Generic Programming Must Go

Andrei Alexandrescu

Heap Building Blocks

Musings on Design

- Procedural: Work with unseen data
- OO, Functional: Work with unseen code and data
- Generic: Work with unseen code types and data layout

Generic Programming

"...programming paradigm whereby fundamental requirements on types are abstracted from across concrete examples of algorithms and data structures and formalised as concepts, with generic functions implemented in terms of these concepts..." — Wikipedia

Generic Programming

- Focus on algorithms

Generic Programming

- Focus on algorithms
- Good abstraction power

Generic Programming

- Focus on algorithms
- Good abstraction power
- No indirection, so good speed

Generic Programming

+ Focus on algorithms
+ Good abstraction power
+ No indirection, so good speed

− Rigid; very limited adaptability

Generic Programming

+ Focus on algorithms
+ Good abstraction power
+ No indirection, so good speed

– Rigid; very limited adaptability
– Only works for small, scarce-vocabulary domains

Generic Programming

+ Focus on algorithms
+ Good abstraction power
+ No indirection, so good speed

− Rigid; very limited adaptability
− Only works for small, scarce-vocabulary domains
− Obsessed with naming everything

We've already "betrayed" GP

- `InputRange`, `ForwardRange`, `BidirectionalRange`, `RandomAccessRange`

We’ve already “betrayed” GP

- `InputRange`, `ForwardRange`, `BidirectionalRange`, `RandomAccessRange`
- `hasLength`, `isInfinite`, `hasSlicing`, `hasMobileElements`

We've already "betrayed" GP

- `InputRange`, `ForwardRange`, `BidirectionalRange`, `RandomAccessRange`
- `hasLength`, `isInfinite`, `hasSlicing`, `hasMobileElements`

- By the canon: `InputRangeWLength`, `ForwardRangeWLength`, `BidirectionalRangeWLength`, `RandomAccessRangeWLength`, `InputRangeInfinite`, `ForwardRangeInfinite`, `BidirectionalRangeInfinite`, `RandomAccessRangeInfinite`, `RandomAccessRangeWSlicing`, `RandomAccessRangeWLengthWSlicing`, `RandomAccessRangeInfiniteWSlicing`, ...

And It Was Very Good

Uhm, Allocator Connection?

- Memory allocation is high-vocabulary

Uhm, Allocator Connection?

- Memory allocation is high-vocabulary
  - alignment

Uhm, Allocator Connection?

- Memory allocation is high-vocabulary
  - alignment
  - (dynamically) aligned allocation

Uhm, Allocator Connection?

- Memory allocation is high-vocabulary
  - alignment
  - (dynamically) aligned allocation
  - rounding up/quantization

Uhm, Allocator Connection?

- Memory allocation is high-vocabulary
  - alignment
  - (dynamically) aligned allocation
  - rounding up/quantization
  - in-place expansion

Uhm, Allocator Connection?

- Memory allocation is high-vocabulary
  - alignment
  - (dynamically) aligned allocation
  - rounding up/quantization
  - in-place expansion
  - reallocation

Uhm, Allocator Connection?

- Memory allocation is high-vocabulary
  - alignment
  - (dynamically) aligned allocation
  - rounding up/quantization
  - in-place expansion
  - reallocation
  - contiguous vs. non-contiguous

Uhm, Allocator Connection?

- Memory allocation is high-vocabulary
  - alignment
  - (dynamically) aligned allocation
  - rounding up/quantization
  - in-place expansion
  - reallocation
  - contiguous vs. non-contiguous
  - ownership

Uhm, Allocator Connection?

- Memory allocation is high-vocabulary
  - alignment
  - (dynamically) aligned allocation
  - rounding up/quantization
  - in-place expansion
  - reallocation
  - contiguous vs. non-contiguous
  - ownership
  - resolving internal pointers

Uhm, Allocator Connection?

- Memory allocation is high-vocabulary
  - alignment
  - (dynamically) aligned allocation
  - rounding up/quantization
  - in-place expansion
  - reallocation
  - contiguous vs. non-contiguous
  - ownership
  - resolving internal pointers
  - deallocation

Uhm, Allocator Connection?

- Memory allocation is high-vocabulary
  - alignment
  - (dynamically) aligned allocation
  - rounding up/quantization
  - in-place expansion
  - reallocation
  - contiguous vs. non-contiguous
  - ownership
  - resolving internal pointers
  - deallocation
  - per-instance state vs. monostate

Uhm, Allocator Connection?

- Memory allocation is high-vocabulary
  - alignment
  - (dynamically) aligned allocation
  - rounding up/quantization
  - in-place expansion
  - reallocation
  - contiguous vs. non-contiguous
  - ownership
  - resolving internal pointers
  - deallocation
  - per-instance state vs. monostate
  - thread-local vs. shared


I tried to design a
generic allocator,
and I didn't even get
this lousy T-shirt

Let's Go Descartes!

Design by Introspection

Simplest Design That Could Possibly Work

- Make all allocation primitives optional, except:
  - `void[] allocate(size_t);`
  - `enum uint alignment;`
- All others optional, probed introspectively
- e.g. `hasMember!(A, "expand")`

- Combination allocators that define and adapt capabilities to their “hosts”, in very little code

Simplest Allocator

- “Push the pointer”

```
struct Region {
  private void* b, e, p;
  this(void[] buf) {
    p = b = buf.ptr;
    e = b + buf.length;
  }
  enum uint alignment = 1;
  void[] allocate(size_t n) {
    if (e - p < n) return null;
    auto result = p[0 .. n];
    p += n;
    return result;
  }
}
```

Immediate Improvements

- Support better alignments (1 is seldom useful)
- Embed buffer
- Or, release buffer in destructor?
- More primitives such as `deallocateAll`

Simplest Composite Allocator

Let's define `FallbackAllocator`: try one, then another

```
struct FallbackAllocator(P, F) {
  P primary;
  F fallback;
  enum alignment = min(P.alignment,
    F.alignment);
  void[] allocate(size_t n) {
    auto r = p.allocate(n);
    if (r.length != n) r = f.allocate(n);
    return r;
  }
}
```

And Suddenly!

```
alias Local = FallbackAllocator!(
  Region,
  Mallocator
);
```

We Want Deallocation!

- Optional method: `void deallocate(void[]);`

```
static if (hasMember!(P, "owns")
  && (hasMember!(P, "deallocate")
    || hasMember!(F, "deallocate")))
void deallocate(void[] b) {
  if (p.owns(b)) {
    static if (hasMember!(P, "deallocate"))
        primary.deallocate(b);
  } else {
    static if (hasMember!(F, "deallocate"))
        return f.deallocate(b);
  }
}
```

- Need a new method
- Only `P` must define `owns`

Let’s take a look at all *optional* methods

Propagating `owns`

```
static if (hasMember!(P, "owns")
  && hasMember!(F, "owns"))
bool owns(void[] b) {
  return p.owns(b) || f.owns(b);
}
```

How about reallocation?

```
bool reallocate(ref void[] b, size_t newSize) {
  if (newSize == 0) {
    static if (hasMember!(typeof(this), "deallocate"))
        deallocate(b);
    return true;
  }
  if (b is null) {
    b = allocate(newSize);
    return b !is null;
  }
  ...
```

- (Note on introspection: “Would I be able to do that?”)

`reallocate` (2 of 3)

```
...
bool crossAllocatorMove(F, T)(ref F from, ref T to) {
  auto b1 = to.allocate(newSize);
  if (!b1.ptr) return false;
  if (b.length < newSize) b1[0 .. b.length] = b[];
  else b1[] = b[0 .. newSize];
  static if (hasMember!(From, "deallocate"))
      from.deallocate(b);
  b = b1;
  return true;
}
...
```

`reallocate` (the pride)

```
  ...
  if (b is null || p.owns(b)) {
    if (p.reallocate(b, newSize)) return true;
    // Move from p to f
    return crossAllocatorMove(p, f);
  }
  if (f.reallocate(b, newSize)) return true;
  // Interesting. Move from f to p.
  return crossAllocatorMove(f, p);
}
```

Global `reallocate`

```
bool reallocate(A)(ref A a, ref void[] b, size_t s) {
  if (b.length == s) return true;
  static if (hasMember!(A, "expand")) {
    if (b.length <= s && a.expand(b, s - b.length))
      return true;
  }
  auto r = a.allocate(s);
  if (r.length != s) return false;
  if (r.length <= b.length) r[] = b[0 .. newB.length];
  else r[0 .. b.length] = b[];
  static if (hasMember!(A, "deallocate"))
      a.deallocate(b);
  b = r;
  return true;
}
```

Segregating by Size

```
struct Segregator(size_t threshold,
    Small, Large) {
  Small small;
  Large large;
  enum alignment = min(Small.alignment,
    Large.alignment);
  void[] allocate(size_t n) {
    return n <= threshold
      ? small.allocate(n)
      : large.allocate(n);
  }
}
```

```d
static if (hasMember!(SmallAllocator, "expand")
  || hasMember!(LargeAllocator, "expand"))
bool expand(ref void[] b, size_t delta) {
  if (b.length + delta <= threshold) {
      // Old and new allocations handled by _small
      static if (hasMember!(SmallAllocator, "expand"))
        return _small.expand(b, delta);
      else
        return false;
  }
  if (b.length > threshold) {
    // Old and new allocations handled by _large
    static if (hasMember!(LargeAllocator, "expand"))
      return _large.expand(b, delta);
    else
      return false;
  }
  // Oops, cross-allocator transgression
  return false;
}
```

Design by Introspection Tenets

- Compose designs from small pieces
- Distinguish required from optional methods
- No need to *name* all combinations
  - Generic Programming is fail
  - Concepts are fail
- Assemble using introspection
- Use Boolean logic and `static if`
  - Constrain types and signatures

- Yay

Take a look

- https://github.com/andralex/phobos/tree/allocator/std/experimental/allocator
- http://erdani.com/d/phobos-prerelease/std_experimental_allocator.html

Perk: Ouroboros Style

Array of Allocators: Going Too Meta?

- Goal:
  - Define an array of generic allocators
    - e.g. `Regions`, `HeapBlocks`...
  - Grow and shrink the array per application needs
  - Keep some per-allocator metadata
- Question:
  - Where do you store the array?

Solution: Going Ouroboros!

- Create an allocator on the stack
- Use it to allocate the needed metadata memory
- Move it to that memory
- Keep a pointer to the metadata in the meta-allocator

- Problem solved

Summary

- Generic Programming insufficient for flexible designs
- *Design by Introspection* being proposed
- Give components required and optional APIs
- Use introspection to assemble larger designs from small components

For My Money

Static introspection + CTFE + Boolean constraints + `static if` = WIN