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

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 those languages' efficiency, low-level access, and Algol-style syntax. 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, Bright 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. Best known of these would be the Zortech C++ compiler, the first (and to date only) C++-to-native compiler 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 by that time mature C/C++ back end (optimizer and code generator) that Zortech had sold to Symantec (as part of Symantec's acquisition of Zortech) in August 1991, to subsequently license it back in April 2000 as Symantec exited the compiler business.

Between 1999 and 2010, Bright 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'd 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, 2s 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 transferrable. 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.

XXXX-XXXX/2018/11-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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

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

ACM Reference Format:
Walter Bright, Andrei Alexandrescu, and Michael Parker. 2018. Origins of the D Programming Language. 1, 1 (November 2018), 29 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 HISTORICAL CONTEXT AND INFLUENCES

D was originally designed and implemented by Walter Bright. Its origins are inextricably bound with his experience in computer programming, developing compilers, supporting compilers, working on teams, and even working on aircraft design.

In 1979 Bright earned a Mechanical Engineering degree from Caltech. Having no formal training in Computer Science, he taught himself BASIC during his college years to write computer games. 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, such as the commercially successful game Empire [12].

In 1984 he wanted to learn more about implementing a compiler (mainly to make Empire faster), so he took 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). Coupled with actually implementing DFA in Datalight Optimum C, the knowledge he gained provided insight into the kinds of compiler optimizations that could be expected, what was unreasonable for a compiler to do, the kinds of language features that contributed to better optimization, and what detracted from it. This experience would influence the design of D. For example, D's default initialization semantics can result in double stores for a naïve compiler. Modern compilers can be expected to perform DFA, which includes dead assignment elimination [51], and the redundant store can be automatically eliminated if it is unnecessary.

In 1999 Bright retired from the workforce, but soon became eager to work on a new project. In his years maintaining a C++ compiler, he had often considered the language's strengths and weaknesses, and the lessons that could be derived from them in the design of a new language. He decided it was time to test those ideas. In October, 1999 [74], he created a new company, Digital Mars, and in November of that year began work on the design and implementation of a new programming language that he called "Mars".

One of the earliest design decisions that Bright made was to make his new language easy to use with software written in C. Many widely-used libraries are implemented in C [108], or have a C interface. Bright wanted to guarantee that Mars code could interface with existing C code without difficulty so that established software companies would have an easy path to adopt the Mars language.

Bright went a step further and established that Mars programs would be binary compatible with C—the output of a Mars compiler could be linked into a program with the output of a C compiler. To save time, he decided to use the back end of the C++ compiler he had developed and maintained for two decades 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.

With the C++ compiler license in hand, Bright rebranded it as Digital Mars C++ and began selling it from the company web site [61]. He also began telling acquaintenances about his new programming language. It wasn't long before they were jokingly referring to it as "D". The name

stuck, and on December 8, 2001, Bright released the first prototype of DMD, the Digital Mars D compiler [59].

1.1 Boeing Commercial Airplane Company

Bright worked at Boeing from 1979 to 1982 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 behind 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 by implementing backups for all critical systems along with mechanisms for the detection of faults and failover to the backup [13, 15].

These ideas are expressed in D in the form of contracts, invariants, and assertions. Any failures detected by these features are considered programming bugs, i.e. the program is not in a defined state, the error is not recoverable, and the program must exit as soon as possible rather than attempting to recover and continue.

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. D provides mechanisms (such as modules, contracts, and invariants) for enabling the use of well-defined and enforced interfaces.

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 often is the result of a cycle of analyzing mistakes and accidents [80] followed by improvements. For D, this led to an emphasis on reusing existing syntax and semantics familiar and in wide use rather than creating gratuitous incompatibilities.

Another important lesson from airliner design is the attitude toward, and approach to, human error. Many programming languages frame programmer error as mainly an education matter. 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 the 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. Such insights found their way in a variety of aspects of the D language, such as:

- The lower case 'l' cannot be used as an integer literal suffix (i.e. 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 upper case 'L' can be used. (Also found in the JSF Coding Standards [62].)
- Implicit declaration of variables is not allowed, only explicit.
- 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.

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

```
2135555565          // hard to see what that is
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
```

- Cannot use ';' as an empty statement, must use '{' and '}':

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

- Changed C grammar to eschew confusing forms, such as a < b < c, from D

```
((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. Bright 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. Such confusing uses as a < b < c are syntactically illegal and produce a compiler error.

Simple, ergnomic language features can enable the emergence of robust alternatives to code fraught with the risk of human error. Consider a simple C function:

```
#include <stdbool.h>
#include <stdio.h>
bool find(double *array, size_t dim, double t) {
   int i;
   for (i = 0; i <= dim; i++);
   {
      int v = array[i];
      if (v == t)
         return true;
   }
}
```

which has the following errors:

- i should have type size_t
- '<=' should be '<'
- extraneous ';'
- v should have type double
- missing return

In addition, the function can be called with the wrong arguments by matching the wrong `dim` with the pointer. The D form would be:

```
bool find(double[] array, double t) {
   foreach (v; array)
   {
      if (v == t)
         return true;
   }
   return false;
}
```

whereby

- loop index is not required
- loop termination is implicit
- ';' is not allowed as a loop body
- type of v is inferred
- falling off end of function with no return value is not allowed
- calling `find` with a correct array is easy, calling it with an ill-formed array requires unusual and bulky constructs

1.2 The C and C++ Connection

The growing popularity of the IBM PC in the early 1980s offered Bright the opportunity to port the Empire game to the platform, which in turn prompted the question of which compilers were available. Implementations of Pascal [113] 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. Bright set out to write a C compiler that was better than existing implementations, particularly one that could perform improved optimizations.

In 1982, he created the Northwest Software company to sell his new C compiler, which he branded Northwest C. In April, 1985, Bright entered into a contract with the software company Datalight [109] and the compiler became Datalight C. In 1987, it was rebranded as Datalight Optimum C and acquired a key feature—data flow optimizations. Data flow analysis was new for MS-DOS compilers at the time, 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. It was then known as Zorland C.

Bjarne Stroustrup's book "The C++ Programming Language" [91] had convinced Bright 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 only available on Unix, and 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.

Released in early 1988 (after Zorland underwent a name change to avoid legal issues with Borland), Zortech C++ became the first native C++ compiler for DOS [76] (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, Zortech C++'s overnight success improved the landscape of DOS and Windows programming, and the popularity of C++ surged along with it [77]. Subsequently, Borland and Microsoft would focus their efforts on building C++ compilers as well. Zortech was acquired

by Symantec in 1991 for $12.6 million [78]. Bright's compiler would be known as Symantec C++ until its final rebranding in 2001 as Digital Mars C++.

In the wake of these developments, C was a very strong influence on D. The syntax of most programming constructs in D is instantly recognizable to the C programmer. Going further, the semantics are either the same, such as the integer promotion rules, or will issue a compile-time error, such as implicit narrowing conversions or dangerous pointer casts. The goal was to allow the by-rote conversion of C code to D. If it compiled, it would run with the same semantics; if not, it would produce a visible (and hopefully easy to correct) compile-time error.

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 was replaced with modules, imports, version declarations, and `static if` in a hygienic manner.

Because of the existing C code base from which the D compiler was derived, and to make up for the absence of preexisting libraries for D, the D language was designed to be ABI compatible with C.

Negative influences from C that were avoided in D include the necessity for forward declarations, the decay of arrays to pointers when passed to functions [14], uninitialized variables, and implicit narrowing conversions (assigning an integer value of one type to a variable of a smaller-sized type without a cast).

In a separate development starting in the early 1990s, the Numerical C Extensions Group was created with the aim of improving C's numerical programming capability. In an effort to improve the situation, the group released a detailed proposal, "Floating-Point C Extensions" [42]. Most of the proposal was implemented in Bright's C compiler and was later carried over into D.

Bright's experience with C++—both on the using and on the implementing side—has had a strong influence on the D language. D would acquire C++'s value semantics with user-defined copy semantics, though Bright drew a clear distinction between value types (`struct`) and reference types (`class`) in D. Other ideas borrowed from C++ include efficient vtable-driven virtual dispatch of virtual functions, protection levels, and exceptions. Multiple inheritance of implementation was shunned because it had little benefit and added undue complexity in implementation. Instead, Bright chose an approach inspired by Java [6]—single inheritance of implementation and multiple inheritance of interface. All of these features made their way into the language over the course of 2000 and 2001.

Bright found C++ templates useful, but felt they added disproportionate complexity to the front end. He had also heard, and agreed with, other C++ users who said the syntax was difficult to understand. For D, he took some time to devise a radically simpler template syntax starting in 2002. His key insight was that a template declaration could be seen simply as a conventional function declaration with the addition of a set of compile-time parameters. From that perspective, there was no need for a special syntax in the declaration of a template function—it was sufficient to simply require an extra parameter list:

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

Bright also implemented a clean syntax for template instantiation. He selected the character '`!`' as the template instantiation operator. It is paired with the parameter list to simplify the back-end implementation, as in `min!(int)(a, b)`, and is required when type inference is not possible.

1.3 Influences from other languages

BASIC. Bright learned programming with BASIC. He moved on to more advanced languages, but BASIC retained tremendous popularity with developers. Bright thought that a major attraction of BASIC was its simplicity in handling strings in comparison with C, C++, Fortran, etc., and felt that this simplicity was worth carrying into D.

Fortran. Fortran-11 was commonly used at Boeing for custom numerical analysis programs. An appreciation for the art of numerical programming seeped into the D language; strong support for numerical analysis would be a priority.

Ada. Ada's [5] notions of program correctness were impressive. A direct influence on D was allowing underscores in numeric literals. Ada specified `in`, `out`, and `in out` parameters, which also found their way into D.

Modula-2. Symantec acquired the MultiScope debugger for use with Symantec C++. It was implemented in Modula 2, and Symantec wished to translate it to C++. The debugger source made extensive use of the Modula 2 `WITH` statement. In order to ease the translation effort, Bright was asked to extend C++ with a `with` statement [102]. The `with` statement in D [103] was derived from this.

Java. When Symantec decided to build a Java development suite, Bright was asked to reimplement the existing Sun Java compiler in C++ in order to speed up compilation. The resulting product was released as Symantec Café. The project revealed that OOP could be far simpler than the C++ model. Although Java was unsuitable for the demands of systems-level programming, its influence on D is visible in its single inheritance with multiple interfaces model of OOP, implicit dereferencing of object pointers, the use of Unicode strings, and nested/local classes with context pointers.

2 D'S EARLY YEARS

Bright left Symantec in 1999 after the company left the programming language business. Being free of obligations, he decided to create a new programming language based on his accumulated experience. He founded Digital Mars as a one-man company for the purpose of distributing his C++ compiler (under a licensing agreement with Symantec), and the nascent D compiler.

Bright was initially the sole developer for both the creation and implementation of D until the first alpha release in August 2001. From that time, the D programming language community began to coalesce and became increasingly involved. Bright started accepting contributions from the community, and these increased until the first major release in January 2007. Phobos, the D runtime library, was initially written by Bright, but morphed into much more of a community effort. Throughout this period he remained the sole designer and implementer of the core language.

Having written and supported professional compilers for C, C++, Java, and Javascript for decades, Bright 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.

The motivation for D did not change over the period of development leading to the first major release, but notions of how best to develop code are constantly evolving. After the first release, it became apparent that multithreaded programming, memory safety, and functional programming had become much more important, and D would need to adapt.

2.1 Design Goals

D studiously avoids decreeing one overarching principle that overrides all others.

The design goals for a complex and ambitious language are bound to conflict. D has been, and still is, driven not by ideology but by practicality. Conflicts are resolved on a case-by-case basis, using the best judgment available at the time.

For example, an often-mentioned ideal when designing a language is *orthogonality*—each feature should be completely orthogonal to others. As Hoare argues, however [45], orthogonality is desirable as a means to a higher-end goal (such as overall simplicity), lest it hurt practicality by allowing a variety of useless combinations that weigh down the language's "real estate." Designing a language is not unlike designing a house in terms of trade-offs—to make the closets bigger, the bathrooms must become smaller. All language features suffer trade-offs with other features, so Bright chose not to attempt a strict hierarchy of principles. Instead, each feature is judged by its utility and its relationship with other features. D does have overlapping features, such as signed and unsigned integral types. As a contrasting example, in Java the overlap between signed and unsigned integers is handled by forgoing an unsigned integer type, with the consequence that when an unsigned type is actually required, the user is forced to use a comparatively awkward API [48].

That said, Bright did commit to a few core choices when defining D on the landscape of programming languages.

2.1.1 General Purpose and Native. D was designed as a general purpose, native-code-generating language for a very broad spectrum of uses. It was meant to be fit for systems programming and application programming. It would be a polyglot language, pulling the best ideas from many languages, having enough capability to obviate the need for multi-language applications, such as projects that combine C and Python.

The general purpose, native niche was dominated at the time by C and C++. Those languages were weighed down by their past and backwards compatibility considerations. There was much to learn from other languages that could be applied, such as modules, if one was willing to dispense with obsolete ideas (such as the preprocessor, and redundant manually-written headers).

2.1.2 Execution Efficiency. Bright's goal was that the execution efficiency of D code be similar to the equivalent C code. Execution size was a constant value larger, that constant value resulting from the size of the core functionality (like the GC) in the runtime library. The marginal size of the generated code was to be the same as for C.

2.1.3 Scalability. Programs are constantly increasing in size and complexity. Language features must accommodate large scale projects being written by a team. Features based on well-known matters of principle, such as static typing, separate compilation, modularity, and implementation hiding, are critical to acheiving that goal.

In addition, good scalability is dramatically helped by simple tool support for unit testing and semi-automated documentation, features described in more detail below.

3 DISTINGUISHING FEATURES OF THE D LANGUAGE AND THEIR GENESIS

Much of the conventional part of the D language is deliberately unremarkable. D is C-style in syntax and bears many similarities with C. Gratuitous departures are avoided. This is in keeping with the overall philosophy of simple, intuitive ergonomics.

Certain features not present in C have been important to the development and adoption of the language, sometimes in disproportion to their complexity or ingenuity.

3.1 Slices and Ranges

D supports C-style pointers, but their use is 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 [93]. C++ iterators, being a generalization of pointers, have inherited some of this lack of safety [73]. To address this issue, the *slice* [4] was added to D in 2003 [60] as a built-in aggregate type at the suggestion of Jan Knepper, an early D contributor.

Given a type `T`, a slice (denoted as type `T[]`) is an encapsulated pair of a pointer to `T` and the number of contiguous items of type `T` that reside in memory starting at that address. With this construct, slices offer bounds-checked random access and subslicing (reducing the extent of the slice) "for free," with no complex code instrumentation or typechecking cleverness. In contrast, C code using pointer/length or pointer/pointer pairs to represent arrays needs to use conventions to convey how pair elements are related, and rely on the programmer to combine them correctly. The SAL annotation language for C and C++ [93], used at scale in Microsoft's codebase, works around the fundamental shortcomings of using bare pointers to manipulate contiguous arrays.

Slices led subsequently to a natural generalization as *ranges* [3, 96], safe abstractions akin to encapsulated pairs of C++-style iterators. Ranges offer many of the advantages of C++ iterators (decoupling many algorithms from the data structures they operate on) and add safety features, better encapsulation, and significantly reduced syntactic overhead. Initial discussions about adding ranges to D took place in emails between D contributor Matthew Wilson and Bright in 2004. Work on supporting ranges in D and its standard library began in 2007 and continued into 2009, based on a design by Andrei Alexandrescu [3].

3.2 Compile-Time Evaluation

From the beginning, Bright took a conventional approach to evaluating expressions during compilation in D. Initial releases performed simple constant propagation (in a manner similar to C and C++) to evaluate arithmetic expressions during compilation, such as:

```
enum { a = 100 };
enum { b = a * 10 + 2 }// 1002, known during compilation
```

Beyond simple arithmetic, D's constant propagation facilities did not include other compile-time expression evaluation. Over time experience with metaprogramming suggested that much advanced metaprogramming code was using types to perform computations that can be expressed rather trivially using imperative or functional code, the only problem being that computations such as function calls, array indexing, loops, or assignments could not be performed during compilation. A long-standing example coming from C++ is the compile-time factorial function [85], which, expressed via type manipulation, is not only difficult to write and understand, but also unnecessarily inefficient in its use of recursion:

```
template Factorial(ulong n : 0) {
  enum Factorial = 1;
}
template Factorial(ulong n) {
  enum Factorial = n * Factorial!(n - 1);
```

```
}
```

In contrast, there are several simple ways to write the same function tersely and efficiently using elementary recursion or iteration, for example as shown below:

```
ulong factorial(ulong n) {
  ulong result = 1;
  foreach (i; 0 .. n) result *= i;
  return result;
}
```

Based on this observation and the increased use of template metaprogramming, in February 2007 Bright added a full-fledged interpreter to D that operates during compilation. Whenever a function call occurs in a context where a constant value is required, the function's body is interpreted (source code must be available); upon return evaluation continues with the function's result as a constant expression. Thus, in the example below, the first expression is evaluated at run time whereas the second is evaluated during compilation (in D, all `static` variables must be initialized with compile-time constants):

```
ulong x1 = factorial(10);
static ulong x2 = factorial(10);
```

The notion of compile-time evaluation is quite old; LISP code can easily manipulate programs (or fragments thereof) as data and evaluate them as desired. In the Forth programming environment [50], the compiler itself is available during compilation and can be paused and reentered with ease. Finally, the related research area of partial evaluation [49] is an active research area of program optimization focused on shifting computational burden from run time to compile time.

The initial implementation was limited in capabilities (there was no support for pointers, globals, or memory allocation), yet compile-time function evaluation (commonly dubbed CTFE) quickly became popular across the D community, which created incentive to improve it. For example, in 2009 and 2010, D contributor Don Clugston added support for safe pointer manipulation, memory allocation via `new` (which enabled the use of classes), and pointer aliasing. Currently most D code can be evaluated during compilation, except for:

- functions without an available body (although technically feasible, allowing such presents important security risks);
- reading and writing global variables; and
- unsafe code (such as forging pointers or pointer arithmetic) that may corrupt the compiler's own memory.

(Interestingly, the aforementioned built-in slices were a key enabler of safe compile-time evaluation; if only pointers were available to represent arrays, then using arrays safely during compilation would have been a much more difficult proposition because indexed access through pointers is unsafe.)

CTFE makes a large and useful subset of D readily available during compilation. Consequently, many D functions already written are usable during compilation with no change in implementation or use. This tremendously lowers the barrier of entry to advanced compile-time processing techniques because the user does not need to learn a distinct language or macro system.

3.3 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.

In 2007, introspection primitives were added to D which use the syntax

```
__traits(trait, x)
```

where *trait* is one of a predefined list of possible traits, and `x` is a type or an expression (depending on the trait queried). The construct yields a value, a type, or a built-in tuple, again depending on the trait [87]. For example the construct `__traits(isUnsigned, uint)` is a compile-time expression yielding `true`, and `__traits(getOverloads, C, "foo")` yields all overloads of method `foo` in class `C` as a built-in tuple.

The syntax intentionally lacks aesthetic appeal, instead aiming for unvarnished uniformity. This is because introspection primitives are low-level and meant to be used in higher-level introspection facilities. The name was selected for its meaning of "a distinguishing quality or characteristic" [65], an apt description of the information one typically wants to query during introspection. Over time, the standard library has offered a growing collection of introspection facilities in the module `std.traits` [58].

The most powerful but also the least structured introspection mechanism is `__traits(compiles, expr)`, where *expr* can be any grammatically correct D expression. The construct (introduced in September 2007) is an expression that returns `true` if the expression could be compiled within the current environment, and `false` otherwise (without stopping compilation on errors such as e.g. invalid operation or name lookup failure). This "speculative compilation" device is useful in generic code for querying arbitrary capabilities of unknown types, and acting accordingly. On the downside, it is also difficult to use correctly (for example a typo could lead to the wrong conclusion); wherever more specific introspection mechanisms are possible, they should be preferred.

Introspection has limited power. It can query for type capabilities, deconstruct types, enumerate names in a scope, and check whether an expression can compile, but cannot inspect function bodies. Full AST inspection and manipulation (of expressions, statements, and function bodies) would add a whole new dimension to introspection, and is occasionally requested.

3.4 Conditional Compilation

Introspection must be complemented by a means to conditionally compile code in response to introspection queries. To that end, D offers a coarse-grained mechanism with the `version` declaration, and a fine-grained mechanism with the `static if` declaration.

3.4.1 `version` *Declarations.* Version conditionals key off of 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 expresions 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 C's and C++'s "`#ifdef` hell" [37, 54, 63, 81] 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.

3.4.2 `static if` *Declarations.* For fine-grained conditional compilation, D has offered (since May 2005) a `static if` declaration of the form:

```
static if (expression) {
  ...
} else {
  ...
}
```

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`, *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) { ... }
  }
  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 can be considered a proxy for assessing generality of code.

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 D's memory allocation framework [56].

3.5 Code Generation: `mixin`

There is a certain connection between generic programming [2, 7, 10, 29] and generative programming [25]. If generic programming, the argument goes, is the pursuit of defining code that "consumes" many existing and possible data structures, generative programming focuses on the converse, i.e. emitting potentially large quantities of code from terse specifications (often in the form of domain-specific languages). Generative programming is used in automated generation of specialized, efficient code for a variety of computing environments starting from high-level programs [1, 19].

Generally, programming languages suitable for generative programming offer a combination of the following features:

- an ability to create DSLs for writing high-level specifications tailored to various problem domains;
- some facilities for manipulating DSLs and integrating them within the existing language's computational model; and
- primitives for code generation, such as syntax macros or template instantiation.

Generative approaches using C++ templates have a combinatorial flavor—they rely on combining fragments of data structures or behavior in multi-argument templates, to the effect of generating many possible structural and behavioral combinations from a small, orthogonal codebase [25], an approach known within the C++ community as policy-based design [2]. The generative effect is attained by using C++'s template instantiation engine as a macro expansion engine. Other languages use syntax macros based on text or tokens [21, 55] (e.g. C, MASM [47], ABEL-HDL [23]), or syntax trees [99] (e.g. Common Lisp [89], Racket [31], Clojure [30]). Although syntax macros are well researched, there is no established theory integrating all steps from embedded DSL definition to code generation in the host language, and they are not in the mainstream of Generative Software Development [24]. Nevertheless, the paradigm has obvious power and applications, and continues to hold the attention of researchers and practitioners.

By 2006, D had a template expansion engine similar to that of C++. The addition of `static if` offered good generic and generative capabilities. Bright considered that the work on compile-time function evaluation would greatly enhance these facilities; however, CTFE's power remained

hamstrung without a mechanism for unrestricted code generation. He saw syntax macros—as proposed for Java [8] or implemented in existing languages such as LISP [52], Scheme [35], and Dylan [9]—as an advanced facility that is difficult to master, both as an author and as a reader. There was also the "domain-specific" part, which calls for supporting foreign grammars (such as regular expressions, EBNF grammars, or embedded SQL) and integrating them into the host language's grammar. Some macro systems allow extending the syntax of the host language [107], but none provides the ability to embed a whole different grammar.

With CTFE, the entire power of D was already on offer during compilation, forming a very rich macro processing engine able to process strings, tokens, syntax trees, or virtually any abstraction the user would choose. The only missing component was the "output," i.e. the last expansion step. To that end, Bright defined the `mixin` construct (which may be an expression, statement, or declaration) and released it in February 2006. The construct `mixin(`*string*`)` simply takes a string known during compilation and compiles it as D code. As such, the statement:

```
mixin("writeln(42);");
```

first converts the string `"writeln(42);"` to the corresponding D code `writeln(42);` as if it had been written in the program text, and then proceeds to compile the code to ultimately result in printing `42` at the terminal during execution. (One important giveaway of the two distinct processing steps is the presence of *two* semicolons in the code, both necessary. One is for the `mixin` statement itself, and the other is part of the generated statement.)

`mixin` would be just an arcane way to write code, were it not for CTFE. Key to the power of `mixin` is the possibility that the string being mixed in is the result of a function that is evaluated during compilation. `mixin` completes a compelling troika:

- functions of arbitrary complexity can be run during compilation to process DSLs presented as ordinary string literals;
- introspection provides integration with the surrounding D code;
- complex D code can be generated as a string and readily emitted by using `mixin`.

The approach of generating strings lacks syntactic safety [107] because the generated code may be syntactically incorrect. This indeed makes `mixin`-based code difficult to debug; a common approach is to use a built-in pragma that prints the string just before being compiled. This drawback is compensated handily in practice by its conceptual simplicity and endless flexibility. Expertise in manipulating abstract syntax trees is not common, whereas outputting formatted strings is a skill any programmer is expected to master.

Applications combining introspection, CTFE, and `mixin` were soon to follow. Dmitry Olshansky's library FReD (Fast Regular Expressions for D) [70], first launched in 2011, is a regular expression engine able to accept a regular expression string during compilation and build D code specialized for that regular expression. That approach is significantly faster than building an automaton, and closer in flavor and speed to Google's V8 engine [11], which uses just-in-time compilation to generate custom code from the regular expression. FReD also allows run-time strings in conjunction with a more conventional interpreter, with a large proportion of code being shared across these distinct engines. Somewhat confusingly (but serving as an important validator of the entire approach), the engine can not only build a specialized recognizer during compilation, but can also use it during compilation. FReD was accepted as D's standard library package `std.regex` in October 2011 [57].

Another significant application is Pegged [83], a Parsing Expression Grammar (PEG) [39] engine. The engine takes a string representing a PEG grammar specification, and generates during compilation a custom parser for that language. As such, Pegged is akin to a lex/yacc toolchain that is perfectly integrated and requires no special tools or additional processing steps. Pegged opens

the door for arbitrary user-defined embedded DSLs. Like FReD, Pegged can not only build, but also run, a custom parser during compilation. The PEG grammar itself can be expressed in Pegged, which makes meta-circularity possible. Finally, an experimental D grammar expressed in Pegged generates a D parser from only 1000 lines of specification.

3.6 Attributes

Attributes (also known as annotations) are language- or user-defined tags attached to declarations. Many modern languages (e.g. Java [71], Rust [33], Swift [46]) support annotations to various degrees, most often intended as a simple mechanism to add user-defined metadata to data and functions. Bright added user-defined attributes to D in November, 2012 [17].

The interesting aspect of D attributes is their unique power and integration with the rest of the language. In D, a user-defined attribute is a value of *any* type, as long as it is known during compilation. Subsequently, attributes are accessible through simple introspection. This clever design eschews the necessity of defining a small DSL dedicated to attributes, (as Java does), an approach marred by disadvantages in learning and usage [18, 106].

D's design leverages CTFE and introspection, which means that a single language is used for writing conventional code, writing generative code, defining annotations, and acting upon annotations. There is no need for external tools or additional processing steps; attribute processing occurs organically during compilation.

To add one or more attributes to a declaration, the programmer uses the following syntax [105]:

```
@("Smith", 1.2, "2018-08-31") int myFunction(int x);
```

The declaration of `myFunction` is annotated with three attributes: the first with the value "`Smith`" of type `string`, the second with the value `1.2` of type `double`, and the third the string "`2018-08-31`". The attributes may be retrieved by using `__traits(getAttributes, fun)`, which yields a built-in tuple `("Smith", 1.2, "2018-08-31")` during compilation. Arbitrary further processing is possible by using regular D code.

Attributes of built-in types are of limited appeal and are liable to misinterpretation and "stringly typed" attribute manipulation. Imparting more structure to the example above is immediate:

```
struct Author { string name; }
struct Version { double value; }
struct Date { ... }
@(Author("Smith"), Version(1.2), Date(2018, 8, 31)) int fun(int x);
```

Attributes may be arbitrarily elaborate as long as the code creating them can be evaluated during compilation.

3.7 Relaxed Functional Purity

As a fundamentally imperative language, D has made remarkable inroads into enabling coexistence of imperative and functional code in the same program. Functional programming's basic tenets—data immutability and functional purity—pose fundamental difficulties in interfacing with imperative code.

D added two qualifiers for representing functional data: `immutable`, which expresses transitive immutability of data, and the weaker `const`, which denotes an unmodifiable view of data that may be mutable or not [88]. It also added a `pure` attribute which, when attached to a function, prompts the compiler to enforce functional purity. Under the initial implementation, this also required functions be referentially transparent [110].

Surprisingly, `pure` was not adopted enthusiastically by the community. Two reasons emerged:

- *Coding style:* Programmers used to an imperative style had to use a completely different style (e.g. recursion instead of iteration) to implement seemingly equivalent algorithms;
- *Rigidity:* A host of functions that ostensibly had no side effects did not pass the purity test.

Consider for example a simple summation function, written in a functional style:

```
pure int sum(int[] values) {
  if (values.length == 0) return 0;
  return values[0] + sum(values[1 .. $]);
}
```

The same function written using iteration would not be referentially transparent:

```
int sum(int[] values) {
  int result = 0;
  foreach (v; values) result += v;
  return result;
}
```

Unlike other languages, the D language does allow the imperative function to pass the purity test [67] because all mutation is *transient and local to the function* and is not observable outside. As such, `sum` is observably pure regardless of whether it uses imperative code in its implementation.

D's rules for observable purity [101] work as follows:

- *no escape into impure code:* a `pure` function can call only `pure` functions;
- *result depends only on parameters:* a `pure` function cannot read process-global or thread-local variables;
- *no side effects:* a `pure` function cannot write process-global or thread-local variables.

The notion of observable purity went a long way toward harmonizing functional and imperative style. Using local imperative code, D programmers could define and use pure functions with ease using a mix of imperative and functional artifacts, often with gains in efficiency as well (note, for example, that the recursive version above requires linear stack space because the recursive call is not a tail call). After the relaxation of purity in D, the use of functional programming came to be viewed as idiomatic [34].

3.8 Uniform Function Call Syntax

A popular style of program composition is to encapsulate processing steps as functions with one or more inputs and one output, and arrange code such that the output of one function is the input of the next. This way, complex processing sequences can be assembled tersely and understandably. Such programming styles, colloquially known as pipelining/streaming code or functional pipelines [41], first emerged as a natural idiom in functional languages [36]. Other languages such as C#/LINQ [64], C++ [68], and Java [94] followed with language-specific flavors.

In pipelines, the traditional function call notation—function name to the left followed by parameters—is at a disadvantage as it requires parentheses and must be read "inside out". For example `uniq(sort(filter(input)))` first calls `filter`, then `sort`, and finally `uniq`. Ideally, the reading order would be the same as the order in which pipeline components are processed. Additional parameters create additional syntactic noise. To work around this awkwardness, object-oriented languages make use of the "method chaining" idiom [111] whereby a method returns its object, so multiple methods can be invoked on the result object by using dot notation as in `object.method1().method2()`.

In languages that allow both methods and top-level functions (such as C++ or D itself), this dynamic creates incentive for the programmer to prefer creating methods over top-level functions; however, encapsulation and modularity considerations give strong reasons to the contrary [66].

D's solution to this tension is to allow an alternative syntax for invoking object methods. In the source code `a.b(c, d)`, the symbol `b` is first looked up inside `a`'s type. If the name is found (whether it is a method or not), the lookup ends and conventional typechecking of the call continues. If the name is not found, the call is rewritten as `b(a, c, d)` and the name lookup proceeds accordingly [20, Ch. 61]. The feature is called "Uniform Function Call Syntax" (UFCS) [16, 112] because it reduces the unnecessary syntactic distinctions between invoking methods and free functions.

The use of UFCS consistently improves D code, which often relies heavily on generic algorithms and generic ranges. A processing chain that reads, sorts, and prints text lines would be written as follows using conventional function call syntax:

```
copy(
  sort(
    array(
      map!(a => a.idup)(stdin.byLine(KeepTerminator.yes)))),
  stdout.lockingTextWriter());
```

and in the following form using UFCS:

```
stdin.byLine(KeepTerminator.yes)
  .map!(a => a.idup)
  .array
  .sort
  .copy(stdout.lockingTextWriter);
```

D does not require empty parentheses '`()`' in a parameterless function invocation, a feature Bright implemented before the release of the first prototype version. Its original purpose was to ease refactoring when replacing object fields with methods, but its biggest benefit proved to be that it further improves the effect of UFCS.

Constructs similar to UFCS later appeared in other languages such as Nim [112], C++[92], and Rust [32].

3.9 Safe Conversion Between Different Sized Integers

The C language is cavalier about potentially lossy integral conversions (such as from a 64-bit integer to a 32-bit integer), an attitude partially inherited by C++. Such flexibility makes it easy to write code that does a lot of integer and bit manipulation, but also opens the door to a variety of programmer errors that can be subtle and difficult to detect. Some programmers believe that unintended loss of information is undesirable and should not be allowed. To avoid this, languages such as Pascal, Java, and C# [44] require an explicit cast to perform a potentially lossy conversion. One disadvantage is that the code for manipulating integers of various widths—common in systems work—becomes bureaucratically verbose. The deeper problem is that the explicit cast itself is a blunt instrument that shifts the responsibility for correctness from the type system to the programmer. This may lead to hiding errors instead of exposing them, particularly during code maintenance [82]. In generic code, casts from generic (parameterized) types to non-generic types are even more dangerous because new instantiations can transform a benign cast into a dangerous one.

For D, Bright aimed at a middle ground that would preserve the advantages of both implicit conversion and explicit casts. The solution borrows an idea from code optimizers: Value Range

Propagation (VRP) [75], which can be thought of as a refined form of constant propagation. Each integral (sub)expression carries with it its statically-known possible range of values, from a minimum to a maximum. (Expressions with an unknown range would start with the entire range allowed by their type, e.g. -2^{63} through $2^{63} - 1$ for a 64-bit integer). Then, dedicated algorithms propagate the range through the arithmetic operators (negation, addition, subtraction, multiplication, division, modulus, and exponentiation) bitwise operators (NOT, AND, OR, XOR, shift), and the conditional operator '`?:`'.

Extant VRP applications [69, 75, 90] focus on range propagation through loops and multiple instructions, while using simple conservative algorithms for arithmetic and logic operators. On the contrary, D's VRP is flow- and context-insensitive but uses algorithms that find the tightest ranges for all operators. Long-time D contributor Timon Gehr's algorithms for VRP through operators, implemented at the end of 2017 [27], are nontrivial and work in the presence of two's complement wraparound and mixed-sign expressions.

Consider the following example (in D, `ubyte` denotes an 8-bit unsigned number and `int` denotes a 32-bit signed integer):

```
ubyte someFunction(int x) {
  return (1 + (x & 7)) << 2;
}
```

Even though `x` has unknown (i.e. maximum possible) range, and the result of the expression is of type `int`, no cast is required to coerce the expression's result to the function return type of `ubyte` as the entire expression has a statically-inferred range of [4, 32].

Although a flow-sensitive and context-sensitive solution would be more precise, Bright chose to deliberately limit VRP to single expressions; range information is not carried from one statement to the next. Thus, the following code will not compile although it is semantically identical to the code above:

```
ubyte someFunction(int x) {
  int tmp = x & 7;
  return (1 + tmp) << 2; // Error: cannot implicitly convert expression
                         // 1 + tmp << 2 of type int to ubyte
}
```

A possible improvement would be to allow sets of possible ranges to each expression, as opposed to a single range. In that setup, the expression `x ? (y & 7) : 10` would have possible values in the union of [0, 7] and [10, 10], similar to Patterson's approach [75].

There are good reasons to keep VRP limited. Expression-level, single-range VRP is easy to document and understand, does not slow down compilation significantly, and does not lead to puzzling compile-time errors. The maintainers of D's standard library (i.e. a large preexisting code base) have experimentally determined that VRP allows eliminating most casts from code, so further refinements would have high cost and marginal benefit.

3.10 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 motivations for Ddoc, a documentation generation framework akin to Javadoc [53], which drastically improved the matter by integrating documentation into the D source

code itself. Ddoc was added in September 2005 [104] 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++ [98], 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 [28] have been built on top of it in a compatible manner, the most popular being Rejected Software's DDOX [86], and Adam D. Ruppe's adrdox [79].

3.11 Unit Testing Support

D has built-in support for unit testing. 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 unittests 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
  }
  ...
}
```

Unit tests defined inside generic classes will be instantiated and executed for each instantiation. This is onerous with library-based approaches:

```
class Generic(T) {
  void method() { ... }
  unittest {
    // One definition 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 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, with tremendous cultural

consequences. A community environment in which doing the right thing is this easy fostered unit testing as common courtesy, thus ensuring it by social means instead of tooling or language rules.

4 ORGANIZATION

In the early years, D was developed by Digital Mars, Bright's one-person company. Initially, D had no funding and no budget, and Bright paid the occasional expenses as they arose. Jan Knepper donated server space for the D forums, and Brad Roberts donated server space for Bugzilla. There was no staff to pay and equipment costs were minimal. No paid advertising was used. Excluding the licensing deal with Symantec, Bright's out-of-pocket expenses were likely less than $10,000 over the first 15 years of the language's life.

Over time, more people volunteered to help [40], working on tasks that interested them or were of use in their own projects. Bright informally led the ad-hoc team, by virtue of being the original designer and implementer. There was no formal organization. Communication among nearly all the members was enabled by the Internet, mostly through the D forums [97], sometimes via email. The first time all members met in person was at the first D Language Conference in 2007. Regular annual international D conferences started in 2013.

Since everyone was a volunteer, this posed unique management challenges. The most obvious was that Bright, despite being the project leader, had no authority to order anyone to do anything. People worked on what they wanted to, when they wanted to, and appeared and disappeared as they chose. Often, volunteers would ask Bright what they should work on. Bright would provide a list, and the volunteers would then choose to work on something else. This often meant that Bright did the work nobody wanted to do, but which had to be done.

The goal was always to create a first class programming language—there was never any other agenda. Since there were no sponsors, the language was free to evolve in the direction desired by the people working on it.

Testing of the compiler was carried out by Bright. Many people contributed to the test suite, in particular Thomas Kuehne.

There were no marketing, community relations, or training personnel, until 2016, when the D Language Foundation was created.

The difficulties of developing the first version from front to back were mitigated by licensing Symantec's C++ compiler, offering a professional code optimizer, code generator, linker, and related tools. The immediate necessities implementation-wise were the front end for D and the initial runtime library. This meant the D compiler would be implemented in C++, but that did not affect the design of the language.

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. In January 2015 [26], the D Languagage Foundation began publishing Vision documents plotting semester-at-a-time the broad direction of language development would start being published .

Bright's estimate of the time he invested in D up to its first major release is about 7 man-years. The total investment by others in the community remains unknown.

4.1 The D Language Foundation

D's growth created an increasing need for a formal organization to hold the copyright of the compiler and online properties, to oversee conferences, promote the D language, and as a rallying point for the entire community.

The D Language Foundation was incorporated in the state of Washington, USA, on October 15, 2015 and received approval as a non-profit 501(c)(3) educational charity on 29 August, 2016. The officers of the Foundation are Walter Bright (President), Andrei Alexandrescu (Vice President

and Treasurer), and Ali Çehreli (Secretary). The Foundation is financed by corporate and private donations. Its bylaws prevent granting salaries to officers.

The Foundation is carrying out a number of important community initiatives:

- *Scholarships:* A graduate student scholarship program was initiated in September 2016. The program offers financial assistance to Computer Science graduate students that work on research relevant to the D language. Five students from University "Politehnica" Bucharest, Romania and one from Technische Universität München, Germany, have received scholarships since the program's inception. A collaboration with Universidade Federal de Minas Gerais, in Brazil, is in the works.
- *Software Releases:* Releases of the D language reference compiler, documentation, and toolchain are coordinated by the Foundation.
- *Blog and Community Relations:* Michael Parker is the Foundation's community liaison, social media editor, and conference organizer.
- *Hackathons and Coding Contests:* Drawing inspiration from its participation in Google Summer of Code [84], the Foundation organizes coding projects, often in collaboration with companies interested in D technology [95].
- *A Formal Process for Language Changes:* The "D Improvement Proposal" (DIP) [72] is a systematic approach to proposing and effecting language changes, currently under evolution.

4.2 Online Community

The first D forum post was published on August 12, 2001 [38]. Jan Knepper was hosting digitalmars.com on his servers, and initiated hosting of the NNTP newsgroup software for the forums. (The choice of NNTP protocol had surprising advantages. NNTP requires few resources, and its minimalism and out-of-fashion status kept spam away. In 2011, Vladimir Panteleev would write DFeed [100], a web forum interface to the NNTP server, in the D language. DFeed would receive accolades for its speed [43].) Aside from the general forum, there are additional forums dedicated to announcements, learning, and specialty niches.

Although the forums are not official, they are routinely used as a sounding board for ideas and staging area for collaborative work.

5 DISTRIBUTION

The language definition and tool binaries have been free for unrestricted use since their inception. Documentation (language and standard library), compiler and tools binaries, and standard library sources have always been freely available online. The standard library was initially marked as Public Domain, but as some countries do not recognize Public Domain, the licensing was switched to the Boost License. The Boost License was already familiar to C++ users, and was as close as could be found to Public Domain.

The D front end was open-sourced in April 2002, initially dual licensed under the GNU General Public License and the Artistic License. It, too, was eventually moved to the Boost License. The optimizer and back end were licensed from Symantec and could not be relicensed as Boost until Michael Spertus, Fellow Engineer at Symantec, facilitated a code donation from Symantec to the Open Source community in April 2017 [114] . With that, the tool chain was fully Boost Licensed.

The code was initially distributed via download links from the Digital Mars website digitalmars.com. Currently the D Language Foundation is responsible for the free distribution of source and binaries using D's official site dlang.org.

6 MISTAKES

Mistakes are inevitable when designing a general-purpose programming language from scratch. The challenge is finding means to overcome them and adapt to new developments.

The first problem was the failure to use a version control system from the very beginning, a mistake for which there is no excuse. 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 history of the early development of the D compiler has been lost.

Another mistake was the failure to recognize that the world had changed with respect to the value of Open Source. Bright initially tried to stick with the old closed source model, which had been successful for years in the software world at large. It became increasingly clear that D was going to grow and succeed only if it was Open Source. First the library was open sourced. Later, the front end source was released, prompting the initiation of what became the GDC compiler. Eventually, everything was open sourced, which was an unmitigated success.

Used to running his own show, Bright had a great deal of difficulty delegating responsibility for various aspects of development. The project grew too big for one person, and parts of it became neglected, particularly the Phobos runtime library. This resulted in an independent D runtime library, Tango, being developed by some members of the D community (managed by Kris Bell). The Phobos/Tango dichotomy split the D community right down the middle. Attempts to merge Tango back into Phobos failed due to Phobos having a more permissive license (Boost) than Tango (BSD), and the unwillingness of some of the Tango developers to relicense their code. Tango did not survive the transition to D 2.0, and many members of the community were lost as a result.

Managing volunteers is especially difficult, a lesson learned the hard way through trial and error. Particularly difficult was learning to say "no" to numerous feature requests without driving advocates out of the community. It hasn't gotten any easier over the years. As in many other distributed open-source projects with no funding, a variety of aspects of teamwork are difficult: exchanging negative feedback, getting unglamorous work assigned and done, ensuring timely response to requests.

A new programming language design inevitably has a lot of cross-feature interactions that are unanticipated. Even if technical hurdles are resolved, the true value of a feature in context can often be determined only by experience in the field. Pushing the envelope is almost inevitably accompanied by failed experiments. The designer has to accept a certain amount of failure, and D is no exception. Here are some of our failed ideas:

- Having `bit` as a basic type. It was there from the beginning, but proved to be unworkable, as it required the existence of two types of pointers in the type system, and was removed in February, 2006.
- Having a `typedef` that introduced a new type. It was removed because a simple alias (similar to C's `typedef`) was much more popular. (Ironically, the D `typedef` resurfaced years later as a library facility, where it arguably belongs.)
- Having complex and imaginary numbers built in turned out to be pointless, as they were easily implemented in the standard library. Removal is still ongoing.
- The Numerical C Extensions Group (NCEG) comparison operators taking NaN into account [22] were so unusual that few programmers used them. They were removed after a long deprecation period.
- Safety, immutability and purity should have been the default to more pervasively encourage the use of functional programming.
- Allowing destructors to throw exceptions increases the probability of the "double fault exception", which is difficult to reason about. C++11 has taken steps to discourage their use.

Fortunately, the D community has been very supportive of correcting such mistakes, and were willing to abandon their reliance on them. The same goes for mistakes in the Phobos runtime library, which amply proved the utility of D's deprecation feature.

7 OUTLOOK

Eleven years after its 1.0 release, D continues to be at the forefront of exciting developments in programming languages. Established D features constantly trickle into other programming languages. The D community grows steadily.

D is not without its controversies, though. Many express dissatisfaction with both the pace of change and desire for more capability. Resolving these contradictory pressures is an ongoing problem. D, being a relatively small operation, continues to work hard to compete with very well-financed alternatives.

Major focal points of D moving forward are:

- Establishing a 100% guarantee of memory safety. Mechanically guaranteed memory safety is expected by the D team to be a fundamental requirement for programming languages going forward.
- Produce an automatic, reference-counting memory management scheme, which is a requirement for many no-pause and restricted memory applications.
- Create a formal specification of the language.

The ongoing success of D proves that a small group of determined developers can indeed establish a new programming language with compelling capabilities. This is very inspiring. D's small organization has the advantage of having little bureaucracy to impede the developers.

8 TIMELINE

The following list summarizes the more important events in the evolution of the D programming language to date.

- 1999, November: the initial idea and decisions to move forward with the design and implementation of D.
- 2000, April 12: Digital Mars secures a license of Symantec C++ from Symantec
- 2001, January 31: first documentation of initial idea by Walter Bright
- 2001, August 16: first Slashdot appearance
- 2001, August 16: preliminary specifications by Walter Bright
- 2001, December 8: first public release of prototype
- 2002, February: inline assembler added, converted from Digital Mars C++ compiler's;
- 2002, February, Dr. Dobb's "The D Programming Language" by Walter Bright
- 2002, March: began opening the source code, starting with the lexer and parser
- 2002, April: began allowing user contributions to the Phobos library
- 2002, May: added `public`/`protected`/`private`
- 2002, May: David Friedman began creation of another D compiler by adapting the D front end sources for use with gcc, calling it the BrightD Compiler project.
- 2002, Aug: added operator overloading
- 2002, September: type inference for declarations, Resource Acquisition Is Initialization (RAII) for classes
- 2003, February: function literals, nested functions and static closures
- 2003, May: first Linux version
- 2003, June: `private import` added, so public declarations in a file imported by an imported file are not accessible

- 2003, August: `dchar` for UTF-32 characters, embedded '_' in numeric literals (from Ada)
- 2003, September: `foreach`, `static assert`, user defined properties can be implemented as functions as well as fields
- 2003, Fall: Windows Developer Network compares D with C, C++, C#, and Java
- 2004, January: `typeof`, `pragma`, template alias parameters
- 2004, February: first release of Friedman's compiler, called GDMD (and later DGCC)
- 2004, March: Dr. Dobb's "Collection Enumeration: Loops, Iterators & Nested Functions" by Walter Bright and Matthew Wilson
- 2004, March 17: presentation from Bright on The D Programming Language at Software Development and Expo West 2004
- 2004, April: translated the game Empire to D
- 2004, December: translated DScript (a Javascript engine) to D
- 2004, May: added `mixin template`
- 2005, January, Dr. Dobb's "`printf` Revisited" by Walter Bright
- 2005, March: named character entities, `__FILE__` and `__LINE__` support
- 2005, May: `static if`
- 2005, June: typesafe variadic functions using runtime type information, `void` initializers which leave a declaration uninitialized
- 2005: Ddoc
- 2006, February: Bugzilla/Mail/news for D went live
- 2006, February: removed `bit`, added `bool`, added scope guards which add cleanup code depending on how a block scope is exited
- 2006, June: improved lambda syntax, making it much more usable
- 2006, July: greatly improved flexibility of `import` declarations
- 2006, August: `lazy` parameter storage class
- 2006, September: array literals
- 2006, October: `foreach_reverse`, use of delegates for `foreach`
- 2006, November: templates could accept alias parameters for locals and nested functions, added variadic template parameters, added extensive support for tuples (such as slicing and indexing)
- 2007, January 01: Slashdot announced D version 1.0
- 2007, January 23: first full release 1.001
- 2007, February 5: `mixin` expressions which insert text from a string to be compiled inline, and `import` expressions which insert text from a file to be compiled inline.
- 2007, February 7: final D 1.0 specification
- 2007, February 15: compile time execution of functions (CTFE)
- 2007, June 17: Added `const`, `immutable`, and `final` indicating a class member function that cannot be overridden.
- 2007, August 23–25: first D conference (organized by Brad Roberts)
- 2007, September 5: `__traits(compiles, ...)`
- 2007, March 6: `struct` destructors
- 2008, June 17: constraints to templates.
- 2008, September 2: `struct` constructors
- 2008, December 11: `pure` and `nothrow` attributes, `shared` qualifier
- 2009, February 14: Mac OSX support
- 2009, March 31: `alias this` enabling structs to be subtyped
- 2009, May 11: global variables now default to thread-local storage
- 2009, September 2: CTFE significantly more powerful

- 2009, October 5: Phobos switched to Boost license
- 2009, December 30: `auto ref` functions and parameters
- 2011, February 17: 64 bit support for Linux
- 2011, May 12: FreeBSD support
- 2011, July 10: `@safe` attribute, automatic inference for `@safe`, `pure`, `nothrow`, support heap allocation in CTFE
- 2012, February 14: new '=>' lambda syntax
- 2012, April 12: Uniform Function Call Syntax (UFCS)
- 2013, January 1: Windows 64 bit support, user-defined attributes
- 2014, August 18: `@nogc` attribute, `extern(C++, namespace)`
- 2015, March 24: sealed references where when a parameter passed by reference to a function is also returned by reference from that function, it must be so annotated
- 2015, November 3: DMD front end ported to D, BetterC support (where D programs can be built without requiring the D runtime library), basic support for Objective-C types so D code can interface with Objective-C
- 2017, September 1: `static foreach`, improve BetterC
- 2018, January 1: RAII and `try`/`finally` for BetterC mode
- 2018, March 1: `@nogc` exception throwing, where exceptions can be reference counted rather than relying on the GC
- 2018, July 1: expression-based contract syntax; C++ compatible constructors and destructors

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

[1] Baris Aktemur, Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. 2013. Shonan challenge for generative programming: short position paper. In *Proceedings of the ACM SIGPLAN 2013 workshop on Partial evaluation and program manipulation*. ACM, 147–154.

[2] Andrei Alexandrescu. 2001. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley.

[3] Andrei Alexandrescu. 2009. On Iteration. (Nov. 2009). http://www.informit.com/articles/article.aspx?p=1407357

[4] Andrei Alexandrescu. 2010. *The D programming language*. Addison-Wesley Professional.

[5] Anonymous. 1983. *The Programming language Ada: reference manual*. Vol. 155. ix + 330 pages. American National Standards Institute, Inc. ANSI/MIL-STD-1815 A-1983, approved 17 February 1983.

[6] Ken Arnold, James Gosling, and David Holmes. 2005. *The Java programming language*. Addison Wesley Professional.

[7] Matthew H Austern. 1999. *Generic programming and the STL: using and extending the C++ Standard Template Library*. Vol. 7. Addison-Wesley Reading.

[8] Jonthan Bachrach and Keith Playford. 2001. The Java syntactic extender (JSE). In *ACM SIGPLAN Notices*, Vol. 36. ACM, 31–42.

[9] Jonathan Bachrach, Keith Playford, and C Street. 1999. D-expressions: Lisp power, Dylan style. *Style DeKalb IL* (1999).

[10] Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. 1998. Generic programming. In *International School on Advanced Functional Programming*. Springer, 28–115.

[11] V8 Javascript Engine Official Blog. 2017. Speeding up V8 Regular Expressions. (10 Jan. 2017). Retrieved Aug 31, 2018 from https://v8project.blogspot.com/2017/01/speeding-up-v8-regular-expressions.html

[12] Walter Bright. 1979. Empire in Fortran-10 for the PDP-10. Retrieved Aug 31, 2018 from https://github.com/DigitalMars/Empire-for-PDP-10

[13] Walter Bright. 2009. Designing Safe Software Systems Part 2. (Nov. 2009). https://digitalmars.com/articles/b40.html

[14] Walter Bright. 2009. Designing Safe Software Systems Part 2. (Nov. 2009). https://digitalmars.com/articles/b44.html

[15] Walter Bright. 2009. Safe Systems from Unreliable Parts. (Oct. 2009). https://digitalmars.com/articles/b39.html

[16] Walter Bright. 2012. Uniform Function Call Syntax. (March 2012). http://www.drdobbs.com/cpp/uniform-function-call-syntax/232700394

[17] Walter Bright. 2012. User Defined Attributes. Retrieved Nov 30, 2018 from https://forum.dlang.org/post/k7afq6$2832$1@digitalmars.com

[18] Yegor Bugayenko. 2016. Java Annotations Are a Big Mistake. (12 April 2016). Retrieved Aug 31, 2018 from https://www.yegor256.com/2016/04/12/java-annotations-are-evil.html

[19] Damien Cassou, Benjamin Bertran, Nicolas Loriant, and Charles Consel. 2009. A generative programming approach to developing pervasive computing systems. In *ACM Sigplan Notices*, Vol. 45. ACM, 137–146.

[20] Ali Çehreli. 2015. *Programming in D: Tutorial and Reference*. Ali Çehreli.

[21] Thomas E Cheatham Jr. 1966. The introduction of definitional facilities into higher level programming languages. In *Proceedings of the November 7-10, 1966, fall joint computer conference*. ACM, 623–637.

[22] Don Clugston. [n. d.]. Real Close to the Machine: Floating Point in D. Retrieved Aug 31, 2018 from https://dlang.org/articles/d-floating-point.html

[23] Lattice Semiconductor Corporation. [n. d.]. ABEL-HDL Reference Manual. Retrieved Nov 29, 2018 from https://www.latticesemi.com/-/media/LatticeSemi/Documents/UserManuals/1D/ABEL-HDLReferenceManual.ashx?document_id=589

[24] Krzysztof Czarnecki. 2005. Overview of generative software development. In *Unconventional Programming Paradigms*. Springer, 326–341.

[25] Krzysztof Czarnecki, Kasper Østerbye, and Markus Völter. 2002. Generative programming. In *European Conference on Object-Oriented Programming*. Springer, 15–29.

[26] D Language Foundation 2015. D Vision Document for H1 2015. Retrieved Aug 31, 2018 from https://wiki.dlang.org/Vision/2015H1

[27] D Language on Github 2017. Value Range Propagation Algorithms. Retrieved Aug 31, 2018 from https://github.com/dlang/dmd/pull/7355

[28] D Language Wiki 2018. Documentation Generators. Retrieved Aug 31, 2018 from https://wiki.dlang.org/Documentation_Generators

[29] James C Dehnert and Alexander Stepanov. 2000. Fundamentals of generic programming. In *Generic Programming*. Springer, 1–11.

[30] The Clojure Project Developers. [n. d.]. The Clojure Language Reference. Retrieved Nov 29, 2018 from https://clojure.org/reference/macros

[31] The Racket Project Developers. [n. d.]. The Racket Documentation. Retrieved Nov 29, 2018 from https://docs.racket-lang.org/reference/Macros.html

[32] The Rust Project Developers. [n. d.]. The Rust Programming Language. Retrieved Aug 31, 2018 from https://doc.rust-lang.org/book/2018-edition/ch19-03-advanced-traits.html

[33] The Rust Project Developers. [n. d.]. The Rust Reference. Retrieved Nov 29, 2018 from https://doc.rust-lang.org/reference/attributes.html

[34] DLang Tour [n. d.]. Functional Programming. Retrieved Nov 30, 2018 from https://tour.dlang.org/tour/en/gems/functional-programming

[35] R Kent Dybvig, Robert Hieb, and Carl Bruggeman. 1993. Syntactic abstraction in Scheme. *Lisp and symbolic computation* 5, 4 (1993), 295–326.

[36] Richard Fateman. 2003. Compiling functional pipe/stream abstractions into conventional programs: Software Pipelines. *University of California, Berkeley* (2003).

[37] 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.

[38] First D forum post 2001. Digital Mars C Newsgroups! Retrieved Aug 31, 2018 from https://digitalmars.com/d/archives//index2001.html

[39] Bryan Ford. 2004. Parsing expression grammars: a recognition-based syntactic foundation. In *ACM SIGPLAN Notices*, Vol. 39. ACM, 111–122.

[40] D Language Foundation. [n. d.]. Contributors. Retrieved Nov 29, 2018 from https://dlang.org/foundation/contributors

[41] Martin Fowler. 2015. Collection Pipeline. Retrieved Aug 31, 2018 from https://martinfowler.com/articles/collection-pipeline/

[42] Numerical C Extensions Group/X3J11.1. 1993. Floating-Point C Extensions. (Jan. 1993).

[43] Hacker News 2015. DFeed. Retrieved Aug 31, 2018 from https://news.ycombinator.com/item?id=9990763

[44] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. 2003. *C# language specification.* Addison-Wesley Longman Publishing Co., Inc.

[45] CA Hoare. 1973. *Hints on programming language design.* Technical Report. STANFORD UNIV CA DEPT OF COMPUTER SCIENCE.

[46] Apple Inc. 2018. The Swift Programming Language. Retrieved Nov 29, 2018 from https://docs.swift.org/swift-book/ReferenceManual/Attributes.html

[47] Kip R. Irvine. [n. d.]. Assembly Language for Intel-Based Computers. Retrieved Nov 29, 2018 from https://www.csie.ntu.edu.tw/~acpang/course/asm_2004/slides/chapt_10_PartIIbw.pdf

[48] Jeff Friesen 2014. Java SE 8's New Compact Profiles and Integer APIs. Retrieved Aug 31, 2018 from https://informit.com/articles/article.aspx?p=2216988

[49] Neil D Jones, Carsten K Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation.* Peter Sestoft.

[50] Peter J Knaggs. 1993. *Practical and Theoretical Aspects of Forth Software Development.* Ph.D. Dissertation. University of Teesside.

[51] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. 1994. Partial Dead Code Elimination. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94).* ACM, New York, NY, USA, 147–158. https://doi.org/10.1145/178243.178256

[52] Eugene Kohlbecker, Daniel P Friedman, Matthias Felleisen, and Bruce Duba. 1986. Hygienic macro expansion. In *Proceedings of the 1986 ACM conference on LISP and functional programming.* ACM, 151–161.

[53] 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

[54] 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.

[55] Burt M Leavenworth. 1966. Syntax macros and extended translation. *Commun. ACM* 9, 11 (1966), 790–793.

[56] The D Language Standard Library. [n. d.]. `std.experimental.allocator`. Retrieved Aug 31, 2018 from https://dlang.org/phobos/std_experimental_allocator.html

[57] The D Language Standard Library. [n. d.]. `std.regex`. Retrieved Aug 31, 2018 from https://dlang.org/phobos/std_regex.html

[58] The D Language Standard Library. [n. d.]. `std.traits`. Retrieved Aug 31, 2018 from https://dlang.org/phobos/std_traits.html

[59] Digital Mars. [n. d.]. D Change Log to Nov 7 2005. Retrieved Nov 29, 2018 from https://digitalmars.com/d/1.0/changelog1.html#new000

[60] Digital Mars. [n. d.]. D Change Log to Nov 7 2005. Retrieved Nov 29, 2018 from https://digitalmars.com/d/1.0/changelog1.html#new073

[61] Digital Mars. [n. d.]. Digital Mars C, C++, and D Compilers. Retrieved Nov 29, 2018 from https://www.digitalmars.com/

[62] Lockheed Martin. 2005. Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program.

[63] 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.

[64] Erik Meijer, Brian Beckman, and Gavin Bierman. 2006. Linq: reconciling object, relations and xml in the. net framework. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data.* ACM, 706–706.

[65] Merriam-Webster.com. 2018. "trait". Retrieved Nov 29, 2018 from https://www.merriam-webster.com/help/citing-the-dictionary

[66] Scott Meyers. 2000. How Non-Member Functions Improve Encapsulation. *C/C++ Users Journal* 18, 2 (2000), 44–52.

[67] David Nadlinger. 2012. Purity in D. (27 May 2012). Retrieved Aug 31, 2018 from http://klickverbot.at/blog/2012/05/purity-in-d/

[68] Eric Niebler, Sean Parent, and Andrew Sutton. 2014. Ranges for the Standard Library, Revision 1. Retrieved Aug 31, 2018 from https://ericniebler.github.io/std/wg21/D4128.html

[69] Diego Novillo. 2005. A propagation engine for GCC. In *Proceedings of the 2005 GCC Developers Summit.* 175–185.

[70] Dmitry Olshansky. [n. d.]. Fast Regular Expressions for D. Retrieved Aug 31, 2018 from https://github.com/DmitryOlshansky/FReD

[71] Oracle. 2018. Java SE 10 Reference. Retrieved Nov 29, 2018 from https://docs.oracle.com/javase/10/docs/api/javax/naming/directory/Attributes.html

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

[73] 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.

[74] United States Patent and Trademark Office. [n. d.]. Digital Mars. Retrieved Nov 29, 2018 from http://tmsearch.uspto.gov/bin/showfield?f=doc&state=4805:qhzzxl.2.2

[75] Jason RC Patterson. 1995. Accurate static branch prediction by value range propagation. In *ACM SIGPLAN Notices*, Vol. 30. ACM, 67–78.

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

[77] PC-Week. 1988. Zortech's Compiler Sparks Interest in the C++ Language. (5 Dec. 1988).

[78] PC-Week. 1991. Symantec Buys Zortech To Vie In C++ Market. (19 Aug. 1991).

[79] Adam D. Ruppe. [n. d.]. adrdox. Retrieved Nov 29, 2018 from https://github.com/adamdruppe/adrdox

[80] TV Series. 2011–. Air Disasters. (2011–). https://imdb.com/title/tt2091498/

[81] 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.

[82] Michael Siff, Satish Chandra, Thomas Ball, Krishna Kunchithapadam, and Thomas Reps. 1999. Coping with Type Casts in C. *SIGSOFT Softw. Eng. Notes* 24, 6 (Oct. 1999), 180–198. https://doi.org/10.1145/318774.318942

[83] Philippe Sigaud. [n. d.]. A Parsing Expression Grammar (PEG) module, using the D programming language. Retrieved Aug 31, 2018 from https://github.com/PhilippeSigaud/Pegged

[84] Jefferson O. Silva, Igor Scaliante Wiese, Igor Steinmacher, and Marco Aurélio Gerosa. 2017. Students' Engagement in Open Source Projects: An Analysis of Google Summer of Code. In *SBES*. ACM, 224–233.

[85] Ádám Sipos, Zoltán Porkoláb, Norbert Pataki, and Viktória Zsók. 2007. Meta< Fun>-Towards a functional-style interface for C++ template metaprograms. In *Proceedings of 19th International Symposium of Implementation and Application of Functional Languages (IFL 2007)*. 489–502.

[86] Rejected Software. [n. d.]. ddox. Retrieved Nov 29, 2018 from https://github.com/rejectedsoftware/ddox

[87] The D Language Specification. [n. d.]. Traits. Retrieved Aug 31, 2018 from https://dlang.org/spec/traits.html

[88] The D Language Specification. [n. d.]. Type Qualifiers. Retrieved Aug 31, 2018 from https://dlang.org/spec/const3.html

[89] Guy Steele Jr. 1990. *Common Lisp the Language, 2nd Edition*. Digital Press.

[90] Mark William Stephenson. 2000. *Bitwise: Optimizing bitwidths using data-range propagation*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[91] Bjarne Stroustrup. 2013. *The C++ programming language*. Pearson Education.

[92] Bjarne Stroustrup. 2016. A bit of background for the unified call proposal. *ISO C++ Blog* (feb 2016). https://isocpp.org/blog/2016/02/a-bit-of-background-for-the-unified-call-proposal

[93] Visual Studio. [n. d.]. *Using SAL annotations to reduce C/C++ code defects*. Technical Report. Technical report, Microsoft Developer Network, 2015. 62 Bibliography.

[94] Venkat Subramanian. 2017. Function Composition and the Collection Pipeline Pattern. Retrieved Aug 31, 2018 from https://www.ibm.com/developerworks/library/j-java8idioms2/index.html

[95] Symmetry Investments and D Language Foundation 2018. Symmetry Autumn of Code. Retrieved Aug 31, 2018 from https://dlang.org/blog/symmetry-autumn-of-code

[96] H.S. Teoh. 2013. Component programming with ranges. (Aug. 2013). https://wiki.dlang.org/Component_programming_with_ranges

[97] The D Community [n. d.]. The D Forums. http://forum.dlang.org/

[98] Dimitri Van Heesch. 2008. Doxygen: Source code documentation generator tool. *URL: https://doxygen.org* (2008).

[99] Jorge Vidart. 1974. *Extensions syntaxiques dans un contexte LL (1)*. Ph.D. Dissertation. Institut National Polytechnique de Grenoble-INPG; Université Joseph-Fourier-Grenoble I.

[100] Vladimir Panteleev 2012. DFeed, an NNTP/mailing list web frontend/forum software, news aggregator and IRC bot. Retrieved Aug 31, 2018 from https://github.com/CyberShadow/DFeed

[101] Walter Bright [n. d.]. D Language Specification: Pure Functions. Retrieved Nov 30, 2018 from https://dlang.org/spec/function.html#pure-functions

[102] Walter Bright 1992. `with` Statement for C++. Retrieved Aug 31, 2018 from https://digitalmars.com/ctg/CPP-Language-Implementation.html#with

[103] Walter Bright 2000. D Language Specification: `with` Statement. Retrieved Aug 31, 2018 from https://dlang.org/spec/statement.html#with-statement

[104] Walter Bright 2005. Adding Ddoc. Retrieved Aug 31, 2018 from https://digitalmars.com/d/1.0/changelog1.html#new0132

[105] Walter Bright 2016. D Language Specification: User-Defined Attributes. Retrieved Aug 31, 2018 from https://dlang.org/spec/attribute.html#uda

[106] Adam Warski. 2017. The case against annotations. (13 Oct. 2017). Retrieved Aug 31, 2018 from https://blog.softwaremill.com/the-case-against-annotations-4b2fb170ed67
[107] Daniel Weise and Roger Crew. 1993. Programmable syntax macros. In *ACM SIGPLAN Notices*, Vol. 28. ACM, 156–165.
[108] Wikipedia. [n. d.]. C libraries. Retrieved Nov 29, 2018 from https://en.wikipedia.org/wiki/Category:C_libraries
[109] Wikipedia. [n. d.]. Datalight. Retrieved Nov 29, 2018 from https://en.wikipedia.org/wiki/Datalight
[110] Wikipedia. [n. d.]. Functional Programming. https://en.wikipedia.org/wiki/Functional_programming#Referential_transparency
[111] Wikipedia. [n. d.]. Method chaining. Retrieved Aug 31, 2018 from https://en.wikipedia.org/wiki/Method_chaining
[112] Wikipedia. [n. d.]. Uniform Function Call Syntax. Retrieved Aug 31, 2018 from https://en.wikipedia.org/wiki/Uniform_Function_Call_Syntax
[113] Niklaus Wirth. 1971. The programming language Pascal. *Acta informatica* 1, 1 (1971), 35–63.
[114] Serdar Yegulalp. 2017. Free at last! D language's official compiler is open source. (10 April 2017). https://infoworld.com/article/3188427/application-development/free-at-last-d-languages-official-compiler-is-open-source.html