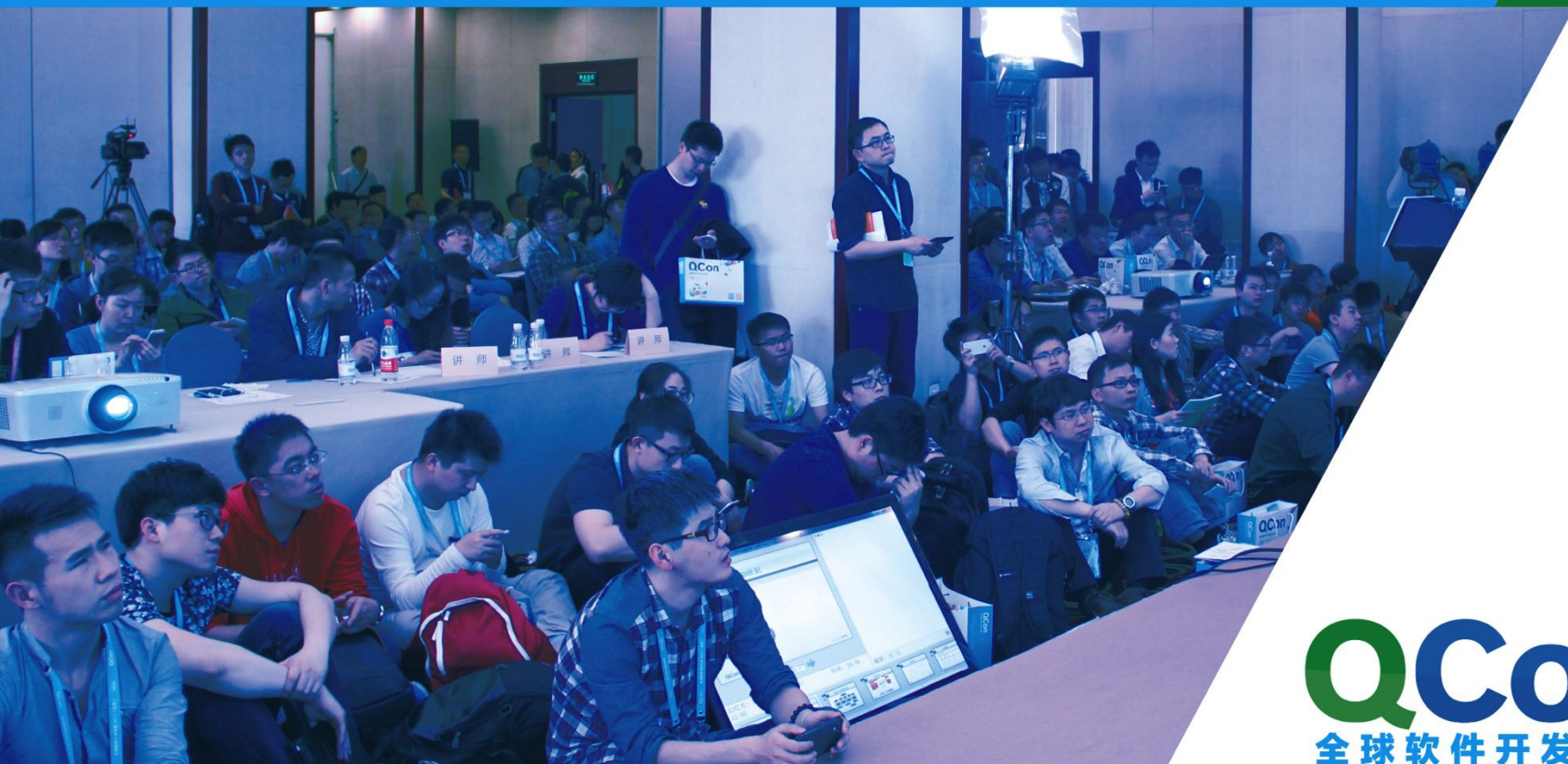


QCon全球软件开发大会

International Software Development Conference



QCon
全球软件开发大会

Geekbang>

极客邦科技

全球领先的技术人学习和交流平台

扫我，码上开启新世界



Geekbang>

InfoQ | EGO NETWORKS | StuQ

InfoQ

专注中高端技术
人员的社区媒体

EGO NETWORKS

EXTRA GEEKS' ORGANIZATION
高端技术人员
学习型社交网络

StuQ

实践驱动的IT职业
学习和服务平台



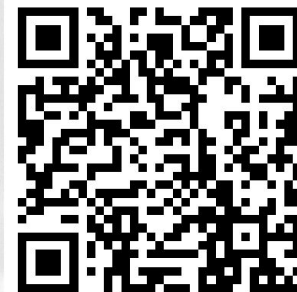
促进软件开发领域知识与创新的传播



实践第一 案例为主

时间：2015年12月18-19日 / 地点：北京·国际会议中心

欢迎您参加ArchSummit北京2015, 技术因你而不同



ArchSummit北京二维码



【北京站】

2016年04月21日-23日



关注InfoQ官方信息
及时获取QCon演讲视频信息

C++11 核心特性简析



芒果TV 林喆

C++ 11 特性列表

Contents [hide]

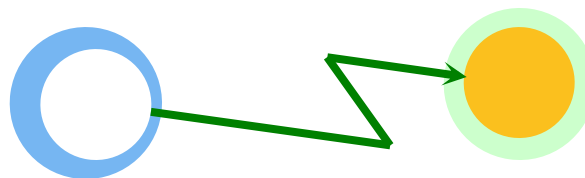
- 1 Changes from the previous version of the standard
- 2 Extensions to the C++ core language
 - 2.1 Core language runtime performance enhancements
 - 2.1.1 Rvalue references and move constructors
 - 2.1.2 constexpr – Generalized constant expressions
 - 2.1.3 Modification to the definition of plain old data
 - 2.2 Core language build-time performance enhancements
 - 2.2.1 Extern template
 - 2.3 Core language usability enhancements
 - 2.3.1 Initializer lists
 - 2.3.2 Uniform initialization
 - 2.3.3 Type inference
 - 2.3.4 Range-based for loop
 - 2.3.5 Lambda functions and expressions
 - 2.3.6 Alternative function syntax
 - 2.3.7 Object construction improvement
 - 2.3.8 Explicit overrides and final
 - 2.3.9 Null pointer constant
 - 2.3.10 Strongly typed enumerations
 - 2.3.11 Right angle bracket
 - 2.3.12 Explicit conversion operators
 - 2.3.13 Template aliases
 - 2.3.14 Unrestricted unions
 - 2.4 Core language functionality improvements
 - 2.4.1 Variadic templates
 - 2.4.2 New string literals

取自维基百科
呃 似乎有点太多了 ... 
就挑一点聊聊

核心特性

- 转移语义与右值引用

- 更安全和快速的对象资源管理



- Lambda 与 `std::function`

- 令开发更便捷, 提高 STL 算法库的出镜率

[] { }

- 多线程

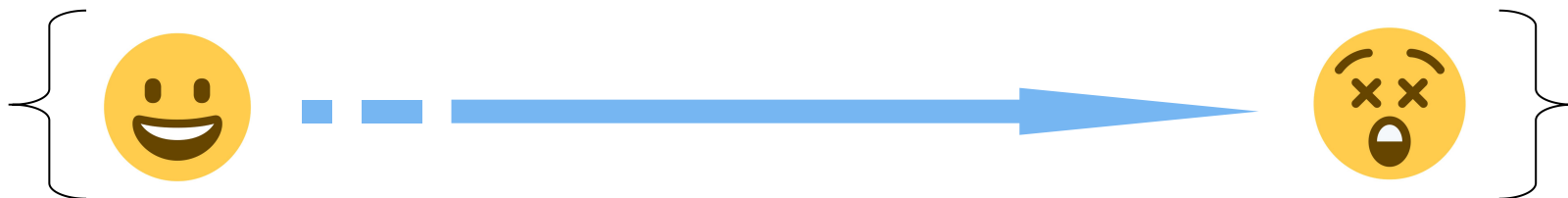
- 让很快的程序变得更快



C++ 本身的核心特性是？

所有对象在生命周期结束时会被销毁

典型地, 栈对象在函数结束后析构函数被调用



所谓生命周期结束

```
std::vector<int> input_values()
{
    std::vector<int> r;
    std::fstream inp("values.txt", std::fstream::in);
    int i;
    while (inp >> i) {
        if (i < 0) {
            throw std::runtime_error("only positive integer allowed");
        }
        r.push_back(i);
    }
    return r;
}
```

无论是函数**正常返回**还是**抛出异常**退出
r 都会执行析构而释放其堆内存
inp 都会析构而释放其使用的文件句柄

然而问题来了

```
std::vector<int> input_values()
{
    std::vector<int> r;
    std::fstream inp("values.txt", std::fstream::in);
    int i;
    while (inp >> i) {
        /* ... */
    }
    return r;
}
```

既然函数退出时栈对象都销毁了，
怎么向调用栈外传递返回值？

愚蠢的问题

与并不聪明的解答

```
std::vector<int> input_values()
{
    std::vector<int> r;
    std::fstream inp("values.txt", std::fstream::in);
    int i;
    while (inp >> i) {
        /* ... */
    }
    return r;
}
```



复制构造函数

将整个容器内容复制一份

并不是所有东西都能复制

能复制的东西也尽量不要复制

```
void input_values(std::vector<int>& r)
{
    // r.clear();
    ASSERT (r.empty());
    std::fstream inp("values.txt", std::fstream::in);
    int i;
    while (inp >> i) {
        if (i < 0) {
            r.clear();
            throw std::runtime_error("only positive integer allowed");
        }
        r.push_back(i);
    }
}
```

虽然大部分编译器声称能进行返回值优化 (RVO)
但是程序员往往喜欢自顾自地玩幺蛾子



于是 C++11 决定...

干脆让

对象

可以

转移

在返回值外面套个 `std::move`

```
std::vector<int> input_values()
{
    std::vector<int> r;
    std::fstream inp("values.txt", std::fstream::in);
    int i;
    while (inp >> i) {
        if (i < 0) {
            throw std::runtime_error("only positive integer allowed");
        }
        r.push_back(i);
    }
    return std::move(r);
}
```

此处会匹配并调用类的**转移构造函数**

转移构造函数实现举例

```
struct vector {  
    pointer _M_start;  
    pointer _M_finish;  
    pointer _M_end_of_storage;  
  
    vector(vector&& rhs) /* 转移构造函数 T(T&& rhs) */  
        noexcept        /* 声明此函数不会抛出异常 */  
    {  
        std::swap(this->_M_start, rhs._M_start);  
        std::swap(this->_M_finish, rhs._M_finish);  
        std::swap(this->_M_end_of_storage, rhs._M_end_of_storage);  
    }  
};
```

某些 STL 实现中转移构造函数的典型实现方式
虽然使用 `std::swap`, 然而因为 `this` 是新构造的对象
故函数结束后, `rhs` 的值被置换为了一个空容器
而 `this` 则持有原容器的数据

这个设计是不是眼熟...

```
std::auto_ptr<int> a(new int(91));  
std::auto_ptr<int> b(a);
```

```
std::cout << *b << std::endl; // 91  
std::cout << (a.get() == NULL) << std::endl; // 1
```

在上古时期, `std::auto_ptr` 的设计正是这么干的
其 "复制" 构造函数会将参数重置为空
类似这样

```
auto_ptr(auto_ptr& rhs)  
{  
    std::swap(this->_ptr, rhs._ptr);  
}
```

std::move 的作用

```
std::vector<int> a;  
std::vector<int> b(a);
```



```
vector(const vector&);
```

```
std::vector<int> c(std::move(a));
```

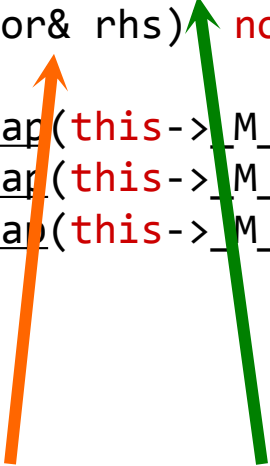


```
vector(vector&&);  
template class<T>  
std::move(T)
```

std::move 的作用只是将左值引用转换为右值引用
以匹配接受右值引用的重载
它不会对参数对象造成任何额外的影响

右值引用与左值的区别？

```
struct vector {  
    vector(vector&& rhs) noexcept  
    // vector(vector& rhs) noexcept  
    {  
        std::swap(this->_M_start, rhs._M_start);  
        std::swap(this->_M_finish, rhs._M_finish);  
        std::swap(this->_M_end_of_storage, rhs._M_end_of_storage);  
    }  
};
```



有何区别？

右值引用可以作为对临时对象的引用

```
std::vector<int> input_values();
```

```
// -----
```

```
std::vector<int>& lref = input_values();
```



```
const std::vector<int>& cref = input_values();
```



旧标准的做法, 不区分临时引用
和 const 限定引用

```
/* 为了让返回值可以直接传给另一个函数 */
```

```
void another_fn(const std::vector<int>& x);  
another_fn( input_values() );
```

右值引用可以作为对临时对象的引用

```
std::vector<int> input_values();
```

```
// -----
```

```
/* 新标准中, 函数返回的对象可以由右值引用接管 */
```

```
std::vector<int>&& rvalue_ref = input_values();
```

临时对象匹配的重载为转移构造函数

```
std::vector<int> input_values();
```

```
// -----
```

```
std::vector<int> values ( input_values() );
```

匹配到转移构造函数, 不会复制

有何好处?

- 转移比复制更快
- 有些对象并不能复制, 只能转移
- 还有其它好处吗?

自动指针与容器的不兼容

`std::vector<std::auto_ptr<?> >` 是无法编译的
因为 `vector<T>` 要求 `T` 有复制构造函数 `T(const T& rhs)`

但 `auto_ptr` 的 "复制" 构造函数声明为

`auto_ptr(auto_ptr&);`
并不与之兼容

这样一来的后果是...

带来语义理解的负担

```
class MultiFiles {  
    /* 如果能这样写是极好的  
       阅读代码的时候一目了然  
       input_streams 内的元素由 MultiFiles 掌控生命周期  
       然而由于不能这样写  
    */  
    std::vector<std::auto_ptr<std::istream> > input_streams;  
  
    /* 只能退而求编译通过, 写成这样  
       可能必须读了 MultiFiles 的析构函数才会知道 (同时增加析构函数代码量)  
       所有的元素是随着 MultiFiles 析构而死掉的  
    */  
    std::vector<std::istream*> input_streams;  
};
```

而所有的这一切已经有了新标准

新的模板类型 `std::unique_ptr` 替代 `std::auto_ptr` 成为标准库中的自动内存管理类型, 支持转移构造, 且**不支持**复制构造

```
std::vector<std::unique_ptr<std::istream>> input_streams;
```

```
std::unique_ptr<std::istream> s(new std::ifstream(/* ... */));  
input_streams.push_back(std::move(s));
```

`std::vector` 在添加元素时将默认使用其转移构造函数
如果没有转移构造函数, 才进行复制

总之右值引用与转移语义是 ...

为了填上 C++ 无脑复制的坑
有了**转移语义/转移构造**

为了使用转移语义
有了**右值引用**
顺便区分了 const 限定引用与临时对象引用

C++ 易用性改善

- 更简洁的函数对象
 - Lambda
 - `std::function` 对象

[] () { }

谁会去用 STL 算法函数

- 找一个比指定数大的值, 你可曾用过 `std::find_if`

```
struct GreaterThan {  
    int value;  
  
    GreaterThan(int v)  
        : value(v)  
    {}  
  
    bool operator()(int item)  
    {  
        return item > value;  
    }  
};
```

```
std::vector<int> m;  
int base_value;  
  
std::vector<int>::iterator i =  
    std::find_if(  
        m.begin(), m.end(),  
        GreaterThan(base_value));  
if (i != m.end()) {  
    std::cout << *i << std::endl;  
}
```

惰性使得人们总是用最容易写的方法

- 用 for 循环不知道简单到哪里去了

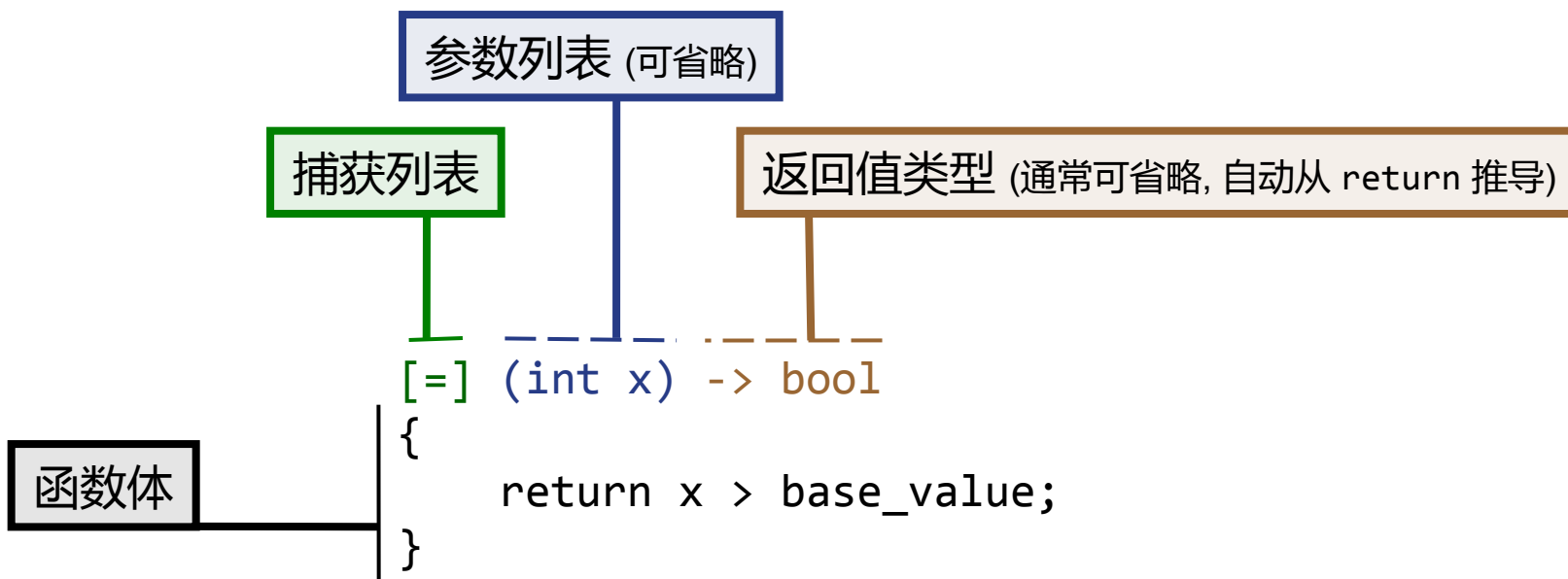
```
std::vector<int> m;  
int base_value;  
  
for (std::vector<int>::iterator i = m.begin(); i < m.end(); ++i) {  
    if (*i > base_value) {  
        std::cout << *i << std::endl;  
        break;  
    }  
}
```

使用 C++11 的 Lambda 表达式

- 新标准中的 Lambda 不再需要写那么严肃的 struct

```
std::vector<int> m;  
int base_value;  
  
std::vector<int>::iterator i = std::find_if(  
    m.begin(), m.end(),  
    [=](int x) -> bool  
        { return x > base_value; });  
if (i != m.end()) {  
    std::cout << *i << std::endl;  
}
```

Lambda 表达式语法



Lambda 捕获上下文

捕获列表

- [] 不捕获任何上下文
- [&] 以引用方式捕获上下文
- [=] 以复制方式捕获上下文
- [this] 捕获 this
- [=x, &y] 以复制方式捕获 x, 以引用方式捕获 y
- ...

以转移方式捕获上下文是在 C++14 中加入的

Lambda 的内存模型

在 C++ 中任何看起来很黑科技的东西, 第一个问题都会是这个对象以及它引用的东西是怎么存在于内存中的

```
int base_value;  
auto f = [=](int x) -> bool  
        { return x > base_value; };  
  
sizeof f; /* 会与 sizeof int 相同 */
```

Lambda 的尺寸等同于其捕获的上下文变量的尺寸
实际上, 它的实现就是一个匿名的自动构造的结构体
结构体的成员包含所有捕获的上下文变量, 以及一个 `operator()` 重载

Lambda 好用, 但是无法直接返回

```
???? fn() /* 此函数的返回值类型是无法定义的 */  
{  
    int base_value;  
    auto f = [=](int x) -> bool  
            { return x > base_value; };  
    return f; /* 无法达成 */  
}
```

如果想要将 lambda 以及其捕获的上下文返回到函数外
则必须使用 `std::function`

使用 `std::function` "固化" Lambda

```
std::function<bool(int)> fn()  
{  
    int base_value = 1729;  
    auto f = [=](int x) -> bool  
        { return x > base_value; };  
    return f;  
}
```

`std::function< RETURN_TYPE (ARGS_TYPES...) >`

是 C++ 新标准中引入新的模板类型
支持从任何**看起来像函数**的东西构造, 语法简单, 使用灵活

当作轻量级虚函数使用

● 便捷的书写

- 不用再定义基类, 虚函数, 并在子类中覆盖/实现这些函数
- 直接引用上下文变量, 不用复制代码到每一个子类中当作成员
- 调用处就能定义, 而不用把类单独引出到其他地方

std::function 用作策略函数

```
struct Person {  
    std::string name;  
    int age;  
};  
  
std::map<std::string, std::function<bool(const Person&, const Person&>> m;  
m["name"] = [](const Person& lhs, const Person& rhs) {  
    return lhs.name < rhs.name;  
};  
m["age"] = [](const Person& lhs, const Person& rhs) {  
    return lhs.age < rhs.age;  
};
```

// 之后可以根据输入参数进行排序之类的操作

```
std::vector<Person> persons;  
std::sort(persons.begin(), persons.end(), m.at("name"));
```

多线程与锁



`std::thread`



`std::mutex`



`std::lock_guard`

线程对象

- 相对于其他语言的线程, C++11 的线程一点也不会闲着
 - 构造参数为一个可调用的对象 (或不传, 形成一个空线程, 也不会执行)
 - 构造后线程立即开始执行

```
int main()
{
    std::thread th(
        []() { std::cout << "I'm a thread." << std::endl; });
    th.join();
    return 0;
}
```

线程对象也是对象

- 栈上线程对象在函数结束时也会析构
 - 如果线程对象在执行时发生析构 程序会死

```
int main()
{
    std::thread th(
        []() { std::cout << "I'm a thread." << std::endl; });
    th.join();
    return 0;
}
```

std::terminate

线程对象生命周期管理

```
void start_thread()
{
    std::thread th(
        []() { std::cout << "I'm a thread." << std::endl; });
    th.detach();
}

int main()
{
    start_thread();
    other_stuff();
    return 0;
}
```

一旦 thread 对象的 `detach` 方法被调用
线程即被托管为后台线程, 无法再调用 `join`
此后即使线程对象被析构, 线程也将继续执行

可转移的线程对象

```
std::thread start_thread()
{
    std::thread th(
        []() { std::cout << "I'm a thread." << std::endl; });
    return std::move(th);
}

int main()
{
    std::thread th(start_thread());
    other_stuff();
    th.join();
    return 0;
}
```

将线程对象转移到需要控制的地方
甚至是 STL 容器中

C++11 并发锁

- 手动加锁, 解锁

```
std::vector<std::unique_ptr<std::istream>> input_streams;  
std::mutex streams_mutex;
```

```
void add_stream(std::unique_ptr<std::istream> s)  
{  
    streams_mutex.lock();  
    input_streams.push_back(std::move(s));  
    streams_mutex.unlock();  
}
```

对加锁行为应用自动生命周期管理

- 对锁实施自动的生命周期管理, 在对象析构时自动释放

```
std::vector<std::unique_ptr<std::istream>> input_streams;  
std::mutex streams_mutex;
```

```
void add_stream(std::unique_ptr<std::istream> s)  
{  
    std::lock_guard<std::mutex> __guard__(streams_mutex);  
    input_streams.push_back(std::move(s));  
}
```

总结 - 对象生命周期

所有对象在生命周期结束时会被销毁

右值引用和转移语义使对象资源能灵活转移了
C++ 特色的 `lock_guard` 利用析构函数自动解锁

总结 - Lambda 与函数对象

C++ 正在向更灵活更先进的方向发展

标准库中的算法函数从此不再是鸡肋
可以用函数对象代替继承-虚函数体制

THANKS

Brought by **InfoQ**

International Software Development Conference