D: A PROGRAMMING LANGUAGE FOR OUR TIME*

Charles D. Allison
Utah Valley University
Orem, UT 84058
801-863-6389
chuck.allison@uvu.edu

ABSTRACT

The D Programming Language is a hybrid of C++ and modern scripting languages: it compiles statically to native code, but is also garbage collected. It is a multi-paradigm language, supporting imperative, object-oriented and functional programming styles, and has a number of software engineering features built-in. This paper explores how D is suitable for courses in the CS curriculum as well as for the workplace.

INTRODUCTION

Modern, general-purpose programming languages in widespread use include Java, PHP, Python, C#, and Ruby.[1] All of these are interpreted languages, although Java and C# have a compile step that transforms source code to byte-code. The other three languages are dynamically typed and are considered to be "scripting languages", since they excel at quickly crafting small software solutions. All have varying degrees of support for the functional programming paradigm, which is enjoying a resurgence as software developers are rediscovering the utility of higher-level functions and applicative programming.

While programs written in C++ tend to have superior performance, C++ programmers must manage memory manually, which is tedious and error prone. C++ is also known for its complexity. The popularity of languages such as Java and C# are due in part to being more accessible to the average programmer than is C++, which is fraught with arcane syntax rules and pointer "gotchas". It is not uncommon for software projects to use a scripting language such as Python whenever possible and C++ only when necessary.

THE D PROGRAMMING LANGUAGE

While it is unreasonable to expect a single programming language to be ideal for all purposes, it is not unreasonable to envision a language that has the power of a language like C++ and the ease of use of a language like Python. Walter Bright, chief architect and implementer of the D programming language, explained how such a vision led to the creation of this new language:

"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." [2]

D combines many modern language features with the efficiency of C++. D has a familiar, C-like syntax and has language support for garbage collection, higher-level functions (including lambda expressions), generics, compile-time meta-programming, concurrency, unit testing, and programming by contract.

HELLO, D

The following D program serves as a point of departure for discussing the structure and features of the D language.

```
// hello.d: Greet the user or the entire world
import std.stdio;
void main(string[] args) {
    if (args.length > 1)
        foreach (a; args[1..$])
            writeln("Hello " ~ a);
    else
        writeln("Hello, Modern World");
}
```

If command-line arguments are present, each argument to this program is greeted on a line by itself. Otherwise the string "Hello, Modern World" is displayed. For those that prefer a graphical development environment, IDEs are available for D and there is a D plugin for the Eclipse editing system.

As the dot between **std** and **stdio** above suggests, modules can be grouped hierarchically. The variable **args** is a *dynamic array* of strings. The ".." notation denotes an array slice, a contiguous subset of an array inclusive of the first position and exclusive of the last. The dollar sign is an abbreviation for the **length** property of an array, so the expression **args[1..$]** is equivalent to **args[1..args.length]** and represents the elements in positions **1** through **args.length-1** (**args[0]** is the program name.) Slices are *not* copies; they represent a mutable reference range of the original array, and are therefore very efficient. The type of the iteration variable, **a**, in the **foreach** loop is deduced by its context. The ~ operator is the array *concatenation* operator.

D also supports *associative arrays*, also known as maps, dictionaries, or hash tables. Associative arrays use normal array syntax, even in their declaration, as the following program illustrates.

```
// wc.d: Displays the word count of a text file
void wc(string filename) {
    string[] words = split(cast(string) read(filename));
    int[string] counts;
    foreach (word; words) ++counts[word];
    foreach (w; counts.keys.sort) writefln("%s: %d", w, counts[w]);
}

void main(string[] args) {
    if (args.length == 2) wc(args[1]);
```

The **foreach** construct supports a special form for associative arrays, where the iteration variables represent the key and value, respectively. Associative arrays have a **keys** property that returns a new dynamic array of the keys of each association. The **sort** property sorts an array in place (using <) and returns the array by reference.

FUNCTIONS AND PARAMETERS

Functions can be defined at file scope, in classes, and inside other functions in D. Some functions can even execute at *compile time*. Consider the following program adapted from the documentation on the official D website (digitalmars.com):

```
// ctfe.d: Illustrates compile-time function execution
int square(int i) { return i * i; }

void main() {
    static int n = square(3);       // compile time execution
    writefln(text(n));
    writefln(text(square(4)));      // runtime execution
}
```

The first call to **square** actually runs at compile time because the function argument is a *literal*, and the result is being used to initialize a static integer, hence **n** is initialized to 9 before **main** runs.

Function parameters can have the following attributes:

in	(read-only copy of the calling argument)
out	(write-only lvalue referring to the calling argument)
ref	(pass by reference)
lazy	(argument evaluated on demand only in called function)
const	(locally read-only, as in C++, but fully transitive ("deep"))
immutable	(argument allows no changes anywhere, once initialized)

A function parameter defined without one of these attributes defaults to pass-by-value.

The lazy storage class defers evaluation of an argument until it is actually used in the called function, and the argument is evaluated on each access, as the following program illustrates.

```
void printif(bool b, lazy string s) {
    if (b) {
        writeln(s);
        writeln(s);
    }
}

string f(string s) {
    writeln("f called");
    return s;
}

void main() {
    writeln("first call to printif...");
    printif(false, f("this won't print"));
    writeln("second call to printif...");
    printif(true, f("this will print"));
}

/* Output:
first call to printif...
second call to printif...
f called
this will print
f called
this will print
*/
```

The first call to **printif** shows that the call to **f** in its second argument is not evaluated, since the first argument is **false**, while second call to **printif** evaluates **s** each time it is accessed.

HIGHER-ORDER FUNCTIONS AND FUNCTIONAL PROGRAMMING

Functions can be passed to and from other functions in D, and functions can be defined anonymously in lambda expressions. The following example defines a function, **gtn**, which returns an anonymous function closure.

```
// gtn.d: Creates a "greater-than-n" function
auto gtn(int n) {
    return (int m) {return m > n;}; // (parm list) + {body}
}

void main() {
    auto g5 = gtn(5);    // Returns the "> 5" function
    writeln(g5(1));      // false
    writeln(g5(6));      // true
}
```

The return type of **gtn** is **bool delegate(int),** a closure that takes an **int** and returns **bool**, but it's not necessary to declare it as such, since the **auto** keyword infers the type of **g5** from its intitializer. A delegate holds a pointer to the function to execute, as well as a pointer to the execution environment for the function. In this case the execution environment is the current activation record for **gtn**, which is automatically moved from the stack to the garbage-collected heap so that n can be accessed later.

Applicative programming is enforced in D with *pure* functions, which must have parameters non-mutable parameters, and must neither read nor write any non-local, mutable state nor call an "impure" function. The following iterative version of a Fibonacci number function is a pure function.

```
pure ulong fib(const uint n) {       // Note pure keyword
    if (n < 2) return n;
    ulong a = 1, b = 1;
    foreach (i; 2..n) {          // Reminder: exclusive of n
        ulong t = b;
        b += a;
        a = t;
    }
    return b;
}
```

D'S SOFTWARE ENGINEERING SUPPORT

D provides language features to ensure code reliability. Suppose a sequence of three operations must succeed or fail together. A typical **try-catch-finally** approach that achieves rollback semantics is somewhat complex, but D offers a simpler alternative via its **scope** statement:

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

The **scope** statement activates a code block that may or may not run when a scope is exited. The three scope-guard options are:

`scope(exit)`	the code *always* runs (like `finally`)
`scope(failure)`	the code runs *only* if an exception occurs
`scope(success)`	the code runs only if *no* exception occurs

When execution leaves a scope, all scope-guard blocks that have executed are visited in last-in-first-out order, so transactions roll back gracefully.

Programming by contract[3,4] is supported in D with *class invariants*, and function *preconditions* and *postconditions*. The following example, which represents the beginnings of a value type that simulates rational number, illustrates a class invariant and a method precondition, as well as unit testing and operator overloading.

```
struct Rational {
    int num = 0, den = 1;

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

    this(int n, int d = 1)     // Constructor
    in {
        assert(d != 0);        // Pre-condition
    }
    body {
      num = n;
      den = d;
      int 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);
    }
}

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

Assertions are placed in **in** and **out** blocks, respectively, for preconditions and postconditions. When either or both of these are present, the function body must itself appear in a **body** block. The separation of contract conditions from the body of a function allows the compiler to combine them properly in inheritance hierarchies. Since preconditions are *contravariant* (i.e., they can be *weakened* in subclass methods) and the invariants and postconditions are *covariant* (they can be *strengthened* in subclass methods), D automatically checks that *at least one* of the applicable preconditions is met and that all of the applicable invariants and postconditions are satisfied when dispatching polymorphic methods.

CONCLUSION

The D programming language combines many valuable and popular features from both classic and modern languages. Its design emphasizes a clear, high-level, familiar C-like syntax as well as pragmatics important to effective software development. It also offers robust support for the imperative, object-oriented, and functional programming paradigms. The author has used it with favorable results for years in an upper-division course on the analysis of languages to illustrate important programming constructs in a modern, strongly typed language.

REFERENCES

[1] The Tiobe Index, http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, retrieved June 2010.

[2] Bell, K., Igesund, L., Kelly, S., Parker, M., *Learn to Tango with D*, aPress, 2007, Foreword.

[3] Parnas, D., A Technique for Software Module Specification with Examples, *CACM*, 15(5), May 1972.

[4] Meyer, B., Applying "Design by Contract", *Computer (IEEE)*, 25(10), October 1992, pp. 40–51.