A "D" in Programming, Part 2

by Chuck Allison

ISTOCKPHOTO

As I write this, my last official pitch for the D programming language, I notice that D is at position twelve and climbing in Tiobe's ranking of the twenty most popular languages for February 2008 [1]. Earlier in March, the first book on D, *Learn to Tango with D*, hit the shelves [2]. In its preface, D's designer, Walter Bright, said:

"Amazingly, there is no language that enables precise control over execution while offering modern and proven constructs that improve productivity and reduce bugs … 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."

Like C++, D supports down-to-the-metal programming when you need it, and it compiles to fast-and-lean, native executables. It also supports generic programming and templates in all their glory. The standard C library is available directly from D code. Like Python, D supports modules and packages, garbage collection, functions that behave as first-class entities, a clean (though C-like) syntax, and flexible, built-in data structures. Like Java, D has inner classes. Like C#, D gives you delegates, but in a more flexible way. D is a multi-paradigm language that may be just what you need most of the time.

For many of you, I suppose the software engineering features of D would be of most interest, but in this article I'd like to bring to "closure" (pun intended) a running example from previous Code Craft articles as I explore some powerful features of the D language.

Nested Functions and Closures

Much of what we enjoy today about objects was accomplished in earlier days through other means. To hide data in C, for example, you just declared a file-scope variable to be *static*. The file was the container, the "object," if you will, that held data values not directly accessible to users. See listing 1.

```
/* file1.c */
static int theData = 7;   // Private data
int getData() { return theData; }
```

Listing 1

This approach breaks down when you need multiple instances. File I/O is too cumbersome and expensive to use as a model for objects that contain hidden data. Object-oriented languages have direct support for in-memory object creation at runtime, of course, as in listing 2.

```
// MyClass.java
class MyClass {
    private int theData;
    public int getData() { return theData;}
}
```

Listing 2

How did we ever survive without classes?

There were many ways, but I'd like to mention one that is still important: *nested functions*. Nested functions are not permitted in many modern languages but were *de riguer* in languages like Lisp, Algol, PL/I, Pascal, and Ada. C dropped nested functions for simplicity, but Python and D have brought them back because sometimes they are superior to using objects.

To illustrate, let me return to an example I've used in recent Code Craft articles: function composition. In the January 2008 Code Craft, I presented the generic C++ function composer shown in listing 3.

A `Composer` is an object that takes a list of single-valued functions (or entities callable as such) and calls them in turn in nested fashion so you end up with `f1(f2(…fn(x)…))`. The private data here is the sequence of functions, indicated by the pair of iterators, `beg` and `end`. Since objects aren't the only way to hide data, let's look at the nested-function solution in D shown in Listing 4.

This rendition of `compose` is a function template, evidenced by the `(T)` following its name, and it accepts a dynamic array of functions, each of which takes a single `T` argument and returns a `T` value. It returns a *delegate* that is callable as a single-valued function of type `T`. The nested function `doit` iterates through the list of functions, `funs`, in reverse order, applying each function and accumulating the result as it goes. Let's examine this more closely.

Notice that `compose` has only two statements: a definition of the nested function `doit` and a statement that returns a

```
template<class Iter>
class Composer {
private:
    typedef typename iterator_traits<Iter>::value_type Fun;
    typedef typename Fun::result_type T;
    typedef reverse_iterator<Iter> RevIter;
    RevIter beg, end;
    static T apply(T sofar, Fun f) {
        return f(sofar);
    }
public:
    Composer(Iter b, Iter e) : beg(RevIter(e)), end(RevIter(b)) {}
    T operator()(T x) {
        return accumulate(beg, end, x, apply); // Function Applicator
    }
};
```

Listing 3

```
T delegate(T) compose(T)(T function(T)[] funs) {
   T doit(T n) {
      T result = n;
      foreach_reverse (f; funs)
         result = f(result);
      return result;
   }
   return &doit;
}
```

Listing 4

pointer to the function `doit`. The function `doit` itself uses the array `funs`, which is defined in `compose`'s parameter list. It is common to say that `funs` is in the "calling environment" of `doit`.

So what happens when a pointer to `doit` is returned from a call to `compose`? In order for `funs` to persist for use by subsequent calls to `doit`, it has to somehow be saved and connected to `doit`. Consider how `compose` is used in listing 5.

The `function` keyword has two related uses in D: to declare a function pointer and to define an anonymous *function literal* (like "lambda" expressions in functional programming languages). The first usage occurs in the definition of `compose` in the first line of listing 4 and also in the first line inside `main` in listing 5. In each case, `funs` is declared a dynamic array of single-valued function pointers. In the second and third lines of

```
import std.stdio;

void main() {
   int function(int)[] funs;
   funs ~= function int(int x){return x*x;};
   funs ~= function int(int x){return x+1;};
   auto c = compose(funs);
   writeln(c(3));    // 16
}
```

Listing 5

`main` in listing 5, two unnamed functions are created and stored in `funs`.

The variable `c` holds the delegate returned by the call to `compose`. So what is a delegate in D? It is the very mechanism that allows `funs` to persist and be used by the instance of `doit` returned by a call to `compose`. A delegate is a *pair* that contains a pointer to a function (which could be a class method) and the calling context of the function. That calling context could be an activation (stack frame) of an enclosing function, as `compose` is for `doit`, or it could be an object or class used as context for a method. So that its calling context—the activation of the call to `compose`—is preserved for `doit` to do its work, `doit` must be returned as a delegate. That context is preserved even after `compose` has terminated. Such a persistent calling context is called a *closure*. The closure for `compose` is moved from the runtime stack to the garbage-collected heap so it persists as long as the variable `c` in listing 5 does.

There is no need to create a class to solve this problem, and the nested-function approach is simpler anyway.

Whither Function Objects?

C++ popularized the use of function objects—sometimes called *functors*—with the introduction of STL. A function object is nothing more than an instance of a class that overloads the function-call operator. To illustrate, listing 6 creates a C++ function object, `gtn`, which determines whether its argument is greater than a previously stored value.

Similar code can be written in D using the `opCall` special function, but the nested-function version in listing 7 is simpler.

Once again we have an outer function that returns a nested function that has access to the outer calling context. It appears that nested functions and closures can replace function objects

```
template<class T>
class gtn {
    T n;
public:
    gtn(T x) : n(x) {}
    bool operator()(T m) {
        return m > n;
    }
};

int main() {
    gtn<int> g5(5);
    cout<< g5(1) <<endl;      // false
    cout<< g5(6) <<endl;      // true
}
```

Listing 6

altogether in D. These classic language constructs are as useful today as ever. **{end}**

REFERENCES:

1] www.tiobe.com/tiobe_index/index.htm

2] Bell, Kris; Igesund, Lars Ivar; Kelly, Sean; and Parker, Michael. *Learn to Tango with D.* Apress, 2007.


```
bool delegate(T) gtn(T)(T n) {
    bool doit(T m) {
        return m > n;
    }
    return &doit;
}

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

Listing 7