VC10 中的 C++0x 特性

简要介绍：
这一系列文章来自 Visual C++ Team Blog，介绍 Microsoft Visual Studio 2010 中支持的 C++0x 特性，目前有三部分。
Part 1：介绍了 Lambdas，auto，以及 static_assert；
Part 2：介绍了右值引用（Rvalue References）；
Part 3：介绍了表达式之类型（decltype）。

由于水平有限，翻译中的错误之处，恳请指正。译文在这里 我的博客 或这里 译言 上找到，也可以发电子邮件给我：kesalin@gmail.com 。

飘飘白云 2009 年 6 月 3 日星期三

由于未获原作者许可，翻译此文仅为学习交流之用。转载时请注明原作者，译者以及出处等信息。未经许可，请勿用于商业用途。

VC10 中的 C++0x 特性 Part 1：Lambdas，auto，以及 static_assert

来源：vcblog　翻译：飘飘白云 kesalin@gmail.com
原文链接：点击这里

Microsoft Visual Studio 2010 九月社区技术预览版 (CTP)所带的 Visual C++编译器对四个 C++0x 语言特性提供了支持，也就是 lambdas，auto，static_assert，以及 rvalue references （右值引用，译注：后面不再对这个词进行翻译）。今天，我将详细介绍前三个特性。（很快我将贡献一整篇幅的文章来解释右值引用，仅仅是因为再在这里解释的 话将会加大这篇已经很长的文章的篇幅）

首先，说明一些事情：

1, 今天的这篇文 章是由 Stephan T. Lavavej，Visual C++库的开发人员以及 C, A, 与 T 读者投书栏带给你们的。注意作为库的开发人员，我并没有实现这些特性。那是 Jonathan Caves，前端编译器开发者，选举标准委员会成员以及所有“忍者”（鲜为人知的高手）的成果。

2, 我将 Visual C++ compiler in VS 2010 简称为 VC10 （ VS 2008 包含 VC9，VS 2005 包含 VC8，等等。 - 10 并不比 2010 简短）

3, C++0x 指的是即将到来的 C++ 标准，现在还在起草中。（C++标准委员会希望它可以在 2009 年完成，称作 C++ 09；玩笑话说如果它推迟到 2010 或者更晚的话，“x” 将是十六进制的了）。C++98 和 C++03 指的是当前的 C++ 标准。（在这里不回顾历史了， C++ 标准 2003 仅仅是最初的 C++1998 标准 的“补丁”版，对大部分人来说可以忽略两者间的区别。C++ 03 和 C++ 0x 模样虽然看起来差不多，但完全不同）

4, 我要感谢标准委员会开发出这些奇妙而有用并富有艺术的特性。他们也在以下站点上提供了重要的文档：
C++0x 语言特性:
http://open-std.org/JTC1/SC22/WG21/docs/papers/2008/n2705.html
C++0x 库特性:
http://open-std.org/JTC1/SC22/WG21/docs/papers/2008/n2706.html
C++0x 进行中的草案:
http://open-std.org/JTC1/SC22/WG21/docs/papers/2008/n2798.pdf

5. 总是会有 bug 的（虽然希望不会太多），这也就是发布 CTP 版本的主要目的（让用户测试发现 bug ）。请通过 Microfsoft 把这些 bug 报告给我们。

现在，让我们来审视这些特性吧！

Lambdas

在 C++ 0x 中，**“lambda 表达式”隐式定义并构建不具名函数对象**，这些对象就像手写函数对象一样。下面是 lambda “Hello，World”入门级的示例：

```
C:\Temp>type meow.cpp
#include <algorithm>
#include <iostream>
#include <ostream>
#include <vector>
using namespace std;

int main() {
    vector<int> v;

    for (int i = 0; i < 10; ++i) {
        v.push_back(i);
    }

    for_each(v.begin(), v.end(), [](int n) { cout << n << "
"; });
    cout << endl;
}
C:¥Temp>cl /EHsc /nologo /W4 meow.cpp > NUL && meow
0 1 2 3 4 5 6 7 8 9
```

[] 操作符是 lambda 导引符， 它告诉编译器一个 lambda 表达式开始了。 (int n) 是 lambda 参数声明，它告诉编译器不具名函数对象类的函数调用操作符带有哪些参数， { cout << n << ″ ″; } 是复合声明，它是不具名函数对象类的函数调用操作符的函数体。不具名函数对象类的函数调用操作符默认返回 void。

这样，C++0x 在内部将它转换成如你在 C++ 98 下编写的一样代码：

```
C:¥Temp>type meow98.cpp
#include <algorithm>
#include <iostream>
#include <ostream>
#include <vector>
using namespace std;

struct LambdaFunctor {
    void operator()(int n) const {
        cout << n << " ";
    }
};

int main() {
    vector<int> v;

    for (int i = 0; i < 10; ++i) {
        v.push_back(i);
    }

    for_each(v.begin(), v.end(), LambdaFunctor());
    cout << endl;
}

C:¥Temp>cl /EHsc /nologo /W4 meow98.cpp > NUL && meow98
0 1 2 3 4 5 6 7 8 9
```

现在我将不再累述类似“不具名函数对象类的函数调用操作符默认返回 void”这样的话，开始换用“lambda 函数返回 void”的说法，但是**记住 lambda 表达式做了些什么是很重要的，那就是：定义类并构建对象**。

当然，lambda 的复合声明部分（函数体部分）可以包含多个声明语句，譬如：

```
C:¥Temp>type multimeow.cpp
#include <algorithm>
#include <iostream>
#include <ostream>
#include <vector>
using namespace std;

int main() {
    vector<int> v;

    for (int i = 0; i < 10; ++i) {
        v.push_back(i);
    }

    for_each(v.begin(), v.end(), [](int n) {
        cout << n;

        if (n % 2 == 0) {
            cout << " even ";
        } else {
            cout << " odd ";
        }
    });

    cout << endl;
}

C:¥Temp>cl /EHsc /nologo /W4 multimeow.cpp > NUL && multimeow
0 even 1 odd 2 even 3 odd 4 even 5 odd 6 even 7 odd 8 even 9
odd
```

lambda 函数也并不总是必须返回 void。如果 lambda 的复合声明语句像是这样的 { return expression; } ，那么 lambda 的返回类型就会自动地被推断成 expression 的类型。

```
C:¥Temp>type cubicmeow.cpp
#include <algorithm>
#include <deque>
#include <iostream>
#include <iterator>
#include <ostream>
#include <vector>
using namespace std;

int main() {
    vector<int> v;

    for (int i = 0; i < 10; ++i) {
        v.push_back(i);
    }

    deque<int> d;

    transform(v.begin(), v.end(), front_inserter(d), [](int
n) { return n * n * n; });

    for_each(d.begin(), d.end(), [](int n) { cout << n << "
"; });
    cout << endl;
}

C:¥Temp>cl /EHsc /nologo /W4 cubicmeow.cpp > NUL && cubicmeow
729 512 343 216 125 64 27 8 1 0
```

在这里， n * n * n 的类型是 int，所以 lambda 函数返回 int。
有着复杂复合声明语句的 lambda 函数不会自动推断返回类型，你必须显式指定返回类型。

```
C:¥Temp>type returnmeow.cpp
#include <algorithm>
#include <deque>
```

```
#include <iostream>
#include <iterator>
#include <ostream>
#include <vector>
using namespace std;

int main() {
    vector<int> v;

    for (int i = 0; i < 10; ++i) {
        v.push_back(i);
    }

    deque<double> d;

    transform(v.begin(), v.end(), front_inserter(d), [](int
n) -> double {
        if (n % 2 == 0) {
            return n * n * n;
        } else {
            return n / 2.0;
        }
    });

    for_each(d.begin(), d.end(), [](double x) { cout << x <<
" "; });
    cout << endl;
}

C:¥Temp>cl /EHsc /nologo /W4 returnmeow.cpp > NUL && returnmeow
4.5 512 3.5 216 2.5 64 1.5 8 0.5 0
```

-> double 是可选的 lambda 返回类型从句。为什么它不放在左边（译注：返回类型一般在函数左边声明），就像程序员一直以来在 C 函数中做的那样？因为那样的话 lambda 导引符 [] 就不会第一个出现了，而正是它告诉编译器一个 lambda 函数开始了。（核心工作组最擅长解决这样的问题；尝试猜测 C++ 中一个

给定的概念是否是可被解析的会让我头疼。）

如果忘记了指定 lambda 返回类型从句，编译器就会抱怨每一个返回语句：

```
C:¥Temp>cl /EHsc /nologo /W4 borkedreturnmeow.cpp
borkedreturnmeow.cpp
borkedreturnmeow.cpp(20) : error C3499: a lambda that has been
specified to have a void return type cannot return a value
borkedreturnmeow.cpp(22) : error C3499: a lambda that has been
specified to have a void return type cannot return a value
```

到目前为止我所介绍的 lambda 都是无状态的：它们不包含数据成员。你也可以有有状态的 lambda，这是通过“传递”（原文用加引号的 capturing 这个词，在这里我翻译成传递似乎不太妥，故我都加括号引用原文，下同）局部变量来实现的。空的 lambda 导引符 [] 意味着“一个无状态的 lambda”，但在 lambda 导引符 [] 中你可以指定 capture-list ：

```
C:\Temp>type capturekittybyvalue.cpp
#include <algorithm>
#include <iostream>
#include <ostream>
#include <vector>
using namespace std;

int main() {
    vector<int> v;

    for (int i = 0; i < 10; ++i) {
        v.push_back(i);
    }

    int x = 0;
    int y = 0;

    // op>>() leaves newlines on the input stream,
    // which can be extremely confusing. I recommend
```

```
    // avoiding it, and instead using non-member
    // getline(cin, str) to read whole lines and
    // then parse them. But in the interests of
    // brevity, I'll use evil op>>():

    cout << "Input: ";
    cin >> x >> y;

     v.erase(remove_if(v.begin(),  v.end(),  [x,  y](int  n)
{ return x < n && n < y; }), v.end());

    for_each(v.begin(), v.end(), [](int n) { cout << n << "
"; });
    cout << endl;
}

C:¥Temp>cl /EHsc /nologo /W4 capturekittybyvalue.cpp > NUL &&
capturekittybyvalue
Input: 4 7
0 1 2 3 4 7 8 9
```

如果你忘记了 capture-list，编译器就会抱怨：

```
C:¥Temp>cl /EHsc /nologo /W4 borkedcapturekittybyvalue.cpp
borkedcapturekittybyvalue.cpp
borkedcapturekittybyvalue.cpp(27) : error C3493: 'x' cannot be
implicitly captured as no default capture mode has been
specified
borkedcapturekittybyvalue.cpp(27) : error C3493: 'y' cannot be
implicitly captured as no default capture mode has been
specified
```

（我很快就会解释默认的传递（capture））

记着，lambda 表达式隐式地定义了一个不具名函数对象类。复合声明语句 { return x < n && n < y; } 在这个类中被当作函数调用操作符的函数体。虽然

从词法结构上看复合声明语句是在 main() 块之内，但在概念上它是在 main() 块之外的，因此如果不传递（capture）到 lambda 中去，就不能在其中使用来自 main() 中的局部变量。

上面的代码在内部被翻译成：

```
C:¥Temp>type capturekittybyvalue98.cpp
#include <algorithm>
#include <iostream>
#include <iterator>
#include <ostream>
#include <vector>
using namespace std;

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;
};

int main() {
    vector<int> v;

    for (int i = 0; i < 10; ++i) {
        v.push_back(i);
    }

    int x = 0;
    int y = 0;

    cout << "Input: ";
```

```
    cin >> x >> y; // EVIL!

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

    copy(v.begin(), v.end(), ostream_iterator<int>(cout, "
"));
    cout << endl;
}

C:¥Temp>cl /EHsc /nologo /W4 capturekittybyvalue98.cpp > NUL
&& capturekittybyvalue98
Input: 4 7
0 1 2 3 4 7 8 9
```

在这里你可以清楚 地看到是“按值”传递（captures）的。函数对象存储了局部变量的拷贝。这就使得函数对象可以比通过传递（capture）来创建它们的局部变量有 更长的生命期。但是，要注意：(a)在 lambda 中不能修改通过传递(capture)获得的拷贝，因为默认情况下函数调用操作符是 const 属性的，（b）一些对象的拷贝开销是昂贵的，（c）局部变量的更新不会反应到通过传递（capture）获得的拷贝（在语义上它们是原始值）。很快我就会 解释如有需要应该如何来处理以上情况。

但是首先，你可以 “按值传递（capture）任何东西”，而不用特别指明每一个你想要传递(capture)的局部变量。其语法是使用这种形式的 lambda 导引符 [=]（默认传递（capture-default） = 应该可以让你想起赋值或者拷贝初始化 Foo foo = bar; 虽然这里的拷贝实际上是直接初始化（通过初始化列表进行赋值），就像上面的 m_a(a)）。

```
C:¥Temp>type defaultcapturekittybyvalue.cpp
#include <algorithm>
#include <iostream>
#include <ostream>
#include <vector>
using namespace std;
```

```
int main() {
    vector<int> v;

    for (int i = 0; i < 10; ++i) {
        v.push_back(i);
    }

    int x = 0;
    int y = 0;

    cout << "Input: ";
    cin >> x >> y; // EVIL!

    v.erase(remove_if(v.begin(), v.end(), [=](int n) { return
x < n && n < y; }), v.end());

    for_each(v.begin(), v.end(), [](int n) { cout << n << "
"; });
    cout << endl;
}

C:¥Temp>cl /EHsc /nologo /W4 defaultcapturekittybyvalue.cpp >
NUL && defaultcapturekittybyvalue
Input: 4 7
0 1 2 3 4 7 8 9
```

当编译器看到 lambda 中的 x 和 y， 就会从 main() 中按值传递（capture）。

情形(a)要修改通过传递(capture)获得拷贝该怎样呢？默认情况下，一个 lambda 函数调用操作符是 const 属性的，但是可以通过使用 mutable 把它变成 non-const。

```
C:¥Temp>type capturekittybymutablevalue.cpp
#include <algorithm>
#include <iostream>
#include <ostream>
```

```
#include <vector>
using namespace std;

int main() {
    vector<int> v;

    for (int i = 0; i < 10; ++i) {
        v.push_back(i);
    }

    int x = 1;
    int y = 1;

    for_each(v.begin(), v.end(), [=](int& r) mutable {
        const int old = r;

        r *= x * y;

        x = y;
        y = old;
    });


    for_each(v.begin(), v.end(), [](int n) { cout << n << "
"; });
    cout << endl;

    cout << x << ", " << y << endl;
}

C:¥Temp>cl /EHsc /nologo /W4 capturekittybymutablevalue.cpp >
NUL && capturekittybymutablevalue
0 0 0 6 24 60 120 210 336 504
1, 1
```

这里是依次将 v 中前两个元素相乘。（我得承认真的很难想出一个不

用 partial_sum() 或 adjacent_difference() 表示的例子，partial_sum() 是与前面所有的元素相乘，adjacent_difference()是与前一个元素相乘）。注意到情形（d），对通过传递获得的拷贝的更新操作并没有影 响局部变量（再一次，原始值语义）。

如果你想处理情形（b），（c）和（d）：避免拷贝，在 lambda 中观察局部变量的更新，以及在 lambda 中修改局部变量又该怎么做呢？在这种情况下，你会想通过引用传递（capture by reference）。其语法是这种形式的 lambda 导引符 [&x, &y] （你可以把它想象成 X& x, Y& y ；那是“引用”而不是“取址”）。

```
C:¥Temp>type capturekittybyreference.cpp
#include <algorithm>
#include <iostream>
#include <ostream>
#include <vector>
using namespace std;

int main() {
    vector<int> v;

    for (int i = 0; i < 10; ++i) {
        v.push_back(i);
    }

    int x = 1;
    int y = 1;

    for_each(v.begin(), v.end(), [&x, &y](int& r) {
        const int old = r;

        r *= x * y;

        x = y;
        y = old;
    });
```

```
    for_each(v.begin(), v.end(), [](int n) { cout << n << " "; });
    cout << endl;

    cout << x << ", " << y << endl;
}
C:¥Temp>cl /EHsc /nologo /W4 capturekittybyreference.cpp > NUL && capturekittybyreference
0 0 0 6 24 60 120 210 336 504
8, 9
```

注意与 capturekittybymutablevalue.cpp 的区别：（1），lambda 导引符是 [&x, &y]，（2）没有 mutable，（3），局部变量 x 和 y 最后的值是 8 和 9，反应了在 lambda 中对他们的修改。

上面的代码在内部被翻译成：

```
C:¥Temp>type capturekittybyreference98.cpp
#include <algorithm>
#include <iostream>
#include <iterator>
#include <ostream>
#include <vector>
using namespace std;

#pragma warning(push)
#pragma warning(disable: 4512) // assignment operator could not be generated

class LambdaFunctor {
public:
    LambdaFunctor(int& a, int& b) : m_a(a), m_b(b) { }

    void operator()(int& r) const {
        const int old = r;
```

```
        r *= m_a * m_b;

        m_a = m_b;
        m_b = old;
    }

private:
    int& m_a;
    int& m_b;
};

#pragma warning(pop)

int main() {
    vector<int> v;

    for (int i = 0; i < 10; ++i) {
        v.push_back(i);
    }

    int x = 1;
    int y = 1;

    for_each(v.begin(), v.end(), LambdaFunctor(x, y));

    copy(v.begin(), v.end(), ostream_iterator<int>(cout, "
"));
    cout << endl;

    cout << x << ", " << y << endl;
}
C:¥Temp>cl /EHsc /nologo /W4 capturekittybyreference98.cpp >
NUL && capturekittybyreference98
0 0 0 6 24 60 120 210 336 504
8, 9
```

当你通过引用传递（capture）局部变量时，函数对象存储局部变量的引用。这样避免了拷 贝，能够让函数对象观测对局部变量的更新，以及能够让函数对象通过引用修改局部变量。（注意函数调用操作符是 const 属性的，因为我们没有把它声明成 mutable，但这只阻止我们修改函数对象的数据成员。在这里数据成员是引用，我们无法修改它，但是我们可以修改它所指向的东西。函数调用操作符的常量 性是,且一直都是，表面的）

当然，如果函数对象比用于引用传递的局部变量的生命期更长的话，程序会毁灭性地崩溃。（译注：就是说引用所指向的东西已经销毁了，但你可能还在使用这个引用）

同样，你可以使用默认的按引用传递语法；这种形式 lambder 导引符 [&] 表明“按引用传递”。

如果你想混合使用按值传递和按引用传递该怎么做呢？你可以像这样声明 [a, b, c, &d, e, &f, g]，但你也可以指定默认按值传递，再为特定的（需要按引用传递的）局部变量而重载它。下面是一个修改 capturekittybymutablevalue.cpp 而来的例子：

```
C:\Temp>type overridekitty.cpp
#include <algorithm>
#include <iostream>
#include <ostream>
#include <vector>
using namespace std;

int main() {
    vector<int> v;

    for (int i = 0; i < 10; ++i) {
        v.push_back(i);
    }

    int sum = 0;
    int product = 1;
```

```
    int x = 1;
    int y = 1;

    for_each(v.begin(), v.end(), [=, &sum, &product](int& r)
mutable {
        sum += r;

        if (r != 0) {
            product *= r;
        }

        const int old = r;

        r *= x * y;

        x = y;
        y = old;
    });

    for_each(v.begin(), v.end(), [](int n) { cout << n << "
"; });
    cout << endl;

    cout << "sum: " << sum << ", product: " << product << endl;
    cout << "x: " << x << ", y: " << y << endl;
}

C:¥Temp>cl   /EHsc   /nologo   /W4   overridekitty.cpp   &&
overridekitty
overridekitty.cpp
0 0 0 6 24 60 120 210 336 504
sum: 45, product: 362880
x: 1, y: 1
```

在这里，我们想通过按值传递 x 和 y （因为我们只想在 lambda 内部修改它们，而不影响外部），同时我们想按传引用捕获 sum 和 product（因为我们确实想让

对它们的修改在外部也生效）。像这样的 lambda 导入符组合 [&, x, y] 可以起到同样的效果（按引用传递所有参数，除了 x 和 y 是按值传递之外）

嗯，如果你想这么做又该如何？

```
C:¥Temp>type memberkitty.cpp
#include <algorithm>
#include <iostream>
#include <ostream>
#include <vector>
using namespace std;

class Kitty {
public:
    explicit Kitty(int toys) : m_toys(toys) { }

    void meow(const vector<int>& v) const {
        for_each(v.begin(), v.end(), [m_toys](int n) {
            cout << "If you gave me " << n << " toys, I would
have " << n + m_toys << " toys total." << endl;
        });
    }

private:
    int m_toys;
};

int main() {
    vector<int> v;

    for (int i = 0; i < 3; ++i) {
        v.push_back(i);
    }

    Kitty k(5);
```

```
    k.meow(v);
}


C:¥Temp>cl /EHsc /nologo /W4 memberkitty.cpp
memberkitty.cpp
memberkitty.cpp(12) : error C3480: 'Kitty::m_toys': a lambda
capture variable must be from an enclosing function scope
```

lambda 表达式语法允许你传递局部变量，但是数据成员不是局部变量。通过特别的方法，你也可以传递数据成员：

```
C:¥Temp>type workingmemberkitty.cpp
#include <algorithm>
#include <iostream>
#include <ostream>
#include <vector>
using namespace std;

class Kitty {
public:
    explicit Kitty(int toys) : m_toys(toys) { }

    void meow(const vector<int>& v) const {
        for_each(v.begin(), v.end(), [this](int n) {
            cout << "If you gave me " << n << " toys, I would
have " << n + m_toys << " toys total." << endl;
        });
    }

private:
    int m_toys;
};

int main() {
    vector<int> v;
```

```
    for (int i = 0; i < 3; ++i) {
        v.push_back(i);
    }

    Kitty k(5);
    k.meow(v);
}

C:¥Temp>cl /EHsc /nologo /W4 workingmemberkitty.cpp > NUL &&
workingmemberkitty
If you gave me 0 toys, I would have 5 toys total.
If you gave me 1 toys, I would have 6 toys total.
If you gave me 2 toys, I would have 7 toys total.
```

当你传递 this，你可以使用 m_toys，它隐含意味着 this->m_toys，就如平常一样（译注：作为类自身成员变量）。你也可以显式使 用 this->m_toys。（在一个 lambda 表达式内，只有当你传递了 this，你才能使用它；你永不可能取得 lambda 对象它自身的 this 指针）。

你也可以隐式地传递 this ：

```
C:¥Temp>type implicitmemberkitty.cpp
#include <algorithm>
#include <iostream>
#include <ostream>
#include <vector>
using namespace std;

class Kitty {
public:
    explicit Kitty(int toys) : m_toys(toys) { }

    void meow(const vector<int>& v) const {
        for_each(v.begin(), v.end(), [=](int n) {
            cout << "If you gave me " << n << " toys, I would
```

```
have " << n + m_toys << " toys total." << endl;
        });
    }

private:
    int m_toys;
};

int main() {
    vector<int> v;

    for (int i = 0; i < 3; ++i) {
        v.push_back(i);
    }

    Kitty k(5);
    k.meow(v);
}

C:¥Temp>cl /EHsc /nologo /W4 implicitmemberkitty.cpp > NUL &&
implicitmemberkitty
If you gave me 0 toys, I would have 5 toys total.
If you gave me 1 toys, I would have 6 toys total.
If you gave me 2 toys, I would have 7 toys total.
```

你可以使用 [&]，但它对传递是怎样进行的（默认是按值）没有影响。你不能使用 [&this]。

如果你想使用一个空无的 lambda （不带参数），你可以省略整个 lambda 参数声明：

```
C:¥Temp>type nullarykitty.cpp
#include <algorithm>
#include <iostream>
#include <iterator>
#include <ostream>
```

```
#include <vector>
using namespace std;

int main() {
    vector<int> v;

    int i = 0;

    generate_n(back_inserter(v), 10, [&] { return i++; });

    for_each(v.begin(), v.end(), [](int n) { cout << n << "
"; });
    cout << endl;

    cout << "i: " << i << endl;
}

C:¥Temp>cl /EHsc /nologo /W4 nullarykitty.cpp > NUL &&
nullarykitty
0 1 2 3 4 5 6 7 8 9
i: 10
```

这比 [&]() { return i++; } 少两个字符。省略 lambda 参数声明是否是良好的风格是由你来决定的。

仅博一笑，这意味着下面的代码在 C++0x 中是合法的。

```
C:¥Temp>type nokitty.cpp
int main() {
    [](){}();
    []{}();
}
```

上面构建了两个什么也不做的 lambda （一个有 lambda 参数声明，一个没有）并立即调用它们（通过最后一个空括号组合）。

注意可选的 lambda 参数声明语法上包括：
(lambda-parameter-declaration-listopt) mutableopt exception-specificationopt lambda-return-type-clauseopt

因此，如果你想使用 mutable 或 -> ReturnType，你需要在 lambda 导引符和它们之间插入空括号。

最后，因为 lambda 产生普通的函数对象，你可以在 tr1::function 中存储它们。

```
C:¥Temp>type tr1kitty.cpp
#include <algorithm>
#include <functional>
#include <iostream>
#include <ostream>
#include <vector>
using namespace std;
using namespace std::tr1;

void meow(const vector<int>& v, const function<void (int)>& f)
{
    for_each(v.begin(), v.end(), f);
    cout << endl;
}

int main() {
    vector<int> v;

    for (int i = 0; i < 10; ++i) {
        v.push_back(i);
    }

    meow(v, [](int n) { cout << n << " "; });
    meow(v, [](int n) { cout << n * n << " "; });

    function<void (int)> g = [](int n) { cout << n * n * n <<
" "; };
```

```
    meow(v, g);
}

C:¥Temp>cl /EHsc /nologo /W4 tr1kitty.cpp > NUL && tr1kitty
0 1 2 3 4 5 6 7 8 9
0 1 4 9 16 25 36 49 64 81
0 1 8 27 64 125 216 343 512 729
```

auto

关键词 auto 是从 C++ 98 得来的，它在 C++ 98 中实际上并没起什么作用，但在 C++ 0x 中被重用于自动类型推导（automatic type deduction）。 在一个声明中使用 auto ，意味着“**让它和初始化它的事物具有相同的类型**”。你看：

```
C:¥Temp>type autocat.cpp
#include <iostream>
#include <map>
#include <ostream>
#include <regex>
#include <string>
using namespace std;
using namespace std::tr1;

int main() {
    map<string, string> m;

    const regex r("(\\w+) (\\w+)");

    for (string s; getline(cin, s); ) {
        smatch results;

        if (regex_match(s, results, r)) {
            m[results[1]] = results[2];
        }
    }
```

```
    for (auto i = m.begin(); i != m.end(); ++i) {
        cout << i->second << " are " << i->first << endl;
    }
}

C:¥Temp>cl /EHsc /nologo /W4 autocat.cpp > NUL && autocat
cute kittens
ugly puppies
evil goblins
^Z
kittens are cute
goblins are evil
puppies are ugly
```

map<string, string>::iterator，你近 10 年的恐怖统治终于到头了。（注意：m.begin()返回 iterator，而不是 cosnt_iterator，因为 map 不是 const 属性的。C++ 0x cbegin() 允许你从一个 non-cosnt 容器中获得一个 const_iterator。）

lambdas and auto

先前我提到过在 tr1::functions 中存储 lambda 函数。但是如非必要请别这么做，因为 tr1::function 会增加一些开销。如果你想重用 lambda 函数，或只想给它命名的话，你可以使用 auto。这儿有一个简洁的例子：

```
C:¥Temp>type autolambdakitty.cpp
#include <algorithm>
#include <iostream>
#include <ostream>
#include <vector>
using namespace std;

template <typename T, typename Predicate> void
keep_if(vector<T>& v, Predicate pred) {
    auto notpred = [&](const T& t) {
```

```
        return !pred(t);
    };

      v.erase(remove_if(v.begin(),    v.end(),    notpred),
v.end());
}

template <typename Container> void print(const Container& c)
{
      for_each(c.begin(),    c.end(),    [](const    typename
Container::value_type& e) { cout << e << " "; });
    cout << endl;
}

int main() {
    vector<int> a;

    for (int i = 0; i < 100; ++i) {
        a.push_back(i);
    }

    vector<int> b;

    for (int i = 100; i < 200; ++i) {
        b.push_back(i);
    }

    auto prime = [](const int n) -> bool {
        if (n < 2) {
            return false;
        }

        for (int i = 2; i <= n / i; ++i) {
            if (n % i == 0) {
                return false;
            }
```

```
        }

        return true;
    };

    keep_if(a, prime);
    keep_if(b, prime);

    print(a);
    print(b);
}

C:¥Temp>cl /EHsc /nologo /W4 autolambdakitty.cpp
autolambdakitty.cpp

C:¥Temp>autolambdakitty
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79
83 89 97
101 103 107 109 113 127 131 137 139 149 151 157 163 167 173 179
181 191 193 197 199
```

notpred 是一个否定 lambda 函数，注意我们不能使用 C++ 98 <functional>中的 not1() 函数，因为那要求你的谓词函数继承自 unary_function，而 lambda 函数并不满足这个条件。

static_assert

static_assert 让你能够在触发编译器错误时使用定制的错误信息:

```
C:¥Temp>type staticfluffykitten.cpp
template <int N> struct Kitten {
    static_assert(N < 2, "Kitten<N> requires N < 2.");
};

int main() {
    Kitten<1> peppermint;
```

```
    Kitten<3> jazz;
}

C:¥Temp>cl /EHsc /nologo /W4 staticfluffykitten.cpp
staticfluffykitten.cpp
staticfluffykitten.cpp(2) : error C2338: Kitten<N> requires N
< 2.
        staticfluffykitten.cpp(8) : see reference to class
template instantiation 'Kitten<N>' being compiled
        with
        [
            N=3
        ]
```

如果你有任何疑问，我很乐意在评论中回答它们。

Stephan T. Lavavej
Visual C++ Libraries Developer
Published Tuesday, October 28, 2008 9:31 AM by vcblog

翻译：飘飘白云
译文链接：http://www.cppblog.com/kesalin/archive/2009/05/28/85983.html

(转载时请注明作者和出处。未经许可，请勿用于商业用途)

VC10 中的 C++0x 特性 Part 2 ：右值引用

来源：vcblog　翻译：飘飘白云 kesalin@gmail.com

今天我要讲的是 rvalue references （右值引用），它能实现两件不同的事情：move 语意和完美转发。刚开始会觉得它们难以理解，因为需要区分 lvalues 和 rvalues ，而只有极少数 C++98/03 程序员对此非常熟悉。这篇文章会很长，因为我打算极其详尽地解释 rvalue references 的运作机制。

不用害怕，使用 ravlue references 是很容易的，比听起来要容易得多。要在你的代码中实现 move semantics 或 perfect forwarding 只需遵循简单的模式，后文我会对此作演示的。学习如何使用 rvalue references 是绝对值得的，因为 move semantics 能带来巨大的性能提升，而 perfect forwarding 让高度泛型代码的编写变得非常容易。

C++ 98/03 中的 lvalues 和 rvalues

要理解 C++ 0x 中的 rvalue references，你得先理解 C++ 98/03 中的 lvalues 与 rvalues。

术语 “lvalues” 和 “rvalues” 是很容易被搞混的，因为它们的历史渊源也是混淆。（顺带一提，它们的发音是 ‘L values“ 和 ”R values“， 尽管它们都 写成一个单词）。这两个概念起初来自 C，后来在 C++ 中被加以发挥。为节省时间，我跳过了有关它们的历史，比如为什么它们被称作 “lvalues” 和 “rvalues”，我将直接讲它们在 C++ 98/03 中是如何运作的。（好吧，这不是什么大秘密： “L” 代表 “left”，“R” 代表 “right”。它们的含义一直在演化而名字却没变，现在已经“名”不副“实”了。与其帮你上一整堂历史课，不如随意地把它们当作像“上夸克”和“下夸克 ”之类的名字，也不会有什么损失。）

C++ 03 标准 3.10/1 节上说： “每一个表达式要么是一个 lvalue ，要么就是一个 rvalue 。” 应该谨记 lvalue 跟 rvalue 是针对表达式而言的，而不是对象。

lvalue 是指那些单一表达式结束之后依然存在的持久对象。例如： obj，*ptr，

prt[index]， ++x 都是 lvalue。

rvalue 是指那些表达式结束时（在分号处）就不复存在了的临时对象。例如：1729 ， x + y ， std::string("meow") ， 和 x++ 都是 rvalue。

注意 ++x 和 x++ 的区别。当我们写 int x = 0; 时， x 是一个 lvalue，因为它代表一个持久对象。 表达式 ++x 也是一个 lvalue，它修改了 x 的值，但还是代表原来那个持久对象。然而，表达式 x++ 却是一个 rvalue，它只是拷贝一份持久对象的初值，再修改持久对象的值，最后返回那份拷贝，那份拷贝是临时对象。 ++x 和 x++ 都递增了 x，但 ++x 返回持久对象本身，而 x++ 返回临时拷贝。这就是为什么 ++x 之所以是一个 lvalue，而 x++ 是一个 rvalue。**lvalue 与 rvalue 之分不在于表达式做了什么，而在于表达式代表了什么（持久对象或临时产物）。**

另一个培养判断一个表达式是不是 lvalue 的直觉感的方法就是自问一下“我能不能对表达式取址？”，如果能够，那就是一个 lvalue；如果不能，那就是 一个 rvalue。 例如：&obj ， &*ptr ， &ptr[index] ， 和 &++x 都是合法的（即使其中一些例子很蠢），而 &1729 ， &(x + y) ， &std::string("meow") ， 和 &x++ 是不合法的。为什么这个方法凑效？因为取址操作要求它的“操作数必须是一个 lvalue”（见 C++ 03 5.3.1/2）。为什么要有那样的规定？因为对一个持久对象取址是没问题的，但对一个临时对象取址是极端危险的，因为临时对象很快就会被销毁（译注：就 像你有一个指向某个对象的指针，那个对象被释放了，但你还在使用那个指针，鬼知道这时候指针指向的是什么东西）。

前面的例子不考虑操作符重载的情况，它只是普通的函数调用语义。“一个函数调 用是一个 lvalue 当且仅当它返回一个引用”（见 C++ 03 5.2.2/10）。因此，给定语句 vercor<int> v(10, 1729); ， v[0] 是一个 lvalue，因为操作符 []() 返回 int& (且 &v[0] 是合法可用的)； 而给定语句 string s("foo");和 string t("bar");，s + t 是一个 rvalue，因为操作符 +() 返回 string（而 &(s + t) 也不合法）。

lvalue 和 rvalue 两者都有非常量（modifiable，也就是说 non-const）与常量（const ）之分。举例来说：

```
string one("cute");
const string two("fluffy");
```

```
string three() { return "kittens"; }
const string four() { return "are an essential part of a healthy
diet"; }

one;     // modifiable lvalue
two;     // const lvalue
three(); // modifiable rvalue
four();  // const rvalue
```

Type& 可绑定到非常量 lvalue （可以用这个引用来读取和修改原来的值），但不能绑定到 const lvalue，因为那将违背 const 正确性；也不能把它绑定到非常量 rvalue，这样做极端危险，你用这个引用来修改临时对象，但临时对象早就不存在了，这将导致难以捕捉而令人讨厌的 bug，因此 C++ 明智地禁止这这么做。（我要补充一句：VC 有一个邪恶的扩展允许这么蛮干，但如果你编译的时候加上参数 /W4 ，编译器通常会提示警告"邪恶的扩展被激活了"）。也不能把它绑定到 const ravlue，因为那会是双倍的糟糕。（细心的读者应该注意到了我在这里并没有谈及模板参数推导）。

const Type& 可以绑定到： 非常量 lvalues， const lvalues，非常量 rvalues 以及 const values。（然后你就可以用这个引用来观察它们）

引用是具名的，因此一个绑定到 rvalue 的引用，它本身是一个 lvalue（没错！是 L)。(因为只有 const 引用可以绑定到 rvalue,所以它是一个 const lvalue)。这让人费解，（不弄清楚的话）到后面会更难以理解，因此我将进一步解释。给定函数 void observe(const string& str)， 在 observe()'s 的实现中， str 是一个 const lvalue，在 observe() 返回之前可以对它取址并使用那个地址。这一点即使我们通过传一个 rvalue 参数来调用 observe()也是成立的 ， 就像上面的 three() 和 four()。也可以调用 observe("purr")，它构建一个临时 string 并将 str 绑定到那个临时 string。three() 和 foure() 的返回对象是不具名的，因此他们是 rvalue，但是在 observe()中，str 是具名的，所以它是一个 lvalue。正如前面我说的" lvalue 跟 rvalue 是针对表达式而言的，而不是对象"。当然，因为 str 可以被绑定到一个很快会被销毁的临时对象，所以在 observe() 返回之后我们就不应该在任何地方保存这个临时对象的地址。

你有没有对一个绑定到 rvalue 的 const 引用取址过么？当然，你有过！每当你写一个带自赋值检查的拷贝赋值操作符： Foo& operator=(const Foo& other),

if(this != &other) {copy struff;}； 或从一个临时变量来拷贝赋值，像： Foo make_foo(); Foo f; f = make_foo(); 的时候，你就做了这样的事情。

这个时候，你可能会问“那么非常量 rvalues 跟 const rvalues 有什么不同呢？我不能将 Type& 绑定到非常量 rvalue 上，也不能通过赋值等操作来修改 rvalue，那我真的可以修改它们？” 问的很好！在 C++ 98/03 中，这两者存在一些细微的差异： non-constrvalues 可以调用 non-const 成员函数。 C++ 不希望你意外地修改临时对象，但直接在 non-const rvalues 上调用 non-const 成员函数，这样做是很明显的，所以这是被允许的。在 C++0x 中，答案有了显著的变化，它能用来实现 move 语意。

恭喜！你已经具备了我所谓的“lvalue/rvalue 观”，这样你就能够一眼就判断出一个表达式到底是 lvalue 还是 rvalue。再加上你原来对 const 的认识，你就能完全理解为什么给定语句 void mutate(string& ref) 以及前面的变量定义，mutate(one) 是合法的，而 mutate(two)， mutate(three())， mutate(four())， mutate("purr") 都是不合法的。如果你是 C++ 98/03 程序员，你已经可以分辨出这些调用中的哪些是合法的，哪些是不合法的；是你的“本能直觉”，而不是你的编译器，告诉你 mutate(three()) 是假冒的。你对 lvalue/rvalue 的新认识让你明确地理解为什么 three() 是一个 rvalue，也知道为什么非常量引 用不能绑定到右值。知道这些有用么？对语言律师而言，有用，但对普通程序员来说并不见得。毕竟，你如果不理解关于 lvalues 和 rvalues 一切就要领悟这个还隔得远呢。但是重点来了：与 C++98/03 相比， C++0x 中的 lvalue 和 rvalue 有着更广泛更强劲的含义（尤其是判断表达式是否是 modifiable / const 的 lvalue/rvalue，并据此做些处理）。要有效地使用 C++ 0x，你也需具备对 lvalue/rvalue 的理解。现在万事具备，我们能继续前行了。

拷贝的问题

C++ 98/03 将不可思议的高度抽象和不可思议的高效执行结合到了一起，但有个问题：它过度滥用拷贝。对行为像 int 那样有着值语意的对象而言，源对象的拷贝是独立存在的，并不会影响源对象。值语意很好，除了在会导致冗余拷贝之外，像 拷贝 strings，vectors 等重型对象那样的情况。（“重型”意味着“昂贵的拷贝开销”；有着 100 万个元素的 vector 是重型的）。返回值优化（RVO） 和命名返回值优化（NRVO） 在特定情况下可以优化掉拷贝构造操作，这有助于减缓问题的严重性，但是它们不能够消除所有冗余的拷贝。

最 最没有必要的拷贝是拷贝那些立马会被销毁的对象。你有过复印一份文件，并马上把原件扔掉的经历么（假定原件和复件是相同的）？那简直是浪费，你应该持有原 件而不必费劲去复印。下面是被我称作“杀手级的示例”，来自标准委员会的例子(见提案 N1377)，假设你有一大堆 string 像这样的：

```
string s0("my mother told me that");
string s1("cute");
string s2("fluffy");
string s3("kittens");
string s4("are an essential part of a healthy diet");
```

然后你想像这样把它们串接起来：

```
string dest = s0 + " " + s1 + " " + s2 + " " + s3 + " " + s4;
```

这样做的效率如何？（我们不用为这个特殊的例子而担忧，它的执行只要几微秒；我们担忧它的一般化情况，在语言层面上的情况）。

每 次调用操作符 +() 就会返回一个临时 string。上面调用了 8 次操作符 +()，因而产生了 8 个临时 string。 每一个临时 string，在构造过程中分配动态内存，再拷贝所有已连接的字符，最后在析构过程中释放分配的动态内存。（你听说过短串优化技术么，为了避免动态内存的 分配与释放，VC 是这么干的，在这个被我精心挑选的有着合适长度的 s0 面前短串优化技术也无能为力，即使执行了这样的优化，也无法避免拷贝操作。如果你还听说过写时拷贝优化（Copy - On - Write），忘了它吧，在这里也不适用，并且在多线程环境下这种优化会恶化问题，因此标准库实现根本就不再做这个优化了）。

事实上，因为每一个串接操作都会拷贝所有已经串接好的字符，所以那个复杂度是字符串长度的平方了。哎呀！这太浪费了！这点确实让 C++ 尴尬。事情怎么会搞成这样呢？有没有改善的办法？

问题是这样的，operator+() 接收两个参数，一个是 const string&，另一个是 const string& 或 const char * (还有其他重载版本，但在这里我们没有用到)，但 operator+() 无法分辨出你塞给它的是 lvalue 还是 rvalue 参数，所以它只好总是创建一个临时 string，并返回这个临时 string。 为什么这跟

vavlue/rvalue 有关系？

当我们要计算 s0 + ″ ″ 的值时，很明显这里有必要创建一个新的临时 string。s0 是一个 lvalue，它已经命名了一个持久对象，因此我们不能修改它。（有人注意到了！）。如果要计算（s0 + “ ”）+ s1 的值，我们可以简单地将 s1 的内容追加到第一个临时 string 上，而不用创建第二个临时 string 再把第一个丢弃掉。这就是 move 语意背后的核心观念： 因为 s0 + ″ ″ 是一个 rvalue，只有那个在整个程序中唯一能够觉察到临时对象存在的表达式可以引用临时对象。如果我们能检测到表达式是一个非常量 rvalue，我们就可以任意修改临时对象，而不会有人发现。 操作符 +() 本不应该修改它的参数，但如果其参数是非常量 rvalue，谁在乎？照这种方法，每次调用操作符 +() 都把字符追加到唯一的临时对象上，这样就彻底省掉了不必要的动态内存管理和冗余的拷贝操作，呈现出线性复杂度。耶！

从技术上讲，C++ 0x 中，每次调用操作符 +() 还是会返回一个单独的临时 string。 然而，第二个临时 string （产生自（s0 + “ ”）+ s1 ）可以通过“窃取”第一个临时 string （产生自 s0 + ″ ″ ）的内存而被构造出来，然后再把 s1 的内容追加到那块内存后面（这将会引发一个普通的重分配操作）。“窃取”是通过指针的操作实现的：第二个临时 string 会先拷贝第一个临时 string 的内部指针，然后再清空这个指针。第一个临时 string 最后被销毁（在分号那地方）时，它的指针已经置为 null 了，因此它的析构函数什么也不会做（译注：也就是说不会释放它的内存，这部分内存现在是第二个临时 string 在使用了）。

通常，如果能够检测到非常量 rvalue，你就能够做些“资源窃取”的优化。如果非常量 rvalue 所引用的那些对象持有任何资源（如内存），你就能窃取它们的资源而不用拷贝它们，反正它们很快就会被销毁掉。通过窃取非常量 rvalue 持有的资源来构建或赋值的手法通常被称作 “moving”，可移动对象拥有 “move 语意”。

在大多数情况下这相当有用，比如 vector 的重新分配。当一个 vector 需要更多空间（如 push_back() 时）和进行重分配操作时，它需要从旧的内存块中拷贝元素到新的内存块中去。这些拷贝构造调用的开销很大。(对 vector<string> 来说，需要拷贝每一个 string 元素，这涉及动态内存分配)。但是等一等！旧内存块中的那些元素很快会被销毁掉的呀，所以我们可以挪动这些元素，而不用拷贝它们。在这种情形下，旧内存块 中的元素依然存在于内存中，用来访问它们的表达式，如 old_ptr[index]，还是 lvalue。在重分配过程中，我们想用非常量

rvalue 表达式来引用旧内存块中的元素。假定它们是非常量 rvalue，那我们就能够移动它们，从而省去拷贝构造开销。（说”我想假定这个 lvalue 是一个非常量 rvalue “等同于说”我知道这是一个 lvalue，它指向一个持久对象，但我不关心随后会对这个 lvalue 进行怎样的操作，或销毁它，或给它赋值，或进行任意操作。因此如果你能从它那里窃取资源的话，尽管行动吧”）

C++0x 的 rvalue 引用概念给与我们检测非常量 rvalue 并从中窃取资源的能力，这让我能够实现 move 语意。rvalue 引用也让我们能够通过把 lvalue 伪装成非常量 rvalue 而随意触发 move 语意。现在，我们来看看 rvalue 引用是如何工作的！

ravlue 引用：初始化

C++0x 引进了一种新的引用，ravlue 引用，其语法是 Type&& 和 const Type&& 。目前 C++0x 草案 N2798 8.3.2/2 上说：“用 & 声明的引用类型被称作 lvalue 引用，而用 && 声明的引用类型被称作 rvalue 引用。lvalue 引用与 rvalue 引用是截然不同的类型。除非特别注明，两者在语意上是相当的并且一般都被称作引用。”这意味着对 C++98/03 中引用(即现在的 lvalue 引用)的直觉印象可以延伸用于 rvalue 引用；你只需要学习这两者的不同之处。

（说明：我选择把 Type& 读作 “Type ref”，Type&& 读作 ″Type ref ref″。它们的全称分别是 “lvalue reference to Type” 和 ″rvalue reference to Type″，就像 “cosnt pointer to int” 被写成 “int＊const”，而被读作 “int star const”一样。）

两者有什么区别？与 lvalue 引用相比， rvalue 引用在初始化与重载决议时表现出不同的行为。两者的区别在于它们会优先绑定到什么东西上（初始化时）和什么东西会优先绑定到它们身上(重载决议时)。首先让我们来看看初始化：

- 我们已经明白为何非常量 lvalue 引用(Type&) 只能绑定到非常量 lvalue 上，而其他的一概不能（如 const lvalues，非常量 rvalues，const rvalues）
- 我们已经明白为何 const lvalue 引用(const Type&) 能绑定到任何东西上。
- 非常量 rvalue (Type&&) 能够绑定到非常量 lvalue 以及非常量 rvalue 上，而不能绑定到 const lvalues 和 const rvalues （这会违背 const 正确性）
- const rvalue 引用(const Type&&) 能够绑定到任何东西上。

这些规则听起来可能有些神秘，但是他们来源于两条简单的规则：

- 遵守 const 正确性，所以你不能把非常量引用绑定到常量上。
- 避免意外修改临时对象，所以你不能把非常量 lvalue 引用绑定到非常量 rvalue 上来。

如果你更喜欢阅读编译器错误信息，而不是阅读文字描述，下面是一个示例：

```
C:¥Temp>type initialization.cpp
#include <string>
using namespace std;

string modifiable_rvalue() {
    return "cute";
}

const string const_rvalue() {
    return "fluffy";
}

int main() {
    string modifiable_lvalue("kittens");
    const string const_lvalue("hungry hungry zombies");

    string& a = modifiable_lvalue;          // Line 16
    string& b = const_lvalue;               // Line 17 - ERROR
    string& c = modifiable_rvalue();        // Line 18 - ERROR
    string& d = const_rvalue();             // Line 19 - ERROR

    const string& e = modifiable_lvalue;    // Line 21
    const string& f = const_lvalue;         // Line 22
    const string& g = modifiable_rvalue();  // Line 23
    const string& h = const_rvalue();       // Line 24

    string&& i = modifiable_lvalue;         // Line 26
    string&& j = const_lvalue;              // Line 27 - ERROR
```

```
    string&& k = modifiable_rvalue();       // Line 28
    string&& l = const_rvalue();            // Line 29 - ERROR

    const string&& m = modifiable_lvalue;   // Line 31
    const string&& n = const_lvalue;        // Line 32
    const string&& o = modifiable_rvalue(); // Line 33
    const string&& p = const_rvalue();      // Line 34
}

C:¥Temp>cl /EHsc /nologo /W4 /WX initialization.cpp
initialization.cpp
initialization.cpp(17) : error C2440: 'initializing' : cannot
convert from 'const std::string' to 'std::string &'
        Conversion loses qualifiers
initialization.cpp(18) : warning C4239: nonstandard extension
used : 'initializing' : conversion from 'std::string' to
'std::string &'
        A non-const reference may only be bound to an lvalue
initialization.cpp(19) : error C2440: 'initializing' : cannot
convert from 'const std::string' to 'std::string &'
        Conversion loses qualifiers
initialization.cpp(27) : error C2440: 'initializing' : cannot
convert from 'const std::string' to 'std::string &&'
        Conversion loses qualifiers
initialization.cpp(29) : error C2440: 'initializing' : cannot
convert from 'const std::string' to 'std::string &&'
        Conversion loses qualifiers
```

非常量 rvalue 引用绑定到非常量 rvalue 是没问题的；要领就是它们可以被用来修改临时对象。

虽然 lvalue 引用和 rvalue 引用在初始化时有着相似的行为（只有第 18 和 28 行不同），但在重载决议的时候它们的区别就很显著了。

rvalue 引用：重载决议

函数可根据非常量和常量 lvalue 引用参数的不同而重载，这一点你应该很熟悉了。在 C++0x 中，函数也可根据非常量和常量 rvalue 引用参数的不同而重载。如果给出这四种形式的重载一元函数，你不应为表达式能优先绑定到与之相对应的引用上而决议出相应的重载函数这一点感到惊奇：

```
C:¥Temp>type four_overloads.cpp
#include <iostream>
#include <ostream>
#include <string>
using namespace std;

void meow(string& s) {
    cout << "meow(string&): " << s << endl;
}

void meow(const string& s) {
    cout << "meow(const string&): " << s << endl;
}

void meow(string&& s) {
    cout << "meow(string&&): " << s << endl;
}

void meow(const string&& s) {
    cout << "meow(const string&&): " << s << endl;
}

string strange() {
    return "strange()";
}

const string charm() {
    return "charm()";
}
```

```
int main() {
    string up("up");
    const string down("down");

    meow(up);
    meow(down);
    meow(strange());
    meow(charm());
}

C:¥Temp>cl /EHsc /nologo /W4 four_overloads.cpp
four_overloads.cpp

C:¥Temp>four_overloads
meow(string&): up
meow(const string&): down
meow(string&&): strange()
meow(const string&&): charm()
```

在实践中，全部重载 Type& ， const Type& ， Type&& ， const Type&& 并不是很有用。只重载 const Type& 和 Type&& 更有意思些：

```
C:¥Temp>type two_overloads.cpp
#include <iostream>
#include <ostream>
#include <string>
using namespace std;

void purr(const string& s) {
    cout << "purr(const string&): " << s << endl;
}

void purr(string&& s) {
    cout << "purr(string&&): " << s << endl;
}
```

```
string strange() {
    return "strange()";
}

const string charm() {
    return "charm()";
}

int main() {
    string up("up");
    const string down("down");

    purr(up);
    purr(down);
    purr(strange());
    purr(charm());
}

C:¥Temp>cl /EHsc /nologo /W4 two_overloads.cpp
two_overloads.cpp

C:¥Temp>two_overloads
purr(const string&): up
purr(const string&): down
purr(string&&): strange()
purr(const string&): charm()
```

上面的重载决议是怎么作出的呢？下面是规则：

（1） 初始化规则拥有否决权。
（2） lvalue 最优先绑定到 lvalue 引用，rvalue 最优先绑定到 rvalue 引用。
（3） 非常量表达式倾向于绑定到非常量引用上。

（我说的“否决权”是指：进行重载决议时初始化规则否决那些不可行(译注：不满足 const 正确性)的候选函数，这些函数阻止将表达式绑定到引用上） 让我们一条一条来看看这些规则是怎么运作的。

• 对 purr(up) 而言，决议（1）初始化规则既不否决 purr(const string&) 也不否决 purr(string&&)。 up 是 lvalue，因此满足决议（2）中的 lvalue 最优先绑定到 lvalue 引用，即 purr(const string&)。up 还是非常量，因此满足决议（3）非常量表达式倾向于绑定到非常量引用上，即 purr(string&&)。两者放一块决议时，决议（2） 胜出，选择 purr(const string&)。

• 对 purr(down) 而言， 决议（1）初始化规则基于 const 正确性否决掉 purr(string&&)，因此 purr(const string&) 胜出。

• 对 purr(strange()) 而言，决议(1)初始化规则既不否决 purr(const string&) 也不否决 purr(string&&)。strange() 是 rvalue， 因此满足决议（2） rvalue 最优先绑定到 rvalue 引用，即 purr(string&&)。strange() 还是非常量，因此满足决议（3）非常量表达式倾向于绑定到非常量引用上，即 purr(string&&)上。purr(string&&) 在这里两票胜出。

• 对 purr(charm()) 而言，决议（1）初始化规则基于 const 正确性否决掉 purr(string&&)，因此 purr(const string&) 胜出。

值得注意的是当你只重载了 const Type& 和 Type&& ，非常量 rvalue 绑定到 Type&&，而其它的都绑定到 const Type&。因此，这一组重载用来实现 move 语义。

重要说明：返回值的函数应当返回 Type（如 strange() ）而不是返回 const Type（如 charm()）。后者不会带来什么好处（阻止非常量成员函数调用），还会阻止 move 语意优化。

move 语义：模式

下 面是一个简单的类 remote_integer，内部存储一个指向动态分配的 int 指针（“远程拥有权”）。你应该对这个类的默认构造函数，一元构造函数，拷贝构造函数，拷贝赋值函数和析构函数都很熟悉了。我给它增加了 move 构造函数和 move 赋值函数，它们被#ifdef MOVABLE 围起来了，这样我就可以演示在有和没有这两个函数的情况下会有什么差别，在真实的代码中是不会这么做的。

```
C:¥Temp>type remote.cpp
#include <stddef.h>
#include <iostream>
#include <ostream>
using namespace std;

class remote_integer {
public:
    remote_integer() {
        cout << "Default constructor." << endl;

        m_p = NULL;
    }

    explicit remote_integer(const int n) {
        cout << "Unary constructor." << endl;

        m_p = new int(n);
    }

    remote_integer(const remote_integer& other) {
        cout << "Copy constructor." << endl;

        if (other.m_p) {
            m_p = new int(*other.m_p);
        } else {
            m_p = NULL;
        }
    }

#ifdef MOVABLE
    remote_integer(remote_integer&& other) {
        cout << "MOVE CONSTRUCTOR." << endl;

        m_p = other.m_p;
        other.m_p = NULL;
```

```
    }
#endif // #ifdef MOVABLE

    remote_integer& operator=(const remote_integer& other) {
        cout << "Copy assignment operator." << endl;

        if (this != &other) {
            delete m_p;

            if (other.m_p) {
                m_p = new int(*other.m_p);
            } else {
                m_p = NULL;
            }
        }

        return *this;
    }

#ifdef MOVABLE
    remote_integer& operator=(remote_integer&& other) {
        cout << "MOVE ASSIGNMENT OPERATOR." << endl;

        if (this != &other) {
            delete m_p;

            m_p = other.m_p;
            other.m_p = NULL;
        }

        return *this;
    }
#endif // #ifdef MOVABLE

    ~remote_integer() {
        cout << "Destructor." << endl;
```

```
        delete m_p;
    }

    int get() const {
        return m_p ? *m_p : 0;
    }

private:
    int * m_p;
};

remote_integer square(const remote_integer& r) {
    const int i = r.get();

    return remote_integer(i * i);
}

int main() {
    remote_integer a(8);

    cout << a.get() << endl;

    remote_integer b(10);

    cout << b.get() << endl;

    b = square(a);

    cout << b.get() << endl;
}

C:¥Temp>cl /EHsc /nologo /W4 remote.cpp
remote.cpp

C:¥Temp>remote
```

```
Unary constructor.
8
Unary constructor.
10
Unary constructor.
Copy assignment operator.
Destructor.
64
Destructor.
Destructor.

C:¥Temp>cl /EHsc /nologo /W4 /DMOVABLE remote.cpp
remote.cpp

C:¥Temp>remote
Unary constructor.
8
Unary constructor.
10
Unary constructor.
MOVE ASSIGNMENT OPERATOR.
Destructor.
64
Destructor.
Destructor.
```

这里有几点值得注意：

• 我们重载了拷贝构造函数和 move 构造函数，还重载了拷贝赋值函数和 move 赋值函数。在前面我们已经看到了当函数通过 const Type& 和 Type&& 进行重载时，会有怎样的结果。当 move 语意可用时，b = square(a) 会自动选择调用 move 赋值函数。

• move 构造函数和 move 赋值函数只是简单的从 other 那里“窃取”内存，而不用动态分配内存。当“窃取”内存时，我们只是拷贝 other 的指针成员，然后再把它置为 null。于是当 other 被销毁时，析构函数什么也不做。

• 拷贝赋值函数和 move 赋值函数都需要进行自我赋值检查，为何拷贝赋值函数需要进行自我赋值检查是广为人知的。这是因为像 int 这样的内建数据(POD)类型能够正确地自我赋值（如：x = x ），因此，用户自定义的数据类型理应也可以正确地自我赋值。自我赋值实际上在手写代码里面是不存在的，但是在类似 std::sort() 之类的算法中，却很常见。在 C++0x 中，像 std::sort() 之类的算法能够通过挪动而非拷贝元素来实现。在这里（move 赋值函数）也需要进行自我赋值检查。

这时，你可能会想它们（ move 拷贝构造函数和 move 赋值函数）与编译器自动生成（标准中用词“隐式声明”）的默认拷贝构造函数和默认赋值函数有什么相互影响呢。

• 永远不会自动生成 move 构造函数和 move 赋值函数。
• 用户声明的构造函数，拷贝构造函数和 move 构造函数会抑制住默认构造函数的自动生成。
• 用户声明的拷贝构造函数会抑制住默认拷贝构造函数的自动生成，但是用户声明的 move 构造函数做不到。
• 用户声明的拷贝赋值函数会抑制住默认拷贝赋值函数的自动生成，但是用户声明的 move 赋值函数做不到。

基本上，除了声明 move 构造函数会抑制默认构造函数的自动生成以外，自动生成规则不影响 move 语意。

move 语意：从 lvalue 移动

现在，如果你喜欢用拷贝赋值函数来实现你的拷贝构造函数该怎样做呢，那你也可能试图用 move 拷贝赋值函数来实现 move 构造函数。这样作是可以的，但是你得小心。下面就是一个错误的实现：

```
C:¥Temp>type unified_wrong.cpp
#include <stddef.h>
#include <iostream>
#include <ostream>
using namespace std;
```

```
class remote_integer {
public:
    remote_integer() {
        cout << "Default constructor." << endl;

        m_p = NULL;
    }

    explicit remote_integer(const int n) {
        cout << "Unary constructor." << endl;

        m_p = new int(n);
    }

    remote_integer(const remote_integer& other) {
        cout << "Copy constructor." << endl;

        m_p = NULL;
        *this = other;
    }

#ifdef MOVABLE
    remote_integer(remote_integer&& other) {
        cout << "MOVE CONSTRUCTOR." << endl;

        m_p = NULL;
        *this = other; // WRONG
    }
#endif // #ifdef MOVABLE

    remote_integer& operator=(const remote_integer& other) {
        cout << "Copy assignment operator." << endl;

        if (this != &other) {
            delete m_p;
```

```
            if (other.m_p) {
                m_p = new int(*other.m_p);
            } else {
                m_p = NULL;
            }
        }

        return *this;
    }

#ifdef MOVABLE
    remote_integer& operator=(remote_integer&& other) {
        cout << "MOVE ASSIGNMENT OPERATOR." << endl;

        if (this != &other) {
            delete m_p;

            m_p = other.m_p;
            other.m_p = NULL;
        }

        return *this;
    }
#endif // #ifdef MOVABLE

    ~remote_integer() {
        cout << "Destructor." << endl;

        delete m_p;
    }

    int get() const {
        return m_p ? *m_p : 0;
    }
```

```
private:
    int * m_p;
};

remote_integer frumple(const int n) {
    if (n == 1729) {
        return remote_integer(1729);
    }

    remote_integer ret(n * n);

    return ret;
}

int main() {
    remote_integer x = frumple(5);

    cout << x.get() << endl;

    remote_integer y = frumple(1729);

    cout << y.get() << endl;
}

C:¥Temp>cl /EHsc /nologo /W4 /O2 unified_wrong.cpp
unified_wrong.cpp

C:¥Temp>unified_wrong
Unary constructor.
Copy constructor.
Copy assignment operator.
Destructor.
25
Unary constructor.
1729
Destructor.
```

```
Destructor.

C:¥Temp>cl /EHsc /nologo /W4 /O2 /DMOVABLE unified_wrong.cpp
unified_wrong.cpp

C:¥Temp>unified_wrong
Unary constructor.
MOVE CONSTRUCTOR.
Copy assignment operator.
Destructor.
25
Unary constructor.
1729
Destructor.
Destructor.
```

(编译器在这里进行了返回值优化(RVO)，但不是具名返回值优化(NRVO)。就像我之前提到的，有些拷贝构造函数被 RVO 或 NRVO 优化掉了，但编译器并不总是能够做这样的优化，这时剩余的就由 move 构造函数来优化。)

move 构造函数中标记为 WRONG 的那一行，调用了拷贝赋值函数，编译能通过也能运行，但这违背了 move 构造函数的本意。（译注：因为那个拷贝赋值函数只是进行普通的拷贝赋值，而不是 move 赋值！）

这 是怎么回事呢？记住：在 C++98/03 中，具名 lvalue 引用是左值(给定语句 int& r = *p; r 是 lvalue)，不具名 lvalue 引用还是左值(给定语句 vector<int> v(10, 1729)， v[0] 返回 int&， 你可以对这个不具名 lvalue 引用取址)。但是 rvalue 引用就不一样了：

具名 lvalue 引用是 lvalue。
不具名 rvalue 引用是 rvalue。

一个具名 rvalue 引用是一个 lvalue 是因为可以对它施加多重操作，重复使用。相反，如果它是一个 ravlue 的话，那么对它施加的第一个操作能够“窃取”它，而后续操作就没机会了。这里的“窃取”是说不会被察觉到，所以这是行不通的。另一方面，不具名 rvalue 引用不能被重复使用，所以它仍保持右值(rvalueness)

语意。

如果你真的打算用 move 赋值函数来实现 move 构造函数，你需要从 lvalue move，就像是从 rvalue move 一样。C++0x 〈utility〉中的 std::move() 具备这样的能力，VC10 将会有这个（实际上，开发版中已经有了），但 VC10 TCP 版还没有，所以我会教你从头做起：

```
C:¥Temp>type unified_right.cpp
#include <stddef.h>
#include <iostream>
#include <ostream>
using namespace std;

template <typename T> struct RemoveReference {
     typedef T type;
};

template <typename T> struct RemoveReference<T&> {
     typedef T type;
};

template <typename T> struct RemoveReference<T&&> {
     typedef T type;
};

template <typename T> typename RemoveReference<T>::type&&
Move(T&& t) {
    return t;
}

class remote_integer {
public:
    remote_integer() {
        cout << "Default constructor." << endl;

        m_p = NULL;
```

```
    }

    explicit remote_integer(const int n) {
        cout << "Unary constructor." << endl;

        m_p = new int(n);
    }

    remote_integer(const remote_integer& other) {
        cout << "Copy constructor." << endl;

        m_p = NULL;
        *this = other;
    }

#ifdef MOVABLE
    remote_integer(remote_integer&& other) {
        cout << "MOVE CONSTRUCTOR." << endl;

        m_p = NULL;
        *this = Move(other); // RIGHT
    }
#endif // #ifdef MOVABLE

    remote_integer& operator=(const remote_integer& other) {
        cout << "Copy assignment operator." << endl;

        if (this != &other) {
            delete m_p;

            if (other.m_p) {
                m_p = new int(*other.m_p);
            } else {
                m_p = NULL;
            }
        }
```

```
        return *this;
    }

#ifdef MOVABLE
    remote_integer& operator=(remote_integer&& other) {
        cout << "MOVE ASSIGNMENT OPERATOR." << endl;

        if (this != &other) {
            delete m_p;

            m_p = other.m_p;
            other.m_p = NULL;
        }

        return *this;
    }
#endif // #ifdef MOVABLE

    ~remote_integer() {
        cout << "Destructor." << endl;

        delete m_p;
    }

    int get() const {
        return m_p ? *m_p : 0;
    }

private:
    int * m_p;
};

remote_integer frumple(const int n) {
    if (n == 1729) {
        return remote_integer(1729);
```

```
    }

    remote_integer ret(n * n);

    return ret;
}

int main() {
    remote_integer x = frumple(5);

    cout << x.get() << endl;

    remote_integer y = frumple(1729);

    cout << y.get() << endl;
}

C:¥Temp>cl /EHsc /nologo /W4 /O2 /DMOVABLE unified_right.cpp
unified_right.cpp

C:¥Temp>unified_right
Unary constructor.
MOVE CONSTRUCTOR.
MOVE ASSIGNMENT OPERATOR.
Destructor.
25
Unary constructor.
1729
Destructor.
Destructor.
```

（我将交替使用 std::move() 和我自己的 Move()，因为它们的实现是等价的）std::move() 是怎样工作的呢？目前，我只能跟你说这是“魔法”。（后面会有完整的解释，并不复杂，但它与模板参数推导和引用折叠（reference collapsing，译注：引用的引用）有 关，后面讲完美转发的时候我们还会遇到这两个东西）。我可以用一个具体的例子来略过“魔法”：给定一个 string 类型的左值，像前

面重载决议例子中的 up ，std::move(up) 调用 string&& std::move(string&)，这个函数返回一个不具名的 rvalue 引用，它是一个 rvalue。给定一个 string 类型的 rvalue，像前面重载决议例子中的 strange()， std::move(strange()) 调用 string&& std::move(string&&)，同样这个函数还是返回一个不具名的 rvalue，还是 rvalue。

std::move() 除了让你能用 move 复制函数来实现 move 构造函数之外，还能在其他地方发挥作用。无论何时，只要你有一个左值，而它的值也不再重要了（例如，它将被销毁或被赋值），你就可以使用 std::move(你的左值表达式) 来使用 move 语意。

move 语意： 可移动成员（movable member,指带 move 语意的成员）

C++0x 的标准类型（像 vector, string, regex） 都有 move 构造函数和 move 赋值函数。而且我们也已经看到了如何在我们自己的类中通过手动管理资源来实现 move 语意（像前面的 remote_integer 类）。如果类中包含可移动数据成员（像 vector, string, regex, remote_integer ）时该怎么办呢？编译器不会自动帮我们自动产生 move 构造函数和 move 赋值函数，所以我们必须手动编写它们。很幸运，有了 std::move() 编写它们是很容易的。

```
C:¥Temp>type point.cpp
#include <stddef.h>
#include <iostream>
#include <ostream>
using namespace std;

template <typename T> struct RemoveReference {
     typedef T type;
};

template <typename T> struct RemoveReference<T&> {
     typedef T type;
};

template <typename T> struct RemoveReference<T&&> {
```

```
     typedef T type;
};

template <typename T> typename RemoveReference<T>::type&&
Move(T&& t) {
    return t;
}

class remote_integer {
public:
    remote_integer() {
        cout << "Default constructor." << endl;

        m_p = NULL;
    }

    explicit remote_integer(const int n) {
        cout << "Unary constructor." << endl;

        m_p = new int(n);
    }

    remote_integer(const remote_integer& other) {
        cout << "Copy constructor." << endl;

        if (other.m_p) {
            m_p = new int(*other.m_p);
        } else {
            m_p = NULL;
        }
    }

    remote_integer(remote_integer&& other) {
        cout << "MOVE CONSTRUCTOR." << endl;

        m_p = other.m_p;
```

```
    other.m_p = NULL;
}

remote_integer& operator=(const remote_integer& other) {
    cout << "Copy assignment operator." << endl;

    if (this != &other) {
        delete m_p;

        if (other.m_p) {
            m_p = new int(*other.m_p);
        } else {
            m_p = NULL;
        }
    }

    return *this;
}

remote_integer& operator=(remote_integer&& other) {
    cout << "MOVE ASSIGNMENT OPERATOR." << endl;

    if (this != &other) {
        delete m_p;

        m_p = other.m_p;
        other.m_p = NULL;
    }

    return *this;
}

~remote_integer() {
    cout << "Destructor." << endl;

    delete m_p;
```

```
    }

    int get() const {
        return m_p ? *m_p : 0;
    }

private:
    int * m_p;
};

class remote_point {
public:
    remote_point(const int x_arg, const int y_arg)
        : m_x(x_arg), m_y(y_arg) { }

    remote_point(remote_point&& other)
        : m_x(Move(other.m_x)),
          m_y(Move(other.m_y)) { }

    remote_point& operator=(remote_point&& other) {
        m_x = Move(other.m_x);
        m_y = Move(other.m_y);
        return *this;
    }

    int x() const { return m_x.get(); }
    int y() const { return m_y.get(); }

private:
    remote_integer m_x;
    remote_integer m_y;
};

remote_point five_by_five() {
    return remote_point(5, 5);
}
```

```
remote_point taxicab(const int n) {
    if (n == 0) {
        return remote_point(1, 1728);
    }

    remote_point ret(729, 1000);

    return ret;
}

int main() {
    remote_point p = taxicab(43112609);

    cout << "(" << p.x() << ", " << p.y() << ")" << endl;

    p = five_by_five();

    cout << "(" << p.x() << ", " << p.y() << ")" << endl;
}

C:¥Temp>cl /EHsc /nologo /W4 /O2 point.cpp
point.cpp

C:¥Temp>point
Unary constructor.
Unary constructor.
MOVE CONSTRUCTOR.
MOVE CONSTRUCTOR.
Destructor.
Destructor.
(729, 1000)
Unary constructor.
Unary constructor.
MOVE ASSIGNMENT OPERATOR.
MOVE ASSIGNMENT OPERATOR.
```

```
Destructor.
Destructor.
(5, 5)
Destructor.
Destructor.
```

现在你看到啦，按成员移动（memberwise move）是很容易做到的。注意，remote_point 的 move 赋值函数没有进行自我赋值检查，是因为 remote_integer 已经检查过了。也要注意到 remote_point 隐式声明的拷贝构造函数，拷贝赋值函数和析构函数都正常运作。

到现在，你应该对 move 语意已经非常熟悉了。（希望不是抓狂啊！）为了测试你新获得的这个不可思议技能，为前面的例子写一个 +() 操作符函数当作练习吧。

最后的提醒：只要你的类支持 move 语意，你就应该实现 move 构造函数和 move 赋值函数。因为不仅仅是你平常使用这些类时可从 move 语意中获利， STL 容器和算法也能从中获利，通过廉价的 move 省下昂贵的拷贝开销。

转发问题

在 程序员不用写高度泛化的代码的时候，C++98/03 的 lvalue， rvalue， 引用，还有模板看起来是很完美的。假设你要写一个完全泛化的函数 outer（），这个函数的目的是将任意数目个任意类型的参数传递（也就是“转发”）给函数 inner（）。已有很多不错的解决方案，比如 factory 函数 make_shared<T>(args) 是把 args 传给 T 的构造函数，然后返回 shared_ptr<T>。（这样就把 T 对象和用于对它进行引用计数的代码存储到同一块动态内存中，性能上与侵入式引用计数一样好）； 而像 function<Ret(args)> 这样的包装类是把参数传给其内部存储的函数对象（functor），等等。在这篇文章里，我们只对 outer() 是如何把参数传递给 inner() 这部分感兴趣。至于 outer() 的返回类型是怎么决定的是另外的问题（有时候很简单，如 make_shared<T>(args) 总是返回 shared_prt<T>，），但要在完全搞定这个问题的一般化情况，你就要用到 C++0x 的 decltype 特性了）。

如果不带参数，就不存在这样的问题，那么带一个参数情况呢？让我们尝试写个

outer（）：

```
template <typename T> void outer(T& t) {
    inner(t);
}
```

问 题来了，如果传给它的参数是非常量 rvalue，那我们就无法调用 outer()。如果 inner() 接收 const int& 型的参数，那 inner(5) 是可以通过编译的，但是 outer(5) 就编译不过了。因为 T 会被推导为 int， 而 int& 是不能绑定到常量 5 的。

好吧，让我们试试这个：

```
template <typename T> void outer(const T& t) {
    inner(t);
}
```

如果 inner（） 接收 int& 型参数，那就会违法 const 正确性，编译都过不了。

现在，你可以重载两个分别带 T& 和 const T& 参数的 outer（），这确实管用。当你调用 outer（）时，就像直接调用 inner（） 一样。

可 惜的是，这中方法在多参数的情况下就麻烦了（译注：要写的重载函数太多了）。你就得为每一个参数像 T1& 和 const T1&, T2& 和 const T2& 等这样进行重载，要重载的函数数目呈指数级增长。（VC9 SP1 的 tr1：：bind() 就够让人感到绝望了，它为 5 个参数这么重载出了 63 个函数。如果不这么蛮干的话，没有像这里的长篇累述，我们就很难跟使用者解释为什么不能调用用 1729 这样的 ravlue 做参数的函数。为了产生出这些重载函数使用了令人作呕的预处理机制，恶心到你都不想知道它）。

在 C++98/03 中，转发问题是很严重的，而且本质上无解（必须求助于恶心的预处理机制，这会严重拖慢编译速度，还让代码变得难以阅读）。总算， rvalue 优雅地解决了这个问题。

完美转发： 模式

完 美转发让你能简单而清晰地只写一个模板函数就可以转发所有的参数给任意函数，不管它带几个参数，也不管参数类型是什么。而且参数的非常量/常量，lvalue/rvalue 属性都能得以保留，让你可以像使用 inner（） 一样使用 outer（），还可以和 move 语意一起用从而获得额外的好处。（ C++0x 的变长模板技术解决了“任意数目”这部分，我们在这里把 N 看做任意数目）。乍看之下很神奇，实际上很简单：

```
C:\Temp>type perfect.cpp
#include <iostream>
#include <ostream>
using namespace std;

template <typename T> struct Identity {
    typedef T type;
};

template <typename T> T&& Forward(typename Identity<T>::type&& t) {
    return t;
}

void inner(int&, int&) {
    cout << "inner(int&, int&)" << endl;
}

void inner(int&, const int&) {
    cout << "inner(int&, const int&)" << endl;
}

void inner(const int&, int&) {
    cout << "inner(const int&, int&)" << endl;
}

void inner(const int&, const int&) {
    cout << "inner(const int&, const int&)" << endl;
```

```
}

template <typename T1, typename T2> void outer(T1&& t1, T2&&
t2) {
    inner(Forward<T1>(t1), Forward<T2>(t2));
}

int main() {
    int a = 1;
    const int b = 2;

    cout << "Directly calling inner()." << endl;

    inner(a, a);
    inner(b, b);
    inner(3, 3);

    inner(a, b);
    inner(b, a);

    inner(a, 3);
    inner(3, a);

    inner(b, 3);
    inner(3, b);

    cout << endl << "Calling outer()." << endl;

    outer(a, a);
    outer(b, b);
    outer(3, 3);

    outer(a, b);
    outer(b, a);

    outer(a, 3);
```

```
    outer(3, a);

    outer(b, 3);
    outer(3, b);
}

C:¥Temp>cl /EHsc /nologo /W4 perfect.cpp
perfect.cpp

C:¥Temp>perfect
Directly calling inner().
inner(int&, int&)
inner(const int&, const int&)
inner(const int&, const int&)
inner(int&, const int&)
inner(const int&, int&)
inner(int&, const int&)
inner(const int&, int&)
inner(const int&, const int&)
inner(const int&, const int&)

Calling outer().
inner(int&, int&)
inner(const int&, const int&)
inner(const int&, const int&)
inner(int&, const int&)
inner(const int&, int&)
inner(int&, const int&)
inner(const int&, int&)
inner(const int&, const int&)
inner(const int&, const int&)
```

两行！完美转发只用了两行！够简洁吧！

这个例子示范了怎么把 t1 和 t2 从 outer() 透明地转发给 inner()； inner() 可以知道它们的非常量/常量， lvalue/ravlue 属性，就像 inner 是被直接

调用的那样。

跟 std::move() 一样， std::identify 和 std::forward() 都是在 C++<utility> 中定义的（ VC10 会有， VC10 CTP 中没有）。我将演示怎么来实现它们。（再次，我将交替使用 std::identity 和我的 Identity，std::forward() 和我的 Forward()，因为他们的实现是等价的。）

现在，让我们来揭开“魔术“的神秘面纱，其实它靠的就是模板参数推导和引用折叠(reference collapsing)技术。

rvalue 引用：模板参数推导和引用折叠(reference collapsing)

rvalue 引用与模板以一种特别的方式相互作用。下面是一个示例：

```
C:¥Temp>type collapse.cpp
#include <iostream>
#include <ostream>
#include <string>
using namespace std;

template <typename T> struct Name;

template <> struct Name<string> {
    static const char * get() {
        return "string";
    }
};

template <> struct Name<const string> {
    static const char * get() {
        return "const string";
    }
};

template <> struct Name<string&> {
```

```
    static const char * get() {
        return "string&";
    }
};

template <> struct Name<const string&> {
    static const char * get() {
        return "const string&";
    }
};

template <> struct Name<string&&> {
    static const char * get() {
        return "string&&";
    }
};

template <> struct Name<const string&&> {
    static const char * get() {
        return "const string&&";
    }
};

template <typename T> void quark(T&& t) {
    cout << "t: " << t << endl;
    cout << "T: " << Name<T>::get() << endl;
    cout << "T&&: " << Name<T&&>::get() << endl;
    cout << endl;
}

string strange() {
    return "strange()";
}

const string charm() {
    return "charm()";
```

```
}

int main() {
    string up("up");
    const string down("down");

    quark(up);
    quark(down);
    quark(strange());
    quark(charm());
}

C:¥Temp>cl /EHsc /nologo /W4 collapse.cpp
collapse.cpp

C:¥Temp>collapse
t: up
T: string&
T&&: string&

t: down
T: const string&
T&&: const string&

t: strange()
T: string
T&&: string&&

t: charm()
T: const string
T&&: const string&&
```

这里藉由 Name 的显式规格说明来打印出类型。

当 我们调用 quark(up) 时，会进行模板参数推导。 quark() 是一个带有模板参数 T 的模板函数，但是我们还没有为它提供显式的类型参数（比如像

quark<X>(up)这样的）。通过比较函数形参类型 Type&& 和函数实参类型（一个 string 类型的 lvalue）我们就能推导出模板实参类型。（译注：原文用 argument 表示实参，parameter 表示形参）

C++0x 会转换函数实参的类型和形参的类型，然后再进行匹配。

首先，转换函数实参的类型。这遵循一条**特殊规则**（提案 N2798 14.8.2.1[temp.deduct.call]/3）：如果模板形参类型为 T，函数形参类型为 T&&，且函数实参类型为 A 的 lvalue，那么模板实参类型会被推导为 A&。（但这条特殊规则不适用于函数形参类型为 T& 或 const T& 的情况，这种情况下会和 C++98/03 保持一致，另外它也不适用于函数形参类型为 const T&& 的情况）。在 quark（up） 这个例子中，依照这个规则我们会把 string 转换成 string&。

然后，转换函数形参的类型。不管是 C++98/03 还是 C++0x 都会解除引用（lvalue 引用和 rvalue 引用在 C++0x 中都会被解除掉）。在前面例子的四种情形中，这样我们会把 T&& 转换成 T。

于是，**T 会被推导成函数实参转换之后的类型**。up 和 down 都是 lvalue，它们遵循那条特殊规则，这就是为什么 quark(up) 打印出"T:string&"，而 quark(down) 打印出 "T: cosnt string&"的原因。strange() 和 charm() 都是右值，它们遵循一般规则，这就是为什么 quark(strange()) 打印出 "T: string" 而 quark(charm()) 打印出"T: const string" 的原因。

替换操作会在类型推导之后进行。模板形参 T 出现的每一个地方都会被替换成推导出来的模板实参类型。在 quark(string()) 中 T 是 string，因此 T&& 会是 string&&。同样，在 quark(charm()) 中，T 是 const string， 因此 T&& 是 const string&&。但 quark(up) 和 quark(down) 不同，它们遵循另外的特殊规则。

在 quark(up) 中， T 是 string&。进行替换的话 T&& 就成了 string& &&，在 C++0x 中会折叠（collapse）引用的引用，**引用折叠的规则就是“lvalue 引用是传染性的”**。 X& &, X& && 和 X&& & 都会被折叠成 X&，只有 X&& && 会被折叠成 X&&。因此 string& && 被折叠成 string&。在模板世界里，那些看起来像 rvalue 引用的东西并不一定真的就是。 因而 quark(up) 被实例化为 quark<string&>()，进而 T&& 经替换与折叠之后变成 string&。我们可以调用 Name<T&&>::get() 来验证这个。 同样， quark(down) 被实例化为 quark<const

string&>() ，进而 T&& 经替换与折叠之后变成 const string& 。在 C++98/03 中，你可能习惯了常量性(constness)隐藏于模板形参中(也就是说可以传 const Foo 对象作实参来调用形参为 T& 的模板函数，就像 T& 会是 const Foo& 一样)，在 C++0x 中，左值属性(lvalueness) 也能隐藏于模板形参中。

那好，这两条特殊规则对我们有什么影响？在 quark() 内部，类型 T&& 有着和传给 quark() 的实参一样的左/右值属性(lvalueness/rvalueness)和常量性。这样 rvalue 引用就能保持住左右值属性和常量性，做到完美转发。

完美转发： std::forward() 和 std::identidy 是怎样工作的

让我们再来看看 outer() ：

```
template <typename T1, typename T2> void outer(T1&& t1, T2&& t2) {
    inner(Forward<T1>(t1), Forward<T2>(t2));
}
```

现在我们明白了为什么 outer() 的形参是 T1&& 和 T2&& 类型的了，因为它们能够保持住传给 outer() 的实参的信息。那为什么这里要调用 Forward<T1>() 和 Forward<T2>() 呢？还记得么，具名 lvalue 引用和具名 rvalue 引用都是 lvalue 。如果 outer() 调用 inner(t1, t2) ，那么 inner() 总是会当 lvalue 来引用 t1 和 t2 ，这就破坏了完美转发。

幸运的是，不具名 lvalue 引用是 lvalue，不具名 rvalue 引用还是 rvalue 。因此，为了将 t1 和 t2 转发给 inner()，我们需要将它们传到一个帮助函数中去，这个帮助函数移除它们的名字，保持住它们的属性信息。这就是 std::forward() 做的事情：

```
template <typename T> struct Identity {
    typedef T type;
};

template <typename T> T&& Forward(typename Identity<T>::type&& t) {
```

```
    return t;
}
```

当我们调用 Forward<T1>(t1) ， Identidy 并没有修改 T1 （很快我们讲到 Identidy 对 T1 做了什么）。因此 Forward<T1>() 接收 T1&& ，返回 T1&& 。这样就移除了 t1 的名字，保持住 t1 的类型信息（而不论 t1 是什么类型，string& 也好，const string& 也好，string&& 也好或 const string&& 也好）。这样 inner() 看到的 Forward<T1>(t1) ，与 outer() 接收的第一个实参有着相同的信息，包括类型，lvalueness/rvalueness，常量性等等。完美转发就是这样工作的。

你可能会好奇如果不小心写成 Forward<T1&&>(t1) 又会怎样呢？（这个错误还是蛮诱人的，因为 outer() 接收的就是 T1&& t1 ）。很幸运，没什么坏事情会发生。 Forward<T1&&>() 接收与返回的都是 T1&& && ，这会被折叠成 T1&& 。于是，Forward<T1>(t1) 和 Forward<T1&&>(t1) 是等价的，我们更偏好前者，是因为它要短些。

Identidy 是做什么用的呢？为什么下面的代码不能工作？

```
template <typename T> T&& Forward(T&& t) { // BROKEN
    return t;
}
```

如果 Forward() 像是上面那样，它就能被隐式调用（不带明确的模板参数）。当我们传给 Forward() 一个 lvalue 实参时，模板参数推导就介入了，如我们前面看到的那样会将 T&& 变成 T&，也就是变成一个 lvalue 引用。问题来了，即使形参 T1&& 和 T2&& 指明是 rvalue 引用，但在 outer() 中，具名的 t1 和 t2 却是 lvaue ，这个问题是我们一直想要解决的！使用上面那个错误的实现，Forward<T1>(t1) 是可以工作的，而 Foarward(t1) 虽然能通过编译（很诱人哦）但会出错，就如它就是 t1 一样。真是痛苦的源泉啊，因此，Identity **被用来阻止模板参数推导**。typename Identity<T>::type 中的那对冒号就像绝缘体，模板参数推导无法穿越它，有模板编程经验的程序员应该对此很熟悉了，因为这在 C++98/03 和 C++0x 中是一样的。（要解释这个是另外的事情了）

move 语意： std::move() 是怎样工作的

现在我们已经学习了模板参数推导和引用折叠的特殊规则，让我们再来看看 std::move() ：

```
template <typename T> struct RemoveReference {
     typedef T type;
};

template <typename T> struct RemoveReference<T&> {
     typedef T type;
};

template <typename T> struct RemoveReference<T&&> {
     typedef T type;
};

template <typename T> typename RemoveReference<T>::type&&
Move(T&& t) {
    return t;
}
```

RemoveReference 机制基本上是复制 C++0x <type_traits> 中的 std::remove_reference 。举例来说， RemoveReference<string>::type ， RemoveReference<string&>::type 和 RemoveReference<string&&>::type 都是 string 。

同样， move() 机制也基本上是复制 C++0x <utility> 中的 std::move()。

- 当调用 Move(string), string 是一个 lvalue 时，T 会被推导为 string& ，于是 Move() 接收的就是 string& （经折叠之后）并返回 string&& （经 RemoveReference 之后）。

- 当调用 Move(const string), const string 是一个 lvalue 时， T 会被推导为 const string& ，于是 Move() 接收的就是 const string&&（经折叠之后）并返回 const string&&（经 RemoveReference 之后）。

• 当调用 Move(string), string 是一个 rvalue 时，T 会被推导为 string ，于是 Move() 接收的就是 string&& 并返回 string&& 。

• 当调用 Move(const string), const string 是一个 rvalue 时， T 会被推导为 const string ，于是 Move() 接收的就是 const string&& 并返回 const string&& 。

这就是 Move() 如何保持其参数的类型和常量性，还能把 lvalue 转换成 rvalue 的过程。

回顾

如果你想对 rvalue 引用有更多了 解，你可以去读有关它们的提案。要注意，提案与现在的决定可能已经不同了， rvalue 引用已经被整合到 C++0x 草案中来了，在那里它得到持续的改进。有些提案或已不再正确，或已过时，或已有了替代方案，就没有被采纳。无论怎样，它们还是能提供一些有用信息的。

N1377, N1385, 和 N1690 是主要的提案，N2118 包含被整合进标准草案之前的最后版本。 N1784, N1821, N2377, 和 N2439 记录了“将 Move 语意扩展到 *this ”的演化过程，这个也被整合到 C++0x 中来了，但还没有在 VC10 中得到实现。

展望

N2812 “Rvalue 引用的安全问题（以及如何解决）” 提出了对初始化规则的修改，它禁止 rvalue 引用绑定到 lvalue 。 这不会影响 move 语意和完美转发，所以它不会让你刚学到的新技术失效（它只是修改了 std::move() 和 std::forward() 的实现）。

Stephan T. Lavavej
Visual C++ Libraries Developer
Published Tuesday, February 03, 2009 9:27 AM by vcblog

翻译：飘飘白云
译文链接：http://www.cppblog.com/kesalin/archive/2009/05/28/85983.html
(转载时请注明作者和出处。未经许可，请勿用于商业用途)

VC10 中的 C++0x 特性 part 3 ：声明之类型

翻译：飘飘白云

译文链接：http://www.cppblog.com/kesalin/archive/2009/05/28/85983.html

(转载时请注明作者和出处。未经许可，请勿用于商业用途)

今天我要讲 decltype，它让完美转发函数能够返回任意类型的东西。对编写高度泛型的人来说这是很有趣的的特性。

返回类型问题

C++98/03 有一个有意思的盲点：给定一个像 x * y 的表达式， x 和 y 是任意类型，你却没法知道 x*y 的类型。假如 x 是 Watts 类型的， y 是 Seconds 类型的，那 x*y 的类型可能会是 Joules 类型的。 给定声明 print(const T& t)，调用 print(x * y) ，在这里 T 会被推导为 Joules 类型。但反过来却不是这样的：当你写个函数 multiply(const A& a, const B& b) ，你无法指定它的通用返回类型。即使是实例化成 multiply<A, B>() ，编译器也晓得 x*y 的类型，但你就是没办法得到那样的信息（指返回类型）。C++0x 中的关键词 decltype 扫除了这个盲点，让你能够说“ multiply() 返回 x*y 类型的东西”。（decltype 是 "declared type" 的缩写，我把它读作谐音 “speckle type”。）

decltype：模式

下面是一个完全泛化的封装 +() 操作符的函数因子。这个“加法”因子不是一个模板，但它有一个模板函数，这个模板函数带两个任意类型（当然是不同类型的）参数，并把它们想加，然后返回任意类型（可能跟两个参数的类型完全不同）的结果。

```
C:¥Temp>type plus.cpp
#include <algorithm>
#include <iostream>
#include <iterator>
#include <ostream>
#include <string>
#include <utility>
#include <vector>
using namespace std;
```

```
struct Plus {
    template <typename T, typename U>
    auto operator()(T&& t, U&& u) const
    -> decltype(forward<T>(t) + forward<U>(u)) {
        return forward<T>(t) + forward<U>(u);
    }
};

int main() {
    vector<int> i;
    i.push_back(1);
    i.push_back(2);
    i.push_back(3);

    vector<int> j;
    j.push_back(40);
    j.push_back(50);
    j.push_back(60);

    vector<int> k;

    vector<string> s;
    s.push_back("cut");
    s.push_back("flu");
    s.push_back("kit");

    vector<string> t;
    t.push_back("e");
    t.push_back("ffy");
    t.push_back("tens");

    vector<string> u;

         transform(i.begin(),      i.end(),      j.begin(),
back_inserter(k), Plus());
```

```
        transform(s.begin(),      s.end(),      t.begin(),
back_inserter(u), Plus());

    for_each(k.begin(), k.end(), [](int n) { cout << n << "
"; });
    cout << endl;

    for_each(u.begin(), u.end(), [](const string& r) { cout <<
r << " "; });
    cout << endl;
}

C:¥Temp>cl /EHsc /nologo /W4 plus.cpp
plus.cpp

C:¥Temp>plus
41 52 63
cute fluffy kittens
```

拿 C++98/03 <functional> 中的 std::plus<T>（在 C++0x 没有变动)来作对比，后者是一个类模板，你不得不传递模板参数类型来调用 plus<int>() 和 plus<string>()，重复声明一次元素类型。并且后者那个形式为 T operator()(const T& x, const T& y) 的非模板函数调用操作符，如果不借助于隐式类型转换，就不能将两种不同类型的东西相加，更不用说 3 种不同类型的情况了（译注：两个参数类型 + 一个返回类型）。（你可以传递 string 和 const char * 类型的实参来调用 const plus<string>()，那样的话就会在串接操作之前，基于第二个参数(const char *)构建一个临时 string，这样做在性能上不可取）。 再者，因为它的参数是 const T& 形式的，这就不能使用 C++0x 的 move 语意来获得好处。Plus 避免了上述问题：调用 Plus() 不需要重复声明元素的类型，它也可以处理 “3 种” 不同类型的情况，并且它用了完美转发，因而能够使用 move 语意。

trailing return type

现在让我们再来看看这个模板函数调用操作符：

```
template <typename T, typename U>
auto operator()(T&& t, U&& u) const
-> decltype(forward<T>(t) + forward<U>(u)) {
    return forward<T>(t) + forward<U>(u);
}
```

这里的 auto 与 for (auto i = v.begin(); i != v.end(); ++i) 中的含义完全不同，在 for 中它是指“把用来初始化对象的类型当做对象的类型”，而在这里它是指“这个函数有 trailing-return-type，只有指定实参之后，才能确定它返回什么类型”(C++0x 提案 N2857 中把这个称作 “*late-specified return type*”，但它将被重命名为 “trailing-retrun-type”（提案 N2859 ））。这里看起来和 lambda 函数是如何指定返回类型的很相似，其实它们就是一样的。lambda 函数的返回类型必须出现在 lambda 导引符 [] 之后（右边）。在这里，decltype-powered 类型也必须出现在函数参数 t 和 u 之后（右边）。autoT 和 U 对它是可见的，但是函数参数 t 和 u 还不可见，这就是为什么需要 decltype 的原因。（从技术上来讲， decltype(forward<T>(*static_cast<T *>(0)) + forward<U>(*static_cast<U *>(0))) 可以在左边出现，但那看起来会让人不舒服）。

至于在返回语句中还要使用与传给 decltype 的表达式相同的形式，是为了确保在任何情况下都能正确工作。(突击测验：为什么 decltype(t + u) 就不对呢？)。这里的重复是不可避免的，但因为集中-只出现一次且代码位置靠近，所以不会有什么危险。

另一个例子

考虑到例子的完整性，下面是一个 “3 种” 不同类型的示例：

```
C:¥Temp>type mult.cpp
#include <algorithm>
#include <iostream>
#include <iterator>
#include <ostream>
#include <utility>
#include <vector>
using namespace std;
```

```
struct Multiplies {
    template <typename T, typename U>
    auto operator()(T&& t, U&& u) const
    -> decltype(forward<T>(t) * forward<U>(u)) {
        return forward<T>(t) * forward<U>(u);
    }
};

class Watts {
public:
    explicit Watts(const int n) : m_n(n) { }
    int get() const { return m_n; }
private:
    int m_n;
};

class Seconds {
public:
    explicit Seconds(const int n) : m_n(n) { }
    int get() const { return m_n; }
private:
    int m_n;
};

class Joules {
public:
    explicit Joules(const int n) : m_n(n) { }
    int get() const { return m_n; }
private:
    int m_n;
};

Joules operator*(const Watts& w, const Seconds& s) {
    return Joules(w.get() * s.get());
}
```

```
int main() {
    vector<Watts> w;
    w.push_back(Watts(2));
    w.push_back(Watts(3));
    w.push_back(Watts(4));

    vector<Seconds> s;
    s.push_back(Seconds(5));
    s.push_back(Seconds(6));
    s.push_back(Seconds(7));

    vector<Joules> j;

         transform(w.begin(),      w.end(),      s.begin(),
back_inserter(j), Multiplies());

    for_each(j.begin(), j.end(), [](const Joules& r) { cout <<
r.get() << endl; });
}

C:¥Temp>cl /EHsc /nologo /W4 mult.cpp
mult.cpp

C:¥Temp>mult
10
18
28
```

你可能会问“所有的这些处理真的有必要么”，答案是 Yes ，有必要。我已经介绍了完美转发和 decltype 是如何让算术运算函数因子使用起来更容易（不用重复声明元素类型），更灵活（可以混合使用不同的参数和返回类型），更有效率（使用 move 语意）。最重要的是，完美转发和 decltype 让你能够编写更简洁明了的代码，而不灵活和低效的代码不是简洁明了的-这点是我们无法忽视的。

高级规则

decltype 是有一些规则来驱动的。然而，如果你遵照上面的模式就没关系，能正常工作。我很少那样说 C++ ，但是在这里是这样的。

虽然大多数 decltype 应用遵循上面介绍的模式，但 decltype 还可以用于其他环境。在那些情况下，你就用到了 decltype 的高级模式，你应该全面地阅读那些规则，它们在 C++0x 提案 N2857　7.1.6.2 [dcl.type.simple]/4 中被给出。

等等，还有一些要说的

decltype 是第五个且是最后一个添加到 VC10 中的 C++0x 核心语言特性。虽然 VC10 CTP 中还没有，但 VC10 Bata 1 中会有。而且 VC10 Beta 1 中还有很多 C++0x 标准库特性，我会在后续文章中介绍它们。

Stephan T. Lavavej
Visual C++ Libraries Developer
Published Wednesday, April 22, 2009 10:06 AM by vcblog
翻译：飘飘白云
译文链接：http://www.cppblog.com/kesalin/archive/2009/05/28/85983.html

(转载时请注明作者和出处。未经许可，请勿用于商业用途)

<- 全文完 ->