Analysis of the Design Space of a Container Library for D (Academic rigor or pedanticism? You decide.)

Dr. Robert Schadek

DConf Online 2024

The Problem

- Everybody wants good containers like C++'s std::vector in D.

The Problem

- Everybody wants good containers like C++'s std::vector in D.

What is good?

Design Space

- @safe
- const ness
- const container
- const values
- allocators
- nested container
- iteration
- ranges
- bound checked index
- Exceptions / Error Handling
- value type, reference types, pointer types
- small size optimization
- multi threading / shared / lock-free ness
- performance
- container types

Bound Checks / Exceptions

```
struct Vec(T) {
@safe:
  T[] arr;

  void append(T t) {
    this.arr ~= t;
  }

  ref T opIndex(size_t idx) @trusted {
    if(idx >= this.arr.length) {
      throw new Exception("OOB");
    }
    return *(arr.ptr + idx);
  }
```

```
  Nullable!(T*) opIndexN(size_t idx) {
    if(idx >= this.arr.length) {
      return Nullable!(T*).init;
    }
    return nullable(&arr[idx]);
  }

  bool opIndexNN(size_t idx
      , out Nullable!(T*) o)
  @trusted {
    if(idx >= this.arr.length) {
      o = Nullable!(T*).init;
      return false;
    }
    o = nullable(this.arr.ptr + idx);
    return true;
  }
```

Allocators

```
alias FList = FreeList!(Mallocator, 0, unbounded);
alias Allocator = 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, AllocatorList!(
        (n) => BitmappedBlock!(4096)(
                cast(ubyte[])(Mallocator.instance.allocate(
                    max(n, 4072 * 1024)))),
    Mallocator
);
```

Allocators

```
struct Vector1(T,A) {
  A* allocator;
}

unittest {
  Vector1!(int, Allocator) vec;

  {
    Allocator tuMalloc;
    vec = Vector1!(int, Allocator)(&tuMalloc);
  }
}
```

Allocators

```d
struct Vector2(T,A) {
  A* allocator;

  @disable this(this);
  @disable ref typeof(this) opAssign()(auto ref typeof(this) rhs);
}

unittest {
  Vector2!(int, Allocator) vec;

  {
    Allocator tuMalloc;
    auto vec2 = Vector2!(int, Allocator)(&tuMalloc);

    // The next line doesn't compile
    //vec = Vector2!(int, Allocator)(&tuMalloc);
  }
}
```

Iteration / Ranges

```
struct Vec(T) {
@safe:
  T[] arr;

  void append(T t) {
    this.arr ~= t;
  }

  size_t length() @property {
    return this.arr.length;
  }

  ref T opIndex(size_t idx) return scope {
    return this.arr[idx];
  }

  ref T opIndexFast(size_t idx) @trusted {
    return *(arr.ptr + idx);
  }
```

```
  ViaPtr!(T) slicePtr(size_t b
      , size_t e) @trusted
  {
    return ViaPtr!(T)(this.arr.ptr + b
        , this.arr.ptr + e);
  }

  ViaIdx!(T) sliceIdx(size_t b
      , size_t e) @trusted
  {
    return ViaIdx!(T)(&this, b, e);
  }
}
```

ViaIdx

```
struct ViaIdx(T) {
  Vec!(T)* vec;
  size_t idx;
  size_t end;

  ref T front() @property {
    return this.vec.opIndexFast(idx);
  }

  void popFront() {
    this.idx++;
  }

  bool empty() @property {
    return this.idx >= this.end;
  }
}
```

```
unittest {
  Vec!(int) v;
  foreach(i; 0 .. 10) {
    v.append(i);
  }

  int i;
  foreach(it; v.sliceIdx(0, 10)) {
    int f = it;
    assert(f == i);
    ++i;
  }
  assert(i == 10);
}
```

ViaPtr

```d
struct ViaPtr(T) {
  T* ptr;
  T* end;

  ref T front() @property {
    return *this.ptr;
  }

  void popFront() {
    this.ptr++;
  }

  bool empty() @property {
    return this.ptr >= this.end;
  }
}
```

```d
unittest {
  Vec!(int) v;
  foreach(i; 0 .. 10) {
    v.append(i);
  }

  int i;
  foreach(it; v.slicePtr(0, 10)) {
    int f = it;
    assert(f == i);
    ++i;
  }
  assert(i == 10);
}
```

Constness

```
struct Vec(T) {
@safe:
  T[] arr;

  void append(T t) {
    this.arr ~= t;
  }

  size_t length() const @property {
    return this.arr.length;
  }

  ref inout(T) opIndex(size_t idx) inout {
    return this.arr[idx];
  }
```

This feels wrong

What can we gain

- Allocators
- Deterministic destruction

Container Types / What do we want

- Vector / Growable Array
- Hash Map / Hash Set

Container Types / What do we want

- Vector / Growable Array
- Hash Map / Hash Set

- ~~Linked List~~

Container Types / What do we want

- Vector / Growable Array
- Hash Map / Hash Set

- ~~Linked-List~~
- ~~Map / Set~~

What do we have to pay

- Allocators → ~~@safe~~
- Deterministic destruction → interesting

Allocators → ~~@safe~~

```
unsafe impl GlobalAlloc for MyAllocator {
    unsafe fn alloc(&self, layout: Layout) -> *mut u8 {
        System.alloc(layout)
    }

    unsafe fn dealloc(&self, ptr: *mut u8, layout: Layout) {
        System.dealloc(ptr, layout)
    }
}
```

Perfect is the enemy of good

Magic

```
void fun(const(int)[] arr) {
}

unittest {
  const(int)[] a = [1,2,3];
  fun(a);
  const(int[]) b = [1,2,3];
  fun(b);
}
```

More Magic

```
double median(ulong[] arr) pure {
  if(arr.length % 2 == 0) {
    return (cast(double)arr[($-1)/2] + arr[$/2]) / 2;
  } else {
    return arr[$ / 2];
  }
}

unittest {
  ulong[] byLength = readText(__FILE__)
    .splitter()
    .map!(s => s.length)
    .array
    .sort
    .uniq
    .array;
  writefln("%s", median(byLength));
}
```

More Magic

```
unittest {
  ulong[] byLength = readText(__FILE__)
    .splitter()
    .map!(s => s.length)
    .array
    .sort
    .uniq
    .array;

  writefln("%s", median(byLength));
  GC.free(byLength.ptr); // compiler generated
}
```

More Magic

```d
unittest {
  string txt = readText(__FILE__);
  ulong[] a = txt
    .splitter()
    .map!(s => s.length)
    .array
    .sort
    .array;

  GC.free(cast(void*)txt.ptr);    // compiler generated

  ulong[] byLength = a
    .uniq
    .array;

  GC.free(a.ptr);                 // compiler generated
  writefln("%s", median(byLength));
  GC.free(byLength.ptr);          // compiler generated
}
```

Magic The Gathering

```
unittest {
  void[string] i_am_a_set;
}
```

Rant

We can't have the cake and eat it too

- `@safe` + allocator
- barrier to entry
- understandable api
- good look and feel

More ranting

Why don't we have a de facto container library in code.dlang.org?

Conclusion

Conclusion

- Almost always use `int[]` or `int[string]`

Conclusion

- Almost always use `int[]` or `int[string]`
- Un-safe container that require an allocator

The End