D: A Programming Language for Our Time

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

Summer 2010

Abstract

Modern programming languages such as C#, Ruby and Java have much in common: they support (and some *require*) object-oriented programming, have large, useful libraries, manage memory via garbage collection, and to a large degree shield the programmer from low-level details. They also tend to be interpreted languages, so while they are applicable to most programming problems, they lack the efficiency of systems programming languages like C and C++. The D Programming Language is a modern hybrid of C++ and languages like Ruby: it compiles statically to native code, but is also garbage collected. It inherits the C and C++ libraries, but has its own modern library. Like C++, it is a multi-paradigm language, supporting imperative, object-oriented and functional programming styles. Its pure functional subset is suitable for teaching the functional paradigm in a survey of languages course. It also supports delegates and anonymous functions, and has a number of software engineering features built-in. This paper explores how D is suitable for courses at various levels in the CS curriculum as well as in the workplace.

Introduction

Computer Science curricula seek to ground students in the fundamentals of computation while also preparing them for graduate study as well as to make meaningful contributions to current computing problems in the workplace. But traditional CS curricula have recently proven unattractive to many of today's students resulting in falling CS enrollments, while the need for competent software professionals is increasing.[1] Feedback from industry indicates that areas such as software security, parallelism, quality, performance, and software engineering best practices need more emphasis in CS-related degree programs, while not neglecting the foundations of computing. The ACM/IEEE Interim CS Curriculum Report includes the following statement from an industrial commentator:

"The thing that we can't afford to do [on the job] is teach candidates how to think critically, to be effective problem solvers, and to have basic mastery of programming languages, data structures, algorithms, concurrency, networking, computer architecture, and discrete math / probability / statistics. I can't begin to emphasize the importance of algorithms and data structures to the work that we do here … With multi-terabyte disks, bigger broadband pipes, etc., on the way, the big data problems demand that these skills … are quickly going to be in need in a huge number of programming contexts."[2].

Programming languages play a crucial role in modern computation. Modern, general-purpose programming languages in widespread use include

- Java
- PHP
- Python
- C#
- Ruby. [3]

All of these are interpreted languages, although Java and C# are statically typed and 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. Both Java and C# employ a Just-in-Time (JIT) compilation strategy, which dynamically converts byte-code instructions to native machine code to improve runtime performance. The scripting languages are popular for their ease of use but do not perform as well as Java or C# at runtime.

All of these languages are known for being "programmer friendly" by virtue of a high-level syntax and by supporting such powerful features as:

- Garbage collection
- Object-orientation
- Modules
- Unit Testing
- Robust Runtime Libraries

PHP, Python, Perl are *multi-paradigm* languages, while Java and C# force adherence to the object-oriented paradigm. All of these, as well as other emerging languages (e.g., Scala, Groovy, Lua) have varying degrees of support for the *functional* programming paradigm, which is enjoying somewhat of a resurgence as software developers are rediscovering the utility of closures, higher-level functions, and applicative programming.

What About C++?

The C++ programming language, first released to the public in 1985, has its roots in the C language, which become popular in the 1970s and remains in widespread use to this day. These two systems programming languages differ markedly from the five languages mentioned earlier in that they:

- Support direct access to machine addresses via pointers
- Compile entirely to native executable code before runtime, and
- Do not support garbage collection.

The consequences of these differences mean that C and C++ allow direct access to machine components and have better runtime performance, but also require programmers to do their own manual memory management, which is tedious and error prone. While Java and C# have adequate performance, C and C++ are acknowledged as having the best performance of all high-

level languages and are crucial elements in the development of certain types of software systems. Embedded systems, for example, require close attention to low-level details, which these languages allow, and most embedded software is written in C. It is also common for compilers, operating systems, and similar systems to be written in C or C++. The following is a sample list of applications developed in C++:

- Sun's HotSpot Java Virtual Machine
- Microsoft's C# Compiler
- All versions of Microsoft Windows
- All major Adobe products
- Amazon's large-scale e-commerce systems
- Ericsson's cell phone network software
- Google's search engine and Map-Reduce cluster data processing software
- Firefox browser
- NASA's Mars rover autonomous driving system
- Nokia mobile communications software
- U.S Air Force Joint Strike Fighter control systems[4]

C++ programs are also known for their 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 what led to the creation of this new programming language:

"Amazingly, there is no language that enables precise control over execution while offering modern and proven constructs that improve productivity and reduce bugs. What first comes to mind is C++, but C++ remains mired in its need to be backward-compatible with obsolete, decades old decisions. This makes it very difficult for C++ to adopt new ideas. Then there are the languages that compile to virtual machines (VMs) such as Java and C#. But those languages never seem to quite be able to shake off their performance problems, and of course a VM-based language cannot offer to-the-metal capability.

"And lastly, there are the scripting languages such as Perl, Ruby, and Python. These offer many advanced programming features and are highly productive, but have poor runtime performance compared with C++. They also suffer from lingering doubts about their suitability for very large projects.

"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." [5]

D combines many modern language features with the power of C++. D has a C-like syntax (somewhat cleaner than C++), so it is easily grasped by C++, Java, and C# programmers, and supports important language features such as

- Garbage collection
- Function and Operator Overloading
- Object-oriented Programming (single inheritance)
- Deterministic destruction of objects
- Properties
- Inner classes
- Function closures (via delegates)
- Powerful, built-in data structures
- Generic functions and classes
- Functional Programming via nested, higher-order, anonymous, and "pure" functions (no side effects)
- Concurrency via critical sections and message passing
- Contract Programming
- Unit Testing
- Modules
- Documentation Comments
- Compile-time: function execution, assertions, and metaprogramming

Like C++, D is a lexically scoped language that compiles to native code and is available on most major computing platforms. Since it supports multiple paradigms, it is suitable for study in an analysis of programming languages course, and will suffice for most any programming challenge. Like C#, it supports all of the classical parameter-passing mechanisms (by value, by result, by value-result, and by reference) and distinguishes between *value types* (scalar types and **struct**s) and *reference types* (objects on the heap). Like Java, it has labeled **break** and **continue**, static initializers, synchronized code for critical sections, anonymous inner classes, and delegating constructors.

NOTE: This article introduces D 2.0, which, while widely available and quite stable, is still under development at this time. All code samples have been tested with the latest version of D (2.047).

Hello, D

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

```
// hello.d: Greet the users 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");
}
```

To compile this program from the command line, enter the following command:

```
$ dmd hello.d
```

If command-line arguments are present, each argument is greeted on a line by itself. Otherwise the string “Hello, Modern World” is displayed:

```
$ hello Hansel Gretel
Hello Hansel
Hello Gretel
```

For those that prefer a graphical development environment, IDEs are available for D and there is a D plugin for the Eclipse editing system.

Like Java, C++ and C#, D uses semi-colons to terminate program statements. D uses comment syntax similar to Java, including syntax for documentation comments (/**, etc.).

The first non-comment line in **hello.d** above imports the **std.stdio** module. Modules are D’s code-packaging construct, and, as the dot between **std** and **stdio** in *hello.d* above suggests, they can be grouped hierarchically. Like Python, in D there is a one-to-correspondence between a module and a text file. For example, a module **MyModule** would use a **module** statement and reside in a file named **MyModule.d**:

```
// MyModule.d
module MyModule;
[code follows here]
```

The **writeln** function is defined in the **stdio** module in the **std** *package* (i.e., group of modules). There is also a **write** function that does not append a newline to its output string. There are also versions of these functions, namely **writefln** and **writef**, which use their first parameter as a format string in C’s **printf** style. Note that there is no need to qualify the access to **writeln** with its module name (**stdio**). D’s name lookup algorithm first searches the current scope, and then each imported package in declaration order. Names must be qualified only if they are not uniquely defined among the current scope and imported modules. You can also disambiguate conflicting names with the **alias** statement:

```
alias a.foo foo;  // Unqualified references to foo will access a.foo
```

In those cases where a large number of name conflicts may occur, you can force name qualification for all imported names with a *static* import:

```
static import std.stdio;
…
    std.stdio.writefln("Hello " ~ a);     // Qualification required
```

The variable **args** is a *dynamic array* of strings. Dynamic arrays are heap-based, grow on demand, and are defined in the *language*, allowing more robust support and optimization than a library approach.

The brackets that identify an array definition can precede or follow the array name, so the following are equivalent:

```
int[] x;
int x[];
```

This accommodates the different styles in common use in other popular languages.

Static arrays are also supported and behave as they do in C and C++, except that all arrays are bounds-checked in D, and an exception is thrown when an out-of-bounds access is attempted.

The ".." notation used in **hello.d** above is a range that denotes an array *slice*, which refers to a contiguous subset of an array. Ranges are inclusive of their 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**. The string in **args[0]** is the program name ("hello"). An important feature of slices is that they are *not* copies; they represent a mutable reference range of the original array, and are therefore very efficient.

The **foreach** construct iterates through all elements of an array; the iteration variable and the array are separated by a semi-colon. Note that the type of iteration variable, **a**, is deduced by its context. The ~ operator is the *concatenation* operator for arrays. Strings are dynamic arrays of *immutable* Unicode characters.

A variation of **foreach** takes *two* iteration variables, where the first represents the 0-based index of the current array element. The following program illustrates this with **foreach_reverse**, which traverses its array backwards.

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

void main(string[] args) {
    foreach_reverse (i, a; args)
        writefln("%d %s",i,a);
}

/* Sample execution:

$ foreach one two three
3 three
2 two
1 one
0 foreach
*/
```

D also supports *associative arrays*, sometimes called maps, dictionaries, or hashes in other languages. Associative arrays in D can use normal array syntax, even in their declaration, since they are supported by the language, not a standard library module. The "type" in the declaration is the value type, and the key type is placed in brackets, as the following program illustrates.

```
// assoc.d: Associative arrays
import std.stdio;

void main() {
    int[string] keywords;     // A string-to-int mapping
    keywords["foo"] = 3;
    keywords["bar"] = 2;
    keywords["baz"] = 1;

    string abc = "foo";
    assert(keywords[abc] == 3);
    abc = "bar";
    assert(keywords[abc] == 2);

    foreach (kwd, value; keywords)
        writefln("%s = %s", kwd, value);
}

/* Output:
foo = 3
bar = 2
baz = 1
*/
```

Note that **foreach** supports a special form for associative arrays, where the iteration variables represent the key and value, respectively.

The following program illustrates simple file and text processing as well as type inference with the **auto** keyword.

```
// wc.d: Displays word counts in text files
import std.stdio, std.string, std.file;

// This function does all the work (Reads words into a list;
// computes counts; displays results)
void wc(string filename) {
    auto 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]);
}

// A simple driver: process all file arguments
void main(string[] args) {
    foreach (f; args[1..$]) {
        writefln("\n%s:", f);
        wc(f);
```

```
    }
}

/* Abbreviated output from the Gettysburg Address:

But,: 1
Four: 1
God,: 1
It: 3
Liberty,: 1
Now: 1
…
which: 2
who: 3
will: 1
work: 1
world: 1
years: 1
*/
```

The function **std.file.read** takes a filename string and returns an untyped array of bytes (**void [ ]**), which can be easily cast to a single string object. **std.string.split** returns an array containing all the space-delimited tokens of its string argument. We could have declared **words** as a string array, but **auto** infers the type from the initializer. **auto** is heavily used in D code.

The target of the second **foreach** loop illustrates *properties*. Associative arrays have a **keys** property that returns a new dynamic array of the keys of each pair in the original array. The **sort** property sorts an array in place, using the **<** operator to compare elements, and then returns the array by reference. You can define your own properties for the new types you create.

Functions And Parameters

D's imperative features allow stand-alone functions to be defined at the module level. Functions can also be nested in other functions, as in Algol and in functional languages. Some functions can even execute at *compile time*. Consider the following program adapted from the documentation on the D website (digitalmars.com):

```
// ctfe.d: Illustrates compile-time function execution
import std.stdio, std.conv;

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. C++ guarantees only that **n** is initialized before any function in its module executes, but in D **n** is already initialized to 9 before main

begins. The arguments in the function call must be compile-time expressions, such as numeric or string literals, and the context must be one of the following:

- initialization of a static variable
- dimension of a static array
- argument for a template value parameter

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 (i.e., "deep"))
- **immutable** (argument allows no changes *anywhere*, once initialized)

A function parameter defined without one of these attributes defaults to pass-by-value (i.e., the function receives a local, mutable copy of the calling argument). While most of these attributes are self-explanatory, a couple of them need some attention here.

The **lazy** storage class defers evaluation of the argument until it is actually used in the called function, and the parameter cannot be assigned to. This state of affairs is somewhere in between *pass-by-name* (the argument is evaluated on each access) and *pass-by-need* (the parameter is read-only and only evaluated on the first access). Lazy parameters in D are read-only but are evaluated upon *each access*, as the following program illustrates.

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

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. The second call to **printif** evaluates **s** each time it is accessed.

Reevaluating a lazy argument on each access is significant when the passed expression has side effects, as seen in the following example from the online documentation (see http://www.digitalmars.com/d/1.0/function.html).

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

void dotimes(int count, lazy void exp)
{
    for (int i = 0; i < count; ++i)
       exp();
}

void foo()
{
    int x = 0;
    dotimes(10, write(x++));
}

void main() {
    foo();
    writeln();
}

/* Output:
0123456789
*/
```

Procedures that return nothing can be passed lazily as type **void**. The compiler wraps the call to **writeln** above in the thunk passed to **f**. The expression **write(x++)** is evaluated 10 times, yielding a different value each time.

It may appear that **const** and **immutable** are the same thing, but there is a subtle difference. A **mutable** argument can be passed to a **const** parameter, which merely means that the object is **const** *locally*, and will therefore not be changed in the called function. When a variable is declared in the first instance to be **immutable**, however, it means that it may never be passed into a mutable context, which allows the compiler to make suitable optimizations. The **immutable** parameter attribute is for arguments that are declared **immutable** originally.

Nested Functions And Delegates

Although Algol and classic functional languages offered the ability to define a function within other functions, this feature virtually disappeared with the C family of languages. Java instead offers related functionality via inner classes, while C++ and C# support the creation of function objects by overloading the function-call operator. D allows both function objects and nested functions—the latter in two "flavors". Anything that be accomplished with function objects can be accomplished with nested functions in D.

The following example defines a function, **gtn**, which returns a function that determines whether its argument is greater than the original argument to **gtn**.

```
// 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 the "> 5" function
    writeln(g5(1));     // false
    writeln(g5(6));     // true
}
```

Note that the **execute** function is defined within **gtn**, and that it uses **gtn**'s parameter. **n**. Since the current activation of **gtn** will disappear before **execute** is called via **g5** in main, there needs to be some way for the data in **gtn** (the integer **n**) to persist after **gtn** returns. This is handled by *delegates* in D, a modern name for closures. A delegate consists of two items: 1) a pointer to the function to execute, and 2) a pointer to the execution environment for the function (in this case, the current activation record for **gtn**). In the case of **execute**, the activation record for **gtn** is automatically moved from the stack to the garbage-collected heap so that **n** can be accessed later. The return type of **gtn** is **bool delegate(int)**, a delegate that executes a function that takes an **int** and returns a **bool**, which, of course, is the signature of **execute**. Note also the use of **auto** in **main** so we don't have to repeat this signature as the type for **g5**. The execution environment for a delegate can also be a class or an object, as in C#.

The name **execute** doesn't serve much of a purpose, so D allows creating *function literals*, also known as *lambda expressions*. Using a function literal, **gtn** can be rewritten as

```
auto gtn(int n) {
    return delegate bool(int m) {return m > n;};
}
```

In an executable statement, the signature of the nested function completely follows the **delegate** keyword, whereas in the return type of a function declaration the **delegate** keyword occupies the position normally taken by the function name in a signature. Note the use of **auto** as the return

type, where the compiler infers the type from the return expression. In fact, the compiler can actually infer the complete return type even if we write **gtn** as follows, which is idiomatic D.

```
auto gtn(int n) {
    return (int m) {return m > n;}; // parm list + body
}
```

We can go a step further by making **gtn** *generic*, polymorphically handling any type that supports the **>** operator. This is achieved by using *compile-time parameters* in the function definition:

```
// 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
}
```

Whenever two parameter lists appear in a function definition, the first list represents the parameters supplied at compile time, such as types and integral literal expressions, just like template parameters in C++. The type for **T** is inferred from the calls in **main** (**int** for **g5**, **string** for **g5s**).

If a nested function does not use anything in its enclosing environment, it can be returned as a plain function pointer. This is achieved by preceding the nested function definition with the keyword **static**. (So delegates are *non-static* nested functions). Plain function *literals* use the **function** keyword in place of **delegate**.

Lazy parameters, function literals, and functions passed to and returned from functions are the "stuff" of functional programming. A pure functional language also does not allow assignment, so whenever a function is called with the same parameters, the same result is always returned (a quality known as *referential transparency*). Not allowing a variable to change after it is initialized also makes concurrent programming much less of a headache than in imperative languages, since race conditions are a non-issue. Referential transparency is enforced in D with *pure* functions. A pure function must have parameters that are **immutable** or **const**, and must neither read nor write any non-local, mutable state (but local mutables are okay). In addition, a pure function cannot call an "impure" function. A function is declared pure with the **pure** keyword, and the compiler ensures that the requirements just listed are met. An iterative version of a Fibonacci number function, even though it changes local, private state (see **a**, **b**, and **t**

below), satisfies the conditions of referential transparency, and so can be declared **pure**, as follows:

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

D's Software Engineering Support

D supports a number of features that enhance code reliability and maintainability, including scope statements, contact programming, unit testing, debug statements, and versioning. The first three features will be illustrated here.

Unless carefully planned for, runtime errors due to *external forces* can easily place a program in an inconsistent state. Resource management is a typical example. Consider the following function.

```
void f() {
    acquire();  // Acquire some resource
    risky_op(); // Might fail
    release();  // Release the resource
    writeln("f succeeded");
}
```

A Java-like solution uses a **finally** block, similar to the following D code.

```
void f() {
    acquire();
    try {
        risky_op();
        writeln("f succeeded");
    }
    finally {
        release();
    }
}
```

D also lets you explicitly supply specific *cleanup code* for whenever execution exits a scope:

```
void f() {
    acquire();
    scope(exit) release();
    risky_op();
    writeln("f succeeded");
}
```

The **scope** statement, called a *scope guard*, 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

The utility of the **scope** statement becomes more obvious with multi-step resource acquisition requiring rollback semantics in the case of failure, as in the following function.

```
void g() {
   risky_op1();
   risky_op2();
   risky_op3();
   writeln("g succeeded");
}
```

Here we want all three operations to succeed or fail together. The **try-finally** approach is quite complex:

```
void g() {
    risky_op1();
    try {
        risky_op2();
    }
    catch (Exception x) {
        undo_risky_op1();      // Back-out op1
        throw x;               // Rethrow exception
    }
    try {
        risky_op3();
        writeln("g succeeded");
    }
    catch (Exception x) {
        // Back-out op1 and op2 in reverse order
        undo_risky_op2();
        undo_risky_op1();
        throw x;
    }
}
```

It is much easier to back out of complicated transactions with D's **scope** statement:

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

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 is a technique for validating a program's *internal correctness*, and involves making explicit the conditions that govern the interactions between client and server code.[6,7] These conditions include *class invariants*, and function *preconditions* and *postconditions*. Class invariants are conditions concerning the *state* of an object that hold true immediately after the construction of an object, before and after the execution of any *public* method of the class, and immediately before an object's destructor. Preconditions and postconditions apply to a single function and represent conditions that hold true before and after the execution of the method, respectively. The following example, which is a first attempt at a **struct** that simulates a *rational number* type, illustrates a class invariant and a method precondition, as well as unit testing and operator overloading.

```
// rational.d
import std.math;    // For abs()

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

    // Local helper function
    static int gcd(int m, int n) {
        m = abs(m);
        n = abs(n);
        return n == 0 ? m : gcd(n, m%n);
    }

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

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

    // + operator (NOTE: The special "if" is tested at compile time)
    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);
    auto r2 = Rational(3,4);
    auto r3 = r1 + r2;
    assert(r3.num == 5);
    assert(r3.den == 4);
}

void main(){}
```

To implement function-based conditions, assertions are placed in named blocks as follows:

- **in {…}** (for function preconditions)
- **out {…}** (for function postconditions)

When either or both of these are present, the function body must itself appear in a **body** block. The constructor above, named **this**, has the precondition that the denominator must be non-zero. If this condition is not met, an exception is thrown.

This particular implementation also requires that the fraction is represented in lowest terms and that the denominator is positive at all times. This condition is established by the constructor and enforced by the class invariant, which appears in a method named **invariant**. Since class invariants are checked at the end of each constructor automatically, no **out** block is necessary in this case, as there are no other conditions that apply.

The separation of contract conditions from the body of a function is significant. This way the compiler can combine conditions properly in inheritance hierarchies. It is well known that 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 therefore automatically checks the *at least one* of the preconditions is met (searching in top-down, class-hierarchy order to the current subtype) and that *all* of the invariants and postconditions are satisfied when dispatching polymorphic methods. All of these checks can be disabled in deployed code by using the **-release** compiler option. The code in all **unittest** blocks is executed before **main** begins if the **-unittest** compiler option is specified.

Conclusion

The D programming language combines many valuable and popular features from both classic and modern languages. Its design emphasizes a clear, high-level, 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. A reference book written by one of the co-designers of the language was released in June 2010.[8]

References

1. Computer Science Curriculum 2008: An Interim Revision of CS 2001, ACM/IEEE, December 2008, Preface.
2. Ibid, Section 1.2.

3. The Tiobe Index, June 2010, http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.
4. Stroustrup, B., C++ Applications, http://www2.research.att.com/~bs/applications.html, June 2010.
5. Bell, Kelly, & Parker, *Learn to Tango with D*, aPress, 2007, Foreword.
6. Parnas, D.L., A Technique for Software Module Specification with Examples, *CACM*, 15(5), May 1972.
7. Meyer, Bertrand: Applying "Design by Contract", *Computer (IEEE)*, 25(10), October 1992, pp. 40–51.
8. Alexandrescu, A., The D Programming Language, Addison-Wesley, 2010.