The D Programming Language

Andrei Alexandrescu

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Chapter

1

"D"iving In

You know what's coming first, so without further ado:

```
import std.stdio;
void main() {
   writeln("Hello, world!");
}
```

Depending on what other languages you know, you might have a feeling of déjà vu, a mild appreciation for simplicity, or perhaps a slight disappointment that D didn't go the scripting languages' route of allowing top-level statements. (Top-level statements invite global variables, which quickly turn into a liability as the program grows; D does offer ways of executing code outside `main`, just in a more structured manner.) If you're a stickler for precision, you'll be relieved to hear that `void main` is equivalent to an `int main` that returns "success" (code zero) to the operating system if it successfully finishes execution.

But let's not get ahead of ourselves. The purpose of the traditional "Hello, world!" program is not to discuss a language's capabilities, but instead to get you started on writing and running programs using that language. If you don't have some IDE offering transparent builds, the command line is an easy route. After you have typed the code above in a file called, say, `hello.d`, fire a shell and type the following commands:

```
$ dmd hello.d
$ ./hello
Hello, world!
$ _
```

where $ stands in for your command prompt (it could be `C:\Path\To\Dir>` on Windows or `/path/to/dir%` on Unix systems, such as OSX, Linux, and Cygwin). You can even get the program to compile and run automatically if you apply a bit of your system-fu skills. On Windows, you may want to associate the shell command "Run" with the program `rdmd.exe`, which is part of the installation. Unix-like systems support the "shebang notation" for launching scripts, syntax that D understands; adding the line

```
#!/usr/bin/rdmd
```

to the very beginning of your `hello.d` program makes it directly executable. After you make that change, you can simply type at the command prompt:

```
$ chmod u+x hello.d
$ ./hello.d
Hello, world!
$ _
```

(You need to do the `chmod` thing only once.)

On all operating systems, the `rdmd` program is smart enough to cache the generated executable, such that compilation is actually done only after you've changed the program, not every time you run it. This, combined with the fact that the compiler proper is very fast, fosters a rapid edit-run cycle that helps short scripts and large programs alike.

The program itself starts with the directive

```
import std.stdio;
```

which instructs the compiler to look for a module called `std.stdio` and make its symbols available for use. `import` is akin to the `#include` preprocessor directive found in C and C++ but is closer in semantics to Python's `import`: there is no textual inclusion taking place—just a symbol table acquisition. Repeated `import`s of the same file are of no import.

Per the venerable tradition established by C, a D program consists of a collection of declarations spread across multiple files. The declarations can introduce, among other things, types, functions, and data. Our first program defines the `main` function to take no arguments and return "nothingness"—`void`, that is. When invoked, `main` calls the `writeln` function (which, of course, was cunningly defined by the `std.stdio` module), passing it a constant string. The `ln` suffix indicates that `writeln` appends a newline to the printed text.

The following sections provide a quick drive through Deeville. Little illustrative programs introduce basic language concepts. At this point the emphasis is on conveying a feel for the language, rather than giving pedantic definitions. Later chapters will treat each part of the language in greater detail.

1.1 Numbers and Expressions

Are you ever curious how tall foreigners are? Let's write a simple program that displays a range of usual heights in feet + inches and in centimeters.

```
/*
  Compute heights in centimeters for a range of heights
  expressed in feet and inches
*/
import std.stdio;

void main() {
   // Values unlikely to change soon
   immutable inchesPerFoot = 12;
   immutable cmPerInch = 2.54;

   // Loop'n write
   foreach (feet; 5 .. 7) {
      foreach (inches; 0 .. inchesPerFoot) {
         writeln(feet, "'", inches, "''\t",
            (feet * inchesPerFoot + inches) * cmPerInch);
      }
   }
}
```

When executed, this program will print a nice two-column list:

```
5'0''   152.4
5'1''   154.94
5'2''   157.48
...
6'10''  208.28
6'11''  210.82
```

The construct `foreach (feet; 5 .. 7) { ... }` is an iteration statement that defines an integer variable `feet` and binds it in turn to 5 and then 6, but not 7 (the interval is open to the right).

Just like Java, C++, and C#, D supports `/*multiline comments*/` and `//single-line comments` (plus documentation comments, which we'll get to later). One more interesting detail is the way our little program introduces its data. First, there are two constants:

```
immutable inchesPerFoot = 12;
immutable cmPerInch = 2.54;
```

Constants that will never, ever change are introduced with the keyword `immutable`. Constants, as well as variables, don't need to have a manifest type; the actual type can be inferred from the value with which the symbol is initialized. In this case, the literal 12 tells the compiler that `inchesPerFoot` is an integer (denoted in D with the familiar `int`); similarly, the literal `2.54` causes `cmPerInch` to be a floating-point constant (of type `double`). Going forth, we notice that the definitions of `feet` and `inches` avail themselves of the same magic, because they look like variables all right, yet have no explicit type adornments. That doesn't make the program any less safe than one that states:

```
immutable int inchesPerFoot = 12;
immutable double cmPerInch = 2.54;
...
foreach (int feet; 5 .. 7) {
   ...
}
```

and so on, only less redundant. The compiler allows omitting type declarations only when types can be unambiguously inferred from context. But now that types have come up, let's pause for a minute and see what numeric types are available.

In order of increasing size, the signed integral types include `byte`, `short`, `int`, and `long`, having sizes of exactly 8, 16, 32, and 64 bits, respectively. Each of these types has an unsigned counterpart of the same size, named following a simple rule: `ubyte`, `ushort`, `uint`, and `ulong`. (There is no "unsigned" modifier as in C.) Floating-point types consist of `float` (32-bit IEEE 754 single-precision number), `double` (64-bit IEEE 754), and `real` (which is as large as the machine's floating-point registers can go, but no less than 64 bits; for example, on Intel machines `real` is a so-called IEEE 754 double-extended 79-bit format).

Getting back to the sane realm of integral numbers, literals such as 42 can be assigned to any numeric type, but note that the compiler checks whether the target type is actually large enough to accommodate that value. So the declaration

```
immutable byte inchesPerFoot = 12;
```

is as good as the one omitting `byte` because `12` fits as comfortably in 8 bits as in 32. By default, if the target type is to be deduced from the number (as in the sample program), integral constants have type `int` and floating-point constants have type `double`.

Using these types, you can build a lot of expressions in D using arithmetic operators and functions. The operators and their precedence are much like the ones you'd find in D's sibling languages: `+`, `-`, `*`, `/`, and `%` for basic arithmetic, `==`, `!=`, `<`, `>`, `<=`, `>=` for comparisons, `fun(argument1, argument2)` for function calls, and so on.

Getting back to our centimeters-to-inches program, there are two noteworthy details about the call to `writeln`. One is that `writeln` takes five arguments (as opposed to one in the program that opened the hailing frequencies). Much like the I/O facilities found in Pascal (writeln), C (printf), or C++ (cout), D's writeln function accepts a variable number of arguments (it is a "variadic function"). In D, however, users can define their own variadic functions (unlike in Pascal) that are always typesafe (unlike in C) without needing to gratuitously hijack operators (unlike in C++). The other detail is that our call to writeln awkwardly mixes formatting information with the data being formatted. Separating data from presentation is often desirable, so let's use the formatted write function writefln instead:

```
writefln("%s'%s''\t%s", feet, inches,
   (feet * inchesPerFoot + inches) * cmPerInch);
```

The newly arranged call produces exactly the same output, with the difference that writefln's first argument describes the format entirely. % introduces a format specifier similar to C's printf, for example, %i for integers, %f for floating-point numbers, and %s for strings.

If you've used printf, you'd feel right at home were it not for an odd detail: we're printing ints and doubles here—how come they are both described with the %s specifier, which traditionally describes only strings? The answer is simple. D's variadic argument facility gives writefln access to the actual argument types passed, a setup that has two nice consequences: (1) the meaning of %s could be expanded to "whatever the argument's default string representation is," and (2) if you don't match the format specifier with the actual argument types, you get a clean-cut error instead of the weird behavior specific to misformatted printf calls (to say nothing about the security exploits made possible by printf calls with untrusted format strings).

1.2 Statements

In D, just as in its sibling languages, any expression followed by a semicolon is a statement (for example, the "Hello, world!" program's call to writeln has a ; right after it). The effect of the statement is to simply evaluate the expression.

D is a member of the "curly-braces block-scoped" family, meaning that you can group several statements into one by surrounding them with { and }—something that's necessary, for example, when you want to do several things inside a foreach loop. In the case of exactly one statement, you can omit the curly braces entirely. In fact, our entire height conversion double loop could be rewritten as follows:

```
foreach (feet; 5 .. 7)
  foreach (inches; 0 .. inchesPerFoot)
    writefln("%s'%s''\t%s", feet, inches,
      (feet * inchesPerFoot + inches) * cmPerInch);
```

Omitting braces for single statements has the advantage of shorter code and the disadvantage of making edits more fiddly (during code maintenance, you'll need to add or remove braces as you mess with statements). People tend to be pretty divided when it comes to rules for indentation and for placing curly braces. In fact, so long as you're consistent, these things are not as important as they might seem, and as a proof, the style used in this book (full bracing even for single statements, opening braces on the introducing line, and closing braces on their own lines) is, for typographical reasons, quite different from the author's style in everyday code. If he could do this without turning into a werewolf, so could anyone.

The Python language made popular a different style of expressing block structure by means of indentation—"form follows structure" at its best. Whitespace that matters is an odd proposition for programmers of some other languages, but Python programmers swear by it. D normally ignores whitespace but is especially designed to be easily parsed (e.g., parsing does not need to understand the meaning of symbols), which suggests that a nice pet project could implement a simple preprocessor allowing usage of Python indentation style with D without suffering any inconvenience in the process of compiling, running, and debugging programs.

The code samples above also introduced the `if` statement. The general form should be very familiar:

```
if (‹expression›) ‹statement₁› else ‹statement₂›
```

A nice theoretical result known as the *theorem of structure* [10] proves that we can implement any algorithm using compound statements, `if` tests, and loops à la `for` and `foreach`. Of course, any realistic language would offer more than just that, and D is no exception, but for now let's declare ourselves content as far as statements go and move on.

1.3 Function Basics

Let's go beyond the required definition of the `main` function and see how to define other functions in D. Function definitions follow the model found in other Algol-like languages: first comes the return type, then the function's name, and finally the formal parameters[1] as a parenthesized comma-separated list. For example, to define a function called `pow` that takes a `double` and an `int` and returns a `double`, you'd write

```
double pow(double base, int exponent) {
  ...
}
```

Each function parameter (`base` and `exponent` in the example above) has, in addition to its type, an optional *storage class* that decides the way arguments are passed to

1. This book consistently uses *parameter* to refer to the value accepted and used inside the function and *argument* when talking about the value passed from the outside to the function during invocation.