Policy–Based Memory Allocation

Andrei Alexandrescu Emery Berger

October 11, 2005

One of the nice perks of doing hard work for next to no money, activity also known as being a graduate student, is that you get to rub shoulders with researchers working on interesting problems. I've had the pleasure of meeting in person Emery Berger and also Kathryn McKinley, two of the authors (with Benjamin Zorn) of "Composing High-Performance Memory Allocators" [6] paper that I found highly interesting for "I can't believe I didn't think of that" reasons. But let's start with the beginning.

One chapter of Modern C++ Design [2] is somewhat different than all others. Actually, *very* different. While the book is dedicated to creating elegant, flexible, extensible designs, one chapter stands out like a sore thumb by actually describing one fixed, monolithic, rigid, awkward piece of software: Loki's small object allocator. Somehow I thought that memory management is a low-level operation and as such it doesn't lend itself to the elegant exploits of policy–based design. At the same time, some of the compiler–provided `new`/`delete` implementations were pretty bad at dealing with small objects, so I felt there would be a need for something to fill the void. So I sat down and wrote down an implementation that has portability as its only merit; other than that, it's not particularly performant, interesting, or original. The main problem with Loki's allocator, however, does not lie in its implementation (which, by the way, has been entirely rewritten by Rich Sposato, to the end of greatly improving its performance), but in its *design*, which is not configurable nor extensible—just a few monolithic classes, much like what you'd expect in the bad ol' days.

However, about at the same time, Emery Berger and others were experimenting with a mixin–based memory allocator for C++ that is at the same time highly efficient (rivaling the best general–purpose allocators out there), highly customizable (rivaling the best special–purpose allocators too), *and* highly portable by virtue of its customizable design! As soon as I became acquainted with that research, I realized how the chapter on memory allocation in Modern C++ Design should have looked like.

But before we delve into the fascinating topic of configurable memory allocation, let's tie a few knots with past comments received via email.

1 Mailcontainer

I wrote the sidebar "Memory: More Than Just Any Resource" of the last Generic⟨Programming⟩ installment with the serene resignation with which Galileo Galilei must have said his famous "Eppur si muove." "And yet garbage is collectable," I mumbled into my three–days beard when submitting the article for publication. The expected outcome was that the opponents of garbage collection will exemplarily mortify me via an endless flow of angry emails; however, they must've chosen to boycott me, because I haven't gotten even one email of protest.

Ivan Godard sent me an email on the subject of Observers [3, 4]. That message is so smart, and had me look up words in the dictionary so often, that I've decided to quote it almost in entirety, at the risk of blowing up the size of this article:

> Your difficulties with attach and detach while in the middle of notification are canonical concurrency problems, and can be dealt with in the usual way. The (only rea-

sonable?) semantics for Observer is to see subject, observer and event as three independent and asynchronous processes, with a time stream localized at subject. That is, an observer cannot directly determine the time relation between its attempt to attach and the occurrence of a particular event because the event is not within the observer's horizon. Only the subject, which receives both events and requests to attach, can time-order these.

Consequently a putative observer will see some sequence of events which (in the subject's time frame) begins some time after the attempt to attach and ends some time after the attempt to detach. The sequence is guaranteed to be ordered (in subject time) and dense, but where it starts and where it ends is not known. Of course, if the observer is itself a source of events and the channel between observer and subject is order-preserving then the observer can enforce somewhat more stringent bounds on how far the observed sequence extends into the subject's past or subject's future by injecting an event just before the attach or just after the detach.

So much for relativity theory. In practice, this implements naturally by considering the attempt to attach and the attempt to detach as being themselves events. The subject enqueues an event (including an attach/detach) on an action list, and walks the list of observers applying each event in turn to each observer. Both observers and events are timestamped; a simple counter will do. Events are discarded when there are no observers with lesser stamps. The observer list is pre-seeded with an observer (the subject itself) that is watching for attach events and which adds the observer to the observer list. All observers are watching for their own detach events and remove themselves when the subject applies the corresponding detach event. [...]

Most of the difficulties you encountered appear to be a consequence of the assumption that attach and detach were somehow global and absolute, rather than just another event in the sequence of events encountered by the subject. Remove that mistaken idea and your invalidated-iterators problems disappear.

This view of attachment and detachment as simple events is very interesting. This generalization would, however, make it harder to define policies that lead to the simple implementations in the article. Let's admit, such simple solutions are widely used and have their place in spite of their risks and limitations. If anyone would like to embark on defining a policy-based design for such a "relativistic" Observer, and if the design does allow efficient simple implementations as well, I'd be highly interested. Do contact Ivan at `igodard@pacbell.net` with questions (unless you're him).

Brian Wood comments on the "Walking Down Memory Lane" article:

> While you introduce `auto_vector`, you didn't mention either of the `ptr_vector` implementations. Thorsten Ottosen has written an interesting article about `ptr_vector` in the October issue of Doctor Dobb's Journal. `auto_vector` and the `ptr_vector` by Ottosen are obviously closely related, but also have differences. Some mention to `ptr_vector` and the differences would have been helpful I think.
>
> Anyway, from my own experience, I agree with you that things like `auto_vector` and `ptr_vector` are often preferable to `vector<shared_ptr<T> >`.

Great, thanks Brian—and readership, consider yourself tipped on where to look for more material related to scoped memory management.

2 Memory Allocation: One Size Doesn't... you know

Today's general–purpose memory allocators have reached reasonable speed and low fragmentation for a large category of programs. However, in the memory allocation realm, a little information can go a long way. Application–specific information about allocation patterns helps implementing specialized memory allocators that heavily improve the bottom line of many high–performance applications. Sometimes as little intel as "blocks of size 80 are allocated more often than blocks all other sizes," when properly used (as we'll discuss soon), can incredibly improve the bottom line runtime of a program. While general-purpose allocators have average overheads in the hundreds of cycles, a good customized memory allocator can require as few as half a dozen cycles.

That's why many high–profile, high–performance applications (gcc, Apache, and Microsoft's SQL Server to name just a few) implement their own memory allocator. A good idea is, then, to generalize such good specialized allocators and put them in a library. But "generalization" and "specialization" are in tension: your application might have different allocation patterns that would require yet another behavior from your allocator. What to do?

But wait, there's more. If we do devise a method to easily create special–purpose memory allocators, we can go full–circle and define a *general–purpose* allocator as a combination of wisely chosen special–purpose allocators. If the resulting general–purpose allocator compares favorably with the existing monolithic general–purpose allocators, then the design is valid and useful.

Emery's team worked towards that idea, leading to their library HeapLayers (`http://heaplayers.org`). To define configurable allocators, they used mixins (also known as Coplien's curiously recurring pattern in the C++ community): defining classes with parameterized base. Each layer defines only two member functions, `malloc` and `free`.

```
template <class T>
struct Allocator : public T {
  void * malloc(size_t sz);
  void free(void* p);
  // system-dependent value
  enum { Alignment = sizeof(double) };
  // optional interface
  size_t getSize(const void* p);
};
```

Each layer implementation would get a crack on allocation and deallocation, possibly (and likely) requesting memory from its base class. A "self–contained" allocator sits at the very top of the hierarchy—one that forwards requests straight to the system's **new** and **delete** operators, `malloc` and `free` functions, and the such. In HeapLayers terminology, these are the *top heaps*. To exemplify (careful with the qualifications so we don't enter infinite recursion):

```
struct MallocHeap {
  void * malloc(size_t sz) {
    return std::malloc(sz);
  }
  void free(void* p) {
    return std::free(p);
  }
};
```

Top heaps can also be implemented around non-standard system calls for getting memory, such as UNIX's `sbrk`.

The `getSize` function has a special status. Not everybody is going to need it, so defining it is optional (if you combine the wrong layers, the compiler will tell you). If it does, all you have to do is to insert a layer that stores the block size and offers the `getSize` primitive:

```
template <class SuperHeap>
class SizeHeap {
  union freeObject {
    size_t sz;
    double _dummy; // for alignment.
  };
public:
  void * malloc(const size_t sz) {
    // Add room for a size field.
    freeObject * ptr = (freeObject *)
      SuperHeap::malloc(sz +
```

```
        sizeof(freeObject));
    // Store the requested size.
    ptr->sz = sz;
    return ptr + 1;
  }
  void free(void * ptr) {
    SuperHeap::free((freeObject *) ptr - 1);
  }
  static size_t getSize (const void * ptr) {
    return ((freeObject *)ptr - 1)->sz;
  }
};
```

SizeHeap is the perfect illustration of how to implement a useful layer that hooks into its base's malloc and free functions, does something extra, and returns "doctored" results to the client. SizeHeap does its work by allocating extra memory to store the block size, with the appropriate cautions (the **union**) to stay as immune as possible to the alignment issue. The client gets access to the memory right next to that extra data. It's not hard to imagine building a debug heap that pads the memory block before and after with some bytes filled with a particular pattern, and then verify for overruns by checking whether the pattern has been preserved. In fact, that's exactly what HeapLayers' DebugHeap layer does. Pretty neat.

But wait, something's suboptimal here. Some systems already offer a primitive to compute the size of a mallocated block. On those systems, SizeHeap would actually waste space. In that case (for example, on Microsoft Visual C++), you wouldn't need SizeHeap in conjunction with MallocHeap, because MallocHeap would implement getSize out of the box:

```
struct MallocHeap {
  ... as above ...
  size_t getSize(void* p) {
    return _msize(p);
  }
};
```

But wait, something's still suboptimal here. Remember, we're counting *cycles*. What if a system's malloc documentation states that the block size is stored in a word prior to the actual block? In that case, SizeHeap would still waste memory by storing yet another word next to the one already planted by the system. What's needed is a layer that implements getSize just the way SizeHeap does, but doesn't hook malloc and free. That's why HeapLayers segregates the SizeHeap shown above in two:

```
template <class Super>
struct UseSizeHeap : public Super {
  static size_t getSize(const void * ptr) {
    return ((freeObject *) ptr - 1)->sz;
  }
protected:
  union freeObject {
    size_t sz;
    double _dummy; // for alignment.
  };
};

template <class SuperHeap>
class SizeHeap
    : public UseSizeHeap<SuperHeap> {
  typedef typename
    UseSizeHeap<SuperHeap>::freeObject
      freeObject;
public:
  void * malloc(const size_t sz) {
    // Add room for a size field.
    freeObject * ptr = (freeObject *)
      SuperHeap::malloc(sz +
        sizeof(freeObject));
    // Store the requested size.
    ptr->sz = sz;
    return (void *) (ptr + 1);
  }
  void free(void * ptr) {
    SuperHeap::free((freeObject *)ptr - 1);
  }
};
```

Now SizeHeap always (correctly) adds the UseSizeHeap layer and exploits its getSize implementation, while UseSizeHeap can also be used in other configurations—a very elegant design.

3 A useful example: FreelistHeap

Let's face it, so far we kind of set up the stage. We do have an architecture, but no clue on how to write an efficient specialized allocator using layers. A popular and "most bang for the buck" strategy is the following:

1. Collect stats about your application's allocation counts for each size;
2. For the *most often asked* size (call it S), maintain a private singly-linked list;
3. Memory allocations for S return memory from that list if possible, or else from the default allocator (in a layered architecture, from the superior layer);
4. Memory deallocations for blocks of size S push the block into the list.

A refinement of the above would be to use the same freelist for a range of sizes S1 to S2, at the cost of some slack memory. The needed singly-linked list operations are O(1) and actually only a few instructions. In addition, the pointer to the next item can be stored in the actual block (the block stores no useful data—it's always a freed block!) so there's no extra memory required per block. Given that most applications' allocation size distribution is highly skewed, free lists are any allocator implementer's indispensable utensil.

Let's implement a layer that implements a free list for a range of statically-known sizes S1 to S2.

```
template <class Super, size_t S1, size_t S2>
struct FLHeap {
  ~FLHeap() {
    while (myFreeList) {
      freeObject* next = myFreeList->next;
      Super::free(myFreeList);
      myFreeList = next;
    }
  }
  void * malloc(const size_t s) {
    if (s < S1 || s > S2)) {
      return Super::malloc(s);
    }
    if (!myFreeList) {
      return Super::malloc(S2);
    }
    void * ptr = myFreeList;
    myFreeList = myFreeList->next;
    return ptr;
  }
  void free(void * p) {
    const size_t s = getSize(p);
    if (s < S1 || s > S2) {
      return Super::free(p);
    }
    freeObject p =
      reinterpret_cast<freeObject *>(ptr);
    p->next = myFreeList;
    myFreeList = p;
  }
private:
  /// The linked list pointer we embed in the freed object
  class freeObject {
  public:
    freeObject * next;
  };
  /// The head of the linked list of freed objects.
  freeObject * myFreeList;
};
```

Now you can define a custom heap like this:

```
typedef FLHeap<
  SizeHeap<MallocHeap>,
  24,
  32>
SmartoHeapo;
```

SmartoHeapo will be superfast for allocation sizes between 24 and 32, and pretty much the same for all other sizes.

4 In–Place Resizing

Many a C++ programmer has been dreaming for a standard primitive to reallocate memory in place.

You see, C has `realloc`, which can do in–place reallocation if it can, or use `memcpy` when it comes about copying data around. But `memcpy` doesn't work for C++ objects, so `realloc` doesn't work for C++ object, so any sort of `renew` primitive can't be implemented using the standard C allocator. So C++ doesn't have `renew` [1].

To give you an idea of the improvements that in-place reallocation can bring to C++ code, consider:

```
const int n = 10000;
Vec v;
for (int i = 0; i < n; ++i)
  v.push_back(0);
```

Howard Hinnant of Metrowerks has been working on implementing in-place expansion for CodeWarrior's standard library implementation. In Howard's own words:

> I currently have a `vector<T, malloc_allocator<T> >` that does [...] expand-in-place based on N1085 [7]. It blows the doors off a built-in C-like array! :-) When `Vec` is a `vector<int>` without expand-in-place:
>
> 0.00095674 seconds
>
> When `Vec` is a `vector<int>` with expand-in-place:
>
> 0.000416943 seconds
>
> The timings should only be trusted to 2 significant digits. But rest assured, I've done the work to secure those two digits. We are not looking at an anomaly here.

Given the benefits of in–place resizing and that each heap layer has control over its own allocation algorithms and data structures, let's augment the heap layer interface:

```
template <class T>
struct Allocator : public T {
  void * malloc(size_t sz);
  void free(void* p);
  size_t expand(void* p, size_t min, size_t max);
};
```

The semantics of `expand` is, try to expand the block pointed to by `p` to the largest size possible between `min` and `max`. Then return whatever size you could expand the memory to, or zero if no expansion was possible. Fortunately, things can be arranged such that a layer has to fuss about the `expand` routine unless it wants to. That works if all top allocators inherit the following little class:

```
struct TopHeap {
  size_t expand(void*, size_t, size_t) {
    return 0;
  }
  // not intended for standalone usage
protected:
  ~TopHeap() {}
};
```

5 Conclusion

Configurable memory allocation is, as Emery's research has shown, a practical, all–in–one alternative to both specialized and general purpose allocators. Emery's numbers (refer to the paper [6] for details) consistently show that allocators created with HeapLayers perform just as good or better than monolithic allocators, be they general–purpose or specialized. Moreover, HeapLayers' layered architecture encourages easier experimentation, simpler debugging, and unparalleled extensibility. Instead of being an oddball chapter of Modern C++ Design, memory allocation should have been one of the best success stories of policy–based design.

Table 1 shows a relevant subset of the layers implemented in HeapLayers. There would be a lot of goodies to discuss, such as the locked heaps for multithreaded operations, the STL adapter, the various utility heaps, or how the layers can be combined to create a general–purpose allocator. But `wc` warns us that "the mind can only enjoy what the butt can endure," so it's about time to shut down—without forgetting to release the memory in destructors. Happy coding.

6 Acknowledgments

Will go here.

A Library of Heap Layers	
Top Heaps	
mallocHeap	A thin layer over `malloc`
mmapHeap	A thin layer over the virtual memory manager
sbrkHeap	A thin layer over `sbrk` (contiguous memory)
Building-Block Heaps	
AdaptHeap	Adapts data structures for use as a heap
BoundedFreelistHeap	A freelist with a bound on length
ChunkHeap	Manages memory in chunks of a given size
CoalesceHeap	Performs coalescing and splitting
FreelistHeap	A freelist (caches freed objects)
Combining Heaps	
HybridHeap	Uses one heap for small objects and another for large objects
SegHeap	A general segregated fits allocator
StrictSegHeap	A strict segregated fits allocator
Utility Layers	
ANSIWrapper	Provides ANSI-malloc compliance
DebugHeap	Checks for a variety of allocation errors
LockedHeap	Code-locks a heap for thread safety
PerClassHeap	Use a heap as a per-class allocator
PHOThreadHeap	A private heaps with ownership allocator [5]
ProfileHeap	Collects and outputs fragmentation statistics
ThreadHeap	A pure private heaps allocator [5]
ThrowExceptionHeap	Throws an exception when the parent heap is out of memory
TraceHeap	Outputs a trace of allocations
UniqueHeap	A heap type that refers to one heap object
Object Representation	
CoalesceableHeap	Provides support for coalescing
SizeHeap	Records object sizes in a header
Special-Purpose Heaps	
ObstackHeap	A heap optimized for stack-like behavior and fast resizing
ZoneHeap	A zone ("region") allocator
XallocHeap	A heap optimized for stack-like behavior
General-Purpose Heaps	
KingsleyHeap	Fast but high fragmentation
LeaHeap	Not quite as fast but low fragmentation

Table 1: A library of heap layers, divided by category.

References

[1] Andrei Alexandrescu. Generic⟨Programming⟩: Typed Buffers (III). *C++ Experts Online*, December 2001. Available at `http://erdani.org/publications/cuj-12-2001.html`.

[2] Andrei Alexandrescu. *Modern C++ Design*. Addison-Wesley Longman, 2001.

[3] Andrei Alexandrescu. Generic⟨Programming⟩: Prying Eyes: A Policy-Based Observer (I). *C++ Users Journal*, April 2005.

[4] Andrei Alexandrescu. Generic⟨Programming⟩: Prying Eyes: A Policy-Based Observer (II). *C++ Users Journal*, June 2005.

[5] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 117–128. Cambridge, MA, November 2000. URL `citeseer.ist.psu.edu/berger00hoard.html`.

[6] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Composing High-Performance Memory Allocators. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 114–124, 2001. URL `http://citeseer.ist.psu.edu/berger01composing.html`.

[7] Howard Hinnant. Proposal to augment the interface of malloc/free/realloc/calloc. See `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1085.htm`.