MCours.com

# A (System) Programming Language For Our Time

# The D Language

CHUCK ALLISON — UTAH VALLEY UNIVERSITY
WHITE MALE COMPUTER SCIENTIST

# Overview

- What is D?
- Why Might You Care?
- An Invitation to D (Relevant Examples)

# What is D?

- A system programming language conceived in 1996 as a *replacement* for C++
  - Compiles to native code
- Combines the *power* of C++ with the *usability* of languages like Java, C# and Python
  - Garbage collected by default
- Has a robust library and tool support, runs on most platforms of interest

"Often, programming teams will resort to a hybrid approach, where they will mix Python and C++, trying to get the productivity of Python and the performance of C++. The frequency of this approach indicates that there is a large unmet need in the programming language department."

"D intends to fill that need. It combines the ability to do low-level manipulation of the machine with the latest technologies in building reliable, maintainable, portable, high-level code. D has moved well ahead of any other language in its abilities to support and integrate multiple paradigms like imperative, OOP, and generic programming."

— *Walter Bright*, Co-designer of D

# Why Might You Care?

- It has all the *power* of C++ but is *easier* to learn and use
  - Clean, C-style syntax
  - Explicit pointers available, but only needed for to-the-metal access
  - Suitable for CS1 and CS2
- It is highly suitable for an Analysis of Programming Languages course
  - Multi-paradigm (imperative, OO, functional)
  - Supports most parameter-passing mechanisms (value, result, reference, lazy evaluation)
  - Language Support for Software Engineering

# “Modern” Languages

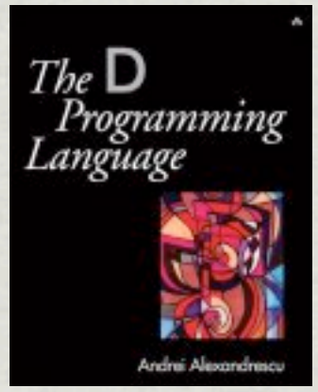
*(Appearing in the last 15 Years)*

- Java
- PHP
- C#
- Ruby
- JavaScript
- MATLAB
- Lua
- Alice
- D
- ActionScript
- Visual Basic .NET
- Haskell

**SOURCE: TIOBE.COM, SEPTEMBER 2010**

# D Reference Book

- Andrei Alexandrescu
- Addison-Wesley, 2010



# D Programming Examples

# Hello, D

```
#!/usr/local/bin/rdmd

import std.stdio;

void main(string[] args) {
    if (args.length > 1)
        foreach (a; args[1..$])
            writeln("Hello " ~ a);
    else
        writeln("Hello, Modern World");
}
```

```
$ dmd hello.d
$ ./hello john jane
Hello john
Hello jane
$
```

```
$ chmod u+x hello.d
$ ./hello.d
Hello, Modern World
$
```

# Word Count in C++

```
void wc (const char* filename) {
    ifstream f(filename);
    string word;
    map<string,int> counts;
    while (f >> word)
        ++counts[word];
    map<string,int>::iterator p = counts.begin();
    while (p != counts.end()) {
        cout << p->first << ": " << p->second << "\n";
        ++p;
    }
}
```

```
But,: 1
Four: 1
God,: 1
It: 3
Liberty,: 1
Now: 1
The: 2
We: 2
a: 7
…
who: 3
will: 1
work: 1
world: 1
years: 1
```

# Associative Arrays

```
void wc(string filename) {
    string[] words = split(cast(string) read(filename));
    int[string] counts;
    foreach (word; words)
        ++counts[word];
    foreach (w; counts.keys.sort)  // Array properties
        writefln("%s: %d", w, counts[w]);
}
```

# Qualifiers

- For *function parameters*:
  - **in | out | inout**
  - **ref**
  - **lazy**
- *General* declaration qualifiers:
  - **const**
  - **immutable** (e.g., **string** is **immutable(char)[ ]**)

# Lazy Parameters

```
// lazyvoid.d
import std.stdio;

void f(bool b, lazy void g) {
    if (b)
        g();
}

void main() {
    f(false, writeln("executing g"));
    f(true, writeln("executing g"));
}
```

```
executing g
```

# Closures

- *Nested* and *higher-level* functions
- Nested functions are returned as (dynamic) *closures*
  - aka “delegates” (a code-environment pair)
    - The referencing environment could be a *function*, *class*, or *object*
    - Escaped activation records are moved from the stack to the garbage-collected heap
- *Plain* function pointers also supported:
  - `int function(int) f;` (vs. “`int (*f)(int);`” in C++)

# Higher-Level Functions and Closures

```
// gtn.d
import std.stdio;

bool delegate(int) gtn(int n) {
    bool execute(int m) {
        return m > n;
    }
    return &execute;
}

void main() {
    auto g5 = gtn(5); // Returns a ">5" delegate; infers type
    writeln(g5(1));   // false
    writeln(g5(6));   // true
}
```

# Lambda Expressions

```
// gtn2.d: Anonymous function with the delegate keyword
auto gtn(int n) {
    return delegate bool(int m) {return m > n;};
}
```

```
// gtn3.d: The delegate keyword isn't really needed
auto gtn(int n) {
      return (int m) {return m > n;};
}
```

# Environments Are Objects

```
void main() {
    class A { int fun() { return 42; } }
    A a = new A;
    auto dg = &a.fun;  // A "bound method"
    writeln(dg());     // 42
}
```

There is no
"Objects are a poor man's closures"
vs.
"Closures are a poor man's objects"
debate.

They are *unified* in D.

# Parametric Polymorphism

```
// gtn4.d
import std.stdio;

auto gtn(T)(T n) {
    return (T m) {return m > n;};
}

void main() {
    auto g5 = gtn(5);
    writeln(g5(1));         // false
    writeln(g5(6));         // true

    auto g5s = gtn("baz");
    writeln(g5s("bar"));  // false
    writeln(g5s("foo"));  // true
}
```

# Compile-Time Constraints

```
// gtn5.d
import std.stdio, std.traits;

auto gtn(T)(T n) if (isNumeric!T) {
    return (T m) {return m > n;};
}

void main() {
    auto g5 = gtn!int(5);
    writeln(g5(1));
    writeln(g5(6));

    auto g5s = gtn!string("baz");   // Error
    writeln(g5s("bar"));
    writeln(g5s("foo"));
}
```

# Referential Transparency via Pure Functions

```
// fib.d: Mutable locals are okay
import std.stdio, std.conv;

pure ulong fib(uint n) {
    if (n == 0 || n == 1) return n;
    ulong a = 1, b = 1;
    foreach (i; 2..n) {    // .. is exclusive of n
        auto t = b;
        b += a;
        a = t;
    }
    return b;
}

void main(string[] args) {
    if (args.length > 1)
        writeln(fib(to!(uint)(args[1])));
}
```

# Program Correctness and Software Engineering

- Resource Management with the **scope** statement
  - **scope(exit | success | failure)**
  - No need for **try-catch-finally**
- Contract Programming:
  - Pre-conditions (enforced *contravariance*)
  - Post-conditions (enforced *covariance*)
  - Class Invariants
- Software Engineering Support
  - **-unittest**, -**debug**, -**release**, -**version**, -**profile** compiler options

# The **scope** Statement

```
void g() {
    risky_op1();
    scope(failure) undo_risky_op1();
    risky_op2();
    scope(failure) undo_risky_op2();
    risky_op3();
    writeln("g succeeded");
}
```

# Preconditions, Postconditions and Class Invariants

```
// rational.d: Shows class-based contract programming
struct Rational {
    private int num = 0;
    private int den = 1;

    // Class invariant
    invariant() {
       assert(den > 0 && gcd(num, den) == 1);
    }
...
```

# Preconditions, Postconditions and Class Invariants

*Continued*

```
    // Constructor
    this(int n, int d = 1)
    // Constructor precondition
    in {
        assert(d != 0);
    }
    body { // Establishes class invariant
        num = n
        den = d;
        auto div = gcd(num, den);
        if (den < 0)
          div = -div;
        num /= div;
        den /= div;
    }

    Rational opBinary(string op)(Rational r) if (op == "+") {
        return Rational(num*r.den + den*r.num, den*r.den);
    }
} // End of struct Rational
```

# Unit Testing

```
unittest {
    auto r1 = Rational(1,2), r2 = Rational(3,4), r3 = r1 + r2;
    assert(r3.num == 5 && r3.den == 4);
}
```

# Summary

- I like it, so it must be good :-)
- Have used it for years to illustrate *parameter passing* mechanisms, *nested functions* and *closures* in a Programming Languages class
- Robust, fast, fun, safe
- Growing user base
- *Acknowledgements*: Thanks to Andrei Alexandrescu and Neil Harrison for their helpful comments