For:pubsdev
Printed on:Mon, Nov 18, 2002 18:18:00
From book:appdev
Document:05Mem
Last saved on:Mon, Nov 18, 2002 18:16:58

Chapter 5.

Allocating and Deallocating Memory within ACIS

Topic: *Memory Management

Allocating heap memory and deallocating it properly is critical to any application. Within ACIS, this is especially true for a number of reasons:

- There are a number of platforms which manage heap memory differently. When allocations/deallocations are done incorrectly, some platforms crash, some platforms leak, and others proceed silently.
- ACIS has the ability to manage small blocks of heap memory – which we call freelisting. Freelisting is a heap management algorithm that provides a substantial performance increase – and is transparent to the end user. If the ACIS allocation/deallocation rules are not followed, a block might taken from the freelist and freed by the OS, or vice versa, which will almost always result in a crash.
- ACIS also has the ability to track all heap allocations and deletions. If the ACIS allocation/deallocation rules are not followed, some allocations show up as memory leaks – when they simply aren't being freed properly.

ACIS rules for heap allocations and deletions

Topic: *Memory Management

Rule 1 : Never allocate memory in static–initialization.

This fundamental rule must not be violated. ACIS provides the ability for the customer to modify the run–time behavior of the ACIS memory management system. That is, the customer can decide, at run–time, to enable freelisting or enable leak–tracking. Once initialize_base is called with a particular memory management configuration, that memory management configuration is set for the duration of the life of the application – even through the end of static–termination.

For example, suppose an ACIS_NEW takes place in static–initialization and the memory is allocated from the ACIS freelist. Then suppose the customer application calls initialize_base and disables freelisting. When the matching ACIS_DELETE is called, the pointer will be freed by the system, not by the ACIS freelisting mechanism. The system will most likely crash because it is trying to delete a pointer of which it has no knowledge.

Allocating memory in static initialization will most likely result in a crash.

Rule 2 : Use STD_CAST appropriately.

If ACIS_NEW is used to allocate a simple data–type, enum, or struct, or an array of simple data–types, enums, or structs then STD_CAST must be used while calling ACIS_DELETE. STD_CAST must not be used while using ACIS_DELETE for deleting a C++ object or array of objects.

Examples:

```
SPAposition *p = ACIS_NEW SPAposition;
ACIS_DELETE p;

SPAposition *p = ACIS_NEW SPAposition[24];
ACIS_DELETE [] p;

SPAposition **p = ACIS_NEW SPAposition *;
ACIS_DELETE STD_CAST p;

SPAposition **p = ACIS_NEW SPAposition *[24];
ACIS_DELETE [] STD_CAST p;
```

Failing to use STD_CAST properly will most likely result in a false leak, but on some systems (e.g., Mac) they will result in a crash.

Rule 3 : Use the array operator [] appropriately when deleting.

This is a rule of C++ programming. While allocating an array of elements using *new*/ACIS_NEW, the array operator must be used while calling *delete*/ACIS_DELETE.

Examples:

```
int *n = new int[6];
delete [] n;

my_object *mo = new my_object[10];
delete [] mo;
```

Failing to use the array operator when deleting may result in memory leaks and/or resource leaks, and in some cases, crashes.

Why is STD_CAST needed?

Topic: *Memory Management

ANSI C++ permits overloading global *new* operator and even provides an overloaded global *new* operator that has parameters (*placement new*). ACIS leverages this feature to send file and line information with every call to ACIS_NEW that allows ACIS to track memory leaks.

ACIS provides a version of ACIS_NEW which overloads global new to send along file and line information. If the class is derived from ACIS_OBJECT, ACIS provides a class–level overloaded new to distinguish between global new's and class level new's behavior. This is an important distinction: these two allocations follow preprocess into completely different function calls.

```
//
// This line would preprocess into the following:
//
//
//   BODY *b = BODY::operator new (size, __FILE__, __LINE__, etc.) BODY;
//
BODY *b = ACIS_NEW BODY;
```

```
//
// This line would preprocess into the following:
//
//   int *n = operator new (size, __FILE__, __LINE__, etc.) int;
//
int *n = ACIS_NEW int;
```

While ACIS_NEW converts into an overloaded *new* with file and line information, ACIS_DELETE does something different. If the class is derived from ACIS_OBJECT, ACIS_DELETE converts into a class level overloaded delete. For simple data types, ACIS_DELETE converts into global *delete*. The last statement requires a critical understanding: Deleting a simple data type using ACIS_DELETE is exactly the same as calling global delete.

A simple case.

```
//
// This line would preprocess into the following:
//
//
//   BODY::operator delete (ptr, etc.) b;
//
ACIS_DELETE b;
```

A complex case that demonstrates the problem with not using STD_CAST.

```
//
// This line would preprocess into the following:
//
//
//   delete n;         // this will show up as a false leak!!
//
ACIS_DELETE n;
```

Since ACIS doesn't (and should not) overload global delete, this will show up as a leak, even though the memory is being given back to the system.

The correct version:

```
//
// This line would preprocess into the following:
//
//
//   ACIS_STD_TYPE_OBJECT::operator delete n;
//
ACIS_DELETE STD_CAST n;
```

The ACIS_STD_TYPE_OBJECT is defined as follows:

```
class DECL_BASE ACIS_STD_TYPE_OBJECT {
public:
     void operator delete(void * alloc_ptr ) {
          acis_discard( alloc_ptr, eClassStd, 0 );
     }

     void operator delete [](void * alloc_ptr ) {
          acis_discard( alloc_ptr, eClassArrayStd, 0 );
     }
};
```

Since the pointer is casted to an object that has an overloaded delete, the ACIS_DELETE call is redirected to the ACIS memory manager and handled properly.