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.
- But 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

```
1  struct Vec(T) {
2  @safe:
3    T[] arr;
4
5    void append(T t) {
6      this.arr ~= t;
7    }
8
9    ref T opIndex(size_t idx) @trusted {
10     if(idx >= this.arr.length) {
11       throw new Exception("OOB");
12     }
13     return *(arr.ptr + idx);
14   }
```

```
19   Nullable!(T*) opIndexN(size_t idx) {
20     if(idx >= this.arr.length) {
21       return Nullable!(T*).init;
22     }
23     return nullable(&arr[idx]);
24   }
25
26   bool opIndexNN(size_t idx
27       , out Nullable!(T*) o)
28   {
29     if(idx >= this.arr.length) {
30       o = Nullable!(T*).init;
31       return false;
32     }
33     o = nullable(&arr[idx]);
34     return true;
35   }
```

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

```
1  struct Vector1(T,A) {
2    A* allocator;
3  }
4
5  unittest {
6    Vector1!(int, Allocator) vec;
7
8    {
9      Allocator tuMalloc;
10     vec = Vector1!(int, Allocator)(&tuMalloc);
11   }
12 }
```

Allocators

```
1  struct Vector2(T,A) {
2    A* allocator;
3
4    @disable this(this);
5    @disable ref typeof(this) opAssign()(auto ref typeof(
       this) rhs);
6  }
7
8  unittest {
9    Vector2!(int, Allocator) vec;
10
11   {
12     Allocator tuMalloc;
13     auto vec2 = Vector2!(int, Allocator)(&tuMalloc);
14
15     // The next line doesn't compile
16     //vec = Vector2!(int, Allocator)(&tuMalloc);
17   }
18 }
```

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

```
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;
  }
}
```

```
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

```
1  struct Vec(T) {
2  @safe:
3    T[] arr;
4
5    void append(T t) {
6      this.arr ~= t;
7    }
8
9    size_t length() const @property {
10     return this.arr.length;
11   }
12
13   ref inout(T) opIndex(size_t idx) inout {
14     return this.arr[idx];
15   }
```

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 $\rightarrow$ ~~@safe~~
- Deterministic destruction $\rightarrow$ interesting

Perfect is the enemy of good

Magic

```
1  void fun(const(int)[] arr) {
2  }
3
4  unittest {
5    const(int)[] a = [1,2,3];
6    fun(a);
7    const(int[]) b = [1,2,3];
8    fun(b);
9  }
```