The Dynamics of Changing Dynamic Memory Allocation in a Large-Scale C++ Application

Neil B. Harrison
Utah Valley State College
800 W. University Parkway
Orem, UT, 84058
801-863-7312
harrisne@uvsc.edu

John H. Meiners
Avaya, Inc.
1300 W. 120th Ave.
Westminster, CO, 80234
303-538-4436
jmeiners@avaya.com

Abstract
Changing the approach to dynamic memory allocation in a large legacy application is challenging. In order to improve the robustness of memory allocation, we fundamentally changed it. We replaced standard heap allocation with class-specific heaps. We were able to do it with almost no changes to the existing class code by overriding the C++ new() and delete() operators, and using templates creatively to insert the changes into class hierarchies.

The results have been very positive. Misuse of dynamic memory has been detected and errors caused by memory misuse have been avoided. Performance of the new memory management code has been as good or better as the previous code. Additional capabilities such as audits have been added to further increase the robustness of dynamic memory usage.

Categories and Subject Descriptors D.3.3 [**Programming Languages**]: Language Contructs and Features – *Dynamic storage management*.
D.3.3 [**Programming Languages**]: Language Contructs and Features – *Data types and structures.*

D.3.3 [**Programming Languages**]: Language Contructs and Features – *Patterns*.

General Terms Reliability

Keywords Memory management, templates, Curiously Recurring Template Pattern.

1. Introduction
We were concerned about our project. For many years, our business communication system had been a leader in the marketplace. Our system was particularly respected for its high reliability and availability, and many large customers had come to rely on it. However, cracks were beginning to appear in our foundation of reliability. The cracks were small, but potentially serious: once or twice, a customer's system crashed. Such crashes were very rare, but problematic to the customer: they lost service for a few seconds while the system recovered.

OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.
ACM 1-59593-491-X/06/0010.

Crashes are notoriously difficult to diagnose, but it became clear that they were associated with memory corruption; specifically improper use of dynamic memory. Instrumentation of our heap management software confirmed this hypothesis.

The cause of the crashes was rooted in history. The original system was written in C, long before the advent of C++. Because of the reliability needs of the system, all memory was statically allocated. This enhanced reliability, but at a cost of space: all tables had to be allocated to the maximum possible size. As the number of features and configurations multiplied, the static memory approach became impractical, and we began to allocate memory dynamically. This coincided with the rise of C++ and the use of polymorphism.

We were well aware that dynamic allocation of memory is fraught with danger, and we were careful to use dynamic memory carefully. But the system is large, complex, and stateful, and in spite of our best efforts, errors can creep in. We were particularly cognizant of potential memory leaks; but other memory errors can be related to freeing memory prematurely. The most serious are often accessing or modifying objects after they have been freed.

Conceptually, reliable management of dynamic objects is reasonably straightforward. One can localize ownership, as described in patterns of managing dynamic objects in C++ [3]. One can use smart pointers with reference counting [1][4]. Tools to detect illegal memory references and memory leaks are available. However, our application processed extremely complex states, in which state information was shared and passed among several resources (think call transfer scenarios,) which made localized ownership impossible. Reference counting was thwarted by the high complexity of the code in general. And no memory usage tools would run on a system as large and complex as ours – we tried.

2. Considerations
We fixed the memory problem that caused the crashes, but wanted to take proactive steps to provide more protection of dynamic memory usage. We considered several options to prevent memory problems. We considered replacing all dynamic memory accesses
with static memory access. This was soon dismissed as unrealistic: because of system requirements and because of legacy code, dynamic memory was here to stay. Eventually it became clear that only one option was viable. We had to replace the accesses to the heap with memory accesses that preserved the dynamic memory usage, but prevented the problems associated

with memory allocation from the heap. In essence, we had to make heap memory allocation safe – preferably as safe as static or automatic memory allocation.

The overall goal was to eliminate all serious system problems related to dynamic memory usage. Specifically, this meant that the system must continue to run properly even if memory was misused. In particular, if an object was deleted twice, or used after it was freed, or if memory leaked, the system must continue to operate.

There were additional constraints as well. In particular, real time performance was important. Memory usage was a key issue. Systems ran on specialized hardware, with hard memory limits. Memory was adequate, but limited. This meant that solutions to the memory problem could not afford to waste significant amounts of memory. This was a significant issue in our low cost small configurations.

In addition, the solution had to be implemented within the current release schedule. The schedule allowed a few months for design, implementation, and testing, but there were over 100 different classes that allocated objects dynamically. Naturally, this demanded a generic solution. Looking forward, the solution had to be easy to implement, as other programmers would invariably be creating new classes with dynamic instantiation.

3. The Solution

3.1 Overall Approach

The most serious problems, both in terms of consequences and in difficulty to diagnose were use of so-called "stale pointers;" accesses to memory after it had been returned to the heap. If the program attempted to use a pointer to memory that had been freed and then allocated to some other place in the program, the access might fail, particularly if the program expected to use a pointer in that memory – the other part of the program may have modified it to be non-pointer data. This could cause a crash, but at least the generated stack backtrace would lead one to the offending code. However, if the program did not read, but rather modified the data, the crash would happen somewhere in innocent code – the memory would become corrupted for no apparent reason. Such problems were nearly impossible to track down, but were even more difficult because of the fact that the behavior could change depending on what data was written, and how the memory was allocated. So it was hopeless to consider finding and fixing all the potential problems. Therefore, the only reasonable approach was to prevent such errors from having disastrous effects, and detect such errors as much as possible.

Note that many heap implementations, including ours, used some sort of a LIFO approach to managing the free memory list, generally for performance reasons. This maximized the likelihood that use of stale pointers would cause problems.

The first decision was to institute class-specific pools of memory. The rationale was that if writes to stale pointers happened, at least the "shape" of the memory would be the same, thus reducing the chance that pointers would turn into non-pointers. This could be done easily by overloading the new() and delete() operators.

Class-specific pools of memory allowed for a number of other significant benefits. Because objects in a pool were all the same size, it became easy to shift from a LIFO management of free resources to FIFO, thus greatly reducing the chances that using stale pointers would be accessing an object in use.

These approaches to memory management are not new; they are tried and true techniques. Our unique challenge was to retrofit them into a very large existing application, and to do it in the context of a normal development cycle. Among other things, the development base had to remain stable even if we added an incomplete solution to the base. To our knowledge, such a wholesale change to memory management had not been attempted before on such a large legacy system.

3.2 Classes and More Classes

Because each (base) class needed its own unique pool of memory from which to draw, it was not practical to replace the global new() and delete() functions – each class needed its own. Since inheritance and polymorphism was used extensively, we preferred to overload new() and delete() at the base class level. We also wanted to minimize the amount of code that had to be changed, since we had so many classes to change Therefore, we defined new and delete in a template class, and used the Curiously Recurring Template Pattern [2][5] to create a custom base class from which the class would inherit the new() and delete() functions.

The Curiously Recurring Template is as follows: A template is created, and a class inherits the template which is instantiated in terms of the class. Therefore, the template instantiation generates a unique base class from which the main class is derived. This allows capabilities to be added to a class hierarchy easily. Of course, in order for this to work, the template cannot instatiate the class, since it is not defined yet.

The use of the Curiously Recurring Template Pattern allowed us to override the new() and delete() functions of a class without surgery somewhere in the innards of the class declaration. It also meant that all classes derived from that classes picked up the overridden functions automatically.

Why could we not simply derive every class from a common base class that overrode new() and delete(). The problem was that each different class hierarchy needed its own separate pool of objects. The template contained a static object, a pointer to the memory pool controller (see below). Since each class hierarchy needed its unique memory pool controller object, they couldn't all derive from a common class. The inherited template provided the uniqueness. Interestingly, the template did not need the class name internally; it was simply used to generate different classes. This is not uncommon in uses of the Curiously Recurring Template Pattern.

The template looked something like this (simplified view):

```
template <class T>
class MemPool
{
public:
```

```
    static void* operator new(size_t t)
       { return mptr->getObj(t); }
    static void operator delete (void* ptr)
       { mptr->returnObj(ptr); }
private:
    static MemPoolCntl* mptr;
};
```

Instantiation of the template was something like this:

```
class MyClass : public MemPool<MyClass>
{
       …
};
```

As templates generate source code, and there were many classes, we made the template itself as small as possible. In fact, the template consisted of a pointer to a memory pool controller object whose methods contained all the memory control logic. The template had a handful of single-line functions that invoked the memory controller's methods. As a result, the code growth was minimal.

3.3 Base Classes and Derived Classes.

Overloading new() and delete() meant that publicly derived classes got the benefit of the allocation without adding the template at each derived class. In fact, instantiating the template at both a base class level and a derived class level produced a compile error. In nearly all cases, it was exactly what we needed. However in one or two cases, a particular derived class was much larger than any of the other derived classes (sometimes by thousands of bytes). Since the pool must consist of objects large enough to hold the largest derived class, this would have resulted in significant memory wastage. In those cases, we overloaded new() and delete() directly, without using the template. We had to put the contents of the template into the desired derived class by hand, but once done, it worked perfectly. With the expectation that this might be done again on rare occasions, we created a macro that generated the code that paralleled the code in the template. The macro was placed with care inside the class definition in order to overload new() and delete(), and provide a pointer to a MemPool memory manager object.

3.3.1 Memory Usage

As indicated before, memory was limited. Every class has an expected maximum number of objects that would be allocated at any one time, based on the runtime configuration of the system. There was enough memory for objects in use for any one configuration, but there were tradeoffs in configurations. There was not enough memory for all objects to have maximum allocation at the same time. Therefore, we could not afford to have static tables of the maximum free objects for each type; the allocation must by dynamic.

In order for class specific memory to work, memory could never be shared among classes – memory must belong to a specific class. MemPool approached these conflicting requirements by reserving blocks of objects all at once. Once it reserved such a block, it was never returned to the general heap. The size of block was set by the implementer on a per class basis. (There was a slight performance penalty for many small blocks, so the block size was a time versus space tradeoff).

Among different customers, the number of objects needed of different classes varied widely, but within one system, the pattern of class usage is very stable over time. No one customer needs the maximum number of objects of all classes. As usage of objects of a class grew, sufficient blocks would be allocated. Over time, the object usage pattern for that system was reached, and no new blocks for any class were needed. In this way, we supported different configurations dynamically with a limited amount of memory.

3.4 Error Detection and Avoidance

We took a double-pronged approach to error detection. First and foremost, we tried to prevent errors that might occur from causing any harm. In these cases, we generated error messages, since they represented errors in the application code that should be cleaned up. Second, where this was impossible, we alerted the user that a problem had occurred that might bring down the system later. Whenever we created an error message, we also captured a stack backtrace, which programmers could use to help determine the source of the error.

3.4.1 Double Deletes

To prevent double deletes from causing problems, we stored free objects in a FIFO free list. Each class' free list had a tunable (at compile time) minimum size, guaranteeing that freed objects would stay in the FIFO for a certain amount of time. The rationale was that when an object was freed, subsequent uses, including deletes, would be most likely to occur shortly afterwards, rather than a long time later. Our experience bore this out.

When a delete happened, it was easy to check whether the object was on the free list. We put out an error message, and allowed processing to continue normally.

3.4.2 Writing to Freed Memory

Prevention followed the same model as for double deletes, using a free list FIFO with a minimum size. Other steps were also employed. When an object was freed, a checksum of the object was generated. When an object was subsequently allocated, the checksum was re-generated, and compared to the original. If the two did not agree, the object had been modified. We created an error message, and then put the object at the back of the free list. The rationale was that if it had been modified, it might be modified again, so don't use it if possible. Instead, select the next object on the free list; presumably one that had not been modified. The code detected the extremely unlikely event that all objects on the free list had been corrupted, and in that case, put out a more serious error message, and selected one of the objects.

3.4.3 Accessing a Freed Object

There was no way to detect that a freed object was being accessed for reading without instrumenting the memory in a way that detected all reads. This would have compromised the real time properties of the system. Instead, we relied on the FIFO with a buffer to maximize the time that an object would be on the free

list. The rationale was that an access of an object after it is freed is most likely to occur soon after the object has been returned to the free list. Our experience tended to corroborate this hypothesis.

3.4.4 Memory Leaks

Without expensive instrumentation, it is impossible to detect memory leaks as they happen. So we adopted a compromise that served us well. Each class' memory pool was declared to have a maximum size (in number of objects.) When the number of active objects exceeded half the limit, an error message was output. The message contained information about the number of objects allocated, as well as the maximum. A memory leak would produce scores of such messages, which were useful to predict how long the system could continue to run before that class' memory pool was exhausted. These messages' stack backtrace could not point to the missing delete (obviously), but did show where the allocations were happening.

This was making the best of a difficult situation. We checked all places where objects were dynamically created to ensure that they checked for the success of the allocation; however, the system would certainly function in degraded mode at that point. The early warning would give people a chance to reset the system at a convenient time.

3.4.5 Exceptions

It might be noted that in no case did we throw exceptions. There were several reasons for this. The main reason was that much of the system had been created before exceptions became part of C++, so exceptions had never been part of the error detection and handling strategy.

3.4.6 Visibility

The system had an interface for accessing certain internal information. It was valuable to see the state of memory allocation for all classes, so functions were written to present the current state of each memory pool, including the number of active and free objects, the total number of allocations that had been done, and the total number of memory errors that had been detected.

When each memory pool was instantiated, it registered with a static object, providing a pointer to itself. This object would retrieve the current information from each memory pool whenever requested.

Part of the information displayed was the name of the class for each memory pool. It turned out that it was easy to use the memory pool's instantiation macro to populate the class name – a sort of poor man's reflection. It meant that implementers had to do nothing special for their class' data to show up; it all happened automatically.

3.5 Implementation Joys

Implementation of the system proved to be easier than anticipated. It consisted of two parts. The first was to implement and thoroughly test the memory allocation software. The second was to implement it for all the classes in the system.

In order to test the memory allocation software itself, we wrote a template function that did normal allocations and deallocations, and exercised all the error paths as well. It attempted to free an object twice. It wrote to an object after it had been freed, and verified that the write was detected, that the object was returned to the tail of the free list, and that a subsequent allocation of that object was successful. It wrote to all the objects on the free list, and verified that this condition was alerted. It attempted to delete a bad address, and a misaligned address. Both were detected and alerted. It simulated a memory leak by allocating all the objects allowed, and one beyond it. It attempted to allocate an object larger than the largest derived class in the class hierarchy. As this was a template function, it was delivered to the official code base, where it could be instantiated as needed.

Once the allocation software was tested and added to the official base, implementation for each class proceeded. For each base class, the class definition line of code was changed, and two initialization lines of code were added. No changes were required for most derived classes. Testing was accomplished by executing the system with normal activity, and checking the object allocation activity through the visibility interface. The interface showed all object allocation activity, including the total number of allocation errors encountered. With this, it was easy to verify correct behavior. Overall, it took a short time to implement memory management for all classes – calendar time was a few weeks, but since developers were engaged in other development, actual time was a matter of days.

3.6 Non-Object Allocations

After the implementation of the MemPool template for all classes that allocated objects dynamically, our instrumentation showed that there was still memory on the heap. Investigation showed that a purchased library, written in C, did nearly all the allocations. We wanted to extend the MemPool benefits to these allocations as well. So we created five dummy classes, representing generic objects of different sizes: 8 bytes, 16, bytes, 32 bytes, 64 bytes, and 128 bytes. Then we intercepted calls to malloc(), and based on the size of the malloc() request, returned the appropriate object.

Unlike the case for classes, we could not determine the maximum number of these "objects" that would be needed. We did determine empirically what the maximum sizes should be, but in case individual usage of the system caused these limits to be breached, we changed the algorithm slightly. If the limit for any of these dummy classes was exceeded, no errors were reported. Instead, the system would revert to having malloc() allocate memory. As usage dropped back below the threshold, MemPool would resume allocating the memory. In practice, we did not see this ever happen. Overall, heap usage dropped to basically nothing.

3.7 Enhancements

After the initial release, we added more capabilities to the memory pool system. These were designed to provide additional reliability in various aspects of memory allocation.

We required that all classes that were dynamically instantiated use the MemPool template. The template made it easy to see which classes did use it, but it was not immediately clear if a class that did not use MemPool did instantiate dynamic objects. Furthermore, if a person wrote a class that dynamically instantiated a class that had heretofore been only statically or

automatically allocated, that class should now use the MemPool template. A simple approach was devised to prevent unauthorized dynamic allocation of objects.

We wrote a template similar to the MemPool template, called NoNew. It defined new() and delete() as private. Any class using the NoNew template could not be dynamically instantiated. Every class was required to use either the MemPool or the NoNew template, or be derived from a class that used one or the other. This made it easy to verify in code inspections that classes allocated memory safely.

A second enhancement was added to detect writes outside of allocated buffers. Four bytes were reserved both before and after each piece of memory. Those bytes were written with bit patterns. When the memory was returned, the bit patterns were checked to see if they had been modified. This provided an added measure of safety, as well as detection after the fact. This could be turned off or on for each class. It was turned on only for the specific size byte buffers, since these allocations were usually made for message buffers which could be of variable sizes. This was purely preventative, since we had never seen such problems; the enhancement did not detect any problems.

A capability to audit the state of objects was added. This was mainly to detect memory leaks, and clean them up before they caused problems. It used a mark and sweep approach: at the beginning of an audit, the application code would notify the MemPool of each object it considered active. It was informed if MemPool considered any of those objects to be free. It was also informed if previously marked objects were marked. At the end of the marking phase, MemPool would sweep the free list and detect any objects that were on its active list but were unmarked, and free them. It would simply put them on its free list, rather than call the object's destructor (because of possible side effects of doing so).

In order to use the audit, the application code had to have a way of accessing all its active objects. In most cases, this was reasonably straightforward, though some classes stored pointers to objects in multiple places. This necessitated the capability to mark objects in such a way that duplicate marks were not considered errors.

The most significant challenge of the audit code was that the audits ran at low priority, so the audit code had to be interruptible. During an audit hiatus, objects could be allocated or freed; the audit code had to account for such changes.

4. Results

Reliability tends to be a negative trait: it can best be measured by the lack of anything bad happening. Memory usage reliability is this way, though the MemPool did give indication of any problems it detected and corrected. It also gave memory usage statistics that gave some indication of flawless activity.

Shortly after the introduction of MemPool, before the system was released, it detected two or three illegal uses of memory. Since it pointed to the class and the line of code where the problem was detected, the problems were readily fixed.

Once the software was released into the field, there were no memory-related crashes. The MemPool system has been in place for over two years, over more than two releases, and we have had no problems. Early on, MemPool detected two memory problems: The first was that MemPool detected writes to objects after they had been deleted. In fact, it reported that every object on the free list had been corrupted. Investigation showed that a state member variable of the objects was changed immediately after the object was deleted – the way it was done turned out not to cause problems, but a very dangerous practice nonetheless. Interestingly, the developer's initial reaction was that the MemPool software must be broken, even though it reported problems with only that single class. Object dumps provided by MemPool vindicated it.

The second problem was a memory leak. MemPool detected the memory leak, and isolated it to the offending class. This enabled the developers to quickly find and fix the problem. In the meantime, the customer was able to restart the system at a convenient time, thus preventing crashes.

On an inhouse system, a problem was detected with calls to malloc. Since the calls went through the specific size byte buffers, we could detect the size of the offending buffers. This helped track down and fix the problem.

Perhaps the most significant positive effect was that MemPool detected several problems with new code shortly after the new code was delivered to the base. This meant not only that such problems were detected long before the system was delivered to customers, but that any such problem was known to be in the new code. This made the problems easy to find and fix.

Since the release of the system to customers, we have had no complaints about crashes related to memory (mis-)use. This lack of negative feedback indicates that the solution is very robust.

5. Performance

The system was a soft real-time system – stimuli had to be serviced within specified intervals. Speed was desirable, but consistency of performance was more critical than speed. Memory allocation through MemPool was sufficiently fast and consistent for our needs.

Allocation of memory through MemPool was very quick: it simply took the first element of the free list. Generating the checksum (see above) required a single pass through the object. Therefore, allocation time was constant, regardless of the number of objects active or free.

Freeing an object required searching through memory to find the right object. However, since the objects were housed in blocks, the time was a function of the number of such blocks. The user could set the number of blocks on a per class basis at compile time. In any case, object deletion was nearly as fast as allocation, even though performance showed very slight super-linear behavior as a function of the number of blocks.

All indications were that little little memory was wasted. We did not see an appreciable increase in memory usage; whatever memory was used by the MemPool approach was well within the amount originally allocated for the heap.

In summary, speed was as good or better than previous, and memory usage did not noticably increase.

6. Summary

In all, the effort to change the way we did dynamic memory allocation was a very strong success. We were able to complete the development more quickly than anticipated, with high quality and reliability. The results indicate that we are now able to detect dynamic memory problems, prevent them from adversely affecting system reliability, and provide information useful to developers for debugging such problems. And we did it on a large legacy system.

7. References

[1] Alexandrescu, A. Modern C++ Design, Addison-Wesley, Reading, MA, 2001, 157-196.

[2] Coplien, J. A Curiously Recurring Template Pattern. In Stanley B. Lippman, ed., *C++ Gems*, Cambridge University Press, New York, NY, 1996, 135-144

[3] Cargill, T. Localized Ownership: Managing Dynamic Objects in C++. In J. Vlissides et al, eds., *Pattern Languages of Program Design, vol 2,* Addison-Wesley, Reading, MA, 1995, 5-18.

[4] Murray, R. *C++ Strategies and Tactics.* Addison-Wesley, Reading, MA, 1993, 149-155.

[5] Vandevoorde, D. and N. Josuttis. *C++ Templates: The Complete Guide.* Addison-Wesley, Reading, MA, 2003, 295-298.