The D Programming Language

Walter Bright

D is a programming language that looks a lot like C and C++, but eliminates features that make programs difficult to write, debug, test, and maintain.

Walter is principle of Digital Mars, a C/C++ compiler firm. He can be contacted at http://www.digitalmars.com/.

C and C++ are spectacularly successful programming languages because they have a mix of features that programmers find extremely useful. They've also proven to be accommodating enough to enable new capabilities to be grafted on to the old structure. But they're old — C is 30 years old and C++ is 15. New capabilities are constrained by backward compatibility with legacy code going back 25 or more years.

In any engineering project, be it a Boeing 707, Ford Mustang, or programming language, there comes a point where it becomes worthwhile to take a step back, look at what works, what doesn't work, dispense with the baggage, and come up with a ground-up new design. That point is when incremental refinement no longer yields much improvement.

This is the primary motivation behind D, a new language design I present in this article. D intentionally looks very much like C and C++. D eliminates features that make programs arbitrarily difficult to write, debug, test, and maintain, while adding features that make it easier to do such tasks. Features that have been supplanted by newer ones, but are retained for backward compatibility with legacy code, are scrapped. D's emphasis is on simple, understandable, and powerful syntax. Contorted syntax necessary to fit in with the old legacy structure of C has been jettisoned.

It's true that nothing D has cannot be emulated in some fashion with C++ by using macros, templates, *#if*s, and rigid adherence to coding-style guidelines. I know that my own C++ coding practice has tended toward the D way (and in fact has formed much of the genesis of ideas for the language). But when I recode in D, the look of the code is satisfyingly simple and straightforward. And code that looks simple and straightforward is easier to code, understand, and debug, and hence more likely to work correctly.

The D Way

Every language should have a set of guiding principles through which style and features should be filtered. D's guiding principles are:

- A short learning curve for C and C++ programmers. Although D will not compile C/C++ code, it looks enough like it that your skills are easily transferrable to D.
- Tokenizing should be independent of syntax, and syntax should be independent of both tokenizing and semantic analysis. This greatly simplifies compiler construction, and enables related program analysis tools (like syntax-directed editors and code profilers) to be easily produced.
- Competent programmers should be able to understand the entire D language without great effort.
- D compilers should be relatively easy to implement and get right. This enables multiple competing implementations to rapidly appear.
- There is no text preprocessor. Examine each of the typical problems the C preprocessor is used to solve, and find a way to support solving it in the language.
- A surprisingly large amount of time in C/C++ programming is spent managing memory. With modern garbage collectors, there is little need for such. D programs will be garbage collected.
- Lint-like features should be part of the language.

- Design by Contract is an essential part of building reliable programs.
- Unit testing should be built into the language.
- When necessary, down and dirty to the metal coding should be supported. While D provides a simple and convenient interface to C, it shouldn't be necessary to write C code to complete an app. D should be complete enough that system apps won't need to be uneasy amalgamations of modules written in different languages.
- Using pointers should be exceptional, rather than ubiquitous. C is dependent on pointers to get routine programming tasks done. Typical uses of pointers should be analyzed and replaced with syntax that either obviates the need for pointers or hides the use of them.

The complete specification and other information for the D language is available at http://www.digitalmars.com/d/index.html. In this article, I'll concentrate on a few of D's unique characteristics — forward references, pointers, and Design by Contract.

Forward References

Listing One(a) is a typical technique seen in about every C program. Because C requires that declarations syntactically appear before use, either the order the functions appear in is artificially constrained by this requirement or a list of forward declarations is used. The reasons for the requirement are to minimize memory consumption of the compiler and because it's so difficult to parse C (and especially C++) without doing semantic analysis.

The first reason is irrelevant these days, and the second is dispensed with by D's requirement that syntax analysis be independent of semantic analysis. Now, the tedious necessity of writing forward declarations goes away. With this in mind, Listing One(a) becomes Listing One(b).

D takes this further. Since forward class declarations are not necessary either, Listing Two(a) becomes Listing Two(b). Taking it to its logical conclusion, the C++ technique of separating class member declaration from definition, so that Listing Three(a) becomes Listing Three(b) in D. Now, a member function prototype only has to be written once. All the information for a class is within its definition, not scattered over arbitrary source files.

Modules. But wait a minute, how do you make an .h header file then? Easy. Header files are not part of D. Header files are part of the text preprocessor eliminated in D and are replaced by the notion of "modules," which are implicitly generated by source file. To get the symbols from that file (for example foo.d), use the *import* statement:

```
import foo;
```

Whenever a D source file is compiled, the compiler takes the global symbols generated by it and adds it to a database. The *import* statement then retrieves the global symbols defined by that file. This is much faster than reparsing a text header file, and makes it quite unnecessary to make two versions of each file (header and source).

Pointers

General pointers are an extremely powerful and useful concept in C and C++. Unfortunately, they are also the source of many, if not most, programming bugs. D examines the most common uses of pointers and replaces them with safer but equivalent language features.

***Out* Function Parameters.** Many functions need to return multiple values. This is typically done by passing a pointer to the return value. An example of this is the C run-time library function *strtol()* defined as:

```
long strtol(char *nptr, char **endptr, int base);
```

Through **endptr* is stored as a pointer to the last character scanned. To fix this, D introduces the notion of an *out* parameter, which is set by the called function:

```
long strtol(char *nptr, out char *endptr, int base);
```

This is much like the reference declaration in C++, with one crucial difference — there is no distinction between *in*, *out*, and *inout* function parameters in C++. Hence, the compiler is crippled in its ability to accurately determine if a variable is used before being initialized, and so on.

Design by Contract

One of the most interesting ideas implemented in D is Design by Contract (DBC), a technique pioneered by B. Meyer. An ordinary program is an implementation of an algorithm. A contract is a specification of the implementation. DBC enables those contracts to be written into the implementation code so that the code can be verified to be correct at run time or even compile time, at a level far more comprehensive than the syntax check expected of a compiler.
Comprehensive use of contracts shifts the specification of a program out of the documentation (which inevitably is missing, wrong, ambiguous, ignored, and out of date) and into the source code itself, guaranteeing that when the contracts don't match the implementation, one or the other or both get corrected.
Contracts prevent the legacy problem of inadvertent reliance on undocumented behavior of a library, and then being forced to support such behavior in future versions. Contracts help customers and other team members use your modules as intended.
Not only can contracts reduce the bugs in a program, but in more advanced compilers they can provide additional semantic information to the optimizer to help it generate better code.
Asserts. The simplest contract is the assert. Asserts are the only contract support in C, and it is not actually even part of the language but a conventional use of the preprocessor. D promotes asserts into an integral, syntactic part of the language.
An assert takes an expression as an argument, and that expression must evaluate to True. If it does not, an *AssertException* is thrown; see, for example, Listing Four. Asserts form the building block for all the DBC support in D.
Preconditions. Preconditions are contracts that must be satisfied upon entry to a function. This checks that the function's input arguments are correct. For example, the C *memcpy()* function is specified to operate on nonoverlapping memory regions. An implementation of it might look like Listing Six(a). Adding a precondition will verify the nonoverlap rule, rather than relying on documentation and chance; see Listing Six(b).
Postconditions. Postconditions are contracts that verify the output of a function. Postconditions should check the function return result, any out parameters, and any expected side effects. The *memcpy()* function can be verified, as in Listing Seven.

Conclusion

D is a new programming language offering fundamental improvements over C and C++. Its focus is on improving programmer productivity and program reliability. Key features of D contributing to this are:

- Elimination of the need for forward declarations.
- Use of symbolic imports rather than textual header files.
- Great reduction in the need and ubiquity of pointers.
- Integral support for the Design by Contract programming paradigm.

Again, many more features of D are discussed at http://www.digitalmars.com/d/index.html.
DDJ

Listing One

(a)

```
static int bar();       // forward reference
int foo()
{
    return bar();
}
static int bar()
{
    ...
}
```

(b)

```
int foo()
{
    return bar();
}
static int bar()
{
    ...
}
```

Listing Two

(a)

```
class Foo;
class Bar { Foo f; }
class Foo { int x; }
```

(b)

```
class Bar { Foo f; }
class Foo { int x; }
```

Listing Three

(a)

```
class Foo
{
    int member();
};

int Foo::member()
{
    ...
}
```

(b)

```
class Foo
{
    int member();
    {
        ...
    }
};
```

Listing Four

```
int i;
Symbol s = null;

for (i = 0; i < 100; i++)
{
    s = search(array[i]);
    if (s != null)
        break;
}
assert(s != null);      // we should have found the Symbol
```

Listing Five

```
(a)
class Year
{
    int year;   // years in A.D.
}

(b)
class Year
{
    int year;   // years in A.D.

    invariant
    {
        // Our business app doesn't care about ancient history
        assert(year >= 1900 && year < 3000);
    }
}
```

Listing Six

```
(a)
byte *memcpy(byte *to, byte *from, unsigned nbytes)
{
    while (nbytes--)
        to[nbytes] = from[nbytes];
    return to;
}

(b)
byte *memcpy(byte *to, byte *from, unsigned nbytes)
    in
    {
        assert(to + nbytes < from || from + nbytes < to);
    }
    body
    {
        while (nbytes--)
            to[nbytes] = from[nbytes];
        return to;
    }
```

Listing Seven

```
byte *memcpy(byte *to, byte *from, unsigned nbytes)
    in
```

```
    {
        assert(to + nbytes < from || from + nbytes < to);
    }
    out(result)
    {
        assert(result == to);
        for (unsigned u = 0; u < nbytes; u++)
            assert(to[u] == from[u]);
    }
    body
    {
        while (nbytes--)
            to[nbytes] = from[nbytes];
        return to;
    }
```