

2007

An Analysis of the D Programming Language

Sumanth Yenduri

University of Mississippi- Long Beach


Louise Perkins

University of Southern Mississippi- Long Beach

Md. Sarder

University of Southern Mississippi- Long Beach

Follow this and additional works at: <http://scholarworks.lib.csusb.edu/jitim>

 Part of the [Business Intelligence Commons](#), [E-Commerce Commons](#), [Management Information Systems Commons](#), [Management Sciences and Quantitative Methods Commons](#), [Operational Research Commons](#), and the [Technology and Innovation Commons](#)

Recommended Citation

Yenduri, Sumanth; Perkins, Louise; and Sarder, Md. (2007) "An Analysis of the D Programming Language," *Journal of International Technology and Information Management*: Vol. 16: Iss. 3, Article 7.

Available at: <http://scholarworks.lib.csusb.edu/jitim/vol16/iss3/7>

This Article is brought to you for free and open access by CSUSB ScholarWorks. It has been accepted for inclusion in Journal of International Technology and Information Management by an authorized administrator of CSUSB ScholarWorks. For more information, please contact scholarworks@csusb.edu.

An Analysis of the D Programming Language

Sumanth Yenduri

Louise Perkins

Md. Sarder

University of Southern Mississippi - Long Beach

ABSTRACT

The C language and its derivatives have been some of the dominant higher-level languages used, and the maturity has stemmed several newer languages that, while still relatively young, possess the strength of decades of trials and experimentation with programming concepts. While C++ was a major step in the advancement from procedural to object-oriented programming (with a backbone of C), several problems existed that prompted the development of new languages. This paper focuses on one such language: D. D was designed as a potential successor to C++, supporting most features of C++'s class design and modifications intended to ease common program development obstacles. This paper compares and contrasts the features of D against C and some of its derivatives.

INTRODUCTION AND BACKGROUND

Numerous technologies and languages are available in the current market but only a small percentage of projects complete on time and within budget (Chen, 2004; Post, 2005). Further, most projects are unreliable. Programmers lack the ability to select and apply the right language/technology for a given application (Galup, 2004; Keith, 2006; Marold, 2004). There is a great need to understand the strengths, weaknesses and various programming paradigms available in various languages. In this paper, we analyze a new and promising programming language called "D". Before delving into the world of current programming languages, many of which are designed to enhance the syntax and semantics of the C programming language, it is useful to understand the evolution of the base language itself. The creation of the C language started in the late 1960s at Bell Laboratories, as a concept of a system programming language for an early Unix operating system – shortly after the phase-out of Multics – on a DEC PDP-7. The development was led by Ken Thompson, a programmer well known for his work with refining assemblers that oftentimes were better than those DEC could produce; and Dennis Ritchie, who later developed Thompson's work into modern day C. Thompson's original motivation behind the language was a response to the Doug McIlroy's first implementation of the TMG language on the PDP-7. Thompson wanted to create a system programming language of his own that freed him from the difficulties of working directly with an assembler (Kernighan, 1970). This initial creation led to B, which was a simplification and refinement of a previous higher-level programming language, BCPL, used on other hardware architectures. B was a typeless language, which meant that all data was stored in fixed-length memory allocations. The operation on the memory determined the actions and calculations made. Pointers in the language were often simply integers that used vector-based logic to reference another cell down the line in memory. Most of the syntax is still very similar to present day C. Several drawbacks in B provided a demand for a new, more efficient language. Because B was typeless, all variables and data were stored in the same size cell, a 16 bit word. Floating point arithmetic had been possible on earlier machines because the size of a word had been larger, but the current hardware did not support the increased sizes necessary that a typeless language needed to perform large, floating point operations. Additionally, since B was so very slow – the compiled code ran in "stack" format, pushing and popping references to libraries in the correct order – the speed advantage of the PDP-11 started to demand optimizations of B. B was simply not capable of handling the new changes in hardware, along with the growing complexity of developers' needs.

Initial versions of C were introduced in 1971 that added a character type (previously handled by basic assembler integer data). Dennis Ritchie also modified the B compiler to produce PDP-11 instructions instead of stack-implemented library calls, thereby increasing the performance greatly. A great leap was made in handling "pointers" in code. Because of the typeless nature of the B language, pointers had to be stored in words in memory, providing an index to an array or other memory structure. In the early (and still current) versions of C, pointers are created "on-the-fly," and an array name became a type that was automatically converted to the address pointer in memory to

the first element. All references in the index simply offset this number dynamically during runtime. This saved a lot of space in memory and reduced execution time because of fewer trips outside the CPU to fetch reference points. By 1973, the C language resembled its modern day counterpart. In 1982, C was first given standards that set guidelines for portable C coding, and it became a dominant program language for years to follow (Ritchie, 1973). C++ was originally called “C with Classes,” first conceived of in 1979 and running on to 1983 before the name and concepts changed. Created by Bjarne Stroustrup, the idea was to provide Simula’s facilities for program organization together with C’s efficiency and flexibility for systems programming. Bjarne originally was interested exclusively in the concept of Simula’s object based design, but the compilers and interpreters available for pure Simula code were not scalable enough to large programs and ran inefficiently. He chose to make his own implementation of those features in C because it was reliable, portable, and efficient. With the new introduction of C++ in 1985, many major applications had begun development through object-oriented processes.

Object-oriented programming has been acclaimed by figures like Grady Booch in recent years as the key to successful, industrial-strength programming. With C’s efficiency powering C++ and allowing higher-level abstractions, complex end-user applications became a possibility at the end of the twentieth and beginning of the twenty-first centuries (Stroustrup, 1993). While C++ was major step in the advancement from procedural to object-oriented programming – with a backbone of C providing efficient and portable code – several problems existed that prompted the development of languages evolving from C++. Though C is a high-level language advertising high portability, it still must produce machine-language code dependent on the architecture. This is generally achieved through the use of multiple compilers for different systems, but the lack of synchronization between them, even with a standard, is often a problem. No garbage collection exists; all memory allocation and cleanup is manual and performed by the user. Thread concurrency code is not natively supported in C++; libraries must be imported, and dependencies are formed. Some of the circular dependencies caused by using header files made management of multi-source programs a potential nightmare.

And despite the advancement of Stroustrup’s use of classes in C++, many features still were missing or prone to errors. Several responses to C++’s shortcomings have been produced in the last few years. Unlike C++’s new object-oriented techniques implemented into C, languages following have improved more upon the practical aspects of the programming (syntax, semantics, safety-checking) than in actual theory. This paper focuses on the features of D programming language. An elaboration on the history, use, and features is presented in the following pages, followed by an analysis and side-by-side comparison.

ANALYSIS OF FEATURES

The D programming language was designed as a potential successor to C++, including support for most features of C++’s class design and modifications intended to ease common program development bugs and obstacles. A community of developers – many affiliated with the GNU Software Foundation - has provided the specifications and direction for the language’s development in coordination with the creator, Walter Bright. The syntax resembles that of C++, C#, and Java. Currently, two main threads of development exist on compilers for the language: Digital Mars’ D compiler, and the GNU D compiler. While being two distinct projects, some coordination between the two goes on. There are packages available for Windows operating systems, Linux, and Mac OS X (restricted to GCC D compiler). While other languages have been created directly from a business environment or motivated for corporate use, D was created through a collaboration of computer scientists in a “Wiki-like” environment (Walter, 1999). In 1995, when Java was released, Microsoft was among many software developers searching for an alternative to C++’s shortcomings. Java provided many of the features which were sought after, while providing language syntax very similar to C++, allowing for rapid adoption by existing C++ programmers. Adapting quickly, Microsoft began using Java alongside their C and C++ code in their development process to solve several issues with their development process, including cross platform portability. D provides a C++ inspired object oriented framework. D code has a look and feel very similar to C and C++, allowing programmers familiar with these languages to adapt quickly. D program texts are simpler and more reusable. It solves some of the difficult problems that exist with respect distributed applications, and provides the framework for analyzing such programs in ways that cannot be done with other generic programming languages (Lopes, 1997).

However, D makes several improvements to its object programming model. With D, the programmer need not write both a header and implementation file for a class. The programmer instead needs to only write the class declaration with functions written in-line, while the compiler generates the required symbolic information. As previously stated

D works well for programmers comfortable with the C++ or C specification. D provides direct access to C's APIs and functions, support for all C data types, and works with existing linkers and debuggers. D also provides a performance-driven approach to coding. D supports integration of C style structs when creating a class is unnecessary, cutting on memory costs. In addition, D provides language specifications for 'synchronization' primitives used with multithreaded applications, at both the object and method levels. The synchronize statement automatically configures mutex and prevents deadlock. D also provides an inline assembler, allowing developers to insert hardware specific assembly code into D programs, where optimization or specific instructions are needed (Walter, 1999). To ensure verifiably correct code, the D specification also includes DBC, a contract programming method which allows the programmer to specify pre condition and post condition contracts to test, before and after execution of a function, to ensure that the results are consistent with what is expected (Walter, 1999). D also natively supports and mandates unit tests; simple coded tests integrated with class source code, to ensure that class implementations aren't inadvertently disrupted during a build. This allows testing to be greatly aided by the development process itself. It also allows programmers to verify the quality of foreign extensions and libraries. Debug is also controlled within the language, with code able to be enabled for debugging or release modes through syntax, allowing for ease of development in testing and release cycles (Walter, 1999).

Sun Microsystems, within their Solaris Operating System, has implemented a kernel-level event tracing system. This software, called DTrace, or dynamic tracing, allows up to 30,000 kernel instrumentation points to be tracked during event tracing. Developers for Solaris OS software are able to specify data to be collected, and actions to be taken by means of an API. The chosen language for implementing the "invaluable" API is the D language, touted by Sun as similar to C, while being easy to learn and use (Cantrill, 2005). D also has a following in independent software development, ranging from game production to database management tools.

Array Management

In the D programming language, four types of arrays exist: pointers to data, static arrays, dynamic arrays, and associative arrays. Numerous enhancements have been made to traditional C and C++ style handling. While C has only operators to affect the handle of an array, D has that and the ability to affect the contents of the array. D has the ability to perform array slicing through assignments of another array to part of a currently existing one. In addition, single pointers can be converted into arrays by specifying bounds that are computed by pointer arithmetic (start one after point and end eight after pointer, etc.). Arrays can be copied easily without providing syntax through repetition loops. The full complement of array operations available in C and C++ has been enhanced, and D is particularly adept at handling strings, which is discussed later.

Another particularly beneficial feature is built-in support for associative arrays – which support keys to access unique array elements. Using a key as an array subscript while referencing a value is directly supported without any additional coding, and can be used in place of the string or integer literal it references in the array. The key need not necessarily be characters; instead, structures, classes, and unions can be used in place, following the appropriate syntax. The associative array also has a property that can be invoked to rehash itself once all symbols are loaded, providing more efficient and distinct keys that produce less collisions. Array bound checking exists both at compile time and during execution. The latter is not meant as an added reliability the programmer has in safely executing the program; rather, the responsibility is still left up to provide good coding. This dynamic runtime feature is intended to enhance what is already provided in parsing. However, the static array bounds checking singles out a large majority of the errors that normally would make it to runtime in C or C++ (Walter, 1999).

Garbage Collection

One of the major improvements of current-generation languages is automatic garbage collection. D is no exception. While C and C++ have manual allocation and deallocation of memory, D is fully automatic. The programmer needs only to allocate memory, and the garbage collector will come through and return unused portions to memory. The collection process works by scanning for all objects without pointer references using recursive search processes. Once a "sweep" has been made, the unused memory gets returned to the heap. This works similarly to the generic "mark and sweep" algorithm. While D garbage collection can improve memory leaks and actually increase performance by managing the amount of a program in cache and memory, it has downsides. Bringing in foreign

code can cause the garbage collector to miss pieces of unused memory to free, and pointers to large data structure roots may prevent all of the linked data structure from being claimed back into memory (Walter, 1999).

Strings

Strings in D support the UTF-32 Unicode string implementation. String internal char types, however, are auto-sized, depending on the characters appended to strings. If a program uses only ASCII, then the string char type will remain of minimum size, and not be reallocated. Strings are of a character array type similar to C++. Strings support most array functionality, including slicing, indexing, concatenation, conversion between UTF formats and ASCII, searching, comparison, and duplication (Walter, 1999). D strings can also be converted to C style strings. D strings have a specifiable and modifiable length parameter. Checking to see if a string is empty is simply checking the `str.length` to determine if it is zero. This length can also be set and modified with the normal assignment operator, allowing for variable length strings. D also supports use of strings as array indices, cases for switch statements, and other comparisons (Walter, 1999). D introduces a new binary operator `'~'` for concatenation.

Functions

The table below is a sample of some of the comparisons of the features of functions.

Table1: Features of Functions among D, C, C++, C# and Java.

| Functions | <u>D</u> | <u>C</u> | <u>C++</u> | <u>C#</u> | <u>Java</u> |
|-----------------------------------|----------|----------|------------|-----------|-------------|
| Function delegates | Yes | No | No | Yes | No |
| Function overloading | Yes | No | Yes | Yes | Yes |
| Out function parameters | Yes | Yes | Yes | Yes | No |
| Nested functions | Yes | No | No | No | No |
| Function literals | Yes | No | No | No | No |
| Dynamic closures | Yes | No | No | No | No |
| Typesafe variadic arguments | Yes | No | No | Yes | Yes |
| Lazy function argument evaluation | Yes | No | No | No | No |

Documentation Generation

Different approaches exist in the D language for adding documentation, but there is no standard form of extraction to parse D-style comments. The idea behind the D language documentation is to prevent redundant material from reaching the pages while providing an easy and natural way to write comments. A programmer would, ideally, write comments in the syntax provided, and the desired extraction program would sort it out without requiring any embedded HTML, unusually difficult syntax, or major obstructions in the D parser's functionality. Some of the extraction programs that provide support for D include Doxygen, a program that returns information from commented source code in many different languages and produces any number of types of documentation (PDF, LaTeX, HTML, Postscript). Another method is to use Javadoc, Sun's documentation generation program, adaptable to programming languages other than Java. Additionally, the form of documentation generation for C#, which is embedded XML, can be adapted to work with the D language. Aside from normal commenting conventions, some of the most important features in D comments include special sections about licensing of the product, a "deprecated feature" selection that allows the extractor to note new functions that override old code, the ability to directly paste test code into comments, and special recognition of parameters for functions – so that the extractor will directly recognize these as arguments and their descriptions. D provides macros that allow HTML-style tags to be inferred from their placement. The advantage here over directly placing HTML tags is that extractors used may not be trying

to output code into XML or HTML format. Most of the extractors that work with D that are not outputting in HTML will simply ignore the macros and convert the documentation to the desired format, reducing the risk of any garbled document generation (Walter, 1999).

CONCLUSIONS

D shares a wide offering of features such as 80-bit floating points, in-line assembly, and code support for array manipulation that give it a performance boost. Automatic memory management, easy class types for semaphore-based threading and mutual exclusion and automatic documentation generation reinforce strong systems development features. The ease of the language to pick up by developers, along with the ease of writing a compiler makes it ideal for specific systems design. Tiobe Software index history for D has shown a slow but steady rise (Tiobe Software, 2005) in recent times. Numerical analysis and scientific programmers would benefit most from D's features, as they are geared towards high performance accurate calculation. D enables researchers and developers to code at a higher level and be more productive (in certain applications). D provides similar benefits to the lower level systems programming area where C/C++ still dominates. D is a necessary and useful evolution and offers developers a set of tools and refined processes. In the scope of language adoption, D is young, and will likely continue to grow in usage and features, perhaps becoming a stepping-stone for future programming language evolution.

REFERENCES

- Cantrill, B. (2005). *Dynamic Tracing (DTrace)*. Retrieved February, 2007, from Sun Microsystems. Web site: <http://partneradvantage.sun.com/protected/solaris10/adoptionkit/general/features/dtrace.html>.
- Chen, H. (2004). The Impacts of Conflicts on Requirements Uncertainty and Project Performance. *Journal of International Technology and Information Management*. 13(3), 57-168.
- Galup, S.D. (2004). The Demand for Information Technology Knowledge and Skills: An Exploratory Investigation, *Journal of International Technology and Information Management*. 13(4), 253-262.
- Keith, N. (2006). Differing Cultural Perceptions Regarding the Appropriate Use of Workplace Computer Technologies, *Journal of International Technology and Information Management*. 15(1), 19-26.
- Kernighan, B. W. (1970). *The Development of the C Language*. AT & T Bell Labs.
- Lopes, C. V. (1997). *D: A Language Framework for Distributed Programming*. Xerox Palo Alto Research Center, Technical Report SPL97-007 P9710047.
- Marold, K.A. (2004). Measuring Online Students' Ability To Apply Programming Theory: Are Web Courses Really Working? *Journal of International Technology and Information Management*. 13(1), 13-20.
- Post, G.V. (2005). Systems Development Tools and the Relationship to Project Design: Cost and Budget Implications, *Journal of International Technology and Information Management*. 14(1).
- Ritchie, D. (1973). *Very Early C compilers and Language*. Retrieved February, 2007, from AT & T Bell Labs. Web site: <http://www.cs.bell-labs.com/who/dmr/primvalC.html>.
- Stroustrup, B. (1993). A History of C++: 1979-1991. *Proceedings of the ACM History of Programming Languages conference (HOPL-2)*, *ACM Sigplan Notices*. 28(3), 271-298.
- Tiobe Software. (2005). Index History D. Retrieved February, 2007, from Tiobe Software. Web site: http://www.tiobe.com/tiobe_index/D.html.
- Walter, B. (1999). *The D Programming Language*. Retrieved February, 2007, from Digital Mars. Web site: <http://www.digitalmars.com/d/index.html>.

