The D Programming Language

Ameer Armaly

Abstract

Familiarize yourself with the powerful compiled and managed language D.

During the past few decades, programming languages have come a long way. In comparison to the dawn of UNIX and C, when compiled languages were just getting their start, today's world is full of languages with all sorts of goals and features. In this article, I discuss one such language, D from Digital Mars. D is a natively compiled, statically typed, multiparadigm C-like language. Its aim is to simplify development of any type of application—from kernels to games—by combining the performance and flexibility of C with the productivity-boosting factors of languages, such as Ruby and Python. It originally was conceived by Walter Bright, the man who wrote the first ANSI C++ compiler for DOS. The reference compiler is freely downloadable for both Windows and Linux, and the front end is licensed under a dual GPL and Artistic license.

GDC is a D compiler, which uses the GCC back end and is distributed under the GPL. D's features include the lack of a preprocessor, a garbage collector, flexible first-class arrays, contracts, inline assembler and more. With all this, it still maintains ABI compatibility with C, allowing you to use all your old C libraries from D easily, with little work. The D standard library includes bindings to all standard C functions as well.

Hello World

In D, the Hello World program goes like this:

```
import std.stdio; // standard i/o module int main(char[][] args) { writefln("Hello world!"); return
    0; }
```

writef is D's typesafe version of printf; writefln adds a newline character at the end. Garbage collector D includes an automatic garbage collector, relieving the programmer of the need to manage memory explicitly. This allows programmers to focus more on the task at hand, as opposed to having to worry about the condition of each memory chunk. Furthermore, it eliminates a whole class of bugs dealing with dangling pointers and invalid memory references. In times when the GC would slow the application down, there is always the option of turning it off altogether or using C's malloc and free for memory management.

Modules

In D, modules are imported using the import statement and have a one-to-one correspondence with source files, with the period as the path separator. Each symbol within a module has two names: the name of the symbol and the name of the symbol prefixed by the module name, which is called the fully qualified name or FQN. For example, writefln can be referred to as writefln or std.stdio.writefln. For cases when the FQN is preferred, the static import statement imports the module's symbols but avoids putting them into the global namespace. For example, both the std.string and std.regexp modules include functions for find, replace and split. Because I'm more likely to use pure string functions than regular expressions, I would statically import std.regexp, so that whenever I wanted to use any of its functions, I would have to be explicit, whereas I simply could call the string functions by their regular names.

Modules can have static constructors and destructors. The static this() function in any module is the static constructor and is invoked before main(); after main has returned the static, ~this() function is invoked. Because modules are imported symbolically, this means there are no header files. Everything is declared once and only once, eliminating the need for declaring functions in advance or declaring classes in two places and trying to keep both declarations consistent.

alias and typedef

In D, there is a distinction made between an alias and a type. A typedef introduces an entirely new type to the type-checking system and to function overloading, which are discussed later. An alias is a simple replacement for a type, or optionally a symbol:

```
alias int size_t; typedef int myint; //can't implicitly convert to int alias
    someReallyLongFunctionName func;
```

Arrays

In D, arrays are first-class types in every way. D contains three types of arrays: static, dynamic and associative arrays. Array declarations read right to left; char[][] is interpreted as an array of arrays of characters:

```
int[] intArray; // dynamic array of ints int[2][4] matrix; // a 2x4 matrix
```

All arrays have length, sort and reverse properties. Associative arrays are arrays where the index is something other than sequential integers,

possibly text strings, structs or arbitrary integers:

```
import std.stdio; int main(char[][] args) { int[char[]] petNumber; petNumber["Dog"] = 212;
    petNumber["cat"] = 23149; int[] sortMe = [2, 9, 341, 23, 74, 112349]; int[] sorted = sortMe.sort; int[] reversed =
    sorted.reverse; return 0; }
```

Dynamic and static arrays can be sliced with the .. operator. The starting parameter is inclusive, but the ending parameter is not. Therefore, if you slice from zero to the length of an array, you get the whole array:

```
int[] numbers = [1, 2, 3, 4, 5, 6, 7]; numbers = numbers[0..2] // 1-3 now
```

Finally, D uses the ~ operator for concatenation, as addition and concatenation are at their most fundamental two different concepts:

```
char[] string1 = "Hello "; char[] string2 = "world!"; char[] string = string1 ~ string2; // Hello
    world!
```

This is a prime example of how D implements a lot of syntactic sugar on top of more low-level routines to make the programmer more focused on the implementation of the task itself. Strings D takes arrays one step further. Because strings are logically arrays of characters, D has no built-in string type; instead we simply declare an array of characters.

Furthermore, D has three types of strings: char, a UTF-8 codepoint; wchar, a UTF-16 codepoint; and dchar, a UTF-32 codepoint. These types, along with standard library routines for manipulating unicode characters, make D a language suited to internationalized programming. In comparison with C, D strings know their length, eliminating even more bugs and security issues dealing with finding the elusive null terminator.

Contracts

D implements techniques that make contract programming easy, which makes for better quality assurance in programs. Making contracts part of the language itself makes it much more likely that they actually will be used, because the programmer doesn't have to implement them or use an outside library for them.

The simplest type of contract is the assert statement. It checks whether the value that is passed to it is true, and if not, it throws an exception. Assert statements can be passed optional message arguments to be more informative. Functions have two types of contracts, pre and post, signified by the in and out blocks preceding a function. The in contract must be fulfilled before the rest of the function is executed; otherwise, an AssertError is thrown. The post contract is passed the return value of the function and checks to make sure the function did what it was supposed to do before the value is passed to the application. When a program is compiled with the release option turned on, all asserts and contracts are removed for speed purposes:

```
int sqrt(int i) in { assert(i > 0); } out(result) { // the return value is // always assigned to
    result assert((result * result) ==i); } body {...}
```

Another type of contract is the unit test. Its purpose is to ensure that a particular function or set of functions is working according to specification with various possible arguments. Suppose we have the following rather useless addition function:

```
int add(int x, int y) { return x + y; }
```

The unit test would be placed in the same module, and if the unittest option is enabled, it would be run as soon as the module is imported and any function from it is executed. In this case, it probably would look something like this:

```
unittest { assert(add(1, 2) == 3); assert(add( -1, -2) == -3); }
```

Conditional Compilation

Because D has no preprocessor, conditional compilation statements are part of the language itself. This removes the numerous headaches caused by the preprocessor along with the infinite ways in which it can be used, and it makes for a faster compile. The version statement is a lot like #ifdef in C. If a version identifier is defined, the code under it gets compiled in; otherwise, it doesn't:

```
version(Linux) import std.c.linux.linux; else version(Win32) import
    std.windows.windows;
```

The debug statement is a lot like the version statement, but it doesn't necessarily need an identifier. Debugging code can be placed in the global debug condition or in a specific identifier:

```
debug writefln("Debug: something is happening."); debug (socket) writefln("Debug: something is
    happening concerning sockets.");
```

The static if statement allows for the compile-time checking of constants:

```
const int CONFIGSOMETHING = 1; void doSomething() { static if(CONFIGSOMETHING == 1) { .... }
    }
```

Scope Statement

The scope statement is designed to make for a more natural organization of code by allowing a scope's cleanup, success and failure code to be logically grouped:

```
void doSomething() { scope(exit) writefln("We exited."); scope(success) writefln("We exited
    normally."); scope(failure) writefln("We exited due to an exception."); ... }
```

Scope statements are executed in reverse order. Script syntax DMD, the reference D compiler, supports the -run option, which runs the program taken from standard input. This allows you to have self-compiling D scripts, as long as the appropriate line is at the top, like so:

```
#!/usr/bin/dmd -run
```

Type Inference

D allows the automatic inferring of the optimal type of a variable with the auto-declaration:

```
auto i = 1; // int auto s = "hi"; // char[4]
```

This allows the compiler to choose the optimal type when that functionality is needed.

foreach

Some of you might be familiar with the foreach construct; it essentially says, “do this to every element of this array”as opposed to “do this a set number of times, which happens to be the length of the array”. foreach loops simplify iteration through arrays immensely, because the programmer no longer even has to care about the counter variable. The compiler handles that along with making each element of the array available:

```
char[] str = "abcdefghijklmnop"; foreach(char c; str) writefln(c);
```

You also can obtain the index of the element by declaring it in the loop:

```
int [] y = [5, 4, 3, 2, 1]; foreach(int i, int x; y) writefln("number %d is %d", i,
    x);
```

Finally, you can avoid worrying about the types of the variables, and instead use type inference:

```
foreach(i, c; str)
```

This opens up the field for numerous compiler optimizations that could be performed—all because the compiler is taking care of as much as possible while still providing the programmer with the flexibility to accomplish any given task.

Exceptions

As a rule, D uses exceptions for error handling as opposed to error codes. D uses the try-catch-finally model for exceptions, which allows cleanup code to be inserted conveniently in the finally block. For those cases when the finally block is insufficient, scope statements come in quite handy.

Classes

Like any object-oriented language, D has the ability to create object classes. One major difference is the lack of a virtual keyword, unlike with C++. This is handled automatically by the compiler. D uses a single-inheritance paradigm, relying on interfaces and mixins, which are discussed later to fill in the gaps. Classes are passed by reference rather than by value, so the programmer doesn't have to worry about treating it like a pointer. Furthermore, there is no -> or :: operator; the . is used in all situations to access members of structs and classes. All classes derive from Object, the root of the inheritance hierarchy:

```
class MyClass { int i; char[] str; void doSomething() { ... }; }
```

Classes can have defined properties by having multiple functions with the same name:

```
class Person { private char[] PName; char[] name() {return PName;} void name(char[] str) { // do
    whatever's necessary to update any // other places where the name is stored PName = name; }
    }
```

Classes can have constructors and destructors, namely this and ~this:

```
class MyClass { this() { writefln("Constructor called");} this(int i) { writefln("Constructor
    called with %d", i); } ~this() { writefln("Goodbye");}
```

Classes have access to the constructors of their base class:

```
this(int i) { super(1, 32, i); // super is the name of the // base class constructor
```

```
    }
```

Classes can be declared inside other classes or functions, and they have access to the variables in that scope. They also can overload operators, such as comparison, to make working with them more obvious, as in C++.

Classes can have invariants, which are contracts that are checked at the end of constructors, before destructors and before public members, but removed when compiling for release:

```
class Date
{
int day;
int hour;
invariant
    {
        assert(1 <= day && day <= 31);
        assert(0 <= hour && hour < 24);
    }
}
```

To check whether two class references are pointing to the same class, use the is operator:

```
MyClass c1 = new MyClass; MyClass c2; if(c1 is c2) writefln("These point to the same
    thing.");
```

Interfaces

An interface is a set of functions that any class deriving from it must implement:

```
interface Animal { void eat(Food what); void walk(int direction); void makeSound();
}
```

Functions

In D, there is no inline keyword—the compiler decides which functions to inline, so the programmer doesn't even have to worry about it. Functions can be overloaded—that is to say, two functions with the same name can take different parameters, but the compiler is smart enough to know which one you're talking about:

```
void func(int i) // can implicitly take // longs and shorts too {...} void func(char[] str) {...}
    void main() { func(23); func("hello"); }
```

Function parameters can be either in, out, inout or lazy, with in being the default behavior. Out parameters are simple outputs:

```
void func(out int i) { I += 4; } void main() { int n = 5; writefln(n); func(n); writefln(n);
    }
```

inout parameters are read/write, but no new copy is created:

```
void func(inout int i) { if(i >= 0) ... else ... }
```

Lazy parameters are computed only when they are needed. For example, let's say you called a function like this:

```
log("Log: error at "~toString(i)~" file not found.");
```

Notice that every time you call it, the strings are concatenated and passed to the function. The lazy storage class means that the strings are put together only if they are called upon, increasing performance and efficiency. Nested functions in D allow the nesting of functions within other functions:

```
void main() { void func() { ... } }
```

Nested functions have read/write access to the variables of the enclosing function:

```
void main() { int i; void func() { writefln(i + 1); } }
```

Templates

D has a totally redesigned and highly flexible template system. For starters, the ! operator is used for template instantiation. This eliminates the numerous ambiguities caused by <> instantiation and is more readily recognizable. Here is a simple copier template:

```
template TCopy(t) { void copy(T from, out T to) { to = from; } } void main() { int from = 7; int
    to; TCopy!(int).copy(from, to); }
```

Template declarations can be aliased:

```
alias TFoo!(int) temp;
```

Templates can be specialized for different types, and the compiler deduces which type you are referring to:

```
template TFoo(T) { .... } // #1 template TFoo(T : T[]) { .... } // #2 template TFoo(T : char) { .... }
    // #3 template TFoo(T,U,V) { .... } // #4 alias TFoo!(int) foo1; // instantiates #1 alias TFoo!(double[]) foo2; //
    instantiates #2 // with T being double alias TFoo!(char) foo3; // instantiates #3 alias TFoo!(char, int) fooe; //
    error, number of // arguments mismatch alias TFoo!(char, int, int) foo4; // instantiates #4
```

Function Templates

If a template has one member function and nothing else, it can be declared like this:

```
void TFunk(T) (T i) { ... }
```

Implicit Function Template Instantiation

Function templates can be instantiated implicitly, and the types of the arguments deduced:

```
TFoo!(int, char[]) (2,"foo"); TFoo(2, "foo");
```

Class Templates

In those cases when you need to declare a template and its only member is a class, use the following simplified syntax:

```
class MyTemplateClass (T) { ... }
```

Mixins

A mixin is just like cutting and pasting a template's code into a class; it doesn't create its own scope:

```
template TFoo(t) { t i; } class test { mixin TFoo!(int); this() { i = 5; } } void main() { Test t =
    new Test; writefln(t.i); }
```

Conclusion

D is a promising language that is able to supply many different needs in the programming community. Features such as arrays, SH syntax and type inference make D comparable to languages, such as Ruby and Python, in those regards, while still leaving the door open for low-level system programmers with the inline assembler and other features. D can be applied to many programming paradigms—be they object-oriented with classes and interfaces, generic programming with templates and mixins, procedural programming with functions and arrays, or any other. The garbage collector relieves the programmer of the need to manage all memory chunks manually, while still making it possible for those situations in which manual memory management is preferred. It brings in features of imperative languages, such as Lisp, with the lazy storage class, which drastically speeds up efficiency. The language is relatively stable, with the occasional new features or changes added in.

In short, D is a language ready for real-world deployment.

Resources *9307s1.qrk*

D Specification and Reference Compiler: http://www.digitalmars.com/d

GDC: http://dgcc.sf.net

Numerous Open-Source Projects and Tutorials: http://www.dsource.org