The Set of Natural Code

Mark Isaacson

An invitation for questions

A Little History, C to C++

```
void* myCContainer =
containerMake();
containerAddElt(myCContainer, 5);
containerFree(myCContainer);
```

```
vector<int> container{5};
```

A Little History, C to C++

```
void* myCContainer =
containerMake();
containerAddElt(myCContainer, 5);
containerFree(myCContainer);
```

```
vector<int> container{5};
```

```
#define SQUARE(x) ((x)*(x))
```

```
template<T> T square(const T& x) {
  return x*x;
}
```

D Basics and Buzzwords

- A systems language
- Compiles to native code
- Statically typed
- Choice of value or reference semantics
- Modules (not textual inclusion)
- Garbage collected (can also manage memory manually)
- Built-in associative arrays with slicing
- Ranges
- Generic programming
- Compile-time function evaluation
- Polymorphism

Getting our feet wet

```
#!/usr/bin/rdmd
import std.stdio;
void main() {
  writeln("Hello world!");
}
```

Power to weight ratio

```
#!/usr/bin/rdmd
import std.stdio, std.array, std.algorithm;
void main() {
  auto file = File("file.txt");
  file.byLine
      .array
      .sort
      .joiner("\n")
      .writeln;
}
```

A Rewrite

```
#!/usr/bin/rdmd
import std.stdio, std.array, std.algorithm;
void main() {
  auto file = File("file.txt");
  auto sortedFile = sort(file.byLine.array);
  writeln(joiner(sortedFile, "\n"));
}
```

The C++

```
int main() {
  ifstream infile("file.txt");
  vector<string> lines;
  string line;
  while (getline(infile, line)) {
    lines.emplace_back(move(line));
  }
  sort(begin(lines), end(lines));
  ostringstream ss;
  move(begin(lines), end(lines), ostream_iterator<string>(ss, "\n"));
  cout << ss.str() << endl;
}
```

Primitives: Arrays

```
int[] arr = [1, 2, 3];
arr ~= [4, 5]; //Concatenate
assert(arr == [1, 2, 3, 4, 5] && arr.length == 5);
```

Primitives: Arrays

```
int[] arr = [1, 2, 3];
arr ~= [4, 5]; //Concatenate
assert(arr == [1, 2, 3, 4, 5] && arr.length == 5);

int[] aliasOfArr = arr; //Default reference semantics (for arrays)
aliasOfArr[0] = 0;
assert(arr == [0, 2, 3, 4, 5]);
int[] copy = arr.dup; //A deep copy
```

Primitives: Arrays

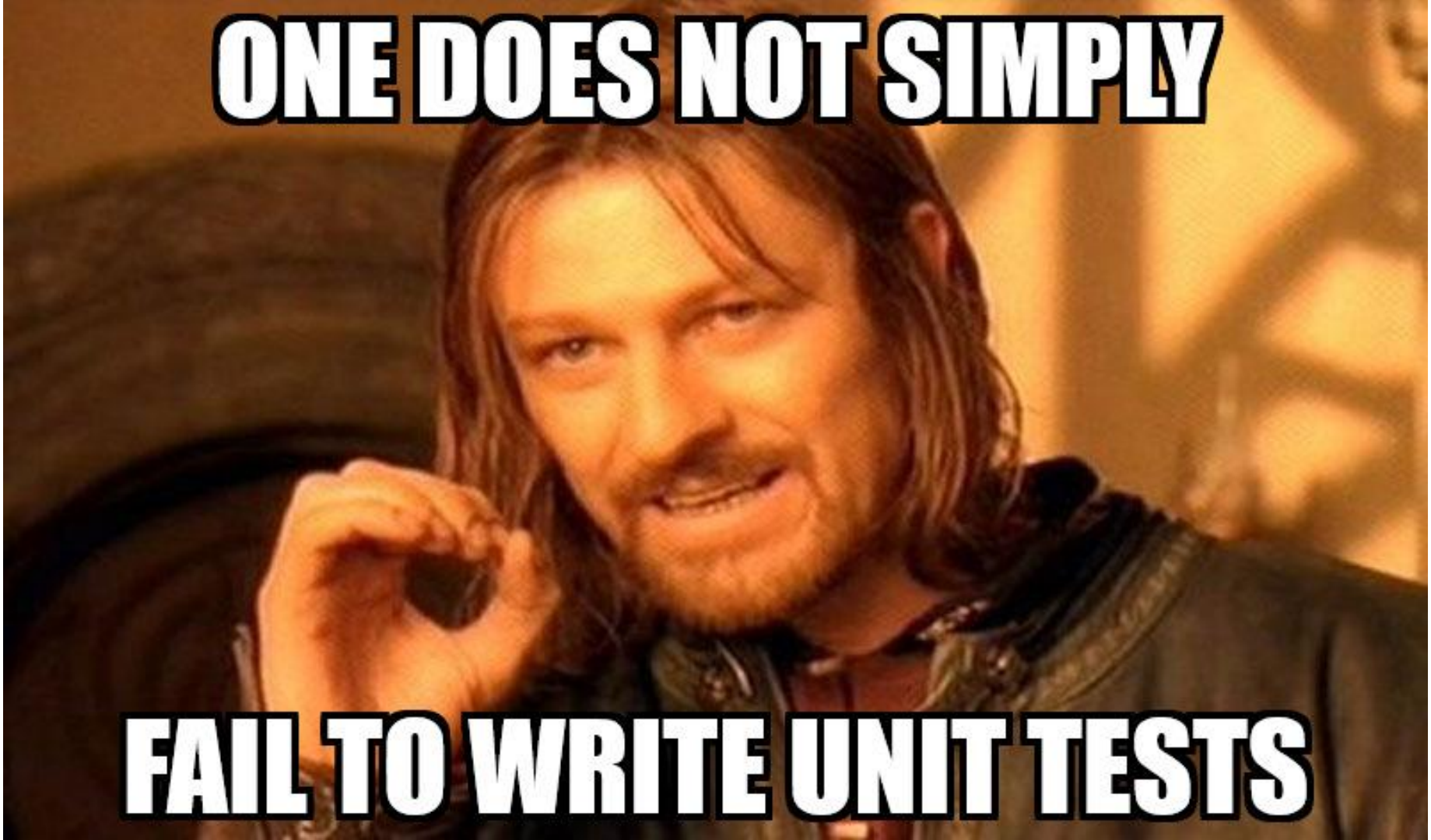
```
int[] arr = [1, 2, 3];
arr ~= [4, 5]; //Concatenate
assert(arr == [1, 2, 3, 4, 5] && arr.length == 5);

int[] aliasOfArr = arr; //Default reference semantics (for arrays)
aliasOfArr[0] = 0;
assert(arr == [0, 2, 3, 4, 5]);
int[] copy = arr.dup; //A deep copy

int[] slice = arr[1 .. $];
assert(slice == [2, 3, 4, 5]);
```

Our first D algorithm

```
T[] find(T)(T[] haystack, T needle) {
  while (haystack.length > 0 && haystack[0] != needle) {
    haystack = haystack[1 .. $];
  }
  return haystack;
}
unittest {
  int[] arr = [1, 2, 3, 4, 5];
  assert(find(arr, 2) == [2, 3, 4, 5]);
  assert(arr.find(4) == [4, 5]);
  assert(arr.find(0) == []);
}
```


ONE DOES NOT SIMPLY
FAIL TO WRITE UNIT TESTS

Our first D algorithm

```
T[] find(T)(T[] haystack, T needle) {
  while (haystack.length > 0 && haystack[0] != needle) {
    haystack = haystack[1 .. $];
  }
  return haystack;
}
unittest {
  int[] arr = [1, 2, 3, 4, 5];
  assert(find(arr, 2) == [2, 3, 4, 5]);
  assert(arr.find(4) == [4, 5]);
  assert(arr.find(0) == []);
}
```

What's the meaning of this?!

```
T[] find(T)(T[] haystack, T needle) {
  while (haystack.length > 0 && haystack[0] != needle) {
    haystack = haystack[1 .. $];
  }
  return haystack;
}
```

What's the meaning of this?!

```
T[] find(T)(T[] haystack, T needle) {
  while (!haystack.empty && haystack[0] != needle) {
    haystack = haystack[1 .. $];
  }
  return haystack;
}
```

What's the meaning of this?!

```
T[] find(T)(T[] haystack, T needle) {
  while (!haystack.empty && haystack.front != needle) {
    haystack = haystack[1 .. $];
  }
  return haystack;
}
```

Find with input ranges

```
R find(R, T)(R haystack, T needle) {
  while (!haystack.empty && haystack.front != needle) {
    haystack.popFront();
  }
  return haystack;
}
```

An input range interface for arrays

```
bool empty(T)(const(T[]) arr) {

}

ref inout(T) front(T)(inout(T[]) arr) {

}

void popFront(T)(ref T[] arr) {

}
```

An input range interface for arrays

```
bool empty(T)(const(T[]) arr) {
  return arr.length == 0;
}

ref inout(T) front(T)(inout(T[]) arr) {

}

void popFront(T)(ref T[] arr) {

}
```

An input range interface for arrays

```
bool empty(T)(const(T[]) arr) {
  return arr.length == 0;
}

ref inout(T) front(T)(inout(T[]) arr) {
  return arr[0];
}

void popFront(T)(ref T[] arr) {

}
```

An input range interface for arrays

```
bool empty(T)(const(T[]) arr) {
  return arr.length == 0;
}

ref inout(T) front(T)(inout(T[]) arr) {
  return arr[0];
}

void popFront(T)(ref T[] arr) {
  arr = arr[1 .. $];
}
```

Usage

```
unittest {
  int[] arr = [1, 2, 3, 4, 5];
  assert(find(arr, 2) == [2, 3, 4, 5]);
  assert(arr.find(4) == [4, 5]);
  assert(arr.find(0) == []);

  auto tree = redBlackTree("a", "c", "b");
  assert(!tree[].find("b").empty);
  assert(tree[].find("foobar").empty);
}
```

Generators

```
unittest {
  assert(!PowGenerator!2().find(2).empty);
  assert(!PowGenerator!2().find(4).empty);
  assert(!PowGenerator!2().find(8).empty);
  assert(!PowGenerator!2().find(512).empty);
  assert(PowGenerator!2().find(15).empty);

  assert(!PowGenerator!3().find(81).empty);
  assert(!PowGenerator!5().find(125).empty);
  assert(!PowGenerator!(5)().find(125).empty);
}
```

Pow generator

```
struct PowGenerator(ulong base) {
  bool empty() const {
    return ulong.max / base < value;
  }
  ulong front() const {
    return value;
  }
  void popFront() {
    value *= base;
  }
  private ulong value = 1;
}
```

Pow generator

```
struct PowGenerator(ulong base) if (base != 0) {
  bool empty() const {
    return ulong.max / base < value;
  }
  ulong front() const {
    return value;
  }
  void popFront() {
    value *= base;
  }
  private ulong value = 1;
}
```

An Argument for Ranges

```
vector<int> v { 3, 2, 1, 5, 4 };
sort(begin(v), end(v));
```

```
auto arr = [ 3, 2, 1, 5, 4 ];
sort(arr);
```

An Argument for Ranges

```
vector<int> v { 3, 2, 1, 5, 4 };
sort(begin(v), end(v));

vector<int> out;
copy(begin(v), end(v), begin(out));

copy_n(begin(v), 9001, begin(out));
```

```
auto arr = [ 3, 2, 1, 5, 4 ];
sort(arr);

int[] out;
copy(arr, out);

copy(take(arr, 9001), out);
```

An Argument for Ranges

```
vector<int> v { 3, 2, 1, 5, 4 };

sort(v.rbegin(), v.rend());
```

```
auto arr = [ 3, 2, 1, 5, 4 ];

sort(arr.retro);
```

An Argument for Ranges

```
vector<int> v { 3, 2, 1, 5, 4 };

sort(v.rbegin(), v.rend());

vector<int> evens;
copy_if(begin(v), end(v),
  back_inserter(evens),
  [](int x) { return x % 2 == 0; }
);
```

```
auto arr = [ 3, 2, 1, 5, 4 ];

sort(arr.retro);

auto evens =
  arr.filter!(a => a % 2 == 0));
```

An Argument for Ranges

```
string msg = "Hello world";
auto fun = string{msg.rbegin()+6,
                  msg.rend() };
fun.append(begin(msg) + 5, end(msg));


cout << fun << endl;
//olleH world
```

```
string msg = "Hello world";
auto rHello = msg[0 .. 5].retro;
auto world = msg[5 .. $];
auto fun = chain(rHello, world);


writeln(fun);
//olleH world
```

An Argument for Ranges

- Safe
- Memory Efficient
- Convenient
- Range modifiers

Why aren’t range modifiers in C++?

Implementating C++ Iterators vs. D Ranges

```
struct MyItr {
  MyItr& operator=(const MyItr&);
  MyItr& operator++();
  MyItr operator++(int);
  int operator*() const;
  int& operator*();

  friend void swap(MyItr& lhs, MyItr& rhs);
  friend bool operator==(const MyItr&, const MyItr&);
  friend bool operator!=(const MyItr&, const MyItr&);
};
```

```
struct MyItr {
  bool empty() const;
  ref inout(int) front() const;
  void popFront();
};
```

More Pow(er)!

```
unittest {
  assert(2.pow(3) == 8);
  assert(2.pow(4) == 16);
  assert(3.pow(3) == 81);

  static assert(2.pow(3) == 8);
}
```

More Pow(er)!

```
unittest {
  assert(2.pow(3) == 8);
  assert(2.pow(4) == 16);
  assert(3.pow(3) == 81);

  static assert(2.pow(3) == 8);
}

struct MyHashTable {
  enum initialSize = 2.pow(20) - 1;
  …
}
```

The C++98: TMP

```
template<unsigned long long base, size_t exponent>
struct pow {
    const static auto value = pow<base, exponent - 1>::value * base;
};

template<unsigned long long base>
struct pow<base, 0> {
    const static auto value = 1uLL;
};

static_assert(pow<2, 4>::value == 16, "2^4 is 16");
```

The C++11: constexpr

```
template<typename T, typename E>
constexpr T pow(const T& base, const E& exponent) {
  return exponent == 0 ? 1 : pow(base, exponent - 1) * base;
}
```

```
static_assert(pow(2, 4) == 16, "2^4 is 16");
```

The C++14: constexpr

```
template<typename T, typename E>
constexpr T pow(const T& base, const E& exponent) {
  auto result = T{};
  for (auto i = 0u; i < exponent; ++i) {
    result *= base;
  }
  return result;
}

static_assert(pow(2, 4) == 16, "2^4 is 16");
```

CTFE

```
auto pow(T, E)(T base, E exponent) {
  auto result = T.init;
  foreach (i; 1 .. exponent) {
    result *= base;
  }
  return result;
}
```

CTFE

```
auto pow(T, E)(T base, E exponent) {
  auto result = T.init;
  foreach (i; 1 .. exponent) {
    result *= base;
  }
  return result;
}

auto pow(T, E)(T base, E exponent) {
  return PowGenerator!base().dropExactly(exponent).front;
}
```

A Delicious Aside: The Appetizer

```
struct SillyStruct {
  int value;
  SillyStruct opBinary(string op)(SillyStruct rhs) {
    mixin("return SillyStruct(value " ~ op ~ "rhs.value);");
  }
}
unittest {
  SillyStruct x, y;
  auto a = x + y;
  auto b = x * y;
}
```

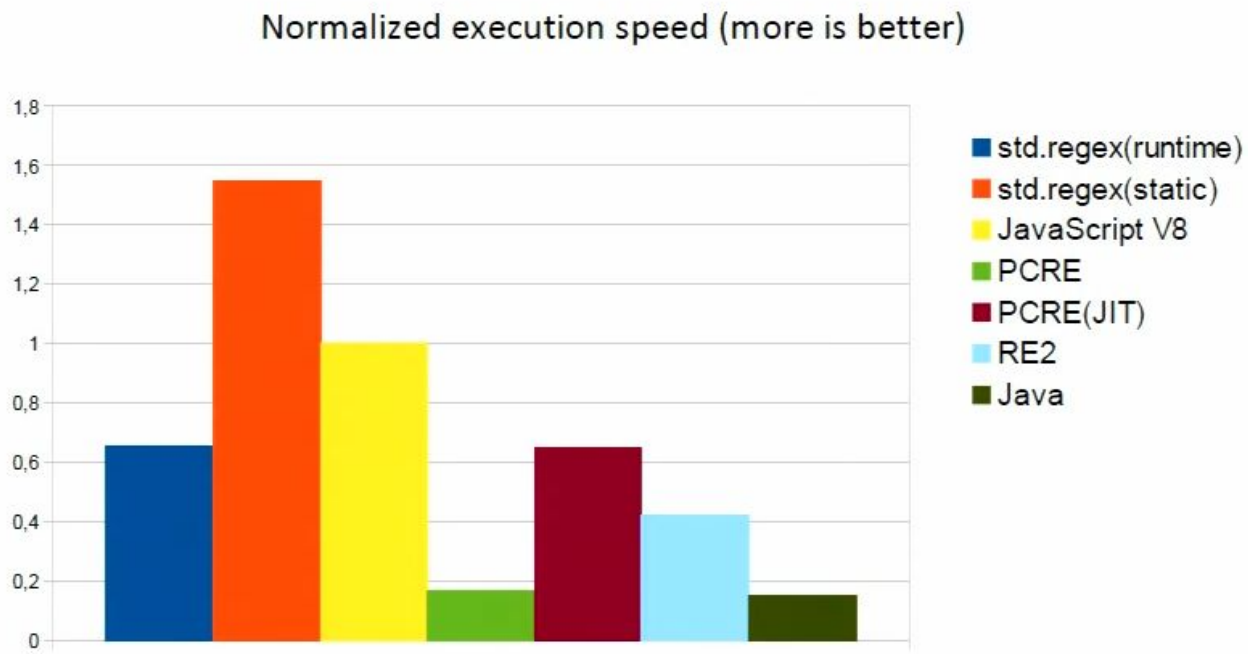
A Delicious Aside: The Main

```
unittest {
  import std.regex;
  auto r = regex("oh m(y)*");
  auto ctr = ctRegex!"oh m(y)*";

  assert("oh myyyyy".matchFirst(r));
  assert("oh myyyyy".matchFirst(ctr));
}
```

Static → highly specialized

Regex-DNA, popular benchmark



https://github.com/DmitryOlshansky/FReD/tree/master/bench

A Delicious Aside: Dessert

```
struct PhoneEntry {
  string name;
  ulong phone;
}
unittest {
  auto bob = PhoneEntry("Bob", 5);
  auto jill = PhoneEntry("Jill", 3);
  auto tree = redBlackTree!("a.name < b.name")(bob, jill);
  foreach (entry; tree) {
    writeln(entry.name, " ", entry.phone);
  }
}
```

A Delicious Aside: Dessert

```
struct PhoneEntry {
  string name;
  ulong phone;
}
unittest {
  auto bob = PhoneEntry("Bob", 5);
  auto jill = PhoneEntry("Jill", 3);
  auto tree = redBlackTree!("a.name < b.name")(bob, jill);
  foreach (entry; tree) {
    writeln(entry.name, " ", entry.phone);
  }
}
```

How does it compare?

```
{
  auto bob = make_pair("Bob"s, 5uL);
  auto jill = make_pair("Jill"s, 3uL);
  map<string, unsigned long> phoneEntries{bob, jill};
  for (auto& entry : phoneEntries) {
    cout << entry.first << " " << entry.second << endl;
  }
}
```

Credit Where Credit's Due

- The D Programming Language (book)
- http://www.informit.com/articles/article.aspx?p=1407357
- http://www.slideshare.net/andreialexandrescu1/handouts-35661779
- http://dconf.org/2014/talks/olshansky.html

Take Away

Modern language and compiler trends

- Natural and expressive code
- Fast by default
- Memory efficient by default
- Moving work to compile-time for better optimization
- Safer primitives
- Lower barrier to entry

For the curious: A migration pattern

D

- Compile-time function evaluation
- Shallow copy by default arrays
- Ranges
- Universal calling syntax (UCS)
- `unittest`
- Functional purity (`pure` keyword)
- `scope` statement (keyword)
- Variadic templates

C++

- `constexpr` (partial)
- `string_view`, `array_view`
- Proposed ranges over iterators/boost
- Proposed UCS (partial)
- Google Test macro framework[1]
- `constexpr` (partial)
- Scope guard (partial, library)
- Variadic templates (partial/convoluted)

Thanks :)

markisaa@fb.com

http://modernmaintainablecode.com