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

Dr. Robert Schadek
DConf Online 2024

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

- 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

```
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   @trusted {
29     if(idx >= this.arr.length) {
30       o = Nullable!(T*).init;
31       return false;
32     }
33     o = nullable(this.arr.ptr + idx);
34     return true;
35   }
```

Allocators

```
1  alias FList = FreeList!(Mallocator, 0, unbounded);
2  alias Allocator = Segregator!(
3      8, FreeList!(Mallocator, 0, 8),
4      128, Bucketizer!(FList, 1, 128, 16),
5      256, Bucketizer!(FList, 129, 256, 32),
6      512, Bucketizer!(FList, 257, 512, 64),
7      1024, Bucketizer!(FList, 513, 1024, 128),
8      2048, Bucketizer!(FList, 1025, 2048, 256),
9      3584, Bucketizer!(FList, 2049, 3584, 512),
10     4072 * 1024, AllocatorList!(
11         (n) => BitmappedBlock!(4096)(
12                 cast(ubyte[])(Mallocator.instance.allocate(
13                     max(n, 4072 * 1024)))))),
14     Mallocator
15 );
```

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 }
```

```
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 }
```

6

```
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() @property {
10     return this.arr.length;
11   }
12
13   ref T opIndex(size_t idx) return scope {
14     return this.arr[idx];
15   }
16
17   ref T opIndexFast(size_t idx) @trusted {
18     return *(arr.ptr + idx);
19   }
```

```
1  ViaPtr!(T) slicePtr(size_t b
2      , size_t e) @trusted
3  {
4    return ViaPtr!(T)(this.arr.ptr + b
5        , this.arr.ptr + e);
6  }
7
8  ViaIdx!(T) sliceIdx(size_t b
9      , size_t e) @trusted
10 {
11   return ViaIdx!(T)(&this, b, e);
12 }
13 }
```

ViaIdx

```
1  struct ViaIdx(T) {
2    Vec!(T)* vec;
3    size_t idx;
4    size_t end;
5
6    ref T front() @property {
7      return this.vec.opIndexFast(idx);
8    }
9
10   void popFront() {
11     this.idx++;
12   }
13
14   bool empty() @property {
15     return this.idx >= this.end;
16   }
17 }
```

```
1  unittest {
2    Vec!(int) v;
3    foreach(i; 0 .. 10) {
4      v.append(i);
5    }
6
7    int i;
8    foreach(it; v.sliceIdx(0, 10)) {
9      int f = it;
10     assert(f == i);
11     ++i;
12   }
13   assert(i == 10);
14 }
```

ViaPtr

```
1  struct ViaPtr(T) {
2    T* ptr;
3    T* end;
4
5    ref T front() @property {
6      return *this.ptr;
7    }
8
9    void popFront() {
10     this.ptr++;
11   }
12
13   bool empty() @property {
14     return this.ptr >= this.end;
15   }
16 }
```

```
1  unittest {
2    Vec!(int) v;
3    foreach(i; 0 .. 10) {
4      v.append(i);
5    }
6
7    int i;
8    foreach(it; v.slicePtr(0, 10)) {
9      int f = it;
10     assert(f == i);
11     ++i;
12   }
13   assert(i == 10);
14 }
```

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

- Allocators
- Deterministic destruction

- Vector / Growable Array
- Hash Map / Hash Set

- Vector / Growable Array
- Hash Map / Hash Set

- ~~Linked-List~~

- Vector / Growable Array
- Hash Map / Hash Set

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

What do we have to pay

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

```
1  unsafe impl GlobalAlloc for MyAllocator {
2      unsafe fn alloc(&self, layout: Layout) -> *mut u8 {
3          System.alloc(layout)
4      }
5
6      unsafe fn dealloc(&self, ptr: *mut u8, layout: Layout) {
7          System.dealloc(ptr, layout)
8      }
9  }
```

Perfect is the enemy of good

```
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  }
```

More Magic

```
1  unittest {
2    ulong[] a = readText(__FILE__)
3      .splitter()
4      .map!(s => s.length)
5      .array
6      .sort
7      .uniq
8      .array;
9  }
```

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
  GC.free(byLength.ptr);       // compiler generated
}
```

```
1  unittest {
2    void[string] i_am_a_set;
3  }
```

Rant

- `@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 `int[]` or `int[string]`

Conclusion

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

- Un-safe container that require an allocator

The End