An overview of D for embedded development

Abstract

The D programming language is a multi-paradigm successor to C/C++, which borrows most of the syntax while overhauling all the semantics. It has a larger runtime, a garbage collector, greater features for memory safety, and, being designed for systems programming, the option to abandon many of these features to write essentially an upgraded version of C. While the D language supports programming for embedded systems, ultimately, its overall design clashes with the goal of lightweight software for small hardware.

Motivation

Our goals for choosing a language are such:

1. clean software, easy to audit and maintain
2. freestanding program; no OS needed
3. lightweight and minimal; stripped runtime
4. interface with low-level hardware

We will judge D based on how well it meets these goals.

Overview of D

D, version 2.095.0, comes with many modern, high-level features that are common for languages its age. It supports many paradigms - while primarily imperative and object-oriented, functional features are present, and the standard library supports parallel/-concurrent models of programming. The overview of D on the language website [9] essentially describes D as a language as powerful as C/C++, but easier to write/read, more explicit, more safe ("[D is for] people who compile with maximum warning levels turned on"), and more usable ("...these goals will conflict. Resolution will be in favor of usability.").

D comes with exception handling, garbage collection/RAII (either can be chosen), runtime type idenification, templates, and most of the features that C and C++ provide. Multiple inheritance is left out - and you can't write C family code directly in a D program like in C++. However, D's foreign function interface (which resembles that of C's) allows you to easily run C/C++ code in D programs with little effort, as well as vice-versa.

Written D

Like C/C++, D is statically typed. Its type system offers many improvements in expressive power over C, culminating in attributes and function types. D has two type qualifiifers, for instance - `const` and `immutable` specify different ways data can be made invariant, with the latter opening the door to program optimization, and with both improving the specificity of D software.

D's attributes look and write like C's, except that there are more of them, and that they have extended functions. For example, while lifetimes are implicitly present in all of C, C++, and D, D provides a language feature `return ref` that restricts a returned reference from outliving its matching argument (code adapted from [6]):

```
ref int id(return ref int x) {
  return x;
}
```

```
ref int fun() {
  int x;
  // Err: escaping reference to x
  return id(x);
}

ref int gun(return ref int x) {
  return id(x); // OK
}
```

Attributes like these facilitate a reputedly memory-safe subset of D known as "SafeD". Some attributes annotate function safety - `@safe`, `@trusted`, `@system` - others, like `return`, control scope and lifetimes. These explicit annotations help programmers keep track of the flow of data and reason about their program's function.

D also supports some functional restraints - for example, you can specify that a function is `pure`, preventing it from mutating program state and from calling impure functions.

Overall, D provides a marginally smoother experience than C when it comes to writing expressive and explicit programs that bare their semantics readily. While the language is somewhat dated by now, D still evolves in its expressiveness - for example, the core creator of the language recently [2] suggested adding annotations for memory ownership and borrowing (akin to those of Rust's) into D, improving its memory safety at no performance cost.

Independence

D was not originally meant for usage without an operating system. From stackoverflow user 'larsivi', project leader of Tango (historically one of the main D standard libraries) [10]:

> "All current D libraries requires a C standard library below it, and thus typically also an OS, so even that works against using D. However, there do exist experimental kernels in D, so it is not impossible per se. There just wouldn't be any libraries for it, as of today."

While the quote is by now severely outdated, it is telling that D was not designed with a significant focus on embedded development. (For example, D still does not provide any support for 16-bit processors [9].)

It is actually challenging to find any prior examples of D software written for hardware sans OS. The language's wiki has many articles concerning "bare metal" embedded compilers - for Linux hosts, that is (these usually concern setting up the D runtime on embedded devices as in [11] and [3]). Looking through the D forum, there are few examples of D embedded projects, and those without operating systems are even scarcer (I failed to find any such examples).

Runtime

D is packaged with a significantly larger runtime. The base of 'D' could be said to be a C#/Java/Go by default, and a C/C++/Rust by choice. This is to say that base D comes with a garbage collector, and a runtime which is packaged with the standard compiler (dmd). But despite this, D has eased up its attachment to its runtime [1] and advanced sufficiently enough to the point where disabling runtime features is easy, though tedious as the runtime is typically a core part of the language.

This is most obvious when looking at all the features that rely on the GC [7]:

- *NewExpression* (allocation with `new`)
- Array appending/concatenation
- Array literals (except when used to initialize static data)
- Associative arrays
- Taking the address of (i.e. making a delegate to) a nested function that accesses variables in an outer scope
- A function literal that accesses variables in an outer scope

Those who need dynamically sized containers will have to manually allocate/manage memory for them, if not write their own APIs in the first place.

D supports writing custom allocators: the Phobos standard library includes `std.experimental.allocator` which provides interfaces for allocation, as well as bindings to C++'s own allocator. When the garbage collector is not used, RAII and explicit memory management are the dominant paradigms of control over state, similar to C and C++.

It is this attachment to C/C++ that is most notable for D's strategy of memory management. D provides an easy interface for calling C/C++ functions (and vice versa); it packages much of the C and C++ standard libraries in Phobos; the `dmd` compiler (standard compiler) even has a `-betterC` flag - in which the D code written is essentially C (as a subset of D), but with a few ergonomics features attached - for example, the D writing experience, RAII, metaprogramming, bounds checking, etc [5].

Overall, most standard libraries built for D (Phobos and Tango) assume the use of the garbage collector - teams who want to distance themselves from the runtime will need to provide their own implementations of library functionality.

Interface

The D standard library (Phobos) comes with modules for interfacing with hardware, like any systems programming language should. D provides an inline assembler, which is "standardized for D implementations across the same CPU family" [8], if it is needed. There are modules in `core` of the standard library that simply provide bindings for C and C++ functions; anything needed that isn't available in D can be repurposed from C.

The most inconvenient aspect of D is its lack of popularity. There is little information readily available about D's ability to interface with hardware - this is because barely anyone has tried it. For a language which should be more mature than its younger competitors, there has been a stunning lack of growth and thus exploration on these parts for D.

The LLVM backend compiler LDC allows D to support a wider range of CPU architectures, including those running ARM and MIPS [4]. Unfortunately, I could find few examples of such usage - which can be chalked down mostly to D's relative obscurity. This obscurity may result in challenges not experienced with other, more fleshed-out languages. The LDC compiler is, after all, not as commonly used as DMD, and not part of the GNU compiler collection in GDC - there are a few levels of indirection there.

Summary

We conclude that while D is a fine language, being truly general purpose and multi-paradigm, it fails to provide the optimal experience for bare-metal development akin to that necessary for the proposed application. Its features are quite impressive - but these are features that mostly concern the D runtime, and if minimality is a goal, D adds friction programmers must smooth over in trying to strip the language down to its barest. Of course, D is not incapable - it has many fine features, especially those regarding program annotation - but its prime claim to memory safety is found in its garbage collector; without that, it resembles an advanced, more flexible form of C - as its language foundation puts it, "betterC". Writing in D, one can expect an improved experience compared to plain C, and if desired, most program functionality can be in C/C++, as it is easy to interface between the two languages, as with assembly. In total, D is a good choice for the target application, but perhaps not the best.

References

[1] Walter Bright. *Changelog: 2.079.0*. URL: https://dlang.org/changelog/2.079.0.html. (accessed: 3/12/2021).

[2] Walter Bright. *Ownership and Borrowing in D*. URL: https://dlang.org/blog/2019/07/15/ownership-and-borrowing-in-d/. (accessed: 3/11/2021).

[3] Dangbinhoo. *Programming in D tutorial on Embedded Linux ARM devices*. URL: https://wiki.dlang.org/Programming_in_D_tutorial_on_Embedded_Linux_ARM_devices. (accesed: 3/12/2021).

[4] D Language Foundation. *Areas of D Usage*. URL: https://dlang.org/areas-of-d-usage.html#system-programming. (accessed: 3/12/2021).

[5] D Language Foundation. *Better C*. URL: https://dlang.org/spec/betterc.html. (accessed: 3/12/2021).

[6] D Language Foundation. *Functions*. URL: https://dlang.org/spec/function.html. (accessed: 3/11/2021).

[7] D Language Foundation. *Garbage Collection*. URL: https://dlang.org/spec/garbage.html. (accessed: 3/12/2021).

[8] D Language Foundation. *Inline Assembler*. URL: https://dlang.org/spec/iasm.html. (accessed: 3/12/2021).

[9] D Language Foundation. *Overview*. URL: https://dlang.org/overview.html. (accessed: 3/11/2021).

[10] Lars Ivar Igesund. *Getting Embedded with D (the programming language)*. URL: https://stackoverflow.com/questions/1207958/getting-embedded-with-d-the-programming-language. (answer posted: 7/30/2009; accessed: 3/11/2021).

[11] Verax. *Bare Metal ARM Cortex-M GDC Cross Compiler*. URL: https://wiki.dlang.org/Bare_Metal_ARM_Cortex-M_GDC_Cross_Compiler. (accesed: 3/12/2021).