附录 A

C++11 对其他标准的不兼容项目

在附录部分，我们会详细描述C++11的不兼容性（incompatibility）、废弃的特性（deprecated feature），以及编译器支持状况（compiler support status）。虽然这些内容不及第2 ~ 8章的“核心”内容重要，不过却常常具有很高的实用性。我们建议读者可以粗略地阅读一遍相关内容。这样在遇到一些实际编程问题的时候，读者就可能理解问题的来由。这几个附录的内容上可能存在着一些重复，不过，我们还是保持了这样的重复，以保证从每个视角出发的描述的完整性。

那么本附录要讲解的是C++11的一些不兼容性。虽然C++11作为C/C++的“嫡传后裔”，对C/C++98/03做到了最大的兼容，不过一些显著的不兼容性还是存在的，我们可以分别通过比较C++11与C++03、C++与ISO C，以及比较C++11与C11来进行了解。

A.1 C++11和C++03的不兼容项目

条目1 在C++11中R、u8、u8R、u、uR、U、UR和LR是新的字符串修饰符，当用它们来修饰字符串时，即使它们是宏名，也将作为修饰符来解释。比如：

```
#define u8 "AAAAA"
const char * s = u8"u-eight-string";
```

在C++03中s是字符串“AAAAAu-eight-string”；在C++11中s是一个UTF-8的字符串，其内容是“u-eight-string”。

条目2 C++11支持用户自定义的字面常量，这会引起一些和C++03不一致的行为，比如：

```
#define _x " world"
"hello"_x
```

在C++03中"hello"_x会拼接成hello world；而在C++11中"hello"_x会作为一个用户自定义字面常量来使用，例如：

```
std::string operator ""_x(const char* s) {
   return std::string(s);
}
```

那么"hello"_x将会作为函数调用返回一个类型为std::string的变量，这个返回变量的内

容是 hello。

条目 3 C++11 引入了一些新的关键字，如果 C++03 代码用到这些标识符会被 C++11 视为非法的代码。这些关键字包括 alignas、alignof、char16_t、char32_t、constexpr、decltype、noexcept、nullptr、static_assert 和 thread_local。

条目 4 C++11 引入了 C99 的新类型 long long。类似 C99，对于长于 long 类型的整型常量将会被转换成 signed long long 类型。而在 C++03 中，长于 long 类型的整型常量将会被转换成无符号整数，如 unsigned long。例如 214748364700 在 C++11 中将会被识别为 long long 类型数据。

条目 5 C++11 和 C99 一样，对整数向“0”取商（/）或取余（%）；而 C++03 允许向负无穷取商或取余。

条目 6 关键字 auto 不再被用来作为存储类型的修饰符，而其表示修饰的类型是由初始化表达式推导而来。

条目 7 C++11 要求数组初始化时，不能将数据的类型收窄。下面的代码在 C++03 中合法，而在 C++11 中非法。

```
int arr[] = {1.0};
```

这里 1.0 是一个 double 类型，使用它初始化 int 类型数组会导致数据收窄。因此在 C++11 中无法通过编译。

条目 8 可能导致问题的隐式函数在 C++11 被定义为 deleted。这些隐式函数不能被使用。而 C++03 中可以使用。比如说下面这段代码：

```
struct A{ const int a; };
```

由于常量（const）a 总是应该被静态初始化的，因此程序员应该为 struct A 提供一个构造函数来完成这样的初始化。在 C++11 中，遇到这种可能导致问题（would be ill-formed）的情况下，缺省构造函数将被删除，以提示用户可能存在问题。

条目 9 C++11 中去除了无用的关键字：export。

条目 10 C++11 中，模板嵌套时可以直接使用双右尖括号，C++03 则需要空白字符填充尖括号。例如：

```
template <typename T> struct X { };
template <int N> struct Y { };
X< Y< 1 >> 2 > > x;
```

C++03 中会解释为 X< Y< (1 >> 2) > > x; ==> X< Y< 0 > > x。

C++11 中这是非法的声明，因为“X< Y< 1 >>”被视为有效的模板使用。

条目 11 C++11 引入了一些新的标准头文件：<array>、<atomic>、<chrono>、<codecvt>、<condition_variable>、<forward_list>、<future>、<initializer_list>、<mutex>、<random>、<ratio>、<regex>、<scoped_allocator>、<system_error>、<thread>、<tuple>、<typeindex>、<type_traits>、<unordered_map>、<unordered_set>。

还有一些新加入的和 C 兼容的头文件：<ccomplex>、<cfenv>、<cinttypes>、<cstdalign>、<cstdbool>、<cstding>、<ctgmath>、<cuchar>。

条目 12 C++11 中，swap 方法从 <algorithm> 移到了 <utility> 中。

条目 13 C++11 加入了一个新的顶级 namespace：posix。

条目 14 通用属性中的标记符如 carries_dependency、noreturn 不能作为宏名。

条目 15 C++03 假设全局的 new 操作符只会抛出类型为 std::bad_alloc 的异常；而 C++11 允许全局的 new 操作符抛出其他类型的异常。

条目 16 C++11 要求 errno 变量是线程局部的，而不是全局的。

条目 17 C++11 支持轻量级的垃圾回收机制。

条目 18 标准库中的仿函数（函数对象）不再继承自 std::unary_function 和 std::binary_function。

条目 19 标准容器要提供的 size() 成员函数要求是 O(0) 复杂度；C++03 中 std::list 的成员 size() 允许线性复杂度。

条目 20 C++11 中改变了一些函数方法的原型，比如 erase 和 insert 的返回值类型 iterator 变成了 const_iterator，resize 函数的参数从传值改为了传引用。

条目 21 C++11 允许一些类和函数方法的实现不同于 C++03，比如：std::remove、std::remove_if、std::complex、std::ios_base::failure。

A.2 C++ 和 ISO C 标准的不兼容项目

条目 1 C++ 中很多关键字是 C 所没有的（不详细列举）。

条目 2 C 中字符常量的类型是 int，C++ 中是 char。如果在 C++ 代码中同时有以下两个版本的 f 函数的定义：

```
void f (char c);
void f(int);
```

那么，函数调用 f('x') 会选择 void f (char c) 版本。

条目 3 C++ 中字符串常量的类型是 const char[]，而 C 中字符串常量的类型是 char[]。

条目 4 C 中允许文件范围中的变量重复定义。如：

```
int var;
int var;
```

在 C 中是允许的；而这在 C++ 中是不允许的。

条目 5 不带 extern 关键字的 const 变量在 C++ 中是 internal linkage（内部链接的，即不可以被其他文件中同名变量引用）；而在 C 中则是 external liankage（外部链接的，即可以被其他文件中的同名变量引用）。

条目 6 C++ 要求从 void* 类型变量到其他类型的转化必须是显式的；而 C 中则不需要显式转换。

条目 7 C++ 中只有非常量非易变对象（non-const ,non-volatile）指针可以转换为 void* 类型。

条目 8 C++ 不接受隐式声明函数。这个特性在 ISO C 中也逐渐被抛弃，比如：

```
int main(){ printf("hello\n");}
```

这里因为 printf 没有定义（没有 #include <stdio.h>），C++ 会编译时报错 printf 未声明；而 C 会把 printf 当作一个 int printf（任意参数）的函数类型。如果运行时动态库中不存在 printf 这个函数的话，则会导致运行时错误。

条目 9 不能在结构体或类型的声明上加 static 关键字。比如：

```
static struct st { int i; };
```

在 C 中 static 关键字将被忽略；C++ 中这则是错误的语法。

条目 10 C++ 中 typedef 的类型别名不能和已有的类型同名。

条目 11 常量（const）对象在 C++ 中必须初始化；在 C 中则没这个限制。

条目 12 隐式 int 类型在 C++ 中被禁止，C 中也逐渐抛弃。比如：func(){} 这样的声明方式在 C 语言中是可以的；而在 C++ 中，因为 func 没有返回值类型则是非法表达式。

条目 13 关键字 auto 在 C++11 中有新的语义：用于类型自动推导；而 C 中 auto 是修饰对象的存储类型的关键字。

条目 14 C++ 中 enum 对象只能用同类型的 enum 赋值；而 C 中可以用任意的整型数对 enum 变量赋值。C++ 中 enum 变量的类型是对应的 enum 类型；而 C 中 enum 对象的类型是 int 整型。

条目 15 函数声明中的空参数在 C++ 中意味着函数没有参数；而在 C 中则意味着该函数的参数个数未知。

条目 16 C++ 不允许类型定义在函数的参数或返回值类型的位置上；而形如：

```
void f(struct S {int I;} s);
```

这样的表达式在 C 中则可以接受。

条目 17 C++ 不接受老的废弃的函数定义格式：参数在 () 之外，比如：

```
void bar() int par1 {}
```

在 C++ 中就是非法的声明。

条目 18 作用域内部的结构体在 C++ 中会覆盖作用域外部的同名变量，比如：

```
char s;
void f(){
    struct s {
        int i;
    }; // struct s 覆盖 char s
}
```

而在 C 中不会。

条目 19 C++ 中，嵌套的结构体仅在其父结构体作用域中可见；而在 C 中，嵌套的结构体则在全局可见，比如：

```
struct Outter {
    struct Inner {
        int I;
    };
};
```

在 C 中使用 Inner 类型可以直接写：struct Inner in，而 C++ 中使用 Inner 类型则必须带上其父结构体 Outter，则只能写：Out::Inner in。

条目 20 C++ 中 typedef 形成的类型别名不能重定义为其他类型或变量。如下面的代码就是这样一种情况：

```
typedef int Int;
struct S{
   int Int; // 在 C 中合法，而在 C++ 中非法
};
```

条目 21 C++ 中 volatile 的对象不能作为隐式构造函数和隐式赋值函数的参数，比如：

```
struct X { int i; };
volatile struct X x1 = {0};
struct X x2(x1); // 在 C++ 中非法
struct X x3;
x3 = x1; // 在 C++ 中同样非法
```

A.3 C++11 与 C11 的区别

虽然 C11 标准开始起草的时间比 C++0x 晚很多，但 C11 的发布却只比 C++11 晚了几个月。这是因为它们的草案中很多都是相互参考的。因此 C++11 与 C11 的不兼容点并不多。

下面简单地列一些 C++11 和 C11 的特别区别点。

条目 1 C++11 中没有 C11 中支持的 _Generic 关键字，因为 C++ 能够很好地支持重载。

条目 2 C++11 中 noreturn 是一个通用属性。相应地要表示函数永不返回的话，在 C11 中可以使用 _Noreturn 关键字。比如：

```
_Noreturn void outfunc() { abort(); }
```

是 C11 中的表示 outfunc 的方法，它等价于在 C++11 中使用通用属性的 outfunc。

```
[[ noreturn ]] void outfunc() { abort(); }
```

条目 3 许多 C11 的新特性在 C++11 中有对应特性。只是关键字上有一些细微的区别。比如 C++11 中的关键字：

```
alignas, alignof, thread_local, static_assert
```

在 C11 中对应的关键字分别是：

```
_Alignas, _Alignof, _Thread_local,  _Static_assert
```

条目 4 C++11 和 C11 都有 atomic 支持。

C11 中用 _Atomic 修饰符来修饰一个原子数据类型。如：_Atomic int i;

C++11 在 std namespace 下定义了 atomic 模板来支持 atomic 类型，比如：

```
template<class T> struct atomic;
template<> struct atomic<integral>;
template<class T> struct atomic<T*>;
```

条目 5 C11 中用 <threads.h> 中定义的一系列用于互斥操作的函数，比如：mtx_destroy, mtx_init, mtx_lock, mtx_trylock, mtx_unlock 等。

C++11 中通过使用 std namespace 下的一些类：mutex、recursive_mutex 等，通过这些类的成员函数 lock、unlock、trylock 支持互斥操作。

条目 6 C11 用 <threads.h> 中定义的一系列函数来支持线程，如：thrd_create，thrd_current，thrd_detach，thrd_equal，thrd_exit，thrd_join，thrd_sleep，thrd_yield。

C++11 有 thread 类型，该类型有 join、detach 等成员函数。

A.4 针对 C++03 的完善

而谈及兼容性的话，除了上面列举的 C++11 与 C++03、ISO C 以及 C11 的区别外，在 C++11 起草的过程中，也包含了一些对以前标准（C++03）的修改和完善。我们把一些针对 C++03 的完善也列举了出来。

条目 1 .* 和 ->* 操作符的第二个参数不再要求是一个完全类（complete class），即包含了全部声明体的类型。

条目 2 内存释放函数不因抛出异常而终止。

条目 3 C++03 要求，当第一个参数是空指针（null pointer）的时候，内存释放函数（如用户自定义的 delete 操作符）相当于无任何作用。现在不再有这样的限制。

条目 4 如果 typeid 操作符的参数是 cv 修饰的，其结果是对应的无 cv 修饰的类型。

条目 5 在常量表示式中可以使用 throw，如：const char * s = (n == m) ? throw “bad” : “ok”；在 C++11 中是合法的表达式。

在 C++11 中，这样的改进还有很多。更多的信息读者可以参考以下链接：http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_defects.html。

附录 B

弃用的特性

随着 C++11 的发布和新特性的出现，一些 C++98 与 C++03 中的特性也即将被淘汰。其中一些被更强大的新特性所取代，如 auto_ptr 等，也有因为各种缺陷而在实际编程中很少被使用的，如 export、register 等。相比于不兼容性，了解为什么弃用往往更能了解语言在如何发展。本章将详细描述并总结被 C++11 所弃用的各种特性。

条目 1　auto 关键字

旧特性：auto 用来标识具有自动存储期的局部变量。

改动：auto 关键字可以用来从变量的初始值中推导出变量的类型。

在旧标准中，auto 用来声明具有自动存储期的局部变量，这样变量就是自动存储类别，属于动态存储方式，当变量离开作用域后存储空间会被自动释放。实际上，所有非静态的局部变量默认都具有自动的存储期，因此 auto 关键字很少被用到。

在 C++11 新标准中，auto 被作为一个新的类型修饰符，声明变量时不用指定变量类型，而是根据该变量的初始化表达式或者一个具有追踪返回类型的函数定义推导得来。下面举一个例子，见以下代码：

```
for (vector<int>::iterator i = vec.begin(); i < vec.end(); i++)  {
   vector<int>::iterator j = i;
}
```

在 C++11 中，我们可以使用 auto 关键字来提高可读性。

```
for (auto i = vec.begin(); i < vec.end(); i++) {
   auto j = i;
}
```

此外，auto 类型推导使用非常灵活，它几乎可以用在任何需要声明变量类型的上下文中，比如命名空间、for 循环的循环变量初始化及 for 循环体，以及判断语句中，甚至能被使用在模板中。但是，auto 不可以声明函数参数，也不能推导数组类型。

由于 auto 在 C++11 中被赋予了新的语义，为了避免混淆，C++11 中 auto 不再作为存储的变量声明，而只作为类型修饰符。

条目 2　语言特性 export

旧特性：用来定义非内联的模板对象和模板函数。

改动：export 特性被移除。export 关键字被保留，但是不包含任何语义。

我们可以使用关键字 extern 来访问其他编译单元中普通类型的变量或对象，而对于模板来说，则需要使用 export 关键字。export 的设计初衷是想要创建一个折中的设计方法，来同时支持模板实例化的包含模型（inclusion model）和独立编译模型（separate compilation model），然而，没有一种增强机制来确保每一种模型的实现都可以很简单。因此，由于实现的难度，很多编译器都没有实现。此外，export 关键字在实际编程过程中也很少被用到。

所以，在 C++11 中，export 关键字的语义被移除。但是 export 仍作为一个无语义的关键字被保留下来。

条目 3 register 关键字（作为存储类）

旧特性：声明将变量存放在寄存器中。

改动：改变为声明存储类的关键字。

在旧标准中，变量用 register 来声明时，表示此变量会被大量用到，因此建议将此变量存放在寄存器中，这样可以提高读取的速度。但是，寄存器的数量是有限的，如果寄存器已满，变量依旧会被存放在存储器中。另一方面，它对于编译器来说只是一种建议，而编译器不一定会执行，实际上，大部分编译器都选择忽略它。因此，register 关键字其实很少被用到，而且，大多数情况下是没有意义的。

在 C++11 新标准中，register 关键字的作用有所改变。用 register 关键字仅能用于一个区块内的变量声明或作为函数参数的声明。它仅仅表示变量拥有自动存储的生命期（像 C++03 中的 auto 一样）。

条目 4 隐式拷贝函数

旧特性：如果类中已经声明了其他拷贝函数或者析构函数，编译器依旧会自动生成一个隐式拷贝函数。

改动：隐式拷贝函数不会自动生成。

在 C++11 中，如果用户已经声明了一个拷贝复制操作符或者一个析构函数，那么编译器不会隐式声明一个拷贝构造函数。同样，如果用户已经声明了一个拷贝构造函数或者析构函数，编译器则不会隐式地声明一个拷贝复制操作符。

条目 5 auto_ptr

旧特性：智能指针，当系统因异常退出时避免资源泄漏。

改动：auto_ptr 被 unique_ptr 所取代。

auto_ptr 类模板中存放了一个指针，它指向一个可以通过 new 得到的对象，并且在此智能指针被析构时向堆归还该对象。这里需要注意的是 auto_ptr 拥有一个严格的所有权机制。auto_ptr 拥有其指针指向的对象的所有权，而复制 auto_ptr 的操作会复制该指针，并将对象的所有权交给目标类。这是为了避免两个 auto_ptr 同时拥有同一个对象，否则程序的行为将是不确定的。

在 C++11 中，unique_ptr 提供了一种比 auto_ptr 更好的解决方案，并取代了 auto_ptr。unique_ptr 是一个对象，它拥有另一个对象，并且能够通过指针来管理它。更准确地说，unique_ptr 对象中有一个指向另一个对象的指针，并且在它自身析构时析构该对象。这些特性都与 auto_ptr 相同。

此外，unique_ptr 也具备了 auto_ptr 的绝大部分特性，除了 auto_ptr 的不安全隐性的左值转移（move）。对于 auto_ptr 来说，拷贝 auto_ptr ，会导致所有权转移，如以下语句：

```
std::auto_ptr<int> a(new int);
std::auto_ptr<int> b = a;
```

同样，拷贝构造函数也会进行所有权的转移，如以下语句：

```
std::auto_ptr<int> c(a);
```

由此可见，auto_ptr 的转移是隐性的，因此程序员可能会在不经意的情况下就把对象转移了，因此是不安全性。

在 unique_ptr 中，要进行对象的转移，需要使用 std::move 函数将对象转换为右值，例如以下语句：

```
std::unique_ptr<int> a(new int);
std::unique_ptr<int> b = std::move(a);
```

这是因为 unique_ptr 对于拷贝行为作了限制。而对于拷贝构造函数来说，unique_ptr 并没有类似以下的构造函数：

```
std::unique_ptr<T>::unique_ptr(std::unique_ptr<T> const&) // 默认 deleted
```

如果在构造时想要复制整数的值，可以用以下语句：

```
std::unique_ptr<int> c(new int(*a));
```

而如果确实想要将 a 中的指针进行转移，则需要调用 std::move：

```
std::unique_ptr<int> d(std::move(a));
```

另外，值得一提的是，unique_ptr 可以存放在标准容器之中。

```
vector<unique_ptr<int>> v;
v.push_back(unique_ptr<int>(new int(0)));
unique_ptr<int> a(new int(0));
v.push_back(move(a));
```

但是，由于 unique_ptr 本身不支持拷贝构造，因此元素类型为 unique_ptr 的容器同样也不支持拷贝构造，这时也需要用到转移构造。

条目 6　bind1st/bind2nd

旧特性：将二元函数对象绑定成一元仿函数（函数对象）。

改动：被 bind 模板所取代。

bind1st 和 bind2nd 函数可以将一个二元函数绑定成一元函数，也就是将二元函数所接受的两个参数之一绑定下来，以此来使函数变成一元的。bind1st 绑定第一个参数，bind2nd 绑定第二个参数。例如：

```
find_if(v.begin(), v.end(), bind2nd(greater<int>(), 5));
```

绑定 greater<int> 的第二个参数为 5，亦即找到向量中第一个大于 5 的整数。

```
find_if(v.begin(), v.end(), bind1st(greater<int>(), 5));
```

绑定 greater<int> 的第一个参数为 5，亦即找到向量中第一个小于 5 的整数。

在 C++11 中，新的 bind 函数模板提供了一种更好的可调用类的参数绑定机制。

```
namespace std {
    template<class T> struct is_bind_expression
    : integral_constant<bool, see below> { };
}
```

接下来我们来看 bind 模板函数，它有如下形式：

```
template<class F, class... BoundArgs>
    unspecified bind(F&& f, BoundArgs&&... bound_args);
```

其中 f 是函数的右值引用，表示要进行绑定的函数对象，BoundArgs 是函数对象的参数类型列表，而 bound_args 是需要绑定的值。如果一个参数需要绑定，那么在调用 bind 函数时传具体参数进去即可，而如果不需要绑定，那么就需要使用占位符，std::placeholders::_J，J 为从 1 开始的正整数。bind 的返回类型为可调用实体，可以直接赋值给 std::function。

如下这个例子：

```
int Func(int x, int y);
function< int(int)> f = bind(Func, 1, placeholders::_1);
f(2); // the same as Func(1, 2);
```

我们可以用 is_bind_expression 来检查由 bind 生成的函数对象，而 bind 也凭借 is_bind_expression 来检查子表达式。对于用户来说，可以借由它来表示在 bind 调用中某个类型应该被当做子表达式来对待。如果 T 是 bind 的返回类型，那么 is_bind_expression 由 integral_

constant<bool, true> 得到，否则由 integral_constant<bool, false> 得到。

另一个值得一提的函数是 is_placeholder，它可以检查标准占位符 _1、_2 等。bind 凭借 is_placeholder 来检查占位符，用户也可以借由此模板来表示占位符类型。如果 T 的类型是 std::placeholders::_J，则 is_placeholder<T> 由 integral_constant<int, J> 得到，否则就由 integral_constant<int, 0> 得到。

由上可以看出 bind 相对于 bind1st 和 bind2nd 来说要灵活得多，它不像 bind1st 和 bind2nd 那样限制原函数对象的参数个数为两个，bind 所接受的函数对象的参数数量没有限制，而且用户可以随意绑定任意个数的参数而不受限制，因此，有了 bind，bind1st 和 bind2nd 明显没有了用武之地而被弃用（deprecated）。

条目 7 函数适配器（adaptor）

旧特性：ptr_fun, mem_fun, mem_fun_ref, unary_function, binary_function

新特性：弃用。

在旧特性中，提供了多个函数适配器。

```
template <class Arg1, class Arg2, class Result>
    pointer_to_binary_function<Arg1,Arg2,Result>
     ptr_fun(Result (*f)(Arg1, Arg2));
```

ptr_fun 的返回值是 pointer_to_binary_function<Arg1,Arg2,Result>(f)。简单来说，它可以将一个函数转化为一个函数对象。以下是一个例子：

```
int compare(const char*, const char*);
    replace_if(v.begin(), v.end(),
    not1(bind2nd(ptr_fun(compare), "abc")), "def");
```

上例将所有 v 序列中的 abc 替换为 def。

另外，ptr_fun 除了可以转化二元函数以外，也可以转化一元函数，此时返回值是 pointer_to_binary_function<Arg,Result>(f)。

```
template <class Arg, class Result>
    pointer_to_unary_function<Arg,Result>
    ptr_fun(Result (*f)(Arg));
```

除了常规函数适配器 ptr_fun 外，还有成员函数适配器 mem_fun 和 mem_fun_ref。

```
template <class S, class T> class mem_fun_t
     : public unary_function<T*, S>
template<class S, class T> mem_fun_t<S,T> mem_fun(S (T::*f)());

template <class S, class T> class mem_fun_ref_t
     : public unary_function<T, S>
```

```
template<class S, class T> mem_fun_ref_t<S,T> mem_fun_ref(S (T::*f)());
```

为了说明 mem_fun 和 mem_fun_ref，看一下以下的例子：

```
void f(C& c);
vector<C> vc;
for_each(vc.begin(), vc.end(), f);
```

就上例来说代码是可以编译通过的，但是，当 f 是类 C 的成员函数时呢?

```
class C {
    public:
        void f();
};
```

此时我们就需要使用到类成员函数适配器了，在这时，mem_fun、mem_fun_ref 的区别在于 mem_fun 需要指针，而 mem_fun_ref 需要对象的引用。如上述例子中，应该使用 mem_fun_ref。

```
for_each(vc.begin(), vc.end(), mem_fun_ref(&C::f));
```

mem_fun 的用法类似，在此不赘述。由此可见，类成员函数适配器可以将一个不含参数的成员函数转换为一个一元函数，其中参数类型为类本身。此外，它也可以将一个一元成员函数转换为一个二元函数。

mem_fun 和 mem_fun_ref 的返回类型不仅仅只有 mem_fun_t 和 mem_fun_ref_t，而是根据转换后的函数类型不同而有所不同，所有的返回类型如表 B-1 所示。

表 B-1　类成员函数适配器的返回类型

	mem_fun	mem_fun_ref
一元函数	mem_fun_t	mem_fun_ref_t
二元函数	mem_fun1_t	mem_fun1_ref_t
const 修饰的一元函数	const_mem_fun_t	const_mem_fun_ref_t
const 修饰的二元函数	const_mem_fun1_t	const_mem_fun1_ref_t

现在我们来看 C++11 的新特性。我们前面已经介绍过了新的 bind 函数模板，它取代了原有的 bind1st 和 bind2nd。bind 不再需要 ptr_fun，因此 ptr_fun 被弃用。而对于 mem_fun 和 mem_fun_ref，C++11 提出了一个新的函数模板 mem_fn，它实现了 mem_fun 和 mem_fun_ref 的所有功能，而且更为强大。它不像 mem_fun 和 mem_fun_ref 那样只能处理一元函数或二元函数，它能够针对任意多个参数的函数进行转换；使用时，也不用再区分是指针还是一般对象。因此 mem_fun 和 mem_fun_ref 也被弃用，不仅如此，它们所有的返回类型也被弃用。另外，被弃用的还包括 unary_function 和 binary_function。

条目 8　动态异常声明（exception specification）

旧特性：异常声明 throw()

改动：有参数的异常声明被弃用，空异常声明 throw() 被 noexcept 取代。

函数可以通过异常声明来列出它直接或者间接可能抛出的异常。

形如 throw(T) 的异常声明成为动态异常声明。当一个函数抛出 E 类型的异常时，如果它的动态异常声明包含一个类型 T，且它的处理函数（handler）和类型 E 是匹配的，那么这个函数就允许 E 类型的异常。当抛出一个异常时，编译器会搜索处理函数，如果直到最外层的带有异常声明的代码模块都不允许此异常的话，那么如果是动态异常声明，就会调用 std::unexpected()。

实践证明，动态异常声明是没有价值的，只能给程序带来更多的开销。它主要的问题如下：

- C++ 的异常声明是一种运行时检查，而不是编译时，也就是说，在编译时不能确保所有的异常都能被处理，而运行时的失败模式（failure mode），也就是调用 std::unexpected() 并不能自身恢复。
- 运行时的检查会要求编译器生成更多代码，而这些代码会阻碍优化，增大运行时开销。
- 在泛型的代码中，很难预知在模板参数的操作过程中会抛出什么类型的异常，所以不太可能写出准确的异常声明。

因此，在 C++11 中，动态异常声明被弃用。

在动态异常声明中，作为唯一的例外而被认为有价值的是空异常声明，也就是 throw()。在实践中，只有两种异常的抛出确实是有用的：程序会抛出异常或者程序不会抛出异常。前者可以由完全省略异常声明来表示；后者则可以由 throw() 来表示。但是由于性能方面的考虑，还是很少被用到。

在 C++11 中，提出了一种新的异常声明 noexcept，关键字 noexcept 表示函数不会抛出异常，或者说异常不会被接获并处理。noexcept 异常声明除了有 noexcept 关键字的形式，还可以是 noexcept (constant-expression) 的形式，这里 constant-expression 要求可以被转换为 bool 类型。这样，noexcept 可以通过条件判断来决定函数是不是能够抛出异常。另外，noexcept 关键字的意义其实就等于 noexcept(true)。当用 noexcept 修饰的函数，也就是不允许抛出异常的函数中抛出异常时，编译器会调用 std::terminate()。

与 throw() 不同，noexcept 不需要编译器生成额外的代码来进行运行时检查，而且使用上更为灵活，因此完全可以取代 throw()。

综上所述，由于含有参数的动态异常声明在实际使用中没有价值，而空动态异常声明 throw() 已被 noexcept 取代，所以动态异常声明，也就是 throw (type-id-listopt) 被弃用。

附录 C

编译器支持

C++11 是否能够在 20 世纪的第二个 10 年光芒依旧，必不可少地需要整个行业的生态环境的支持。这意味着一方面是 C++11 在学习使用上的闪光点深入人心，而另外一方面，则是有广泛的编译器支持。

如同我们在第一章中提到的，事实上，大多数编译器组织或厂商都在着手支持 C++11。但 C++11 的特性非常多，以至于编译器组织或厂商通常需要若干个版本才能完全支持。在本书编写时，地球上的所有编译器都还未能完全地支持所有的 C++11 特性。不过这样的状况很快就会得到改变。一些开源的编译器项目，比如 GCC 以及 Clang，应该在不久的时间内即将成为第一个完全支持 C++11 的编译器（现在从我们得知的情况看来，GCC 可能会最早完成）。而商业编译器则相对会慢一些，而且是否完整支持 C++11 有时候也会依据客户需要而定。

在 IBM Power 平台上，最为常用的编译器是 IBM 的 XL C/C++ 及 GCC。截止本书完成，IBM 的 XL C/C++ 编译器的最新版本是 XL C/C++ V12.1，可以用于 AIX 以及 Linux 平台。XL C/C++ V12.1 支持了最为核心的特性，包括了：auto、c99、常量表达式 constexpr、decltype、委托构造函数、显式类型转换 operator、扩展的 friend 声明、外部模板、内联名字空间、long long、追踪返回类型的函数声明、右值引用以及移动语义、右尖括号、静态断言、强类型枚举、变长模板等。

在 XL C++12.1 中（请参见 http://www-01.ibm.com/software/awdtools/xlcpp/），程序员可以通过选项 -qlanglvl=extended0x 来开启对 C++11 大部分特性的支持，-qwarn0x 选项用来诊断 C++11 与 C++98 有区别的代码。此外，程序员还可以在 XLC++ 中仅仅通过一些子选项开启某一个 C++11 的功能，详细如表 C-1 所示。

表 C-1　IBM XL C/C++ 中有关于 C++11 的选项

IBM XL C/C++ 编译器选项	说　明
-qlanglvl=[no]autotypededuction	支持 C++11 中 auto 特性
-qlanglvl=[no]constexpr	支持 C++11 的常量表达式类型
-qlanglvl=[no]decltype	支持 decltype
-qlanglvl=[no]delegatingctors	支持委托构造函数
-qlanglvl=[no]c99longlong	支持 long long 数据类型
-qlanglvl=[no]inlinenamespace	支持内联名字空间

（续）

IBM XL C/C++ 编译器选项	说　明
-qlanglvl=[no]rvaluereferences	支持右值引用
-qlanglvl=[no]static_assert	支持 static assert
-qlanglvl=[no]variadic[templates]	支持变参模板

此外，一如既往，IBM 为所发布的特性都提供了良好的文档支持（请参见 http://pic.dhe.ibm.com/infocenter/lnxpcomp/v121v141/index.jsp）。

而在 x86 及 x86_64 平台上，编译器在 C++11 的支持上则呈现了百花齐放的状态。从商业编译器上讲，主要是 Intel 的 Intel C/C++ 编译器及微软的 MSVC 对 C++11 做了大量的支持。两者最新版本分别为 Intel C/C++ V13 以及 MSVC 2012（本书截稿时还没有正式发布）。

而在开源编译器上，GCC 及基于 llvm 的 clang 则同样站在 C++11 支持的最前列。GCC 对 C++11 的支持在 http://gcc.gnu.org/projects/cxx0x.html 可以找到，同样，clang 对 C++11 的支持可以从 http://clang.llvm.org/cxx_status.html 上找到。可以看见，两款编译器除了依赖于底层的并行特性和少量未完成的特性外，大部分 C++11 的特性都已经得到了支持。

虽然两款编译器都实现了极高的 C++11 支持度，不过两者现在也并未默认开启 C++11 编译支持。程序员可以使用 -std=c++11 来打开 C++11 模式，而选项 -std=gnu++11 可以同时支持 C++11 和 GNU 的扩展功能。

注意　可能读者对 clang 编译器还不是非常了解。不过在我们的使用中，clang++ 编译器则表现了很好的实用性，clang++ 基本上兼容了所有的 g++ 的编译选项，其错误输出在 shell 的支持下能够显示颜色，所以显得更加友好。有一些 Linux 的发布版中，我们已经看到使用 clang 代替 gcc 作为默认编译器的状况。

一些 GCC 中其他 C++11 相关选项则如表 C-2 所示。

表 C-2　GCC 一些与 C++11 有关的编译选项

GCC 编译器选项	说　明
-fabi-version=6	增强 abi 对 C++11 中限定范围的 enum 类型提升的支持
-fconstexpr-depth=n	设置 C++11 常量表达式的计算层数
-Wnarrowing	按 C++11 要求，对数据截断提供诊断信息
-Wc++11-compat	对 C++11 与 C++98 有区别的地方报错
-Wzero-as-null-pointer-constant	当数字 0 作为空指针使用时报错，C++11 中空指针是 nullptr

如第一章所描述的，在本书编写时，我们主要使用了 xl c/c++、gcc 和 clang 三种编译器。因此对其状态也较为熟悉。其他的，比如跟 Intel 编译器同样使用 EDG（Edison Design Group，一个专业的编译器前端厂商）前端的 HP C/aC++ 编译器、Comeau 编译器，以及

Borland/CodeGear 的 C++Builder 也都或多或少地加入了部分 C++11 的支持。

事实上，读者可以通过网页 http://wiki.apache.org/stdcxx/C++0xCompilerSupport 来获知主流编译器组织或厂商对 C++11 编译器的支持情况。这是一张横向比较的表格，如果读者想使用的 C++11 特性不在你的编译器包含之中的话，那么你应该写信催促一下开发者了。

附录 D

相关资源

在本书的编写过程中，作者参考了大量的资料。这些资料主要是一些特性的草案，以及一些源自网络的资源。前者往往通过草案的提出、讨论、修改、决议等各方面揭示了 C++ 特性发展演化的过程的所有情况，而后者则在对标准的阐释、辩析、理解上对本书的编写起了很大的帮助。同样，我们将一些资源罗列出来，以供试图了解 C++ 发展或者仅仅是由于本书未能解除心中疑惑的读者使用。

D.1 C++11 特性建议稿

所有关于 C++ 特性的建议案都在 WG21 的文档库中管理，其链接为：http://www.open-std.org/jtc1/sc22/wg21/docs/papers/，在 WG21 的文档库中，所有的文档都是按时间排列的。这些文档最终演化为 C++ 标准的一部分。表 D-1 则按照主题的方式将这些文档串联起来。由于文档数量较大，而且 WG21 也不时会改变文档的存储路径，所以我们建议读者通过搜索的方式来寻找需要的文档，即在 Google 中输入关键字“WG21”以及文档编号，如“N1377”。一般我们就可以获得该文档的有效 URL 路径。

表 D-1 按主题排列的 C++11 特性建议稿

主　题	英文主题（topic）	文档编号
右值引用	A Proposal to Add an Rvalue Reference to the C++ Language	N1377，N1385，N1690，N1610，N1770，N1855，N1952，N2118
静态断言	static_assert	N1381，N1604，N1617，N1720
模板别名	Template aliases for C++	N1406，N1449，N1451，N1489，N2112，N2258
外部模板	Extern template	N1448，N1960，N1987
*this 的移动语义	Extending Move Semantics To *this (Revision 2)	N1784，N1821，N2377，N2439
通过右值引用初始化类对象	Clarification of Initialization of Class Objects by rvalues	N1610
变长模板	Variadic Templates	N1483，N1603，N1704，N2080，N2152，N2191，N2242
扩展变长模板的参数	Extending Variadic Template Template Parameters	N2488，N2555

（续）

主　题	英文主题（topic）	文档编号
强类型枚举	Strongly Typed Enums	N1513，N1579，N1719，N2213，N2347
枚举类的前置声明	Forward Declaration of Enums	N2499，N2568，N2678，N2764
扩展的 friend 声明	Extended friend Declarations	N1520，N1616，N1722，N1791
泛化的常量表达式	Generalized Constant Expressions	N1521，N1972，N1980，N2116，N2235
一些关于 C99 宏定义	Synchronizing the C++ preprocessor with C99 : variadic macros, empty macro argument, concatenation of mixed char and wchar literals	N1545，N1566，N1653
对齐支持	Adding Alignment Support to the C++ Programming Language	N1546，N1877，N1971，N2140，N2165，N2252，N2301，N2341
条件支持的行为	"Conditionally-Supported Behavior"	N1564，N1627
将未定义行为变为可诊断的错误	Changing Undefined Behavior into Diagnosable Errors	N1727
增加 long long 类型	Adding the long long type to C++	N1565，N1693，N1735，N1811
扩展的整型	Adding extended integer types to C++	N1746，N1988
委托构造函数	Delegating Constructors	N1581，N1618，N1895，N1986
新字符类型 char16_t 和 char32_t	New Character Types in C++ char16_t,char32_t	N1628，N1823，N1955，N2018，N2149，N2249
右尖括号	Right Angle Brackets	N1649，N1699，N1757
由初始化表达式进行类型推导	Deducing the type of variable from its initializer expression	N1721，N1794，N1894，N1984
	A Proposal to Restore Multi-declarator auto Declarations	N1737
auto 的语法	The Syntax of auto Declarations	N2337，N2546
追踪返回类型	New function declaration syntax for deduced return types	N2445，N2541
	A finer-grained alternative to sequence points	N1944，N2052，N2171，N2239
__func__ 预定义标识符	Proposed addition of __func__ predefined identifier from C99	N1970，N2202，N2251，N2340
POD	PODs unstrung	N2062，N2102，N2172，N2230，N2294，N2342
在 thread join 的时候复制异常	Propagating exceptions when joining threads	N2096，N2179
decltype	Decltype	N2115，N2343
Decltype 及调用表达式	Decltype and Call Expressions	N3276
扩展的 sizeof	Extending sizeof	N2150，N2253
显示缺省和删除的函数	Defaulted and Deleted Functions	N2210，N2326，N2346
	Not so trivial issues with trivial	N2762

（续）

主　题	英文主题（topic）	文档编号
lambda 函数	(monomorphic) lambda expressions and closures for C++	N1968，N2329，N2413，N2487，N2529，N2550
lambda 函数的正确性	Constness of lambda functions	N2561，N2658
指针空值 nullptr	A name for the null pointer: nullptr	N1488，N1601，N2214，N2431
	Core issue 654:convertibility of 0-literal	N2656
继承构造函数	Inheriting Constructors	N1898，N2119，N2203，N2254，N2376，N2438，N2512，N2540
显式转换操作符	Explicit Conversion Operators	N1592，N2223，N2333，N2380，N2437
原生 Unicode 字符串字面量	Raw.unicode String Literals	N2053，N2146，N2295，N2384，N2442
字符串中的 unicode 字符	Universal Character Names in Literals	N2170
名字空间的联合	Namespace Association ("Strong Using")	N1526，N2013，N2331，N2535
非受限联合体	Unrestricted Unions	N2248，N2430，N2544
原子操作	Atomic operations with multi-threaded environments	N2047，N2145，N2324，N2381，N2393，N2427
顺序及内存模型	Sequencing and the concurrency memory model	N2052，N2171，N2300，N2334，N2429
快速退出程序	Abandoning a Process(at_quick_exit)	N2383，N2440
允许信号捕捉函数使用 atomic	Allow atomics use in signal handlers	N2459，N2547
多线程库	A Multi-threading Library for Standard C++	N2320，N2447
UTF8 字面量	Unicode Strings UTF8 Literals	N2159，N2209，N2295，N2384，N2442
type_trait 及局部类	type_trait names Making Local Classes more Useful	N1427，N1945，N2187，N2402，N2635，N2657
初始化列表	Initializer lists	N1509，N1890，N1919，N2100，N2215，N2385，N2531，N2672
线程局部存储	Thread-Local Storage	N1874，N1966，N2147，N2280，N2545，N2659
数据相关的排序：原子操作与内存模型	C++ Data-Dependency Ordering: Atomics and Memory Model	N2492，N2556，N2664
数据相关的排序：函数	C++ data-dependency ordering: function annotation	N2361，N2493
动态初始化及并行	Dynamic initialization and concurrency	N2148，N2325，N2382，N2444，N2513，N2660
最小垃圾回收支持	Minimal Support for Garbage Collection and Reachability-Based Leak Detection	N2481，N2527，N2585，N2586，N2670
SFINAE	Solving the SFINAE problem for expressions	N2634
双向栅栏	Proposed text for bidirectional fences	N2633，N2731，N2752
成员快速初始化	Member Initializers	N1959，N2354，N2426，N2756

（续）

主　题	英文主题（topic）	文档编号
一些概念	Concepts (unified proposal)	N2042，N2081，N2193，N2307，N2398，N2421，N2501，N2617，N2676，N2710，N2741
一些概念	Named requirements for C++0x concepts	N2581，N2780
基于范围的 for	Wording for range-based for-loop (revision 3)	N1868，N1961，N2049，N2196，N2243，N2394
通用属性	General Attributes for C++	N2224，N2236，N2379，N2418，N2466，N2553，N2751，N2761
用户自定义字面量	Extensible Literals	N1511，N1892，N2282，N2378，N2747，N2750，N2765
显式重载	Explicit Virtual Overrides	N2928
允许移动构造函数抛出异常	Allowing move constructors to throw [noexcept]	N3050
默认移动构造函数	Defining move special member functions	N3053
强 CAS 操作	Strong compare and exchange	N27485

值得注意的是，编号较小的文档通常会对相关主题的描述较多，而编号较大的文档则通常着重改善之前的特性，以及着重于如何对 C++ 标准进行修改。因此最后一篇文档常常未必是读者需要的。

D.2　其他有用的资源

在本书写作时，关于 C++11 的资源还算不上丰富（相比于它的前任 C++98/03 而言）。因此，大多数的其他资源都来自于网络。网络的缺点是链接常常会失效。不过在本书新鲜出炉的时候，相信这些链接还是有效的。

特性介绍类

- http://en.wikipedia.org/wiki/C%2B%2B11，这是 wikipedia 中关于 C++11 介绍。比较全面，如果想对所有特性进行快速学习，wiki 总是不容错过的。
- http://www.stroustrup.com/C++11FAQ.html，C++ 之父 Bjarne Stroustrup 关于 C++11 的介绍。Bjarne 会不时更新一些部分。该页面上也有中文翻译的链接。不过看起来还有不少特性 Bjarne 还没来得及写。
- *C++ Primer Plus Sixth Edition*，这是 Primer Plus 系列的第六版，其中对 C++11 有一些介绍。本书针对的是 C++ 初学者，而中文版现在也已经问世了。
- http://www.cprogramming.com/c++11/what-is-c++0x.html，这是 Alex Allain 关于 C++11 的一篇综述。不过文章末尾有一些链接，则是 Alex Allain 对 C++11 一系列的特性分别详述的文章。而且最为难能可贵的是，每一篇都保持了高质量。

❑http://zh.wikipedia.org/wiki/C++0x，这是 wiki 中文中对 C++11 的描述。跟英文版一样，保持了很好的特性分类。这也是我们推荐的为数不多的中文资源。

技术参考类

❑http://en.cppreference.com/w/，cppreference 应该是所有同类型网站中我们最喜欢的。通过搜索，读者可以找到任何关于 C++ 的特性描述、库工具等，而且大多数带有简单易懂的例子。

❑http://stackoverflow.com/，可以肯定的是，在 stackoverflow 的网上，常会有世界级的 C++ 专家出没。任何困难的 C++11 问题，基本上都可以在 stackoverflow 上搜索到相关的答案。

其他

❑http://www.open-std.org/jtc1/sc22/wg21/，WG21 的主页，读者可以在这里找到 C++ 标准委员会的邮件、文档、会议等各种信息，甚至是一些标准的草稿。

❑Adve, Gharachorloo, Shared Memory Consistency Models: A Tutorial，这是一篇介绍内存模型的非常好的论文，作者通过对多个硬件平台的比较，总结归纳了软硬件平台的内存一致性实现的方式。读者应该很容易搜索到。