This article appeared C++ report *in two parts, in the July/August 1993 and December 1993 issues.*

Memory Management in C++

by **Nathan C. Myers**

Memory usage in C++ is as the sea come to land:
a tide rolls in, and sweeps out again,
leaving only puddles and stranded fish.
At intervals, a wave crashes ashore; but the
***ripples never cease**.*

Introduction

Many programs have little need for memory management; they use a fixed amount of memory, or simply consume it until they exit. The best that can be done for such programs is to stay out of their way. Other programs, including most C++ programs, are much less deterministic, and their performance can be profoundly affected by the memory management policy they run under. Unfortunately, the memory management facilities provided by many system vendors have failed to keep pace with growth in program size and dynamic memory usage.

Because C++ code is naturally organized by class, a common response to this failure is to overload member operator new for individual classes. In addition to being tedious to implement and maintain, however, this piece-meal approach can actually hurt performance in large systems. For example, applied to a tree-node class, it forces nodes of each tree to share pages with nodes of other (probably unrelated) trees, rather than with related data. Furthermore, it tends to fragment memory by keeping large, mostly empty blocks dedicated to each class. The result can be a quick new/delete cycle that accidentally causes virtual memory thrashing. At best, the approach interferes with system-wide tuning efforts.

Thus, while detailed knowledge of the memory usage patterns of individual classes can be helpful, it is best applied by tuning memory usage for a whole program or major subsystem. The first half of this article describes an interface which can ease such tuning in C++ programs. Before tuning a particular program, however, it pays to improve performance for all programs, by improving the global memory manager. The second half of this article covers the design of a global memory manager that is as fast and space-efficient as per-class allocators.

But raw speed and efficiency are only a beginning. A memory management library written in C++ can be an organizational tool in its own right. Even as we confront the traditional problems involving large data structures, progress in operating systems is yielding different *kinds* of memory -- shared memory, memory-mapped files, persistent storage -- which must be managed as well. With a common interface to all types of memory, most classes need not know the difference. This makes quite a contrast with systems of classes hard-wired to use only regular memory.

Global Operator New

In C++, the only way to organize memory management on a larger scale than the class is by overloading the global operator new. To select a memory management policy requires adding a *placement argument*, in this case a reference to a class which implements the policy:

```
  extern void* operator new(size_t, class Heap&);
```

When we overload the operator new in this way, we recognize that the regular operator new is implementing a policy of its own, and we would like to tune it as well. That is, it makes sense to offer the same choices for the regular operator new as for the placement version.

In fact, one cannot provide an interesting placement operator new without also replacing the regular operator new. The global operator delete can take no user parameters, so it must be able to tell what to do just by looking at the memory being freed. This means that the operator delete and all operators new must agree on a memory management architecture.

For example, if our global operators new were to be built on top of malloc(), we would need to store extra data in each block so that the global operator delete would know what to do with it. Adding a word of overhead for each object to malloc()'s own overhead (a total of 16 bytes, on most RISCs), would seem a crazy way to improve memory management. Fortunately, all this space overhead can be eliminated by bypassing malloc(), as will be seen later.

The need to replace the global operators new and delete when adding a placement operator new has profound effects on memory management system design. It means that it is impossible to integrate different memory management architectures. Therefore, the top-level memory management architecture must be totally general, so that it can support any policy we might want to apply. Total generality, in turn, requires absolute simplicity.

An Interface

How simple can we get? Let us consider some declarations. Heap is an abstract class:

```
  class Heap {
   protected:
    virtual ~Heap();
   public:
    virtual void* allocate(size_t) = 0;
    static Heap& whatHeap(void*);
  };
```

(The static member function whatHeap(void*) is discussed later.) Heap's abstract interface is simple enough. Given a global Heap pointer, the regular global operator new can use it:

```
  extern Heap* __global_heap;

  inline void*
  operator new(size_t sz)
    { return ::__global_heap->allocate(sz); }
```

Inline dispatching makes it fast. It's general too; we can use the Heap interface to implement the placement operator new, providing access to any private heap:

```
  inline void*
  operator new(size_t size, Heap& heap
    { return heap.allocate(size); }
```

What kind of implementations might we define for the Heap interface? Of course the first must be a general purpose memory allocator, class HeapAny. (HeapAny is the memory manager described in detail in the second half of this article.) The global heap pointer, used by the regular operator new defined above, is initialized to refer to an instance of class HeapAny:

```
  extern class HeapAny __THE_global_heap;
  Heap* __global_heap = &__THE_global_heap;
```

Users, too, can instantiate class HeapAny to make a private heap:

```
  HeapAny& myheap = *new HeapAny;
```

and allocate storage from it, using the placement operator new:

```
  MyType* mine = new(myheap) MyType;
```

As promised, deletion is the same as always:

```
  delete mine;
```

Now we have the basis for a memory management architecture. It seems that all we need to do is provide an appropriate implementation of class Heap for any policy we might want. As usual, life is not so simple.

Complications

What happens if MyType's constructor itself needs to allocate memory? That memory should come from the same heap, too. We could pass a heap reference to the constructor:

```
  mine = new(myheap) MyType(myheap);
```

and store it in the object for use later, if needed. However, in practice this approach leads to a massive proliferation of Heap& arguments -- in constructors, in functions that call constructors, everywhere! -- which penetrates from the top of the system (where the heaps are managed) to the bottom (where they are used). Ultimately, almost every function needs a Heap& argument. Applied earnestly, the result can be horrendous. Even at best, such an approach makes it difficult to integrate other libraries into a system.

One way to reduce the proliferation of Heap arguments is to provide a function to call to discover what heap an object is on. That is the purpose of the the Heap::whatHeap() static member function. For example, here's a MyType member function that allocates some buffer storage:

```
  char* MyType::make_buffer()
  {
    Heap& aHeap = Heap::whatHeap(this);
    return new(aHeap) char[BUFSIZ];
  }
```

(If "this" points into the stack or static space, whatHeap() returns a reference to the default global heap.)

Another way to reduce Heap argument proliferation is to substitute a private heap to be used by the global operator new. Such a global resource calls for gingerly handling. Class HeapStackTop's constructor replaces the default heap with its argument, but retains the old default so it can be restored by the destructor:

```
  class HeapStackTop {
    Heap* old_;
   public:
    HeapStackTop(Heap& h);
    ~HeapStackTop();
  };
```

We might use this as follows:

```
  { HeapStackTop top = myheap;
    mine = new MyType;
  }
```

Now space for the MyType object, and any secondary store allocated by its constructor, comes from myheap. At the closing bracket, the destructor ~HeapStackTop() restores the previous default global heap. If one of MyType's member functions might later want to allocate more space from the same heap, it can use whatHeap(); or the constructor can save a pointer to the current global heap before returning.

Creating a HeapStackTop object is very clean way to install any global memory management mechanism: a HeapStackTop object created in main() quietly slips a new memory allocator under the whole program.

Some classes must allocate storage from the top-level global heap regardless of the current default. Any object can force itself to be allocated there by defining a member operator new, and can control where its secondary storage comes from by the same techniques described above.

With HeapStackTop, many classes need not know about Heap at all; this can make a big difference when integrating libraries from various sources. On the other hand, the meaning of Heap::whatHeap() (or a Heap& member or argument) is easier to grasp; it is clearer, and therefore safer. While neither approach is wholly satisfactory, a careful mix of the two can reduce the proliferation of Heap& arguments to a reasonable level.

Uses for Private Heaps

But what can private heaps do for us? We have hinted that improved locality of reference leads to better performance in a virtual memory environment, and that a uniform interface helps when using special types of memory.

One obvious use for private heaps is as a sort of poor man's garbage collection:

```
  Heap* myheap = new HeapTrash;
  ... // lots of calls to new(*myheap)
  delete myheap;
```

Instead of deleting objects, we discard the whole data structure at one throw. The approach is sometimes called "lifetime management". Since the destructors are never called, you must carefully control what kind of objects are put in the heap; it would be hazardous ever to install such a heap as the default (with HeapStackTop) because many classes, including iostream, allocate space at unpredictable times. Dangling pointers to objects in the deleted heap must be prevented, which can be tricky if any objects secretly share storage among themselves. Objects whose destructors do more than just delete other objects require special handling; the heap may need to maintain a registry of objects that require "finalization".

But private heaps have many other uses that don't violate C++ language semantics. Perhaps the quietest one is simply to get better performance than your vendor's malloc() offers. In many large systems, member operator new is defined for many classes just so they may call the global operator new less often. When the global operator new is fast enough, such code can be deleted, yielding easier maintenance, often with a net gain in performance from better locality and reduced fragmentation.

An idea that strikes many people is that a private heap could be written that is optimized to work well with a particular algorithm. Because it need not field requests from the rest of the program, it can concentrate on the needs of that algorithm. The simplest example is a heap that allocates objects of only one size; as we will see later, however, the default heap can be made fast enough that this is no great advantage. A mark/release mechanism is optimal in some contexts (such as parsing), if it can be used for only part of the associated data structure.

When shared memory is used for interprocess communication, it is usually allocated by the operating system in blocks larger than the objects that you want to share. For this case a heap that manages a shared memory region can offer the same benefits that regular operator new does for private memory. If the interface is the same as for non-shared memory, objects may not need to know they are in shared memory. Similarly, if you are constrained to implement your system on an architecture with a tiny address space, you may need to swap memory segments in and out. If a private heap knows how to handle these segments, objects that don't even know about swapping can be allocated in them.

In general, whenever a chunk of memory is to be carved up and made into various objects, a Heap-like interface is called for. If that interface is the same for the whole system, then other objects need not know where the chunk came from. As a result, objects written without the particular use in mind may safely be instantiated in very peculiar places.

In a multi-threaded program, the global operator new must carefully exclude other threads while it operates on its data structures. The time spent just getting and releasing the lock can itself become a bottleneck in some systems. If each thread is given a private heap which maintains a cache of memory available without locking, the threads need not synchronize except when the cache becomes empty (or too full). Of course, the operator delete must be able to accept blocks allocated by any thread, but it need not synchronize if the block being freed came from the heap owned by the thread that is releasing it.

A heap that remembers details about how, or when, objects in it were created can be very useful when implementing an object- oriented database or remote procedure call mechanism. A heap that segregates small objects by type can allow them to simulate virtual function behavior without the overhead of a virtual function table pointer in each object. A heap that zero-fills blocks on allocation can simplify constructors.

Programs can be instrumented to collect statistics about memory usage (or leakage) by substituting a specialized heap at various places in a program. Use of private heaps allows much finer granularity than the traditional approach of shadowing malloc() at link time.

In the remainder of this article we will explore how to implement HeapAny efficiently, so that malloc(), the global operator new(size_t), the global operator new(size_t, Heap&), and Heap::whatHeap(void*) can be built on it.

A Memory Manager in C++

An optimal memory manager has minimal overhead: space used is but fractionally larger than the total requested, and the new/delete cycle time is small and constant. Many factors work against achieving this optimum.

In many vendor libraries, memory used by the memory manager itself, for bookkeeping, can double the total space used. Fragmentation, where blocks are free but unavailable, can also multiply the space used. Space matters, even today, because virtual memory page faults slow down your program (indeed, your entire computer), and swap space limits can be exceeded just as can real memory.

A memory manager can also waste time in many ways. On allocation, a block of the right size must be found or made. If made, the remainder of the split block must be placed where it can be found. On deallocation, the freed block may need to be coalesced with any neighboring blocks, and the result must be placed where it can be found again. System calls to obtain raw memory can take longer than any other single operation; a page fault that results when idle memory is touched is just a hidden system call. All these operations take time, time spent not computing results.

The effects of wasteful memory management can be hard to see. Time spent thrashing the swap file doesn't show up on profiler output, and is hard to attribute to the responsible code. Often the problem is easily visible only when memory usage exceeds available swap space. Make no mistake: poor memory management can multiply your program's running time, or so bog down a machine that little else can run.

Before buying (or making your customers buy) more memory, it makes sense to see what can be done with a little code.

Principles

A memory manager project is an opportunity to apply principles of good design: Separate the common case from special cases, and make the common case fast and cheap, and other cases tolerable; make the user of a feature bear the cost of its use; use hints; reuse good ideas. [Lampson]

Before delving into detailed design, we must be clear about our goals. We want a memory manager that satisfies the following:

- Speed:
  It must be much faster than existing memory managers, especially for small objects. Performance should not suffer under common usage patterns, such as repeatedly allocating and freeing the same

block.

- Low overhead:
  The total size of headers and other wasted space must be a small percentage of total space used, even when all objects are tiny. Repeated allocation and deallocation of different sizes must not cause memory usage to grow without bound.
- Small working set:
  The number of pages touched by the memory manager in satisfying a request must be minimal, to avoid paging delays in virtual memory systems. Unused memory must be returned to the operating system periodically.
- Robustness:
  Erroneous programs must have difficulty corrupting the memory manager's data structures. Errors must be flagged as soon as possible, not allowed to accumulate. Out-of-memory events must be handled gracefully.
- Portability:
  The memory manager must adapt easily to different machines.
- Convenience:
  Users mustn't need to change code to use it.
- Flexibility:
  It must be easily customized for unusual needs, without imposing any additional overhead.

Techniques

Optimal memory managers would be common if they were easily built. They are scarce, so you can expect that a variety of subtle techniques are needed even to approach the optimum.

One such technique is to treat different request sizes differently. In most programs, small blocks are requested overwhelmingly more often than large blocks, so both time and space overhead for them is felt disproportionately.

Another technique results from noting that there are only a few different sizes possible for very small blocks, so that each such size may be handled separately. We can even afford to keep a vector of free block lists for those few sizes.

A third is to avoid system call overhead by requesting memory from the operating system in big chunks, and by not touching unused (and possibly paged-out) blocks unnecessarily. This means data structures consulted to find a block to allocate should be stored compactly, apart from the unused blocks they describe.

The final, and most important, technique is to exploit address arithmetic which, while not strictly portable according to language standards, works well on all modern flat-memory architectures. A pointer value can be treated as an integer, and bitwise logical operations may be used on it to yield a new pointer value. In particular, the low bits may be masked off to yield a pointer to a header structure that describes the block pointed to. In this way a block need not contain a pointer to that information. Furthermore, many blocks can share the same header, amortizing its overhead across all. (This technique is familiar in the LISP community, where it is known as "page-tagging".)

With so many goals, principles, and techniques to keep track of, it should be no surprise that there are plenty of pitfalls to avoid. They will be discussed later.

A Design

The first major feature of the design is suggested by the final two techniques above. We request memory from the operating system in units of a large power of two (e.g. 64K bytes) in size, and place them so they are aligned on such a boundary. We call these units "segments". Any address within the segment may have its low bits masked off, yielding a pointer to the segment header. We can treat this header as an instance of the abstract class HeapSegment:

```
class HeapSegment {
 public:
  virtual void free(void*) = 0;
  virtual void* realloc(void*) = 0;
  virtual Heap& owned_by(void*) = 0;
};
```

The second major feature of the design takes advantage of the small number of small-block sizes possible. A segment (with a header of class HeapPageseg) is split up into pages, where each page contains blocks of only one size. A vector of free lists, with one element for each size, allows instant access to a free block of the right size. Deallocation is just as quick; no coalescing is needed. Each page has just one header to record the size of the blocks it contains, and the owning heap. The page header is found by address arithmetic, just like the segment header. In this way, space overhead is limited to a few percent, even for the smallest blocks, and the time to allocate and deallocate the page is amortized over all usage of the blocks in the page.

For larger blocks, there are too many sizes to give each a segment; but such blocks may be packed adjacent to one another within a segment, to be coalesced with neighboring free blocks when freed. (We will call such blocks "spans", with a segment header of type HeapSpanseg.) Fragmentation, the proliferation of free blocks too small to use, is the chief danger in span segments, and there are several ways to limit it. Because the common case, small blocks, is handled separately, we have some breathing room: spans may have a large granularity, and we can afford to spend more time managing them. A balanced tree of available sizes is fast enough that we can use several searches to avoid creating tiny unusable spans. The tree can be stored compactly, apart from the free spans, to avoid touching them until they are actually used. Finally, aggressive coalescing helps reclaim small blocks and keep large blocks available.

Blocks too big to fit in a segment are allocated as a contiguous sequence of segments; the header of the first segment in the sequence is of class HeapHugeseg. Memory wasted in the last segment is much less than might be feared; any pages not touched are not even assigned by the operating system, so the average waste for huge blocks is only half a virtual-memory page.

Dispatching for deallocation is simple and quick:

```
void operator delete(void* ptr)
{
  long header = (long)ptr & MASK;
  ((HeapSegment*)header)->free(ptr);
}
```

HeapSegment::free() is a virtual function, so each segment type handles deallocation its own way. This allows different Heaps to coexist. If the freed pointer does not point to allocated memory, the program will most likely crash immediately. (This is a feature. Bugs that are allowed to accumulate are extremely difficult to track down.)

The classical C memory management functions, malloc(), calloc(), realloc(), and free() can be implemented on top of HeapAny just as was the global operator new. Only realloc() requires particular support.

The only remaining feature to implement is the function Heap::whatHeap(void* ptr). We cannot assume that ptr refers to heap storage; it may point into the stack, or static storage, or elsewhere. The solution is to keep a bitmap of allocated segments, one bit per segment. On most architectures this takes 2K words to cover the entire address space. If the pointer refers to a managed segment, HeapSegment::owned_by() reports the owning heap; if not, a reference to the default global heap may be returned instead. (In the LISP community, this technique is referred to as BBOP, or "big bag o' pages".)

Pitfalls

Where we depart from the principles of good design mentioned above, we must be careful to avoid the consequences. One example is when we allocate a page to hold a small block: we are investing the time to get that page on behalf of all the blocks that may be allocated in it. If the user frees the block immediately, and we free the page, then the user has paid to allocate and free a page just to use one block in it. In a loop, this could be much slower than expected. To avoid this kind of thrashing, we can add some hysteresis by keeping one empty page for a size if there are no other free blocks of that size. Similar heuristics may be used for other boundary cases.

Another pitfall results from a sad fact of life: programs have bugs. We can expect programs to try to free memory that was not allocated, or that has already been freed, and to clobber memory beyond the bounds of allocated blocks. The best a regular memory manager can do is to throw an exception as early as possible when it finds things amiss. Beyond that, it can try to keep its data structures out of harm's way, so that bugs will tend to clobber users' data and not the memory manager's. This makes debugging much easier.

Initialization, always a problem for libraries, is especially onerous for a portable memory manager. C++ offers no way to control the order in which libraries are initialized, but the memory manager must be available before anything else. The standard iostream library, with a similar problem, gets away by using some magic in its header file (at a sometimes intolerable cost in startup time) but we don't have even this option, because modules that use the global operator new are not obliged to include any header file. The fastest approach is to take advantage of any non-portable static initialization ordering available on the target architecture. (This is usually easy.) Failing that, we can check for initialization on each call to operator new or malloc(). A better portable solution depends on a standard for control of initialization order, which seems (alas!) unlikely to appear.

Measurements

Tests of vendor-supplied memory managers can yield surprising results. We know of a program that used 250 megabytes of swap space (before crashing when it ran out) when linked with the vendor-supplied library, but only a steady 50 megabytes after it was relinked with our memory manager. We have a simple presentation graphics animation program that uses less than half a megabyte of memory, but draws twice as fast when relinked.

Dividends

The benefits of careful design often go beyond the immediate goals. Indeed, unexpected results of this design include a global memory management interface which allows different memory managers to coexist. For most programs, though, the greatest benefit beyond better performance is that all the *ad hoc* apparatus intended to compensate for a poor memory manager may be ripped out. This leaves algorithms and data structures unobscured, and allows classes to be used in unanticipated ways.

Thanks to Paul McKenney and Jim Shur for their help in improving this

article.