### [Named Return Value Optimization (具名返回值优化)](http://blog.csdn.net/fanster28_/article/details/5006993)

分类： [C++ 学习](http://blog.csdn.net/fanster28_/article/category/631628)2009-12-14 23:09 736人阅读 [评论](http://blog.csdn.net/fanster28_/article/details/5006993#comments)(5) 收藏 [举报](http://blog.csdn.net/fanster28_/article/details/5006993#report)

[optimization](http://www.csdn.net/tag/optimization)[优化](http://www.csdn.net/tag/%e4%bc%98%e5%8c%96)[constructor](http://www.csdn.net/tag/constructor)[编译器](http://www.csdn.net/tag/%e7%bc%96%e8%af%91%e5%99%a8)[delete](http://www.csdn.net/tag/delete)[object](http://www.csdn.net/tag/object)

Named Return Value Optimization 具名返回值优化

问题描述 :

当一个函数返回一个对象的实例，一个临时对象被创建并通过拷贝构造函数传回其值。 C++ 标准允许省略拷贝构造函数的对象（即使会导致程序结果的不一致）。

前提是所有的路径返回相同的对象。

问题由来：

看如下的函数：

X bar() {

X xx;

//… 处理 xx

return xx;

}

如果看到了这篇文章，大概应该知道编译期面对这个函数会做如下的转化：

//Pseudo C++ Code

void bar( X &\_\_result ) {

X xx;

xx.X::X(); //default constructor

// .. 处理 xx

\_\_result.X::X( xx ); // copy constructor

return;

}

如上述 bar() 这样，所有的路径都返回相同的具名数值 (Named Value) ，因此编译器可以做自己的优化。以 \_\_result 取代 Named Return Valued 。

//Pseudo C++ Code

void bar( X &\_\_result ) {

\_\_result.X::X(); //default constructor

// .. 处理 xx

return;

}

编译器呢？

B.Lippman 的 Inside the C++ Object Model 一书中如是说 :

NRV 优化如今被视为标准 C++ 编译器的一个义不容辞的优化操作。真的如此吗？

其实 Visual C++ 熬到了 8.0 才支持了 [Named Return Value Optimization](http://msdn.microsoft.com/en-us/library/ms364057%28VS.80%29.aspx) 。

实际的例子：

以下例子均是在 VS2010 beta2 中测得。

#include <ctime>

#include <memory>

#include <iostream>

using namespace std ;

class test {

friend test foo ( double );

public :

test () {

memset ( array , 0 , 100 \*sizeof ( double ) );

cout << "test constructor " << '/n' ;

}

~test () {

cout << "test deconstructor " << '/n' ;

}

test ( const test &t );

private :

double array [ 100 ];

};

inline

test ::test ( const test &t ) {

memcpy ( this , &t , sizeof ( test ) );

cout << "test copy " << '/n' ;

}

test foo ( double val ) {

test local , local2 ;

local .array [ 0 ] = val ;

local .array [ 99 ] = val ;

if (val > -1 )

return local ;

}

int main() {

test t1 = foo( 2 );

return 0;

}

在没有 NRVO 优化的时候，编译优化选项关闭 (cl /Od) ，得到

test constructor

test constructor

test copy

test deconstructor

test deconstructor

test deconstructor

在打开 NRVO 优化时， (cl /O2) ，得到

test constructor

test constructor

test deconstructor

test deconstructor

可以看到，恰好是少了一次局部对象 xx 到 \_\_result 的拷贝，以及析构。

注意的地方：

B.Lippman 在书中说到， NRVO 要由 Copy Constructor 激活，实际上不是的，至少 MS 在 VC中没有这样做。

注释掉上述拷贝构造函数， NRVO 依然可进行。实际上注释掉之后编译器会产生一个版本（bitwise copy ），可以从结果的两次构造，三次析构中看出来，少了的那次就是默认拷贝构造函数的调用。实行了优化后可见到只有两次析构的调用了，说明没有了默认拷贝构造函数的调用，编译器甚至不用产生它（需求时才产生）。

事实上，即使是没有定义拷贝构造函数和析构函数，而是有编译器产生（在需要时）， NRVO依然会对程序的效率产生影响。如此能否说明 cfront 中以 是否存在拷贝构造作为 NRVO 的开关 存在争议？

首先注释调拷贝构造函数和析构函数 ，并在上述代码中增加：

friend test foo2 ( double );

test foo2 ( double val ) {

test local , local2 ;

local .array [ 0 ] = val ;

local .array [ 99 ] = val ;

if (val > -1 )

return local ;

return local2 ; //return 了不同的对象，编译器傻眼了 ^\_^ ，不能 NVRO

}

注释调函数中的 cout 那些行，并 重写 main 函数如下：

int main () {

test \*p = new test [100 ]; // 让 VC 不能把整个循环当常量表达式优化了

int cnt ;

clock\_t begin = clock ();

for ( cnt = 0 ; cnt < 10000000 ; cnt ++ ) {

p [cnt %100 ] = foo ( double ( cnt ) );

}

clock\_t end = clock ();

cout << "foo with NRVO used " << end - begin << "ms" << endl ;

begin = clock ();

for ( cnt = 0 ; cnt < 10000000 ; cnt ++ ) {

p [cnt %100 ] = foo2 ( double ( cnt ) );

}

end = clock ();

cout << "foo2 without NRVO used " << end - begin << "ms" << endl ;

delete []p ;

return 0 ;

}

在我这里测得结果如下：

foo with NRVO used 1480ms

foo2 without NRVO used 2631ms

可以看到没有 NVRO 时多的那一次 biwise copy 100\*8bits 以及析构多花费的时间。

如果要知道更多编译器施行 NRVO 的限制，参考 [这里](http://msdn.microsoft.com/en-us/library/ms364057%28VS.80%29.aspx) 。

总结：

NRVO 曾经也饱受批评，有人认为编译器默默的做了这件事情，而程序员们并不知道是否做了，或者有些人不希望自己的程序被优化。

对于 NRVO 带来的副作用，可以考虑对象中有一个记录了其被拷贝次数的 static 变量，显然，NRVO 的应用可能会导致程序结果的不一致。