OPEN CASCADE 学习笔记
——内存是否泄漏了

著: Roman Lygin

译： George Feng

这是一篇关于开源三维建模软件 OPEN CASCADE 内核的博文：ROMAN LYGIN 是 OPEN CASCADE 的前程序开发员和项目经理，曾经写过许多关于该开源软件开发包的深入文章，可以在他的博客 (HTTP://OPENCASCADE.BLOGSPOT.COM) 上面找到这些文章。

序

在 Open Cascade 的论坛上知道了 Roman Lygin 在他的博客上写了 Open Cascade notes 系列文章，考虑到 Open Cascade 的学习资料并不多，于是从他的博客上下载了其中绝大部分文章，将其翻译过来以方便大家学习交流。如果大家发现文中翻译有错误或不足之处，望不吝赐教，可以发到我的邮箱 fenghongkui@sina.com.cn ，十分感谢。

2012 年 11 月 22 日星期四

第 1 节 泄漏的症状

经常在 Open CASCADE 的论坛上出现帖子抱怨存在永久内存泄漏 (persistent memory leaks) 问题。实话说，在我访问的其他软件产品的论坛上也有这样问题，所以这个问题不是 OCC 所特有的。

所以我希望在此讲述下该问题，希望帮助人们理解这个问题。欢迎各种扩展问题和评论。

那么怎么发现存在内存泄漏呢？我认为有下面这些可能 (按照程序员出错多少的

顺序由高到低排列)。

使用 Visual Studio 的内置特性
(虽然我从来没有使用过)

将下面的代码加入到源程序或者可执行程序中：

```
#define _CRTDBG_MAP_ALLOC
#include <stdlib.h>
#include <crtdbg.h>
```

在 main() 函数中添加：

```
_CrtSetDbgFlag ( _CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF );
```

要获取更多信息，请阅读 MSDN 页面 (选择正确的 VS 版本)。

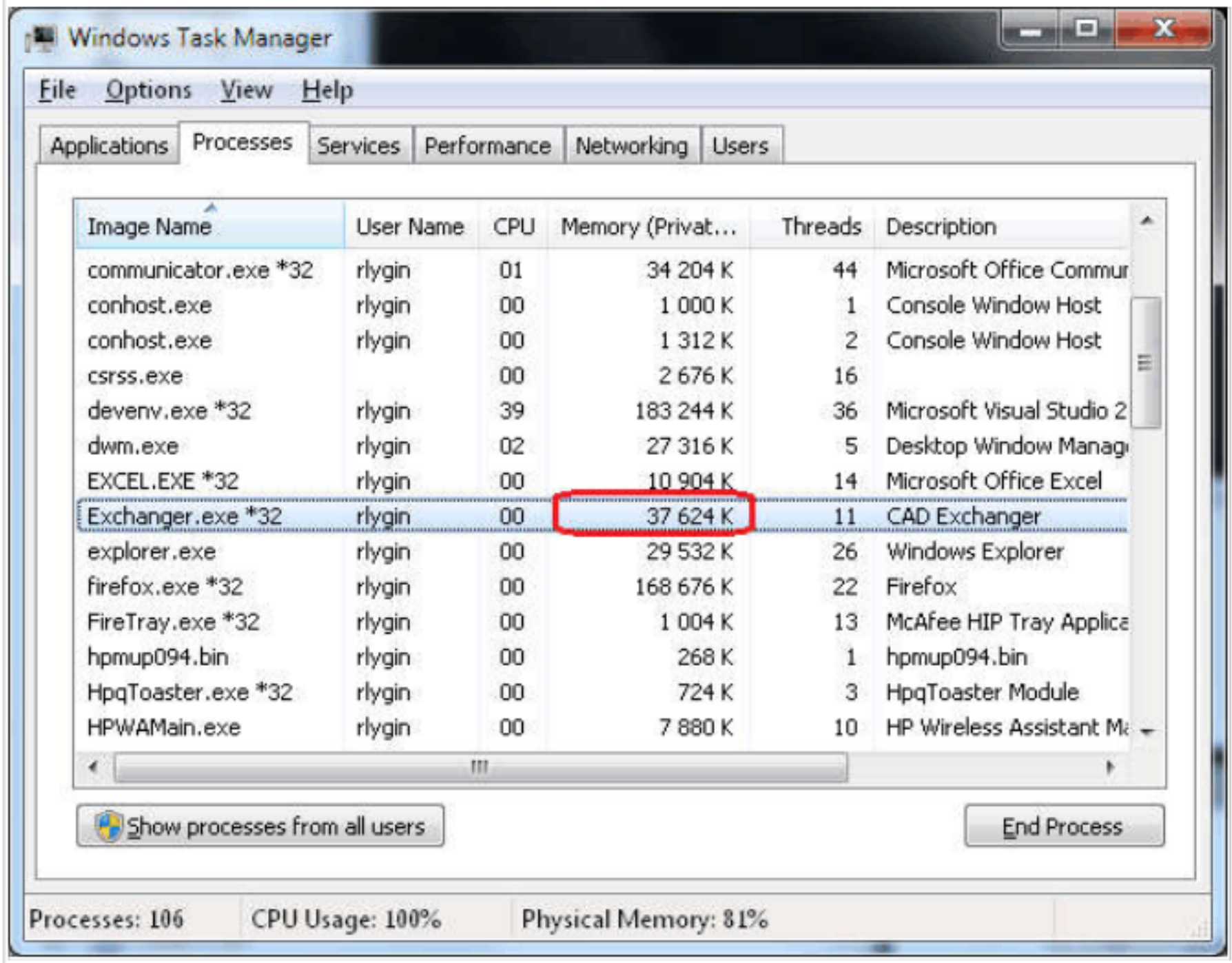
在调试器中，输出 (Output) 窗口可以看到像下面的提示：

```
Detected memory leaks!
Dumping objects ->
{35171} normal block at 0x04CD0068, 260 bytes long.
Data: < > 02 00 00 00 00 00 00 00 CD CD CD CD CD CD CD CD
{349} normal block at 0x0330E350, 100 bytes long.
Data: < > CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD
{246} normal block at 0x03309390, 108 bytes long.
Data: < E ( > 01 00 00 00 1A 00 00 00 45 01 00 00 28 0A 00
00
```

一旦看到这些东西，你可能发现自己在责怪——“ 怎么这个 !@#$% 产品存在这么多的基础错误？！ !”。但是很快你就发现你自己的程序中也有这样的问题，你就会说“ 嗯，这真是个错误吗？ ”随着你深入调试，事情可能就不一样了，你可能会发现你的产品中的一个错误或者内存检测工具的局限 (虽然后者更有可能)。

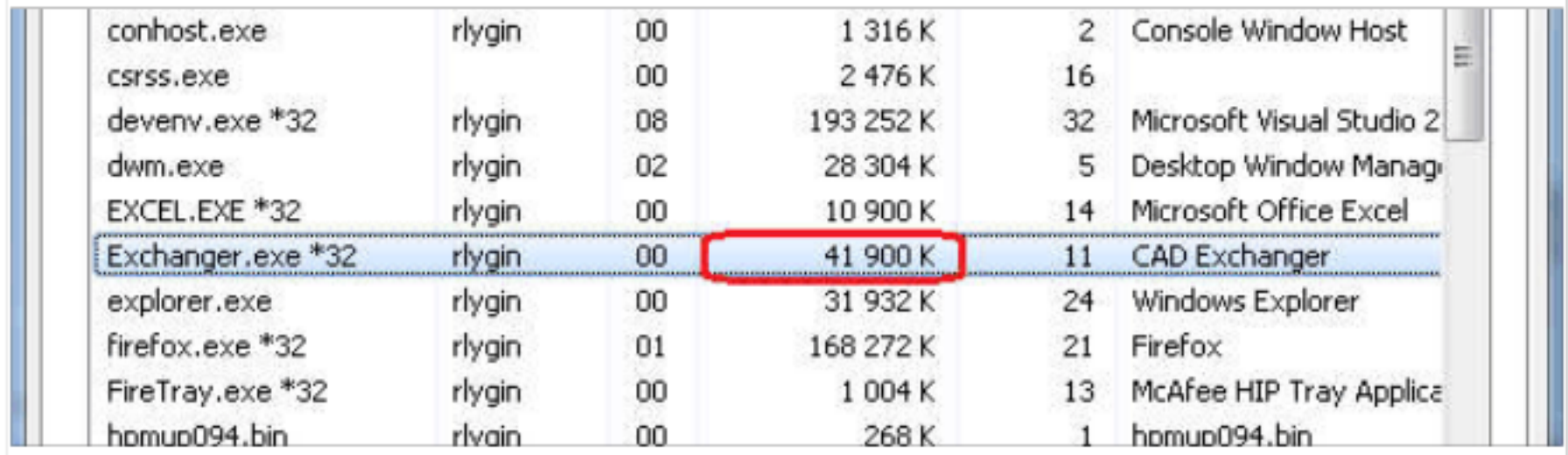
Windows 任务管理器

启动程序后在开始任何操作之前 (例如，打开多文档界面 (Multi-Document Interface, MDI) 应用程序中的一个文档)，可以在 Windows 任务管理器中看到执行程序占用的内存。



在打开文档之前的 MDI 应用程序的内存占用大小

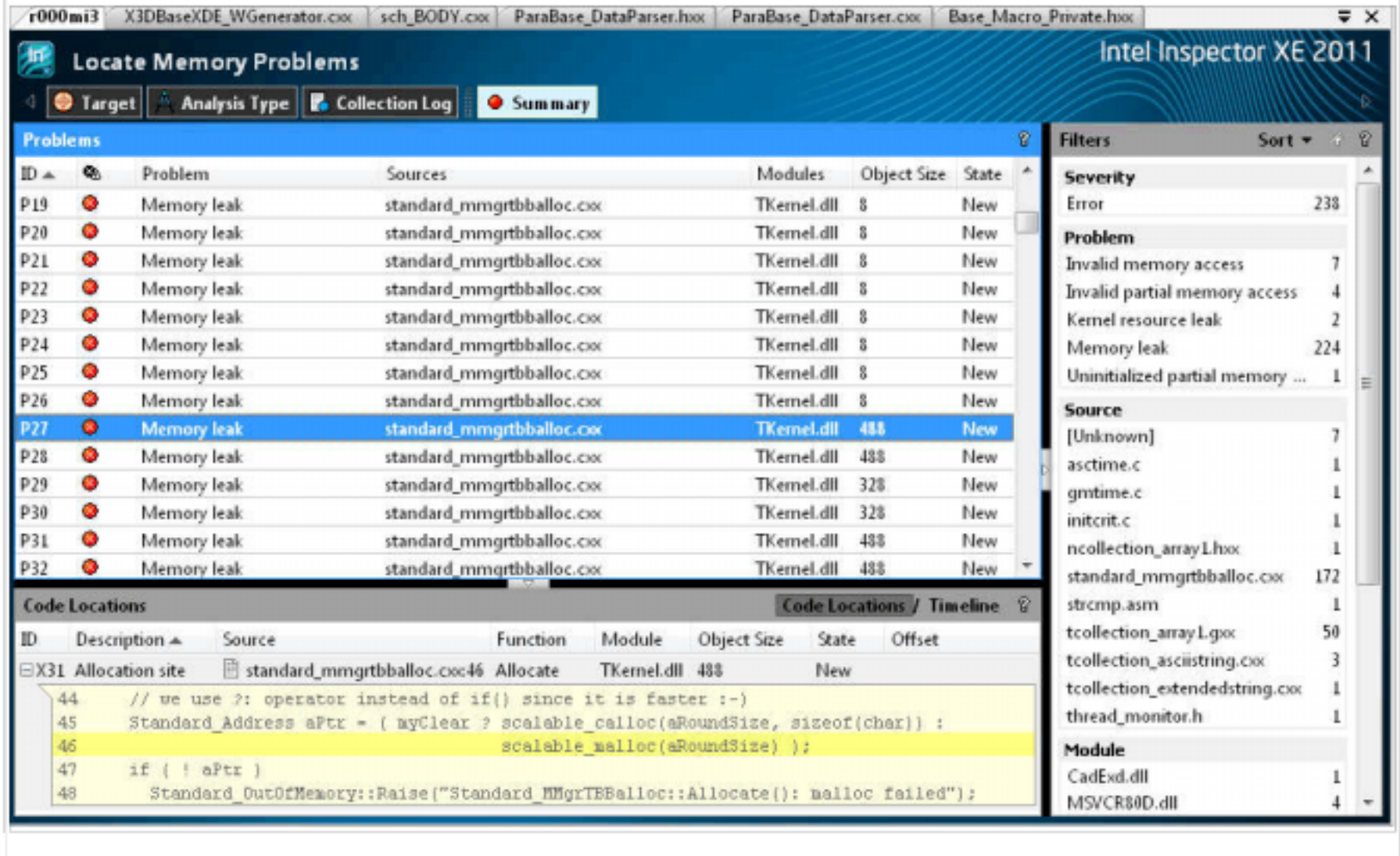
然后进行一些操作，再看看消耗的内存。例如，在关闭 MDI 应用程序的文档之后，内存占用应该回到之前的占用大小。 假如不是这样 (通常来说， 不是这样！)，于是你开始问为什么。



在关闭打开的文档之后同样的应用程序内存占用大小

专用内存检测工具

专用内存检测工具包括 Valgrind, Bounds Checker (我从来没有用过这两个软件), Rational Purify, Intel Parallel Inspector XE, 等。我在 1990 年代末使用过 Purify，从 2008 起就一直使用 Intel Inspector 。下面是 Inspector 生成报告的例子。



Intel Parallel Inspector XE 报告内存泄漏

你可以使用从 Intel 网站下载 试用版。

调试 (Debug) 打印

如果要检测实例是不是在你希望的时候销毁， 在构造函数和析构函数中添加输出是最简单的做法，但是非常的有效。

用户检测内存

你可能想自己写一些专用的内存检测的程序——一些钩子程序可以追踪内存分配和释放函数 (malloc/free, new/delete) ——记录分配和释放内存的字节数。我自己也写过一个这样的程序来追踪 CAD Exchanger 的内存使用情况。假如大家很感兴趣， 我可以把它发布出来。 这个程序的功能是检测代码在执行完成之后所分配的内存是否都释放出来了。

(待续... ...)

POSTED BY ROMAN LYGIN AT 12:45, 2011-06-02 

第 **2** 节　内存泄漏的原因

(接上节… …)

上面讨论了可能出现的症状，现在让我们来找一下泄漏的原因。

1. 真的泄漏

当用原始语言 (C/C++) 开发程序时，你可能会忘记释放已经分配的内存。下面是一个例子：

a. 就像下面这段代码一样简单：

```
{
char* p = (char*)malloc (1 * 1024);

//do work...

//free (p); //    不要忘记去掉注释符号
}
```

b. 结构设计缺陷。对象隶属关系不清，对象寿命 (life-span) 管理混乱从而导致没有正确的销毁对象。

我在 Salome(SMESH 模块)中发现了这个问题。其中各种普通指针 (不是类似于 boost::shared_ptr 的智能指针) 在各个对象之间具有复杂的依赖关系。我估计有很多程序员在共同维护这些代码， 某天忘记了哪些对象要销毁。 下面是我最近解决的一个问题——使用空实体 (a null shape) 调用 SetShapeToMesh() ，从而销毁 SMESH_Mesh 中的子网格 (sub-meshes) 。这样就可以销毁所有的存储在内部映射图 (internal map) 中的对象，否则就会导致泄漏 (SMESH_Mesh::~SMESH_Mesh() 析构函数不能销毁它们) ：

```
/*!    释放 SMESH_Mesh 中分配的资源，否则会出现泄漏
```

- Salome 中的错误。

```
*/
Mesh_MeshImpl::~Mesh_MeshImpl()
{
TopoDS_Shape aNull;
mySMesh->ShapeToMesh (aNull);
}
```

其中 mySMesh 定义为：

```
boost::shared_ptr mySMesh;
```

c. 智能指针循环指向。假如你有两个智能指针相互指向，它们就不能销毁 (因为引用计数永远不能到 0)。我在 第一个帖子 (Let's handle'em)中就讨论了。

真实内存泄漏 (True leaks) 很容易就被内存检测软件捕获到了。

2. 内存分配器的缓存

许多复杂的软件都具有集成的内存分配器，能够比默认的分配器 (操作系统的一部分或者 C 运行时库)更有效管理内存 (至少在速度或者日志 (footprint) 方面）。Open CASCADE 也具有自己的内存分配器 (由环境变量 MMGT_OPT=1 激活), Intel TBB (MMGT_OPT=2) ，或者缺省的系统分配器 (MMGT_OPT=0) 。

虽然操作系统提供的分配器越来越好，自定义分配器在可预见的未来仍然会存在，因为自定义分配器可以更有效的解决特定的问题 (例如 TBB 中能够解决线程安全和可量测性问题)。假如你有疑问，你或许想在缺省的分配器、 OCC 分配器和 TBB 分配器之间做一些对比。

分配器的主要思想是缓存以及重用之前分配的内存块。 所以，当你的应用程序对象销毁时，分配器会收回内存，而不是回到系统中。哪就是为什么，你不会在任务管理器中看到程序内存占用恢复到之前的大小，即使是在关闭了 MDI 文档后所有的文档对象都销毁了。 分配器可能会使用不同的策略保留 / 返还这些内存块。例如， OCC 和 TBB 对于小块和大块内存有不同的策略。大块内存可能很快就返还到系统中 (因为重新使用它们的机会较小)，小块内存可能直到应用程序结束才会返还到系统中。

3. 驻留在内存中的静态对象

实际当中经常会碰到生成静态对象， 这些静态对象存在于整个应用程序的生命周期，只有在程序结束的时候才会销毁。考虑下面的代码：

myfile.cpp:

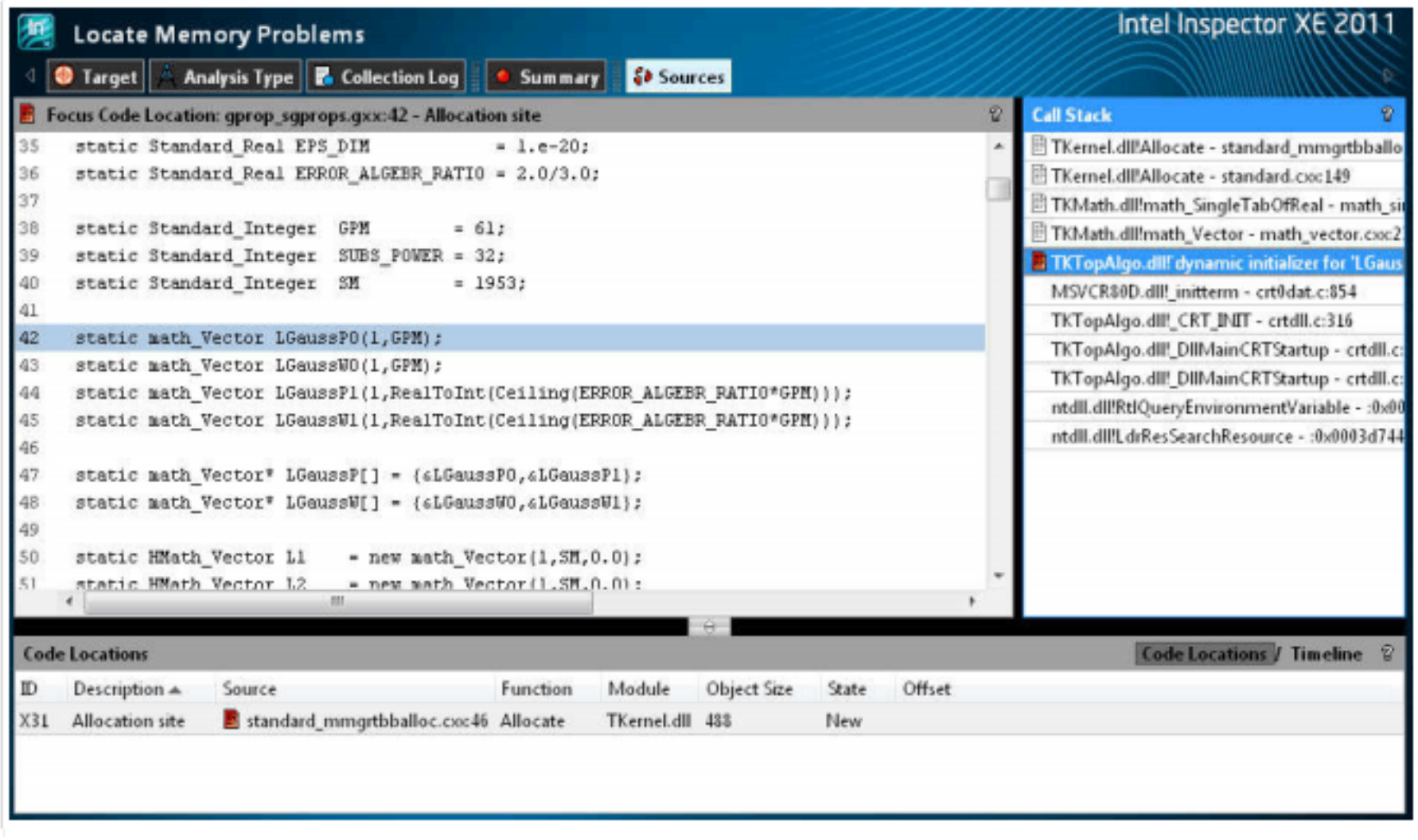
```
static boost::shared theSingleton = new MyClass();

MyClass* MyClass::Instance()
{
return theSingleton.get();
}
```

theSingleton 在装载包含它的库时将会生成，在库被卸载时销毁 (直到应用程序结束时才生效，除非显示卸载库) 。

在 OCC 的代码中有好几个这样构造的例子。

下图是第 1 节中的 Inspector 检测到内存泄漏 (虚假报警)的屏幕截图：



TKTopAlgo 中的静态对象

4. 驻留在内存中的不再使用的数据

与上面类似，有些数据是以静态对象的方式存储的，并且在算法调用之间传递。在更早的帖子 (Unnoticeable memory leaks) 里面我给出了一些例子。 我认为这是一种很坏的设计方式， 应该尽量避免， 但是在第三方软件中仍然可能发生这种情况。它不是内存泄漏， 但是非常浪费内存， 这些内存也只能到程序结束的时候释放。

(待续… …)

POSTED BY ROMAN LYGIN AT 20:45, 2011-06-09 

第 3 节 下一步怎么办

(接上节… …)

好了，现在你知道一些可能内存泄漏的原因了。下一步怎么办？

#1. 假如你认为你找到了一处内存泄漏的地方，首先，不要犹豫。耐心的确定这个泄漏是真的内存泄漏还是虚假报警 (a false positive) 。要确定这一点，需要接着做下面的步骤：

#2. 使用有效的内存检测工具 (Intel Parallel Inspector XE, Valgrind 等)。

#3. 假如你想排除自定义内存分配器 (custom memory allocator) 的问题，一定要设置环境变量 MMGT_OPT=0 ，然后进一步实验。

#4. 假如问题解决了， 那就可能是分配器的问题， 那么就有可能是一个虚假报警，而不是分配器问题。当然你也可以继续查看。

#5. 当涉及你自己的应用程序框架时， 一定要明白你的内存是如何管理的。 一定要始终使用智能指针——例如， boost 的智能指针或者 Open CASCADE 的句柄。这会节省你的枯燥的调试时间。

#6. 在使用 OCC 分配器 (MMGT_OPT=1) 时，假如你对内存的消耗有怀疑，你可以周期性的调用 Standard::Purge() 。这样就会释放不再使用的小块的内存。例如可以在关闭 MDI 文档时调用 (虽然我自己从来不使用)。

#7. 高级程序员可能想更深入的研究问题并且使用更好的优化技术， 例如内存池 (或者区)。参见 NCollection_IncAllocator ，它就是一个这样的例子。 NCollection 容器能够接受 NCollection_IncAllocator 参数。TBB 在将来的版本中将支持线程安全的内存池。

#8. 黑带选手可能想试验 OCC 分配器允许的内存追踪函数。参见函数 Standard_MMgrOpt::SetCallBackFunction() ，在每一次分配 / 释放内存后可以调用该函数。也可以是任意追踪分配 / 释放内存大小和地址的用户回调函数。

好了，这就是我能够想起来的关于这个问题的所有内容， 希望对你有所帮助。 正如我在一开始就说的，欢迎提出任何扩展内容或者其他好的方法。

Roman
POSTED BY ROMAN LYGIN AT 20:30, 2011-06-19