

Three Cool Things About D

Andrei Alexandrescu

Research Scientist

Facebook

What's the Deal?

PL Landscape

- Systems/Productivity languages
- Imperative/Declarative/Functional languages
- n^{th} generation languages
- Compiled/Interpreted/JITed languages
-

Sahara

- Sahara-sized desert around the crossroads of
 - High efficiency
 - Systems-level access
 - Modeling power
 - Simplicity
 - High productivity
 - Code correctness (of parallel programs, too)

Sahara

- Sahara-sized desert around the crossroads of
 - High efficiency
 - Systems-level access
 - Modeling power
 - Simplicity
 - High productivity
 - Code correctness (of parallel programs, too)

- Traditionally: focus on a couple and try to keep others under control

Sahara

- Sahara-sized desert around the crossroads of
 - High efficiency
 - Systems-level access
 - Modeling power
 - Simplicity
 - High productivity
 - Code correctness (of parallel programs, too)
- Traditionally: focus on a couple and try to keep others under control
- C++: ruler of 1-2, strong on 3, goes south after that

Sahara

- Sahara-sized desert around the crossroads of
 - High efficiency
 - Systems-level access
 - Modeling power
 - Simplicity
 - High productivity
 - Code correctness (of parallel programs, too)

- Traditionally: focus on a couple and try to keep others under control
- C++: ruler of 1-2, strong on 3, goes south after that
- *D: holistic approach toward the centroid*

Enter D

- Doesn't replace C's memory model
 - Builds *structure* on top of it
 - All of C's `stdlib` available to D
 - You can use `malloc` and `free` at full speed
 - `alloca` too if you really wish
 - Garbage collection fostered but not required
- Layered approach to safety
- Easy and powerful generic and generative capabilities
- Principled yet practical approach to correctness
- Threading without races

Compilation model

- Compiled language, JITable subset
- Context-independent declarations
- All top-level declarations are conceptually entered in parallel
- Reader needs minimal context
- Each character is looked at *once*
- Each symbol is loaded *once*

- Fastest language to compile by a large margin

This Talk

- Introduction
- Hello, Correctness
- Not one more gorram handwritten search, linked list, nearest neighbor, merge... *Ever.*
- Actors with benefits

- Conclusions

Hello, Correctness

Correctness Starts With...

```
import std.stdio;
void main()
{
    writeln("Hello, world!");
}
```

- This program is *correct*.

Hello, World! Do You Copy? Over.

- C's hello world is *incorrect*:

```
#include <stdio.h>
main()
{
    printf("hello, world\n");
}
```

- What does the program return in case of success?
- What does the program return in case of failure?

The Greeting that Always Befriendeth

- C++'s hello world is *also incorrect*:

```
#include <iostream>
int main()
{
    std::cout << "Hello, world!\n";
}
```

- 20 years of progress
 - Develop a positive attitude
 - Learn more quirks in day one
 - More trouble with multithreaded I/O



**EVERYTHING
WENT
BETTER
THAN
EXPECTED**

A Litmus Test

- Read tutorial examples
 - Engineered to make the language look good
- Consider various production scenarios
 - I/O may fail
 - Input may be unreasonably large
 - API calls (incl. allocating memory) may fail
 - Input may actually contain non-ASCII characters
- What does it take to make the examples ready for production?

echo in 9 correct lines

```
#!/usr/bin/env rdm  
import std.getopt, std.stdio, std.string;  
void main(string[] args) {  
    bool noNewline;  
    getopt(args, config.passThrough,  
           config.stopOnFirstNonOption,  
           "n", &noNewline);  
    write(join(args[1 .. $], " "),  
          noNewline ? "" : "\n");  
}
```

Past approaches to correctness

- Design and implementation validation: contracts
 - Ubiquitous as idiom (`assert!`)
 - Deserve better language support
- Error handling:
 - 40 years anniversary of the MACeAEC (Movement Against Checking `errno` And Error Codes)
 - Propagating errors manually is relentless friction even for seasoned professionals
- `try/catch`, `using`, `unwind-protect` will never scale
 - (beyond cutesy examples)
 - They all affect the normal code path

Approach to correctness

- Design/implementation validation:
 - Full-bore contracts
 - Built-in `unittest`
- Error handling:
 - Modern exception handling
 - Can `throw` from *anywhere* without loss of info
 - Destructors may `throw`!
 - Writing transactional code is easy and linear
 - No need to define types
 - No need to `try`
 - Proper sequencing, not flow, determines correctness

Transactional file copy in 9 correct lines

```
#!/usr/bin/env rdmd
import std.exception, std.file;
void main(string[] args) {
    enforce(args.length == 3,
        "Usage: trcopy file1 file2");
    auto tmp = args[2] ~ ".messedup";
    scope(failure) if (exists(tmp)) remove(tmp);
    copy(args[1], tmp);
    rename(tmp, args[2]);
}
```

Truth in advertising

- The `std.file.copy` routine itself:
 - Control flow: one `if` and one `for` in 47 lines (!)
 - Either succeeds or fails successfully
 - Makes 6 calls to `enforce`
 - Has 4 `scope` statements
 - No extra aides: uses straight `fcntl`, `stat`, and even `malloc`
- Replaces a function:
 - 105 lines long
 - 9 `gotos` to 5 labels
 - 11 other control flow statements

Results may vary. Combine with diet and exercise for best results.

Genericity

What is Generic Programming?

What is Generic Programming?

- Find the representation of an algorithm using the narrowest requirements possible
 - Big- $O()$ encapsulation should be a crime in 48 states
 - Low-level efficiency important too: reduces impetus for regressing to hand-written code

What is Generic Programming?

- Find the representation of an algorithm using the narrowest requirements possible
 - Big- $O()$ encapsulation should be a crime in 48 states
 - Low-level efficiency important too: reduces impetus for regressing to hand-written code
- Define types to implement said requirements

What is Generic Programming?

- Find the representation of an algorithm using the narrowest requirements possible
 - Big- $O()$ encapsulation should be a crime in 48 states
 - Low-level efficiency important too: reduces impetus for regressing to hand-written code
- Define types to implement said requirements
- Leverage the algorithm for ultimate reuse

What is Generic Programming?

- Find the representation of an algorithm using the narrowest requirements possible
 - Big- $O()$ encapsulation should be a crime in 48 states
 - Low-level efficiency important too: reduces impetus for regressing to hand-written code
- Define types to implement said requirements
- Leverage the algorithm for ultimate reuse
- ...
- Profit!

What is Generic Programming?

- Find the representation of an algorithm using the narrowest requirements possible
 - Big- $O()$ encapsulation should be a crime in 48 states
 - Low-level efficiency important too: reduces impetus for regressing to hand-written code
 - Define types to implement said requirements
 - Leverage the algorithm for ultimate reuse
 - ...
 - Profit!
-
- Arguably one of the noblest endeavors of our métier

Generic Programming

- “Write once, instantiate anywhere”
 - Implementations tailored for enhanced interfaces are welcome
- Prefers static type information and static expansion (macro style)
- Fosters strong mathematical underpinnings
- Has tenuous relationship with binary interfaces
- Starts with algorithms, not interfaces or objects
- “Premature encapsulation is the root of all evil”

D: Generic Programming Democratized

- Accessible and powerful generic programming capabilities

```
auto min(L, R)(L lhs, R rhs) {  
    return rhs < lhs ? rhs : lhs;  
}  
auto a = min(x, y);
```

- This function is as efficient as any specialized, handwritten version (true genericity)

(Compare At

```
template <class T>
T& min(T& lhs, T& rhs) {
    return rhs < lhs ? rhs : lhs;
}

template <class T>
const T& min(const T& lhs, const T& rhs) {
    return rhs < lhs ? rhs : lhs;
}
```

- Function is incomplete
- Revised implementation, rejected proposal N2199: 175 lines, 10 types, 12 specializations)

Iterators and Ranges

- Iterators: very useful primitives for many algorithms
 - Flexible but fundamentally uncheckable
 - D's standard library introduces ranges: abstracted pairs of iterators
 - Defines superset of STL on top of ranges
 - Ranges pervade D: I/O, lazy evaluation, random numbers...
-
- google terms: "On Iteration", std.algorithm

Containers

- No container hierarchy
 - Federation of independent containers
- Containers have too much “personality”
- Futile to classify in formal interfaces
- Yet: duck typing should work
- Efficient allocation with `malloc` possible without affecting safety (upcoming article: “Sealed Containers”)
 - Trickier & more interesting than it may seem

Algorithm example: edit distance

```
size_t levenshteinDistance
    (alias equals = "a==b", R1, R2)
    (R1 s, R2 t)
    if (isForwardRange!R1 && isForwardRange!R2)
    {
        ...
    }
```

- Usual assumptions: strict equality, inputs must offer random access
- Yet algorithm has modular comparison and requires only forward access for both inputs

Edit Distance (continued)

- D strings are array of immutable characters:

```
alias immutable(char) [] string;  
alias immutable(wchar) [] wstring;  
alias immutable(dchar) [] dstring;
```

- `string`, `wstring` formally offer only bidirectional range interface
- Consequence: `levenshteinDistance` works correctly for all strings with custom comparison—no copy, no handcrafting!
- Similar discussion: Boyer-Moore with custom predicate

Compile-Time Evaluation

- Classic programming example: factorial function

```
ulong factorial(uint n) {  
    ulong result = 1;  
    foreach (i; 2 .. n) result *= i;  
    return result;  
}  
  
...  
auto f = factorial(10);
```

Compile-Time Evaluation

- Classic *metaprogramming* example: factorial template

```
template factorial(uint n) {  
    static if (n <= 1)  
        enum ulong factorial = 1;  
    else  
        enum ulong factorial =  
            n * factorial(n - 1);  
}  
auto f = factorial!(10);
```

- BTW: `static if` == awesome

Forget What You Just Saw

- If a regular function is evaluated in a compile-time context, the compiler attempts immediate interpretation:

```
// evaluate at runtime  
auto f1 = factorial(10);  
// evaluate at compile time  
enum f2 = factorial(10);  
// only one function for both!  
ulong factorial(uint n) {  
    ... as above ...  
}
```

- Doing compile-time tricks without even learning them.

**Message Passing +
Immutability = No Races**

Concurrent Programming

- Today: several competing approaches
- Share-intensive (C, C++, Java) vs. no sharing and message passing (CSP/Actor)
- PGAS fit for HPC fork-join parallelism
- No definitive solution in sight
- On older machines, shared memory was faster
- Key trend: on newer machines, passing messages becomes faster and more scalable than shared memory

Concurrent Programming

- Today: several competing approaches
- Share-intensive (C, C++, Java) vs. no sharing and message passing (CSP/Actor)
- PGAS fit for HPC fork-join parallelism
- No definitive solution in sight
- On older machines, shared memory was faster
- Key trend: on newer machines, passing messages becomes faster and more scalable than shared memory

- Traditional languages placed a bet on shared memory
- D places a bet on message passing and hedges it with explicit sharing

Actor with benefits

- Basic tenet:

Memory is thread-private by default, shared on demand.

- Even statics and globals are allocated per-thread
 - Most of them were intended that way anyway!
- Shared is explicit:

```
int threadLocal;  
shared int global;
```

Key Points

- `shared` is *transitive*
- `immutable` is transitive too
- `immutable` is implicitly `shared`

- All of these are *consequences*, not decisions!

```
shared int * pInt; // pointee shared as well
string name;      // share at no cost
```

Actor with benefits

- Flagship approach: isolated threads, asynchronous messages
 - Safe, modular, maintainable programs that are easy to understand
- `shared` variables, lock-free code, old-school critical sections
- Non-`@safe` code may use `casts` for unchecked data sharing
- `asm` statements for ultimate control
- Miniature soldering iron (not included)

Reduced Sharing == Increased Safety

- Process-level concurrency with OS-grade isolation: safe and robust, but heavyweight
- Thread-level concurrency with shared memory: fast, but fraught with peril

Conclusion

Conclusion

- Q: System-level language without the agonizing pain?

Conclusion

- Q: System-level language without the agonizing pain?
- Q: Application language without the boredom?

Conclusion

- Q: System-level language without the agonizing pain?
- Q: Application language without the boredom?
- Q: Principled language without the high-brow snobbery?

Conclusion

- Q: System-level language without the agonizing pain?
- Q: Application language without the boredom?
- Q: Principled language without the high-brow snobbery?

Conclusion

- Q: System-level language without the agonizing pain?
- Q: Application language without the boredom?
- Q: Principled language without the high-brow snobbery?

- $D = \sum_i w_i q_i$

Conclusion

- Q: System-level language without the agonizing pain?
- Q: Application language without the boredom?
- Q: Principled language without the high-brow snobbery?

- $D = \sum_i w_i q_i$

Grill the Speaker!