命名返回值优化和成员初始化队列

命名返回值优化

对于一个如 `foo()` 这样的函数，它的每一个返回分支都返回相同的对象，编译器 有可能对其做 Named return Value 优化（下文都简称 NRV 优化），方法是以一个参 数 `result` 取代返回对象。

`foo()` 的原型：

```
X foo() {
    X xx;
    if(...)
        returnxx;
    else
        returnxx; }
```

优化后的 `foo()` 以 `result` 取代 `xx`：

```
void foo(X &result){
    result.X::X();
    if(...)
    {
        //直接处理 result
        return;
    }
    else
    {
        //直接处理 result
        return;
    }}
```

对比优化前与优化后的代码可以看出，对于一句类似于 `X a = foo()` 这样的代 码，NRV 优化后的代码相较于原代码节省了一个临时对象的空间（省略了 xx）,同 时减少了两次函数调用（减少 xx 对象的默认构造函数和析构函数，以及一次拷贝 构造函数的调用，增加了一次对 a 的默认构造函数的调用）。

注：Lippman 在《深度探索 C++》书中指出 NRV 的开启与关闭取决于是否有显式定 义一个拷贝构造函数，我实在想不出有什么理由必须要有显示拷贝构造函数才能 开启 NRV

优化，于是在 vs2010 中进行了测试，测试结果表明，在 release 版本中， 不论是否定义了一个显式拷贝构造函数，NRV 都会开启。由此可见 vs2010 并不以 是否有一个显式拷贝构造函数来决定 NRV 优化的开启与否。但同时，立足于这一 点，可以得出 Lippman 所说的以是否有一个显式定义的拷贝构造函数来决定是否 开启 NRV 优化，应该指的是他自己领导实现的 cfront 编译器，而非泛指所有编译 器。那么 cfront 又为什么要以是否定义有显示的拷贝构造函数来决定是否开启 NRV 优化呢？我猜测，他大概这样以为，当显式定义有拷贝构造函数的时候一般 代表着要进行深拷贝，也就是说此时的拷贝构造函数将费时较长，在这样的情况 下 NRV 优化才会有明显的效果。反之，不开启 NRV 优化也不是什么大的效率损失。

另外，有一点要注意的是，NRV 优化，有可能带来程序员并不想要的结果，最明 显的一个就是——当你的类依赖于构造函数或拷贝构造函数，甚至析构函数的调用 次数的时候，想想那会发生什么。由此可见、Lippman 的 cfront 对 NRV 优化抱有更 谨慎的态度，而 MS 显然是更大胆。

成员初始化队列（Member Initialization List）

对于初始化队列，我相信厘清一个概念是非常重要的：在构造函数中对于对象 成员的初始化发生在初始化队列中——或者我们可以把初始化队列直接看做是对 成员的定义，而构造函数体中进行的则是赋值操作。所以不难理解有四种情况 必须用到初始化列表：

- 有 const 成员
- 有引用类型成员
- 成员对象没有默认构造函数
- 基类对象没有默认构造函数

前两者因为要求定义时初始化，所以必须明确的在初始化队列中给它们提供初 值。后两者因为不提供默认构造函数，所有必须显示的调用它们的带参构造函 数来定义即初始化它们。

显而易见的是当类中含有对象成员或者继承自基类的时候，在初始化队列中初 始化成员对象和基类子对象会在效率上得到提升——省去了一些赋值操作嘛。

最后，一个关于初始化队列众所周知的陷阱，初始化队列的顺序，请参考《C++ primer》或者《深度探索 C++对象模型》。