`std::allocator` Is to Allocation what `std::vector` Is to Vexation

Prepared for CppCon
Bellevue, WA, Sep 20–25, 2015

Andrei Alexandrescu, Ph.D.
andrei@erdani.com

Working Titles

`std::allocator` Is to Allocation what Alligator Is to Allegation

Working Titles

Designing Allocators That Don’t Not Work

A Cyclical Business

- 1983: “`malloc` is awesome. Use it.”
- 1993: “`malloc` is awful. Replace it.”
- 2003: “`malloc` is great. Use it.”
- 2013: “`malloc` is wanting. Supplement it.”

History

It was a Dark and Stormy Night

- It all started with C:

```
void* malloc(size_t size);
void free(void* p);
```

- Caller knows the size to allocate
- Caller knows the size to deallocate
- Caller doesn't tell `free` the size to deallocate

$\Rightarrow$ Allocator must manage size

Size management adds undue difficulties and inefficiencies to any allocator design

Turning Time Back

```
struct Blk { void* ptr; size_t length; };

Blk malloc(size_t size);
void free(Blk block);
```

`operator new`: The Next Generation

`operator new`

- Had to work with `malloc`
- Added type specificity
- Several syntactic oddities
- No communication with the constructor
- "The Great Array Distraction"

- Generally unrecommended for any serious use

To Wit: N3536

"[...] omission [of size information in global `operator delete`] has unfortunate performance consequences. Modern memory allocators often allocate in size categories, and, for space efficiency reasons, do not store the size of the object near the object. Deallocation then requires searching for the size category store that contains the object. This search can be expensive, particularly as the search data structures are often not in memory caches."

Along came

`std::allocator`

`std::allocator`

- Failure?
- But it’s still used!
- “Making Allocators Work”, Alisdair Meredith, CppCon 2014

- What’s good?
- What could be done better?

Development

- Meant as a fix for `near`/`far` pointers
- No intent to expand scope beyond that
- Just distraction in the midst of a large design

- `std::allocator` isn’t a good allocator because it wasn’t designed to be a good allocator

Obvious Mistakes

- Type as parameter
  - Misunderstanding of charter
  - Allocator, not factory!
  - Yet still traffics in `void*`
  - Allocators should trade in `Blk`
  - See also: `rebind<U>::other`, indexed under “awful crutch” in Merriam-Webster

More Obvious Mistakes

- Stateless assumption
  - *All* allocators have state!
  - Some is monostate
  - Assume state present
  - A global allocator object optional

Subtle Issue: Composition

- Most actual allocators are compositions
- Battery specialized by size classes
- Arrays/lists/trees of other allocators
- Hooks on top of others (freelists, debug, stats)

- Getting composition right is getting allocators right

New Design

A Leg Up

- Efficiency
  - Leaving size to the client
  - Scoped allocation patterns
  - Thread-local allocation patterns
- Features
  - Better configurability (debug/stats)
  - Specialization/adaptation
  - No legacy, no nonsense

Simplest Composite Allocator

Let's define `FallbackAllocator`

```
template <class Primary, class Fallback>
class FallbackAllocator
      : private Primary
      , private Fallback {
public:
   Blk allocate(size_t);
   void deallocate(Blk);
};
```

FallbackAllocator::allocate

```
template <class P, class F>
Blk FallbackAllocator<P, F>::allocate(
      size_t n) {
   Blk r = P::allocate(n);
   if (!r.ptr) r = F::allocate(n);
   return r;
}
```


FallbackAllocator::deallocate

```
template <class P, class F>
void FallbackAllocator<P, F>::deallocate(
      Blk b) {
   if (P::owns(b)) P::deallocate(b);
   else F::deallocate(b);
}
```

- New optional method: `bool owns(Blk)`
- `Primary` must define it

Good Citizenry

```
template <class P, class F>
bool FallbackAllocator::owns(Blk b) {
   return P::owns(b) || F::owns(b);
}
```

- Relies on MDFINAE

Suddenly!

```
template <size_t s> class StackAllocator {
   char d_[s];
   char* p_;
   ...
};
class Mallocator { ... };
...
using Localloc = FallbackAllocator<
   StackAllocator<16384>,
   Mallocator>;
...
```

Use

```
void fun(size_t n) {
   Localloc a;
   auto b = a.allocate(n * sizeof(int));
   SCOPE_EXIT { a.deallocate(b); };
   int* p = static_cast<int*>(b.ptr);
   ... use p[0] through p[n - 1] ...
}
```


StackAllocator

```
template <size_t s> class StackAllocator {
   ...
   StackAllocator() : p_(d_) {}
   Blk allocate(size_t n) {
      auto n1 = roundToAligned(n);
      if (n1 > (d_ + s) - p_) {
         return { nullptr, 0 };
      }
      Blk result = { p_, n };
      p_ += n1;
      return result;
   }
   ...
};
```

StackAllocator

```
...
void deallocate(Blk b) {
   if (b.ptr + roundToAligned(n) == p_) {
      p_ = b.ptr;
   }
}
bool owns(Blk b) {
   return b.ptr >= d_ && b.ptr < d_ + s;
}
// NEW: deallocate everything in O(1)
void deallocateAll() {
   p_ = d_;
}
...
```


Freelist

- Keeps list of previous allocations of a given size

```
template <class A, size_t s> class Freelist {
   A parent_;
   struct Node { Node* next; } root_;
public:
   Blk allocate(size_t n) {
      if (n == s && root_) {
         Blk b = { root_, n };
         root = root_.next;
         return b;
      }
      return parent_.allocate(n);
   }
   ...
};
```

Freelist

```
...
bool owns(Blk b) {
   return b.length == s || parent_.owns(b);
}
void deallocate(Blk b) {
   if (b.length != s) return parent_.deallocate(b);
   auto p = (Node*)b.ptr;
   p.next = root_;
   root_ = p;
}
```


Freelist improvements

- Add tolerance: `min...max`
- Allocate in batches
- Add upper bound: no more than `top` elems

`Freelist` improvements

```
Freelist<A, // parent allocator
   17, // Use list for object sizes 17...
   32, // .. through 32
   8, // Allocate 8 at a time
   1024 // No more than 1024 remembered
> alloc8r;
```


Adding Affixes

```
template <class A, class Prefix,
      class Suffix = void>
class AffixAllocator;
```

- Add optional prefix and suffix
- Construct/destroy appropriately
- Uses: debug, stats, info, . . .

“How are we doing?”

```
template <class A, ulong flags>
class AllocatorWithStats;
```

- Collect info about any other allocator
- Per-allocation and global
  - primitive calls, failures, byte counts, high tide
- Per-allocation
  - caller file/line/function/time

Bitmapped Block

```
template <class A, size_t blockSize>
class BitmappedBlock;
```

- Orgnized by constant-size blocks
- Tremendously simpler than `malloc`’s heap
- Faster, too
- Layered on top of any memory hunk
- 1 bit/block sole overhead (!)
- Multithreading tenuous (needs full interlocking for >1 block)

When one fills up...

```
template <class Creator>
class CascadingAllocator;
...
auto a = cascadingAllocator({
   return Heap<...>();
});
```

- Keep a list of allocators, grown lazily
- Coarse granularity
- Linear search upon allocation
- May return memory back

Segregator

```
template <size_t threshold,
   class SmallAllocator,
   class LargeAllocator>
struct Segregator;
```

- Sizes $\leq$ `threshold` goes to `SmallAllocator`
- All else goes to `LargeAllocator`

`Segregator`: self-composable!

```
typedef Segregator<4096,
   Segregator<128,
      Freelist<Mallocator, 0, 128>,
      MediumAllocator>,
   Mallocator>
Allocator;
```

- Could implement any “search” strategy
- Works only for a handful of size classes

`Bucketizer`: Size Classes

```
template <class Allocator,
      size_t min,
      size_t max,
      size_t step>
struct Bucketizer;
```

- `[min, min + step)`, `[min + step, min + 2 * step)...`
- Within a bucket allocates the maximum size

We’re done!

Most major allocation tropes covered

Approach to copying

- Allocator-dependent
- Some noncopyable & immovable: stack/in-situ allocator
- Some noncopyable, movable: non-stack regions
- Many refcounted, movable
- Perhaps other models, too

Granularity

- Where are types, factories, . . . ?
- Design surprise: these are extricable
- Focus on `Blk`

Realistic Example

```
using FList = Freelist<Mallocator, 0, -1>;
using A = Segregator<
   8, Freelist<Mallocator, 0, 8>,
   128, Bucketizer<FList, 1, 128, 16>,
   256, Bucketizer<FList, 129, 256, 32>,
   512, Bucketizer<FList, 257, 512, 64>,
   1024, Bucketizer<FList, 513, 1024, 128>,
   2048, Bucketizer<FList, 1025, 2048, 256>,
   3584, Bucketizer<FList, 2049, 3584, 512>,
   4072 * 1024, CascadingAllocator<
      decltype(newHeapBlock)>,
   Mallocator
>;
```

Complete API

```
static constexpr unsigned alignment;
static constexpr goodSize(size_t);
Blk allocate(size_t);
Blk allocateAll();
bool expand(Blk&, size_t delta);
void reallocate(Blk&, size_t);
bool owns(Blk);
void deallocate(Blk);
void deallocateAll();
```


Complete API: aligned versions

```
Blk alignedAllocate(size_t, unsigned);
Blk alignedReallocate(size_t, unsigned);
```

- `AlignedMallocator`
- `posix_memalign` on Posix
- `_aligned_malloc` on Windows
- Regions support this as well!

Summary

Summary

- Fresh approach from first principles
- Understanding history
  - Otherwise: “. . . doomed to repeat it”
- Composability is key