Origins of the D Programming Language

WALTER BRIGHT, The D Language Foundation, USA
ANDREI ALEXANDRESCU, The D Language Foundation, USA
MICHAEL PARKER, The D Language Foundation, USA

Shepherd: Roberto Ierusalimschy, PUC-Rio, Brazil

As its name suggests, the initial motivation for the D programming language was to improve on C and C++ while keeping their spirit. The D language was to preserve the efficiency, low-level access, and Algol-style syntax of those languages. The areas D set out to improve focused initially on rapid development, convenience, and simplifying the syntax without hampering expressiveness.

The genesis of D has its peculiarities, as is the case with many other languages. Walter Bright, D's creator, is a mechanical engineer by education who started out working for Boeing designing gearboxes for the 757. He was programming games on the side and, in trying to make his game Empire run faster, became interested in compilers. Despite having no experience, Walter set out in 1982 to implement a compiler that produced better code than those on the market at the time.

This interest materialized into a C compiler, followed by compilers for C++, Java, and JavaScript. The best known of these would be the Zortech C++ compiler, the only C++-to-native compiler to have been developed by a single person. The D programming language began in 1999 as an effort to pull the best features of these languages into a new one. Fittingly, D would use the mature C/C++ back end (optimizer and code generator) that had been under continued development and maintenance since 1982.

Between 1999 and 2006, Walter worked alone on the D language definition and its implementation, although a steadily increasing volume of patches from users was incorporated. The new language would be based on the past successes of the languages he had used and implemented, but would be clearly looking to the future. D started with choices that are obvious today but were less clear winners back in the 1990s: full support for Unicode, IEEE floating point, two's complement arithmetic, and flat memory addressing (memory is treated as a linear address space with no segmentation). It would do away with certain compromises from past languages imposed by shortages of memory (for example, forward declarations would not be required). It would primarily appeal to C and C++ users, as expertise with those languages would be readily transferable. The interface with C was designed to be zero cost.

The language design was begun in late 1999. An alpha version appeared in 2001 and the initial language was completed, somewhat arbitrarily, at version 1.0 in January 2007. During that time, the language evolved considerably, both in capability and in the accretion of a substantial worldwide community that became increasingly involved with contributing. The front end was open-sourced in April 2002, and the back end was donated by Symantec to the open source community in 2017. Meanwhile, two additional open-source back ends became mature in the 2010s: `gdc` (using the same back end as the GNU C++ compiler) and `ldc` (using the LLVM back end).

The increasing use of the D language in the 2010s created an impetus for formalization and development management. To that end, the D Language Foundation was created in September 2015 as a nonprofit corporation

Authors' addresses: Walter Bright, The D Language Foundation, 6830 NE Bothell Way, Suite C-162, Kenmore, WA, 98028, USA, walter@dlang.org; Andrei Alexandrescu, The D Language Foundation, 6830 NE Bothell Way, Suite C-162, Kenmore, WA, 98028, USA, andrei@dlang.org; Michael Parker, The D Language Foundation, 6830 NE Bothell Way, Suite C-162, Kenmore, WA, 98028, USA, social@dlang.org.

2475-1421/2020/6-ART73
https://doi.org/10.1145/3386323

overseeing work on D's definition and implementation, publications, conferences, and collaborations with universities.

CCS Concepts: • **Software and its engineering** → **General programming languages**; • **Social and professional topics** → *History of programming languages.*

Additional Key Words and Phrases: Programming Languages

ACM Reference Format:
Walter Bright, Andrei Alexandrescu, and Michael Parker. 2020. Origins of the D Programming Language. *Proc. ACM Program. Lang.* 4, HOPL, Article 73 (June 2020), 38 pages. https://doi.org/10.1145/3386323

Contents

1 THE SEATTLE SUMMIT

On August 23, 2007, over three dozen developers descended upon the Amazon campus in Seattle for the first official D Programming Language Conference [NA Roberts 2007a]. Organized by Brad Roberts, a D contributor and Amazon employee, the three-day event had been a year in the making [NA Bright 2006a]. To the D community, it was a sign that, after seven years of development and community building, D was finally going places.

D is a general-purpose programming language with support for procedural, object-oriented, generic, generative, and functional programming. As of 2020, the language is employed by a variety of companies [NA D Language Foundation 2020b] in industry, systems programming, research, and academia [NA D Language Foundation 2020a]. But in 2007, it was barely a blip on anyone's radar and had yet to be proven in production. D had no sponsors and no funding. The conference was organized on a shoestring budget [NA Roberts 2006] and likely would not have happened at all had Amazon not agreed to make its conference rooms available [NA Roberts 2007b]. It was truly a community-driven language.

Walter Bright and Andrei Alexandrescu opened and closed the conference with joint presentations on their plans for D's future [NA Bright and Alexandrescu 2007]. Under headings such as "Object Model Improvements" and "Simplifying Code", they laid out a set of new language features that were more transformative than the headings implied. And on the second day, while Bartosz Milewski was starting his talk about software transactional memory in D, Walter and three members of the D community found a quiet place to sit and discuss some differences they had about D's runtime library.

From 1999 until Andrei's involvement in 2006, Walter had been the sole developer and maintainer of the D language. Community contributions had come in at a steadily increasing rate. Accustomed to running his own show, Walter was unprepared to cope with the demands of a growing community. Over time, the project had grown too big for one person, and he struggled with delegating responsibility for the myriad development tasks it required. Parts of it had become neglected, most notably Phobos, the D runtime library. Frustration with its staleness [NA Wrede 2006] and lack of consistency [NA Hay 2006] led three D enthusiasts to begin work on Tango, an alternative D runtime library, in 2006 [NA Kelly 2006].

The first public release of Tango was announced on Jan 31, 2007 [NA Igesund 2007]. The announcement was greeted with positive reactions, even from Walter [NA Bright 2007d], so development proceeded apace. Unfortunately, Phobos and Tango were incompatible [NA BCS 2007; Hasemann 2007], creating an environment where developers were forced to choose one or the other for new D

projects. This was a particularly difficult choice for library authors and became a source of friction among D users [NA Burrell 2008]. In April, 2007, one Tango supporter announced Tangobos, a library that aimed to bridge the gap [NA Richards 2007], but it failed to gain traction.

How the meeting on the sidelines of the 2007 conference came about is lost to the fog of time. It is possible that it was suggested, if not brokered, by a D community member prior to the conference. What is certain is that the Tango team extended an invitation to Walter and he accepted. The Tango team were ideally hoping for Walter to agree to a merger with or wholesale replacement of Phobos [NA Igesund 2018], but Walter was adamantly opposed. He felt Phobos was and should remain the official D runtime library. Somewhere in the middle of their practical and philosophical differences, a compromise was waiting for the two sides to arrive.

What Walter could not have foreseen was that the community rift that had arisen over Tango and Phobos was a minor hiccup compared to what was to come. The future language features Walter and Andrei presented at the conference were just the tip of the iceberg. Radical changes were coming to D, and the community response would be such that in 2016 Walter would write [NA Bright 2016]:

> There are no plans for D3 at the moment. All plans for improvement are backwards compatible as much as possible. D had its wrenching change with D1->D2, and it nearly destroyed us.

The D language the conference attendees had come to know was quite different from the D language of today. It was a one-man project, its history inextricably bound with Walter's experience in computer programming, developing compilers, supporting compilers, working on teams, and even participating in aircraft design.

It all started with a computer game.

2 THE FORMATIVE YEARS

The D programming language first came to life in 1999, but it was in the late 70s that Walter, while working toward a Mechanical Engineering degree at Caltech, stumbled on to the path that ultimately led to the genesis of the language. In an era when students would spend hours playing games in university computer labs across the country, Walter discovered he was more interested in learning how to make games than in playing them.

Lacking any formal training in Computer Science, he taught himself BASIC in pursuit of his new hobby. When programs began to exceed the memory available to BASIC, he switched to Fortran-10 (on Caltech's PDP-10), allowing him to create much faster and larger programs. One such was the commercially successful game Empire [NA Bright 1979].

It was his desire to improve the performance of Empire that would launch Walter on a career of compiler development and, ultimately, the creation of the D language. Before that came to pass, he would learn some unexpected lessons as a mechanical engineer.

2.1 Boeing Commercial Airplane Company

Walter left Caltech in 1979 and landed at Boeing, where he would remain until 1982. He was put to work on the design of the 757, with a focus on flight-critical systems such as the stabilizer trim system. His time with the company introduced him to the concepts underpinning the development of safe systems from unreliable parts—all systems can fail, and the failure of any one system must not impair the safe flying of the airplane. Such safety is achieved through the implementation of backups for all critical systems, along with mechanisms for the detection of faults and failover to the backup [NA Bright 2009b,c].

At Boeing, 10,000 engineers could work together on a system as complicated as the 757 and complete its design on schedule. One key enabler was modularity—each subsystem is compartmentalized behind a very well-defined interface.

Conversations with flight control engineers revealed an understanding of what is and what is not "intuitive" about user experience design. An intuitive design is one in which the user's natural reaction is the correct thing to do, with the caveat that one's natural reaction is based on one's previous experience in similar situations. Intuitive design is difficult to determine in advance, and airliner cockpit design is often the result of a cycle of mistake and accident analysis [NA Vaillot et al. 2003] followed by the implementation of improvements.

Another important lesson from airliner design is the attitude toward, and approach to, human error. Many programming languages frame programmer error primarily as a matter of education. Aviation design assumes that people (factory workers, pilots, and maintenance engineers), no matter how careful and well trained, will make mistakes. That assumption guides the entire design process toward creating artifacts that are easy to use correctly and difficult to use incorrectly.

For example, consider a square assembly that has four bolts holding it in place. One would immediately think to design the four bolts in a square as well, for perfect symmetry. An unintended consequence is that the assembly can be installed in four different orientations. If only one of those is correct, relying on the mechanic to choose the right one introduces the risk of human error. The solution is to break symmetry by offsetting one of the holes and/or by making one bolt hole a different size than the others. Consequently, only one assembly is possible—the correct one. Many other foolproof designs in the aerospace industry (such as color coding, asymmetric threading, size/shape matching) render invalid assemblies or combinations impossible or visibly awkward.

These lessons would have a lasting impact on Walter, influencing his thoughts about the design of existing programming languages and, two decades later, the design of D. But first, there was the matter of improving the performance of Empire.

2.2 A Better C Compiler

The growing popularity of the IBM PC in the early 1980s offered Walter the opportunity to port his Empire game to the platform, which in turn prompted him to examine the state of available compilers. Implementations of Pascal [Wirth 1971] and Fortran in the early days of the IBM PC were of poor quality. In contrast, early implementations of C, such as Telecom C, were quite useful with the 16-bit memory model of IBM PC DOS. Still, Walter found the optimization characteristics of existing C compiler implementations wanting, so he set out to write one that could do better.

In 1982, Walter left Boeing and created the Northwest Software company to sell his new C compiler, which he branded Northwest C. Eager to make Empire faster and knowing what he didn't know, in 1984 he decided he should formally learn more about compiler implementation. He signed up for a two-week Stanford course in compiler construction taught by John Hennessy, Susan Graham, and Jeffrey Ullman. Of particular influence was the portion devoted to Data Flow Analysis (DFA). The knowledge he gained provided insight into the kinds of optimizations a compiler could be expected to perform, what was unreasonable for a compiler to do, and the kinds of language features that contributed to and detracted from better optimization.

In April, 1985, Walter entered into a contract with the software company Datalight [NA Wikipedia 2010] and his compiler was rebranded as Datalight C. Eventually, he found the opportunity to apply some of the insights he had derived from the compiler course. In 1987, he implemented DFA in the Datalight C compiler and changed its name to Datalight Optimum C.

Data flow analysis was new for MS-DOS compilers, so Optimum C had a competitive edge. This attracted the interest of a company called Zorland, which took over the contract of the compiler in February, 1988. They began selling the compiler as Zorland C and retained Walter to maintain it.

2.3 The C++ Compiler

Bjarne Stroustrup's book *The C++ Programming Language* [Stroustrup 1985] had convinced Walter in 1987 that the market was ripe for a native, high-performance C++ compiler on MS-DOS. The extant compilers were marred by disadvantages. g++ was in beta with an incomplete feature set and was only available on Unix. `cfront`, which translated C++ to C, required an expensive third-party C compiler to compile its output. It was also slow and unsuitable for the 16-bit memory model as it was designed for a flat memory space, not the segmented world of MS-DOS where near and far pointers were required. Walter began to implement support for C++ in Zorland C.

In 1988, in order to avoid legal issues with Borland, Zorland's name was changed to Zortech. In October, 1988, Zortech C++ 1.05 [NA EDM/2 2017] became the first native C++ compiler for DOS [PC-Week 1988a], and possibly the first native end-to-end C++ compiler released for any platform. Featuring a new front end for the mature Zorland C back end, the overnight success of Zortech C++ improved the landscape of DOS and Windows programming and the popularity of C++ surged along with it [PC-Week 1988b].

In March, 1990, Zortech C++ 2.0 was released with support for Windows and OS/2. Version 3.0 was released in June, 1991. In August, Zortech was acquired by Symantec for $12.6 million [PC-Week 1991]. Symantec continued to market Zortech C++ until 1993 [NA EDM/2 2017]. In September of that year, the compiler was rebranded and released as Symantec C++ [NA EDM/2 2018]. Walter would remain the primary developer and maintainer of the Windows version of the compiler until its final release.

Symantec's primary focus with the C++ compiler was drop-in compatibility with Microsoft C++. In the days before the C++ Standard, when compilers did not agree on how to interpret the rules of the C++ language, maintaining compatibility consumed a great deal of Walter's time. As part of that effort, he personally provided support to Symantec C++ customers. Often, this took place outside of the official Symantec tech support channels through CompuServe and Usenet. This was not a normal practice, and users were frequently surprised to have their questions answered directly by the compiler maintainer.

Walter also sometimes provided support via email. This was how he established a relationship with Jan Knepper, who had first begun using the compiler before Symantec acquired it. The two would remain in contact and Jan would later become an early, and important, contributor to the D programming language.

2.4 The Java Compiler

Symantec decided to get into the Java compiler business and in May, 1996, released the beta version of their Java development suite, Symantec Café. In 1997, Walter was asked to reimplement the existing Sun Java compiler in C++ in order to speed up compilation.

A key component of Java is the garbage collector (GC). Walter had never implemented a GC. To get up to speed, he turned to *Garbage Collection: Algorithms for Automatic Dynamic Memory Management* [Jones 1996], the authoritative source on garbage collection at the time. This provided him with the information he needed to implement a conventional mark/sweep collector. Afterward, he was able to convert the implementation to a generational collector using an algorithm known as *mostly-copying collection* [Bartlett 1989].

Prior to this experience, Walter had been convinced that garbage collection was a performance killer to be avoided. Having acquired a more thorough understanding of the subject, he recognized a number of potential benefits [NA Bright 2001d] that would influence his thoughts on language design going forward.

- Counterintuitively, and for numerous reasons, garbage-collected programs can be faster.

- By reclaiming unused memory, garbage collectors are less prone to memory leaks, bringing stability to long-running programs.
- Garbage-collected programs have fewer hard-to-find pointer bugs.
- Without the burden of manual memory management, garbage-collected programs can be faster to develop and debug.
- For the same reason, garbage-collected programs can be smaller in size.

Walter would continue to maintain Symantec Café alongside Symantec C++ until he left the company.

3 DIGITAL MARS

In 1999, Symantec exited the compiler business. With no C++ compiler to maintain, Walter retired from the workforce. It wasn't long before he grew bored. He was itching to work on a new project. He had often reflected upon the strengths and weaknesses of C++ and the lessons that one could derive from them in the design of a new language. He decided it was time to put those ideas into practice. In October, 1999, he created a new company, Digital Mars. In November of that year, he began work on the design and implementation of a new programming language that he called *Mars*.

To save time on the implementation of the compiler, he decided to use the back end of the C and C++ compiler he had developed and maintained since the early 80's, as it could already generate binary output for the Windows platform. This required obtaining a license for the compiler from Symantec, which he was able to do in April, 2000.

When former Symantec C++ customer Jan Knepper heard that Walter was preparing to relaunch the compiler under a new brand, he decided to switch to the new release. He offered to host the Digital Mars website and newsgroups. Walter accepted and, with the C++ compiler license in hand, began offering the rebranded Digital Mars C++ as a free download from digitalmars.com [NA Digital Mars 1999] and selling a full development suite that included an IDE and other tools. (As of 2019, Jan continues to host digitalmars.com along with dlang.org.)

3.1 A Brief Interlude

In the spring of 2000, Walter was invited by friend and colleague Eric Engstrom to join his new startup, Chromium [NA Orlowski 2001]. The company was under contract from Sun Microsystems to develop a JavaScript implementation and Walter was asked to get it done. Despite the work he was already doing in running a business and designing a programming language, Walter accepted.

This project provided a second opportunity for Walter to develop a garbage collector. Building upon his previous experience, he chose to implement a conservative collector, as he was confident he could produce a reliable one in short order. It would ultimately serve his own interests. He left the company in the fall of 2001 when the script engine was complete, but was able to license it from Chromium for his own use. The GC he developed for the JavaScript implementation would be repurposed and employed in the runtime library of his new language.

3.2 From Mars to D

Walter began talking to acquaintances about Mars and looking for feedback. It wasn't long before they were jokingly referring to it as "D." The name stuck, and on December 8, 2001, he released the first prototype of DMD, the Digital Mars D compiler [NA Bright 2001a].

The draft specification of the D Programming Language [NA Bright 2001b] opens with the following paragraphs.

> The software industry has come a long way since the C language was invented. Many new concepts were added to the language with C++, but backwards compatibility with

> C was maintained. Additionally, C++ was very much designed by committee, and hence is loaded with multiple overlapping ways of doing the same thing, and compromises for political purposes. C++ has become so complicated that nobody really understands it, and no C++ compiler exists that properly implements 100% of the spec (even if someone did understand 100% of the spec).
> Software has grown thousands of times more complex. What is needed is for the language to be forward looking to solve today's and future software programming needs, not looking backwards to bring along code written 30 years ago. For legacy code written 30 years ago, they can be ably compiled by existing C and C++ compilers.
> A new language needs to be developed that takes the best features and capabilities of C++, adds in modern features that are impractical in C++, and puts it all in a package that is easy for compiler writers to implement, and which enables compilers to easily generate optimized code.

The initial design of D was informed by the lessons learned from Walter's careers as a mechanical engineer and compiler writer, and those derived from his exposure to different programming languages. Having written and supported professional compilers for C, C++, Java, and JavaScript for two decades, he felt he was in a good position to judge their strengths and weaknesses and what could be improved upon. D would retain the strengths of its ancestors, avoid the problems that caused bugs and awkward code, and add new capabilities that would make programming more pleasant and more reliable. It was to be a general purpose, native-code-generating language for a broad spectrum of uses, fit for systems programming and application programming.

D was intended for more experienced programmers who wanted the most out of a language and were willing to commit some time to learning it. This would be facilitated by maintaining a syntax with which experienced programmers would be familiar from C, C++, and Java. Though D was not initially considered suitable for beginner or casual programmers, it would become more amenable to them over time with the development of comprehensive documentation and tutorials by a thriving community of D users.

3.3 You Can't Spell "Compatibility" Without 'C'

One of the earliest design decisions that Walter made about D was that it would be easy to use with software written in C. Many widely-used libraries are implemented in C or have a C interface. He wanted to provide an easy path for established software companies to adopt the D language. A straightforward approach was to guarantee that users of D could immediately take advantage of any C library their project required without the need to reimplement it in D.

To facilitate this goal, D maintains interface compatibility with C, which allows C headers to be translated to D modules so that C symbols can be used in D code. D is also ABI-compatible with C which means that the binary output of C compilers can be combined with the binary output of a D compiler in a single executable.

Going further, a fundamental design goal is that a syntactical construction should have the same semantics in D as in C, such as the integer promotion rules, or fail to compile. This allows for the by-rote conversion of C code to D. If it compiles, it should run with the same semantics; if not, it should produce a visible (and hopefully easily corrected) compile-time error. In practice, this has not always been desirable. For example, in the declaration of multiple pointers on a single line: `int* p1, p2`. This creates two pointers to `int` in D, whereas in C only `p1` would be a pointer, and is consistent with other D declarations in that the type, in this case `int*`, is always on the left.

Negative influences that were avoided in D include the necessity for forward declarations, the decay of arrays to pointers when passed to functions [NA Bright 2009a], uninitialized variables,

and implicit narrowing conversions (assigning an integer value of one type to a variable of a smaller-sized type without a cast).

Notably absent from D is a C-style preprocessor. In C, a text macro system was necessary in order to add metaprogramming features while keeping the compiler small. Such technology constraints were no longer a problem when work began on D, and the functionality of the preprocessor is covered by several different features in D. Some, such as modules and version declarations, were in place from the beginning, others would be added over time.

3.3.1 Arrays and Slices. Though Walter would have preferred to abandon support for C-style pointers, he found it necessary to support them. Without pointers, the goal of facilitating D's adoption would be severely hampered; users would be unable to make use of any pointer-heavy C API or to easily translate C source to D. In contrast, Walter decided D would not support some C++ notions, such as rvalue references, even though it would make connecting with C++ code more difficult.

The need to support pointers did not mean their use was encouraged. It would instead be discouraged in circumstances where code safety may be compromised. The prime example is the use of pointers to access contiguous arrays; because there is no array extent information embedded in pointers, that information must be maintained separately in user code, a circumstance that collective experience has shown is prone to human error. Tools for modular bounds checking commonly require function signatures to be annotated with information denoting the association between pointer parameters and the lengths of the arrays they represent, or the relationships among pointer parameters [Microsoft 2015]. C++ iterators, being a generalization of pointers, have inherited some of this lack of safety [Pataki et al. 2011].

To address this issue, D arrays would be *fat pointers*, an encapsulated pair of a pointer to a contiguous memory block and a field to track the number of elements residing in memory starting at that address. With this construct, D arrays could offer bounds-checked random access and subslicing (reducing the extent of the slice) "for free," with no complex code instrumentation or type-checking cleverness. In contrast, C code using pointer/length or pointer/pointer pairs to represent arrays requires conventions to convey how pair elements are related and relies on the programmer to combine them correctly.

Again for compatibility, C's postfix array declaration syntax (e.g., `int a[]`) would be retained, but the D prefix syntax `int[] a` would be preferred. The brackets, '`[]`', are considered part of the type, hence conceptually they should remain together. (For the same reason, the D convention for pointer declarations discourages spaces between the element type and the star.)

To the D programmer, an array slice is no different than a dynamic array. They support the same operations, they each have `ptr` and `length` properties, and can be passed as an argument to any function that accepts dynamic arrays. The only difference is a semantic one: a slice is produced from an existing dynamic or static array. A slice shares the same backing memory store as its source array until an element is appended to the slice, in which case new memory is allocated and the contents of the original store are copied over. The similarity between dynamic arrays and slices has led to inconsistent usage of the two terms, with some users maintaining the distinction and others referring to both as slices.

Disagreement over jargon aside, one would be hard pressed to uncover a D programmer complaining about slices as a language feature. Originally suggested by Jan Knepper, they would become a primary tool in the D toolbox. Projects such as the vibe.d web application framework [NA Rejected Software 2012b] would come to rely upon them for fast string parsing, and slices would provide a convenient base upon which ranges and the `std.algorithm` API could be implemented in D2, which was a complete overhaul of the language that began in 2007 [NA Bright 2007b]. The

D2 standard library would also introduce new mechanisms to enhance the performance of slices by giving the programmer more control over the usage of a slice's backing memory store [NA Schveighoffer 2011].

3.3.2 Modules. To replace the usage of C headers and `#include` directives, each D source file is a module that can be imported by other modules. By default, a D module takes the name of its source file without the `.d` extension, but best practice dictates including a `module` declaration, of the form

```
module modulename;
```

at the top of every source file. There is no need to separate function and type declarations from their definitions. By default, symbols in a module are publicly accessible, but can be made internal via the `private` keyword. This is analogous to the usage of `static` in C, but given that D supports access modifiers in `class` and `struct` definitions, where `static` has nothing to do with access control, it made sense to use `private` at module scope instead of `static`.

Modules were in place in the initial alpha release of DMD and were followed by packages with the release of DMD 0.15 on Jan 20, 2002 [NA Digital Mars 2002b]. Whereas modules correspond to files in the file system, packages correspond to directories. Packages may be nested by including each package name in the module declaration:

```
module packageone.packagetwo.modulename;
```

This complete module name is used with the `import` statement to make the symbols from one module accessible to another module.

Any symbol declared in a module is prefixed with the name of the module and any package hierarchy to which the module belongs to form the symbol's Fully-Qualified Name (FQN) in the form `packagename.modulename.symbolName`. The FQN can be used when symbols from two or more imported modules conflict. Over time, the `import` statement would be enhanced to support several variations:

```
// Alias std.stdio to io, so that the FQN of symbol names in that module
// become io.symbolName.
import io = std.stdio;

// Require the fully qualified name to be used on symbols from std.string.
static import std.string;

// Selectively import the toInt function template from std.conv.
import std.conv : toInt;

void main() {
  // getExt and getName are only visible and accessible in this function
  import std.path : getExt, getName;

  {
    // read is only visible and accessible in this block
    import std.file : read;
  }
}
```

3.3.3 Conditional Compilation. Walter's dislike of the C preprocessor did not extend to all of the purposes for which it was put to use. One such usage that C and C++ programmers had come to rely on was the employment of `#if` and friends to conditionally include certain blocks of code in, or exclude them from, compilation. The absence of a feature in D that could achieve the same effect would significantly complicate porting and interacting with some preprocessor-heavy existing C codebases. To that end, the `version` conditional became part of the initial feature set.

Version conditionals accept an identifier and are allowed in only one form:

```
version (identifier) {
  ...
} else {
  ...
}
```

(The `else` branch is optional.) One key detail is the braces '`{`' and '`}`' do *not* introduce scopes here; they are used as punctuation only. The introduction of scopes would have hamstrung usage of `version` by hiding all declarations within the braces from the rest of the code.

The identifier can be set on the command line or programmatically with the syntax:

```
version = identifier;
```

The compiler commonly predefines version identifiers associated with the operating system, target processor, or pointer width. The namespace of versions is distinct from all other identifier namespaces. Setting a version after it has been queried is not allowed. Combining versions in Boolean expressions as in `version(a || b)` is not allowed, although that would not be technically difficult. The limitations are intentional and aim at keeping `version` simple and coarse-grained; decades of experience with the "`#ifdef` hell" [Feigenspan et al. 2013; Le et al. 2011; Medeiros et al. 2018; Siegmund et al. 2012] of C and C++ provided ample motivation for a highly structured system driven exclusively by named tags. Experience with `version` provides good empirical evidence that the feature provides a good balance of expressiveness and maintainability.

Conversations with Eric Engstrom led to a more specific sort of version conditional. He mentioned that different groups had frequently invented their own standard for handling debug builds with the C preprocessor. This presented difficulties in sharing code. The same could happen in D if different groups were to use different version identifiers for their debug builds. If the language provided a standard instead, groups might be more likely to use it rather than inventing their own. This line of thought resulted in the debug conditional being another inaugural feature of the language.

```
debug {
  // Only compiled when the -debug flag is given to the compiler.
}
```

`version` and `debug` were the first members in what would grow to become a comprehensive set of compile-time features for generic and generative programming.

3.4 Broader Influences

D acquired C++'s value semantics with user-defined copy semantics, though with a clear distinction drawn between value types (`struct`) and the Java-inspired reference types (`class`). Other borrowed ideas include efficient vtable-driven virtual dispatch of virtual functions, protection levels, and exceptions.

Walter's experience with Symantec Café had revealed that OOP could be far simpler than the C++ model. This led him to shun multiple inheritance of implementation because it has little benefit

and adds undue complexity in implementation. He opted instead for Java-style single inheritance of implementation and multiple inheritance of interface. The implicit dereferencing of object references and the use of Unicode strings were also directly inspired by Java.

In contrast to the designers of Java, Walter did not see operator overloading as something to be avoided. When used appropriately, it was a beneficial feature. Inappropriate use generally means overloading an operator to carry out an operation that is counter to the commonly understood meaning of the operator's symbol, such as overloading the '+' operator to perform concatenation rather than addition. (D opted for the '~' operator for array concatenation and '~=' for appending).

D would eschew the C++ approach of incorporating the operator's symbol in the function name and use the (abbreviated where appropriate) name of the operator instead: `opAdd`, `opMul`, `opAppend`, etc. The hope was that the operator names would mentally reinforce the intended usage of the operator. It's unknown whether the approach had the desired effect, but it was largely abandoned in D2 for a more flexible template-based implementation using a different naming scheme.

C++ features Walter planned to avoid included namespaces and templates. Namespaces as a distinct language feature were unnecessary in a language featuring modules and packages as part of a symbol's fully-qualified name. He initially objected to templates on the grounds that they added disproportionate complexity to the front end, they could lead to overly complex user code, and, as he had heard many C++ users complain, they had a syntax that was difficult to understand. He would later be convinced to change his mind.

3.4.1 Private Friends. One D feature that some C++ and Java programmers find surprising is that `private` members in `struct` and `class` declarations are "private to the module" rather than "private to the type." In other words, any private symbol declared anywhere in a module is accessible anywhere else in the same module. The module, not the class, is the unit of encapsulation.

```
// Top-level private symbols are accessible only within the module,
// and this is never a surprise.
private const int windowWidth = 1024;
private const int windowHeight = 768;

class Window {
  // Private class members are accessible elsewhere in the
  // the same module, which can be surprising
  private this(int width, int height) {
    ...
  }
}

Window createWindow(int width = windowWidth, int height = windowHeight) {
  // Has access to private constructor of Window
  return new Window(width, height);
}
```

This behavior was implemented as an alternative to the C++ `friend` feature. According to Walter [NA Bright 2018]:

> C++ friend is a hackish thing, with consequences in appearance, name lookup and scope. Being able to declare a "friend" that is somewhere in some other file runs against notions of encapsulation.

In the D forums, there have been a few long debates over the years between D users who find the feature convenient and those who expect `private` to behave as it does in C++ and Java. The thread from which the above quote was taken serves as a good example [NA Alex 2018]. Such criticism has never been widespread and there are no plans to change the behavior of `private`.

D's version of `protected` is allowed only in `class` and `interface` declarations, as only those entities support inheritance. As in other object-oriented languages, it makes class members accessible to subclasses. Unlike Java's `protected`, it does not grant access at package level (D has a fourth access modifier, `package`, for that), but it does make members accessible in the same module. For reasons unknown, this behavior has not generated the same level of controversy as that of `private`.

3.4.2 Automatic Memory Management. Walter's experience implementing a garbage collector for Symantec's Java implementation had convinced him of the benefits of garbage collection and motivated him to make it an integral part of D's initial design [NA Bright 2002b]. Certain language features, such as the `new` operator and array concatenation, were implemented with the assumption that a GC is always present. He was cognizant of the fact that his enthusiasm for garbage collection was not widely shared and that some people would need convincing that a GC could have a place in a systems programming language. He iterated the benefits as he saw them, along with some downsides, in an article on the Digital Mars website [NA Bright 2001d] aimed at persuading potential D users that a GC in a systems programming language was not a deficiency.

Garbage collection is not a mandatory feature in D. If one does not allocate memory via `new`, directly call one of the GC's allocation functions, or make use of a language feature that allocates from the GC memory pool, then no scanning of memory or collecting of garbage will ever take place. Programmers have full access to the C standard library's memory allocation API and other third-party allocators can be used if preferred. Programs may even mix GC-managed memory with manual memory management. The D runtime provides an API to add unmanaged memory blocks to the list of blocks it scans and to "pin" GC-allocated memory so that it is never collected until it is unpinned. The latter feature makes interaction with C libraries less cumbersome.

Garbage collection in D would become a perennial point of criticism, often cited as a reason to avoid the language. Though Walter implemented an API to disable and enable the GC at will and to force collections to occur, the option to remove it completely would become a frequent feature request. Walter was unconvinced that the benefits of implementing such a feature would justify the costs. Certain language features would be unavailable in the absence of a GC, and the standard library would need to be refactored to accommodate programs that use the GC as well as those that completely remove it. Eventually, as the language evolved and made inroads into industries where high performance was critical and the aversion to garbage collection strong, Walter would be persuaded that a "no GC" option could be a boon to D's adoption. The `@nogc` function attribute would be incorporated into D2, along with runtime options to assist in profiling GC usage and performance in a D program.

3.4.3 Unit Testing. Walter understood the value of unit tests, but he had seen firsthand that depending upon third-party tools and frameworks to implement them was a recipe for outdated and missing tests. This inspired him to add support for unit testing as a feature of the language.

The `unittest` keyword introduces a compound statement at module level, or inside a `class` or `struct` definition. Passing a dedicated command-line argument during compilation instructs the compiler to build and run unit tests just before running the application itself. For example:

```
void someFunction() { ... }
```

```
unittest {
  // Unit tests for someFunction() go here
}

class Widget {
  void transmogrify() { ... }

  unittest {
    // Unit tests for Widget.transmogrify() go here
  }
  ...
}
```

Later, the addition of templates to the language would raise the question of how to handle unit tests defined inside generic classes. The answer is that they are instantiated and executed for each instantiation. This is onerous with library-based approaches, but it's part of the natural workflow in D:

```
class Generic(T) {
  void method() { ... }

  unittest {
    // One definition and execution per instantiation of Generic
  }
  ...
}
```

The compiler front end instruments the function body's code (in unit testing mode) such that after the unit tests are run, a code coverage report is automatically generated. In conjunction with simple scripting, arrangements can be made to a build system such that code coverage percentage does not decrease as functionality is added.

Placing the unit tests for a function adjacent to its body makes it natural to develop them in concert with the function in a test-driven manner rather than as an afterthought. More importantly, the simplicity of the approach lowers the barrier of entry to unit testing [NA Davis 2014], as can be seen by examining the source code of many open-source D projects [NA Rejected Software 2012a]. However, some who are used to or require more complex testing find the implementation too simple [NA Kröplin 2017].

3.5 The Engineering Influence

The insights Walter derived from his time as a Boeing engineer influenced a number of D features. The behavior of language features should not defy user expectations, and the language syntax should follow the principle that invalid assemblies should be impossible. In other words, make it easy for the user to do the right thing and difficult to do the wrong one.

This is easier said than done. Programmers come to D from a variety of backgrounds that influence their expectations, and one person designing a language alone is bound to overlook the potential for error-preventing syntax in some places. The effort to reduce the potential for errors would be ongoing. Still, there were some obvious, low-hanging items that could be taken care of from the beginning.

3.5.1 Integer Literal Suffixes. The lower case 'l' cannot be used as an integer literal suffix (a character appended to an integer literal to change its type from `int` to, in this case, `long`), due to potential confusion with the digit '1'. Only the uppercase 'L' is allowed. (Also found in the JSF Coding Standards [NA Lockheed Martin 2005].)

3.5.2 Implicit Variable Declaration. In languages like JavaScript, variables need not be explicitly declared before they may be used. Instead, the initial usage of a variable via the assignment of a value, as in `number = 2`, creates the variable. This can be problematic, as a typo in subsequent usage of the variable will cause another variable to be created with the misspelled name. D does not support implicit variable declarations, requiring all variables to be explicitly declared.

3.5.3 Shadowing. Local declarations that shadow other local declarations are not allowed:

```
void foo(int i) {int i = 3; /* error, shadows parameter 'i' */}
```

Global symbols are supported for C compatibility (they can also be convenient). The introduction of a global should not suddenly render existing code invalid, so the shadowing of globals is allowed for overriding modularity concerns.

3.5.4 Underscores in Numeric Literals. Underscores can be used in numeric literals to make them easier for human eyes to interpret (a feature borrowed from Ada):

```
2135555565          // difficult to read
2_135_555_565       // separated at thousands
213_555_5565        // looks like a phone number
1234_5678_9000_7777 // looks like a credit card number
```

3.5.5 Empty Statements. A solitary ';' as an empty statement is illegal—'{' and '}' must be used instead:

```
if (i > 10); // oops!
  sum += i;
```

3.5.6 Confusing Behavior. Confusing forms allowed in C, such as `a < b < c`, are illegal:

```
((a < b) ? 1 : 0) < c  // C rules (motivated by uniformity)
a < b && b < c         // Python rules (motivated by math notation)
```

The C rules are motivated by consistency with the other parts of the language; all operators are associative, and most other binary operators are left associative. That consistency leads in this case to a mostly useless composition rule. Python addressed the matter by taking inspiration from the usual math semantics. Walter aimed at avoiding silently changing the semantics of code ported or pasted from C. The solution adopted was simple, robust, and obvious in hindsight: comparison operators are not associative in D's grammar. Confusing uses such as `a < b < c` are syntactically illegal and produce a compiler error.

3.6 Early Mistakes

Although he had the design and implementation details covered, Walter missed two key insights about the development process that he would come to regret. There were also two language features that misfired and would later be removed.

3.6.1 Process. In hindsight, the failure to use a version control system from the very beginning was a major mistake. Version control is easy to use and has many advantages, one of which is the maintenance of a complete history of a project's development. A consequence is that the early history of the development of the D compiler was not recorded.

Another mistake was the failure to recognize that the world had changed with respect to the value of Open Source. Walter initially opted to stick with the old closed source model which had been successful for years in the software world at large. It would later become clear that D was going to grow and succeed only if it was Open Source.

3.6.2 Features. An uncommon numeric type incorporated into D, which seemed like a natural idea at the time, was the `bit`. When a `bool` type is a full byte in size, with its two possible values, seven bits are wasted. That raises the question of what happens if any of the other seven bits aren't zero. A `bit` type could represent a Boolean without the waste or the ambiguity.

In time, this proved to be a misguided feature. Because individual bits don't have an address, then a `bit*` must be a special pointer that contains both the address of the byte in which the bit is contained and the bit number. The presence of multiple pointer types means special-casing `bit*` would filter through every type in the type system, a cost that far outweighs the advantages of the type. `bit` would be removed from the language and a byte-sized `bool` added in 2006 [NA Digital Mars 2006].

Another feature that never found its purpose was the ability to embed D code directly into HTML. The idea was that one file could contain both the documentation for the code and the code itself, literate programming style [Knuth 1984]. The documentation could then be displayed simply by opening the file directly in a browser without any need for an intermediate generation step.

It never caught on. Quite likely, part of the reason is that HTML is ugly to look at and is only a marginally human-readable format. A major part is that D programmers just never accepted the paradigm of code embedded in the documentation. Walter never used it himself. The reverse, documentation embedded in code, was an established paradigm that did find success in the D community in the form of Ddoc comments. D-in-HTML remained part of the language through the final release of the D1 compiler [NA Digital Mars 2012a], but was removed from D2 [NA Bright 2008].

4 LEAVING THE NEST (2001–2003)

Walter was initially the sole developer for both the creation and implementation of D. He had no staff to pay and equipment costs were minimal. He made no use of paid advertising. Excluding the licensing deal with Symantec, his out-of-pocket expenses were likely less than $10,000 over the first fifteen years of the language's life.

Releases of the compiler were frequent throughout 2001 and 2002, the last being 0.50 on November 20, 2002. This series of releases solidified the initial language features through bug fixes and behavioral tweaks, enhanced the runtime library, and streamlined the processes of building and using the compiler. From that point, stabilization of the core features was a major emphasis of Walter's efforts, but that didn't prevent the introduction of new features that were never considered in the original design. When he was opposed to adding a specific feature, he could sometimes be persuaded to change his mind. His approach to designing D at this point could be boiled down to "try it and see if it works."

4.1 Communal Roots

As Walter's focus moved away from C++ toward D, Jan Knepper continued to provide support for web hosting and the newsgroups. The digitalmars.D newsgroup was launched on August 12, 2001 [NA Knepper 2001], the date that marks the birth of the D programming language community. On the same day, Walter announced the availability of the D Programming Language Specification [NA Bright 2001c].

The first comment from someone investigating the new language appeared the day after the newsgroup's launch [NA Frohne 2001], and activity would increase over the next few years as users left feedback, asked questions, requested features, and reported issues. When users asked how they could help, Walter encouraged them to spread the word about D by "putting up a personal web page" and mentioning the language "in topical newsgroup posts" [NA Bright 2002d]. He followed his own advice and published an article, "The D Programming Language", at Dr. Dobb's Journal in February, 2002 [NA Bright 2002a]. In it, he presented D as an alternative to C++ and explained the primary motivation behind the language:

> 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.

Eventually, Walter began accepting contributions in the form of patches. He was the initial author of Phobos, the D runtime library, which morphed into much more of a community effort. Throughout this period of increasing contributions, he remained the sole designer and implementer of the core language.

As the list of contributors grew [NA dlang.org 2018], so too did the number of challenges that Walter had not anticipated. The newsgroup was the primary means of communication, but he found a steadily increasing number of emails in his inbox. Keeping up with communication increasingly took time away from working on D. Since everyone was a volunteer, he had no authority to order anyone to do anything. People worked on what they wanted, when they wanted, and appeared and disappeared as they chose. Often, volunteers would ask him what they should work on. He would provide a list and the volunteers would then choose to work on something else. This often meant that he was left doing work that no one else wanted to do.

Without the backing of a major organization, there were no marketing, community relations, or training personnel. There were no official IDE plugins, GUI libraries, or build systems. The project had no schedule or deadline. Task selection was governed by an informal assessment of need divided by an estimate of the time required to implement. The difficulties of developing the first version from front to back had been mitigated by licensing the C++ compiler from Symantec. This provided a foundation consisting of a professional code optimizer, code generator, linker, and related tools, all of which meant that DMD was initially limited to the 32-bit Windows platform. The immediate necessities implementation-wise had been the front end for D and the initial runtime library. The D compiler was developed with C++, but that did not have any impact on the initial design of the language.

4.2 Templates

As more programmers downloaded DMD and put the language through its paces, some of them began posting in the newsgroup requesting that Walter add new features to the language. Though he was not averse to adding new features, he wasn't interested in adding any features he had

already considered and rejected when he had drafted the language specification. His position on such requests is seen in a response he gave [NA Bright 2001e] to someone arguing in favor of multiple inheritance who posited that the existence of a feature did not mean everyone had to use it.

> The counterargument (and I've discussed this at length with my colleagues) is that C++ gives you a dozen ways and styles to do X. Programmers tend to develop specific styles and do things in certain ways. This leads to one programmer's use of C++ to be radically different than another's, almost to the point where they are different languages. C++ is a huge language, and C++ programmers tend to learn particular "islands" in the language and not be too familiar with the rest of it.
> Hence one idea behind D is to *reduce* the number of ways X can be accomplished, and reduce the balkanization of programmer expertise. Then, one programmer's coding style will look more like another's, with the intended result that legacy D code will be more maintainable.

The conversation had veered off the original topic of that newsgroup thread, which was titled "Templates." It was one of several threads in which templates were requested and debated throughout 2001 and 2002. The discussions eventually persuaded Walter that he needed to reconsider his opposition to adding templates to D. He still didn't intend to implement C++ style templates and wanted to find an alternative. It wasn't long before he hit on a solution.

His key insight was that the compile-time parameters of a template declaration could be seen similarly to the run-time parameters of a function declaration. From that perspective, there was no need for a special syntax for a template's parameter list—the same parentheses used for a function's parameter list could pull double duty. As he saw it, it made templates much easier to understand. He has likened it to the experience of a colleague who once moonlighted as a remedial algebra teacher: if she asked her students to "solve for x", they would freeze up and fail to answer, but when she removed the 'x' and instead asked them to "fill in the blank", she would have the answer before she could blink.

A D implementation of the `min` function template looks like:

```
template min(T) {
  T min(T a, T b) {
    return b < a ? b : a;
  }
}
```

In modern D parlance, this an eponymous template, a template which contains one member that shares the template's name. Normally, template members are accessed using the same dot syntax used for aggregate types, e.g., `min.min(10,20)`. Eponymous templates allow the eponymous member to be accessed without the prefix, so that in this case the function can be called directly as `min(10, 20)`. In D2, eponymous template declarations would also be allowed a special shorthand syntax that eliminates the need for the `template` keyword:

```
T min(T)(T a, T b) {
  return b < a ? b : a;
}
```

Multiple type and function declarations can appear inside a template declaration. The shorthand syntax for eponymous templates is not restricted to functions, but may be applied to any eponymous declaration inside the template:

```
template Point(T)
  struct Point {
    T x, y;
  }
}

// No need to use Point.Point!float
alias Pointf = Point!float;
```

As seen in the declaration of `Pointf`, Walter also implemented a clean syntax for template instantiation. He selected the character '!' as the template instantiation operator. It is paired with the template argument list to simplify the back-end implementation, as in `min!(int)(a, b)`, and is required when type inference is not possible. When the template is instantiated with only one parameter, as in this case, the parentheses may be omitted: `min!int(a, b)`. When type inference is possible, again as in this case, the parameter list may be omitted: `min(a, b)`.

Templates made their debut on September 8, 2002, in DMD 0.40 [NA Digital Mars 2002c]. Over the years, they have been improved and enhanced with features such as default parameters, alias parameters, tuple parameters, and in D2, template constraints.

4.3 An Ergonomic Loop

The proposal for a `foreach` loop first appeared in the newsgroup in May, 2002 [NA Yates 2002]. Walter's response (and one of his earliest mentions of a future D2):

> It's a good thought and many people have suggested it. It won't be in version 1, though, but maybe version 2.

By October, he was "thinking about adding it in" [NA Bright 2002c]. With the release of D 0.71 on September 3, 2003 [NA Digital Mars 2003b], the `foreach` statement officially became part of the language. It was his engineering background and the lesson of safety ergonomics, that incorrect configurations (or syntax in this case) should be impossible, that led him to change his mind.

D's support for the traditional `for` loop is only slightly less error prone than C's, as demonstrated here:

```
bool find(int needle, int[] haystack) {
  for(int i=0; i<=haystack.length; i++) {
    if(haystack[i] == needle)
      return true;
  }
  return false;
}
```

The first mistake is that `i` is of type `int` when it should be `size_t` for a proper comparison with `haystack.length`. This is a common mistake with `for` loops that D's fat pointer arrays alone cannot prevent. The second mistake, perhaps less common but a subtle source of bugs, is the use of the wrong comparison operator: '<=' instead of '<'. D's default bounds checking on array accesses can catch this at runtime, but when bounds checking is disabled the behavior is just as undefined as it is in C.

The introduction of `foreach` brought with it an opportunity for a new syntax which could eliminate the possibility for such errors. The final implementation was something that was distinct from the standard `for` loop, yet simple and familiar.

```
bool find(int needle, int[] haystack) {
  foreach(index, e; haystack) {
    if(e == needle)
      return true;
  }
  return false;
}
```

Here, `index` is an optional value (it may be omitted) assigned the array index of the element in the current iteration. By omitting the type, it is `size_t` by default. `e` is the element at the current array index, its type inferred from the array declaration. This employs the compiler's knowledge about arrays, that they are fat pointers which know their length, to eliminate two common `for` loop errors. The language would later gain a complementary `foreach_reverse` statement.

In D2, `foreach` would be expanded to iterate over ranges, sparing the programmer the need to do so manually via the D2 range API.

4.4 The `is` Operator

Over the years, and before a formal process for introducing language features was established [NA Parker 2016], Walter occasionally surprised D users by implementing requests for minor language changes without any indication of his plans beforehand. The `is` operator serves as a demonstrative example.

Operators for identity ('===' / '!==') and equivalence ('==' / '!=') were part of D's early feature set. When the latter was used on `struct` or `class` instances, it was rewritten into a call to an `opEquals` method. This would always work for a `struct`, but would cause an access violation (thereby terminating the program) when used with a `null` class reference.

The identity operator was intended to be used when comparing class references. Unfortunately, the two operators sometimes caused confusion, a point raised in the newsgroup by Matthew Wilson in March of 2003 [NA Wilson 2003b]:

> If I remember, last year there was debate on using == and === operators for representing equivalence and identity. Is this correct? Still the case? If so, which one is which?

In a subsequent discussion [NA Wilson 2003c], Walter did not appear inclined to accept that there was a problem. The behavior of both '==' and '===' was well documented and, according to his view of software development, an access violation was not a bad thing.

In October, Matthew was at it again [NA Wilson 2003a].

> Please, please, please, please, please can we get rid of === and !==, and replace them with something that will not (or at least, less easily) facilitate undetectable errors entering code? My suggestion is "is" and "is not", but I'm not particularly stuck on that. Anything that will avoid these errors.

Walter did not respond in that thread but he was thinking about it. In the end, he decided the feature was easier to add than to fight. The release of DMD 0.76 on November, 21[NA Digital Mars 2003c], included a new `is` operator with behavior identical to '==='. When a commenter expressed excitement for the new operator, Walter replied, "See, I do read this ng <g>."[NA Bright 2003]

The suggested '`is not`' was never implemented. Instead, only `!(a is b)` was supported. The `is` operator subsequently became the recommended means of testing identity. '===' and '!===' would be deprecated in Jun of 2005 [NA Digital Mars 2005b] when support for `!is` was added.

5 GROWING PAINS (2004–2006)

By 2004, D was still a long way from becoming more than an obscure language, but there were signs that it was starting to gain steam. One such was the growth of activity in the D newsgroup, where a total of 734 topics posted in 2002 had more than doubled to 1,532 in 2003 [NA Digital Mars 2002a, 2003a].

In April of 2004, in response to the increased activity and as a future-proofing measure, Walter launched two new newsgroups under a common namespace. digitalmars.D was intended as a replacement for the existing D group for general discussion of the language and digitalmars.D.bugs [NA Bright 2004e] for bug reports. In March of 2005, he created digitalmars.D.announce and digitalmars.D.learn [NA Bright 2005a] to accommodate a further increase in activity and topic variety.

Also in April, Walter announced in the new newsgroup that he had ported his old game, Empire, to D [NA Bright 2004c].

> Every language needs a game written in it, and now Empire is in D (at least version 0.86). You can pick it up at www.classicempire.com. Warning: Empire has a long track record of being an enormous and unproductive time waster. It's been rumored to me to have caused many students to flunk out of college, job loss, and was even reputed to have instigated a divorce. Start playing it at your own risk.

The ported code was still close to the original C, demonstrating that C code could be ported to D with minimal effort and then D-specific features could be added in as desired.

Walter was also busy implementing a long-requested language feature.

5.1 Template Mixins

In August of 2001, D user Richard Krehbiel published a newsgroup post titled "Macros" [NA Krehbiel 2001]. It began with the question, "Okay, why is it that everybody thinks the C preprocessor is terrible and needs to be avoided?"

In response, Tim Sweeney said, "Because it's terrible and needs to be avoided, of course!" [NA Sweeney 2001] He went on to list five ways in which preprocessor macros had been used in the C++ codebase of the first Unreal game.

> 1. To expose metaclass information (i.e. class names, default constructors that a serializer can call) – like MFC's techniques. All of this code would be unnecessary if the language supported classes as first-class objects (i.e. you can pass around a classref* which "represent" the class and exposes its static functions), static virtual functions, and static constructors.
> 2. To comment out large blocks of code. Would be unnecessary if /*...*/ comments could be nested.
> 3. To implement debug-specific code. This is actually unnecessary, a bad old habbit [sic]. We would be just as well off having a global constant debug=0 or 1, and having if(debug) ... instead of #if debug.
> 4. To implement platform-specific headers. Only necessary because headers are necessary.
> 5. To perform template-style tricks. If the language has a great facility for type dependency (whether like C++ templates, or more general like Haskell), all of these things would be unnecessary. Even C++ templates aren't complete enough, i.e. there are no template typedefs (true type synonyms), and most production compilers have bizarre template bugs limiting what you can do.

He ended by saying that if C++ had built-in support for all of the above, then "the Unreal code would be simpler and cleaner."

By 2004, D had features that matched four of the listed use cases: nested comments, debug version blocks, modules, and templates (though they weren't yet as powerful as the templates Tim was envisioning). There was no D feature that supported the first use case.

In October, 2002, Patrick Down started a newsgroup thread titled "D Mixins". The idea he proposed was a feature called "mixins" as an extension to D's templates [NA Down 2002]:

> You will find various explanations of mixins on the web but to me they are just aggregation with a twist. The twist is that aggregated object has some access to the object in which it is aggregated into.
> I think that with D's templates could be extended to cover this feature.

He then went on to propose a syntax for such a feature. The proposal received very little feedback and none from Walter.

Over a year later, in December, 2003, Mikkel Jørgensen emailed Walter his own proposal on mixins. This proposal differed from Patrick's in that it revolved around the idea that a `mixin` definition would be analogous to a `class` or `interface` rather than a template. Walter's response:

> I think mixins are a good idea. Why not post this to the D newsgroup? There are some pretty smart people there to comment on it.

Mikkel followed his advice [NA Jørgensen 2003]. In the discussion thread, Patrick gave the proposal his support, as did other users who left feedback, including Matthew Wilson. He and Walter began discussing the idea via email, where Matthew presented his own proposal similar to Mikkel's.

Walter came to see mixins in the same light as Patrick Downs had, as a natural extension of templates, but there was nothing in is experience upon which he could base an implementation other than the discussions he had followed in the forums. Following his "try it and see" philosophy, he decided to take a shot in the dark. On May 16, 2004, he made the following announcement in a new post titled "mixins" [NA Bright 2004d]:

> I have this mostly implemented. It's been based on a lot of your suggestions. The way I did it is, I think, pretty unexplored territory. That means there may be some pretty cool uses for it I've never thought of. Let me know what you think, and if I missed the boat completely <g>.

Semantically, template mixins [NA Digital Mars 2012b] were intended to allow some of the functionality expressed in Tim Sweeney's first use case ("like MFC's techniques") while avoiding the weaknesses of the C preprocessor. When the `mixin` keyword is prefixed to a template instantiation, the body of the template is "inserted" into that location. During a normal template instantiation, the template body takes on the scope in which it is implemented. When the `mixin` keyword is applied to a template instantiation, this behavior is turned upside down and the template body takes on the scope in which it is instantiated. In other words, if a template needs access to any private symbols in the module in which it is declared, attempting to mix it in to another module will cause compiler errors.

As a basic example, consider a C library that makes use of simulated `struct` inheritance. When creating a D binding to the library, template mixins can be employed to good effect in the `struct` declarations.

```
// This template is mixed in at the top of struct declarations so that they
// all begin with the same three members.
template EventCommon() {
  EventType type;
  uint timestamp;
```

```
  WindowHandle window;
}

struct KeyboardEvent {
  mixin EventCommon;
  uint keyCode;
  ...
}

struct MouseMoveEvent {
  mixin EventCommon;
  int deltaX, deltaY;
  ...
}
```

The body of a mixin template isn't just blindly pasted into the scope into which it's mixed. It comes wrapped in its own scope with its internal symbols aliased to the external scope. Without this precaution, a mixin could not be used multiple times in the same module or in conjunction with other mixins containing identical symbol names, or without potentially conflicting with existing symbols in the same scope. With this precaution, both are possible when the mixin instantiation is accompanied by an identifier to disambiguate conflicting symbols.

```
template addVars(T) {
  T x;
  T y;
  T z;
}

mixin addVars!float v1;
mixin addVars!double v2;

void main() {
  v1.x = 10; v2.x = 20;
}
```

Typically, templates are written with the intention that they either be instantiated normally or that they be mixed in, but rarely both. Attempting to use a template in a way it is not intended will often cause compiler errors (frequently due to scoping issues) that are hard to decipher. To remedy this, D2 allows the `mixin` keyword to be prefixed to the declaration of any template intended to be mixed in. The compiler will then generate an informative error message if a normal instantiation is attempted. Even though normal templates may still be erroneously mixed in, the rarity of related bug reports and help requests from confused users suggests it is not a problem in practice.

5.2 The Sudden Implementation of `static if`

On May 4, 2005, Bill Baxter made a post to the newsgroup on the topic of improving support for metaprogramming in D [NA Baxter 2005]. He opened by discussing the limitations of C++ templates for metaprogramming. As an example, he linked to an article demonstrating the use

of templates to implement portable parameterized integers in C++ [Pescio 1997]. The technique required an implementation that, in Bill's words, "is just an if-else, but dressed up in C++ template metaprogramming it takes about a page of code." In pondering first-class language support for full metaprogramming capabilities, he posited the following imaginary example:

```
metafun type integer(int numbits) {
  if (numbits<=sizeof(char)) return char;
  if (numbits<=sizeof(short)) return short;
  if (numbits<=sizeof(int)) return int;
  if (numbits<=sizeof(long)) return long;
  if (numbits<=sizeof(cent)) return cent;

  metathrow "Compiler error";
}
```

Four days later, Walter replied with the following [NA Bright 2005c]:

> I agree with you on all points - especially on the one that if metaprogramming was easier to do, it would be a lot more practical. I'd like to bring this to D.

And five days after that, he said in a new post, "You've inspired me" [NA Bright 2005d]. The remark was accompanied by the following snippet of code:

```
template Integer(int nbits) {
  static if (nbits <= 8)
    alias byte Integer;
  else static if (nbits <= 16)
    alias short Integer;
  else static if (nbits <= 32)
    alias int Integer;
  else static if (nbits <= 64)
    alias long Integer;
  else
    static assert(0);
}

int main() {
  Integer!(8) i;
  Integer!(16) j;
  Integer!(29) k;
  Integer!(64) l;
  printf("%d %d %d %d\n", i.sizeof, j.sizeof, k.sizeof, l.sizeof);
  return 0;
}
```

And just like that, `static if` became a feature of D. It was released in DMD 0.124 only six days after its demonstration in the newsgroup [NA Digital Mars 2005a].

Similar to `version`, the `else` branch is optional and the braces '{' and '}' do not introduce a new scope. Also in keeping with `version`, `static if` may occur at declaration level (including top

module level) so its use is not limited to function bodies. Unlike `version`, the evaluated expression can be an arbitrary Boolean expression computable during compilation.

The charter of `static if` is code generation driven by introspection. Typically, generic code queries the parameterized types received and makes decisions depending on their capabilities. Consider, for a simple example, a buffer abstraction that is backed by either statically- or dynamically-allocated memory, depending on the constant size chosen (if zero, dynamic allocation is to be used). The allocation choice leads to radically different data layouts; therefore, a typical implementation would implement the two choices in separation (in a language like C the data structures and APIs would be entirely different; in C++, template specialization might be used). It is worth noting that the vast majority of the code is identical across the two layouts. The typical implementation in D unifies the two definitions, as shown below.

```
struct Buffer(T, size_t max = 0) {
  // Layout {
  static if (max != 0) private T[max] data;
  else private T[] data;
  private size_t used;
  // }
  // Interface
  static if (max == 0) {
    // This API is for dynamic allocation only
    void reserve(size_t capacity) { ... }
    ...
  }
  // This API is common to both
  size_t capacity() {
    static if (max != 0) { ... }
    else { ... }
  }
  size_t used() { ... }
  ...
}
```

The definition of `Buffer` above demonstrates how the programmer is able to manually, and precisely, arrange code related to data layout, interface, and implementation, and cater to various distinctions derived from the allocation decision, by means of simple `static if` decisions that are instantly self-explanatory to the casual reader. The resulting code is unusually compact in relation to its generality. This is because merging multiple design decisions together eliminates several subtle forms of duplication that cannot be addressed via conventional coding techniques. Conversely, the density of `static if` declarations indicates the number of possible distinct binary programs that can be generated from the same codebase, much as the density of regular `if` statements indicates the number of possible execution paths.

The expressions being tested in the example above have been kept very simple to facilitate exposition. Generally, tests may involve much more detailed introspection queries for elaborate types, as in the design of the memory allocation framework in D2 [NA dlang.org 2015a]. The key insight Walter had when implementing this feature, that meta programming should not be a

separate language but should just be regular D code, would later provide the impetus to expand the power and scope of D's compile-time programming features.

5.3 Documentation Generation

D's documentation (language, standard library reference, and website) was originally written manually, without special tooling. The documentation of the runtime library was implemented in files separate from the code, sometimes by different people. In short order, the resulting state of affairs fulfilled the adage that separate documentation is always incomplete, wrong, or missing entirely. Thus arose the motivation for Ddoc, a documentation generation framework akin to Javadoc [Kramer 1999], which drastically improved the matter by integrating documentation into the D source code itself. Ddoc was added in September 2005 [NA Bright 2005b] and is currently used for building the entire language site dlang.org, which includes the language reference and the standard library reference.

Third-party documentation tools, such as Doxygen for C++ [NA Van Heesch 1997], were just becoming popular at the time. Yet building documentation support in the compiler's front end had certain advantages. First, a built-in documentation generator would side-step all matters of platform availability, tool installation, version matching, or subtle parsing differences; and second, the built-in documentation generator has access to rich and consistent semantic information from the compiler. A simple default choice of documentation generator inculcated a culture of expecting Ddoc documentation to be present early on with any coding artifact, rather than as a distinct endeavor and responsibility. Even though Ddoc lacked the sophistication of dedicated documentation tools, it has been a strong trendsetter; subsequent documentation generators [NA DWiki 2018] have been built on top of it in a compatible manner, the most popular being Rejected Software's DDOX [NA Rejected Software 2012c], and Adam D. Ruppe's adrdox [NA Ruppe GitHub repository]. It has also been put to use beyond its original purpose of documentation. As one example, Ali Çehreli used Ddoc to write his book, "Programming in D."

Documentation unit tests would be introduced in D2 in 2013, allowing the insertion of the source code from `unittest` blocks in the generated documentation. Currently, virtually all examples in the standard library documentation at dlang.org/library are executable unit tests.

5.4 DSource

In March, 2004, in response to an increasing number of open-source D projects being announced in the newsgroups, Brad Anderson made it known that he and J.C. Calvareese had been working on a new website geared toward such projects [NA Anderson 2004]. DSource provided Subversion source control hosting, project management software, and a dedicated discussion forum for each topic, all free of charge. Server space was donated by Brad Anderson's employer. The site became the center of open-source D ecosystem activity for several years. Though Walter never moved DMD to DSource, he did eventually move the project to GitHub. The use of DSource subsequently began to decline. In 2014, Vladimir Panteleev took over management of the domain. He made the site available in read-only mode [NA Panteleev 2014], preserving a period of D's history involving over 200 projects [NA Dsource.org 2004].

A major project that emerged from DSource was DWT [NA DSource.org 2004a], an ambitious effort to port the Eclipse Foundation's Standard Widget Toolkit (SWT) [NA The Eclipse Foundation 2003] from Java to D. At the time, there were no practical options for GUI development in D. Other initiatives followed shortly thereafter, such as Mike Wey's D bindings for Gtk [NA The GTK Team 1997], called GtkD [NA Wey 2004], and Christopher Miller's D Forms Library (DFL) [NA Miller 2004], a Windows-specific GUI library that included a WYSIWYG editor known as Entice Designer. Neither of these projects caught Walter's attention in the way that DWT did. He was so enthusiastic

about DWT that he added Java-style inner classes as a new language feature to facilitate porting the Java code base to D. In February, 2006, he declared DWT the official D GUI project and opened the digitalmars.D.dwt newsgroup [NA Bright 2006b].

5.5 A Serendipitous Encounter

Walter and Andrei Alexandrescu first crossed paths at the SD West [NA SD West] conference. The following month, after a discussion with Eric Niebler comparing D templates with those of C++, Andrei emailed Walter his thoughts on the subject (all of which would ultimately influence the implementation of D's templates). The opening line gives an indication of the nature of his thoughts:

> I'm going to be ruthless. Put your bulletproof vest on.

At a subsequent Northwest C++ User Group event, though neither man can recall which one, they entered into a discussion on language design and Andrei's intent to apply his ideas in a language he called Weasel. Likely candidates are the October 18, 2004 gathering where Walter spoke about D [NA Bright 2004a] and the January 12, 2005 event at which Andrei gave a presentation on lock-free programming [NA Alexandrescu 2005]. Whatever the date, their conversation that evening eventually led them to arrange to meet for more in-depth discussion, making it one of the most influential events in the history of the D language.

On March 20, 2005, at a restaurant in the University of Washington district, Andrei showed Walter his plans for Weasel. One of the features in which he was interested was User-Defined Syntax (UDS). Walter disliked the idea on the grounds that it was too similar to macros, a concept to which he was antipathetic, and he tried to talk Andrei into abandoning it. There were other features, such as *scope guards* [NA Alexandrescu and Marginean 2000], that he felt would be worthwhile incorporating into D. The back and forth between them left the door open for more meetings and even further discussion. Both mark this meeting as the beginning of their collaboration.

Andrei insists he did little in the way of contribution throughout 2005. In 2006, their meetings became both larger and more consequential. They were joined by Brad Roberts, Bartosz Milewski, and Eric Niebler in discussions that focused on the features that D lacked which a modern programming language ought to have: an intuitive concurrency model, memory safety, and more. According to Andrei, this group of people "for better or worse influenced the definition of D", as the ideas that materialized in their meetings would become the foundation of D2.

6 STAKES IN THE GROUND (2007)

By 2007, some of the core cadre of early adopters were still around, but they had been joined by a more varied and eclectic mix of programmers. Some of them weren't happy that the D "forums" weren't the sort of feature-rich forums [NA Bobef 2005] they had grown accustomed to. In comparison to the modern forums at DSource, the official D NNTP-based forums were ancient technology.

At some point, Walter had installed an open-source web interface to make newsgroup access more palatable, but it was an old Perl CGI program with an equally ancient interface that never caught on. He had gotten complaints about it over the years, so in December of 2006, he upgraded to a new web interface [NA Bright 2006c]. It was only a marginal improvement. In 2007, Brad Roberts created a mailing list interface for each active D newsgroup [NA Digital Mars 2007b]. Several users gravitated to the mailing lists, but that didn't satisfy everyone.

A number of IRC channels for D had been created by community members in the early years, but the one that stuck was the `#D` channel at freenode.net. Established in January 2003, its usage had slowly grown along with that of the D newsgroup. By February 2006, the channel had an

average concurrent user base of 20 with a peak of 30 [NA Miller 2006]. Not only was #D a place where beginners could find help, it was a place where users could vent their frustrations outside of the official newsgroups. In effect, the IRC channel became a sort of in-the-open underground network. With most of the regular newsgroup users rarely, if ever, visiting, some ideas that would have faced opposition in the newsgroup found fertile ground in the IRC channel.

6.1 Tango

In 2006, among the topics of discussion in IRC were complaints about the state of Phobos and the unannounced work of a team of three developers. Two of the three were the maintainers of two open source projects at DSource: Mango, a collection of packages aimed at network programming and servlet development [NA DSource.org 2004b], and Ares, which was intended as a full replacement for Phobos, the D standard runtime library [NA DSource.org 2004]. Together with another like-minded D user, they began work on a library that combined aspects of Mango with Ares in a broader set of packages that they would announce to the world as Tango.

The initial announcement of Tango on December 31, 2006 [NA Kelly 2006] was a stake planted in the ground by the Tango team and their supporters a point around which those who were unhappy with the state of Phobos could rally. And rally they did. With the first release on January 31, 2007, Tango began its steady accumulation of users and contributors [NA Igesund 2007].

In late August, as the D Language Conference was winding down, the D community was waiting for news, as demonstrated by newsgroup user BLS [NA BLS 2007]:

> Seems that nobody has the heart to ask . So I will : Any clarification regarding having 2 standard libraries ?

By all accounts, the meeting between Walter and the Tango team was cordial. Given Walter's desire to maintain Phobos in its current form, the place to look for a compromise was in bridging the differences between the two libraries. According to Walter [NA Bright 2007c]:

> Myself and the Tango team both agree that the current situation is not good, and to fix it we need to remove the incompatibilities between the two, and we intend to do so.

Sean Kelly, a Tango developer and the former maintainer of Ares, concurred [NA Kelly 2007]:

> I think we all agreed that the current situation isn't ideal and that we'd like to rectify it. However, there are some technical and workflow issues to address, at the very least.

The reaction to the news was mostly positive, but those hoping to hear of a merger were disappointed.

Walter felt that a wrapper API like Tangobos [NA Richards 2007] was a suboptimal solution. The approach they settled on was to separate the language runtime from the standard library. By sharing a common runtime, both libraries could exist side-by-side in the same program. Sean had begun his Ares project from the same code at the core of Phobos. He had enhanced it with new features, such as multithreading support, but had changed the original code very little. He took on the job of working out the issues and establishing a common code base. Coming as it did during the early development work on D2, the new DRuntime would become the common language runtime for that version of the language.

Tango had become so firmly established by the time of the conference that the Tango team were invited by Apress to author a book about D and Tango. The book was part of the publisher's "FirstPress" series, which they printed in order to gauge interest in various programming topics and the potential for future publications. Michael Parker had been writing about D at "The One With D" since early 2006 [NA Parker 2006]. The Tango team invited him to join them on the book project. Together, the four spent the autumn of 2007 writing and revising. "Learn to Tango with

D," was published in February of 2008 [Bell et al. 2008], though it was too late to claim the honor of being the first published book about D. It was preceded the previous November by a German book titled "Programmieren in D" by Tobias Wasserman and Christian Speer[Wasserman and Speer 2007]. The Tango book was later used in a D course by the Faculty of Mathematics and Computer Science at Nicolaus Copernicus University in Torun, Poland, who also hosted the first and only Tango Conference in late September, 2008 [NA Tango Wiki 2008].

6.2 The Penultimate Version

At the end of 2003, some users were already eager to see a 1.0 release of DMD [NA T. 2003]. Walter, ever the optimist, said that he "think[s] it's pretty close" [NA Bright 2004b]. IRC users idly wondered if a 1.0 release would ever come and posts in the newsgroups reared up from time to time. Two days after the announcement of Tango, on January 2, 2007, their wish finally came true.

DMD 1.00 was arguably an underwhelming release. The change log lists 38 bug fixes, an enhancement to the `-v` compiler flag, and a new module that provides D bindings to the `libpthread` API [NA Digital Mars 2007c]. Aside from the whole number in the version, there was nothing to distinguish it from the 179 compiler releases that had come before. It appeared as if Walter was shoving his own stake in the ground at a random point and declaring that henceforth, DMD would be known as 1.0.

There was method to his madness. The language design discussions he had been having with Andrei and the others had been fruitful, but implementing the ideas they had been exploring would bring about breaking changes. The declaration of the 1.0 release allowed Walter to designate a "stable" version of the language, freeing him up to implement potentially disruptive features in a new 2.0 version.

Few in the community knew anything of those plans. They were happy just to see that "1" in the version number. The announcement of the release in the newsgroups [NA Bright 2007a] was met with 63 cheerfully congratulatory posts. It wasn't simply a major milestone, it symbolized the achievement of years of effort by Walter and the community members who had contributed in big ways and small along the way. It also represented stability and, to some degree, it was a confirmation that D had finally arrived.

7 TO INFINITY AND BEYOND

Any tale of the *origins* of the D programming language must reasonably end with the release of version 1.00. The development of D2 is another tale, but it would be a disservice to the reader not to provide a glimpse of what came next.

7.1 From D1 to D2

Subsequent releases of the DMD 1.xxx series would see comparatively few new features, so in the end stability did prove to be a reality. Two early releases in the series would introduce new features that would transform how D programmers write code. DMD 1.005 brought *import expressions* to the language, allowing the contents of any file in the source tree, e.g., `import("font.bmp")`. The same release included *mixin expressions*. Also known as string mixins, this feature allows mixing code directly into any scope without the precaution of preventing against symbol clashes. This would prove to be a powerful feature that, when coupled with Compile-Time Function Evaluation introduced in the subsequent release, would open the door to generative programming in D.

D1 was considered stable enough that Sociomantic Labs (now Dunhumby), based in Berlin, adopted D and DMD for their real-time ad bidding platform. Funkwerk, a German company based in Munich, began experimenting with D in 2008 and ultimately used it to replace the Java-based components of their passenger information system [NA Parker 2017]. Over time, other companies

would build their businesses around D or use it to implement tooling [NA D Language Foundation 2020b].

Andrei Alexandrescu's involvement gradually increased after the 2007 conference. The ideas that he and Walter had discussed in their meetings with Brad Roberts, Bartosz Milewski, and Eric Niebler would start to take shape in the DMD 2.0x series of compilers, the first of which was released on June 17, 2007 [NA Digital Mars 2007a]. This release was the first that would bring incremental changes over the existing language.

Ultimately, D2 would effectively become a different language; it wasn't long before D2 code became incompatible with D1 code. Major new features that contributed to the divergence include:

Ranges. Ranges were a transformative feature that brought the functional programming paradigm to D [NA Alexandrescu 2009]. A comprehensive set of algorithms in the standard library enabled the construction of lazy, functional pipelines [NA Bright 2012; Teoh 2013] that opened D to a new audience.

CTFE. Compile-Time Function Evaluation expanded the primitive compile-time capabilities of D1 by introducing a compile-time interpreter that supports a large subset of the D language. This allows any D function which meets a small set of criteria to be evaluated at compile time to compute compile-time values, such as constant initializers. When coupled with another D2 feature, string mixins, CTFE brings the power of code generation to the D programmer.

Compile-Time Introspection. The introduction of compile-time function evaluation provided motivation for introspection features. The two features work in tandem: better introspection of code begets more interesting applications of compile-time evaluation, which in turn computes more interesting introspection artifacts. The feature was backported to D1, but was less useful there without D2's other powerful compile-time features.

Transitive Immutability. D2 added the `immutable` qualifier, which expresses transitive immutability of data, and changed the existing contract of `const`, which denotes an unmodifiable view of data that may be mutable or not [NA D Language Foundation dlang.org]. The transitivity of `const` is a feature to which programmers accustomed to C++ often have difficulty adjusting. Transitive immutability paved the way for the support of functional purity and stronger support for multi-threaded programming in the runtime library.

Attributes. Attributes (also known as annotations) are language- or user-defined tags attached to declarations. D2 provides a handful of built-in function attributes that prohibit the use of GC-dependent or memory unsafe features inside a function body or enforce functional purity. User-Defined Attributes can be used with D's compile-time features for code generation and conditional compilation.

Thread-Local Storage. Variables in D are thread-local by default. The `shared` attribute (and its brute-force cousin `__gshared`) can be applied to change this behavior and make a variable available to all threads. While TLS variables have proven to be a boon, `shared` has never been implemented to its full potential. Work began in June, 2019, toward shoring up the specification of the feature.

7.2 The Community

Though Walter had created the 2.x series of DMD out of concern for the potential disruption of breaking changes, the actual source of contention turned out to be the very nature of the new features. Some of them required D programmers to undergo a paradigm shift in order to use them. For example, certain Phobos modules required familiarity with the functional programming

paradigm in order to use ranges effectively, and the transitive nature of the new `immutable` and `const` required much more forethought than D1's C++-style `const` to employ effectively.

Given the reluctance of the Tango developers to port their library to D2, it became the de facto standard library for those who continued to use D1. Library authors now had to choose not only which standard library to support, but also which language version. For several years, even after active development on Tango came to a halt, comments about D's "two standard libraries" inevitably appeared in any social media discussion about D, most often as a reason to avoid the language. As usage of D2 began to grow and that of D1 to dwindle, Tango usage also decreased.

There was no single person or team with whom Walter could hold a meeting to heal the rift over the new language features. It required time and the help of members of the community who made themselves available to answer questions on social media [NA Davis 2010]. By 2012, the original developers were no longer active [NA Igesund 2012]. Sociomantic maintained and evolved their own fork of Tango, which they ultimately released as an open source library called Ocean [NA Ocean 2017]. A handful of D community members forked Tango and completed the port to D2 and the common runtime [NA Tango-D2 2012].

Andrei published the book "The D Programming Language" (TDPL) in 2010 [Alexandrescu 2010]. At that point, D2 was not yet feature complete. The book was intended to serve both as a guide to the completed features and a roadmap to those that were yet to be implemented. Four more D books would hit the market in subsequent years, but TDPL would be considered the D bible for years after its publication.

In 2011, Vladimir Panteleev created DFeed [NA Panteleev 2012], a web forum interface to the existing NNTP server, in the D language (DFeed would receive accolades for its speed [NA Hacker News 2015]). This made the newsgroups more accessible to a wider audience, presenting a more user-friendly and somewhat familiar, if minimal, forum interface.

In 2013, another D programming language conference was held, this time at the headquarters of Facebook in Menlo Park, CA, (where Andrei was working as a researcher) and branded as DConf. Money for the event was raised via a Kickstarter campaign [NA Bright 2013b]. The conference drew approximately 50 attendees from around the world and was considered a success [NA Bright 2013a]. A repeat event was held in the same location the following year and, with the third edition hosted by Utah Valley University, became an annual event in 2015. In 2016, DConf moved to Europe, where it has remained every year up to the time of this writing [NA DConf 2013].

The D Language Foundation was founded in October of 2015 with the intention of raising money for the development and maintenance of the D language, organizing the annual DConf, sponsoring scholarships at universities around the world, responding to the needs of organizations using D, and generally promoting the language. Walter, Andrei, and Ali Çehreli were the Foundation's founding officers [NA dlang.org 2015b].

As the decade beginning in 2011 continued, the focus in D2's development shifted from becoming feature-complete to becoming stable and improving quality of implementation. D1 was deprecated and development discontinued in 2012, though Walter continued to provide support for companies still using it until they could make the transition to D2. Other companies newly adopting D in part or in whole put the second version of the language through its paces [NA D Language Foundation 2020b]. With the D Language Foundation in place, public relations became more of a focus. This included the launch of an official D blog and several social media accounts [NA dlang.org 2017].

Over the years, a collection of semi-formal D Improvment Proposals (DIP) had accumulated at the D Wiki. In June of 2016, Mihails Strasuns established a formal process and took on the role of DIP Manager [NA Parker 2016]. Michael Parker took over in April of 2017. Once the DIP process was in place, the days of Walter changing the language at will officially came to an end. Feature

proposals are subjected to at least three rounds of community review before a rigorous evaluation by the language maintainers.

Until May 2019, the term "language maintainers" referred to Walter and Andrei. At DConf 2019, Andrei formally stepped down from his role as a decision maker in the development of the D programming language. He continues as an officer of the D Language Foundation, as a contributor, and as an active participant in the community. He was succeeded by Átila Neves, a long-time D user and contributor.

D, as a language and as a community, sprang to life from the ideas of a solitary programmer, shaped by the knowledge and experience he had gained over two decades. In the first few years of its existence, D grew and evolved both as a language and as a community to a point where it was ready to be adopted commercially. It survived a massive rift in the user base and the transition from one major version to another.

No one knows with certainty what D's future may hold, but there are those who say it looks rather bright indeed.

ACKNOWLEDGMENTS

Although D started as a one-person project, it would never have advanced beyond a curiosity without the enthusiastic support of hundreds of volunteers who contributed tirelessly, some for years, without recompense. Ascribing credit appropriately would be its own investigative project, and for each mentioned contributor there is the risk several others would be neglected. The authors of this paper would like to thank all who contributed, at any time and in any respect, to the progress of the D programming language.

REFERENCES

Andrei Alexandrescu. 2010. *The D Programming Language.* Addison-Wesley Professional, Boston.

Joel F. Bartlett. 1989. Mostly-Copying Garbage Collection Picks up Generations and C++. (oct 1989). https://www.hpl.hp.com/techreports/Compaq-DEC/WRL-TN-12.pdf

Kris Bell, Lars Ivar Igesund, Sean Kelly, and Michael Parker. 2008. *Learn to Tango with D.* Apress, New York.

Janet Feigenspan, Christian Kästner, Sven Apel, Jörg Liebig, Michael Schulze, Raimund Dachselt, Maria Papendieck, Thomas Leich, and Gunter Saake. 2013. Do background colors improve program comprehension in the `#ifdef` hell? *Empirical Software Engineering* 18, 4 (2013), 699–745.

Richard Jones. 1996. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* Wiley, New Jersey.

Donald Ervin Knuth. 1984. Literate programming. *Comput. J.* 27, 2 (1984), 97–111.

Douglas Kramer. 1999. API Documentation from Source Code Comments: A Case Study of Javadoc. In *Proceedings of the 17th Annual International Conference on Computer Documentation (SIGDOC '99).* ACM, New York, NY, USA, 147–153. https://doi.org/10.1145/318372.318577

Duc Le, Eric Walkingshaw, and Martin Erwig. 2011. `#ifdef` confirmed harmful: Promoting understandable software variation. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC).* IEEE, 143–150.

Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Sven Apel, Christian Kästner, Bruno Ferreira, Luiz Carvalho, and Baldoino Fonseca. 2018. Discipline matters: Refactoring of preprocessor directives in the `#ifdef` hell. *IEEE Transactions on Software Engineering* 44, 5 (2018), 453–469.

Microsoft 2015. *Using SAL annotations to reduce C/C++ code defects.* Technical Report. Technical report, Microsoft Developer Network, 2015. 62 Bibliography.

Norbert Pataki, Zalán Szűgyi, and Gergely Dévai. 2011. Measuring the overhead of C++ standard template library safe variants. *Electronic Notes in Theoretical Computer Science* 264, 5 (2011), 71–83.

PC-Week. 1988a. Zortech Readies What It Claims Is First C++ Compiler for PC Market. (May 1988).

PC-Week. 1988b. Zortech's Compiler Sparks Interest in the C++ Language. (Dec. 1988).

PC-Week. 1991. Symantec Buys Zortech To Vie In C++ Market. (Aug. 1991).

Carlo Pescio. 1997. Template Metaprogramming: Make parameterized integers portable with this novel technique. *C++ Report* 9, 7 (July-August 1997). http://www.eptacom.net/pubblicazioni/pub_eng/paramint.html

Janet Siegmund, Norbert Siegmund, Jana Fruth, Sven Kuhlmann, Jana Dittmann, and Gunter Saake. 2012. Program comprehension in preprocessor-based software. In *International Conference on Computer Safety, Reliability, and Security.*

Springer, 517–528.
Bjarne Stroustrup. 1985. *The C++ Programming Language.* Addison Wesley, Boston.
Tobias Wasserman and Christian Speer. 2007. *Programmieren in D.* Entwickler Press, Frankfurt.
Niklaus Wirth. 1971. The programming language Pascal. *Acta informatica* 1, 1 (1971), 35–63.

NON-ARCHIVAL REFERENCES

Alex. 2018. *Why do private member variables behaved like protected in the same module when creating deriving class?* https://forum.dlang.org/post/vauxqjztncmrbciqadga@forum.dlang.org (retrieved 12 March 2020)
Andrei Alexandrescu. 2005. *Lock-Free Programming.* https://nwcpp.org/january-2005.html (retrieved 12 March 2020)
Andrei Alexandrescu. 2009. On Iteration. (Nov.). http://www.informit.com/articles/article.aspx?p=1407357
Andrei Alexandrescu and Petru Marginean. 2000. Change the way you write exception-safe code—forever. (1 Dec.). http://www.drdobbs.com/cpp/generic-change-the-way-you-write-excepti/184403758 (retrieved 12 March 2020; also at Internet Archive 19 Feb. 2019 02:29:17).
Brad Anderson. 2004. *www.dsource.org.* https://digitalmars.com/d/archives//26588.html#N26588 (retrieved 12 March 2020)
Bill Baxter. 2005. *Meta-programming supported in the language.* https://digitalmars.com/d/archives/digitalmars/D/23175.html#N23175 (retrieved 12 March 2020)
BCS. 2007. *installing Phobos and Tango.* https://digitalmars.com/d/archives/digitalmars/D/installing_Phobos_and_Tango_56782.html#N56782 (retrieved 12 March 2020)
This post shows another common issue that arose from the Tango and Phobos split. The author is uncertain how to install both libraries at the same time.
BLS. 2007. *D Conference Tango Phobos.* https://www.digitalmars.com/d/archives/digitalmars/D/D_Conference_Tango_Phobos_57371.html#N57371 (retrieved 12 March 2020)
Bobef. 2005. *What about a real forum?* https://digitalmars.com/d/archives/digitalmars/D/20025.html#N20025 (retrieved 12 March 2020)
Walter Bright. 1979. Empire in Fortran-10 for the PDP-10. GitHub repository. https://github.com/DigitalMars/Empire-for-PDP-10 (retrieved 12 March 2020)
This is this source code for Walter's original Fortran-10 version of Empire.
Walter Bright. 2001a. *D Alpha Compiler.* Digital Mars. https://digitalmars.com/d/archives/2226.html#N2226 (retrieved 12 March 2020)
Walter Bright. 2001b. The D Programming Language. https://digitalmars.com/2001-01-31DraftDSpecification.pdf (retrieved 12 March 2020)
Walter Bright. 2001c. *D Programming Language spec.* https://digitalmars.com/d/archives/2.html#N2 (retrieved 12 March 2020)
The link to the spec in this newsgroup post is now a redirect to dlang.org. A much later 1.0 version of the old spec can be found at https://digitalmars.com/d/1.0/index.html.
Walter Bright. 2001d. Garbage Collection. https://digitalmars.com/d/1.0/garbage.html (retrieved 12 March 2020)
Walter Bright. 2001e. *Templates.* https://digitalmars.com/d/archives/369.html#N858 (retrieved 12 March 2020)
Walter Bright. 2002a. The D Programming Language. (1 Feb.). http://www.drdobbs.com/cpp/the-d-programming-language/184404968 (retrieved 12 March 2020; also at Internet Archive 2 Aug. 2019 08:34:17).
Walter Bright. 2002b. *Garbage Collection.* https://digitalmars.com/d/archives/4186.html#N4202 (retrieved 12 March 2020)
Walter Bright. 2002c. *One more vote for 'foreach'.* https://digitalmars.com/d/archives//8773.html#N8807 (retrieved 12 March 2020)
Walter Bright. 2002d. *Re: Helping Out.* https://digitalmars.com/d/archives/8574.html#N8575 (retrieved 12 March 2020)
Walter Bright. 2003. *Re: DMD 0.76 release.* https://digitalmars.com/d/archives/19380.html#N19389 (retrieved 12 March 2020)
Walter Bright. 2004a. *The D Programming Language.* https://nwcpp.org/october-2004.html (retrieved 12 March 2020)
Walter Bright. 2004b. *D wish list for 2004.* https://digitalmars.com/d/archives//20938.html#N20951 (retrieved 12 March 2020)
Walter Bright. 2004c. *Empire is now in D.* https://digitalmars.com/d/archives/digitalmars/D/13.html#N13 (retrieved 12 March 2020)
Walter Bright. 2004d. *mixins.* https://digitalmars.com/d/archives/digitalmars/D/1228.html#N1228 (retrieved 12 March 2020)
Walter Bright. 2004e. *new D newsgroups.* https://digitalmars.com/d/archives/digitalmars/D/bugs/2.html#N2 (retrieved 12 March 2020)
Walter Bright. 2005a. *Announcing new newsgroups.* https://digitalmars.com/d/archives/digitalmars/D/announce/2.html#N2 (retrieved 12 March 2020)
Walter Bright. 2005b. DMD 0.132 release - introducing the new Ddoc documentation generator. https://www.digitalmars.com/d/archives/digitalmars/D/announce/1553.html#N1553 (retrieved 12 March 2020)

Walter Bright. 2005c. *Re: Meta-programming supported in the language.* https://digitalmars.com/d/archives/digitalmars/D/23175.html#N23345 (retrieved 12 March 2020)

Walter Bright. 2005d. *Re: Meta-programming supported in the language.* https://digitalmars.com/d/archives/digitalmars/D/23175.html#N23568 (retrieved 12 March 2020)

Walter Bright. 2006a. *D Conference 2007.* https://digitalmars.com/d/archives/digitalmars/D/announce/D_Conference_2007_5536.html#N5536 (retrieved 12 March 2020)

This was the first newsgroup post Walter wrote that referred to the conference. It was an announcement that he and Brad Roberts had been discussing the potential for a conference and was intended to "gauge the level of interest."

Walter Bright. 2006b. *New newsgroup digitalmars.D.dwt.* https://digitalmars.com/d/archives/digitalmars/D/dwt/1.html#N1 (retrieved 12 March 2020)

Despite Walter's enthusiasm, DWT was never widely adopted. As of 2020, DWT is no longer considered the "official" GUI toolkit for D.

Walter Bright. 2006c. *New web interface to forums.* https://digitalmars.com/d/archives/digitalmars/D/announce/New_web_interface_to_forums_6086.html#N6086 (retrieved 12 March 2020)

Walter Bright. 2007a. *DMD 1.00 - here it is!* https://digitalmars.com/d/archives/digitalmars/D/announce/DMD_1.00_-_here_it_is_6581.html#N6581 (retrieved 12 March 2020)

Walter Bright. 2007b. *DMD 2.000 alpha release.* https://digitalmars.com/d/archives/digitalmars/D/announce/DMD_2.000_alpha_release_9037.html#N9037 (retrieved 12 March 2020)

Walter Bright. 2007c. *Re: So, what happened?* https://digitalmars.com/d/archives/digitalmars/D/So_what_happened_57378.html#N57429 (retrieved 12 March 2020)

Walter Bright. 2007d. *Tango 0.95 beta1 released.* https://digitalmars.com/d/archives/digitalmars/D/announce/Tango_0.95_beta1_released_7035.html#N7096 (retrieved 12 March 2020)

Tango was a large, complex project. There were no other D projects of the same scale and scope at the time. Walter was genuinely impressed by what the Tango team had accomplished and pleased to see D employed in such a project.

Walter Bright. 2008. *Removing D embedded in HTML feature.* https://digitalmars.com/d/archives/digitalmars/D/Removing_D_embedded_in_HTML_feature_68747.html#N68747 (retrieved 12 March 2020)

Walter Bright. 2009a. C's biggest mistake. (22 Dec.). https://digitalmars.com/articles/b44.html (retrieved 12 March 2020)

Walter Bright. 2009b. Designing Safe Software Systems Part 2. (Nov.). https://digitalmars.com/articles/b40.html (retrieved 12 March 2020)

In this article, originally published at Dr. Dobb's, Walter shows how the ideas he outlined in his article "Safe Systems from Unreliable Parts" can be incorporated into software.

Walter Bright. 2009c. Safe Systems from Unreliable Parts. (Oct.). https://digitalmars.com/articles/b39.html (retrieved 12 March 2020)

This article was originally published at the Dr. Dobb's website, where Walter maintained a blog for some time. In it, Walter explains that safe systems must have an independent backup or a failsafe shutdown that is decoupled from the primary system.

Walter Bright. 2012. Component Programming in D. (2 Oct.). http://www.drdobbs.com/architecture-and-design/component-programming-in-d/240008321 (retrieved 12 March 2020; also at Internet Archive 21 July 2019 12:00:10).

Walter Bright. 2013a. *DConf 2013 - what a show!* https://forum.dlang.org/post/km2fr9$1jn0$1@digitalmars.com (retrieved 12 March 2020)

Walter Bright. 2013b. *Re: Registration now open on dconf.org.* https://forum.dlang.org/post/kfjg8gap1@digitalmars.com (retrieved 12 March 2020)

Walter Bright. 2016. *Re: Vision for the D language - stabilizing complexity?* https://forum.dlang.org/post/nmf48b$1ckm$1@digitalmars.com (retrieved 12 March 2020)

Walter Bright. 2018. *Re: Why do private member variables behaved like protected in the same module when creating deriving class?* https://forum.dlang.org/post/pr110b$9j5$1@digitalmars.com (retrieved 12 March 2020)

Walter Bright and Andrei Alexandrescu. 2007. *The Future of D.* https://dconf.org/2007/WalterAndrei.pdf/WalterAndrei.pdf (retrieved 12 March 2020) Also at http://s3.amazonaws.com/dconf2007/WalterAndrei.pdf

This PDF file contains the slides from the two presentations that Walter and Andrei gave at the 2007 D Programming Language Conference.

Tim Burrell. 2008. *Standard Library Concerns (Phobos / Tango).* https://digitalmars.com/d/archives/digitalmars/D/Standard_Library_Concerns_Phobos_Tango_65925.html#N65925 (retrieved 12 March 2020)

The author of this newsgroup post expressed a common complaint about Phobos (it's difficult to contribute to the project) and notes the confusion caused by having two standard libraries. He proposed that Walter cede control of Phobos and that Gregor Richardson's bridge library, Tangobos, be adopted as the official standard library. Many of the responses were in favor of idea. This discussion thread is but one example of conversations that arose from the confusion and frustration at having two incompatible standard libraries.

D Language Foundation 2020a. *Areas of D Usage*. https://dlang.org/areas-of-d-usage.html (retrieved 12 March 2020)
This page provides an overview of the real-world D usage.
D Language Foundation 2020b. *Organizations using the D Language*. https://dlang.org/orgs-using-d.html (retrieved 12 March 2020)
This is a list of organizations known to be using the D programming language. It is generally updated upon request.
Jonathan Davis. 2010. *Answer: Does the D language have multiple standard libraries and issues with GC?* https://stackoverflow.com/a/3206985 (retrieved 12 March 2020)
Jonathan Davis. 2014. *Answer: What is the accepted way to do unit testing in D?* https://stackoverflow.com/a/22148472 (retrieved 12 March 2020)
Jonathan M. Davis is a long-time D user and contributor. He is well known in the D community for his in-depth and informative answers to questions in the D forums and elsewhere, such as on Stack Overflow. This particular answer is cited as a reference for its on-the-nose explanation of D's unit testing feature. As we say in the paper, it is a simple approach, but more complex testing is possible when making use of other D features, such as compile-time reflection.
D Language Foundation 2013. *DConf Homepage*. D Language Foundation. https://dconf.org/ (retrieved 12 March 2020)
The web pages for every D Programming Language Conference from 2013 onward are available from here. DConf 2020 was canceled due to the outbreak of the COVID-19 virus.
Digital Mars 1999. *Digital Mars C, C++, and D Compilers*. Digital Mars. https://digitalmars.com/ (retrieved 12 March 2020)
Digital Mars 2002a. *news.digitalmars.com - digitalmars.D*. https://digitalmars.com/d/archives/index2002.html (retrieved 12 March 2020)
This is the Digital Mars D newsgroup archive for 2002.
Digital Mars 2002b. What's New for D 0.15. D changelog. https://digitalmars.com/d/1.0/changelog1.html#new015 (retrieved 12 March 2020)
Digital Mars 2002c. What's New for D 0.40. D change log. https://digitalmars.com/d/1.0/changelog1.html#new040 (retrieved 12 March 2020)
Digital Mars 2003a. *news.digitalmars.com - digitalmars.D*. https://digitalmars.com/d/archives/index2003.html (retrieved 12 March 2020)
This is the Digital Mars D newsgroup archive for 2003.
Digital Mars 2003b. What's New for D 0.71. D change log. https://digitalmars.com/d/1.0/changelog1.html#new071 (retrieved 12 March 2020)
Digital Mars 2003c. *What's New for D 0.76*. Digital Mars. https://digitalmars.com/d/1.0/changelog1.html#new076 (retrieved 12 March 2020)
Digital Mars 2005a. What's New for D 0.124. D change log. https://digitalmars.com/d/1.0/changelog1.html#new0124 (retrieved 12 March 2020)
Digital Mars 2005b. *What's New for D 0.126*. Digital Mars. https://digitalmars.com/d/1.0/changelog1.html#new0126 (retrieved 12 March 2020)
Digital Mars 2006. Version D 0.149. D change log. https://digitalmars.com/d/1.0/changelog2.html#new0149 (retrieved 12 March 2020)
Digital Mars 2007a. Change Log: 2.000. D change log. https://dlang.org/changelog/2.000.html (retrieved 12 March 2020)
Digital Mars 2007b. *News Groups*. https://digitalmars.com/NewsGroup.html (retrieved 12 March 2020)
This page provides links to the newsgroups, the modern web interface to each newsgroup, the mailing lists, and the newsgroup archives.
Digital Mars 2007c. Version D 1.00. D change log. https://digitalmars.com/d/1.0/changelog2.html#new1_00 (retrieved 12 March 2020)
Digital Mars 2012a. *Embedding D in HTML*. https://digitalmars.com/d/1.0/html.html (retrieved 12 March 2020)
Digital Mars 2012b. *Template Mixins*. https://digitalmars.com/d/1.0/template-mixin.html (retrieved 12 March 2020)
D Language Foundation 2015a. `std.experimental.allocator`. D Language Foundation. https://dlang.org/phobos/std_experimental_allocator.html (retrieved 12 March 2020)
D Language Foundation 2015b. *The D Language Foundation*. D Language Foundation. https://dlang.org/foundation/about.html (retrieved 12 March 2020)
D Language Foundation [n.d.]. *Type Qualifiers*. D Language Foundation. https://dlang.org/spec/const3.html (retrieved 12 March 2020)
Patrick Down. 2002. *Mixins*. https://digitalmars.com/d/archives/9093.html#N9093 (retrieved 12 March 2020)
DSource.org 2004. *Ares Project Page*. http://www.dsource.org/projects/ares (retrieved 12 March 2020)
Dsource.org 2004. *DSource Projects*. http://www.dsource.org/projects/ (retrieved 12 March 2020)
DSource.org 2004a. *DWT Project Page*. http://www.dsource.org/projects/dwt (retrieved 12 March 2020)
DSource.org 2004b. *Mango Project Page*. http://www.dsource.org/projects/mango (retrieved 12 March 2020)
DWiki 2018. *Documentation Generators*. https://wiki.dlang.org/Documentation_Generators (retrieved 12 March 2020)

EDM/2. 2017. *Zortech C++.* http://www.edm2.com/index.php/Zortech_C%2B%2B (retrieved 12 March 2020)
The "Electronic Developer Magazine for OS/2" was the only online source we could find with what appears to be a full listing of Zortech C++ and Symantec C++ compiler versions.

EDM/2. 2018. *Symantec C++.* http://www.edm2.com/index.php/Symantec_C%2B%2B (retrieved 12 March 2020)
The "Electronic Developer Magazine for OS/2" was the only online source we could find with what appears to be a full listing of Zortech C++ and Symantec C++ compiler versions.

Ivan Frohne. 2001. *D - Multidimensional arrays; reference operator; longs.* https://digitalmars.com/d/archives/4.html#N4 (retrieved 12 March 2020)

Hacker News 2015. *DFeed.* https://news.ycombinator.com/item?id=9990763 (retrieved 12 March 2020)

Henning Hasemann. 2007. *phobos / tango / ares.* https://digitalmars.com/d/archives/digitalmars/D/learn/phobos_tango_ares_6358.html#N6358 (retrieved 12 March 2020)
This newsgroup post is demonstrative of the confusion some D users had at the time. Between Phobos, Tango, and Sean Kelly's Ares (a Tango precursor), new D users especially, but also those who did not closely follow the newsgroups, were often uncertain about the differences between the libraries or how and when to use them.

Brian Hay. 2006. *little thing - D and phobos naming conventions.* https://digitalmars.com/d/archives/digitalmars/D/little_thing_-_D_and_phobos_naming_conventions_43299.html#N43299 (retrieved 12 March 2020)
This newsgroup post is demonstrative of complaints about consistency in the Phobos API in terms of naming convention and API functionality.

Lars Ivar Igesund. 2007. *Tango 0.95 beta1 released.* https://digitalmars.com/d/archives/digitalmars/D/announce/Tango_0.95_beta1_released_7035.html#N7035 (retrieved 12 March 2020)

Lars Ivar Igesund. 2012. *What happened?* http://www.dsource.org/projects/tango/forums/topic/920 (retrieved 12 March 2020)

Lars Ivar Igesund. 2018. Facebook message.
In a discussion through the Facebook Messenger application, Tango developer Lars Ivar Igesund recounted his memories of Tango's development and the meeting at the Seattle conference. Another Tango developer, who has asked to remain anonymous, recounted his own memories via a conversation over Skype. Their memories of the meeting diverge, with one suggesting a third-party helped organize it through IRC and the other that it came together spontaneously. The D newsgroup archives provide little help here; we were unable to uncover any postings mentioning the meeting prior to the conference. Walter's memory of the meeting is also unclear. The two points upon which the parties agree are described in the text.

Mikkel Jørgensen. 2003. *interfaces and mixins.* https://digitalmars.com/d/archives/20345.html#N20345 (retrieved 12 March 2020)

Sean Kelly. 2006. *Announcing a New Library.* https://digitalmars.com/d/archives/digitalmars/D/announce/Announcing_a_new_library_6522.html#N6522 (retrieved 12 March 2020)
This newsgroup post was published to announce the existence of Tango and its design features (modularity, atomic mark/sweep garbage collection, support for concurrency, etc.). The beta release was planned to follow the release of DMD 1.000.

Sean Kelly. 2007. *Re: So, what happened?* https://digitalmars.com/d/archives/digitalmars/D/So_what_happened_57378.html#N57419 (retrieved 12 March 2020)

Jan Knepper. 2001. *Digital Mars D Newsgroups!* https://digitalmars.com/d/archives/1.html#N1 (retrieved 12 March 2020)

Richard Krehbiel. 2001. *Macros.* https://digitalmars.com/d/archives/30.html#N30 (retrieved 12 March 2020)

Mario Kröplin. 2017. Unit Testing in Action. (20 Oct.). https://dlang.org/blog/2017/10/20/unit-testing-in-action/ (retrieved 12 March 2020)
Mario Kröplin is a developer at Funkwerk AG, a German company that first chose to use D in 2008 and were the second company to use D in production. In this blog post, Mario explores D's built-in unit testing feature, touches on some of the third-party libraries that enhance it, and shows how to implement slightly more complex testing without resorting to any third-party libraries.

Lockheed Martin 2005. Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program. Document. http://www.stroustrup.com/JSF-AV-rules.pdf (retrieved 12 March 2020)

Christopher Miller. 2004. *DFL.* http://www.dprogramming.com/dfl.php (retrieved 12 March 2020)
Although the DFL page is still available as of 2020, and some D newcomers sometimes download the source code with the intent to use it, the project has been long dead.

Christopher Miller. 2006. *#D on IRC.* https://digitalmars.com/d/archives/digitalmars/D/announce/2618.html#N2618 (retrieved 12 March 2020)

Ocean 2017. sociomantic-tsunami/ocean. GitHub repository. https://github.com/sociomantic-tsunami/ocean (retrieved 12 March 2020)

Andrew Orlowski. 2001. Linux is the future, say former MS execs. (7 May). https://www.theregister.co.uk/2001/05/07/linux_is_the_future_say/ (retrieved 12 March 2020)

This article provides some additional information about Eric Engstrom and how he left Microsoft to found Chromium.

Vladimir Panteleev. 2012. DFeed, an NNTP/mailing list web frontend/forum software, news aggregator and IRC bot. GitHub repository. https://github.com/CyberShadow/DFeed (retrieved 12 March 2020)

Vladimir Panteleev. 2014. *dsource.org moved.* https://forum.dlang.org/post/jclqnnkkcbqnmehpnokv@forum.dlang.org (retrieved 12 March 2020)

Mike Parker. 2006. *My D Blog.* https://digitalmars.com/d/archives/digitalmars/D/announce/2563.html#N2563 (retrieved 12 March 2020)

Michael Parker. 2016. The Why and Wherefore of the New D Improvement Proposal Process. (20 July). https://dlang.org/blog/2016/07/20/the-why-and-wherefore-of-the-new-d-improvement-proposal-process (retrieved 12 March 2020)

Michael Parker. 2017. Project Highlight: Funkwerk. (28 July). https://dlang.org/blog/2017/07/28/project-highlight-funkwerk/ (retrieved 12 March 2020)

Rejected Software 2012a. *The D Package Registry.* https://code.dlang.org/ (retrieved 12 March 2020)

The D Package Registry lists a growing number of open source D projects that have been registered for use with DUB, the official D build tool and package manager. On the registry page of each package, one can find a link to the package's git repository. It would be interesting to crawl the registry and examine all the D source code there to gather data on the usage of specific D features, such as the built-in unit tests.

2012b. D Slices. https://vibed.org/features#native-code (retrieved 12 March 2020)

Rejected Software 2012c. ddox. GitHub repository. https://github.com/rejectedsoftware/ddox (retrieved 12 March 2020)

Gregor Richards. 2007. *Tangobos += existence.* https://digitalmars.com/d/archives/digitalmars/D/Tangobos_existence_52380.html#N52380 (retrieved 12 March 2020)

Gregor Richards was the maintainer of a popular build tool for D called DSSS. Tangobos was his attempt to solve the Tango/Phobos divide. It never caught on. The two Tango developers who provided feedback for this paper have conflicting memories about the library. One of them recalls the team was not enthused about it, preferring to see Tango replace or merge with Phobos. The other remembers the team viewing Walter's adoption of Tangobos as a viable solution to the split, though it was an alternative to which Walter would never have agreed.

Brad Roberts. 2006. *Re: D Conference 2007.* https://digitalmars.com/d/archives/digitalmars/D/announce/D_Conference_2007_5536.html#N5538 (retrieved 12 March 2020)

In the discussion thread opened by Walter's post to gauge interest in a D conference, Brad made it clear that there was no budget to sponsor anyone and that they were trying to obtain sponsored space. What he didn't say was that the potential sponsor was his employer, Amazon.

Brad Roberts. 2007a. *Conference 2007—D Programming Language.* http://d.puremagic.com/conference2007/index.html (retrieved 12 March 2020)

Brad Roberts. 2007b. *D Developers Conference.* https://digitalmars.com/d/archives/digitalmars/D/announce/D_Developers_Conference_7642.html#N7642 (retrieved 12 March 2020)

This newsgroup post announcing that a conference location had been secured also asked the community for ideas about organizational details. It was published under the user name "user domain.invalid", but in a subsequent reply Brad made it clear he was the author.

Adam Ruppe. GitHub repository. adrdox. GitHub repository. https://github.com/adamdruppe/adrdox (retrieved 12 March 2020)

Steven Schveighoffer. 2011. D Slices. https://dlang.org/articles/d-array-article.html (retrieved 12 March 2020)

SD West [n.d.]. *Software Development Conference & Expo.* Archived at Internet Archive: https://web.archive.org/web/20040318050849/http://sdexpo.com/ (18 March 2004 05:08:49)

Tim Sweeney. 2001. *Re: Macros.* https://digitalmars.com/d/archives/30.html#N339 (retrieved 12 March 2020)

Mark T. 2003. *D wish list for 2004.* https://digitalmars.com/d/archives//20938.html#N20938 (retrieved 12 March 2020)

Tango-D2 2012. SeigeLord/Tango-D2. GitHub repository. https://github.com/SiegeLord/Tango-D2 (retrieved 12 March 2020)

Tango Wiki 2008. *Tango Conference 2008, Torun, Poland.* http://dsource.org/projects/tango/wiki/TangoConference2008 (retrieved 12 March 2020)

H. S. Teoh. 2013. Component programming with ranges. (Aug.). https://wiki.dlang.org/Component_programming_with_ranges (retrieved 12 March 2020)

The Eclipse Foundation 2003. *SWT: The Standard Widget Toolkit.* http://www.eclipse.org/swt/ (retrieved 12 March 2020)

The GTK Team 1997. *The GTK Project.* https://www.gtk.org/ (retrieved 12 March 2020)

D Language Foundation 2017. *The D Blog.* D Language Foundation. https://dlang.org/blog/ (retrieved 12 March 2020)

dlang.org 2018. *Contributors.* https://dlang.org/foundation/contributors (retrieved 12 March 2020)

Since 2018, this page has been automatically generated on a daily basis to provide a list of all contributors to all of official D tool and documentation projects at GitHub. As such, it is missing the names of contributors from D's earlier years who are no longer actively contributing.

Bernard Vaillot, Andre Barro, and Greg Gransden. 2003. Air Emergency. Docudrama series. https://imdb.com/title/tt2091498/ (retrieved 12 March 2020)

This television series dramatizes air accidents and reconstructs them through the accounts of aviation experts and witnesses. It also explores the lessons learned from accident analysis. We cite it as an excellent source to demonstrate that arriving at an intuitive design is often a proces of trial and error.

Dimitri Van Heesch. 1997. *Doxygen.* http://www.doxygen.nl/ (retrieved 12 March 2020)

Mike Wey. 2004. *GtkD.* https://gtkd.org/ (retrieved 12 March 2020)

As of 2020, Mike Wey still maintains GtkD. Its user base appears to have grown. Ron Tarrant started gtkdcoding.com in early 2019 to publish GtkD tutorials and subsequently attracted a dedicated following.

Wikipedia. 2010. *Datalight.* https://en.wikipedia.org/wiki/Datalight (retrieved 12 March 2020)

Roy Sherrill, the founder of Datalight, was already selling a so-called "Small-C" compiler. When he licensed Northwest C from Walter, he gained a better compiler and Walter gained access to a larger market.

Matthew Wilson. 2003a. *=== / == / !== / != - a recipe for disaster!* https://digitalmars.com/d/archives/18188.html#N18188 (retrieved 12 March 2020)

Matthew Wilson. 2003b. *Identity & equivalence.* https://digitalmars.com/d/archives/12141.html#N12141 (retrieved 12 March 2020)

Matthew Wilson. 2003c. *null == o?* https://digitalmars.com/d/archives//12144.html#N12144 (retrieved 12 March 2020)

Georg Wrede. 2006. *STEP UP PHOBOS DEVELOPMENT 10 FOLD.* https://digitalmars.com/d/archives/digitalmars/D/38911.html#N38911 (retrieved 12 March 2020)

This newsgroup post and the subsequent discussion serve as an example of the opinions some D users had of Phobos at the time. Much of the discussion revolves around the minimalism of the Phobos API in comparison to the standard libraries of Java and C#.

James Yates. 2002. *'foreach' style loop?* https://digitalmars.com/d/archives//5238.html#N5238 (retrieved 12 March 2020)