Open Cascade 中的内存管理

Memory Management in Open Cascade

eryar@163.com

一、***C***++中的内存管理 ***Memory Management in C***++

1. 引言

为了表现出多态，在 ***C***++中就会用到大量的指针和引用。指针所指的对象是从内存空间中借来的，当然要及时归还。特别是指针在程序中随心所欲地创建，因此，一个指针究竟指向哪个对象，一个对象到底被几个指针所指向，是程序员十分关注的事情。

C++中涉及到的内存管理问题可以归结为两方面：正确地掌握它和有效地使用它。好的程序员会理解这两个问题为什么要以这样的顺序列出。因为执行得再快、体积再小的程序，如果不按所期望的方式去执行也是没什么用处的程序。对于大多数程序员，正确地掌握意味着正确地调用内存分配和释放函数；有效地使用意味着编写自定义版本的内存分配和释放函数。显然，正确地掌握它要重要些。

在 ***C*** 中，只要用 ***malloc*** 分配的内存没有用 ***free*** 释放就会产生内存泄露。在 ***C***++中肇事者的名字换成了 ***new*** 和 ***delete***，但是问题依然存在。当然，有了析构函数情况稍有改观。因为析构函数为所有将被销毁的对象提供了一个方便的调用 ***delete*** 的场所，但这同时又带来了更多的烦恼，因为 ***new*** 和 ***delete*** 是隐式地调用构造函数和析构函数的。而且可以在类中和类外自定义 ***new*** 和 ***delete*** 操作符，这又带来了复杂性，增加出错的机会。

2. 内存分配方式

内存分配有三种方式：

- 从静态存储区域分配。内存在编译时就已经分配好，这块内存在程序的整个运行期间都存在。例如全局变量、***static*** 变量；
- 从栈上分配。在执行函数时，函数内的局部变量的存储单元都能在栈上创建，函数执行结束时，这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配内存容量有限；
- 从堆上分配，亦称动态内存分配。程序在运行时用 ***malloc*** 或 ***new*** 申请任意多少的内存，程序员自己负责在用完时使用 ***free*** 或 ***delete*** 来释放内存。动态内存的生存期由我们决定，使用起来很灵活，但问题也最多。

二、***Open Cascade*** 中的内存管理 ***Memory Management in Open Cascade***

在几何建模的过程中，程序创建和删除了大量的对象在动态内存中，也就是堆中。在这种情况下，标准 ***C***++的内存管理方式不是很高效，所以 ***Open Cascade*** 在包 ***Standard*** 中专门写了个内存管理程序（***Memory Manager***）来对内存的分配与删除进行管理。

1. 用法 ***Usage***

为了在 ***C*** 代码中使用 ***Open Cascade*** 提供的内存管理器，只需要将原来使用 ***malloc*** 的地方使用 ***Standard***::***Allocate*** 来代替，原来使用 ***free*** 的地方使用 ***Standard***::***Free*** 来代替。另外，原来使用 ***realloc*** 的地方使用 ***Standard***::***Reallocate*** 来代替即可。

在 ***C***++中，***operator new*** 和 ***delete*** 都重新定义以便使用 ***Open Cascade*** 的内存管理器。

定义代码如下所示：

```
public:
  // Redefined operators new and delete ensure that handles are
  // allocated using OCC memory manager
  void* operator new(size_t,void* anAddress)
  {
    return anAddress;
  }
  void* operator new(size_t size)
  {
    return Standard::Allocate(size);
  }
  void  operator delete(void *anAddress, size_t )
  {
    if (anAddress) Standard::Free(anAddress);
  }
```

上述代码是将 ***operator new*** 和 ***delete*** 及 ***placement new*** 都重新定义了，这样的类的 ***new*** 和 ***delete*** 都将由 ***Open Cascade*** 的内存管理器来管理。

CDL extractor 为在其中所有类都采用这种方式来重新定义 ***operator new*** 和 ***delete***，这样 ***Open Cascade*** 所有的类（少数除外）都是使用 ***Open Cascade*** 的内存管理器来管理。

2. 配置内存管理器 ***Configuring memory manager***

Open CASCADE 内存管理器可以配置，按不同的优化方式来分配内存，主要还是看需要分配内存的大小，或者不使用内存优化而直接使用 ***malloc*** 和 ***free***。

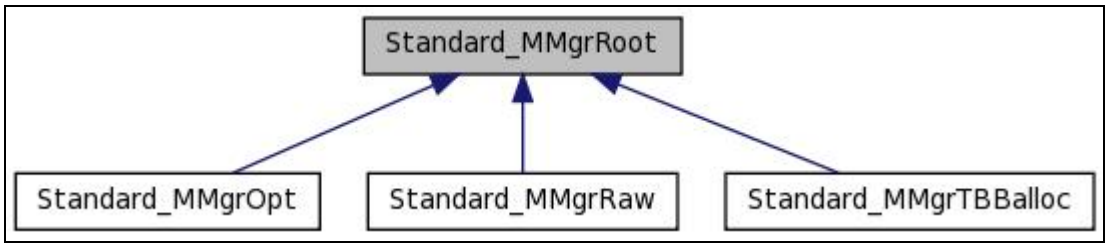
配置方式为设置如下环境变量的值：

- ***MMGT_OPT***：若设置为 ***1***（默认值也是为 ***1***），内存管理器将使用内存优化的方式来管理内存；若设置为 ***0***，则内存的分配就是直接调用 ***C*** 的函数 ***malloc*** 和 ***free*** 来对内存进行管理，此时，所有其它选项除了 ***MMGT_CLEAR*** 外都将被忽略。若设置为 ***2***，则会使用 ***Intel*** 的 ***TBB*** 来对内存的分配进行优化，此时需要有 ***TBB*** 的库。
- ***MMGT_CLEAR***：若设置为 ***1***（默认值也是为 ***1***），分配的内存块将被清零；若设置为 ***0***，则内存块将以分配时的值返回。
- ***MMGT_CELLSIZE***：定义了内存池中可分配内存块的最大值。默认值为 ***200***。
- ***MMGT_NBPAGES***：定义了页面上可分配的小的内存块的数量，默认值为 ***1000***。
- ***MMGT_THRESHOLD***：定义了循环利用的而不是返回给堆的内存块的数量，默认值为 ***4000***。
- ***MMGT_MMAP***：若设置为 ***1***（默认值也是为 ***1***），大内存块的分配将会使用操作系统的内存映射函数。若设置为 ***0***，内存的分配将会直接使用 ***malloc*** 直接在堆上分配。
- ***MMGT_REENTRANT***：若设置为 ***1***（默认值为 ***0***），所有调用内存优化的函数将会被保证安全，即使有多个不同的线程。当在使用内存优化管理（***MMGT_OPT=1***）内存及多线程的程序时，这个值需要设置为 ***1***。

注：为了使用 ***Open Cascade*** 在多线程的程序中表现出更好的性能，推荐如下两种设置方式：

- ***MMGT_OPT=0***
- ***MMGT_OPT=1 and MMGT_REENTRANT=1***

3. 程序实现 ***Implementation details***



类 ***Standard_MMgrRoot*** 为内存管理器的抽象类，它定义了内存分配的释放的虚函数。通过环境变量 ***MMGT_OPT*** 来选择不同的内存管理类，如下代码所示：

```
Standard_MMgrFactory::Standard_MMgrFactory() : myFMMgr(0)
{
  char *var;
  Standard_Boolean bClear, bMMap, bReentrant;
  Standard_Integer aCellSize, aNbPages, aThreshold, bOptAlloc;
  //
  bOptAlloc   = atoi((var = getenv("MMGT_OPT"      )) ? var : "1"    );
  bClear      = atoi((var = getenv("MMGT_CLEAR"    )) ? var : "1"    );
  bMMap       = atoi((var = getenv("MMGT_MMAP"     )) ? var : "1"    );
  aCellSize   = atoi((var = getenv("MMGT_CELLSIZE" )) ? var : "200"  );
  aNbPages    = atoi((var = getenv("MMGT_NBPAGES"  )) ? var : "1000" );
  aThreshold  = atoi((var = getenv("MMGT_THRESHOLD")) ? var : "40000");
  bReentrant  = atoi((var = getenv("MMGT_REENTRANT")) ? var : "0"    );

  if ( bOptAlloc == 1 ) {
    myFMMgr = new Standard_MMgrOpt(bClear, bMMap, aCellSize, aNbPages,
                                   aThreshold, bReentrant);
  }
  else if ( bOptAlloc == 2 ) {
    myFMMgr = new Standard_MMgrTBBalloc(bClear);
  }
  else {
    myFMMgr = new Standard_MMgrRaw(bClear);
  }
  // Set grobal reentrant flag according to MMGT_REENTRANT environment variable
  if ( ! Standard_IsReentrant )
    Standard_IsReentrant = bReentrant;
}
```

当 ***MMGT_OPT*** 设置为 ***1*** 时，将会使用类 ***Standard_MMgrOpt*** 来对内存的分配与释放进行优化。优化方法如下：

- 小型内存块（小于 ***MMGT_CELLSIZE*** 的内存）不是单独分配。而是分配一个大的内存池（每个内存池的大小是 ***MMGT_NBPAGES***），每个新建内存都被安排在当前的内存池中空闲的地方。若当前内存池被占满，则分配另一个内存池。在当前的版本中，内存池不会返回给系统（直到程序结束）。然而，调用函数 ***Standard***::***Free***()被释放的内存块会被 ***free*** 列表记录，以便在下一个相同大小的内存块分配时重新利用（循环使用）。

- 中型内存块（大小在 ***MMGT_CELLSIZE*** 和 ***MMGT_THRESHOLD*** 之间的内存块）由 ***C*** 的函数 ***malloc*** 和 ***free*** 直接管理。当这样的内存块被调用函数 ***Standard*::*Free*** 释放时，它们也像小型内存块那样被循环使用。与小型内存块不同的是，被释放的 ***free*** 列表中包含的中型内存块可以通过函数 ***Standard*::*Purge***，使其返回到堆中。
- 大型内存块（大于 ***MMGT_THRESHOLD*** 的内存块，包含用于管理小型内存块的内存池）的分配取决于 ***MMGT_MMAP*** 的值：若为 ***0***，这些内存块在堆中分配；否则，将会使用操作系统的专用的管理内存映射文件的函数来分配。当使用 ***Standard*::*Free*** 来释放大型内存块时，大型内存块立即返回给系统。

4. 利与弊 ***Benefits and drawbacks***

Open Cascade 使用内存管理器的最大好处就是其对小型内存块的循环使用机制。当程序需要对大量小型内存块进行分配与释放时，这种机制使程序速度更快。实践表明，使用这种方式程序的性能可以提高 ***50***%以上。

相应的弊端就是循环使的内存在程序运行时不会返回给系统。这就可能导致大量的内存消耗，甚至可能导致内存泄露。为了避免这种情况，应该在大量使内存的操作结束后调用函数 ***Standard*::*Purge***。

使用 ***Open Cascade*** 的内存管理器（***Memory Manager***）导致的所有的内存开销有：

- 分配的每个内存块的大小都会以 ***8*** 个字节向上取整。（看其源代码应该是以的个字节向上取整，源程序如下所示：）

```
Standard_Address Standard_MMgrRaw::Allocate(const Standard_Size aSize)
{
  // the size is rounded up to 4 since some OCC classes
  // (e.g. TCollection_AsciiString) assume memory to be double word-aligned
  const Standard_Size aRoundSize = (aSize + 3) & ~0x3;
  // we use ?: operator instead of if() since it is faster :-)
  Standard_Address aPtr = ( myClear ? calloc(aRoundSize, sizeof(char)) :
                                      malloc(aRoundSize) );
  if ( ! aPtr )
    Standard_OutOfMemory::Raise("Standard_MMgrRaw::Allocate(): malloc failed");
  return aPtr;
}
```

- 额外的 ***4*** 个字节（在 ***64*** 位的操作系统上是 ***8*** 个字节）将在每个内存块的开始部分分配，用来保存其大小（或用来保存下一个可用的内存块的地址），只在 ***MMGT_OPT*** 为 ***1*** 时有效。

所以不管 ***Open Cascade*** 的内存管理器以优化方式还是标准方式来管理内存，内存总的消耗都将会大一些。