value in t2 to t1
get<1>(t1) = get<1>(t2);
```

可能是编译器的设置，没有设置alignment，可能默认是8位对其

类型要一样

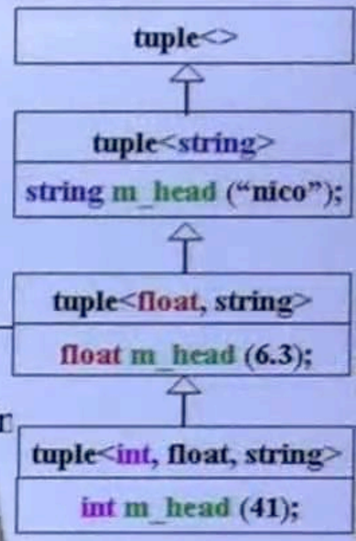
右边都是对类型在操作

meta programming

```
// comparison and assignment
// - including type conversion from tuple<ir
//   to tuple<int,float,string>
if (t1 < t2)  {   // compares value for valu
    cout << "t1 < t2" << endl;
} else {
    cout << "t1 >= t2" << endl;
}
t1 = t2;    // OK, assigns value for value
cout << t1 << endl;

tuple<int, float, string> t3(77, 1.1, "more light");
int i1;
float f1;
string s1;
tie(i1,f1,s1) = t3; //assigns values of t to i,f,and s

typedef tuple<int, float, string> TupleType;
cout << tuple_size<TupleType>::value << endl;    // 3
tuple_element<1,TupleType>::type f1 = 1.0;    // yields float
typedef tuple_element<1,TupleType>::type T;
```

一个一个元素去比较

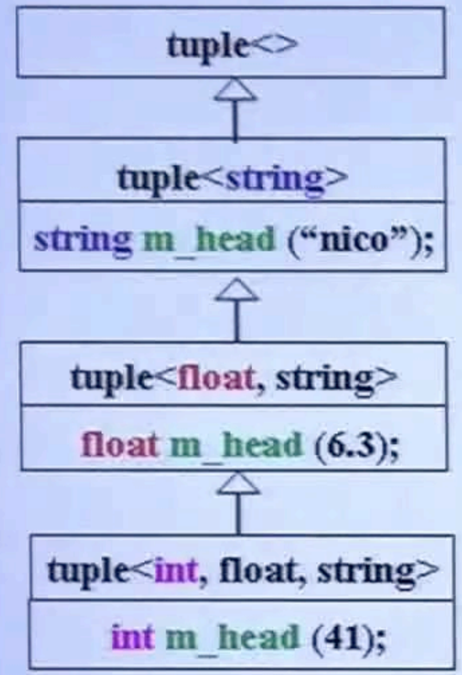
自己写操作符重载

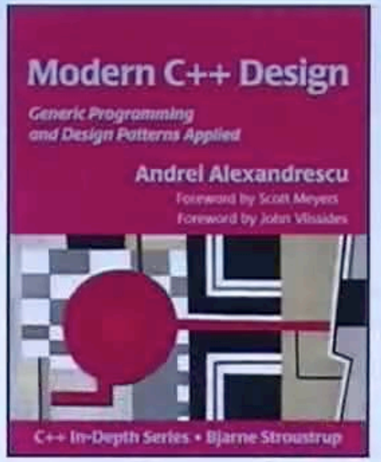


Boolan 博览

話說天寶當年, loki and boost::tuple

std::tuple<int,float,string>





tuple在没有varadic template时，就已经实现过了

分散,撒开

```
typedef GenScatterHiderarchy<
        TYPELIST_4(int, int, string, Widget),
        Holder>  Foo;
    Foo aFoo;
```

會獲得這樣的 object layout：

int
int
string
Widget

```
#define TYPELIST_1(T1) Typelist<T1, NullType>
#define TYPELIST_2(T1, T2) Typelist<T1, TYPELIST_1(T2) >
#define TYPELIST_3(T1, T2, T3) Typelist<T1, TYPELIST_2(T2,T3) >
#define TYPELIST_4(T1, T2, T3, T4) Typelist<T1, TYPELIST_3(T2,T3,T4) >
```

现在2.0把此功能做到语言里了
局限：看写多少#define，就最多放多少类型

...\boost_1_33_0\boost\spirit\phoenix\tuples.hpp

```
template <typename DerivedT>
struct tuple_base {

    typedef nil_t   a_type;
    typedef nil_t   b_type;
    typedef nil_t   c_type;

#if PHOENIX_LIMIT > 3
    typedef nil_t   d_type;
    typedef nil_t   e_type;
    typedef nil_t   f_type;

#if PHOENIX_LIMIT > 6
    typedef nil_t   g_type;
    typedef nil_t   h_type;
    typedef nil_t   i_type;

#if PHOENIX_LIMIT > 9
    typedef nil_t   j_type;
    typedef nil_t   k_type;
    typedef nil_t   l_type;

#if PHOENIX_LIMIT > 12
    typedef nil_t   m_type;
    typedef nil_t   n_type;
    typedef nil_t   o_type;

#endif
#endif
#endif
#endif
...
operator[](tuple_index<N>) const
operator[](tup
};
```

```
struct nil_t {};
```

Boolan 博览

boost::tuple 更早时，boost就有tuple

...\boost_1_33_0\boost\spirit\phoenix\tuples.hpp

struct **tuple**;

tuple<A,.....>
a : A

tuple<A,B,.....>
a : A; b : B;

tuple<A,B,C,.....>
a : A; b : B; c : C;

...

```
// tuple <1 member> class
////////////////////////////////////////////
template <typename A>
struct tuple<A, nil_t, nil_t,
#if PHOENIX_LIMIT > 3
   nil_t, nil_t, nil_t,
#if PHOENIX_LIMIT > 6
   nil_t, nil_t, nil_t,
#if PHOENIX_LIMIT > 9
   nil_t, nil_t, nil_t,
#if PHOENIX_LIMIT > 12
   nil_t, nil_t, nil_t,
#endif
#endif
#endif
#endif
   nil_t  // Unused
> : public tuple_base<tuple<A> > {
...
typedef A a_type;

   A a;
};
```

```
// tuple <2 member> class
////////////////////////////////////////////////////
template <typename A, typename B>
struct tuple<A, B, nil_t,
#if PHOENIX_LIMIT > 3
   nil_t, nil_t, nil_t,
#if PHOENIX_LIMIT > 6
   nil_t, nil_t, nil_t,
#if PHOENIX_LIMIT > 9
   nil_t, nil_t, nil_t,
#if PHOENIX_LIMIT > 12
   nil_t, nil_t, nil_t,
#endif
#endif
#endif
#endif
   nil_t  // Unused
> : public tuple_base<tuple<A, B> > {
...
typedef A a_type; typedef B b_type;

   A a; B b;
};
```

```
// tuple <15 member> class
////////////////////////////////////////////////////////////////////////////////
template <
   typename A, typename B, typename C, typename D, typename E,
   typename F, typename G, typename H, typename I, typename J,
   typename K, typename L, typename M, typename N, typename O>
struct tuple<A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, nil_t>
:  public tuple_base<
      tuple<A, B, C, D, E, F, G, H, I, J, K, L, M, N, O> > {
...
   typedef A a_type; typedef B b_type;
   typedef C c_type; typedef D d_type;
   typedef E e_type; typedef F f_type;
   typedef G g_type; typedef H h_type;
   typedef I i_type; typedef J j_type;
   typedef K k_type; typedef L l_type;
   typedef M m_type; typedef N n_type;
   typedef O o_type;

   A a; B b; C c; D d; E e;
   F f; G g; H h; I i; J j;
   K k; L l; M m; N n; O o;
};
```

替換掉 tuple_base 中
對這些 type 的定義
(當時都定義為 nil_t)

Boolan 博览