The D Programming Language

DMD 0.109
12/18/04 Snapshot

Table of Contents

Comparisons

D Programming Language

"It seems to me that most of the "new" programming languages fall into one of two categories: Those from academia with radical new paradigms and those from large corporations with a focus on RAD and the web. Maybe its time for a new language born out of practical experience implementing compilers." -- Michael

"Great, just what I need.. another D in programming." -- Segfault

This is the reference document for the D programming language. D was conceived in December 1999 by myself as a reengineering of C and C++, and has grown and evolved with helpful suggestions and critiques by my friends and colleagues. I've been told the usual, that there's no chance for a new programming language, that who do I think I am designing a language, etc. Take a look at the document and decide for yourself!

Check out the quick comparison of D with C, C++, C# and Java.

The D newsgroup in news.digitalmars.com server is where discussions of this should go. Suggestions, criticism, kudos, flames, etc., are all welcome there. Alternatively, try the D forum. There also may be a local D user group in your community (or you can start one!).

Download the current version of the compiler for Win32 and x86 Linux and try it out!

David Friedman has integrated the D frontend with GCC.

Alternate versions of this document:

There's a pdf version of the specification.

Kazuhiro Inaba has prepared a Japanese translation.

For SDWest 2004 I gave a presentation on D.

Note: all D users agree that by downloading and using D, or reading the D specs, they will

explicitly identify any claims to intellectual property rights with a copyright or patent notice in any posted or emailed feedback sent to Digital Mars.

-Walter

Overview

What is D?

D is a general purpose systems and applications programming language. It is a higher level language than C++, but retains the ability to write high performance code and interface directly with the operating system API's and with hardware. D is well suited to writing medium to large scale million line programs with teams of developers. D is easy to learn, provides many capabilities to aid the programmer, and is well suited to aggressive compiler optimization technology.

D is not a scripting language, nor an interpreted language. It doesn't come with a VM, a religion, or an overriding philosophy. It's a practical language for practical programmers who need to get the job done quickly, reliably, and leave behind maintainable, easy to understand code.

D is the culmination of decades of experience implementing compilers for many diverse languages, and attempting to construct large projects using those languages. D draws inspiration from those other languages (most especially C++) and tempers it with experience and real world practicality.

Why D?

Why, indeed. Who needs another programming language?

The software industry has come a long way since the C language was invented. Many new concepts were added to the language with C++, but backwards compatibility with C was maintained, including compatibility with nearly all the weaknesses of the original design. There have been many attempts to fix those weaknesses, but the compatibility issue frustrates it. Meanwhile, both C and C++ undergo a constant accretion of new features. These new features must be carefully fitted into the existing structure without requiring rewriting old code. The end result is very complicated - the C standard is nearly 500 pages, and the C++ standard is about 750 pages! C++ is a difficult and costly language to implement, resulting in implementation variations that make it frustrating to write fully portable C++ code.

C++ programmers tend to program in particular islands of the language, i.e. getting very proficient using certain features while avoiding other feature sets. While the code is usually portable from compiler to compiler, it can be hard to port it from programmer to programmer. A

great strength of C++ is that it can support many radically different styles of programming - but in long term use, the overlapping and contradictory styles are a hindrance.

C++ implements things like resizable arrays and string concatenation as part of the standard library, not as part of the core language. Not being part of the core language has several suboptimal consequences.

Can the power and capability of C++ be extracted, redesigned, and recast into a language that is simple, orthogonal, and practical? Can it all be put into a package that is easy for compiler writers to correctly implement, and which enables compilers to efficiently generate aggressively optimized code?

Modern compiler technology has progressed to the point where language features for the purpose of compensating for primitive compiler technology can be omitted. (An example of this would be the 'register' keyword in C, a more subtle example is the macro preprocessor in C.) We can rely on modern compiler optimization technology to not need language features necessary to get acceptable code quality out of primitive compilers.

Major Goals of D

Reduce software development costs by at least 10% by adding in proven productivity enhancing features and by adjusting language features so that common, time-consuming bugs are eliminated from the start.

Make it easier to write code that is portable from compiler to compiler, machine to machine, and operating system to operating system.

Support multi-paradigm programming, i.e. at a minimum support imperative, structured, object oriented, and generic programming paradigms.

Have a short learning curve for programmers comfortable with programming in C or C++.

Provide low level bare metal access as required.

Make D substantially easier to implement a compiler for than C++.

Be compatible with the local C application binary interface.

Have a context-free grammar.

Easily support writing internationalized applications.

Incorporate Contract Programming and unit testing methodology.

Be able to build lightweight, standalone programs.

Features To Keep From C/C++

The general look of D is like C and C++. This makes it easier to learn and port code to D. Transitioning from C/C++ to D should feel natural. The programmer will not have to learn an entirely new way of doing things.

Using D will not mean that the programmer will become restricted to a specialized runtime vm

(virtual machine) like the Java vm or the Smalltalk vm. There is no D vm, it's a straightforward compiler that generates linkable object files. D connects to the operating system just like C does. The usual familiar tools like **make** will fit right in with D development.

The general **look and feel** of C/C++ will be maintained. It will use the same algebraic syntax, most of the same expression and statement forms, and the general layout.

D programs can be written either in C style **function-and-data** or in C++ style **object-oriented**, or any mix of the two.

The **compile/link/debug** development model will be carried forward, although nothing precludes D from being compiled into bytecode and interpreted.

Exception handling. More and more experience with exception handling shows it to be a superior way to handle errors than the C traditional method of using error codes and errno globals.

Runtime Type Identification. This is partially implemented in C++; in D it is taken to its next logical step. Fully supporting it enables better garbage collection, better debugger support, more automated persistence, etc.

D maintains function link compatibility with the **C calling conventions**. This makes it possible for D programs to access operating system API's directly. Programmers' knowledge and experience with existing programming API's and paradigms can be carried forward to D with minimal effort.

Operator overloading. D programs can overload operators enabling extension of the basic types with user defined types.

Templates. Templates are a way to implement generic programming. Other ways include using macros or having a variant data type. Using macros is out. Variants are straightforward, but inefficient and lack type checking. The difficulties with C++ templates are their complexity, they don't fit well into the syntax of the language, all the various rules for conversions and overloading fitted on top of it, etc. D offers a much simpler way of doing templates.

RAII (Resource Acquisition Is Initialization). RAII techniques are an essential component of writing reliable software.

Down and dirty programming. D will retain the ability to do down-and-dirty programming without resorting to referring to external modules compiled in a different language. Sometimes, it's just necessary to coerce a pointer or dip into assembly when doing systems work. D's goal is not to *prevent* down and dirty programming, but to minimize the need for it in solving routine coding tasks.

Features To Drop

C source code compatibility. Extensions to C that maintain source compatibility have already been done (C++ and ObjectiveC). Further work in this area is hampered by so much legacy code it is unlikely that significant improvements can be made.

Link compatibility with C++. The C++ runtime object model is just too complicated -

properly supporting it would essentially imply making D a full C++ compiler too.

The C preprocessor. Macro processing is an easy way to extend a language, adding in faux features that aren't really there (invisible to the symbolic debugger). Conditional compilation, layered with #include text, macros, token concatenation, etc., essentially forms not one language but two merged together with no obvious distinction between them. Even worse (or perhaps for the best) the C preprocessor is a very primitive macro language. It's time to step back, look at what the preprocessor is used for, and design support for those capabilities directly into the language.

Multiple inheritance. It's a complex feature of debatable value. It's very difficult to implement in an efficient manner, and compilers are prone to many bugs in implementing it. Nearly all the value of MI can be handled with single inheritance coupled with interfaces and aggregation. What's left does not justify the weight of MI implementation.

Namespaces. An attempt to deal with the problems resulting from linking together independently developed pieces of code that have conflicting names. The idea of modules is simpler and works much better.

Tag name space. This misfeature of C is where the tag names of struct's are in a separate but parallel symbol table. C++ attempted to merge the tag name space with the regular name space, while retaining backward compatibility with legacy C code. The result is not printable.

Forward declarations. C compilers semantically only know about what has lexically preceded the current state. C++ extends this a little, in that class members can rely on forward referenced class members. D takes this to its logical conclusion, forward declarations are no longer necessary at all. Functions can be defined in a natural order rather than the typical inside-out order commonly used in C programs to avoid writing forward declarations.

Include files. A major cause of slow compiles as each compilation unit must reparse enormous quantities of header files. Include files should be done as importing a symbol table.

Creating object instances on the stack. In D, all class objects are by reference. This eliminates the need for copy constructors, assignment operators, complex destructor semantics, and interactions with exception handling stack unwinding. Memory resources get freed by the garbage collector, other resources are freed by using the RAII features of D.

Trigraphs and digraphs. Unicode is the future.

Preprocessor. Modern languages should not be text processing, they should be symbolic processing.

Non-virtual member functions. In C++, a class designer decides in advance if a function is to be virtual or not. Forgetting to retrofit the base class member function to be virtual when the function gets overridden is a common (and very hard to find) coding error. Making all member functions virtual, and letting the compiler decide if there are no overrides and hence can be converted to non-virtual, is much more reliable.

Bit fields of arbitrary size. Bit fields are a complex, inefficient feature rarely used.

Support for 16 bit computers. No consideration is given in D for mixed near/far pointers and all the machinations necessary to generate good 16 bit code. The D language design assumes at least a 32 bit flat memory space. D will fit smoothly into 64 bit architectures.

Mutual dependence of compiler passes. In C++, successfully parsing the source text relies on having a symbol table, and on the various preprocessor commands. This makes it impossible to preparse C++ source, and makes writing code analyzers and syntax directed editors painfully difficult to do correctly.

Compiler complexity. Reducing the complexity of an implementation makes it more likely that multiple, *correct* implementations are available.

Distinction between . and ->. This distinction is really not necessary. The . operator serves just as well for pointer dereferencing.

Who D is For

Programmers who routinely use lint or similar code analysis tools to eliminate bugs before the code is even compiled.

People who compile with maximum warning levels turned on and who instruct the compiler to treat warnings as errors.

Programming managers who are forced to rely on programming style guidelines to avoid common C bugs.

Those who decide the promise of C++ object oriented programming is not fulfilled due to the complexity of it.

Programmers who enjoy the expressive power of C++ but are frustrated by the need to expend much effort explicitly managing memory and finding pointer bugs.

Projects that need built-in testing and verification.

Teams who write apps with a million lines of code in it.

Programmers who think the language should provide enough features to obviate the continual necessity to manipulate pointers directly.

Numerical programmers. D has many features to directly support features needed by numerics programmers, like direct support for the complex data type and defined behavior for NaN's and infinities. (These are added in the new C99 standard, but not in C++.)

D's lexical analyzer and parser are totally independent of each other and of the semantic analyzer. This means it is easy to write simple tools to manipulate D source perfectly without having to build a full compiler. It also means that source code can be transmitted in tokenized form for specialized applications.

Who D is Not For

Realistically, nobody is going to convert million line C or C++ programs into D, and

since D does not compile unmodified C/C++ source code, D is not for legacy apps. (However, D supports legacy C API's very well.)

Very small programs - a scripting or interpreted language like Python, DMDScript, or Perl is likely more suitable.

As a first programming language - Basic or Java is more suitable for beginners. D makes an excellent second language for intermediate to advanced programmers.

Language purists. D is a practical language, and each feature of it is evaluated in that light, rather than by an ideal. For example, D has constructs and semantics that virtually eliminate the need for pointers for ordinary tasks. But pointers are still there, because sometimes the rules need to be broken. Similary, casts are still there for those times when the typing system needs to be overridden.

Major Features of D

This section lists some of the more interesting features of D in various categories.

Object Oriented Programming

Classes

D's object oriented nature comes from classes. The inheritance model is single inheritance enhanced with interfaces. The class Object sits at the root of the inheritance hierarchy, so all classes implement a common set of functionality. Classes are instantiated by reference, and so complex code to clean up after exceptions is not required.

Operator Overloading

Classes can be crafted that work with existing operators to extend the type system to support new types. An example would be creating a bignumber class and then overloading the +, -, * and / operators to enable using ordinary algebraic syntax with them.

Productivity

Modules

Source files have a one-to-one correspondence with modules. Instead of #include'ing the text of a file of declarations, just import the module. There is no need to worry about multiple imports of the same module, no need to wrapper header files with `#ifndef/#endif` or `#pragma once` kludges, etc.

Declaration vs Definition

C++ usually requires that functions and classes be declared twice - the declaration that goes in the .h header file, and the definition that goes in the .c source file. This is an error prone and tedious process. Obviously, the programmer should only need to write it once, and the compiler should then extract the declaration information and make it available for symbolic importing. This is exactly how D works.

Example:

```
class ABC
{
    int func() { return 7; }
    static int z = 7;
}
int q;
```

There is no longer a need for a separate definition of member functions, static members, externs, nor for clumsy syntaxes like:

```
int ABC::func() { return 7; }
int ABC::z = 7;
extern int q;
```

Note: Of course, in C++, trivial functions like `{ return 7; }` are written inline too, but complex ones are not. In addition, if there are any forward references, the functions need to be prototyped. The following will not work in C++:

```
class Foo
{
    int foo(Bar *c) { return c->bar; }
};

class Bar
{
  public:
    int bar() { return 3; }
};
```

But the equivalent D code will work:

```
class Foo
{
    int foo(Bar c) { return c.bar; }
}

class Bar
{
    int bar() { return 3; }
}
```

Whether a D function is inlined or not is determined by the optimizer settings.

Templates

D templates offer a clean way to support generic programming while offering the power of partial specialization.

Associative Arrays

Associative arrays are arrays with an arbitrary data type as the index rather than being limited to an integer index. In essence, associated arrays are hash tables. Associative arrays make it easy to build fast, efficient, bug-free symbol tables.

Real Typedefs

C and C++ typedefs are really type *aliases*, as no new type is really introduced. D implements real typedefs, where:

```
typedef int handle;
```

really does create a new type **handle**. Type checking is enforced, and typedefs participate in function overloading. For example:

```
int foo(int i);
int foo(handle h);
```

Bit type

The fundamental data type is the bit, and D has a `bit` data type. This is most useful in creating arrays of bits:

```
bit[] foo;
```

Functions

D has the expected support for ordinary functions including global functions, overloaded functions, inlining of functions, member functions, virtual functions, function pointers, etc. In addition:

Nested Functions

Functions can be nested within other functions. This is highly useful for code factoring, locality, and function closure techniques.

Function Literals

Anonymous functions can be embedded directly into an expression.

Dynamic Closures

Nested functions and class member functions can be referenced with closures (also called delegates), making generic programming much easier and type safe.

In, Out, and Inout Parameters

Not only does specifying this help make functions more self-documenting, it eliminates much of the necessity for pointers without sacrificing anything, and it opens up possibilities for more compiler help in finding coding problems.

Such makes it possible for D to directly interface to a wider variety of foreign API's. There would be no need for workarounds like "Interface Definition Languages".

Arrays

C arrays have several faults that can be corrected:

Dimension information is not carried around with the array, and so has to be stored and passed separately. The classic example of this are the argc and argv parameters to `main(int argc, char *argv[])`. (In D, main is declared as `main(char[][] args)`.)

Arrays are not first class objects. When an array is passed to a function, it is converted to a pointer,even though the prototype confusingly says it's an array. When this conversion happens, all array type information gets lost.

C arrays cannot be resized. This means that even simple aggregates like a stack need to be constructed as a complex class.

C arrays cannot be bounds checked, because they don't know what the array bounds are.

Arrays are declared with the [] after the identifier. This leads to very clumsy syntax to declare things like a pointer to an array:

```
int (*array)[3];
```

In D, the [] for the array go on the left:

```
int[3] *array;          declares a pointer to an array of 3 ints
long[] func(int x);     declares a function returning an array
of longs
```

which is much simpler to understand.

D arrays come in 4 varieties: pointers, static arrays, dynamic arrays, and associative arrays. See Arrays.

Strings

String manipulation is so common, and so clumsy in C and C++, that it needs direct support in the language. Modern languages handle string concatenation, copying, etc., and so does D. Strings are a direct consequence of improved array handling.

Resource Management

Garbage Collection

D memory allocation is fully garbage collected. Empirical experience suggests that a lot of the complicated features of C++ are necessary in order to manage memory deallocation. With garbage collection, the language gets much simpler.

There's a perception that garbage collection is for lazy, junior programmers. I remember when that was said about C++, after all, there's nothing in C++ that cannot be done in C, or in assembler for that matter.

Garbage collection eliminates the tedious, error prone memory allocation tracking code necessary in C and C++. This not only means much faster development time and lower maintenance costs, but the resulting program frequently runs faster!

Sure, garbage collectors can be used with C++, and I've used them in my own C++ projects. The language isn't friendly to collectors, however, impeding the effectiveness of it. Much of the runtime library code can't be used with collectors.

For a fuller discussion of this, see garbage collection.

Explicit Memory Management

Despite D being a garbage collected language, the new and delete operations can be overridden for particular classes so that a custom allocator can be used.

RAII

RAII is a modern software development technique to manage resource allocation and deallocation. D supports RAII in a controlled, predictable manner that is independent of the garbage collection cycle.

Performance

Lightweight Aggregates

D supports simple C style struct's, both for compatibility with C data structures and because they're useful when the full power of classes is overkill.

Inline Assembler

Device drivers, high performance system applications, embedded systems, and specialized code sometimes need to dip into assembly language to get the job done. While D implementations are not required to implement the inline assembler, it is defined and part of the language. Most assembly code needs can be handled with it, obviating the need for separate assemblers or DLL's.

Many D implementations will also support intrinsic functions analogously to C's support of intrinsics for I/O port manipulation, direct access to special floating point operations, etc.

Reliability

A modern language should do all it can to help the programmer flush out bugs in the code. Help can come in many forms; from making it easy to use more robust techniques, to compiler flagging of obviously incorrect code, to runtime checking.

Contracts

Contract Programming (invented by B. Meyer) is a revolutionary technique to aid in ensuring the correctness of programs. D's version of DBC includes function preconditions, function postconditions, class invariants, and assert contracts. See Contracts for D's implementation.

Unit Tests

Unit tests can be added to a class, such that they are automatically run upon program startup. This aids in verifying, in every build, that class implementations weren't inadvertantly broken. The unit tests form part of the source code for a class. Creating them becomes a natural part of the class development process, as opposed to throwing the finished code over the wall to the testing group.

Unit tests can be done in other languages, but the result is kludgy and the languages just aren't accommodating of the concept. Unit testing is a main feature of D. For library functions it works out great, serving both to guarantee that the functions actually work and to illustrate how to use the functions.

Consider the many C++ library and application code bases out there for download on the web. How much of it comes with *any* verification tests at all, let alone unit testing? Less than 1%? The usual practice is if it compiles, we assume it works. And we wonder if the warnings the

compiler spits out in the process are real bugs or just nattering about nits.

Along with Contract Programming, unit testing makes D far and away the best language for writing reliable, robust systems applications. Unit testing also gives us a quick-and-dirty estimate of the quality of some unknown piece of D code dropped in our laps - if it has no unit tests and no contracts, it's unacceptable.

Debug Attributes and Statements

Now debug is part of the syntax of the language. The code can be enabled or disabled at compile time, without the use of macros or preprocessing commands. The debug syntax enables a consistent, portable, and understandable recognition that real source code needs to be able to generate both debug compilations and release compilations.

Exception Handling

The superior *try-catch-finally* model is used rather than just try-catch. There's no need to create dummy objects just to have the destructor implement the *finally* semantics.

Synchronization

Multithreaded programming is becoming more and more mainstream, and D provides primitives to build multithreaded programs with. Synchronization can be done at either the method or the object level.

```
synchronized int func() { . }
```

Synchronized functions allow only one thread at a time to be executing that function.

The synchronize statement puts a mutex around a block of statements, controlling access either by object or globally.

Support for Robust Techniques

- Dynamic arrays instead of pointers
- Reference variables instead of pointers
- Reference objects instead of pointers
- Garbage collection instead of explicit memory management
- Built-in primitives for thread synchronization
- No macros to inadvertently slam code
- Inline functions instead of macros
- Vastly reduced need for pointers
- Integral type sizes are explicit

No more uncertainty about the signed-ness of chars

No need to duplicate declarations in source and header files.

Explicit parsing support for adding in debug code.

Compile Time Checks

Stronger type checking

No empty ; for loop bodies

Assignments do not yield boolean results

Deprecating of obsolete API's

Runtime Checking

assert() expressions

array bounds checking

undefined case in switch exception

out of memory exception

In, out, and class invariant Contract Programming support

Compatibility

Operator precedence and evaluation rules

D retains C operators and their precedence rules, order of evaluation rules, and promotion rules. This avoids subtle bugs that might arise from being so used to the way C does things that one has a great deal of trouble finding bugs due to different semantics.

Direct Access to C API's

Not only does D have data types that correspond to C types, it provides direct access to C functions. There is no need to write wrapper functions, parameter swizzlers, nor code to copy aggregate members one by one.

Support for all C data types

Making it possible to interface to any C API or existing C library code. This support includes structs, unions, enums, pointers, and all C99 types. D includes the capability to set the alignment of struct members to ensure compatibility with externally imposed data formats.

OS Exception Handling

D's exception handling mechanism will connect to the way the underlying operating system handles exceptions in an application.

Uses Existing Tools

D produces code in standard object file format, enabling the use of standard assemblers, linkers, debuggers, profilers, exe compressors, and other analyzers, as well as linking to code written in other languages.

Project Management

Versioning

D provides built-in support for generation of multiple versions of a program from the same text. It replaces the C preprocessor #if/#endif technique.

Deprecation

As code evolves over time, some old library code gets replaced with newer, better versions. The old versions must be available to support legacy code, but they can be marked as *deprecated*. Code that uses deprecated versions will be optionally flagged as illegal by a compiler switch, making it easy for maintenance programmers to identify any dependence on deprecated features.

No Warnings

D compilers will not generate warnings for questionable code. Code will either be acceptable to the compiler or it will not be. This will eliminate any debate about which warnings are valid errors and which are not, and any debate about what to do with them. The need for compiler warnings is symptomatic of poor language design.

Sample D Program (sieve.d)

```
/* Sieve of Eratosthenes prime numbers */

bit[8191] flags;

int main()
{   int i, count, prime, k, iter;

    printf("10 iterations\n");
    for (iter = 1; iter <= 10; iter++)
```

```
    {   count = 0;
        flags[] = 1;
        for (i = 0; i < flags.length; i++)
        {   if (flags[i])
            {   prime = i + i + 3;
                k = i + prime;
                while (k < flags.length)
                {
                    flags[k] = 0;
                    k += prime;
                }
                count += 1;
            }
        }
    }
    printf ("\n%d primes", count);
    return 0;
}
```

Lexical

In D, the lexical analysis is independent of the syntax parsing and the semantic analysis. The lexical analyzer splits the source text up into tokens. The lexical grammar describes what those tokens are. The D lexical grammar is designed to be suitable for high speed scanning, it has a minimum of special case rules, there is only one phase of translation, and to make it easy to write a correct scanner for. The tokens are readily recognizable by those familiar with C and C++.

Phases of Compilation

The process of compiling is divided into multiple phases. Each phase has no dependence on subsequent phases. For example, the scanner is not perturbed by the semantic analyzer. This separation of the passes makes language tools like syntax directed editors relatively easy to produce. It also is possible to compress D source by storing it in 'tokenized' form.

1. **source character set**
   The source file is checked to see what character set it is, and the appropriate scanner is loaded. ASCII and UTF formats are accepted.
2. **lexical analysis**
   The source file is divided up into a sequence of tokens. Special tokens are processed and removed.
3. **syntax analysis**
   The sequence of tokens is parsed to form syntax trees.
4. **semantic analysis**
   The syntax trees are traversed to declare variables, load symbol tables, assign types, and in general determine the meaning of the program.
5. **optimization**

Optimization is an optional pass that tries to rewrite the program in a semantically equivalent, but faster executing, version.

6. **code generation**
   Instructions are selected from the target architecture to implement the semantics of the program. The typical result will be an object file, suitable for input to a linker.

Source Text

D source text can be in one of the following formats:

ASCII

UTF-8

UTF-16BE

UTF-16LE

UTF-32BE

UTF-32LE

UTF-8 is a superset of traditional 7-bit ASCII. One of the following UTF BOMs (Byte Order Marks) can be present at the beginning of the source text:

Format	BOM
UTF-8	EF BB BF
UTF-16BE	FE FF
UTF-16LE	FF FE
UTF-32BE	00 00 FE FF
UTF-32LE	FF FE 00 00
ASCII	no BOM

There are no digraphs or trigraphs in D.

The source text consists of white space, end of lines, comments, special token sequences, tokens, all followed by end of file.

The source text is split into tokens using the maximal munch technique, i.e., the lexical analyzer tries to make the longest token it can. For example >> is a right shift token, not two greater than tokens.

End of File

```
EndOfFile:
        physical end of the file
        \u0000
        \u001A
```

The source text is terminated by whichever comes first.

End of Line

```
EndOfLine:
        \u000D
        \u000A
        \u000D \u000A
        EndOfFile
```

There is no backslash line splicing, nor are there any limits on the length of a line.

White Space

```
WhiteSpace:
        Space
        Space WhiteSpace

Space:
        \u0020
        \u0009
        \u000B
        \u000C
        EndOfLine
        Comment
```

White space is defined as a sequence of one or more of spaces, tabs, vertical tabs, form feeds, end of lines, or comments.

Comments

```
Comment:
        /* Characters */
        // Characters EndOfLine
        /+ Characters +/
```

D has three kinds of comments:

1. Block comments can span multiple lines, but do not nest.
2. Line comments terminate at the end of the line.
3. Nesting comments can span multiple lines and can nest.

Comment processing conceptually happens before tokenization. This means that embedded strings and comments do not prevent recognition of comment openings and closings:

```
a = /+ // +/ 1;         // parses as if 'a = 1;'
a = /+ "+/" +/ 1";      // parses as if 'a = " +/1";'
a = /+ /* +/ */ 3;      // parses as if 'a = */ 3;'
```

Comments cannot be used as token concatenators, for example, `abc/**/def` is two tokens, `abc` and `def`, not one `abcdef` token.

Tokens

Token:
- *Identifier*
- *StringLiteral*
- *CharacterLiteral*
- *IntegerLiteral*
- *FloatLiteral*
- *Keyword*
- **/**
- **/=**
- **.**
- **..**
- **...**
- **&**
- **&=**
- **&&**
- **|**
- **|=**
- **||**
- **-**
- **-=**
- **--**
- **+**
- **+=**
- **++**
- **<**
- **<=**
- **<<**
- **<<=**
- **<>**
- **<>=**
- **>**
- **>=**
- **>>=**
- **>>>=**
- **>>**
- **>>>**
- **!**
- **!=**
- **!==**
- **!<>**
- **!<>=**
- **!<**
- **!<=**
- **!>**
- **!>=**
- **(**
- **)**
- **[**
- **]**
- **{**

```
}
?
,
;
:
$
=
==
===
*
*=
%
%=
^
^=
~
~=
```

Identifiers

```
Identifier:
        IdentiferStart
        IdentiferStart IdentifierChars

IdentifierChars:
        IdentiferChar
        IdentiferChar IdentifierChars

IdentifierStart:
        _
        Letter
        UniversalAlpha

IdentifierChar:
        IdentiferStart
        Digit
```

Identifiers start with a letter, _, or unicode alpha, and are followed by any number of letters, _, digits, or universal alphas. Universal alphas are as defined in ISO/IEC 9899:1999(E) Appendix D. (This is the C99 Standard.) Identifiers can be arbitrarily long, and are case sensitive. Identifiers starting with __ (two underscores) are reserved.

String Literals

```
StringLiteral:
        WysiwygString
        AlternateWysiwygString
        DoubleQuotedString
        EscapeSequence
        HexString
```

```
        WysiwygString:
                r" WysiwygCharacters "

        AlternateWysiwygString:
                ` WysiwygCharacters `

        WysiwygCharacter:
                Character
                EndOfLine

        DoubleQuotedString:
                " DoubleQuotedCharacters "

        DoubleQuotedCharacter:
                Character
                EscapeSequence
                EndOfLine

        EscapeSequence:
                \'
                \"
                \?
                \\
                \a
                \b
                \f
                \n
                \r
                \t
                \v
                \ EndOfFile
                \x HexDigit HexDigit
                \ OctalDigit
                \ OctalDigit OctalDigit
                \ OctalDigit OctalDigit OctalDigit
                \u HexDigit HexDigit HexDigit HexDigit
                \U HexDigit HexDigit HexDigit HexDigit HexDigit HexDigit
HexDigit HexDigit

        HexString:
                x" HexStringChars "

        HexStringChar
                HexDigit
                WhiteSpace
                EndOfLine
```

A string literal is either a double quoted string, a wysiwyg quoted string, an escape sequence, or a hex string.

Wysiwyg quoted strings are enclosed by r" and ". All characters between the r" and " are part of the string except for *EndOfLine* which is regarded as a single \n character. There are no escape sequences inside r" ":

```
        r"hello"
        r"c:\root\foo.exe"
        r"ab\n"                 string is 4 characters, 'a', 'b', '\', 'n'
```

An alternate form of wysiwyg strings are enclosed by backquotes, the ` character. The ` character is not available on some keyboards and the font rendering of it is sometimes indistinguishable from the regular ' character. Since, however, the ` is rarely used, it is useful to delineate strings with " in them.

```
        `hello`
        `c:\root\foo.exe`
        `ab\n`                  string is 4 characters, 'a', 'b', '\', 'n'
```

Double quoted strings are enclosed by "". Escape sequences can be embedded into them with the typical \ notation. *EndOfLine* is regarded as a single \n character.

```
        "hello"
        "c:\\root\\foo.exe"
        "ab\n"                  string is 3 characters, 'a', 'b', and a
linefeed
        "ab
        "                       string is 3 characters, 'a', 'b', and a
linefeed
```

Escape strings start with a \ and form an escape character sequence. Adjacent escape strings are concatenated:

```
        \n                      the linefeed character
        \t                      the tab character
        \"                      the double quote character
        \012                    octal
        \x1A                    hex
        \u1234                  wchar character
        \U00101234              dchar character
        \r\n                    carriage return, line feed
```

Escape sequences not listed above are errors.

Hex strings allow string literals to be created using hex data:

```
        x"0A"                   same as "\x0A
        x"00 FBCD 32FD 0A"      same as "\x00\xFB\xCD\x32\xFD\x0A"
```

Whitespace and newlines are ignored, so the hex data can be easily formatted. The number of hex characters must be a multiple of 2.

Adjacent strings are concatenated with the ~ operator, or by simple juxtaposition:

```
        "hello " ~ "world" ~ \n // forms the string 'h','e','l','l','o','
','w','o','r','l','d',linefeed
```

The following are all equivalent:

```
"ab" "c"
r"ab" r"c"
r"a" "bc"
"a" ~ "b" ~ "c"
\x61"bc"
```

Character Literals

```
CharacterLiteral:
        ' SingleQuotedCharacter '

SingleQuotedCharacter
        Character
        EscapeSequence
```

Character literals are a single character or escape sequence enclosed by single quotes, ' '.

Integer Literals

```
IntegerLiteral:
        Integer
        Integer IntegerSuffix

Integer:
        Decimal
        Binary
        Octal
        Hexadecimal
        Integer _

IntegerSuffix:
        l
        L
        u
        U
        lu
        Lu
        lU
        LU
        ul
        uL
        Ul
        UL

Decimal:
        0
        NonZeroDigit
        NonZeroDigit Decimal
        NonZeroDigit _ Decimal
```

```
Binary:
        0b BinaryDigits
        0B BinaryDigits

Octal:
        0 OctalDigits

Hexadecimal:
        0x HexDigits
        0X HexDigits
```

Integers can be specified in decimal, binary, octal, or hexadecimal.

Decimal integers are a sequence of decimal digits.

Binary integers are a sequence of binary digits preceded by a '0b'.

Octal integers are a sequence of octal digits preceded by a '0'.

Hexadecimal integers are a sequence of hexadecimal digits preceded by a '0x' or followed by an 'h'.

Integers can have embedded '_' characters, which are ignored. The embedded '_' are useful for formatting long literals, such as using them as a thousands separator:

```
123_456         // 123456
1_2_3_4_5_6_    // 123456
```

Integers can be immediately followed by one 'l' or one 'u' or both.

The type of the integer is resolved as follows:

Decimal Literal	**Type**
0 .. 2147483647	int
2147483648 .. 9223372036854775807	long
Decimal Literal, L Suffix	**Type**
0L .. 9223372036854775807L	long
Decimal Literal, U Suffix	**Type**
0U .. 4294967295U	uint
4294967296U .. 18446744073709551615U	ulong
Decimal Literal, UL Suffix	**Type**
0UL .. 18446744073709551615UL	ulong
Non-Decimal Literal	**Type**

0x0 .. 0x7FFFFFFF	int
0x80000000 .. 0xFFFFFFFF	uint
0x100000000 .. 0x7FFFFFFFFFFFFFFF	long
0x8000000000000000 .. 0xFFFFFFFFFFFFFFFF	ulong
Non-Decimal Literal, L Suffix	**Type**
0x0L .. 0x7FFFFFFFFFFFFFFL	long
0x8000000000000000L .. 0xFFFFFFFFFFFFFFFL	ulong
Non-Decimal Literal, U Suffix	**Type**
0x0U .. 0xFFFFFFFFU	uint
0x100000000UL .. 0xFFFFFFFFFFFFFFFFUL	ulong
Non-Decimal Literal, UL Suffix	**Type**
0x0UL .. 0xFFFFFFFFFFFFFFFFUL	ulong

Floating Literals

```
FloatLiteral:
        Float
        Float FloatSuffix
        Float ImaginarySuffix
        Float FloatSuffix ImaginarySuffix

Float:
        DecimalFloat
        HexFloat
        Float _

FloatSuffix:
        f
        F
        l
        L

ImaginarySuffix:
        i
        I
```

Floats can be in decimal or hexadecimal format, as in standard C.

Hexadecimal floats are preceded with a **0x** and the exponent is a **p** or **P** followed by a power of 2.

Floating literalss can have embedded '_' characters, which are ignored. The embedded '_' are useful for formatting long literals to make them more readable, such as using them as a

thousands separator:

```
123_456.567_8           // 123456.5678
1_2_3_4_5_6_._5_6_7_8   // 123456.5678
1_2_3_4_5_6_._5e-6_     // 123456.5e-6
```

Floats can be followed by one **f**, **F**, **l** or **L** suffix. The **f** or **F** suffix means it is a float, and **l** or **L** means it is an extended.

If a floating literal is followed by **i** or **I**, then it is an *ireal* (imaginary) type.

Examples:

```
0x1.FFFFFFFFFFFFFp1023          // double.max
0x1p-52                         // double.epsilon
1.175494351e-38F                // float.min
6.3i                            // idouble 6.3
6.3fi                           // ifloat 6.3
6.3LI                           // ireal 6.3
```

It is an error if the literal exceeds the range of the type. It is not an error if the literal is rounded to fit into the significant digits of the type.

Complex literals are not tokens, but are assembled from real and imaginary expressions in the semantic analysis:

```
4.5 + 6.2i              // complex number
```

Keywords

Keywords are reserved identifiers.

Keyword:

```
        abstract
        alias
        align
        asm
        assert
        auto

        bit
        body
        break
        byte

        case
        cast
        catch
        cdouble
        cent
        cfloat
        char
```

```
class
const
continue
creal

dchar
debug
default
delegate
delete
deprecated
do
double

else
enum
export
extern

false
final
finally
float
for
foreach
function

goto

idouble
if
ifloat
import
in
inout
int
interface
invariant
ireal
is

long

mixin
module

new
null

out
override

package
pragma
private
protected
public
```

```
real
return

short
static
struct
super
switch
synchronized

template
this
throw
true
try
typedef
typeof

ubyte
ucent
uint
ulong
union
unittest
ushort

version
void
volatile

wchar
while
with
```

Special Token Sequences

Special token sequences are processed by the lexical analyzer, may appear between any other tokens, and do not affect the syntax parsing.

There is currently only one special token sequence, `#line`.

```
SpecialTokenSequence
        # line Integer EndOfLine
        # line Integer Filespec EndOfLine

Filespec
        " Characters "
```

This sets the source line number to *Integer*, and optionally the source file name to *Filespec*, beginning with the next line of source text. The source file and line number is used for printing error messages and for mapping generated code back to the source for the symbolic debugging

output.

For example:

```
int #line 6 "foo\bar"
x;                      // this is now line 6 of file foo\bar
```

Note that the backslash character is not treated specially inside *Filespec* strings.

Modules

```
Module:
        ModuleDeclaration DeclDefs
        DeclDefs

DeclDefs:
        DeclDef
        DeclDef DeclDefs

DeclDef:
        AttributeSpecifier
        ImportDeclaration
        EnumDeclaration
        ClassDeclaration
        InterfaceDeclaration
        AggregateDeclaration
        Declaration
        Constructor
        Destructor
        Invariant
        Unittest
        StaticConstructor
        StaticDestructor
        DebugSpecification
        VersionSpecification
        ;
```

Modules have a one-to-one correspondence with source files. The module name is the file name with the path and extension stripped off.

Modules automatically provide a namespace scope for their contents. Modules superficially resemble classes, but differ in that:

- There's only one instance of each module, and it is statically allocated.
- There is no virtual table.
- Modules do not inherit, they have no super modules, etc.
- Only one module per file.
- Module symbols can be imported.

Modules are always compiled at global scope, and are unaffected by surrounding attributes or other modifiers.

Modules can be grouped together in heirarchies called *packages*.

Module Declaration

The *ModuleDeclaration* sets the name of the module and what package it belongs to. If absent, the module name is taken to be the same name (stripped of path and extension) of the source file name.

```
ModuleDeclaration:
        module ModuleName ;

ModuleName:
        Identifier
        ModuleName . Identifier
```

The *Identifier* preceding the rightmost are the *packages* that the module is in. The packages correspond to directory names in the source file path.

If present, the *ModuleDeclaration* appears syntactically first in the source file, and there can be only one per source file.

Example:

```
module c.stdio;    // this is module stdio in the c package
```

By convention, package and module names are all lower case. This is because those names have a one-to-one correspondence with the operating system's directory and file names, and many file systems are not case sensitive. All lower case package and module names will minimize problems moving projects between dissimilar file systems.

Import Declaration

Rather than text include files, D imports symbols symbolically with the import declaration:

```
ImportDeclaration:
        import ModuleNameList ;

ModuleNameList:
        ModuleName
        ModuleName , ModuleNameList
```

The rightmost *Identifier* becomes the module name. The top level scope in the module is merged with the current scope.

Example:

```
import std.c.stdio;  // import module stdio from the c package
```

```
import foo, bar;     // import modules foo and bar
```

Scope and Modules

Each module forms its own namespace. When a module is imported into another module, by default all its top level declarations are available without qualification. Ambiguities are illegal, and can be resolved by explicitly qualifying the symbol with the module name.

For example, assume the following modules:

```
Module foo
int x = 1;
int y = 2;

Module bar
int y = 3;
int z = 4;
```

then:

```
import foo;
...
q = y;          // sets q to foo.y
```

and:

```
import foo;
int y = 5;
q = y;          // local y overrides foo.y
```

and:

```
import foo;
import bar;
q = y;          // error: foo.y or bar.y?
```

and:

```
import foo;
import bar;
q = bar.y;      // q set to 3
```

If the import is private, such as:

```
module abc;
private import def;
```

then *def* is not searched when another module imports *abc*.

Module Scope Operator

Sometimes, it's necessary to override the usual lexical scoping rules to access a name hidden by a local name. This is done with the global scope operator, which is a leading '.':

```
int x;

int foo(int x)
{
    if (y)
        return x;               // returns foo.x, not global x
    else
        return .x;              // returns global x
}
```

The leading '.' means look up the name at the module scope level.

Static Construction and Destruction

Static constructors are code that gets executed to initialize a module or a class before the main() function gets called. Static destructors are code that gets executed after the main() function returns, and are normally used for releasing system resources.

Order of Static Construction

The order of static initialization is implicitly determined by the *import* declarations in each module. Each module is assumed to depend on any imported modules being statically constructed first. Other than following that rule, there is no imposed order on executing the module static constructors.

Cycles (circular dependencies) in the import declarations are allowed as long as not both of the modules contain static constructors or static destructors. Violation of this rule will result in a runtime exception.

Order of Static Construction within a Module

Within a module, the static construction occurs in the lexical order in which they appear.

Order of Static Destruction

It is defined to be exactly the reverse order that static construction was performed in. Static destructors for individual modules will only be run if the corresponding static constructor successfully completed.

Declarations

Declaration:
        typedef *Decl*
        alias *Decl*
        Decl

Decl:
        StorageClass Decl
        BasicType Declarators ;
        BasicType Declarator FunctionBody

Declarators:
        DeclaratorInitializer
        DeclaratorInitializer , *DeclaratorIdentifierList*

DeclaratorInitializer:
        Declarator
        Declarator **=** *Initializer*

DeclaratorIdentifierList:
        DeclaratorIdentifier
        DeclaratorIdentifier , *DeclaratorIdentifierList*

DeclaratorIdentifier:
        Identifier
        Identifier **=** *Initializer*

BasicType:
        bit
        byte
        ubyte
        short
        ushort
        int
        uint
        long
        ulong
        char
        wchar
        dchar
        float
        double
        real
        ifloat
        idouble
        ireal
        cfloat
        cdouble
        creal
        void
        .IdentifierList
        IdentifierList
        Typeof
        Typeof . *IdentifierList*

BasicType2:
*****
[ ]
[*Expression* **]**
[*Type* **]**
delegate (*ParameterList* **)**
function (*ParameterList* **)**

Declarator:
BasicType2 Declarator
Identifier
(*Declarator* **)**
Identifier DeclaratorSuffixes
(*Declarator* **)** *DeclaratorSuffixes*

DeclaratorSuffixes:
DeclaratorSuffix
DeclaratorSuffix DeclaratorSuffixes

DeclaratorSuffix:
[ ]
[*Expression* **]**
[*Type* **]**
(*ParameterList* **)**

IdentifierList
Identifier
Identifier . IdentifierList
TemplateInstance
TemplateInstance . IdentifierList

Typeof
typeof (*Expression* **)**

StorageClass:
abstract
auto
const
deprecated
final
override
static
synchronized

Type:
BasicType
BasicType Declarator2

Declarator2:
BasicType2 Declarator2
(*Declarator2* **)**
(*Declarator2* **)** *DeclaratorSuffixes*

ParameterList:
Parameter
Paremeter , ParameterList

```
        ...

    Parameter:
        Declarator
        Declarator = AssignExpression
        InOut Declarator
        InOut Declarator = AssignExpression

    InOut:
        in
        out
        inout
```

Declaration Syntax

Declaration syntax generally reads right to left:

```
int x;          // x is an int
int* x;         // x is a pointer to int
int** x;        // x is a pointer to a pointer to int
int[] x;        // x is an array of ints
int*[] x;       // x is an array of pointers to ints
int[]* x;       // x is a pointer to an array of ints
```

Arrays, read left to right:

```
int[3] x;       // x is an array of 3 ints
int[3][5] x;    // x is an array of 5 arrays of 3 ints
int[3]*[5] x;   // x is an array of 5 pointers to arrays of 3 ints
```

Pointers to functions are declared using the **function** keyword:

```
int function(char) x;   // x is a pointer to a function taking a char
argument
                        // and returning an int
int function(char)[] x; // x is an array of pointers to functions
                        // taking a char argument and returning an int
```

C-style array declarations may be used as an alternative:

```
int x[3];       // x is an array of 3 ints
int x[3][5];    // x is an array of 3 arrays of 5 ints
int (*x[5])[3]; // x is an array of 5 pointers to arrays of 3 ints
int (*x)(char); // x is a pointer to a function taking a char argument
                // and returning an int
int (*[] x)(char);      // x is an array of pointers to functions
                        // taking a char argument and returning an int
```

In a declaration declaring multiple declarations, all the declarations must be of the same type:

```
int x,y;        // x and y are ints
int* x,y;       // x and y are pointers to ints
int x,*y;       // error, multiple types
int[] x,y;      // x and y are arrays of ints
int x[],y;      // error, multiple types
```

Type Defining

Strong types can be introduced with the typedef. Strong types are semantically a distinct type to the type checking system, for function overloading, and for the debugger.

```
typedef int myint;

void foo(int x) { . }
void foo(myint m) { . }

.
myint b;
foo(b);         // calls foo(myint)
```

Typedefs can specify a default initializer different from the default initializer of the underlying type:

```
typedef int myint = 7;
myint m;                // initialized to 7
```

Type Aliasing

It's sometimes convenient to use an alias for a type, such as a shorthand for typing out a long, complex type like a pointer to a function. In D, this is done with the alias declaration:

```
alias abc.Foo.bar myint;
```

Aliased types are semantically identical to the types they are aliased to. The debugger cannot distinguish between them, and there is no difference as far as function overloading is concerned. For example:

```
alias int myint;

void foo(int x) { . }
void foo(myint m) { . } error, multiply defined function foo
```

Type aliases are equivalent to the C typedef.

Alias Declarations

A symbol can be declared as an *alias* of another symbol. For example:

```
import string;

alias string.strlen mylen;
...
int len = mylen("hello");       // actually calls string.strlen()
```

The following alias declarations are valid:

```
template Foo2(T) { alias T t; }
alias Foo2!(int) t1;
alias Foo2!(int).t t2;
alias t1.t t3;
alias t2 t4;

t1.t v1;        // v1 is type int
t2 v2;          // v2 is type int
t3 v3;          // v3 is type int
t4 v4;          // v4 is type int
```

Aliased symbols are useful as a shorthand for a long qualified symbol name, or as a way to redirect references from one symbol to another:

```
version (Win32)
{
    alias win32.foo myfoo;
}
version (linux)
{
    alias linux.bar myfoo;
}
```

Aliasing can be used to 'import' a symbol from an import into the current scope:

```
alias string.strlen strlen;
```

Aliases can also 'import' a set of overloaded functions, that can be overloaded with functions in the current scope:

```
class A {
    int foo(int a) { return 1; }
}

class B : A {
    int foo( int a, uint b ) { return 2; }
}

class C : B {
    int foo( int a ) { return 3; }
```

```
        alias B.foo foo;
    }

    class D : C  {
    }


    void test()
    {
        D b = new D();
        int i;

        i = b.foo(1, 2u);   // calls B.foo
        i = b.foo(1);       // calls C.foo
    }
```

Note: Type aliases can sometimes look indistinguishable from alias declarations:

```
    alias foo.bar abc;      // is it a type or a symbol?
```

The distinction is made in the semantic analysis pass.

typeof

Typeof is a way to specify a type based on the type of an expression. For example:

```
    void func(int i)
    {
        typeof(i) j;        // j is of type int
        typeof(3 + 6.0) x;  // x is of type double
        typeof(1)* p;       // p is of type pointer to int
        int[typeof[p]] a;   // a is of type int[int*]

        printf("%d\n", typeof('c').size);   // prints 1
        double c = cast(typeof(1.0))j;      // cast j to double
    }
```

Expression is not evaluated, just the type of it is generated:

```
    void func()
    {   int i = 1;
        typeof(++i) j;      // j is declared to be an int, i is not
incremented
        printf("%d\n", i);  // prints 1
    }
```

There are two special cases: **typeof(this)** will generate the type of what **this** would be in a non-static member function, even if not in a member function. Analogously, **typeof(super)** will generate the type of what **super** would be in a non-static member function.

```
    class A { }
```

```
class B : A
{
    typeof(this) x;     // x is declared to be a B
    typeof(super) y;    // y is declared to be an A
}

struct C
{
    typeof(this) z;     // z is declared to be a C*
    typeof(super) q;    // error, no super struct for C
}

typeof(this) r;         // error, no enclosing struct or class
```

Where *Typeof* is most useful is in writing generic template code.

Types

Basic Data Types

Keyword	Description	Default Init
void	no type	-
bit	single bit	false
byte	signed 8 bits	0
ubyte	unsigned 8 bits	0
short	signed 16 bits	0
ushort	unsigned 16 bits	0
int	signed 32 bits	0
uint	unsigned 32 bits	0
long	signed 64 bits	0L
ulong	unsigned 64 bits	0L
cent	signed 128 bits (reserved for future use)	0
ucent	unsigned 128 bits (reserved for future use)	0
float	32 bit floating point	float.nan
double	64 bit floating point	double.nan

`real`	largest hardware implemented floating point size (**Implementation Note:** 80 bits for Intel CPU's)	real.nan
`ifloat`	imaginary float	float.nan * 1
`idouble`	imaginary double	double.nan *
`ireal`	imaginary real	real.nan * 1.
`cfloat`	a complex number of two float values	float.nan + f
`cdouble`	complex double	double.nan + 1.0i
`creal`	complex real	real.nan + re
`char`	unsigned 8 bit UTF-8	0xFF
`wchar`	unsigned 16 bit UTF-16	0xFFFF
`dchar`	unsigned 32 bit UTF-32	0x0000FFFF

Derived Data Types

pointer

array

function

User Defined Types

alias

typedef

enum

struct

union

class

Pointer Conversions

Casting pointers to non-pointers and vice versa is allowed in D, however, do not do this for any pointers that point to data allocated by the garbage collector.

Implicit Conversions

D has a lot of types, both built in and derived. It would be tedious to require casts for every type conversion, so implicit conversions step in to handle the obvious ones automatically.

A typedef can be implicitly converted to its underlying type, but going the other way requires an explicit conversion. For example:

```
typedef int myint;
int i;
myint m;
i = m;                  // OK
m = i;                  // error
m = cast(myint)i;       // OK
```

Integer Promotions

Integer Promotions are conversions of the following types:

from	to
bit	int
byte	int
ubyte	int
short	int
ushort	int
char	int
wchar	int
dchar	uint

If a typedef or enum has as a base type one of the types in the left column, it is converted to the type in the right column.

Usual Arithmetic Conversions

The usual arithmetic conversions convert operands of binary operators to a common type. The operands must already be of arithmetic types. The following rules are applied in order:

1. Typedefs are converted to their underlying type.
2. If either operand is real, the other operand is converted to real.
3. Else if either operand is double, the other operand is converted to double.
4. Else if either operand is float, the other operand is converted to float.
5. Else the integer promotions are done on each operand, followed by:
6. If both are the same type, no more conversions are done.

7. If both are signed or both are unsigned, the smaller type is converted to the larger.
8. If the signed type is larger than the unsigned type, the unsigned type is converted to the signed type.
9. The signed type is converted to the unsigned type.

Delegates

There are no pointers-to-members in D, but a more useful concept called *delegates* are supported. Delegates are an aggregate of two pieces of data: an object reference and a function pointer. The object reference forms the *this* pointer when the function is called.

Delegates are declared similarly to function pointers, except that the keyword **delegate** takes the place of (*), and the identifier occurs afterwards:

```
int function(int) fp;   // fp is pointer to a function
int delegate(int) dg;   // dg is a delegate to a function
```

The C style syntax for declaring pointers to functions is also supported:

```
int (*fp)(int);         // fp is pointer to a function
```

A delegate is initialized analogously to function pointers:

```
int func(int);
fp = &func;             // fp points to func

class OB
{   int member(int);
}
OB o;
dg = &o.member;         // dg is a delegate to object o and
                        // member function member
```

Delegates cannot be initialized with static member functions or non-member functions.

Delegates are called analogously to function pointers:

```
fp(3);          // call func(3)
dg(3);          // call o.member(3)
```

Properties

Every type and expression has properties that can be queried:

```
int.sizeof      // yields 4
float.nan       // yields the floating point nan (Not A Number) value
```

```
(float).nan     // yields the floating point nan value
(3).sizeof      // yields 4 (because 3 is an int)
2.sizeof        // syntax error, since "2." is a floating point number
int.init        // default initializer for int's
```

Properties for Integral Data Types

```
.init           initializer (0)
.sizeof         size in bytes (equivalent to C's sizeof(type))
.alignof        alignment size
.max            maximum value
.min            minimum value
```

Properties for Floating Point Types

```
.init           initializer (NaN)
.sizeof         size in bytes (equivalent to C's sizeof(type))
.alignof        alignment size
.infinity       infinity value
.nan            NaN value
.dig            number of decimal digits of precision
.epsilon        smallest increment
.mant_dig       number of bits in mantissa
.max_10_exp     maximum exponent as power of 10
.max_exp        maximum exponent as power of 2
.min_10_exp     minimum exponent as power of 10
.min_exp        minimum exponent as power of 2
.max            largest representable value that's not infinity
.min            smallest representable value that's not 0
```

.init Property

.init produces a constant expression that is the default initializer. If applied to a type, it is the default initializer for that type. If applied to a variable or field, it is the default initializer for that variable or field. For example:

```
int a;
int b = 1;
typedef int t = 2;
t c;
t d = cast(t)3;

int.init        // is 0
a.init          // is 0
b.init          // is 1
t.init          // is 2
```

```
c.init          // is 2
d.init          // is 3

struct Foo
{
    int a;
    int b = 7;
}

Foo.a.init      // is 0
Foo.b.init      // is 7
```

Class and Struct Properties

Properties are member functions that can by syntactically treated as if they were fields. Properties can be read from or written to. A property is read by calling a method with no arguments; a property is written by calling a method with its argument being the value it is set to.

A simple property would be:

```
struct Foo
{
    int data() { return m_data; }       // read property

    int data(int value) { return m_data = value; } // write property

  private:
    int m_data;
}
```

To use it:

```
int test()
{
    Foo f;

    f.data = 3;          // same as f.data(3);
    return f.data + 3;   // same as return f.data() + 3;
}
```

The absence of a read method means that the property is write-only. The absence of a write method means that the property is read-only. Multiple write methods can exist; the correct one is selected using the usual function overloading rules.

In all the other respects, these methods are like any other methods. They can be static, have different linkages, be overloaded with methods with multiple parameters, have their address taken, etc.

Note: Properties currently cannot be the lvalue of an *op*=, ++, or -- operator.

Attributes

```
AttributeSpecifier:
    Attribute :
    Attribute DeclDefBlock
    Pragma ;

AttributeElseSpecifier:
    AttributeElse :
    AttributeElse DeclDefBlock
    AttributeElse DeclDefBlock else DeclDefBlock

Attribute:
    LinkageAttribute
    AlignAttribute
    Pragma
    deprecated
    private
    package
    protected
    public
    export
    static
    final
    override
    abstract
    const
    auto

AttributeElse:
    DebugAttribute
    VersionAttribute

DeclDefBlock
    DeclDef
    { }
    { DeclDefs }
```

Attributes are a way to modify one or more declarations. The general forms are:

```
attribute declaration;          affects the declaration

attribute:                      affects all declarations until the next }
    declaration;
    declaration;
    ...

attribute                       affects all declarations in the block
{
    declaration;
    declaration;
    ...
}
```

For attributes with an optional else clause:

```
attribute
    declaration;
else
    declaration;

attribute                       affects all declarations in the block
{
    declaration;
    declaration;
    ...
}
else
{
    declaration;
    declaration;
    ...
}
```

Linkage Attribute

```
LinkageAttribute:
        extern
        extern ( LinkageType )

LinkageType:
        C
        D
        Windows
        Pascal
```

D provides an easy way to call C functions and operating system API functions, as compatibility with both is essential. The *LinkageType* is case sensitive, and is meant to be extensible by the implementation (they are not keywords). **C** and **D** must be supplied, the others are what makes sense for the implementation. **Implementation Note:** for Win32 platforms, **Windows** and **Pascal** should exist.

C function calling conventions are specified by:

```
extern (C):
        int foo();      call foo() with C conventions
```

D conventions are:

```
extern (D):
```

or:

```
extern:
```

Windows API conventions are:

```
extern (Windows):
    void *VirtualAlloc(
    void *lpAddress,
    uint dwSize,
    uint flAllocationType,
    uint flProtect
    );
```

Align Attribute

AlignAttribute:
align
align **(** *Integer* **)**

Specifies the alignment of struct members. **align** by itself sets it to the default, which matches the default member alignment of the companion C compiler. *Integer* specifies the alignment which matches the behavior of the companion C compiler when non-default alignments are used. A value of 1 means that no alignment is done; members are packed together.

Deprecated Attribute

It is often necessary to deprecate a feature in a library, yet retain it for backwards compatibility. Such declarations can be marked as deprecated, which means that the compiler can be set to produce an error if any code refers to deprecated declarations:

```
deprecated
{
        void oldFoo();
}
```

Implementation Note: The compiler should have a switch specifying if deprecated declarations should be compiled with out complaint or not.

Protection Attribute

Protection is an attribute that is one of **private**, **package**, **protected**, **public** or **export**.

Private means that only members of the enclosing class can access the member, or members and functions in the same module as the enclosing class. Private members cannot be overridden.

Private module members are equivalent to **static** declarations in C programs.

Package extends private so that package members can be accessed from code in other modules that are in the same package. This applies to the innermost package only, if a module is in nested packages.

Protected means that only members of the enclosing class or any classes derived from that class, or members and functions in the same module as the enclosing class, can access the member. Protected module members are illegal.

Public means that any code within the executable can access the member.

Export means that any code outside the executable can access the member. Export is analogous to exporting definitions from a DLL.

Const Attribute

const

The **const** attribute declares constants that can be evaluated at compile time. For example:

```
const int foo = 7;

const
{
    double bar = foo + 6;
}
```

Override Attribute

override

The **override** attribute applies to virtual functions. It means that the function must override a function with the same name and parameters in a base class. The override attribute is useful for catching errors when a base class's member function gets its parameters changed, and all derived classes need to have their overriding functions updated.

```
class Foo
{
    int bar();
    int abc(int x);
}

class Foo2 : Foo
{
    override
    {
        int bar(char c);        // error, no bar(char) in Foo
        int abc(int x);         // ok
```

```
    }
}
```

Static Attribute

static

The **static** attribute applies to functions and data. It means that the declaration does not apply to a particular instance of an object, but to the type of the object. In other words, it means there is no **this** reference. **static** is ignored when applied to other declarations.

```
class Foo
{
    static int bar() { return 6; }
    int foobar() { return 7; }
}

...

Foo f = new Foo;
Foo.bar();      // produces 6
Foo.foobar();   // error, no instance of Foo
f.bar();        // produces 6;
f.foobar();     // produces 7;
```

Static functions are never virtual.

Static data has only one instance for the entire program, not once per object.

Static does not have the additional C meaning of being local to a file. Use the **private** attribute in D to achieve that. For example:

```
module foo;
int x = 3;              // x is global
private int y = 4;      // y is local to module foo
```

Static can be applied to constructors and destructors, producing static constructors and static destructors.

Auto Attribute

auto

The auto attribute is used for local variables and for class declarations. For class declarations, the auto attribute creates an *auto* class. For local declarations, **auto** implements the RAII (Resource Acquisition Is Initialization) protocol. This means that the destructor for an object is

automatically called when the auto reference to it goes out of scope. The destructor is called even if the scope is exited via a thrown exception, thus auto is used to guarantee cleanup.

Auto cannot be applied to globals, statics, data members, inout or out parameters. Arrays of autos are not allowed, and auto function return values are not allowed. Assignment to an auto, other than initialization, is not allowed. **Rationale:** These restrictions may get relaxed in the future if a compelling reason to appears.

Pragmas

```
Pragma:
    pragma ( Identifier )
    pragma ( Identifier , ExpressionList )
```

Pragmas are a way to pass special information to the compiler and to add vendor specific extensions to D. Pragmas can be used by themselves terminated with a ';', they can influence a statement, a block of statements, a declaration, or a block of declarations.

```
pragma(ident);          // just by itself

pragma(ident) declaration; // influence one declaration

pragma(ident):          // influence subsequent declarations
        declaration;
        declaration;

pragma(ident)           // influence block of declarations
{       declaration;
        declaration;
}

pragma(ident) statement; // influence one statement

pragma(ident)           // influence block of statements
{       statement;
        statement;
}
```

The kind of pragma it is is determined by the *Identifier*. *ExpressionList* is a comma-separated list of *AssignExpression*s. The *AssignExpression*s must be parsable as expressions, but what they mean semantically is up to the individual pragma semantics.

Predefined Pragmas

All implementations must support these, even if by just ignoring them:

msg

Prints a message while compiling, the *AssignExpression*s must be string literals:

```
pragma(msg, "compiling...");
```

Vendor Specific Pragmas

Vendor specific pragma *Identifier*s can be defined if they are prefixed by the vendor's trademarked name, in a similar manner to version identifiers:

```
pragma(DigitalMars_funky_extension) { ... }
```

Compilers must diagnose an error for unrecognized *Pragma*s, even if they are vendor specific ones. This implies that vendor specific pragmas should be wrapped in version statements:

```
version (DigitalMars)
{
    pragma(DigitalMars_funky_extension) { ... }
}
```

Expressions

C and C++ programmers will find the D expressions very familiar, with a few interesting additions.

Expressions are used to compute values with a resulting type. These values can then be assigned, tested, or ignored. Expressions can also have side effects.

Expression:
AssignExpression
AssignExpression , *Expression*

AssignExpression:
ConditionalExpression
ConditionalExpression = *AssignExpression*
ConditionalExpression += *AssignExpression*
ConditionalExpression -= *AssignExpression*
ConditionalExpression *= *AssignExpression*
ConditionalExpression /= *AssignExpression*
ConditionalExpression %= *AssignExpression*
ConditionalExpression &= *AssignExpression*
ConditionalExpression |= *AssignExpression*
ConditionalExpression ^= *AssignExpression*
ConditionalExpression ~= *AssignExpression*
ConditionalExpression <<= *AssignExpression*
ConditionalExpression >>= *AssignExpression*
ConditionalExpression >>>= *AssignExpression*

```
ConditionalExpression:
        OrOrExpression
        OrOrExpression ? Expression : ConditionalExpression

OrOrExpression:
        AndAndExpression
        OrOrExpression || AndAndExpression

AndAndExpression:
        OrExpression
        AndAndExpression && OrExpression

OrExpression:
        XorExpression
        OrExpression | XorExpression

XorExpression:
        AndExpression
        XorExpression ^ AndExpression

AndExpression:
        EqualExpression
        AndExpression & EqualExpression

EqualExpression:
        RelExpression
        EqualExpression == RelExpression
        EqualExpression != RelExpression
        EqualExpression is RelExpression

RelExpression:
        ShiftExpression
        InExpression
        RelExpression < ShiftExpression
        RelExpression <= ShiftExpression
        RelExpression > ShiftExpression
        RelExpression >= ShiftExpression
        RelExpression !<>= ShiftExpression
        RelExpression !<> ShiftExpression
        RelExpression <> ShiftExpression
        RelExpression <>= ShiftExpression
        RelExpression !> ShiftExpression
        RelExpression !>= ShiftExpression
        RelExpression !< ShiftExpression
        RelExpression !<= ShiftExpression

InExpression:
        RelExpression in ShiftExpression

ShiftExpression:
        AddExpression
        ShiftExpression << AddExpression
        ShiftExpression >> AddExpression
        ShiftExpression >>> AddExpression

AddExpression:
```

```
        MulExpression
        AddExpression + MulExpression
        AddExpression - MulExpression
        AddExpression ~ MulExpression

MulExpression:
        UnaryExpression
        MulExpression * UnaryExpression
        MulExpression / UnaryExpression
        MulExpression % UnaryExpression

UnaryExpression:
        PostfixExpression
        & UnaryExpression
        ++ UnaryExpression
        -- UnaryExpression
        * UnaryExpression
        - UnaryExpression
        + UnaryExpression
        ! UnaryExpression
        ~ UnaryExpression
        delete UnaryExpression
        NewExpression
        cast ( Type ) UnaryExpression
        ( Type ) . Identifier
        ( Expression )

PostfixExpression:
        PrimaryExpression
        PostfixExpression . Identifier
        PostfixExpression ++
        PostfixExpression --
        PostfixExpression ( ArgumentList )
        IndexExpression
        SliceExpression

IndexExpression:
        PostfixExpression [ ArgumentList ]

SliceExpression:
        PostfixExpression [ AssignExpression .. AssignExpression ]

PrimaryExpression:
        Identifier
        .Identifier
        this
        super
        null
        true
        false
        NumericLiteral
        CharacterLiteral
        StringLiteral
        FunctionLiteral
        AssertExpression
        BasicType . Identifier
        typeid ( Type )
```

```
        AssertExpression:
                assert ( Expression )

        ArgumentList:
                AssignExpression
                AssignExpression , ArgumentList

        NewExpression:
                new BasicType Stars [ AssignExpression ] Declarator
                new BasicType Stars ( ArgumentList )
                new BasicType Stars
                new ( ArgumentList ) BasicType Stars [ AssignExpression ]
Declarator
                new ( ArgumentList ) BasicType Stars ( ArgumentList )
                new ( ArgumentList ) BasicType Stars

        Stars
                nothing
                *
                * Stars
```

Evaluation Order

Unless otherwise specified, the implementation is free to evaluate the components of an expression in any order. It is an error to depend on order of evaluation when it is not specified. For example, the following are illegal:

```
i = ++i;
c = a + (a = b);
func(++i, ++i);
```

If the compiler can determine that the result of an expression is illegally dependent on the order of evaluation, it can issue an error (but is not required to). The ability to detect these kinds of errors is a quality of implementation issue.

Expressions

```
        AssignExpression , Expression
```

The left operand of the **,** is evaluated, then the right operand is evaluated. The type of the expression is the type of the right operand, and the result is the result of the right operand.

Assign Expressions

```
        ConditionalExpression = AssignExpression
```

The right operand is implicitly converted to the type of the left operand, and assigned to it. The result type is the type of the lvalue, and the result value is the value of the lvalue after the assignment.

The left operand must be an lvalue.

Assignment Operator Expressions

```
ConditionalExpression += AssignExpression
ConditionalExpression -= AssignExpression
ConditionalExpression *= AssignExpression
ConditionalExpression /= AssignExpression
ConditionalExpression %= AssignExpression
ConditionalExpression &= AssignExpression
ConditionalExpression |= AssignExpression
ConditionalExpression ^= AssignExpression
ConditionalExpression <<= AssignExpression
ConditionalExpression >>= AssignExpression
ConditionalExpression >>>= AssignExpression
```

Assignment operator expressions, such as:

```
a op= b
```

are semantically equivalent to:

```
a = a op b
```

except that operand *a* is only evaluated once.

Conditional Expressions

```
OrOrExpression ? Expression : ConditionalExpression
```

The first expression is converted to bool, and is evaluated. If it is true, then the second expression is evaluated, and its result is the result of the conditional expression. If it is false, then the third expression is evaluated, and its result is the result of the conditional expression. If either the second or third expressions are of type void, then the resulting type is void. Otherwise, the second and third expressions are implicitly converted to a common type which becomes the result type of the conditional expression.

OrOr Expressions

```
OrOrExpression || AndAndExpression
```

The result type of an OrOr expression is bool, unless the right operand has type void, when the result is type void.

The OrOr expression evaluates its left operand. If the left operand, converted to type bool, evaluates to true, then the right operand is not evaluated. If the result type of the OrOr expression is bool then the result of the expression is true. If the left operand is false, then the right operand is evaluated. If the result type of the OrOr expression is bool then the result of the expression is the right operand converted to type bool.

AndAnd Expressions

```
AndAndExpression && OrExpression
```

The result type of an AndAnd expression is bool, unless the right operand has type void, when the result is type void.

The AndAnd expression evaluates its left operand. If the left operand, converted to type bool, evaluates to false, then the right operand is not evaluated. If the result type of the AndAnd expression is bool then the result of the expression is false. If the left operand is true, then the right operand is evaluated. If the result type of the AndAnd expression is bool then the result of the expression is the right operand converted to type bool.

Bitwise Expressions

Bit wise expressions perform a bitwise operation on their operands. Their operands must be integral types. First, the default integral promotions are done. Then, the bitwise operation is done.

Or Expressions

```
OrExpression | XorExpression
```

The operands are OR'd together.

Xor Expressions

```
XorExpression ^ AndExpression
```

The operands are XOR'd together.

And Expressions

```
AndExpression & EqualExpression
```

The operands are AND'd together.

Equality Expressions

```
EqualExpression == RelExpression
EqualExpression != RelExpression
EqualExpression is RelExpression
```

Equality expressions compare the two operands for equality (==) or inequality (!=). The type of the result is bool. The operands go through the usual conversions to bring them to a common type before comparison.

If they are integral values or pointers, equality is defined as the bit pattern of the type matches exactly. Equality for struct objects means the bit patterns of the objects match exactly (the existence of alignment holes in the objects is accounted for, usually by setting them all to 0 upon initialization). Equality for floating point types is more complicated. -0 and +0 compare as equal. If either or both operands are NAN, then both the == and != comparisons return false. Otherwise, the bit patterns are compared for equality.

For complex numbers, equality is defined as equivalent to:

```
x.re == y.re && x.im == y.im
```

and inequality is defined as equivalent to:

```
x.re != y.re || x.im != y.im
```

For class objects, equality is defined as the result of calling Object.eq(). If one or the other or both objects are null, an exception is raised.

For static and dynamic arrays, equality is defined as the lengths of the arrays matching, and all the elements are equal.

Identity Expressions

```
EqualExpression is RelExpression
```

The **is** compares for identity. To compare for not identity, use `!(e1 is e2)`. The type of the result is bool. The operands go through the usual conversions to bring them to a common type before comparison.

For operand types other than class objects, static or dynamic arrays, identity is defined as being the same as equality.

For class objects, identity is defined as the object references are for the same object. Null class

objects can be compared with **is**.

For static and dynamic arrays, identity is defined as referring to the same array elements.

The identity operator **is** cannot be overloaded.

Relational Expressions

```
RelExpression < ShiftExpression
RelExpression <= ShiftExpression
RelExpression > ShiftExpression
RelExpression >= ShiftExpression
RelExpression !<>= ShiftExpression
RelExpression !<> ShiftExpression
RelExpression <> ShiftExpression
RelExpression <>= ShiftExpression
RelExpression !> ShiftExpression
RelExpression !>= ShiftExpression
RelExpression !< ShiftExpression
RelExpression !<= ShiftExpression
```

First, the integral promotions are done on the operands. The result type of a relational expression is bool.

For class objects, the result of Object.cmp() forms the left operand, and 0 forms the right operand. The result of the relational expression (o1 op o2) is:

```
(o1.cmp(o2) op 0)
```

It is an error to compare objects if one is null.

For static and dynamic arrays, the result of the relational op is the result of the operator applied to the first non-equal element of the array. If two arrays compare equal, but are of different lengths, the shorter array compares as "less" than the longer array.

Integer comparisons

Integer comparisons happen when both operands are integral types.

Integer comparison operators

Operator	Relation
<	less
>	greater
<=	less or equal
>=	greater or equal

==	equal
!=	not equal

It is an error to have one operand be signed and the other unsigned for a <, <=, > or >= expression. Use casts to make both operands signed or both operands unsigned.

Floating point comparisons

If one or both operands are floating point, then a floating point comparison is performed.

Useful floating point operations must take into account NAN values. In particular, a relational operator can have NAN operands. The result of a relational operation on float values is less, greater, equal, or unordered (unordered means either or both of the operands is a NAN). That means there are 14 possible comparison conditions to test for:

Floating point comparison operators

Operator	Greater Than	Less Than	Equal	Unordered	Exception	Relation
==	F	F	T	F	no	equal
!=	T	T	F	T	no	unordered, less, or greater
>	T	F	F	F	yes	greater
>=	T	F	T	F	yes	greater or equal
<	F	T	F	F	yes	less
<=	F	T	T	F	yes	less or equal
!<>=	F	F	F	T	no	unordered
<>	T	T	F	F	yes	less or greater
<>=	T	T	T	F	yes	less, equal, or greater
!<=	T	F	F	T	no	unordered or greater
!<	T	F	T	T	no	unordered, greater, or equal
!>=	F	T	F	T	no	unordered or less
!>	F	T	T	T	no	unordered, less, or equal
!<>	F	F	T	T	no	unordered or equal

Notes:

1. For floating point comparison operators, (a !op b) is not the same as !(a op b).
2. "Unordered" means one or both of the operands is a NAN.
3. "Exception" means the Invalid Exception is raised if one of the operands is a NAN.

In Expressions

```
RelExpression in ShiftExpression
```

An associative array can be tested to see if an element is in the array:

```
int foo[char[]];
.
if ("hello" in foo)
        .
```

The **in** expression has the same precedence as the relational expressions <, <=, etc. The return value of the *InExpression* is **null** if the element is not in the array; if it is in the array it is a pointer to the element.

Shift Expressions

```
ShiftExpression << AddExpression
ShiftExpression >> AddExpression
ShiftExpression >>> AddExpression
```

The operands must be integral types, and undergo the usual integral promotions. The result type is the type of the left operand after the promotions. The result value is the result of shifting the bits by the right operand's value.

<< is a left shift. >> is a signed right shift. >>> is an unsigned right shift.

It's illegal to shift by more bits than the size of the quantity being shifted:

```
int c;
c << 33;        error
```

Add Expressions

```
AddExpression + MulExpression
AddExpression - MulExpression
AddExpression ~ MulExpression
```

If the operands are of integral types, they undergo integral promotions, and then are brought to a common type using the usual arithmetic conversions.

If either operand is a floating point type, the other is implicitly converted to floating point and they are brought to a common type via the usual arithmetic conversions.

If the operator is + or -, and the first operand is a pointer, and the second is an integral type, the

resulting type is the type of the first operand, and the resulting value is the pointer plus (or minus) the second operand multiplied by the size of the type pointed to by the first operand.

If it is a pointer to a bit, the second operand is divided by 8 and added to the pointer. It is illegal if the second operand modulo 8 is non-zero.

```
bit* p;
p += 1;         // error, 1%8 is non-zero
p += 8;         // ok
```

If the second operand is a pointer, and the first is an integral type, and the operator is +, the operands are reversed and the pointer arithmetic just described is applied.

Add expressions for floating point operands are not associative.

Mul Expressions

```
MulExpression * UnaryExpression
MulExpression / UnaryExpression
MulExpression % UnaryExpression
```

The operands must be arithmetic types. They undergo integral promotions, and then are brought to a common type using the usual arithmetic conversions.

For integral operands, the *****, /, and **%** correspond to multiply, divide, and modulus operations. For multiply, overflows are ignored and simply chopped to fit into the integral type. If the right operand of divide or modulus operators is 0, a DivideByZeroException is thrown.

For floating point operands, the operations correspond to the IEEE 754 floating point equivalents. The modulus operator only works with reals, it is illegal to use it with imaginary or complex operands.

Mul expressions for floating point operands are not associative.

Unary Expressions

```
& UnaryExpression
++ UnaryExpression
-- UnaryExpression
* UnaryExpression
- UnaryExpression
+ UnaryExpression
! UnaryExpression
~ UnaryExpression
delete UnaryExpression
NewExpression
cast ( Type ) UnaryExpression
( Type ) . Identifier
( Expression )
```

New Expressions

New expressions are used to allocate memory on the garbage collected heap (default) or using a class specific allocator.

To allocate multidimensional arrays, the declaration reads in the same order as the prefix array declaration order.

```
char[][] foo;   // dynamic array of strings
...
foo = new char[][30];   // allocate 30 arrays of strings
```

Cast Expressions

In C and C++, cast expressions are of the form:

```
(type) unaryexpression
```

There is an ambiguity in the grammar, however. Consider:

```
(foo) - p;
```

Is this a cast of a dereference of negated p to type foo, or is it p being subtracted from foo? This cannot be resolved without looking up foo in the symbol table to see if it is a type or a variable. But D's design goal is to have the syntax be context free - it needs to be able to parse the syntax without reference to the symbol table. So, in order to distinguish a cast from a parenthesized subexpression, a different syntax is necessary.

C++ does this by introducing:

```
dynamic_cast<type>(expression)
```

which is ugly and clumsy to type. D introduces the **cast** keyword:

```
cast(foo) -p;   // cast (-p) to type foo
(foo) - p;      // subtract p from foo
```

cast has the nice characteristic that it is easy to do a textual search for it, and takes some of the burden off of the relentlessly overloaded () operator.

D differs from C/C++ in another aspect of casts. Any casting of a class reference to a derived class reference is done with a runtime check to make sure it really is a proper downcast. This means that it is equivalent to the behavior of the dynamic_cast operator in C++.

```
class A { ... }
class B : A { ... }

void test(A a, B b)
{
```

```
    B bx = a;          error, need cast
    B bx = cast(B) a;  bx is null if a is not a B
    A ax = b;          no cast needed
    A ax = cast(A) b;  no runtime check needed for upcast
}
```

In order to determine if an object `o` is an instance of a class `B` use a cast:

```
if (cast(B) o)
{
    // o is an instance of B
}
else
{
    // o is not an instance of B
}
```

Postfix Expressions

```
PostfixExpression . Identifier
PostfixExpression -> Identifier
PostfixExpression ++
PostfixExpression --
PostfixExpression ( ArgumentList )
PostfixExpression [ ArgumentList ]
PostfixExpression [ AssignExpression .. AssignExpression ]
```

Index Expressions

```
PostfixExpression [ ArgumentList ]
```

PostfixExpression is evaluated. if *PostfixExpression* is an expression of type static array or dynamic array, the variable **length** is declared and set to be the length of the array. A new declaration scope is created for the evaluation of the *ArgumentList* and **length** appears in that scope only.

Slice Expressions

```
PostfixExpression [ AssignExpression .. AssignExpression ]
```

PostfixExpression is evaluated. if *PostfixExpression* is an expression of type static array or dynamic array, the variable **length** is declared and set to be the length of the array. A new declaration scope is created for the evaluation of the *AssignExpression*..*AssignExpression* and

length appears in that scope only.

The first *AssignExpression* is taken to be the inclusive lower bound of the slice, and the second *AssignExpression* is the exclusive upper bound. The result of the expression is a slice of the *PostfixExpression* array.

Primary Expressions

```
Identifier
.Identifier
this
super
null
true
false
NumericLiteral
CharacterLiteral
StringLiteral
FunctionLiteral
AssertExpression
BasicType . Identifier
typeid ( Type )
```

.Identifier

Identifier is looked up at module scope, rather than the current lexically nested scope.

this

Within a non-static member function, **this** resolves to a reference to the object that called the function. If a member function is called with an explicit reference to **typeof(this)**, a non-virtual call is made:

```
class A
{
    char get() { return 'A'; }

    char foo() { return typeof(this).get(); }
    char bar() { return this.get(); }
}

class B : A
{
    char get() { return 'B'; }
}

void main()
{
    B b = new B();
```

```
        b.foo();            // returns 'A'
        b.bar();            // returns 'B'
    }
```

super

Within a non-static member function, **super** resolves to a reference to the object that called the function, cast to its base class. It is an error if there is no base class. **super** is not allowed in struct member functions. If a member function is called with an explicit reference to **super**, a non-virtual call is made.

null

The keyword **null** represents the null pointer value; technically it is of type (void *). It can be implicitly cast to any pointer type. The integer 0 cannot be cast to the null pointer. Nulls are also used for empty arrays.

true, false

These are of type **bit** and resolve to values 1 and 0, respectively.

Character Literals

Character literals are single characters and resolve to one of type **char**, **wchar**, or **dchar**. If the literal is a \u escape sequence, it resolves to type **wchar**. If the literal is a \U escape sequence, it resolves to type **dchar**. Otherwise, it resolves to the type with the smallest size it will fit into.

Function Literals

```
FunctionLiteral
        function FunctionBody
        function ( ParameterList ) FunctionBody
        function Type ( ParameterList ) FunctionBody
        delegate FunctionBody
        delegate ( ParameterList ) FunctionBody
        delegate Type ( ParameterList ) FunctionBody
```

*FunctionLiteral*s enable embedding anonymous functions and anonymous delegates directly into expressions. *Type* is the return type of the function or delegate, if omitted it defaults to **void**. **(** *ParameterList* **)** forms the parameters to the function. If omitted it defaults to the empty parameter list **()**. The type of a function literal is pointer to function or pointer to delegate.

For example:

```
int function(char c) fp;        // declare pointer to a function
```

```
void test()
{
    static int foo(char c) { return 6; }

    fp = &foo;
}
```

is exactly equivalent to:

```
int function(char c) fp;

void test()
{
    fp = function int(char c) { return 6;} ;
}
```

And:

```
int abc(int delegate(long i));

void test()
{   int b = 3;
    int foo(long c) { return 6 + b; }

    abc(&foo);
}
```

is exactly equivalent to:

```
int abc(int delegate(long i));

void test()
{   int b = 3;

    abc( delegate int(long c) { return 6 + b; } );
}
```

Anonymous delegates can behave like arbitrary statement literals. For example, here an arbitrary statement is executed by a loop:

```
double test()
{   double d = 7.6;
    float f = 2.3;

    void loop(int k, int j, void delegate() statement)
    {
        for (int i = k; i < j; i++)
        {
            statement();
        }
    }
```

```
        loop(5, 100, delegate { d += 1; } );
        loop(3, 10,  delegate { f += 1; } );

        return d + f;
    }
```

When comparing with nested functions, the **function** form is analogous to static or non-nested functions, and the **delegate** form is analogous to non-static nested functions. In other words, a delegate literal can access stack variables in its enclosing function, a function literal cannot.

Assert Expressions

AssertExpression:
    assert **(** *Expression* **)**

Asserts evaluate the *expression*. If the result is false, an AssertError is thrown. If the result is true, then no exception is thrown. It is an error if the *expression* contains any side effects that the program depends on. The compiler may optionally not evaluate assert expressions at all. The result type of an assert expression is `void`. Asserts are a fundamental part of the Contract Programming support in D.

Typeid Expressions

TypeidExpression:
    typeid **(** *Type* **)**

Returns an instance of class **TypeInfo** corresponding to *Type*.

Statements

C and C++ programmers will find the D statements very familiar, with a few interesting additions.

Statement:
    LabeledStatement
    BlockStatement
    ExpressionStatement
    DeclarationStatement
    IfStatement
    DebugStatement
    VersionStatement
    WhileStatement
    DoWhileStatement
    ForStatement
    ForeachStatement

Labelled Statements

Statements can be labelled. A label is an identifier that precedes a statement.

```
LabelledStatement:
        Identifier ':' Statement
```

Any statement can be labelled, including empty statements, and so can serve as the target of a goto statement. Labelled statements can also serve as the target of a break or continue statement.

Labels are in a name space independent of declarations, variables, types, etc. Even so, labels cannot have the same name as local declarations. The label name space is the body of the function they appear in. Label name spaces do not nest, i.e. a label inside a block statement is accessible from outside that block.

Block Statement

A block statement is a sequence of statements enclosed by { }. The statements are executed in lexical order.

```
BlockStatement:
        { }
        { StatementList }

StatementList:
        Statement
        Statement StatementList
```

A block statement introduces a new scope for local symbols. A local symbol's name, however, must be unique within the function.

```
void func1(int x)
{   int x;      // illegal, x is multiply defined in function scope
}

void func2()
{
    int x;

    {   int x;  // illegal, x is multiply defined in function scope
    }
}

void func3()
{
    {   int x;
    }
    {   int x;  // illegal, x is multiply defined in function scope
    }
}

void func4()
{
    {   int x;
    }
    {   x++;    // illegal, x is undefined
    }
}
```

The idea is to avoid bugs in complex functions caused by scoped declarations inadvertantly hiding previous ones. Local names should all be unique within a function.

Expression Statement

The expression is evaluated.

```
ExpressionStatement:
        Expression ;
```

Expressions that have no affect, like (x + x), are illegal in expression statements.

Declaration Statement

Declaration statements declare and initialize variables.

```
DeclarationStatement:
        Type IdentifierList ;

IdentifierList:
```

```
        Variable
        Variable , IdentifierList

Variable:
        Identifier
        Identifier = AssignmentExpression
```

If no *AssignmentExpression* is there to initialize the variable, it is initialized to the default value for its type.

If Statement

If statements provide simple conditional execution of statements.

```
IfStatement:
        if ( Expression ) Statement
        if ( Expression ) Statement else Statement
```

Expression is evaluated and must have a type that can be converted to a boolean. If it's true the if statement is transferred to, else the else statement is transferred to.

The 'dangling else' parsing problem is solved by associating the else with the nearest if statement.

While Statement

While statements implement simple loops.

```
WhileStatement:
        while ( Expression ) Statement
```

Expression is evaluated and must have a type that can be converted to a boolean. If it's true the statement is executed. After the statement is executed, the *Expression* is evaluated again, and if true the statement is executed again. This continues until the *Expression* evaluates to false.

A break statement will exit the loop. A continue statement will transfer directly to evaluating *Expression* again.

Do-While Statement

Do-While statements implement simple loops.

```
DoStatement:
```

```
do Statement  while ( Expression )
```

Statement is executed. Then *Expression* is evaluated and must have a type that can be converted to a boolean. If it's true the loop is iterated again. This continues until the *Expression* evaluates to false.

A break statement will exit the loop. A continue statement will transfer directly to evaluationg *Expression* again.

For Statement

For statements implement loops with initialization, test, and increment clauses.

```
ForStatement:
        for (Initialize; Test; Increment) Statement

Initialize:
        empty
        Expression
        Declaration

Test:
        empty
        Expression

Increment:
        empty
        Expression
```

Initializer is executed. *Test* is evaluated and must have a type that can be converted to a boolean. If it's true the statement is executed. After the statement is executed, the *Increment* is executed. Then *Test* is evaluated again, and if true the statement is executed again. This continues until the *Test* evaluates to false.

A break statement will exit the loop. A continue statement will transfer directly to the *Increment*.

If *Initializer* declares a variable, that variable's scope extends through the end of *Statement*. For example:

```
for (int i = 0; i < 10; i++)
        foo(i);
```

is equivalent to:

```
{   int i;
    for (i = 0; i < 10; i++)
        foo(i);
}
```

Function bodies cannot be empty:

```
for (int i = 0; i < 10; i++)
        ;       // illegal
```

Use instead:

```
for (int i = 0; i < 10; i++)
{
}
```

The *Initializer* may be omitted. *Test* may also be omitted, and if so, it is treated as if it evaluated to true.

Foreach Statement

A foreach statement loops over the contents of an aggregate.

```
ForeachStatement:
        foreach (ForeachTypeList; Expression) Statement

ForeachTypeList:
        ForeachType
        ForeachType , ForeachTypeList

ForeachType:
        inout Type Identifier
        Type Identifier
```

Expression is evaluated. It must evaluate to an aggregate expression of type static array, dynamic array, associative array, struct, or class. The *Statement* is executed, once for each element of the aggregate expression. At the start of each iteration, the variables declared by the *ForeachTypeList* are set to be a copy of the contents of the aggregate. If the variable is **inout**, it is a reference to the contents of that aggregate.

If the aggregate expression is a static or dynamic array, there can be one or two variables declared. If one, then the variable is said to be the *value* set to the elements of the array, one by one. The type of the variable must match the type of the array contents, except for the special cases outlined below. If there are two variables declared, the first is said to be the *index* and the second is said to be the *value*. The *index* must be of **int** or **uint** type, it cannot be *inout*, and it is set to be the index of the array element.

```
char[] a;
...
foreach (int i, char c; a)
{
    printf("a[%d] = '%c'\n", i, c);
}
```

If the aggregate expression is a static or dynamic array of **char**s, **wchar**s, or **dchar**s, then the *Type* of the *value* can be any of **char**, **wchar**, or **dchar**. In this manner any UTF array can be decoded into any UTF type:

```
char[] a = "\xE2\x89\xA0";      // \u2260 encoded as 3 UTF-8 bytes

foreach (dchar c; a)
{
    printf("a[] = %x\n", c);    // prints 'a[] = 2260'
}

dchar[] b = "\u2260";

foreach (char c; b)
{
    printf("%x, ", c);  // prints 'e2, 89, a0'
}
```

If the aggregate expression is an associative array, there can be one or two variables declared. If one, then the variable is said to be the *value* set to the elements of the array, one by one. The type of the variable must match the type of the array contents. If there are two variables declared, the first is said to be the *index* and the second is said to be the *value*. The *index* must be of the same type as the indexing type of the associative array. It cannot be *inout*, and it is set to be the index of the array element.

```
double[char[]] a;       // index type is char[], value type is double
...
foreach (char[] s, double d; a)
{
    printf("a['%.*s'] = %g\n", s, d);
}
```

If the aggregate expression is a static or dynamic array, the elements are iterated over starting at index 0 and continuing to the maximum of the array. If it is an associative array, the order of the elements is undefined. If it is a struct or class object, it is defined by the special *opApply* member function.

If the aggregate is a struct or a class object, that struct or class must have an *opApply* function with the type:

```
int opApply(int delegate(inout Type [, ...]) dg);
```

where *Type* matches the *Type* used in the foreach declaration of *Identifier*. Multiple *ForeachType*s correspond with multiple *Type*'s in the delegate type passed to **opApply**. There can be multiple **opApply** functions, one is selected by matching the type of *dg* to the *ForeachType*s of the *ForeachStatement*. The body of the *opApply* function iterates over the elements it aggregates, passing them each to the *dg* function. If the *dg* returns 0, then **opApply** goes on to the next element. If the *dg* returns a nonzero value, **opApply** must cease iterating and return that value. Otherwise, after done iterating across all the elements, **opApply** will return 0.

For example, consider a class that is a container for two elements:

```
class Foo
{
    uint array[2];

    int opApply(int delegate(inout uint) dg)
    {   int result = 0;

        for (int i = 0; i < array.length; i++)
        {
            result = dg(array[i]);
            if (result)
                break;
        }
        return result;
    }
}
```

An example using this might be:

```
void test()
{
    Foo a = new Foo();

    a.array[0] = 73;
    a.array[1] = 82;

    foreach (uint u; a)
    {
        printf("%d\n", u);
    }
}
```

which would print:

```
73
82
```

Aggregates can be string literals, which can be accessed as char, wchar, or dchar arrays:

```
void test()
{
    foreach (char c; "ab")
    {
        printf("'%c'\n", c);
    }
    foreach (wchar w; "xy")
    {
        wprintf("'%c'\n", w);
    }
}
```

which would print:

```
'a'
'b'
'x'
'y'
```

inout can be used to update the original elements:

```
void test()
{
    static uint[2] a = [7, 8];

    foreach (inout uint u; a)
    {
        u++;
    }
    foreach (uint u; a)
    {
        printf("%d\n", u);
    }
}
```

which would print:

```
8
9
```

The aggregate itself must not be resized, reallocated, free'd, reassigned or destructed while the foreach is iterating over the elements.

```
int[] a;
int[] b;
foreach (int i; a)
{
    a = null;           // error
    a.length += 10;     // error
    a = b;              // error
}
a = null;               // ok
```

A *BreakStatement* in the body of the foreach will exit the foreach, a *ContinueStatement* will immediately start the next iteration.

Switch Statement

A switch statement goes to one of a collection of case statements depending on the value of the switch expression.

```
SwitchStatement:
        switch ( Expression ) BlockStatement

CaseStatement:
        case ExpressionList : Statement

DefaultStatement:
        default: Statement
```

Expression is evaluated. The result type T must be of integral type or char[] or wchar[]. The result is compared against each of the case expressions. If there is a match, the corresponding case statement is transferred to.

The case expressions, *ExpressionList*, are a comma separated list of expressions.

If none of the case expressions match, and there is a default statement, the default statement is transferred to.

If none of the case expressions match, and there is not a default statement, a SwitchError is thrown. The reason for this is to catch the common programming error of adding a new value to an enum, but failing to account for the extra value in switch statements. This behavior is unlike C or C++.

The case expressions must all evaluate to a constant value or array, and be implicitly convertible to the type T of the switch *Expression*.

Case expressions must all evaluate to distinct values. There may not be two or more default statements.

Case statements and default statements associated with the switch can be nested within block statements; they do not have to be in the outermost block. For example, this is allowed:

```
switch (i)
{
    case 1:
    {
        case 2:
    }
        break;
}
```

Like in C and C++, case statements 'fall through' to subsequent case values. A break statement will exit the switch *BlockStatement*. For example:

```
switch (i)
{
    case 1:
        x = 3;
    case 2:
        x = 4;
        break;

    case 3,4,5:
        x = 5;
```

```
                    break;
            }
```

will set x to 4 if i is 1.

Note: Unlike C and C++, strings can be used in switch expressions. For example:

```
char[] name;
...
switch (name)
{
    case "fred":
    case "sally":
        ...
}
```

For applications like command line switch processing, this can lead to much more straightforward code, being clearer and less error prone. Both ascii and wchar strings are allowed.

Implementation Note: The compiler's code generator may assume that the case statements are sorted by frequency of use, with the most frequent appearing first and the least frequent last. Although this is irrelevant as far as program correctness is concerned, it is of performance interest.

Continue Statement

A continue aborts the current iteration of its enclosing loop statement, and starts the next iteration.

```
ContinueStatement:
        continue;
        continue Identifier ;
```

continue executes the next iteration of its innermost enclosing while, for, or do loop. The increment clause is executed.

If continue is followed by *Identifier*, the *Identifier* must be the label of an enclosing while, for, or do loop, and the next iteration of that loop is executed. It is an error if there is no such statement.

Any intervening finally clauses are executed, and any intervening synchronization objects are released.

Note: If a finally clause executes a return, throw, or goto out of the finally clause, the continue target is never reached.

Break Statement

A break exits the enclosing statement.

```
BreakStatement:
        break;
        break Identifier ;
```

break exits the innermost enclosing while, for, do, or switch statement, resuming execution at the statement following it.

If break is followed by *Identifier*, the *Identifier* must be the label of an enclosing while, for, do or switch statement, and that statement is exited. It is an error if there is no such statement.

Any intervening finally clauses are executed, and any intervening synchronization objects are released.

Note: If a finally clause executes a return, throw, or goto out of the finally clause, the break target is never reached.

Return Statement

A return exits the current function and supplies its return value.

```
ReturnStatement:
        return;
        return Expression ;
```

Expression is required if the function specifies a return type that is not void. The *Expression* is implicitly converted to the function return type.

At least one return statement is required if the function specifies a return type that is not void.

Expression is allowed even if the function specifies a **void** return type. The *Expression* will be evaluated, but nothing will be returned.

Before the function actually returns, any enclosing finally clauses are executed, and any enclosing synchronization objects are released.

The function will not return if any enclosing finally clause does a return, goto or throw that exits the finally clause.

If there is an out postcondition (see Contract Programming), that postcondition is executed after the *Expression* is evaluated and before the function actually returns.

Goto Statement

A goto transfers to the statement labelled with *Identifier*.

```
GotoStatement:
        goto Identifier ;
        goto default ;
        goto case ;
        goto case Expression ;
```

The second form, `goto default;`, transfers to the innermost *DefaultStatement* of an enclosing *SwitchStatement*.

The third form, `goto case;`, transfers to the next *CaseStatement* of the innermost enclosing *SwitchStatement*.

The fourth form, `goto case Expression;`, transfers to the *CaseStatement* of the innermost enclosing *SwitchStatement* with a matching *Expression*.

Any intervening finally clauses are executed, along with releasing any intervening synchronization mutexes.

It is illegal for a goto to be used to skip initializations.

With Statement

The with statement is a way to simplify repeated references to the same object.

```
WithStatement:
        with ( Expression ) BlockStatement
        with ( Symbol ) BlockStatement
        with ( TemplateInstance ) BlockStatement
```

where *Expression* evaluates to a class instance or struct reference. Within the with body the referenced object is searched first for identifier symbols. The *WithStatement*

```
with (expression)
{
    ...
    ident;
}
```

is semantically equivalent to:

```
{
    Object tmp;
    tmp = expression;
    ...
    tmp.ident;
```

```
}
```

Note that `expression` only gets evaluated once. The with statement does not change what **this** or **super** refer to.

For *Symbol* which is a scope or *TemplateInstance*, the corresponding scope is searched when looking up symbols. For example:

```
struct Foo
{
    typedef int Y;
}
...
Y y;            // error, Y undefined
with (Foo)
{
    Y y;        // same as Foo.Y y;
}
```

Synchronize Statement

The synchronize statement wraps a statement with critical section to synchronize access among multiple threads.

SynchronizeStatement:
        synchronized *Statement*
        synchronized **(** *Expression* **)** *Statement*

synchronized allows only one thread at a time to execute *Statement*.

synchronized (*Expression*), where *Expression* evaluates to an Object reference, allows only one thread at a time to use that Object to execute the *Statement*. If *Expression* is an instance of an *Interface*, it is cast to an *Object*.

The synchronization gets released even if *Statement* terminates with an exception, goto, or return.

Example:

```
synchronized { ... }
```

This implements a standard critical section.

Try Statement

Exception handling is done with the try-catch-finally statement.

```
TryStatement:
        try BlockStatement Catches
        try BlockStatement Catches finally BlockStatement
        try BlockStatement finally BlockStatement

Catches:
        LastCatch
        Catch
        Catch Catches

LastCatch:
        catch BlockStatement

Catch:
        catch ( Parameter ) BlockStatement
```

Parameter declares a variable v of type T, where T is Object or derived from Object. v is initialized by the throw expression if T is of the same type or a base class of the throw expression. The catch clause will be executed if the exception object is of type T or derived from T.

If just type T is given and no variable v, then the catch clause is still executed.

It is an error if any *Catch Parameter* type T1 hides a subsequent *Catch* with type T2, i.e. it is an error if T1 is the same type as or a base class of T2.

LastCatch catches all exceptions.

Throw Statement

Throw an exception.

```
ThrowStatement:
        throw Expression ;
```

Expression is evaluated and must be an Object reference. The Object reference is thrown as an exception.

Volatile Statement

No code motion occurs across volatile statement boundaries.

```
VolatileStatement:
```

```
volatile Statement
```

Statement is evaluated. Memory writes occurring before the *Statement* are performed before any reads within or after the *Statement*. Memory reads occurring after the *Statement* occur after any writes before or within *Statement* are completed.

A volatile statement does not guarantee atomicity. For that, use synchronized statements.

Asm Statement

Inline assembler is supported with the asm statement:

```
AsmStatement:
        asm { }
        asm { AsmInstructionList }

AsmInstructionList:
        AsmInstruction ;
        AsmInstruction ; AsmInstructionList
```

An asm statement enables the direct use of assembly language instructions. This makes it easy to obtain direct access to special CPU features without resorting to an external assembler. The D compiler will take care of the function calling conventions, stack setup, etc.

The format of the instructions is, of course, highly dependent on the native instruction set of the target CPU, and so is implementation defined. But, the format will follow the following conventions:

- It must use the same tokens as the D language uses.
- The comment form must match the D language comments.
- Asm instructions are terminated by a ;, not by an end of line.

These rules exist to ensure that D source code can be tokenized independently of syntactic or semantic analysis.

For example, for the Intel Pentium:

```
int x = 3;
asm
{
    mov EAX,x;          // load x and put it in register EAX
}
```

Inline assembler can be used to access hardware directly:

```
int gethardware()
{
    asm
```

```
        {
                mov EAX, dword ptr 0x1234;
        }
}
```

For some D implementations, such as a translator from D to C, an inline assembler makes no sense, and need not be implemented. The version statement can be used to account for this:

```
version (InlineAsm)
{
    asm
    {
        ...
    }
}
else
{
    ... some workaround ...
}
```

Arrays

There are four kinds of arrays:

int* p;	Pointers to data
int[3] s;	Static arrays
int[] a;	Dynamic arrays
int[char[]] x;	Associative arrays

Pointers

```
int* p;
```

These are simple pointers to data, analogous to C pointers. Pointers are provided for interfacing with C and for specialized systems work. There is no length associated with it, and so there is no way for the compiler or runtime to do bounds checking, etc., on it. Most conventional uses for pointers can be replaced with dynamic arrays, `out` and `inout` parameters, and reference types.

Static Arrays

```
int[3] s;
```

These are analogous to C arrays. Static arrays are distinguished by having a length fixed at compile time.

Dynamic Arrays

```
int[] a;
```

Dynamic arrays consist of a length and a pointer to the array data. Multiple dynamic arrays can share all or parts of the array data.

Array Declarations

There are two ways to declare arrays, prefix and postfix. The prefix form is the preferred method, especially for non-trivial types.

Prefix Array Declarations

Prefix declarations appear before the identifier being declared and read right to left, so:

```
int[] a;        // dynamic array of ints
int[4][3] b;    // array of 3 arrays of 4 ints each
int[][5] c;     // array of 5 dynamic arrays of ints.
int*[]*[3] d;   // array of 3 pointers to dynamic arrays of pointers
to ints
int[]* e;       // pointer to dynamic array of ints
```

Postfix Array Declarations

Postfix declarations appear after the identifier being declared and read left to right. Each group lists equivalent declarations:

```
// dynamic array of ints
int[] a;
int a[];

// array of 3 arrays of 4 ints each
int[4][3] b;
int[4] b[3];
int b[3][4];

// array of 5 dynamic arrays of ints.
int[][5] c;
int[] c[5];
int c[5][];

// array of 3 pointers to dynamic arrays of pointers to ints
int*[]*[3] d;
int*[]* d[3];
```

```
int* (*d[3])[];

// pointer to dynamic array of ints
int[]* e;
int (*e[]);
```

Rationale: The postfix form matches the way arrays are declared in C and C++, and supporting this form provides an easy migration path for programmers used to it.

Usage

There are two broad kinds of operations to do on an array - affecting the handle to the array, and affecting the contents of the array. C only has operators to affect the handle. In D, both are accessible.

The handle to an array is specified by naming the array, as in p, s or a:

```
int* p;
int[3] s;
int[] a;

int* q;
int[3] t;
int[] b;

p = q;          p points to the same thing q does.
p = s;          p points to the first element of the array s.
p = a;          p points to the first element of the array a.

s = ...;        error, since s is a compiled in static
                reference to an array.

a = p;          error, since the length of the array pointed
                to by p is unknown
a = s;          a is initialized to point to the s array
a = b;          a points to the same array as b does
```

Slicing

Slicing an array means to specify a subarray of it. An array slice does not copy the data, it is only another reference to it. For example:

```
int[10] a;      // declare array of 10 ints
int[] b;

b = a[1..3];    // a[1..3] is a 2 element array consisting of
                // a[1] and a[2]
foo(b[1]);      // equivalent to foo(0)
a[2] = 3;
foo(b[1]);      // equivalent to foo(3)
```

The [] is shorthand for a slice of the entire array. For example, the assignments to b:

```
int[10] a;
int[] b;

b = a;
b = a[];
b = a[0 .. a.length];
```

are all semantically equivalent.

Slicing is not only handy for referring to parts of other arrays, but for converting pointers into bounds-checked arrays:

```
int* p;
int[] b = p[0..8];
```

Slicing for bit arrays is only allowed if the slice's lower bound falls on a byte boundary:

```
bit[] b;
...
b[0..8];        // ok
b[8..16];       // ok
b[8..17];       // ok
b[1..16];       // error, lower bound is not on a byte boundary
```

Misaligned bit array slices will cause an ArrayBoundsError exception to be thrown at runtime.

Array Copying

When the slice operator appears as the lvalue of an assignment expression, it means that the contents of the array are the target of the assignment rather than a reference to the array. Array copying happens when the lvalue is a slice, and the rvalue is an array of or pointer to the same type.

```
int[3] s;
int[3] t;

s[] = t;                the 3 elements of t[3] are copied into s[3]
s[] = t[];              the 3 elements of t[3] are copied into s[3]
s[1..2] = t[0..1];      same as s[1] = t[0]
s[0..2] = t[1..3];      same as s[0] = t[1], s[1] = t[2]
s[0..4] = t[0..4];      error, only 3 elements in s
s[0..2] = t;            error, different lengths for lvalue and rvalue
```

Overlapping copies are an error:

```
s[0..2] = s[1..3];      error, overlapping copy
```

```
s[1..3] = s[0..2];      error, overlapping copy
```

Disallowing overlapping makes it possible for more aggressive parallel code optimizations than possible with the serial semantics of C.

Array Setting

If a slice operator appears as the lvalue of an assignment expression, and the type of the rvalue is the same as the element type of the lvalue, then the lvalue's array contents are set to the rvalue.

```
int[3] s;
int* p;

s[] = 3;                same as s[0] = 3, s[1] = 3, s[2] = 3
p[0..2] = 3;            same as p[0] = 3, p[1] = 3
```

Array Concatenation

The binary operator ~ is the *cat* operator. It is used to concatenate arrays:

```
int[] a;
int[] b;
int[] c;

a = b ~ c;      Create an array from the concatenation of the
                b and c arrays
```

Many languages overload the + operator to mean concatenation. This confusingly leads to, does:

```
"10" + 3
```

produce the number 13 or the string "103" as the result? It isn't obvious, and the language designers wind up carefully writing rules to disambiguate it - rules that get incorrectly implemented, overlooked, forgotten, and ignored. It's much better to have + mean addition, and a separate operator to be array concatenation.

Similarly, the ~= operator means append, as in:

```
a ~= b;         a becomes the concatenation of a and b
```

Concatenation always creates a copy of its operands, even if one of the operands is a 0 length array, so:

```
a = b                   a refers to b
a = b ~ c[0..0]         a refers to a copy of b
```

Array Operations

Note: Array operations are not implemented.

In general, (a[n..m] *op* e) is defined as:

```
for (i = n; i < m; i++)
    a[i] op e;
```

So, for the expression:

```
a[] = b[] + 3;
```

the result is equivalent to:

```
for (i = 0; i < a.length; i++)
    a[i] = b[i] + 3;
```

When more than one [] operator appears in an expression, the range represented by all must match.

```
a[1..3] = b[] + 3;      error, 2 elements not same as 3 elements
```

Examples:

```
int[3] abc;                     // static array of 3 ints
int[] def = [ 1, 2, 3 ];        // dynamic array of 3 ints

void dibb(int *array)
{
        array[2];               // means same thing as *(array + 2)
        *(array + 2);           // get 2nd element
}

void diss(int[] array)
{
        array[2];               // ok
        *(array + 2);           // error, array is not a pointer
}

void ditt(int[3] array)
{
        array[2];               // ok
        *(array + 2);           // error, array is not a pointer
}
```

Rectangular Arrays

Experienced FORTRAN numerics programmers know that multidimensional "rectangular" arrays for things like matrix operations are much faster than trying to access them via pointers to pointers resulting from "array of pointers to array" semantics. For example, the D syntax:

```
double[][] matrix;
```

declares matrix as an array of pointers to arrays. (Dynamic arrays are implemented as pointers to the array data.) Since the arrays can have varying sizes (being dynamically sized), this is sometimes called "jagged" arrays. Even worse for optimizing the code, the array rows can sometimes point to each other! Fortunately, D static arrays, while using the same syntax, are implemented as a fixed rectangular layout:

```
double[3][3] matrix;
```

declares a rectangular matrix with 3 rows and 3 columns, all contiguously in memory. In other languages, this would be called a multidimensional array and be declared as:

```
double matrix[3,3];
```

Array Length

Within the [ ] of a static or a dynamic array, the variable **length** is implicitly declared and set to the length of the array.

```
int[4] foo;
int[]  bar = foo;
int*   p = &foo[0];

// These expressions are equivalent:
bar[]
bar[0 .. 4]
bar[0 .. length]
bar[0 .. bar.length]

p[0 .. length]  // 'length' is not defined, since p is not an array
bar[0]+length   // 'length' is not defined, out of scope of [ ]

bar[length-1]   // retrieves last element of the array
```

Array Properties

Static array properties are:

.sizeof	Returns the array length multiplied by the number of bytes per array element.

.length	Returns the number of elements in the array. This is a fixed quantity for static arrays.
.ptr	Returns a pointer to the first element of the array.
.dup	Create a dynamic array of the same size and copy the contents of the array into it.
.reverse	Reverses in place the order of the elements in the array. Returns the array.
.sort	Sorts in place the order of the elements in the array. Returns the array.

Dynamic array properties are:

.sizeof	Returns the size of the dynamic array reference, which is 8 on 32 bit machines.
.length	Get/set number of elements in the array.
.ptr	Returns a pointer to the first element of the array.
.dup	Create a dynamic array of the same size and copy the contents of the array into it.
.reverse	Reverses in place the order of the elements in the array. Returns the array.
.sort	Sorts in place the order of the elements in the array. Returns the array.

Examples:

```
p.length        error, length not known for pointer
s.length        compile time constant 3
a.length        runtime value

p.dup           error, length not known
s.dup           creates an array of 3 elements, copies
                elements s into it
a.dup           creates an array of a.length elements, copies
                elements of a into it
```

Setting Dynamic Array Length

The **`.length`** property of a dynamic array can be set as the lvalue of an = operator:

```
array.length = 7;
```

This causes the array to be reallocated in place, and the existing contents copied over to the new array. If the new array length is shorter, only enough are copied to fill the new array. If the new array length is longer, the remainder is filled out with the default initializer.

To maximize efficiency, the runtime always tries to resize the array in place to avoid extra copying. It will always do a copy if the new size is larger and the array was not allocated via the new operator or a previous resize operation.

This means that if there is an array slice immediately following the array being resized, the resized array could overlap the slice; i.e.:

```
char[] a = new char[20];
char[] b = a[0..10];
char[] c = a[10..20];

b.length = 15;  // always resized in place because it is sliced
                // from a[] which has enough memory for 15 chars
b[11] = 'x';    // a[15] and c[5] are also affected

a.length = 1;
a.length = 20;  // no net change to memory layout

c.length = 12;  // always does a copy because c[] is not at the
                // start of a gc allocation block
c[5] = 'y';     // does not affect contents of a[] or b[]

a.length = 25;  // may or may not do a copy
a[3] = 'z';     // may or may not affect b[3] which still overlaps
                // the old a[3]
```

To guarantee copying behavior, use the .dup property to ensure a unique array that can be resized.

These issues also apply to concatenating arrays with the ~ and ~= operators.

Resizing a dynamic array is a relatively expensive operation. So, while the following method of filling an array:

```
int[] array;
while (1)
{   c = getinput();
    if (!c)
       break;
    array.length = array.length + 1;
    array[array.length - 1] = c;
}
```

will work, it will be inefficient. A more practical approach would be to minimize the number of resizes:

```
int[] array;
array.length = 100;         // guess
for (i = 0; 1; i++)
{   c = getinput();
     if (!c)
        break;
     if (i == array.length)
        array.length = array.length * 2;
     array[i] = c;
}
array.length = i;
```

Picking a good initial guess is an art, but you usually can pick a value covering 99% of the cases. For example, when gathering user input from the console - it's unlikely to be longer than 80.

Array Bounds Checking

It is an error to index an array with an index that is less than 0 or greater than or equal to the array length. If an index is out of bounds, an ArrayBoundsError exception is raised if detected at runtime, and an error if detected at compile time. A program may not rely on array bounds checking happening, for example, the following program is incorrect:

```
try
{
    for (i = 0; ; i++)
    {
        array[i] = 5;
    }
}
catch (ArrayBoundsError)
{
    // terminate loop
}
```

The loop is correctly written:

```
for (i = 0; i < array.length; i++)
{
    array[i] = 5;
}
```

Implementation Note: Compilers should attempt to detect array bounds errors at compile time, for example:

```
int[3] foo;
int x = foo[3];         // error, out of bounds
```

Insertion of array bounds checking code at runtime should be turned on and off with a compile time switch.

Array Initialization

Pointers are initialized to **null**.

Static array contents are initialized to the default initializer for the array element type.

Dynamic arrays are initialized to having 0 elements.

Associative arrays are initialized to having 0 elements.

Static Initialization of Static Arrays

```
int[3] a = [ 1:2, 3 ];          // a[0] = 0, a[1] = 2, a[2] = 3
```

This is most handy when the array indices are given by enums:

```
enum Color { red, blue, green };

int value[Color.max] = [ blue:6, green:2, red:5 ];
```

If any members of an array are initialized, they all must be. This is to catch common errors where another element is added to an enum, but one of the static instances of arrays of that enum was overlooked in updating the initializer list.

Special Array Types

Arrays of Bits

Bit vectors can be constructed:

```
bit[10] x;              // array of 10 bits
```

The amount of storage used up is implementation dependent. **Implementation Note:** on Intel CPUs it would be rounded up to the next 32 bit size.

```
x.length                // 10, number of bits
x.size                  // 4,  bytes of storage
```

So, the size per element is not (x.size / x.length).

Strings

Languages should be good at handling strings. C and C++ are not good at it. The primary difficulties are memory management, handling of temporaries, constantly rescanning the string looking for the terminating 0, and the fixed arrays.

Dynamic arrays in D suggest the obvious solution - a string is just a dynamic array of characters. String literals become just an easy way to write character arrays.

```
char[] str;
char[] str1 = "abc";
```

char[] strings are in UTF-8 format. wchar[] strings are in UTF-16 format. dchar[] strings are in UTF-32 format.

Strings can be copied, compared, concatenated, and appended:

```
str1 = str2;
if (str1 < str3) ...
func(str3 ~ str4);
str4 ~= str1;
```

with the obvious semantics. Any generated temporaries get cleaned up by the garbage collector (or by using alloca()). Not only that, this works with any array not just a special String array.

A pointer to a char can be generated:

```
char *p = &str[3];      // pointer to 4th element
char *p = str;          // pointer to 1st element
```

Since strings, however, are not 0 terminated in D, when transferring a pointer to a string to C, add a terminating 0:

```
str ~= "\0";
```

The type of a string is determined by the semantic phase of compilation. The type is one of: char[], wchar[], dchar[], and is determined by implicit conversion rules. If there are two equally applicable implicit conversions, the result is an error. To disambiguate these cases, a cast is appropriate:

```
cast(wchar [])"abc"     // this is an array of wchar characters
```

String literals are implicitly converted between chars, wchars, and dchars as necessary.

Strings a single character in length can also be exactly converted to a char, wchar or dchar constant:

```
char c;
wchar w;
dchar d;

c = 'b';                // c is assigned the character 'b'
w = 'b';                // w is assigned the wchar character 'b'
w = 'bc';               // error - only one wchar character at a time
w = "b"[0];             // w is assigned the wchar character 'b'
w = \r;                 // w is assigned the carriage return wchar
character
d = 'd';                // d is assigned the character 'd'
```

printf() and Strings

printf() is a C function and is not part of D. **printf()** will print C strings, which are 0 terminated. There are two ways to use **printf()** with D strings. The first is to add a terminating 0, and cast the result to a char*:

```
str ~= "\0";
printf("the string is '%s'\n", (char *)str);
```

The second way is to use the precision specifier. The way D arrays are laid out, the length comes first, so the following works:

```
printf("the string is '%.*s'\n", str);
```

In the future, it may be necessary to just add a new format specifier to **printf()** instead of relying on an implementation dependent detail.

Implicit Conversions

A pointer `T*` can be implicitly converted to one of the following:

*U** where *U* is a base class of *T*.

`void*`

A static array `T[dim]` can be implicitly converted to one of the following:

*T**

T[]

U[*dim*] where *U* is a base class of *T*.

U[] where *U* is a base class of *T*.

*U** where *U* is a base class of *T*.

`void*`

`void[]`

A dynamic array `T[]` can be implicitly converted to one of the following:

*T**

U[] where *U* is a base class of *T*.

*U** where *U* is a base class of *T*.

`void*`

Associative Arrays

D goes one step further with arrays - adding associative arrays. Associative arrays have an index that is not necessarily an integer, and can be sparsely populated. The index for an associative array is called the *key*.

Associative arrays are declared by placing the *key* type within the [] of an array declaration:

```
int[char[]] b;          // associative array b of ints that are
                        // indexed by an array of characters
b["hello"] = 3;         // set value associated with key "hello" to 3
func(b["hello"]);       // pass 3 as parameter to func()
```

Particular keys in an associative array can be removed with the delete operator:

```
delete b["hello"];
```

This confusingly appears to delete the value of b["hello"], but does not, it removes the key "hello" from the associative array.

The *InExpression* yields a pointer to the value if the key is in the associatve array, or null if not:

```
if (("hello" in b) != null)
        ...
```

Key types cannot be functions or voids.

Properties

Properties for associative arrays are:

size	Returns the size of the reference to the associative array; it is typically 8.
length	Returns number of values in the associative array. Unlike for dynamic arrays, it is read-only.
keys	Returns dynamic array, the elements of which are the keys in the associative array.
values	Returns dynamic array, the elements of which are the values in the associative array.
rehash	Reorganizes the associative array in place so that lookups are more efficient. rehash is effective when, f the program is done loading up a symbol table and now needs fast lookups in it. Returns a reference to t reorganized array.

Associative Array Example: word count

```
import std.file;            // D file I/O

int main (char[][] args)
{
    int word_total;
    int line_total;
    int char_total;
    int[char[]] dictionary;

    printf("   lines   words   bytes file\n");
    for (int i = 1; i < args.length; ++i)   // program arguments
```

```
        {
            char[] input;               // input buffer
            int w_cnt, l_cnt, c_cnt;    // word, line, char counts
            int inword;
            int wstart;

            input = std.file.read(args[i]);     // read file into input[]

            foreach (char c; input)
            {
                if (c == '\n')
                    ++l_cnt;
                if (c >= '0' && c <= '9')
                {
                }
                else if (c >= 'a' && c <= 'z' ||
                    c >= 'A' && c <= 'Z')
                {
                    if (!inword)
                    {
                        wstart = j;
                        inword = 1;
                        ++w_cnt;
                    }
                }
                else if (inword)
                {   char[] word = input[wstart .. j];

                    dictionary[word]++;         // increment count for word
                    inword = 0;
                }
                ++c_cnt;
            }
            if (inword)
            {   char[] word = input[wstart .. input.length];
                dictionary[word]++;
            }
            printf("%8ld%8ld%8ld %.*s\n", l_cnt, w_cnt, c_cnt, args[i]);
            line_total += l_cnt;
            word_total += w_cnt;
            char_total += c_cnt;
        }

        if (args.length > 2)
        {
            printf("-------------------------------------\n%8ld%8ld%8ld
total",
                line_total, word_total, char_total);
        }

        printf("-------------------------------------\n");
        char[][] keys = dictionary.keys;        // find all words in
dictionary[]
        for (int i = 0; i < keys.length; i++)
        {   char[] word;

            word = keys[i];
```

```
            printf("%3d %.*s\n", dictionary[word], word);
        }
        return 0;
    }
```

Structs & Unions

AggregateDeclaration:
        Tag **{** *DeclDefs* **}**
        Tag Identifier **{** *DeclDefs* **}**
        Tag Identifier **;**

Tag:
        struct
        union

They work like they do in C, with the following exceptions:

no bit fields

alignment can be explicitly specified

no separate tag name space - tag names go into the current scope

declarations like:

```
struct ABC x;
```

are not allowed, replace with:

```
ABC x;
```

anonymous structs/unions are allowed as members of other structs/unions

Default initializers for members can be supplied.

Member functions and static members are allowed.

Structs and unions are meant as simple aggregations of data, or as a way to paint a data structure over hardware or an external type. External types can be defined by the operating system API, or by a file format. Object oriented features are provided with the class data type.

Static Initialization of Structs

Static struct members are by default initialized to whatever the default initializer for the member

is, and if none supplied, to the default initializer for the member's type. If a static initializer is supplied, the members are initialized by the member name, colon, expression syntax. The members may be initialized in any order. Members not specified in the initializer list are default initialized.

```
        struct X { int a; int b; int c; int d = 7;}
        static X x = { a:1, b:2};       // c is set to 0, d to 7
        static X z = { c:4, b:5, a:2 , d:5};      // z.a = 2, z.b = 5, z.c =
4, z.d = 5
```

C-style initialization, based on the order of the members in the struct declaration, is also supported:

```
        static X q = { 1, 2 };    // q.a = 1, q.b = 2, q.c = 0, q.d = 7
```

Static Initialization of Unions

Unions are initialized explicitly.

```
        union U { int a; double b; }
        static U u = { b : 5.0 };               // u.b = 5.0
```

Other members of the union that overlay the initializer, but occupy more storage, have the extra storage initialized to zero.

Struct Properties

```
        .sizeof                 Size in bytes of struct
        .alignof                Size boundary struct needs to be aligned on
```

Struct Field Properties

```
        .offsetof               Offset in bytes of field from beginning
                                of struct
```

Classes

The object-oriented features of D all come from classes. The class hierarchy has as its root the class Object. Object defines a minimum level of functionality that each derived class has, and a default implementation for that functionality.

Classes are programmer defined types. Support for classes are what make D an object oriented

language, giving it encapsulation, inheritance, and polymorphism. D classes support the single inheritance paradigm, extended by adding support for interfaces. Class objects are instantiated by reference only.

A class can be exported, which means its name and all its non-private members are exposed externally to the DLL or EXE.

A class declaration is defined:

```
ClassDeclaration:
        class Identifier [SuperClass {, InterfaceClass }] ClassBody

SuperClass:
        : Identifier

InterfaceClass:
        Identifier

ClassBody:
        { ClassBodyDeclarations }

ClassBodyDeclaration:
        Declaration
        Constructor
        Destructor
        StaticConstructor
        StaticDestructor
        Invariant
        UnitTest
        ClassAllocator
        ClassDeallocator
```

Classes consist of:

super class
interfaces
dynamic fields
static fields
types
functions
- static functions
- dynamic functions
- constructors
- destructors
- static constructors
- static destructors
- invariants
- unit tests
- allocators
- deallocators

A class is defined:

```
class Foo
{
    ... members ...
}
```

Note that there is no trailing ; after the closing } of the class definition. It is also not possible to declare a variable var like:

```
class Foo { } var;
```

Instead:

```
class Foo { }
Foo var;
```

Fields

Class members are always accessed with the . operator. There are no :: or -> operators as in C++.

The D compiler is free to rearrange the order of fields in a class to optimally pack them in an implementation-defined manner. Consider the fields much like the local variables in a function - the compiler assigns some to registers and shuffles others around all to get the optimal stack frame layout. This frees the code designer to organize the fields in a manner that makes the code more readable rather than being forced to organize it according to machine optimization rules. Explicit control of field layout is provided by struct/union types, not classes.

Field Properties

```
.offsetof               Offset in bytes of field from beginning
                        of class
```

Super Class

All classes inherit from a super class. If one is not specified, it inherits from Object. Object forms the root of the D class inheritance hierarchy.

Constructors

Constructor:
        this() *BlockStatement*

Members are always initialized to the default initializer for their type, which is usually 0 for integer types and NAN for floating point types. This eliminates an entire class of obscure problems that come from neglecting to initialize a member in one of the constructors. In the class definition, there can be a static initializer to be used instead of the default:

```
class Abc
```

```
{
    int a;      // default initializer for a is 0
    long b = 7; // default initializer for b is 7
    float f;    // default initializer for f is NAN
}
```

This static initialization is done before any constructors are called.

Constructors are defined with a function name of **this** and having no return value:

```
class Foo
{
    this(int x)         // declare constructor for Foo
    {   ...
    }
    this()
    {   ...
    }
}
```

Base class construction is done by calling the base class constructor by the name **super**:

```
class A { this(int y) { } }

class B : A
{
    int j;
    this()
    {
        ...
        super(3);       // call base constructor A.this(3)
        ...
    }
}
```

Constructors can also call other constructors for the same class in order to share common initializations:

```
class C
{
    int j;
    this()
    {
        ...
    }
    this(int i)
    {
        this();
        j = i;
    }
}
```

If no call to constructors via **this** or **super** appear in a constructor, and the base class has a constructor, a call to **super**() is inserted at the beginning of the constructor.

If there is no constructor for a class, but there is a constructor for the base class, a default constructor of the form:

```
this() { }
```

is implicitly generated.

Class object construction is very flexible, but some restrictions apply:

1. It is illegal for constructors to mutually call each other:

```
this() { this(1); }
this(int i) { this(); } // illegal, cyclic constructor calls
```

5. If any constructor call appears inside a constructor, any path through the constructor must make exactly one constructor call:

```
this()  { a || super(); }       // illegal

this() { this(1) || super(); }  // ok

 this()
 {
     for (...)
     {
         super();        // illegal, inside loop
     }
 }
```

18. It is illegal to refer to **this** implicitly or explicitly prior to making a constructor call.
19. Constructor calls cannot appear after labels (in order to make it easy to check for the previous conditions in the presence of goto's).

Instances of class objects are created with *NewExpression*s:

```
A a = new A(3);
```

The following steps happen:

1. Storage is allocated for the object. If this fails, rather than return **null**, an **OutOfMemoryException** is thrown. Thus, tedious checks for null references are unnecessary.
2. The raw data is statically initialized using the values provided in the class definition. The pointer to the vtbl is assigned. This ensures that constructors are passed fully formed objects. This operation is equivalent to doing a memcpy() of a static version of the object onto the newly allocated one, although more advanced compilers may be able to optimize much of this away.
3. If there is a constructor defined for the class, the constructor matching the argument list is

called.
4. If class invariant checking is turned on, the class invariant is called at the end of the constructor.

Destructors

```
Destructor:
        ~this() BlockStatement
```

The garbage collector calls the destructor function when the object is deleted. The syntax is:

```
class Foo
{
        ~this()         // destructor for Foo
        {
        }
}
```

There can be only one destructor per class, the destructor does not have any parameters, and has no attributes. It is always virtual.

The destructor is expected to release any resources held by the object.

The program can explicitly inform the garbage collector that an object is no longer referred to (with the delete expression), and then the garbage collector calls the destructor immediately, and adds the object's memory to the free storage. The destructor is guaranteed to never be called twice.

The destructor for the super class automatically gets called when the destructor ends. There is no way to call the super destructor explicitly.

When the garbage collector calls a destructor for an object of a class that has members that are references to garbage collected objects, those references are no longer valid. This means that destructors cannot reference sub objects. This rule does not apply to auto objects or objects deleted with the *DeleteExpression*.

The garbage collector is not guaranteed to run the destructor for all unreferenced objects. Furthermore, the order in which the garbage collector calls destructors for unreference objects is not specified.

Objects referenced from the data segment never get collected by the gc.

Static Constructors

```
StaticConstructor:
        static this() BlockStatement
```

A static constructor is defined as a function that performs initializations before the main() function gets control. Static constructors are used to initialize static class members with values

that cannot be computed at compile time.

Static constructors in other languages are built implicitly by using member initializers that can't be computed at compile time. The trouble with this stems from not having good control over exactly when the code is executed, for example:

```
class Foo
{
    static int a = b + 1;
    static int b = a * 2;
}
```

What values do a and b end up with, what order are the initializations executed in, what are the values of a and b before the initializations are run, is this a compile error, or is this a runtime error? Additional confusion comes from it not being obvious if an initializer is static or dynamic.

D makes this simple. All member initializations must be determinable by the compiler at compile time, hence there is no order-of-evaluation dependency for member initializations, and it is not possible to read a value that has not been initialized. Dynamic initialization is performed by a static constructor, defined with a special syntax `static this()`.

```
class Foo
{
    static int a;               // default initialized to 0
    static int b = 1;
    static int c = b + a;       // error, not a constant initializer

    static this()               // static constructor
    {
        a = b + 1;              // a is set to 2
        b = a * 2;              // b is set to 4
    }
}
```

`static this()` is called by the startup code before `main()` is called. If it returns normally (does not throw an exception), the static destructor is added to the list of functions to be called on program termination. Static constructors have empty parameter lists.

A current weakness of the static constructors is that the order in which they are called is not defined. Hence, for the time being, write the static constructors to be order independent. This problem needs to be addressed in future versions.

Static Destructor

StaticDestructor:
static ~this() *BlockStatement*

A static destructor is defined as a special static function with the syntax `static ~this()`.

```
class Foo
```

```
    {
        static ~this()              // static destructor
        {
        }
    }
```

A static destructor gets called on program termination, but only if the static constructor completed successfully. Static destructors have empty parameter lists. Static destructors get called in the reverse order that the static constructors were called in.

Class Invariants

```
    ClassInvariant:
            invariant BlockStatement
```

Class invariants are used to specify characteristics of a class that always must be true (except while executing a member function). For example, a class representing a date might have an invariant that the day must be 1..31 and the hour must be 0..23:

```
class Date
{
    int day;
    int hour;

    invariant
    {
        assert(1 <= day && day <= 31);
        assert(0 <= hour && hour < 24);
    }
}
```

The class invariant is a contract saying that the asserts must hold true. The invariant is checked when a class constructor completes, at the start of the class destructor, before a public or exported member is run, and after a public or exported function finishes.

The code in the invariant may not call any public non-static members of the class, either directly or indirectly. Doing so will result in a stack overflow, as the invariant will wind up being called in an infinitely recursive manner.

```
class Foo
{
    public void f() { }
    private void g() { }

    invariant
    {
        f();  // error, cannot call public member function from invariant
        g();  // ok, g() is not public
    }
}
```

The invariant can be checked when a class object is the argument to an `assert()` expression, as:

```
Date mydate;
...
assert(mydate);         // check that class Date invariant holds
```

If the invariant fails, it throws an `InvariantException`. Class invariants are inherited, that is, any class invariant is implicitly anded with the invariants of its base classes.

There can be only one *ClassInvariant* per class.

When compiling for release, the invariant code is not generated, and the compiled program runs at maximum speed.

Unit Tests

UnitTest:
unittest *BlockStatement*

Unit tests are a series of test cases applied to a class to determine if it is working properly. Ideally, unit tests should be run every time a program is compiled. The best way to make sure that unit tests do get run, and that they are maintained along with the class code is to put the test code right in with the class implementation code.

D classes can have a special member function called:

```
unittest
{
    ...test code...
}
```

The test() functions for all the classes in the program get called after static initialization is done and before the main function is called. A compiler or linker switch will remove the test code from the final build.

For example, given a class Sum that is used to add two values:

```
class Sum
{
    int add(int x, int y) { return x + y; }

    unittest
    {
        assert(add(3,4) == 7);
        assert(add(-2,0) == -2);
    }
}
```

Class Allocators

ClassAllocator:
new *ParameterList BlockStatement*

A class member function of the form:

```
new(uint size)
{
    ...
}
```

is called a class allocator. The class allocator can have any number of parameters, provided the first one is of type uint. Any number can be defined for a class, the correct one is determined by the usual function overloading rules. When a new expression:

```
new Foo;
```

is executed, and Foo is a class that has an allocator, the allocator is called with the first argument set to the size in bytes of the memory to be allocated for the instance. The allocator must allocate the memory and return it as a `void*`. If the allocator fails, it must not return a **null**, but must throw an exception. If there is more than one parameter to the allocator, the additional arguments are specified within parentheses after the **new** in the *NewExpression*:

```
class Foo
{
    this(char[] a) { ... }

    new(uint size, int x, int y)
    {
        ...
    }
}

...

new(1,2) Foo(a);        // calls new(Foo.size,1,2)
```

Derived classes inherit any allocator from their base class, if one is not specified.

See also Explicit Class Instance Allocation.

Class Deallocators

ClassDeallocator:
delete *ParameterList BlockStatement*

A class member function of the form:

```
delete(void *p)
{
    ...
}
```

is called a class deallocator. The deallocator must have exactly one parameter of type `void*`. Only one can be specified for a class. When a delete expression:

```
delete f;
```

is executed, and f is a reference to a class instance that has a deallocator, the deallocator is called with a pointer to the class instance after the destructor (if any) for the class is called. It is the responsibility of the deallocator to free the memory.

Derived classes inherit any deallocator from their base class, if one is not specified.

See also Explicit Class Instance Allocation.

Auto Classes

An auto class is a class with the auto attribute, as in:

```
auto class Foo { ... }
```

The auto characteristic is inherited, so if any classes derived from an auto class are also auto.

An auto class reference can only appear as a function local variable. It must be declared as being **auto**:

```
auto class Foo { ... }

void func()
{
    Foo f;      // error, reference to auto class must be auto
    auto Foo g = new Foo();     // correct
}
```

When an auto class reference goes out of scope, the destructor (if any) for it is automatically called. This holds true even if the scope was exited via a thrown exception.

Interfaces

```
InterfaceDeclaration:
        interface Identifier InterfaceBody
        interface Identifier : SuperInterfaces InterfaceBody

SuperInterfaces
```

```
        Identifier
        Identifier , SuperInterfaces

InterfaceBody:
        { DeclDefs }
```

Interfaces describe a list of functions that a class that inherits from the interface must implement. A class that implements an interface can be converted to a reference to that interface. Interfaces correspond to the interface exposed by operating system objects, like COM/OLE/ActiveX for Win32.

Interfaces cannot derive from classes; only from other interfaces. Classes cannot derive from an interface multiple times.

```
interface D
{
    void foo();
}

class A : D, D  // error, duplicate interface
{
}
```

An instance of an interface cannot be created.

```
interface D
{
    void foo();
}

...

D d = new D();          // error, cannot create instance of interface
```

Interface member functions do not have implementations.

```
interface D
{
    void bar() { }      // error, implementation not allowed
}
```

All interface functions must be defined in a class that inherits from that interface:

```
interface D
{
    void foo();
}

class A : D
{
    void foo() { }      // ok, provides implementation
}
```

```
class B : D
{
    int foo() { }       // error, no void foo() implementation
}
```

Interfaces can be inherited and functions overridden:

```
interface D
{
    int foo();
}

class A : D
{
    int foo() { return 1; }
}

class B : A
{
    int foo() { return 2; }
}

...

B b = new B();
b.foo();                // returns 2
D d = (D) b;            // ok since B inherits A's D implementation
d.foo();                // returns 2;
```

Interfaces can be reimplemented in derived classes:

```
interface D
{
    int foo();
}

class A : D
{
    int foo() { return 1; }
}

class B : A, D
{
    int foo() { return 2; }
}

...

B b = new B();
b.foo();                // returns 2
D d = (D) b;
d.foo();                // returns 2
A a = (A) b;
D d2 = (D) a;
```

```
        d2.foo();               // returns 2, even though it is A's D, not
B's D
```

A reimplemented interface must implement all the interface functions, it does not inherit them from a super class:

```
        interface D
        {
            int foo();
        }

        class A : D
        {
            int foo() { return 1; }
        }

        class B : A, D
        {
        }               // error, no foo() for interface D
```

COM Interfaces

A variant on interfaces is the COM interface. A COM interface is designed to map directly onto a Windows COM object. Any COM object can be represented by a COM interface, and any D object with a COM interface can be used by external COM clients.

A COM interface is defined as one that derives from the interface `std.c.windows.com.IUnknown`. A COM interface differs from a regular D interface in that:

It derives from the interface `std.c.windows.com.IUnknown`.

It cannot be the argument of a *DeleteExpression*.

References cannot be upcast to the enclosing class object, nor can they be downcast to a derived interface. To accomplish this, an appropriate `QueryInterface()` would have to be implemented for that interface in standard COM fashion.

Enums - Enumerated Types

```
EnumDeclaration:
        enum Identifier EnumBody
        enum EnumBody
        enum identifier : EnumBaseType EnumBody
        enum EnumBaseType : EnumBody

EnumBaseType:
```

```
        Type

EnumBody:
        ;
        { EnumMembers }

EnumMembers:
        EnumMember
        EnumMember ,
        EnumMember , EnumMembers

EnumMember:
        Identifier
        Identifier = Expression
```

Enums are used to define a group of related integral constants.

If the enum *Identifier* is present, the *EnumMembers* are declared in the scope of the enum *Identifier*. The enum *Identifier* declares a new type.

If the enum *Identifier* is not present, then the enum is an *anonymous enum*, and the *EnumMembers* are declared in the scope the *EnumDeclaration* appears in. No new type is created; the *EnumMembers* have the type of the *EnumBaseType*.

The *EnumBaseType* is the underlying type of the enum. It must be an integral type. If omitted, it defaults to **int**.

```
enum { A, B, C }        // anonymous enum
```

Defines the constants A=0, B=1, C=2 in a manner equivalent to:

```
const int A = 0;
const int B = 1;
const int C = 2;
```

Whereas:

```
enum X { A, B, C }      // named enum
```

Define a new type X which has values X.A=0, X.B=1, X.C=2

Named enum members can be implicitly cast to integral types, but integral types cannot be implicitly cast to an enum type.

Enums must have at least one member.

If an *Expression* is supplied for an enum member, the value of the member is set to the result of the *Expression*. The *Expression* must be resolvable at compile time. Subsequent enum members with no *Expression* are set to the value of the previous member plus one:

```
enum { A, B = 5+7, C, D = 8, E }
```

Sets A=0, B=12, C=13, D=8, and E=9.

Enum Properties

```
.min                    Smallest value of enum
.max                    Largest value of enum
.sizeof                 Size of storage for an enumerated value

For example:

X.min                   is X.A
X.max                   is X.C
X.sizeof                is same as int.sizeof
```

Initialization of Enums

In the absence of an explicit initializer, an enum variable is initialized to the first enum value.

```
enum X { A=3, B, C }
X x;            // x is initialized to 3
```

Functions

Virtual Functions

All non-static non-private member functions are virtual. This may sound inefficient, but since the D compiler knows all of the class hierarchy when generating code, all functions that are not overridden can be optimized to be non-virtual. In fact, since C++ programmers tend to "when in doubt, make it virtual", the D way of "make it virtual unless we can prove it can be made non-virtual" results on average much more direct function calls. It also results in fewer bugs caused by not declaring a function virtual that gets overridden.

Functions with non-D linkage cannot be virtual, and hence cannot be overridden.

Functions marked as `final` may not be overridden in a derived class, unless they are also `private`. For example:

```
class A
{
    int def() { ... }
    final int foo() { ... }
    final private int bar() { ... }
    private int abc() { ... }
}
```

```
class B : A
{
    int def() { ... }   // ok, overrides A.def
    int foo() { ... }   // error, A.foo is final
    int bar() { ... }   // ok, A.bar is final private, but not virtual
    int abc() { ... }   // ok, A.abc is not virtual, B.abc is virtual
}

void test(A a)
{
    a.def();    // calls B.def
    a.foo();    // calls A.foo
    a.bar();    // calls A.bar
    a.abc();    // calls A.abc
}

void func()
{   B b = new B();
    test(b);
}
```

Covariant return types are supported, which means that the overriding function in a derived class can return a type that is derived from the type returned by the overridden function:

```
class A { }
class B : A { }

class Foo
{
    A test() { return null; }
}

class Bar : Foo
{
    B test() { return null; }   // overrides and is covariant with
Foo.test()
}
```

Function Inheritance and Overriding

A functions in a derived class with the same name and parameter types as a function in a base class overrides that function:

```
class A
{
    int foo(int x) { ... }
}

class B : A
{
    override int foo(int x) { ... }
}
```

```
        void test()
        {
            B b = new B();
            bar(b);
        }

        void bar(A a)
        {
            a.foo();    // calls B.foo(int)
        }
```

However, when doing overload resolution, the functions in the base class are not considered:

```
        class A
        {
            int foo(int x) { ... }
            int foo(long y) { ... }
        }

        class B : A
        {
            override int foo(long x) { ... }
        }

        void test()
        {
            B b = new B();
            bar(b);
        }

        void bar(A a)
        {
            a.foo(1);           // calls A.foo(int)
            B b = new B();
            b.foo(1);           // calls B.foo(long), since A.foo(int) not
considered
        }
```

To consider the base class's functions in the overload resolution process, use an *AliasDeclaration*:

```
        class A
        {
            int foo(int x) { ... }
            int foo(long y) { ... }
        }

        class B : A
        {
            alias A.foo foo;
            override int foo(long x) { ... }
        }

        void test()
        {
```

```
    B b = new B();
    bar(b);
}

void bar(A a)
{
    a.foo(1);           // calls A.foo(int)
    B b = new B();
    b.foo(1);           // calls A.foo(int)
}
```

A function parameter's default value is not inherited:

```
class A
{
    void foo(int x = 5) { ... }
}

class B : A
{
    void foo(int x = 7) { ... }
}

class C : B
{
    void foo(int x) { ... }
}


void test()
{
    A a = new A();
    a.foo();            // calls A.foo(5)

    B b = new B();
    b.foo();            // calls B.foo(7)

    C c = new C();
    c.foo();            // error, need an argument for C.foo
}
```

Inline Functions

There is no inline keyword. The compiler makes the decision whether to inline a function or not, analogously to the register keyword no longer being relevant to a compiler's decisions on enregistering variables. (There is no register keyword either.)

Function Overloading

In C++, there are many complex levels of function overloading, with some defined as "better" matches than others. If the code designer takes advantage of the more subtle behaviors of

overload function selection, the code can become difficult to maintain. Not only will it take a C++ expert to understand why one function is selected over another, but different C++ compilers can implement this tricky feature differently, producing subtly disastrous results.

In D, function overloading is simple. It matches exactly, it matches with implicit conversions, or it does not match. If there is more than one match, it is an error.

Functions defined with non-D linkage cannot be overloaded.

Function Parameters

Parameters are **in**, **out**, or **inout**. **in** is the default; **out** and **inout** work like storage classes. For example:

```
int foo(int x, out int y, inout int z, int q);
```

x is **in**, y is **out**, z is **inout**, and q is **in**.

out is rare enough, and **inout** even rarer, to attach the keywords to them and leave **in** as the default. The reasons to have them are:

The function declaration makes it clear what the inputs and outputs to the function are.

It eliminates the need for IDL as a separate language.

It provides more information to the compiler, enabling more error checking and possibly better code generation.

It (perhaps?) eliminates the need for reference (&) declarations.

out parameters are set to the default initializer for the type of it. For example:

```
void foo(out int bar)
{
}

int bar = 3;
foo(bar);
// bar is now 0
```

Variadic Function Parameters

Functions can be variadic, meaning they can take an arbitrary number of parameters. A variadic function is declared as taking a parameter of ... after the required function parameters:

```
int foo(int x, int y, ...);

foo(3, 4);       // ok
foo(3, 4, 6.8); // ok, one variadic argument
foo(2);          // error, y is a required argument
```

Variadic functions with non-D linkage must have at least one non-variadic parameter declared.

```
int abc(...);           // ok, D linkage
extern (C) def(...);    // error, must have at least one parameter
```

Variadic functions have a special local variable declared for them, **_argptr**, which is a `void*` pointer to the first of the variadic arguments. To access the arguments, **_argptr** must be cast to a pointer to the expected argument type:

```
foo(3, 4, 5);   // first variadic argument is 5

int foo(int x, int y, ...)
{   int z;

    z = *cast(int*)_argptr;     // z is set to 5
}
```

For variadic functions with D linkage, an additional hidden argument with the name **_arguments** and type `TypeInfo[]` is passed to the function. **_arguments** gives the number of arguments and the type of each, enabling the creation of typesafe variadic functions.

```
class FOO { }

void foo(int x, ...)
{
    printf("%d arguments\n", _arguments.length);
    for (int i = 0; i < _arguments.length; i++)
    {   _arguments[i].print();

        if (_arguments[i] == typeid(int))
        {
            int j = *cast(int *)_argptr;
            _argptr += int.sizeof;
            printf("\t%d\n", j);
        }
        else if (_arguments[i] == typeid(long))
        {
            long j = *cast(long *)_argptr;
            _argptr += long.sizeof;
            printf("\t%lld\n", j);
        }
        else if (_arguments[i] == typeid(double))
        {
            double d = *cast(double *)_argptr;
            _argptr += double.sizeof;
            printf("\t%g\n", d);
        }
        else if (_arguments[i] == typeid(FOO))
        {
            FOO f = *cast(FOO*)_argptr;
            _argptr += FOO.sizeof;
            printf("\t%p\n", f);
        }
        else
```

```
            assert(0);
        }
    }

    void main()
    {
        FOO f = new FOO();

        printf("%p\n", f);
        foo(1, 2, 3L, 4.5, f);
    }
```

which prints:

```
00870FD0
4 arguments
int
        2
long
        3
double
        4.5
FOO
        00870FD0
```

To protect against the vagaries of stack layouts on different CPU architectures, use **std.stdarg** to access the variadic arguments:

```
import std.stdarg;

void foo(int x, ...)
{
    printf("%d arguments\n", _arguments.length);
    for (int i = 0; i < _arguments.length; i++)
    {   _arguments[i].print();

        if (_arguments[i] == typeid(int))
        {
            int j = va_arg!(int)(_argptr);
            printf("\t%d\n", j);
        }
        else if (_arguments[i] == typeid(long))
        {
            long j = va_arg!(long)(_argptr);
            printf("\t%lld\n", j);
        }
        else if (_arguments[i] == typeid(double))
        {
            double d = va_arg!(double)(_argptr);
            printf("\t%g\n", d);
        }
        else if (_arguments[i] == typeid(FOO))
        {
            FOO f = va_arg!(FOO)(_argptr);
            printf("\t%p\n", f);
```

```
            }
            else
                assert(0);
        }
    }
```

Local Variables

It is an error to use a local variable without first assigning it a value. The implementation may not always be able to detect these cases. Other language compilers sometimes issue a warning for this, but since it is always a bug, it should be an error.

It is an error to declare a local variable that is never referred to. Dead variables, like anachronistic dead code, is just a source of confusion for maintenance programmers.

It is an error to declare a local variable that hides another local variable in the same function:

```
void func(int x)
{   int x;              error, hides previous definition of x
    double y;
    ...
    {   char y;         error, hides previous definition of y
        int z;
    }
    {   wchar z;        legal, previous z is out of scope
    }
}
```

While this might look unreasonable, in practice whenever this is done it either is a bug or at least looks like a bug.

It is an error to return the address of or a reference to a local variable.

It is an error to have a local variable and a label with the same name.

Nested Functions

Functions may be nested within other functions:

```
int bar(int a)
{
    int foo(int b)
    {
        int abc() { return 1; }

        return b + abc();
    }
    return foo(a);
}

void test()
```

```
{
    int i = bar(3);     // i is assigned 4
}
```

Nested functions can only be accessed by the most nested lexically enclosing function, or by another nested function at the same nesting depth:

```
int bar(int a)
{
    int foo(int b) { return b + 1; }
    int abc(int b) { return foo(b); }   // ok
    return foo(a);
}

void test()
{
    int i = bar(3);     // ok
    int j = bar.foo(3); // error, bar.foo not visible
}
```

Nested functions have access to the variables and other symbols defined by the lexically enclosing function. This access includes both the ability to read and write them.

```
int bar(int a)
{   int c = 3;

    int foo(int b)
    {
        b += c;         // 4 is added to b
        c++;            // bar.c is now 5
        return b + c;   // 12 is returned
    }
    c = 4;
    int i = foo(a);     // i is set to 12
    return i + c;       // returns 17
}

void test()
{
    int i = bar(3);     // i is assigned 17
}
```

This access can span multiple nesting levels:

```
int bar(int a)
{   int c = 3;

    int foo(int b)
    {
        int abc()
        {
            return c;   // access bar.c
        }
```

```
            return b + c + abc();
        }
        return foo(3);
    }
```

Static nested functions cannot access any stack variables of any lexically enclosing function, but can access static variables. This is analogous to how static member functions behave.

```
int bar(int a)
{   int c;
    static int d;

    static int foo(int b)
    {
        b = d;          // ok
        b = c;          // error, foo() cannot access frame of bar()
        return b + 1;
    }
    return foo(a);
}
```

Functions can be nested within member functions:

```
struct Foo
{   int a;

    int bar()
    {   int c;

        int foo()
        {
            return c + a;
        }
    }
}
```

Member functions of nested classes and structs do not have access to the stack variables of the enclosing function, but do have access to the other symbols:

```
void test()
{   int j;
    static int s;

    struct Foo
    {   int a;

        int bar()
        {   int c = s;          // ok, s is static
            int d = j;          // error, no access to frame of test()

            int foo()
            {
                int e = s;      // ok, s is static
```

```
                        int f = j;      // error, no access to frame of test()
                        return c + a;   // ok, frame of bar() is accessible,
                                        // so are members of Foo accessible
via
                                        // the 'this' pointer to Foo.bar()
                    }
                }
            }
        }
```

Nested functions always have the D function linkage type.

Unlike module level declarations, declarations within function scope are processed in order. This means that two nested functions cannot mutually call each other:

```
void test()
{
    void foo() { bar(); }       // error, bar not defined
    void bar() { foo(); }       // ok
}
```

The solution is to use a delegate:

```
void test()
{
    void delegate() fp;
    void foo() { fp(); }
    void bar() { foo(); }
    fp = &bar;
}
```

Future directions: This restriction may be removed.

Delegates, Function Pointers, and Dynamic Closures

A function pointer can point to a static nested function:

```
int function() fp;

void test()
{   static int a = 7;
    static int foo() { return a + 3; }

    fp = &foo;
}

void bar()
{
    test();
    int i = fp();       // i is set to 10
}
```

A delegate can be set to a non-static nested function:

```
int delegate() dg;

void test()
{   int a = 7;
    int foo() { return a + 3; }

    dg = &foo;
    int i = dg();       // i is set to 10
}
```

The stack variables, however, are not valid once the function declaring them has exited, in the same manner that pointers to stack variables are not valid upon exit from a function:

```
int* bar()
{   int b;
    test();
    int i = dg();       // error, test.a no longer exists
    return &b;          // error, bar.b not valid after bar() exits
}
```

Delegates to non-static nested functions contain two pieces of data: the pointer to the stack frame of the lexically enclosing function (called the *frame pointer*) and the address of the function. This is analogous to struct/class non-static member function delegates consisting of a *this* pointer and the address of the member function. Both forms of delegates are interchangeable, and are actually the same type:

```
struct Foo
{   int a = 7;
    int bar() { return a; }
}

int foo(int delegate() dg)
{
    return dg() + 1;
}

void test()
{
    int x = 27;
    int abc() { return x; }
    Foo f;
    int i;

    i = foo(&abc);      // i is set to 28
    i = foo(&f.bar);    // i is set to 8
}
```

This combining of the environment and the function is called a *dynamic closure*.

Future directions: Function pointers and delegates may merge into a common syntax and be interchangeable with each other.

Anonymous Functions and Anonymous Delegates

See Function Literals.

Operator Overloading

Overloading is accomplished by interpreting specially named struct and class member functions as being implementations of unary and binary operators. No additional syntax is used.

Unary Operator Overloading

Overloadable Unary Operators

op	*opfunc*
-*e*	`opNeg`
+*e*	`opPos`
~*e*	`opCom`
e++	`opPostInc`
e--	`opPostDec`
cast(*type*)*e*	`opCast`

Given a unary overloadable operator *op* and its corresponding class or struct member function name *opfunc*, the syntax:

```
op a
```

where *a* is a class or struct object reference, is interpreted as if it was written as:

```
a.opfunc()
```

Overloading ++*e* and --*e*

Since ++*e* is defined to be semantically equivalent to (*e* += 1), the expression ++*e* is rewritten as (*e* += 1), and then checking for operator overloading is done. The situation is analogous for --*e*.

Examples

```
1.      class A { int opNeg(); }
2.      A a;
3.      -a;     // equivalent to a.opNeg();
4.

5.      class A { int opNeg(int i); }
6.      A a;
7.      -a;     // equivalent to a.opNeg(), which is an error
8.
```

Overloading cast(*type*)*e*

The member function *e*.**opCast()** is called, and the return value of **opCast()** is implicitly converted to *type*. Since functions cannot be overloaded based on return value, there can be only one **opCast** per struct or class. Overloading the cast operator does not affect implicit casts, it only applies to explicit casts.

```
struct A
{
    int opCast() { return 28; }
}

void test()
{
    A a;

    long i = cast(long)a;   // i is set to 28L
    void* p = cast(void*)a; // error, cannot implicitly
                            // convert int to void*
    int j = a;              // error, cannot implicitly convert
                            // A to int
}
```

Binary Operator Overloading

Overloadable Binary Operators

op	commutative?	*opfunc*	*opfunc_r*
+	yes	opAdd	opAdd_r
-	no	opSub	opSub_r
*	yes	opMul	opMul_r

/	no	opDiv	opDiv_r
%	no	opMod	opMod_r
&	yes	opAnd	opAnd_r
|	yes	opOr	opOr_r
^	yes	opXor	opXor_r
<<	no	opShl	opShl_r
>>	no	opShr	opShr_r
>>>	no	opUShr	opUShr_r
~	no	opCat	opCat_r
==	yes	opEquals	-
!=	yes	opEquals	-
<	yes	opCmp	-
<=	yes	opCmp	-
>	yes	opCmp	-
>=	yes	opCmp	-
+=	no	opAddAssign	-
-=	no	opSubAssign	-
*=	no	opMulAssign	-
/=	no	opDivAssign	-
%=	no	opModAssign	-
&=	no	opAndAssign	-
|=	no	opOrAssign	-
^=	no	opXorAssign	-
<<=	no	opShlAssign	-
>>=	no	opShrAssign	-
>>>=	no	opUShrAssign	-
~=	no	opCatAssign	-

Given a binary overloadable operator *op* and its corresponding class or struct member function

name *opfunc* and *opfunc_r*, and the syntax:

```
a op b
```

the following sequence of rules is applied, in order, to determine which form is used:

1. The expression is rewritten as both:

```
a.opfunc(b)
b.opfunc_r(a)
```

5. If any a.opfunc or b.opfunc_r functions exist, then overloading is applied across all of them and the best match is used. If either exist, and there is no argument match, then it is an error.
6. If the operator is commutative, then the following forms are tried:

```
a.opfunc_r(b)
b.opfunc(a)
```

10. If *a* or *b* is a struct or class object reference, it is an error.

Examples

```
class A { int opAdd(int i); }
A a;
a + 1;  // equivalent to a.opAdd(1)
1 + a;  // equivalent to a.opAdd(1)

class B { int opDiv_r(int i); }
B b;
1 / b;  // equivalent to b.opDiv_r(1)

class A { int opAdd(int i); }
class B { int opAdd_r(A a); }
A a;
B b;
a + 1;  // equivalent to a.opAdd(1)
a + b;  // equivalent to b.opAdd_r(a)
b + a;  // equivalent to b.opAdd_r(a)

class A { int opAdd(B b);  int opAdd_r(B b); }
class B { }
A a;
B b;
a + b;  // equivalent to a.opAdd(b)
b + a;  // equivalent to a.opAdd_r(b)
```

```
25.         class A { int opAdd(B b);  int opAdd_r(B b); }
26.         class B { int opAdd_r(A a); }
27.         A a;
28.         B b;
29.         a + b;  // ambiguous: a.opAdd(b) or b.opAdd_r(a)
30.         b + a;  // equivalent to a.opAdd_r(b)
31.
```

Overloading == and !=

Both operators use the **opEquals**() function. The expression (a == b) is rewritten as a.**opEquals**(b), and (a != b) is rewritten as !a.**opEquals**(b).

The member function **opEquals**() is defined as part of Object as:

```
int opEquals(Object o);
```

so that every class object has an **opEquals**().

If a struct has no **opEquals**() function declared for it, a bit compare of the contents of the two structs is done to determine equality or inequality.

Overloading <, <=, > and >=

These comparison operators all use the **opCmp**() function. The expression (a *op* b) is rewritten as (a.**opCmp**(b) *op* 0). The commutative operation is rewritten as (0 *op* b.**opCmp**(a))

The member function **opCmp**() is defined as part of Object as:

```
int opCmp(Object o);
```

so that every class object has a **opCmp**().

If a struct has no **opCmp**() function declared for it, attempting to compare two structs is an error.

Note: Comparing a reference to a class object against null should be done as:

```
if (a === null)
```

and not as:

```
if (a == null)
```

The latter is converted to:

```
if (a.opCmp(null))
```

which will fail if **opCmp**() is a virtual function.

Rationale

The reason for having both **opEquals**() and **opCmp**() is that:

Testing for equality can sometimes be a much more efficient operation than testing for less or greater than.

For some objects, testing for less or greater makes no sense. For these, override **opCmp**() with:

```
class A
{
    int opCmp(Object o)
    {
        assert(0);      // comparison makes no sense
        return 0;
    }
}
```

Function Call Operator Overloading *f*()

The function call operator, (), can be overloaded by declaring a function named **opCall**:

```
struct F
{
    int opCall();
    int opCall(int x, int y, int z);
}

void test()
{   F f;
    int i;

    i = f();            // same as i = f.opCall();
    i = f(3,4,5);       // same as i = a.opCall(3,4,5);
}
```

In this way a struct or class object can behave as if it were a function.

Array Operator Overloading

Overloading Indexing *a*[*i*]

The array index operator, [], can be overloaded by declaring a function named **opIndex** with one or more parameters. Assignment to an array can be overloaded with a function named **opIndexAssign** with two or more parameters. The first parameter is the rvalue of the assignment expression.

```
struct A
{
    int opIndex(int i1, int i2, int i3);
    int opIndexAssign(int value, int i1, int i2);
}

void test()
{   A a;
    int i;

    i = a[5,6,7];       // same as i = a.opIndex(5,6,7);
    a[i,3] = 7;         // same as a.opIndexAssign(7,i,3);
}
```

In this way a struct or class object can behave as if it were an array.

Note: Array index overloading currently does not work for the lvalue of an *op*=, ++, or -- operator.

Overloading Slicing *a*[] and *a*[*i* .. *j*]

Overloading the slicing operator means overloading expressions like `a[]` and `a[i .. j]`.

```
class A
{
    int opSlice();              // overloads a[]
    int opSlice(int x, int y);  // overloads a[i .. j]
}

void test()
{   A a = new A();
    int i;

    i = a[];            // same as i = a.opSlice();
    i = a[3..4];        // same as i = a.opSlice(3,4);
}
```

Future Directions

The operators ., &&, ||, ?:, and a few others will likely never be overloadable. The names of the overloaded operators may change.

Templates

Templates are D's approach to generic programming. Templates are defined with a *TemplateDeclaration*:

```
        TemplateDeclaration:
                template TemplateIdentifier ( TemplateParameterList )
                        { DeclDefs }

        TemplateIdentifier:
                Identifier

        TemplateParameterList
                TemplateParameter
                TemplateParameter , TemplateParameterList

        TemplateParameter:
                TypeParameter
                ValueParameter
                AliasParameter

        TemplateTypeParameter:
                Identifier
                Identifier TemplateTypeParameterSpecialization
                Identifier TemplateTypeParameterDefault
                Identifier TemplateTypeParameterSpecialization
TemplateTypeParameterDefault

        TemplateTypeParameterSpecialization:
                 : Type

        TemplateTypeParameterDefault:
                 = Type

        TemplateValueParameter:
                Declaration
                Declaration TemplateValueParameterSpecialization
                Declaration TemplateValueParameterDefault
                Declaration TemplateValueParameterSpecialization
TemplateValueParameterDefault

        TemplateValueParameterSpecialization:
                 : ConditionalExpression

        TemplateValueParameterDefault:
                 = ConditionalExpression
```

```
TemplateAliasParameter:
        alias Identifier
        alias Identifier TemplateAliasParameterDefault

TemplateAliasParameterDefault:
         = Type
```

The body of the *TemplateDeclaration* must be syntactically correct even if never instantiated. Semantic analysis is not done until instantiated. A template forms its own scope, and the template body can contain classes, structs, types, enums, variables, functions, and other templates.

Template parameters can be types, values, or symbols. Types can be any type. Value parameters must be of an integral type, and specializations for them must resolve to an integral constant. Symbols can be any non-local symbol.

Template parameter specializations constrain the values or types the *TemplateParameter* can accept.

Template parameter defaults are the value or type to use for the *TemplateParameter* in case one is not supplied.

Template Instantiation

Templates are instantiated with:

```
TemplateInstance:
        TemplateIdentifer !( TemplateArgumentList )

TemplateArgumentList:
        TemplateArgument
        TemplateArgument , TemplateArgumentList

TemplateArgument:
        Type
        AssignExpression
        Symbol
```

Once instantiated, the declarations inside the template, called the template members, are in the scope of the *TemplateInstance*:

```
template TFoo(T) { alias T* t; }
...
TFoo!(int).t x; // declare x to be of type int*
```

A template instantiation can be aliased:

```
template TFoo(T) { alias T* t; }
alias TFoo!(int) abc;
abc.t x;        // declare x to be of type int*
```

Multiple instantiations of a *TemplateDeclaration* with the same *TemplateParameterList* all will refer to the same instantiation. For example:

```
template TFoo(T) { T f; }
alias TFoo(int) a;
alias TFoo(int) b;
...
a.f = 3;
assert(b.f == 3);       // a and b refer to the same instance of TFoo
```

This is true even if the *TemplateInstance*s are done in different modules.

If multiple templates with the same *TemplateIdentifier* are declared, they are distinct if they have a different number of arguments or are differently specialized.

For example, a simple generic copy template would be:

```
template TCopy(T)
{
    void copy(out T to, T from)
    {
        to = from;
    }
}
```

To use the template, it must first be instantiated with a specific type:

```
int i;
TCopy!(int).copy(i, 3);
```

Instantiation Scope

*TemplateInstantance*s are always performed in the scope of where the *TemplateDeclaration* is declared, with the addition of the template parameters being declared as aliases for their deduced types.

For example:

```
-------- module a ---------
template TFoo(T) { void bar() { func(); } }

-------- module b ---------
import a;

void func() { }
alias TFoo!(int) f;     // error: func not defined in module a
```

and:

```
-------- module a ---------
template TFoo(T) { void bar() { func(1); } }
```

```
        void func(double d) { }

        -------- module b ---------
        import a;

        void func(int i) { }
        alias TFoo!(int) f;
        ...
        f.bar();        // will call a.func(double)
```

TemplateParameter specializations and default values are evaluated in the scope of the *TemplateDeclaration*.

Argument Deduction

The types of template parameters are deduced for a particular template instantiation by comparing the template argument with the corresponding template parameter.

For each template parameter, the following rules are applied in order until a type is deduced for each parameter:

1. If there is no type specialization for the parameter, the type of the parameter is set to the template argument.
2. If the type specialization is dependent on a type parameter, the type of that parameter is set to be the corresponding part of the type argument.
3. If after all the type arguments are examined there are any type parameters left with no type assigned, they are assigned types corresponding to the template argument in the same position in the *TemplateArgumentList*.
4. If applying the above rules does not result in exactly one type for each template parameter, then it is an error.

For example:

```
        template TFoo(T) { }
        alias TFoo!(int) Foo1;          // (1) T is deduced to be int
        alias TFoo!(char*) Foo2;        // (1) T is deduced to be char*

        template TFoo(T : T*) { }
        alias TFoo!(char*) Foo3;        // (2) T is deduced to be char

        template TBar(D, U : D[]) { }
        alias TBar!(int, int[]) Bar1;   // (2) D is deduced to be int, U is
int[]
        alias TBar!(char, int[]) Bar2;  // (4) error, D is both char and int

        template TBar(D : E*, E) { }
        alias TBar!(int*, int) Bar3;    // (1) E is int
                                        // (3) D is int*
```

When considering matches, a class is considered to be a match for any super classes or interfaces:

```
class A { }
class B : A { }

template TFoo(T : A) { }
alias TFoo!(B) Foo4;            // (3) T is B

template TBar(T : U*, U : A) { }
alias TBar!(B*, B) Foo5;        // (2) T is B*
                                // (3) U is B
```

Value Parameters

This example of template foo has a value parameter that is specialized for 10:

```
template foo(U : int, int T : 10)
{
    U x = T;
}

void main()
{
    assert(foo!(int, 10).x == 10);
}
```

Specialization

Templates may be specialized for particular types of arguments by following the template parameter identifier with a : and the specialized type. For example:

```
template TFoo(T)        { ... } // #1
template TFoo(T : T[])  { ... } // #2
template TFoo(T : char) { ... } // #3
template TFoo(T,U,V)    { ... } // #4

alias TFoo!(int) foo1;          // instantiates #1
alias TFoo!(double[]) foo2;     // instantiates #2 with T being double
alias TFoo!(char) foo3;         // instantiates #3
alias TFoo!(char, int) fooe;    // error, number of arguments mismatch
alias TFoo!(char, int, int) foo4; // instantiates #4
```

The template picked to instantiate is the one that is most specialized that fits the types of the *TemplateArgumentList*. Determine which is more specialized is done the same way as the C++ partial ordering rules. If the result is ambiguous, it is an error.

Alias Parameters

Alias parameters enable templates to be parameterized with any type of D symbol, including

global names, type names, module names, template names, and template instance names. Local names may not be used as alias parameters. It is a superset of the uses of template template parameters in C++.

Global names

```
int x;

template Foo(alias X)
{
    static int* p = &X;
}

void test()
{
    alias Foo!(x) bar;
    *bar.p = 3;         // set x to 3
    int y;
    alias Foo!(y) abc; // error, y is local name
}
```

Type names

```
class Foo
{
    static int p;
}

template Bar(alias T)
{
    alias T.p q;
}

void test()
{
    alias Bar!(Foo) bar;
    bar.q = 3;  // sets Foo.p to 3
}
```

Module names

```
import std.string;
```

```
template Foo(alias X)
{
        alias X.toString y;
}

void test()
{
    alias Foo!(std.string) bar;
    bar.y(3);   // calls std.string.toString(3)
}
```

Template names

```
int x;

template Foo(alias X)
{
    static int* p = &X;
}

template Bar(alias T)
{
    alias T!(x) abc;
}

void test()
{
    alias Bar!(Foo) bar;
    *bar.abc.p = 3;     // sets x to 3
}
```

Template alias names

```
int x;

template Foo(alias X)
{
    static int* p = &X;
}

template Bar(alias T)
```

```
{
    alias T.p q;
}

void test()
{
    alias Foo!(x) foo;
    alias Bar!(foo) bar;
    *bar.q = 3;         // sets x to 3
}
```

Template Parameter Default Values

Trailing template parameters can be given default values:

```
template Foo(T, U = int) { ... }
Foo!(uint,long); // instantiate Foo with T as uint, and U as long
Foo!(uint);      // instantiate Foo with T as uint, and U as int

template Foo(T, U = T*) { ... }
Foo!(uint);      // instantiate Foo with T as uint, and U as uint*
```

Implicit Template Properties

If a template has exactly one member in it, and the name of that member is the same as the template name, that member is assumed to be referred to in a template instantiation:

```
template Foo(T)
{
    T Foo;      // declare variable Foo of type T
}

void test()
{
    Foo!(int) = 6;      // instead of Foo!(int).Foo
}
```

Class Templates

ClassTemplateDeclaration:
        class *Identifier* **(** *TemplateParameterList* **)** [*SuperClass* {, *InterfaceClass* }] *ClassBody*

If a template declares exactly one member, and that member is a class with the same name as the template:

```
template Bar(T)
{
    class Bar
    {
        T member;
    }
}
```

then the semantic equivalent, called a *ClassTemplateDeclaration* can be written as:

```
class Bar(T)
{
    T member;
}
```

Recursive Templates

Template features can be combined to produce some interesting effects, such as compile time evaluation of non-trivial functions. For example, a factorial template can be written:

```
template factorial(int n : 1)
{
    enum { factorial = 1 }
}

template factorial(int n)
{
    // Note . used to find global template rather than enum
    enum { factorial = n* .factorial!(n-1) }
}

void test()
{
    printf("%d\n", factorial!(4));      // prints 24
}
```

Limitations

Templates cannot be used to add non-static members or functions to classes. For example:

```
class Foo
{
    template TBar(T)
    {
        T xx;                   // Error
        int func(T) { ... }     // Error
```

```
            static T yy;                            // Ok
            static int func(T t, int y) { ... }     // Ok
        }
    }
```

Templates cannot be declared inside functions.

Mixins

Mixins mean different things in different programming languages. In D, a mixin takes an arbitrary set of declarations from the body of a *TemplateDeclaration* and inserts them into the current context.

```
    TemplateMixin:
            mixin TemplateIdentifier ;
            mixin TemplateIdentifier MixinIdentifier ;
            mixin TemplateIdentifier !( TemplateArgumentList ) ;
            mixin TemplateIdentifier !( TemplateArgumentList )
MixinIdentifier ;

    MixinIdentifier:
            Identifier
```

A *TemplateMixin* can occur in declaration lists of modules, classes, structs, unions, and as a statement. The *TemplateIdentifier* refers to a *TemplateDeclaration*. If the *TemplateDeclaration* has no parameters, the mixin form that has no **!(***TemplateArgumentList***)** can be used.

Unlike a template instantiation, a template mixin's body is evaluated within the scope where the mixin appears, not where the template declaration is defined. It is analogous to cutting and pasting the body of the template into the location of the mixin. It is useful for injecting parameterized 'boilerplate' code, as well as for creating templated nested functions, which is not possible with template instantiations.

```
    template Foo()
    {
        int x = 5;
    }

    mixin Foo;

    struct Bar
    {
        mixin Foo;
    }

    void test()
    {
        printf("x = %d\n", x);              // prints 5
```

```
    {   Bar b;
        int x = 3;

        printf("b.x = %d\n", b.x);      // prints 5
        printf("x = %d\n", x);          // prints 3
        {
            mixin Foo;
            printf("x = %d\n", x);      // prints 5
            x = 4;
            printf("x = %d\n", x);      // prints 4
        }
        printf("x = %d\n", x);          // prints 3
    }
    printf("x = %d\n", x);              // prints 5
}
```

Mixins can be parameterized:

```
template Foo(T)
{
    T x = 5;
}

mixin Foo!(int);                // create x of type int
```

Mixins can add virtual functions to a class:

```
template Foo()
{
    void func() { printf("Foo.func()\n"); }
}

class Bar
{
    mixin Foo;
}

class Code : Bar
{
    void func() { printf("Code.func()\n"); }
}

void test()
{
    Bar b = new Bar();
    b.func();           // calls Foo.func()

    b = new Code();
    b.func();           // calls Code.func()
}
```

Mixins are evaluted in the scope of where they appear, not the scope of the template declaration:

```
int y = 3;
```

```
template Foo()
{
    int abc() { return y; }
}

void test()
{
    int y = 8;
    mixin Foo;  // local y is picked up, not global y
    assert(abc() == 8);
}
```

Mixins can parameterize symbols using alias parameters:

```
template Foo(alias b)
{
    int abc() { return b; }
}

void test()
{
    int y = 8;
    mixin Foo!(y);
    assert(abc() == 8);
}
```

This example uses a mixin to implement a generic Duff's device for an arbitrary statement (in this case, the arbitrary statement is in bold). A nested function is generated as well as a delegate literal, these can be inlined by the compiler:

```
template duffs_device(alias id1, alias id2, alias s)
{
    void duff_loop()
    {
        if (id1 < id2)
        {
            typeof(id1) n = (id2 - id1 + 7) / 8;
            switch ((id2 - id1) % 8)
            {
                case 0:         do {  s();
                case 7:               s();
                case 6:               s();
                case 5:               s();
                case 4:               s();
                case 3:               s();
                case 2:               s();
                case 1:               s();
                                   } while (--n > 0);
            }
        }
    }
}
```

```
void foo() { printf("foo\n"); }

void test()
{
    int i = 1;
    int j = 11;

    mixin duffs_device!(i, j, delegate { foo(); } );
    duff_loop();        // executes foo() 10 times
}
```

Mixin Scope

The declarations in a mixin are 'imported' into the surrounding scope. If the name of a declaration in a mixin is the same as a declaration in the surrounding scope, the surrounding declaration overrides the mixin one:

```
int x = 3;

template Foo()
{
    int x = 5;
    int y = 5;
}

mixin Foo;
int y = 3;

void test()
{
    printf("x = %d\n", x);      // prints 3
    printf("y = %d\n", y);      // prints 3
}
```

If two different mixins are put in the same scope, and each define a declaration with the same name, there is an ambiguity error when the declaration is referenced:

```
template Foo()
{
    int x = 5;
}

template Bar()
{
    int x = 4;
}

mixin Foo;
mixin Bar;

void test()
{
```

```
    printf("x = %d\n", x);      // error, x is ambiguous
}
```

If a mixin has a *MixinIdentifier*, it can be used to disambiguate:

```
int x = 6;

template Foo()
{
    int x = 5;
    int y = 7;
}

template Bar()
{
    int x = 4;
}

mixin Foo F;
mixin Bar B;

void test()
{
    printf("y = %d\n", y);      // prints 7
    printf("x = %d\n", x);      // prints 6
    printf("F.x = %d\n", F.x);  // prints 5
    printf("B.x = %d\n", B.x);  // prints 4
}
```

A mixin has its own scope, even if a declaration is overridden by the enclosing one:

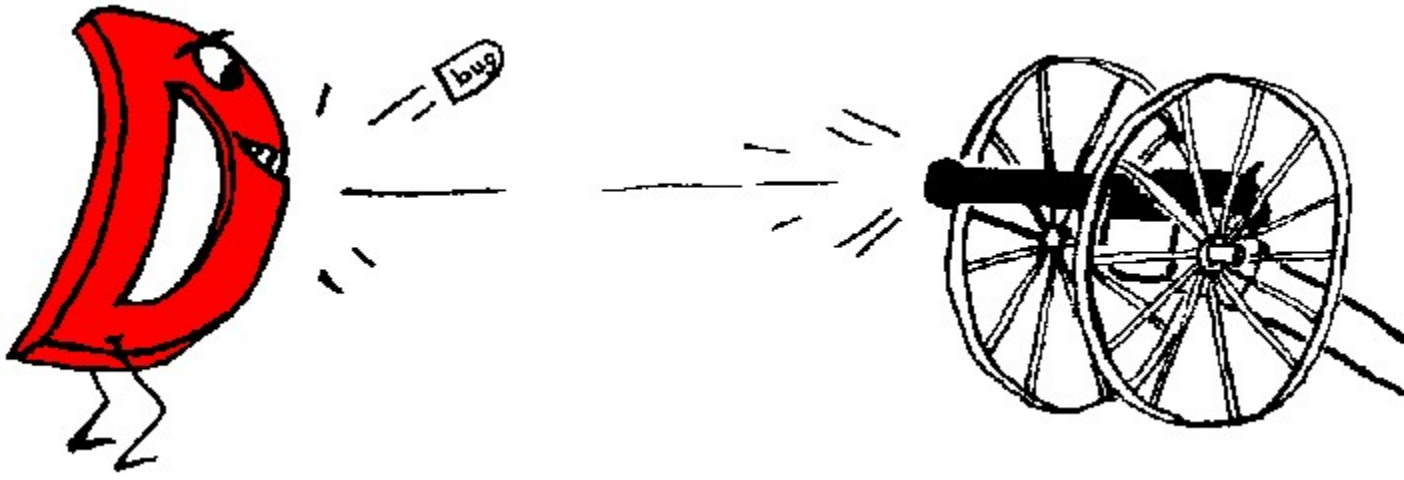
```
int x = 4;

template Foo()
{
    int x = 5;
    int bar() { return x; }
}

mixin Foo;

void test()
{
    printf("x = %d\n", x);              // prints 4
    printf("bar() = %d\n", bar());      // prints 5
}
```

Contract Programming

Contracts are a breakthrough technique to reduce the programming effort for large projects. Contracts are the concept of preconditions, postconditions, errors, and invariants. Contracts can be done in C++ without modification to the language, but the result is clumsy and inconsistent.

Building contract support into the language makes for:

1. a consistent look and feel for the contracts
2. tool support
3. it's possible the compiler can generate better code using information gathered from the contracts
4. easier management and enforcement of contracts
5. handling of contract inheritance



The idea of a contract is simple - it's just an expression that must evaluate to true. If it does not, the contract is broken, and by definition, the program has a bug in it. Contracts form part of the specification for a program, moving it from the documentation to the code itself. And as every programmer knows, documentation tends to be incomplete, out of date, wrong, or non-existent. Moving the contracts into the code makes them verifiable against the program.

Assert Contract

The most basic contract is the assert. An assert inserts a checkable expression into the code, and that expression must evaluate to true:

```
assert(expression);
```

C programmers will find it familiar. Unlike C, however, an `assert` in function bodies works by throwing an `AssertException`, which can be caught and handled. Catching the contract violation is useful when the code must deal with errant uses by other code, when it must be failure proof, and as a useful tool for debugging.

Pre and Post Contracts

The pre contracts specify the preconditions before a statement is executed. The most typical use of this would be in validating the parameters to a function. The post contracts validate the result of the statement. The most typical use of this would be in validating the return value of a function and of any side effects it has. The syntax is:

```
in
{
    ...contract preconditions...
}
out (result)
{
    ...contract postconditions...
}
body
{
    ...code...
}
```

By definition, if a pre contract fails, then the body received bad parameters. An InException is thrown. If a post contract fails, then there is a bug in the body. An OutException is thrown.

Either the `in` or the `out` clause can be omitted. If the `out` clause is for a function body, the variable `result` is declared and assigned the return value of the function. For example, let's implement a square root function:

```
long square_root(long x)
    in
    {
        assert(x >= 0);
    }
    out (result)
    {
        assert((result * result) == x);
    }
    body
    {
        return math.sqrt(x);
    }
```

The assert's in the in and out bodies are called contracts. Any other D statement or expression is allowed in the bodies, but it is important to ensure that the code has no side effects, and that the release version of the code will not depend on any effects of the code. For a release build of the code, the in and out code is not inserted.

If the function returns a void, there is no result, and so there can be no result declaration in the out clause. In that case, use:

```
void func()
   out
   {
        ...contracts...
   }
```

```
body
{
      ...
}
```

In an out statement, *result* is initialized and set to the return value of the function.

The compiler can be adjusted to verify that every in and inout parameter is referenced in the `in { }`, and every out and inout parameter is referenced in the `out { }`.

The in-out statement can also be used inside a function, for example, it can be used to check the results of a loop:

```
in
{
    assert(j == 0);
}
out
{
    assert(j == 10);
}
body
{
    for (i = 0; i < 10; i++)
        j++;
}
```

This is not implemented at this time.

In, Out and Inheritance

If a function in a derived class overrides a function in its super class, then only one of the `in` contracts of the base functions must be satisified Overriding functions then becomes a process of *loosening* the `in` contracts.

Conversely, all of the `out` contracts needs to be satisified, so overriding functions becomes a processes of *tightening* the `out` contracts.

Class Invariants

Class invariants are used to specify characteristics of a class that always must be true (except while executing a member function). They are described in Classes.

References

Contracts Reading List
Adding Contracts to Java

Debug, Version, and Static Assert

D supports building multiple versions and various debug builds from the same source code using the features:

```
DebugSpecification
DebugAttribute
DebugStatement

VersionSpecification
VersionAttribute
VersionStatement

StaticAssert
```

Predefined Versions

Several environmental version identifiers and identifier name spaces are predefined to encourage consistent usage. Version identifiers do not conflict with other identifiers in the code, they are in a separate name space. Predefined version identifiers are global, i.e. they apply to all modules being compiled and imported.

DigitalMars
: Digital Mars is the compiler vendor

X86
: Intel and AMD 32 bit processors

AMD64
: AMD 64 bit processors

Windows
: Microsoft Windows systems

Win32
: Microsoft 32 bit Windows systems

Win64
: Microsoft 64 bit Windows systems

linux
: All linux systems

LittleEndian
: Byte order, least significant first

BigEndian
: Byte order, most significant first

D_InlineAsm
: Inline assembler is implemented

none
: Never defined; used to just disable a section of code

all
: Always defined; used as the opposite of **none**

Others will be added as they make sense and new implementations appear.

It is inevitable that the D language will evolve over time. Therefore, the version identifier namespace beginning with "D_" is reserved for identifiers indicating D language specification or new feature conformance.

Furthermore, predefined version identifiers from this list cannot be set from the command line or from version statements. (This prevents things like both **Windows** and **linux** being simultaneously set.)

Compiler vendor specific versions can be predefined if the trademarked vendor identifier prefixes it, as in:

```
version(DigitalMars_funky_extension)
{
    ...
}
```

It is important to use the right version identifier for the right purpose. For example, use the vendor identifier when using a vendor specific feature. Use the operating system identifier when using an operating system specific feature, etc.

Specification

```
DebugSpecification
    debug = Identifier ;
    debug = Integer ;

VersionSpecification
    version = Identifier ;
    version = Integer ;
```

Version specifications do not declare any symbols, but instead set a version in the same manner that the **-version** does on the command line. The version specification is used for conditional compilation with version attributes and version statements.

*VersionSpecification*s and *DebugSpecification*s apply only to the module they appear in. The only global ones are the predefined ones and any that are specified on the command line.

The version specification makes it straightforward to group a set of features under one major version, for example:

```
version (ProfessionalEdition)
{
    version = FeatureA;
    version = FeatureB;
    version = FeatureC;
}
version (HomeEdition)
{
    version = FeatureA;
}
```

```
...
version (FeatureB)
{
    ... implement Feature B ...
}
```

Debug Statement

Two versions of programs are commonly built, a release build and a debug build. The debug build commonly includes extra error checking code, test harnesses, pretty-printing code, etc. The debug statement conditionally compiles in its statement body. It is D's way of what in C is done with #ifdef DEBUG / #endif pairs.

```
DebugStatement:
        DebugPredicate Statement
        DebugPredicate Statement else Statement

DebugPredicate
        debug Statement
        debug ( Integer )
        debug ( Identifier )
```

Debug statements are compiled in when the -debug switch is thrown on the compiler.

debug(*Integer*) statements are compiled in when the debug level is >= *Integer*.

debug(*Identifier*) statements are compiled in when the debug identifier matches *Identifier*.

If *Statement* is a block statement, it does not introduce a new scope. For example:

```
int k;
debug
{   int i;
    int k;      // error, k already defined

    i = 3;
}
x = i;          // uses the i declared above
```

Version Statement

It is commonplace to conveniently support multiple versions of a module with a single source file. While the D way is to isolate all versioning into separate modules, that can get burdensome if it's just simple line change, or if the entire program would otherwise fit into one module.

```
VersionStatement:
        VersionPredicate Statement
        VersionPredicate Statement else Statement
```

```
VersionPredicate
        version ( Integer )
        version ( Identifier )
```

The version statement conditionally compiles in its statement body based on the version specified by the *Integer* of *Identifier*. Both forms are set by the **-version** switch to the compiler. If *Statement* is a block statement, it does not introduce a new scope. For example:

```
int k;
version (Demo)  // compile in this code block for the demo version
{   int i;
    int k;      // error, k already defined

    i = 3;
}
x = i;          // uses the i declared above
```

The version statement works together with the version attribute for declarations.

Version statements can nest.

The optional else clause gets conditionally compiled in if the version predicate is false:

```
version (X86)
{
    ... // implement custom inline assembler version
}
else
{
    ... // use default, but slow, version
}
```

While the debug and version statements superficially behave the same, they are intended for very different purposes. Debug statements are for adding debug code that is removed for the release version. Version statements are to aid in portability and multiple release versions.

Debug Attribute

```
DebugAttribute:
    debug
    debug ( Integer )
    debug ( Identifier )
```

Two versions of programs are commonly built, a release build and a debug build. The debug build includes extra error checking code, test harnesses, pretty-printing code, etc. The debug attribute conditionally compiles in code:

```
class Foo
{
        int a, b;
```

```
            debug:
                int flag;
        }
```

Conditional Compilation means that if the code is not compiled in, it still must be syntactically correct, but no semantic checking or processing is done on it. No symbols are defined, no typechecking is done, no code is generated, no imports are imported. Various different debug builds can be built with a parameter to debug:

```
        debug(Integer) { }      // add in debug code if debug level is >=
Integer
        debug(identifier) { } // add in debug code if debug keyword is
identifier
```

These are presumably set by the command line as `-debug=`*`n`* and `-debug=`*`identifier`*.

Version Attribute

```
        VersionAttribute:
            version ( Integer )
            version ( Identifier )
```

The version attribute is very similar to the debug attribute, and in many ways is functionally interchangeable with it. The purpose of it, however, is different. While debug is for building debugging versions of a program, version is for using the same source to build multiple release versions.

For instance, there may be a *full* version as opposed to a *demo* version:

```
        class Foo
        {
            int a, b;

            version(full)
            {
                int extrafunctionality()
                {
                    ...
                    return 1;           // extra functionality is supported
                }
            }
            else // demo
            {
                int extrafunctionality()
                {
                    return 0;           // extra functionality is not
supported
                }
            }
        }
```

Various different version builds can be built with a parameter to version:

```
        version(n) { }  // add in version code if version level is >= n
        version(identifier) { } // add in version code if version keyword is
identifier
```

These are presumably set by the command line as `-version=`*n* and `-version=`*identifier*.

Static Assert

```
        StaticAssert:
            static assert ( Expression );
```

Expression is evaluated at compile time, and converted to a boolean value. If the value is true, the static assert is ignored. If the value is false, an error diagnostic is issued and the compile fails.

Unlike *AssertExpression*s, *StaticAssert*s are always checked and evaluted by the compiler unless they appear in a false debug or version conditional.

```
        void foo()
        {
            if (0)
            {
                assert(0);              // never trips
                static assert(0);       // always trips
            }
            version (BAR)
            {
                static assert(0);       // does not trip unless BAR is defined
            }
        }
```

Error Handling in D

All programs have to deal with errors. Errors are unexpected conditions that are not part of the normal operation of a program. Examples of common errors are:

Out of memory.
Out of disk space.
Invalid file name.
Attempting to write to a read-only file.
Attempting to read a non-existent file.
Requesting a system service that is not supported.

The Error Handling Problem

The traditional C way of detecting and reporting errors is not traditional, it is ad-hoc and varies from function to function, including:

Returning a NULL pointer.
Returning a 0 value.
Returning a non-zero error code.
Requiring errno to be checked.
Requiring that a function be called to check if the previous function failed.

To deal with these possible errors, tedious error handling code must be added to each function call. If an error happened, code must be written to recover from the error, and the error must be reported to the user in some user friendly fashion. If an error cannot be handled locally, it must be explicitly propagated back to its caller. The long list of errno values needs to be converted into appropriate text to be displayed. Adding all the code to do this can consume a large part of the time spent coding a project - and still, if a new errno value is added to the runtime system, the old code can not properly display a meaningful error message.

Good error handling code tends to clutter up what otherwise would be a neat and clean looking implementation.

Even worse, good error handling code is itself error prone, tends to be the least tested (and therefore buggy) part of the project, and is frequently simply omitted. The end result is likely a "blue screen of death" as the program failed to deal with some unanticipated error.

Quick and dirty programs are not worth writing tedious error handling code for, and so such utilities tend to be like using a table saw with no blade guards.

What's needed is an error handling philosophy and methodology that is:

Standardized - consistent usage makes it more useful.
Produces a reasonable result even if the programmer fails to check for errors.
Allows old code to be reused with new code without having to modify the old code to be compatible with new error types.
No errors get inadvertently ignored.
Allows 'quick and dirty' utilities to be written that still correctly handle errors.
Easy to make the error handling source code look good.

The D Error Handling Solution

Let's first make some observations and assumptions about errors:

Errors are not part of the normal flow of a program. Errors are exceptional, unusual, and unexpected.
Because errors are unusual, execution of error handling code is not performance critical.
The normal flow of program logic is performance critical.
All errors must be dealt with in some way, either by code explicitly written to handle them, or by some system default handling.

> The code that detects an error knows more about the error than the code that must recover from the error.

The solution is to use exception handling to report errors. All errors are objects derived from abstract class Error. class Error has a pure virtual function called toString() which produces a char[] with a human readable description of the error.

If code detects an error like "out of memory," then an Error is thrown with a message saying "Out of memory". The function call stack is unwound, looking for a handler for the Error. Finally blocks are executed as the stack is unwound. If an error handler is found, execution resumes there. If not, the default Error handler is run, which displays the message and terminates the program.

How does this meet our criteria?

Standardized - consistent usage makes it more useful.
> This is the D way, and is used consistently in the D runtime library and examples.

Produces a reasonable result even if the programmer fails to check for errors.
> If no catch handlers are there for the errors, then the program gracefully exits through the default error handler with an appropriate message.

Allows old code to be reused with new code without having to modify the old code to be compatible with new error types.
> Old code can decide to catch all errors, or only specific ones, propagating the rest upwards. In any case, there is no more need to correlate error numbers with messages, the correct message is always supplied.

No errors get inadvertently ignored.
> Error exceptions get handled one way or another. There is nothing like a NULL pointer return indicating an error, followed by trying to use that NULL pointer.

Allows 'quick and dirty' utilities to be written that still correctly handle errors.
> Quick and dirty code need not write any error handling code at all, and don't need to check for errors. The errors will be caught, an appropriate message displayed, and the program gracefully shut down all by default.

Easy to make the error handling source code look good.
> The try/catch/finally statements look a lot nicer than endless if (error) goto errorhandler; statements.

How does this meet our assumptions about errors?

Errors are not part of the normal flow of a program. Errors are exceptional, unusual, and unexpected.
> D exception handling fits right in with that.

Because errors are unusual, execution of error handling code is not performance critical.
> Exception handling stack unwinding is a relatively slow process.

The normal flow of program logic is performance critical.
> Since the normal flow code does not have to check every function call for error returns, it can be realistically faster to use exception handling for the errors.

All errors must be dealt with in some way, either by code explicitly written to handle them, or by some system default handling.
> If there's no handler for a particular error, it is handled by the runtime library default

handler. If an error is ignored, it is because the programmer specifically added code to ignore an error, which presumably means it was intentional.

The code that detects an error knows more about the error than the code that must recover from the error.

There is no more need to translate error codes into human readable strings, the correct string is generated by the error detection code, not the error recovery code. This also leads to consistent error messages for the same error between applications.

Garbage Collection

D is a fully garbage collected language. That means that it is never necessary to free memory. Just allocate as needed, and the garbage collector will periodically return all unused memory to the pool of available memory.

C and C++ programmers accustomed to explicitly managing memory allocation and deallocation will likely be skeptical of the benefits and efficacy of garbage collection. Experience both with new projects written with garbage collection in mind, and converting existing projects to garbage collection shows that:

Garbage collected programs are faster. This is counterintuitive, but the reasons are:

Reference counting is a common solution to solve explicit memory allocation problems. The code to implement the increment and decrement operations whenever assignments are made is one source of slowdown. Hiding it behind smart pointer classes doesn't help the speed. (Reference counting methods are not a general solution anyway, as circular references never get deleted.)

Destructors are used to deallocate resources acquired by an object. For most classes, this resource is allocated memory. With garbage collection, most destructors then become empty and can be discarded entirely.

All those destructors freeing memory can become significant when objects are allocated on the stack. For each one, some mechanism must be established so that if an exception happens, the destructors all get called in each frame to release any memory they hold. If the destructors become irrelevant, then there's no need to set up special stack frames to handle exceptions, and the code runs faster.

All the code necessary to manage memory can add up to quite a bit. The larger a program is, the less in the cache it is, the more paging it does, and the slower it runs.

Garbage collection kicks in only when memory gets tight. When memory is not tight, the program runs at full speed and does not spend any time freeing memory.

Modern garbage collectors are far more advanced now than the older, slower ones. Generational, copying collectors eliminate much of the inefficiency of early mark and sweep algorithms.

Modern garbage collectors do heap compaction. Heap compaction tends to reduce the number of pages actively referenced by a program, which means that memory accesses are more likely to be cache hits and less swapping.

Garbage collected programs do not suffer from gradual deterioration due to an

accumulation of memory leaks.
Garbage collectors reclaim unused memory, therefore they do not suffer from "memory leaks" which can cause long running applications to gradually consume more and more memory until they bring down the system. GC'd programs have longer term stability.
Garbage collected programs have fewer hard-to-find pointer bugs. This is because there are no dangling references to free'd memory. There is no code to explicitly manage memory, hence no bugs in such code.
Garbage collected programs are faster to develop and debug, because there's no need for developing, debugging, testing, or maintaining the explicit deallocation code.
Garbage collected programs can be significantly smaller, because there is no code to manage deallocation, and there is no need for exception handlers to deallocate memory.

Garbage collection is not a panacea. There are some downsides:

It is not predictable when a collection gets run, so the program can arbitrarily pause.
The time it takes for a collection to run is not bounded. While in practice it is very quick, this cannot be guaranteed.
All threads other than the collector thread must be halted while the collection is in progress.
Garbage collectors can keep around some memory that an explicit deallocator would not. In practice, this is not much of an issue since explicit deallocators usually have memory leaks causing them to eventually use far more memory, and because explicit deallocators do not normally return deallocated memory to the operating system anyway, instead just returning it to its own internal pool.
Garbage collection should be implemented as a basic operating system kernel service. But since they are not, garbage collecting programs must carry around with them the garbage collection implementation. While this can be a shared DLL, it is still there.

These constraints are addressed by techniques outlined in Memory Management.

How Garbage Collection Works

To be written...

Interfacing Garbage Collected Objects With Foreign Code

The garbage collector looks for roots in its static data segment, and the stacks and register contents of each thread. If the only root of an object is held outside of this, then the collecter will miss it and free the memory.

To avoid this from happening,

Maintain a root to the object in an area the collector does scan for roots.
Reallocate the object using the foreign code's storage allocator or using the C runtime library's malloc/free.

Pointers and the Garbage Collector

Pointers in D can be broadly divided into two categories: those that point to garbage collected memory, and those that do not. Examples of the latter are pointers created by calls to C's malloc(), pointers received from C library routines, pointers to static data, pointers to objects on the stack, etc. For those pointers, anything that is legal in C can be done with them.

For garbage collected pointers and references, however, there are some restrictions. These restrictions are minor, but they are intended to enable the maximum flexibility in garbage collector design.

Undefined behavior:

Pointers xor'd them with other values, like the xor'd pointer linked list trick used in C. Do not use the xor trick to swap two pointer values.

Storing pointers into non-pointer variables using casts and other tricks.

```
void* p;
...
int x = cast(int)p;     // error: undefined behavior
```

The garbage collector does not scan non-pointer types for roots.

Taking advantage of alignment of pointers to store bit flags in the low order bits; storing bit flags in the high order bits:

```
p = cast(void*)(cast(int)p | 1);        // error: undefined
behavior
```

Storing values that may point into the garbage collected heap into pointers:

```
p = cast(void*)12345678;        // error: undefined behavior
```

Do not store magic values into pointers, other than `null`.

Writing pointer values out to disk and reading them back in again.

Using pointer values to compute a hash function. A copying garbage collector can arbitrarily move objects around in memory, thus invalidating the computed hash value.

Depending on the ordering of pointers:

```
if (p1 < p2)            // error: undefined behavior
    ...
```

since, again, the garbage collector can move objects around in memory.

Adding or subtracting an offset to a pointer such that the result points outside of the bounds of the garbage collected object originally allocated.

```
char* p = new char[10];
char* q = p + 6;        // ok
q = p + 11;             // error: undefined behavior
q = p - 1;              // error: undefined behavior
```

Things that are reliable and can be done:

Use a union to share storage with a pointer:

```
union U { void* ptr; int value }
```

Using such a union, however, as a substitute for a cast(int) will result in undefined behavior.

A pointer to the start of a garbage collected object need not be maintained if a pointer to the interior of the object exists.

```
char[] p = new char[10];
char[] q = p[3..6];
// q is enough to hold on to the object, don't need to keep
// p as well.
```

One can avoid using pointers anyway for most tasks. D provides features rendering most explicit pointer uses obsolete, such as reference objects, dynamic arrays, and garbage collection. Pointers are provided in order to interface successfully with C API's and for some low level work.

Working with the Garbage Collector

Garbage collection doesn't solve every memory deallocation problem. For example, if a root to a large data structure is kept, the garbage collector cannot reclaim it, even if it is never referred to again. To eliminate this problem, it is good practice to set a reference or pointer to an object to null when no longer needed.

This advice applies only to static references or references embedded inside other objects. There is not much point for such stored on the stack to be nulled, since the collector doesn't scan for roots past the top of the stack, and because new stack frames are initialized anyway.

Memory Management

Any non-trivial program needs to allocate and free memory. Memory management techniques become more and more important as programs increase in complexity, size, and performance. D

offers many options for managing memory.

The three primary methods for allocating memory in D are:

1. Static data, allocated in the default data segment.
2. Stack data, allocated on the CPU program stack.
3. Garbage collected data, allocated dynamically on the garbage collection heap.

This chapter describes techniques for using them, as well as some advanced alternatives:

Strings (and Array) Copy-on-Write

Consider the case of passing an array to a function, possibly modifying the contents of the array, and returning the modified array. Since arrays are passed by reference, not by value, a crucial issue is who owns the contents of the array? For example, a function to convert an array of characters to upper case:

```
char[] toupper(char[] s)
{
    int i;

    for (i = 0; i < s.length; i++)
    {
        char c = s[i];
        if ('a' <= c && c <= 'z')
            s[i] = c - (cast(char)'a' - 'A');
    }
    return s;
}
```

Note that the caller's version of s[] is also modified. This may be not at all what was intended, or worse, s[] may be a slice into a read-only section of memory.

If a copy of s[] was always made by toupper(), then that will unnecessarily consume time and memory for strings that are already all upper case.

The solution is to implement copy-on-write, which means that a copy is made only if the string needs to be modified. Some string processing languages do do this as the default behavior, but there is a huge cost to it. The string "abcdeF" will wind up being copied 5 times by the function. To get the maximum efficiency using the protocol, it'll have to be done explicitly in the code.

Here's toupper() rewritten to implement copy-on-write in an efficient manner:

```
char[] toupper(char[] s)
{
    int changed;
    int i;

    changed = 0;
    for (i = 0; i < s.length; i++)
    {
        char c = s[i];
        if ('a' <= c && c <= 'z')
        {
            if (!changed)
            {   char[] r = new char[s.length];
                r[] = s;
                s = r;
                changed = 1;
            }
            s[i] = c - (cast(char)'a' - 'A');
        }
    }
    return s;
}
```

Copy-on-write is the protocol implemented by array processing functions in the D Phobos runtime library.

Real Time

Real time programming means that a program must be able to guarantee a maximum latency, or time to complete an operation. With most memory allocation schemes, including malloc/free and garbage collection, the latency is theoretically not bound. The most reliable way to guarantee latency is to preallocate all data that will be needed by the time critical portion. If no calls to allocate memory are done, the gc will not run and so will not cause the maximum latency to be exceeded.

Smooth Operation

Related to real time programming is the need for a program to operate smoothly, without arbitrary pauses while the garbage collector stops everything to run a collection. An example of such a program would be an interactive shooter type game. Having the game play pause erratically, while not fatal to the program, can be annoying to the user. There are several techniques to eliminate or mitigate the effect:

Preallocate all data needed before the part of the code that needs to be smooth is run.

Manually run a gc collection cycle at points in program execution where it is already paused. An example of such a place would be where the program has just displayed a prompt for user input and the user has not responded yet. This reduces the odds that a collection cycle will be needed during the smooth code.

Call gc.disable() before the smooth code is run, and gc.enable() afterwards. This will cause the gc to favor allocating more memory instead of running a collection pass.

Free Lists

Free lists are a great way to accelerate access to a frequently allocated and discarded type. The idea is simple - instead of deallocating an object when done with it, put it on a free list. When allocating, pull one off the free list first.

```
class Foo
{
    static Foo freelist;                // start of free list

    static Foo allocate()
    {   Foo f;

        if (freelist)
        {   f = freelist;
            freelist = f.next;
        }
        else
            f = new Foo();
        return f;
    }

    static void deallocate(Foo f)
    {
        f.next = freelist;
        freelist = f;
    }

    Foo next;           // for use by FooFreeList
    ...
}

void test()
{
    Foo f = Foo.allocate();
    ...
    Foo.deallocate(f);
}
```

Such free list approaches can be very high performance.

If used by multiple threads, the allocate() and deallocate() functions need to be synchronized.
The Foo constructor is not re-run by allocate() when allocating from the free list, so the allocator may need to reinitialize some of the members.
It is not necessary to practice RAII with this, since if any objects are not passed to deallocate() when done, because of a thrown exception, they'll eventually get picked up by the gc anyway.

Reference Counting

The idea behind reference counting is to include a count field in the object. Increment it for each additional reference to it, and decrement it whenever a reference to it ceases. When the count hits 0, the object can be deleted.

D doesn't provide any automated support for reference counting, it will have to be done explicitly.

Win32 COM programming uses the members AddRef() and Release() to maintain the reference counts.

Explicit Class Instance Allocation

D provides a means of creating custom allocators and deallocators for class instances. Normally, these would be allocated on the garbage collected heap, and deallocated when the collector decides to run. For specialized purposes, this can be handled by creating *NewDeclaration*s and *DeleteDeclaration*s. For example, to allocate using the C runtime library's `malloc` and `free`:

```
import std.c.stdlib;
import std.outofmemory;
import std.gc;

class Foo
{
    new(uint sz)
    {
        void* p;

        p = std.c.stdlib.malloc(sz);
        if (!p)
            throw new OutOfMemory();
        gc.addRange(p, p + sz);
        return p;
    }

    delete(void* p)
    {
        if (p)
        {   gc.removeRange(p);
            std.c.stdlib.free(p);
        }
    }
}
```

The critical features of new() are:

new() does not have a return type specified, but it is defined to be void*. new() must return a void*.
If new() cannot allocate memory, it must not return null, but must throw an exception.
The pointer returned from new() must be to memory aligned to the default alignment. This is 8 on win32 systems.

The *size* parameter is needed in case the allocator is called from a class derived from Foo and is a larger size than Foo.
A null is not returned if storage cannot be allocated. Instead, an exception is thrown. Which exception gets thrown is up to the programmer, in this case, OutOfMemory() is.
When scanning memory for root pointers into the garbage collected heap, the static data segment and the stack are scanned automatically. The C heap is not. Therefore, if Foo or any class derived from Foo using the allocator contains any references to data allocated by the garbage collector, the gc needs to be notified. This is done with the gc.addRange() method.
No initialization of the memory is necessary, as code is automatically inserted after the call to new() to set the class instance members to their defaults and then the constructor (if any) is run.

The critical features of delete() are:

The destructor (if any) has already been called on the argument p, so the data it points to should be assumed to be garbage.
The pointer p may be null.
If the gc was notified with gc.addRange(), a corresponding call to gc.removeRange() must happen in the deallocator.
If there is a delete(), there should be a corresponding new().

If memory is allocated using class specific allocators and deallocators, careful coding practices must be followed to avoid memory leaks and dangling references. In the presence of exceptions, it is particularly important to practice RAII to prevent memory leaks.

Mark/Release

Mark/Release is equivalent to a stack method of allocating and freeing memory. A 'stack' is created in memory. Objects are allocated by simply moving a pointer down the stack. Various points are 'marked', and then whole sections of memory are released simply by resetting the stack pointer back to a marked point.

```
import std.c.stdlib;
import std.outofmemory;

class Foo
{
    static void[] buffer;
    static int bufindex;
    static const int bufsize = 100;

    static this()
    {   void *p;

        p = malloc(bufsize);
        if (!p)
            throw new OutOfMemory;
        gc.addRange(p, p + bufsize);
        buffer = p[0 .. bufsize];
    }
```

```
    static ~this()
    {
        if (buffer.length)
        {
            gc.removeRange(buffer);
            free(buffer);
            buffer = null;
        }
    }

    new(uint sz)
    {   void *p;

        p = &buffer[bufindex];
        bufindex += sz;
        if (bufindex > buffer.length)
            throw new OutOfMemory;
        return p;
    }

    delete(void* p)
    {
        assert(0);
    }

    static int mark()
    {
        return bufindex;
    }

    static void release(int i)
    {
        bufindex = i;
    }
}

void test()
{
    int m = Foo.mark();
    Foo f1 = new Foo;           // allocate
    Foo f2 = new Foo;           // allocate
    ...
    Foo.release(m);             // deallocate f1 and f2
}
```

The allocation of buffer[] itself is added as a region to the gc, so there is no need for a separate call inside Foo.new() to do it.

RAII (Resource Acquisition Is Initialization)

RAII techniques can be useful in avoiding memory leaks when using explicit allocators and deallocators. Adding the auto attribute to such classes can help.

Allocating Class Instances On The Stack

Allocating class instances on the stack is useful for temporary objects that are to be automatically deallocated when the function is exited. No special handling is needed to account for function termination via stack unwinding from an exception. To work, they must not have destructors, since such a destructor would never get called.

To have a class allocated on the stack that has a destructor, this is the same as a declaration with the auto attribute. Although the current implementation does not put such objects on the stack, future ones can.

```
import std.c.stdlib;

class Foo
{
    new(uint sz, void *p)
    {
        return p;
    }

    delete(void* p)
    {
        assert(0);
    }
}

void test()
{
    Foo f = new(std.c.stdlib.alloca(Foo.classinfo.init.length)) Foo;
    ...
}
```

There is no need to check for a failure of alloca() and throw an exception, since by definition alloca() will generate a stack overflow exception if it overflows.
There is no need for a call to gc.addRange() or gc.removeRange() since the gc automatically scans the stack anyway.
The dummy delete() function is to ensure that no attempts are made to delete a stack based object.

Allocating Uninitialized Arrays On The Stack

Arrays are always initialized in D. So, the following declaration:

```
void foo()
{   byte[1024] buffer;

    fillBuffer(buffer);
    ...
}
```

will not be as fast as it might be since the buffer[] contents are always initialized. If careful

profiling of the program shows that this initialization is a speed problem, it can be eliminated using the following idiom:

```
import std.c.stdlib;

void foo()
{   byte[] buffer = (cast(byte*)std.c.stdlib.alloca(1024))[0 .. 1024];

    fillBuffer(buffer);
    ...
}
```

A good D implementation will recognize that alloca() is called with a constant size argument, and so can be replaced with an uninitialized array of the same size allocated on the stack. This will produce execution performance equivalent to using an uninitialized stack array in C.

Uninitialized data on the stack comes with some caveats that need to be carefully evaluated before using:

> The uninitialized data that is on the stack will get scanned by the garbage collector looking for any references to allocated memory. Since the uninitialized data consists of old D stack frames, it is highly likely that some of that garbage will look like references into the gc heap, and the gc memory will not get freed. This problem really does happen, and can be pretty frustrating to track down.
> It's possible for a function to pass out of it a reference to data on that function's stack frame. By then allocating a new stack frame over the old data, and not initializing, the reference to the old data may still appear to be valid. The program will then behave erratically. Initializing all data on the stack frame will greatly increase the probability of forcing that bug into the open in a repeatable manner.
> Uninitialized data can be a source of bugs and trouble, even when used correctly. One design goal of D is to improve reliability and portability by eliminating sources of undefined behavior, and uninitialized data is one huge source of undefined, unportable, erratic and unpredictable behavior. Hence this idiom should only be used after other opportunities for speed optimization are exhausted and if benchmarking shows that it really does speed up the overall execution.

Floating Point

Floating Point Intermediate Values

On many computers, greater precision operations do not take any longer than lesser precision operations, so it makes numerical sense to use the greatest precision available for internal temporaries. The philosophy is not to dumb down the language to the lowest common hardware denominator, but to enable the exploitation of the best capabilities of target hardware.

For floating point operations and expression intermediate values, a greater precision can be used than the type of the expression. Only the minimum precision is set by the types of the operands, not the maximum. **Implementation Note:** On Intel x86 machines, for example, it is expected (but not required) that the intermediate calculations be done to the full 80 bits of precision implemented by the hardware.

It's possible that, due to greater use of temporaries and common subexpressions, optimized code may produce a more accurate answer than unoptimized code.

Algorithms should be written to work based on the minimum precision of the calculation. They should not degrade or fail if the actual precision is greater. Float or double types, as opposed to the extended type, should only be used for:

reducing memory consumption for large arrays
data and function argument compatibility with C

Complex and Imaginary types

In existing languages, there is an astonishing amount of effort expended in trying to jam a complex type onto existing type definition facilities: templates, structs, operator overloading, etc., and it all usually ultimately fails. It fails because the semantics of complex operations can be subtle, and it fails because the compiler doesn't know what the programmer is trying to do, and so cannot optimize the semantic implementation.

This is all done to avoid adding a new type. Adding a new type means that the compiler can make all the semantics of complex work "right". The programmer then can rely on a correct (or at least fixable) implementation of complex.

Coming with the baggage of a complex type is the need for an imaginary type. An imaginary type eliminates some subtle semantic issues, and improves performance by not having to perform extra operations on the implied 0 real part.

Imaginary literals have an i suffix:

```
ireal j = 1.3i;
```

There is no particular complex literal syntax, just add a real and imaginary type:

```
cdouble cd = 3.6 + 4i;
creal c = 4.5 + 2i;
```

Complex numbers have two properties:

```
.re     get real part
.im     get imaginary part as a real
```

For example:

```
cd.re           is 4.5 double
cd.im           is 2 double
c.re            is 4.5 real
```

```
        c.im            is 2 real
```

Rounding Control

IEEE 754 floating point arithmetic includes the ability to set 4 different rounding modes. D adds syntax to access them: [blah, blah, blah] [NOTE: this is perhaps better done with a standard library call]

Exception Flags

IEEE 754 floating point arithmetic can set several flags based on what happened with a computation: [blah, blah, blah]. These flags can be set/reset with the syntax: [blah, blah, blah] [NOTE: this is perhaps better done with a standard library call]

Floating Point Comparisons

In addition to the usual < <= > >= == != comparison operators, D adds more that are specific to floating point. These are !<>= <> <>= !<= !< !>= !> !<> and match the semantics for the NCEG extensions to C.

```
        Floating point comparison operators

        Operator        Relations       Invalid?        Description

                        > < = ?

        <               F T F F         yes     less than
        >               T F F F         yes     greater than
        <=              F T T F         yes     less than or equal to
        >=              T F T F         yes     greater than or equal to
        ==              F F T F         no      equal to
        !=              T T F T         no      unordered, less than, or
greater than
        !<>=            F F F T         no      unordered
        <>              T T F F         yes     less than or greater than
        <>=             T T T F         yes     less than, equal to, or
greater than
        !<=             T F F T         no      unordered or greater than
        !<              T F T T         no      unordered, greater than, or
equal to
        !>=             F T F T         no      unordered or less than
        !>              F T T T         no      unordered, less than, or
equal to
        !<>             F F T T         no      unordered or equal to
```

D x86 Inline Assembler

D, being a systems programming language, provides an inline assembler. The inline assembler is standardized for D implementations across the same CPU family, for example, the Intel Pentium inline assembler for a Win32 D compiler will be syntax compatible with the inline assembler for Linux running on an Intel Pentium.

Differing D implementations, however, are free to innovate upon the memory model, function call/return conventions, argument passing conventions, etc.

This document describes the x86 implementation of the inline assembler.

```
AsmInstruction:
        Identifier : AsmInstruction
        align IntegerExpression
        even
        naked
        db Operands
        ds Operands
        di Operands
        dl Operands
        df Operands
        dd Operands
        de Operands
        Opcode
        Opcode Operands

Operands
        Operand
        Operand , Operands
```

Labels

Assembler instructions can be labeled just like other statements. They can be the target of goto statements. For example:

```
void *pc;
asm
{
    call L1             ;
 L1:                    ;
    pop EBX             ;
    mov pc[EBP],EBX     ;       // pc now points to code at L1
}
```

align *IntegerExpression*

Causes the assembler to emit NOP instructions to align the next assembler instruction on an *IntegerExpression* boundary. *IntegerExpression* must evaluate to an integer that is a power of 2.

Aligning the start of a loop body can sometimes have a dramatic effect on the execution speed.

even

Causes the assembler to emit NOP instructions to align the next assembler instruction on an even boundary.

naked

Causes the compiler to not generate the function prolog and epilog sequences. This means such is the responsibility of inline assembly programmer, and is normally used when the entire function is to be written in assembler.

db, ds, di, dl, df, dd, de

These pseudo ops are for inserting raw data directly into the code. **db** is for bytes, **ds** is for 16 bit words, **di** is for 32 bit words, **dl** is for 64 bit words, **df** is for 32 bit floats, **dd** is for 64 bit doubles, and **de** is for 80 bit extended reals. Each can have multiple operands. If an operand is a string literal, it is as if there were *length* operands, where *length* is the number of characters in the string. One character is used per operand. For example:

```
        asm
        {
            db 5,6,0x83;    // insert bytes 0x05, 0x06, and 0x83 into code
            ds 0x1234;      // insert bytes 0x34, 0x12
            di 0x1234;      // insert bytes 0x34, 0x12, 0x00, 0x00
            dl 0x1234;      // insert bytes 0x34, 0x12, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00
            df 1.234;       // insert float 1.234
            dd 1.234;       // insert double 1.234
            de 1.234;       // insert extended 1.234
            db "abc";       // insert bytes 0x61, 0x62, and 0x63
            ds "abc";       // insert bytes 0x61, 0x00, 0x62, 0x00, 0x63, 0x00
        }
```

Opcodes

A list of supported opcodes is at the end.

The following registers are supported. Register names are always in upper case.

AL, **AH**, **AX**, **EAX**
BL, **BH**, **BX**, **EBX**
CL, **CH**, **CX**, **ECX**
DL, **DH**, **DX**, **EDX**
BP, **EBP**
SP, **ESP**
DI, **EDI**
SI, **ESI**
ES, **CS**, **SS**, **DS**, **GS**, **FS**
CR0, **CR2**, **CR3**, **CR4**
DR0, **DR1**, **DR2**, **DR3**, **DR6**, **DR7**
TR3, **TR4**, **TR5**, **TR6**, **TR7**
ST
ST(0), **ST(1)**, **ST(2)**, **ST(3)**, **ST(4)**, **ST(5)**, **ST(6)**, **ST(7)**
MM0, **MM1**, **MM2**, **MM3**, **MM4**, **MM5**, **MM6**, **MM7**

Special Cases

lock, **rep**, **repe**, **repne**, **repnz**, **repz**
These prefix instructions do not appear in the same statement as the instructions they prefix; they appear in their own statement. For example:

```
asm
{
    rep   ;
    movsb ;
}
```

pause
This opcode is not supported by the assembler, instead use

```
{
    rep  ;
    nop  ;
}
```

which produces the same result.
floating point ops
Use the two operand form of the instruction format;

```
fdiv ST(1);     // wrong
fmul ST;        // wrong
fdiv ST,ST(1);  // right
fmul ST,ST(0);  // right
```

Operands

```
Operand:
    AsmExp

AsmExp:
    AsmLogOrExp
    AsmLogOrExp ? AsmExp : AsmExp

AsmLogOrExp:
    AsmLogAndExp
    AsmLogAndExp || AsmLogAndExp

AsmLogAndExp:
    AsmOrExp
    AsmOrExp && AsmOrExp

AsmOrExp:
    AsmXorExp
    AsmXorExp | AsmXorExp

AsmXorExp:
    AsmAndExp
    AsmAndExp ^ AsmAndExp

AsmAndExp:
    AsmEqualExp
    AsmEqualExp & AsmEqualExp

AsmEqualExp:
    AsmRelExp
    AsmRelExp == AsmRelExp
    AsmRelExp != AsmRelExp

AsmRelExp:
    AsmShiftExp
    AsmShiftExp < AsmShiftExp
    AsmShiftExp <= AsmShiftExp
    AsmShiftExp > AsmShiftExp
    AsmShiftExp >= AsmShiftExp

AsmShiftExp:
    AsmAddExp
    AsmAddExp << AsmAddExp
    AsmAddExp >> AsmAddExp
    AsmAddExp >>> AsmAddExp

AsmAddExp:
    AsmMulExp
    AsmMulExp + AsmMulExp
    AsmMulExp - AsmMulExp

AsmMulExp:
    AsmBrExp
    AsmBrExp * AsmBrExp
    AsmBrExp / AsmBrExp
```

```
        AsmBrExp % AsmBrExp

    AsmBrExp:
        AsmUnaExp
        AsmBrExp [ AsmExp ]

    AsmUnaExp:
        AsmTypePrefix AsmExp
        offset AsmExp
        seg AsmExp
        + AsmUnaExp
        - AsmUnaExp
        ! AsmUnaExp
        ~ AsmUnaExp
        AsmPrimaryExp

    AsmPrimaryExp
        IntegerConstant
        FloatConstant
        __LOCAL_SIZE
        $
        Register
        DotIdentifier

    DotIdentifier
        Identifier
        Identifier . DotIdentifier
```

The operand syntax more or less follows the Intel CPU documentation conventions. In particular, the convention is that for two operand instructions the source is the right operand and the destination is the left operand. The syntax differs from that of Intel's in order to be compatible with the D language tokenizer and to simplify parsing.

Operand Types

```
    AsmTypePrefix:
            near ptr
            far ptr
            byte ptr
            short ptr
            int ptr
            word ptr
            dword ptr
            float ptr
            double ptr
            extended ptr
```

In cases where the operand size is ambiguous, as in:

```
        add     [EAX],3         ;
```

it can be disambiguated by using an *AsmTypePrefix*:

```
add     byte ptr [EAX],3        ;
add     int ptr [EAX],7         ;
```

Struct/Union/Class Member Offsets

To access members of an aggregate, given a pointer to the aggregate is in a register, use the qualified name of the member:

```
struct Foo { int a,b,c; }
int bar(Foo *f)
{
    asm
    {   mov     EBX,f           ;
        mov     EAX,Foo.b[EBX]  ;
    }
}
```

Special Symbols

$

Represents the program counter of the start of the next instruction. So,

```
jmp     $   ;
```

branches to the instruction following the jmp instruction.

__LOCAL_SIZE

This gets replaced by the number of local bytes in the local stack frame. It is most handy when the **naked** is invoked and a custom stack frame is programmed.

Opcodes Supported

aaa	aad	aam	aas	adc
add	addpd	addps	addsd	addss
and	andnpd	andnps	andpd	andps
arpl	bound	bsf	bsr	bswap
bt	btc	btr	bts	call
cbw	cdq	clc	cld	clflush
cli	clts	cmc	cmova	cmovae

cmovb	cmovbe	cmovc	cmove	cmovg
cmovge	cmovl	cmovle	cmovna	cmovnae
cmovnb	cmovnbe	cmovnc	cmovne	cmovng
cmovnge	cmovnl	cmovnle	cmovno	cmovnp
cmovns	cmovnz	cmovo	cmovp	cmovpe
cmovpo	cmovs	cmovz	cmp	cmppd
cmpps	cmps	cmpsb	cmpsd	cmpss
cmpsw	cmpxch8b	cmpxchg	comisd	comiss
cpuid	cvtdq2pd	cvtdq2ps	cvtpd2dq	cvtpd2pi
cvtpd2ps	cvtpi2pd	cvtpi2ps	cvtps2dq	cvtps2pd
cvtps2pi	cvtsd2si	cvtsd2ss	cvtsi2sd	cvtsi2ss
cvtss2sd	cvtss2si	cvttpd2dq	cvttpd2pi	cvttps2dq
cvttps2pi	cvttsd2si	cvttss2si	cwd	cwde
da	daa	das	db	dd
de	dec	df	di	div
divpd	divps	divsd	divss	dl
dq	ds	dt	dw	emms
enter	f2xm1	fabs	fadd	faddp
fbld	fbstp	fchs	fclex	fcmovb
fcmovbe	fcmove	fcmovnb	fcmovnbe	fcmovne
fcmovnu	fcmovu	fcom	fcomi	fcomip
fcomp	fcompp	fcos	fdecstp	fdisi
fdiv	fdivp	fdivr	fdivrp	feni
ffree	fiadd	ficom	ficomp	fidiv
fidivr	fild	fimul	fincstp	finit
fist	fistp	fisub	fisubr	fld
fld1	fldcw	fldenv	fldl2e	fldl2t
fldlg2	fldln2	fldpi	fldz	fmul
fmulp	fnclex	fndisi	fneni	fninit
fnop	fnsave	fnstcw	fnstenv	fnstsw
fpatan	fprem	fprem1	fptan	frndint

frstor	fsave	fscale	fsetpm	fsin
fsincos	fsqrt	fst	fstcw	fstenv
fstp	fstsw	fsub	fsubp	fsubr
fsubrp	ftst	fucom	fucomi	fucomip
fucomp	fucompp	fwait	fxam	fxch
fxrstor	fxsave	fxtract	fyl2x	fyl2xp1
hlt	idiv	imul	in	inc
ins	insb	insd	insw	int
into	invd	invlpg	iret	iretd
ja	jae	jb	jbe	jc
jcxz	je	jecxz	jg	jge
jl	jle	jmp	jna	jnae
jnb	jnbe	jnc	jne	jng
jnge	jnl	jnle	jno	jnp
jns	jnz	jo	jp	jpe
jpo	js	jz	lahf	lar
ldmxcsr	lds	lea	leave	les
lfence	lfs	lgdt	lgs	lidt
lldt	lmsw	lock	lods	lodsb
lodsd	lodsw	loop	loope	loopne
loopnz	loopz	lsl	lss	ltr
maskmovdqu	maskmovq	maxpd	maxps	maxsd
maxss	mfence	minpd	minps	minsd
minss	mov	movapd	movaps	movd
movdq2q	movdqa	movdqu	movhlps	movhpd
movhps	movlhps	movlpd	movlps	movmskpd
movmskps	movntdq	movnti	movntpd	movntps
movntq	movq	movq2dq	movs	movsb
movsd	movss	movsw	movsx	movupd
movups	movzx	mul	mulpd	mulps

mulsd	mulss	neg	nop	not
or	orpd	orps	out	outs
outsb	outsd	outsw	packssdw	packsswb
packuswb	paddb	paddd	paddq	paddsb
paddsw	paddusb	paddusw	paddw	pand
pandn	pavgb	pavgw	pcmpeqb	pcmpeqd
pcmpeqw	pcmpgtb	pcmpgtd	pcmpgtw	pextrw
pinsrw	pmaddwd	pmaxsw	pmaxub	pminsw
pminub	pmovmskb	pmulhuw	pmulhw	pmullw
pmuludq	pop	popa	popad	popf
popfd	por	prefetchnta	prefetcht0	prefetcht1
prefetcht2	psadbw	pshufd	pshufhw	pshuflw
pshufw	pslld	pslldq	psllq	psllw
psrad	psraw	psrld	psrldq	psrlq
psrlw	psubb	psubd	psubq	psubsb
psubsw	psubusb	psubusw	psubw	punpckhbw
punpckhdq	punpckhqdq	punpckhwd	punpcklbw	punpckldq
punpcklqdq	punpcklwd	push	pusha	pushad
pushf	pushfd	pxor	rcl	rcpps
rcpss	rcr	rdmsr	rdpmc	rdtsc
rep	repe	repne	repnz	repz
ret	retf	rol	ror	rsm
rsqrtps	rsqrtss	sahf	sal	sar
sbb	scas	scasb	scasd	scasw
seta	setae	setb	setbe	setc
sete	setg	setge	setl	setle
setna	setnae	setnb	setnbe	setnc
setne	setng	setnge	setnl	setnle
setno	setnp	setns	setnz	seto

setp	setpe	setpo	sets	setz
sfence	sgdt	shl	shld	shr
shrd	shufpd	shufps	sidt	sldt
smsw	sqrtpd	sqrtps	sqrtsd	sqrtss
stc	std	sti	stmxcsr	stos
stosb	stosd	stosw	str	sub
subpd	subps	subsd	subss	sysenter
sysexit	test	ucomisd	ucomiss	ud2
unpckhpd	unpckhps	unpcklpd	unpcklps	verr
verw	wait	wbinvd	wrmsr	xadd
xchg	xlat	xlatb	xor	xorpd
xorps				

AMD Opcodes Supported

pavgusb	pf2id	pfacc	pfadd	pfcmpeq
pfcmpge	pfcmpgt	pfmax	pfmin	pfmul
pfnacc	pfpnacc	pfrcp	pfrcpit1	pfrcpit2
pfrsqit1	pfrsqrt	pfsub	pfsubr	pi2fd
pmulhrw	pswapd			

Interfacing to C

D is designed to fit comfortably with a C compiler for the target system. D makes up for not having its own VM by relying on the target environment's C runtime library. It would be senseless to attempt to port to D or write D wrappers for the vast array of C APIs available. How much easier it is to just call them directly.

This is done by matching the C compiler's data types, layouts, and function call/return sequences.

Calling C Functions

C functions can be called directly from D. There is no need for wrapper functions, argument swizzling, and the C functions do not need to be put into a separate DLL.

The C function must be declared and given a calling convention, most likely the "C" calling convention, for example:

```
extern (C) int strcmp(char *string1, char *string2);
```

and then it can be called within D code in the obvious way:

```
import std.string;
int myDfunction(char[] s)
{
    return strcmp(std.string.toCharz(s), "foo\0");
}
```

There are several things going on here:

D understands how C function names are "mangled" and the correct C function call/return sequence.
C functions cannot be overloaded with another C function with the same name.
There are no __cdecl, __far, __stdcall, __declspec, or other such C type modifiers in D. These are handled by attributes, such as `extern (C)`.
There are no const or volatile type modifiers in D. To declare a C function that uses those type modifiers, just drop those keywords from the declaration.
Strings are not 0 terminated in D. See "Data Type Compatibility" for more information about this.

C code can correspondingly call D functions, if the D functions use an attribute that is compatible with the C compiler, most likely the extern (C):

```
// myfunc() can be called from any C function
extern (C)
{
    void myfunc(int a, int b)
    {
        ...
    }
}
```

Storage Allocation

C code explicitly manages memory with calls to malloc() and free(). D allocates memory using the D garbage collector, so no explicit free's are necessary.

D can still explicitly allocate memory using c.stdlib.malloc() and c.stdlib.free(), these are useful

for connecting to C functions that expect malloc'd buffers, etc.

If pointers to D garbage collector allocated memory are passed to C functions, it's critical to ensure that that memory will not be collected by the garbage collector before the C function is done with it. This is accomplished by:

Making a copy of the data using c.stdlib.malloc() and passing the copy instead.
Leaving a pointer to it on the stack (as a parameter or automatic variable), as the garbage collector will scan the stack.
Leaving a pointer to it in the static data segment, as the garbage collector will scan the static data segment.
Registering the pointer with the garbage collector with the gc.addRoot() or gc.addRange() calls.

An interior pointer to the allocated memory block is sufficient to let the GC know the object is in use; i.e. it is not necessary to maintain a pointer to the beginning of the allocated memory.

The garbage collector does not scan the stacks of threads not created by the D Thread interface. Nor does it scan the data segments of other DLL's, etc.

Data Type Compatibility

D type	C type
void	**void**
bit	no equivalent
byte	**signed char**
ubyte	**unsigned char**
char	**char** (chars are unsigned in D)
wchar	**wchar_t**
short	**short**
ushort	**unsigned short**
int	**int**
uint	**unsigned**
long	**long long**
ulong	**unsigned long long**
float	**float**
double	**double**

extended	**long double**
imaginary	**long double _Imaginary**
complex	**long double _Complex**
*type******	*type* *****
*type***[***dim***]**	*type***[***dim***]**
*type***[]**	no equivalent
*type***[***type***]**	no equivalent
"string\0"	*"string"* or **L***"string"*
class	no equivalent
*type***(*)(***parameters***)**	*type***(*)(***parameters***)**

These equivalents hold for most 32 bit C compilers. The C standard does not pin down the sizes of the types, so some care is needed.

Calling printf()

This mostly means checking that the printf format specifier matches the corresponding D data type. Although printf is designed to handle 0 terminated strings, not D dynamic arrays of chars, it turns out that since D dynamic arrays are a length followed by a pointer to the data, the `%.*s` format works perfectly:

```
void foo(char[] string)
{
    printf("my string is: %.*s\n", string);
}
```

Astute readers will notice that the printf format string literal in the example doesn't end with \0. This is because string literals, when they are not part of an initializer to a larger data structure, have a \0 character helpfully stored after the end of them.

Structs and Unions

D structs and unions are analogous to C's.

C code often adjusts the alignment and packing of struct members with a command line switch or with various implementation specific #pragma's. D supports explicit alignment attributes that correspond to the C compiler's rules. Check what alignment the C code is using, and explicitly set it for the D struct declaration.

D does not support bit fields. If needed, they can be emulated with shift and mask operations.

Interfacing to C++

D does not provide an interface to C++. Since D, however, interfaces directly to C, it can interface directly to C++ code if it is declared as having C linkage.

D class objects are incompatible with C++ class objects.

Portability Guide

It's good software engineering practice to minimize gratuitous portability problems in the code. Techniques to minimize potential portability problems are:

The integral and floating type sizes should be considered as minimums. Algorithms should be designed to continue to work properly if the type size increases.
Floating point computations can be carried out at a higher precision than the size of the floating point variable can hold. Floating point algorithms should continue to work properly if precision is arbitrarily increased.
Avoid depending on the order of side effects in a computation that may get reordered by the compiler. For example:

```
a + b + c
```

can be evaluated as (a + b) + c, a + (b + c), (a + c) + b, (c + b) + a, etc. Parenthesis control operator precedence, parenthesis do *not* control order of evaluation.

In particular, function parameters can be evaluated either left to right or right to left, depending on the particular calling conventions used.

If the operands of an associative operator + or * are floating point values, the expression is not reordered.
Avoid dependence on byte order; i.e. whether the CPU is big-endian or little-endian.
Avoid dependence on the size of a pointer or reference being the same size as a particular integral type.
If size dependencies are inevitable, put an `assert` in the code to verify it:

```
assert(int.size == (int*).size);
```

32 to 64 Bit Portability

64 bit processors and operating systems are coming. With that in mind:

Integral types will remain the same sizes between 32 and 64 bit code.
Pointers and object references will increase in size from 4 bytes to 8 bytes going from 32

to 64 bit code.
Use **size_t** as an alias for an unsigned integral type that can span the address space.
Use **ptrdiff_t** as an alias for a signed integral type that can span the address space.
The **.length**, **.size**, **.sizeof**, and **.alignof** properties will be of type **size_t**.

OS Specific Code

System specific code is handled by isolating the differences into separate modules. At compile time, the correct system specific module is imported.

Minor differences can be handled by constant defined in a system specific import, and then using that constant in an **if** statement.

Embedding D in HTML

The D compiler is designed to be able to extract and compile D code embedded within HTML files. This capability means that D code can be written to be displayed within a browser utilizing the full formatting and display capability of HTML.

For example, it is possible to make all uses of a class name actually be hyperlinks to where the class is defined. There's nothing new to learn for the person browsing the code, he just uses the normal features of an HTML browser. Strings can be displayed in green, comments in red, and keywords in **boldface**, for one possibility. It is even possible to embed pictures in the code, as normal HTML image tags.

Embedding D in HTML makes it possible to put the documentation for code and the code itself all together in one file. It is no longer necessary to relegate documentation in comments, to be extracted later by a tech writer. The code and the documentation for it can be maintained simultaneously, with no duplication of effort.

How it works is straightforward. If the source file to the compiler ends in .htm or .html, the code is assumed to be embedded in HTML. The source is then preprocessed by stripping all text outside of <code> and </code> tags. Then, all other HTML tags are stripped, and embedded character encodings are converted to ASCII. All newlines in the original HTML remain in their corresponding positions in the preprocessed text, so the debug line numbers remain consistent. The resulting text is then fed to the D compiler.

Here's an example of the D program "hello world" embedded in this very HTML file. This file can be compiled and run.

```
import std.c.stdio;

int main()
{
 printf("hello world\n");
 return 0;
```

```
}
```

D Application Binary Interface

A D implementation that conforms to the D ABI (Application Binary Interface) will be able to generate libraries, DLL's, etc., that can interoperate with D binaries built by other implementations.

Most of this specification remains TBD (To Be Defined).

C ABI

The C ABI referred to in this specification means the C Application Binary Interface of the target system. C and D code should be freely linkable together, in particular, D code shall have access to the entire C ABI runtime library.

Basic Types

TBD

Structs

Conforms to the target's C ABI struct layout.

Classes

An object consists of:

```
offset  contents
------  --------
0:      pointer to vtable
4:      monitor
8...    non-static members
```

The vtable consists of:

```
0:      pointer to instance of ClassInfo
4...    pointers to virtual member functions
```

The class definition:

```
class XXXX
{
    ....
```

```
};
```

Generates the following:

An instance of Class called ClassXXXX.
A type called StaticClassXXXX which defines all the static members.
An instance of StaticClassXXXX called StaticXXXX for the static members.

Interfaces

TBD

Arrays

A dynamic array consists of:

```
0:      array dimension
4:      pointer to array data
```

A dynamic array is declared as:

```
type array[];
```

whereas a static array is declared as:

```
type array[dimension];
```

Thus, a static array always has the dimension statically available as part of the type, and so it is implemented like in C. Static array's and Dynamic arrays can be easily converted back and forth to each other.

Associative Arrays

TBD

Reference Types

D has reference types, but they are implicit. For example, classes are always referred to by reference; this means that class instances can never reside on the stack or be passed as function parameters.

When passing a static array to a function, the result, although declared as a static array, will actually be a reference to a static array. For example:

```
int abc[3];
```

Passing abc to functions results in these implicit conversions:

```
        void func(int array[3]);        // actually
        void func(int *p);              // abc[3] is converted to a pointer
to the first element
        void func(int array[]); // abc[3] is converted to a dynamic array
```

Name Mangling

TBD

Function Calling Conventions

TBD

Exception Handling

Windows

Conforms to the Microsoft Windows Structured Exception Handling conventions. TBD

Linux

Uses static address range/handler tables. TBD

Garbage Collection

TBD

Runtime Helper Functions

TBD

Module Initialization and Termination

TBD

Unit Testing

TBD

Phobos

D Runtime Library

Phobos is the standard runtime library that comes with the D language compiler. Also, check out the wiki for Phobos.

Philosophy

Each module in Phobos conforms as much as possible to the following design goals. These are goals rather than requirements because D is not a religion, it's a programming language, and it recognizes that sometimes the goals are contradictory and counterproductive in certain situations, and programmers have jobs that need to get done.

Machine and Operating System Independent Interfaces
: It's pretty well accepted that gratuitous non-portability should be avoided. This should not be construed, however, as meaning that access to unusual features of an operating system should be prevented.

Simple Operations should be Simple
: A common and simple operation, like writing an array of bytes to a file, should be simple to code. I haven't seen a class library yet that simply and efficiently implemented common, basic file I/O operations.

Classes should strive to be independent of one another
: It's discouraging to pull in a megabyte of code bloat by just trying to read a file into an array of bytes. Class independence also means that classes that turn out to be mistakes can be deprecated and redesigned without forcing a rewrite of the rest of the class library.

No pointless wrappers around C runtime library functions or OS API functions
: D provides direct access to C runtime library functions and operating system API functions. Pointless D wrappers around those functions just adds blather, bloat, baggage and bugs.

Class implementations should use DBC
: This will prove that DBC (Contract Programming) is worthwhile. Not only will it aid in debugging the class, but it will help every class user use the class correctly. DBC in the class library will have great leverage.

Use Exceptions for Error Handling
: See Error Handling in D.

Imports

Runtime library modules can be imported with the **import** statement. Each module falls into one of several packages:

std
: These are the core modules.

  std.windows
  : Modules specific to the Windows operating system.

  std.linux
  : Modules specific to the Linux operating system.

  std.c
  : Modules that are simply interfaces to C functions. For example, interfaces to standard C library functions will be in std.c, such as std.c.stdio would be the interface to C's stdio.h.

    std.c.windows
    : Modules corresponding to the C Windows API functions.

    std.c.linux
    : Modules corresponding to the C Linux API functions.

etc
: This is the root of a hierarchy of modules mirroring the std hierarchy. Modules in etc are not standard D modules. They are here because they are experimental, or for some other reason are not quite suitable for std, although they are still useful.

---

std: Core library modules

std.base64
: Encode/decode base64 format.

std.compiler
: Information about the D compiler implementation.

std.conv
: Conversion of strings to integers.

std.ctype
: Simple character classification

std.date
: Date and time functions. Support locales.

std.file
: Basic file operations like read, write, append.

std.format
Formatted conversions of values to strings.
std.gc
Control the garbage collector.
std.intrinsic
Compiler built in intrinsic functions
std.math
Include all the usual math functions like sin, cos, atan, etc.
std.md5
Compute MD5 digests.
std.mmfile
Memory mapped files.
object
The root class of the inheritance hierarchy
std.outbuffer
Assemble data into an array of bytes
std.path
Manipulate file names, path names, etc.
std.process
Create/destroy threads.
std.random
Random number generation.
std.recls
Recursively search file system and (currently Windows only) FTP sites.
std.regexp
The usual regular expression functions.
std.socket
Sockets.
std.socketstream
Stream for a blocking, connected **Socket**.
std.stdint
Integral types for various purposes.
std.stdio
Standard I/O.
std.stream
Stream I/O.
std.string
Basic string operations not covered by array ops.
std.system
Inquire about the CPU, operating system.
std.thread
One per thread. Operations to do on a thread.
std.uri
Encode and decode Uniform Resource Identifiers (URIs).
std.utf
Encode and decode utf character encodings.
std.zip

Read/write zip archives.

std.zlib

Compression / Decompression of data.

std.c: Interface to C functions

std.c.stdio

Interface to C stdio functions like printf().

std.c.windows: Interface to C Windows functions

std.c.windows.windows

Interface to Windows APIs

std.c.linux: Interface to C Linux functions

std.c.linux

Interface to Linux APIs

---

std.base64

Encodes/decodes base64 data.

---

std.compiler

char[] **name**;

Vendor specific string naming the compiler, for example: "Digital Mars D".

enum **Vendor**

Master list of D compiler vendors.

DigitalMars

Digital Mars

Vendor **vendor**;

Which vendor produced this compiler.

uint **version_major**;
uint **version_minor**;

The vendor specific version number, as in *version_major.version_minor*.

uint **D_major**;
uint **D_minor**;
The version of the D Programming Language Specification supported by the compiler.

---

std.conv

std.conv provides basic building blocks for conversions from strings to integral types. They differ from the C functions `atoi()` and `atol()` by not allowing whitespace or overflows.

For conversion to signed types, the grammar recognized is:

```
Integer:
        Sign UnsignedInteger
        UnsignedInteger

Sign:
        +
        -
```

For conversion to unsigned types, the grammar recognized is:

```
UnsignedInteger:
        DecimalDigit
        DecimalDigit UnsignedInteger
```

Any deviation from that grammar causes a **ConvError** exception to be thrown. Any overflows cause a **ConvOverflowError** to be thrown.

byte **toByte**(char[] s)

ubyte **toUbyte**(char[] s)

short **toShort**(char[] s)

ushort **toUshort**(char[] s)

int **toInt**(char[] s)

uint **toUint**(char[] s)

long **toLong**(char[] s)

ulong **toUlong**(char[] s)

std.ctype

int **isalnum**(dchar c)
Returns !=0 if c is a letter or a digit.

int **isalpha**(dchar c)
Returns !=0 if c is an upper or lower case letter.

int **iscntrl**(dchar c)
Returns !=0 if c is a control character.

int **isdigit**(dchar c)
Returns !=0 if c is a digit.

int **isgraph**(dchar c)
Returns !=0 if c is a printing character except for the space character.

int **islower**(dchar c)
Returns !=0 if c is lower case.

int **isprint**(dchar c)
Returns !=0 if c is a printing character or a space.

int **ispunct**(dchar c)
Returns !=0 if c is a punctuation character.

int **isspace**(dchar c)
Returns !=0 if c is a space, tab, vertical tab, form feed, carriage return, or linefeed.

int **isupper**(dchar c)
Returns !=0 if c is an upper case character.

int **isxdigit**(dchar c)
Returns !=0 if c is a hex digit (0..9, a..f, A..F).

int **isascii**(dchar c)
Returns !=0 if c is in the ascii character set.

dchar **tolower**(dchar c)
If c is upper case, return the lower case equivalent, otherwise return c.

dchar **toupper**(dchar c)
If c is lower case, return the upper case equivalent, otherwise return c.

std.date

Dates are represented in several formats. The **date** implementation revolves around a central type, **d_time**, from which other formats are converted to and from.

typedef **d_time**

Is a signed arithmetic type giving the time elapsed since January 1, 1970. Negative values are for dates preceding 1970. The time unit used is *Ticks*. *Ticks* are milliseconds or smaller intervals.

The usual arithmetic operations can be performed on **d_time**, such as adding, subtracting, etc. Elapsed time in *Ticks* can be computed by subtracting a starting **d_time** from an ending **d_time**.

An invalid value for **d_time** is represented by **d_time.init**.

int **TicksPerSecond**

A constant giving the number of *Ticks* per second for this implementation. It will be at least 1000.

char[] **toString**(d_time t)

Converts *t* into a text string of the form: "Www Mmm dd hh:mm:ss GMT+-TZ yyyy", for example, "Tue Apr 02 02:04:57 GMT-0800 1996". If *t* is invalid, "Invalid date" is returned.

char[] **toDateString**(d_time t)

Converts the date portion of *t* into a text string of the form: "Www Mmm dd yyyy", for example, "Tue Apr 02 1996". If *t* is invalid, "Invalid date" is returned.

char[] **toTimeString**(d_time t)

Converts the time portion of *t* into a text string of the form: "hh:mm:ss GMT+-TZ", for example, "02:04:57 GMT-0800". If *t* is invalid, "Invalid date" is returned.

d_time **parse**(char[] s)

Parses *s* as a textual date string, and returns it as a **d_time**. If the string is not a valid date, `d_time.init` is returned.

void **toISO8601YearWeek**(d_time *t*, out int *year*, out int *week*)

Compute year and week [1..53] from *t*. The ISO 8601 week 1 is the first week of the year that includes January 4. Monday is the first day of the week.

d_time **getUTCtime**()

Get current UTC time.

d_time **UTCtoLocalTime**(d_time t)
Convert from UTC time to local time.

d_time **LocalTimetoUTC**(d_time t)
Convert from local time to UTC time.

typedef **DosFileTime**
Type representing the DOS file date/time format.

d_time **toDtime**(DosFileTime time)
Convert from DOS file date/time to d_time.

DosFileTime **toDosFileTime**(d_time t)
Convert from d_time to DOS file date/time.

---

std.file

class **FileException**
Exception thrown if file I/O errors.

void[] **read**(char[] name)
Read file *name[]*, return array of bytes read.

void **write**(char[] name, void[] buffer)
Write *buffer[]* to file *name[]*.

void **append**(char[] name, void[] buffer)
Append *buffer[]* to file *name[]*.

void **rename**(char[] from, char[] to)
Rename file *from[]* to *to[]*.

void **remove**(char[] name)
Delete file *name[]*.

uint **getSize**(char[] name)
Get size of file *name[]*.

uint **getAttributes**(char[] name)
Get file *name[]* attributes.

int **exists**(char[] name)
Does *name[]* exist (file or directory)?

int **isfile**(char[] name)
Is *name[]* a file? Error if *name[]* doesn't exist.

int **isdir**(char[] name)
Is *name[]* a directory? Error if *name[]* doesn't exist.

void **chdir**(char[] name)
Change directory.

void **mkdir**(char[] name)
Make directory.

void **rmdir**(char[] name)
Remove directory.

char[] **getcwd**()
Get current directory.

char[][] **listdir**(char[] pathname)
Return contents of directory.

---

std.gc

The garbage collector normally works behind the scenes without needing any specific interaction. These functions are for advanced applications that benefit from tuning the operation of the collector.

class **OutOfMemory**
Thrown if garbage collector runs out of memory.

void **addRoot**(void* p)
Add p to list of roots. Roots are references to memory allocated by the collector that are maintained in memory outside the collector pool. The garbage collector will by default look for roots in the stacks of each thread, the registers, and the default static data segment. If roots are held elsewhere, use **addRoot**() or **addRange**() to tell the collector not to free the memory it points to.

void **removeRoot**(void* p)
Remove p from list of roots.

void **addRange**(void* pbot, void* ptop)
Add range to scan for roots.

void **removeRange**(void* pbot)

Remove range.

void **fullCollect**()
Run a full garbage collection cycle. The collector normally runs synchronously with a storage allocation request (i.e. it never happens when in code that does not allocate memory). In some circumstances, for example when a particular task is finished, it is convenient to explicitly run the collector and free up all memory used by that task. It can also be helpful to run a collection before starting a new task that would be annoying if it ran a collection in the middle of that task. Explicitly running a collection can also be done in a separate very low priority thread, so that if the program is idly waiting for input, memory can be cleaned up.

void **genCollect**()
Run a generational garbage collection cycle. Takes less time than a **fullCollect**(), but isn't as effective.

void **minimize**()
Minimize physical memory usage.

void **disable**()
Temporarilly disable garbage collection cycle. This is used for brief time critical sections of code, so the amount of time it will take is predictable. If the collector runs out of memory while it is disabled, it will throw an **OutOfMemory** exception. The **disable**() function calls can be nested, but must be matched with corresponding **enable**() calls.

void **enable**()
Reenable garbage collection cycle after being disabled with **disable**(). It is an error to call more **enable()**s than **disable()**s.

---

std.intrinsic

Intrinsic functions are functions built in to the compiler, usually to take advantage of specific CPU features that are inefficient to handle via external functions. The compiler's optimizer and code generator are fully integrated in with intrinsic functions, bringing to bear their full power on them. This can result in some surprising speedups.

int **bsf**(uint v)
Scans the bits in *v* starting with bit 0, looking for the first set bit.
int **bsr**(uint v)
Scans the bits in *v* from the most significant bit to the least significant bit, looking for the first set bit.

Both return the bit number of the first set bit. The return value is undefined if *v* is

zero.

Example

```
import std.intrinsic;

int main()
{
    uint v;
    int x;

    v = 0x21;
    x = bsf(v);
    printf("bsf(x%x) = %d\n", v, x);
    x = bsr(v);
    printf("bsr(x%x) = %d\n", v, x);
    return 0;
}
```

Output

```
bsf(x21) = 0
bsr(x21) = 5
```

int **bt**(uint* p, uint index)
Tests the bit.
int **btc**(uint* p, uint index)
Tests and complements the bit.
int **btr**(uint* p, uint index)
Tests and resets (sets to 0) the bit.
int **bts**(uint* p, uint index)
Tests and sets the bit.

p is a non-NULL pointer to an array of uints. *index* is a bit number, starting with bit 0 of p[0], and progressing. It addresses bits like the expression:

```
p[index / (uint.size*8)] & (1 << (index & ((uint.size*8) - 1)))
```

All return a non-zero value if the bit was set, and a zero if it was clear.

Example

```
import std.intrinsic;
```

```
int main()
{
    uint array[2];

    array[0] = 2;
    array[1] = 0x100;

    printf("btc(array, 35) = %d\n", btc(array, 35));
    printf("array = [0]:x%x, [1]:x%x\n", array[0],
array[1]);

    printf("btc(array, 35) = %d\n", btc(array, 35));
    printf("array = [0]:x%x, [1]:x%x\n", array[0],
array[1]);

    printf("bts(array, 35) = %d\n", bts(array, 35));
    printf("array = [0]:x%x, [1]:x%x\n", array[0],
array[1]);

    printf("btr(array, 35) = %d\n", btr(array, 35));
    printf("array = [0]:x%x, [1]:x%x\n", array[0],
array[1]);

    printf("bt(array, 1) = %d\n", bt(array, 1));
    printf("array = [0]:x%x, [1]:x%x\n", array[0],
array[1]);

    return 0;
}
```

Output

```
btc(array, 35) = 0
array = [0]:x2, [1]:x108
btc(array, 35) = -1
array = [0]:x2, [1]:x100
bts(array, 35) = 0
array = [0]:x2, [1]:x108
btr(array, 35) = -1
array = [0]:x2, [1]:x100
bt(array, 1) = -1
array = [0]:x2, [1]:x100
```

uint **bswap**(uint x)
Swaps bytes in a 4 byte uint end-to-end, i.e. byte 0 becomes byte 3, byte 1 becomes byte 2, byte 2 becomes byte 1, byte 3 becomes byte 0.

ubyte **inp**(uint port_address)
ushort **inpw**(uint port_address)
uint **inpl**(uint port_address)
Reads I/O port at *port_address*.

ubyte **outp**(uint port_address, ubyte value)

ushort **outpw**(uint port_address, ushort value)
uint **outpl**(uint port_address, uint value)
Writes and returns *value* to I/O port at *port_address*.

real **cos**(real)
real **fabs**(real)
real **rint**(real)
long **rndtol**(real)
real **sin**(real)
real **sqrt**(real)
Intrinsic verions of the math functions of the same name.

---

std.math

const real **PI**
const real **LOG2**
const real **LN2**
const real **LOG2T**
const real **LOG2E**
const real **E**
const real **LOG10E**
const real **LN10**
const real **PI_2**
const real **PI_4**
const real **M_1_PI**
const real **M_2_PI**
const real **M_2_SQRTPI**
const real **SQRT2**
const real **SQRT1_2**
Math constants.

real **acos**(real)
real **asin**(real)
real **atan**(real)
real **atan2**(real, real)

real **cos**(real *x*)
Compute cosine of *x*. *x* is in radians.
Special values:

x	**return value**	**invalid?**
±INFINITY	NAN	yes

real **sin**(real *x*)

Compute sine of *x*. *x* is in radians.
Special values:

x	return value	invalid?
±0.0	±0.0	no
±INFINITY	NAN	yes

real **tan**(real *x*)
Compute tangent of *x*. *x* is in radians.
Special values:

x	return value	invalid?
±0.0	±0.0	no
±INFINITY	NAN	yes

real **cosh**(real)
real **sinh**(real)
real **tanh**(real)
real **exp**(real)

real **frexp**(real *value*, out int *exp*)
Calculate and return *x* and *exp* such that:
value=*x**2^{exp}
.5 <= |*x*| < 1.0
x has same sign as *value*.
Special values:

value	*x*	*exp*
+-0.0	+-0.0	0
+INFINITY	+INFINITY	int.max
-INFINITY	-INFINITY	int.min
+-NAN	+-NAN	int.min

real **ldexp**(real *n*, int *exp*)
Compute `n * 2^exp`

real **log**(real *x*)
Calculate the natural logarithm of *x*.
Special values:

x	return value	divide by 0?	invalid?
±0.0	-INFINITY	yes	no
< 0.0	NAN	no	yes
+INFINITY	+INFINITY	no	no

real **log10**(real *x*)

Calculate the base-10 logarithm of *x*.

Special values:

x	return value	divide by 0?	invalid?
±0.0	-INFINITY	yes	no
< 0.0	NAN	no	yes
+INFINITY	+INFINITY	no	no

real **modf**(real, real*)

real **pow**(real, real)

real **sqrt**(real *x*)
creal **sqrt**(creal *x*)

Compute square root of *x*.

Special values:

x	return value	invalid?
-0.0	-0.0	no
<0.0	NAN	yes
+INFINITY	+INFINITY	no

real **ceil**(real)
real **floor**(real)

real **log1p**(real *x*)

Calculates the natural logarithm of 1 + *x*. For very small *x*, log1p(*x*) will be more accurate than log(1 + *x*).

Special values:

x	log1p(*x*)	divide by 0?	invalid?
±0.0	±0.0	no	no
-1.0	-INFINITY	yes	no
<-1.0	NAN	no	yes
+INFINITY	-INFINITY	no	no

real **expm1**(real *x*)

Calculates the value of the natural logarithm base (e) raised to the power of *x*, minus 1. For very small *x*, expm1(*x*) is more accurate than exp(*x*)-1.

Special values:

x	e^x-1
±0.0	±0.0
+INFINITY	+INFINITY

-INFINITY	-1.0

real **atof**(char*)
Math functions.

real **hypot**(real *x*, real *y*)
Calculates the length of the hypotenuse of a right-angled triangle with sides of length *x* and *y*. The hypotenuse is the value of the square root of the sums of the squares of *x* and *y*:

```
sqrt(x2 + y2)
```

Note that hypot(*x*,*y*), hypot(*y*,*x*) and hypot(*x*,-*y*) are equivalent.
Special values:

x	***y***	**return value**	**invalid?**
x	+-0.0	fabs(*x*)	no
+-INFINITY	*y*	+INFINITY	no
+-INFINITY	NAN	+INFINITY	no

int **isnan**(real e)
Is number a nan?

int **isfinite**(real e)
Is number finite?

int **isnormal**(float f)
int **isnormal**(double d)
int **isnormal**(real e)
Is number normalized?

int **issubnormal**(float f)
int **issubnormal**(double d)
int **issubnormal**(real e)
Is number subnormal? (Also called "denormal".) Subnormals have a 0 exponent and a 0 most significant mantissa bit.

int **isinf**(real e)
Is number infinity?

int **signbit**(real e)
Get sign bit.

real **copysign**(real to, real from)
Copy sign.

std.md5

Computes MD5 digests of arbitrary data. MD5 digests are 16 byte quantities that are like a checksum or crc, but are more robust.

There are two ways to do this. The first does it all in one function call to **sum**(). The second is for when the data is buffered.

The routines and algorithms are derived from the RSA Data Security, Inc. MD5 Message-Digest Algorithm.

void **sum**(ubyte[16] *digest*, void[] *data*)
Compute MD5 digest from data.

void **printDigest**(ubyte[16] *digest*)
Print MD5 *digest* to standard output.

struct **MD5_CTX**
Use when data to be digested is buffered.

void **start**()
Begins an MD5 message-digest operation.

void **update**(void[] *input*)
Continues an MD5 message-digest operation, processing another message block *input*, and updating the context.

void **finish**(ubyte[16] *digest*)
Ends an MD5 message-digest operation and writes the result to *digest*.

Example

```
// This code is derived from the
// RSA Data Security, Inc. MD5 Message-Digest Algorithm.

import std.md5;
import std.string;
import std.c.stdio;

int main(char[][] args)
{
    for (int i = 1; i < args.length; i++)
         MDFile(args[i]);
    return 0;
}

/* Digests a file and prints the result. */
void MDFile(char[] filename)
{
    FILE* file;
```

```
    MD5_CTX context;
    int len;
    ubyte [4 * 1024] buffer;
    ubyte digest[16];

    if ((file = fopen(std.string.toStringz(filename), "rb")) == null)
        printf("%.*s can't be opened\n", filename);
    else
    {
        context.start();
        while ((len = fread(buffer, 1, buffer.size, file)) != 0)
            context.update(buffer[0 .. len]);
        context.finish(digest);
        fclose(file);

        printf("MD5 (%.*s) = ", filename);
        printDigest(digest);
        printf("\n");
    }
}
```

---

std.mmfile

Read and write memory mapped files.

auto class **MmFile**

MmFile objects control the memory mapped file resource. The class is **auto** so it is automatically released when the handle for it goes out of scope. Any errors detected by the MmFile objects will throw an instance of **std.file.FileException**.

enum **Mode**

The mode the memory mapped file is opened with:

Read
: read existing file

ReadWriteNew
: delete existing file, write new file

ReadWrite
: read/write existing file, create if not existing

ReadCopyOnWrite
: read/write existing file, copy on write

this(char[] *filename*);

Open memory mapped file *filename* for reading. File is closed when the object instance is deleted or goes out of scope.

this(char[] *filename*, Mode *mode*, size_t *size*, void* *address*);

Open memory mapped file *filename* in *mode*. File is closed when the object

instance is deleted or goes out of scope.
filename gives the name of the file. If **null**, an anonymous file mapping is created.
mode gives the access mode defined above.
size gives the size of the file. If 0, it is taken to be the size of the existing file.
address gives the preferred address to map the file to, allthough the system is not required to honor it. If **null**, the system selects the most convenient address.

~this()
Flushes pending output and closes the memory mapped file.

void **flush**()
Flushes pending output.

size_t **length**()
Gives size in bytes of the memory mapped file.

Operator overloads

void[] **opSlice**()
Returns entire file contents as an array.

void[] **opSlice**(size_t *i1*, size_t *i2*)
Returns slice of file contents as an array.

ubyte **opIndex**(size_t *i*)
Returns byte at index *i* in file.

ubyte **opIndex**(size_t *i*, ubyte *value*)
Returns sets byte at index *i* in file to *value*.

Notes

Won't work with files larger than the address space.

---

object

This module is implicitly imported.

class **Object**
All class objects in D inherit from **Object**.

static int **printf**(char* format, ...);
C printf function.

char[] **toString**()
Convert Object to a human readable string.

uint **toHash**()
Compute hash function for Object.

int **opCmp**(Object obj)
Compare with another Object *obj*. Returns:
<0 for (this < *obj*)
=0 for (this == *obj*)
>0 for (this > *obj*)

class **ClassInfo**
Runtime type information about a class.

class **Exception**
All exceptions should be derived from class **Exception**.

---

std.outbuffer

class **OutBuffer**
OutBuffer provides a way to build up an array of bytes out of raw data. It is useful for things like preparing an array of bytes to write out to a file. **OutBuffer**'s byte order is the format native to the computer. To control the byte order (endianness), use a class derived from **OutBuffer**. To convert an array of bytes back into raw data, use **InBuffer**.

void **reserve**(uint nbytes)
Preallocate *nbytes* more to the size of the internal buffer. This is a speed optimization, a good guess at the maximum size of the resulting buffer will improve performance by eliminating reallocations and copying.

void **write**(ubyte[] bytes)
void **write**(ubyte b)
void **write**(byte b)
void **write**(char c)
void **write**(ushort w)
void **write**(short s)
void **write**(wchar c)
void **write**(uint w)
void **write**(int i)
void **write**(ulong l)
void **write**(long l)

void **write**(float f)
void **write**(double f)
void **write**(real f)
void **write**(char[] s)
void **write**(OutBuffer buf)
Append data to the internal buffer.

void **fill0**(uint nbytes)
Append *nbytes* of 0 to the internal buffer.

void **alignSize**(uint alignsize)
0-fill to align on an *alignsize* boundary. *alignsize* must be a power of 2.

void **align2**()
Optimize common special case **alignSize**(2)

void **align4**()
Optimize common special case **alignSize**(4)

ubyte[] **toBytes**()
Convert internal buffer to array of bytes.

char[] **toString**()
Convert internal buffer to array of chars.

void **vprintf**(char[] format, va_list args)
Append output of vprintf() to internal buffer.

void **printf**(char[] format, ...)
Append output of printf() to internal buffer.

void **spread**(uint index, uint nbytes)
At offset *index* into buffer, create *nbytes* of space by shifting upwards all data past *index*.

---

std.path

const char[] **sep**;
Character used to separate directory names in a path.

const char[] **altsep**;
Alternate version of sep[], used in Windows.

const char[] **pathsep**;
Path separator string.

const char[] **linesep**;
String used to separate lines.

const char[] **curdir**;
String representing the current directory.

const char[] **pardir**;
String representing the parent directory.

char[] **getExt**(char[] fullname)
Get extension. For example, "d:\path\foo.bat" returns "bat".

char[] **getBaseName**(char[] fullname)
Get base name. For example, "d:\path\foo.bat" returns "foo.bat".

char[] **getDirName**(char[] fullname)
Get directory name. For example, "d:\path\foo.bat" returns "d:\path".

char[] **getDrive**(char[] fullname)
Get drive. For example, "d:\path\foo.bat" returns "d:". Returns null string on systems without the concept of a drive.

char[] **defaultExt**(char[] fullname, char[] ext)
Put a default extension on fullname if it doesn't already have an extension.

char[] **addExt**(char[] fullname, char[] ext)
Add file extension or replace existing extension.

int **isabs**(char[] path)
Determine if absolute path name.

char[] **join**(char[] p1, char[] p2)
Join two path components.

int **fncharmatch**(dchar c1, dchar c2)
Match file name characters. Case sensitivity depends on the operating system.

int **fnmatch**(char[] name, char[] pattern)
Match filename strings with pattern[], using the following wildcards:
***** match 0 or more characters
? match any character
[*chars***]** match any character that appears between the []
[!*chars***]** match any character that does not appear between the [!]

Matching is case sensitive on a file system that is case sensitive.
Returns:
!=0 match
0 no match

---

std.process

int **system**(char[] command)
Execute *command* in a command shell. Returns exit status of *command*.

int **execv**(char[] *program*, char[][] *arguments*)
int **execve**(char[] *program*, char[][] *arguments*, char[][] *environment*)
int **execvp**(char[] *program*, char[][] *arguments*)
int **execvpe**(char[] *program*, char[][] *arguments*, char[][] *environment*)
Execute *program*, passing it the *arguments* and the *environment*, returning the exit status. The 'p' versions of exec search the PATH environment variable setting for *program*.

---

std.random

void **rand_seed**(uint seed, uint index)
The random number generator is seeded at program startup with a random value. This ensures that each program generates a different sequence of random numbers. To generate a repeatable sequence, use **rand_seed**() to start the sequence. *seed* and *index* start it, and each successive value increments *index*. This means that the *n*th random number of the sequence can be directly generated by passing *index* + *n* to **rand_seed**().

uint **rand**()
Get next random number in sequence.

---

std.regexp

RegExp is a D class to handle regular expressions. Regular expressions are a powerful method of string pattern matching. The RegExp class is the core foundation for adding powerful string pattern matching capabilities to programs like grep, text editors, awk, sed, etc. The regular expression language used is the same as that commonly used, however, some of the very advanced forms may behave slightly differently.

The **RegExp** class has these methods:

this(char[] pattern, char[] attributes)
Create a new **RegExp** object. Compile *pattern[]* with *attributes[]* into an internal form for fast execution. Throws a **RegExpError** if there are any compilation errors.

char[][] **split**(char[] string)
Split *string[]* into an array of strings, using the regular expression as the separator. Returns array of slices in *string[]*.

int **search**(char[] string)
Search *string[]* for match with regular expression.

Returns	Description
>=0	index of match
-1	no match

char[][] **match**(char[] string)
Search *string[]* for match.

Attribute	Returns
global	same as call to **exec**(*string*)
not global	array of all matches

char[][] **exec**(char[] string)
Search *string[]* for next match. Returns array of slices into *string[]* representing matches.

int **test**(char[] string)
Search *string[]* for next match.

Returns	Description
0	no match
!=0	match

char[] **replace**(char[] string, char[] format)
Find regular expression matches in *string[]*. Replace those matches with a new string composed of *format[]* merged with the result of the matches.

Attribute	Action
global	replace all matches
not global	replace first match

Returns the new string.

char[] **replace**(char[] format)

After a match is found with **test**(), this function will take the match results and, using the *format[]* string, generate and return a new string. The format commands are:

Format	Description
$$	insert $
$&	insert the matched substring
$`	insert the string that precedes the match
$'	insert the string that following the match
$*n*	replace with the *n*th parenthesized match, *n* is 1..9
$*nn*	replace with the *nn*th parenthesized match, *nn* is 01..99
$	insert $

char[] **replaceOld**(char[] format)

Like **replace**(char[] format), but uses old style formatting:

Format	Description
&	replace with the match
n	replace with the *n*th parenthesized match, *n* is 1..9
c	replace with char *c*.

---

std.socket

enum **AddressFamily**

The communication domain used to resolve an address:

UNIX
: local communication

INET
: internet protocol version 4

IPX
: novell IPX

APPLETALK
: appletalk

enum **SocketType**

Communication semantics:

STREAM
sequenced, reliable, two-way communication-based byte streams
DGRAM
connectionless, unreliable datagrams with a fixed maximum length; data may be lost or arrive out of order
RAW
raw protocol access
RDM
reliably-delivered message datagrams
SEQPACKET
sequenced, reliable, two-way connection-based datagrams with a fixed maximum length

enum **ProtocolType**
Protocol:

IP
internet protocol
ICMP
internet control message protocol
IGMP
internet group management protocol
GGP
gateway to gateway protocol
TCP
transmission control protocol
PUP
PARC universal packet protocol
UDP
user datagram protocol
IDP
Xerox NS protocol

---

class **AddressException** : **Exception**
Base exception thrown from an **Address**.

abstract class **Address**
Address is an abstract class for representing a network addresses.

AddressFamily **addressFamily**()
Family of this address.

char[] **toString**()
Human readable string representing this address.

class **InternetAddress** : **Address**

InternetAddress is a class that represents an IPv4 (internet protocol version 4) address and port.

const uint **ADDR_ANY**
Any IPv4 address number.
const uint **ADDR_NONE**
An invalid IPv4 address number.
const ushort **PORT_ANY**
Any IPv4 port number.

this(uint addr, ushort port)
this(ushort port)
Construct a new Address. addr may be **ADDR_ANY** (default) and port may be **PORT_ANY**, and the actual numbers may not be known until a connection is made.
this(char[] addr, ushort port)
addr may be an IPv4 address string in the dotted-decimal form *a.b.c.d*, or a host name that will be resolved using an InternetHost object. port may be **PORT_ANY** as stated above.

AddressFamily **addressFamily**()
Overridden to return AddressFamily.INET.
ushort **port**()
Returns the IPv4 port number.
uint **addr**()
Returns the IPv4 address number.

char[] **toAddrString**()
Human readable string representing the IPv4 address in dotted-decimal form.
char[] **toPortString**()
Human readable string representing the IPv4 port.
char[] **toString**()
Human readable string representing the IPv4 address and port in the form *a.b.c.d:e*.

static uint **parse**(char[] addr)
Parse an IPv4 address string in the dotted-decimal form *a.b.c.d* and return the number. If the string is not a legitimate IPv4 address, **ADDR_NONE** is returned.

---

class **HostException** : **Exception**

Base exception thrown from an **InternetHost**.

class **InternetHost**

InternetHost is a class for resolving IPv4 addresses.

char[] **name**
char[][] **aliases**
uint[] **addrList**
These members are populated when one of the following functions are called without failure:

bit **getHostByName**(char[] name)
Resolve host name. Returns false if unable to resolve.
bit **getHostByAddr**(uint addr)
Resolve IPv4 address number. Returns false if unable to resolve.
bit **getHostByAddr**(char[] addr)
Same as previous, but addr is an IPv4 address string in the dotted-decimal form *a.b.c.d*. Returns false if unable to resolve.

---

enum **SocketShutdown**
How a socket is shutdown:

RECEIVE
socket receives are disallowed
SEND
socket sends are disallowed
BOTH
both **RECEIVE** and **SEND**

enum **SocketFlags**
Flags may be OR'ed together:

NONE
no flags specified
OOB
out-of-band stream data
PEEK
peek at incoming data without removing it from the queue
DONTROUTE
data should not be subject to routing; this flag may be ignored.

enum **SocketOptionLevel**
The level at which a socket option is defined:

SOCKET
socket level
IP
internet protocol level

TCP
transmission control protocol level
UDP
user datagram protocol level

enum **SocketOption**
Specifies a socket option:

DEBUG
record debugging information
BROADCAST
allow transmission of broadcast messages
REUSEADDR
allow local reuse of address
LINGER
linger on close if unsent data is present
OOBINLINE
receive out-of-band data in band
SNDBUF
send buffer size
RCVBUF
receive buffer size
KEEPALIVE
keep connection alive
DONTROUTE
do not route
TCP_NODELAY
disable the Nagle algorithm for send coalescing

struct **linger**
Linger information for use with **SocketOption.LINGER**.

ushort **on**
Nonzero for on.
ushort **time**
Linger time.

struct **timeval**
Duration timeout value.

int **seconds**
Number of seconds.
int **microseconds**
Number of additional microseconds.

class **SocketException** : **Exception**

Base exception thrown from a **Socket**.

class **Socket**

Socket is a class that creates a network communication endpoint using the Berkeley sockets interface.

this(AddressFamily af, SocketType type, ProtocolType protocol)
this(AddressFamily af, SocketType type)
Create a blocking socket. If a single protocol type exists to support this socket type within the address family, the **ProtocolType** may be omitted.

bit **blocking**
Property to get or set whether the socket is blocking or nonblocking.

bit **isAlive**
Property that indicates if this is a valid, alive socket.

AddressFamily **addressFamily**()
Get the socket's address family.

void **bind**(Address addr)
Associate a local address with this socket.

void **connect**(Address to)
Establish a connection. If the socket is blocking, **connect** waits for the connection to be made. If the socket is nonblocking, **connect** returns immediately and the connection attempt is still in progress.

void **listen**(int backlog)
Listen for an incoming connection. **bind** must be called before you can **listen**. The backlog is a request of how many pending incoming connections are queued until **accept**'ed.

Socket **accept**()
Accept an incoming connection. If the socket is blocking, **accept** waits for a connection request. Throws **SocketAcceptException** if unable to accept.

void **shutdown**(SocketShutdown how)
Disables sends and/or receives.

void **close**()
Immediately drop any connections and release socket resources. Calling **shutdown** before **close** is recommended for connection-oriented sockets. The **Socket** object is no longer usable after **close**.

Address **remoteAddress**()

Remote endpoint **Address**.

Address **localAddress**()
Local endpoint **Address**.

const int **ERROR**
Send or receive error code.

int **send**(void[] buf, SocketFlags flags)
int **send**(void[] buf)
Send data on the connection. Returns the number of bytes actually sent, or **ERROR** on failure. If the socket is blocking and there is no buffer space left, **send** waits.

int **sendTo**(void[] buf, SocketFlags flags, Address to)
int **sendTo**(void[] buf, Address to)
int **sendTo**(void[] buf, SocketFlags flags)
int **sendTo**(void[] buf)
Send data to a specific destination **Address**. If the destination address is not specified, a connection must have been made and that address is used. If the socket is blocking and there is no buffer space left, **sendTo** waits.

int **receive**(void[] buf, SocketFlags flags)
int **receive**(void[] buf)
Receive data on the connection. Returns the number of bytes actually received, 0 if the remote side has closed the connection, or **ERROR** on failure. If the socket is blocking, **receive** waits until there is data to be received.

int **receiveFrom**(void[] buf, SocketFlags flags, out Address from)
int **receiveFrom**(void[] buf, out Address from)
int **receiveFrom**(void[] buf, SocketFlags flags)
int **receiveFrom**(void[] buf)
Receive data and get the remote endpoint **Address**. Returns the number of bytes actually received, 0 if the remote side has closed the connection, or **ERROR** on failure. If the socket is blocking, **receiveFrom** waits until there is data to be received.

int **getOption**(SocketOptionLevel level, SocketOption option, void[] result)
Get a socket option. Returns the number of bytes written to result.
int **getOption**(SocketOptionLevel level, SocketOption option, out int result)
Same as previous, but for the common case of integer and boolean options.

void **setOption**(SocketOptionLevel level, SocketOption option, void[] value)
Set a socket option.
void **setOption**(SocketOptionLevel level, SocketOption option, int value)
Same as previous, but for the common case of integer and boolean options.

static int **select**(SocketSet checkRead, SocketSet checkWrite, SocketSet checkError, timeval* tv)
static int **select**(SocketSet checkRead, SocketSet checkWrite, SocketSet checkError, int microseconds)
static int **select**(SocketSet checkRead, SocketSet checkWrite, SocketSet checkError)
Wait for a socket to change status. A wait timeout **timeval** or int microseconds may be specified; if a timeout is not specified or the timeval is null, the maximum timeout is used. The timeval timeout has an unspecified value when **select** returns. Returns the number of sockets with status changes, 0 on timeout, or -1 on interruption. If the return value is greter than 0, the **SocketSet**s are updated to only contain the sockets having status changes. For a **connect**ing socket, a write status change means the connection is established and it's able to **send**. For a **listen**ing socket, a read status change means there is an incoming connection request and it's able to **accept**.

class **TcpSocket** : **Socket**
TcpSocket is a shortcut class for a TCP **Socket**.

this()
Constructs a blocking TCP **Socket**.
this(InternetAddress connectTo)
Constructs a blocking TCP **Socket** and connects to an **InternetAddress**.

class **UdpSocket** : **Socket**
UdpSocket is a shortcut class for a UDP **Socket**.

this()
Constructs a blocking UDP **Socket**.

---

class **SocketSet**
A collection of sockets for use with **Socket.select**.

this(uint max)
Set the maximum amount of sockets that may be added.
this()
Uses the default maximum for the system.

uint **max**
Property to get the maximum amount of sockets that may be added to this **SocketSet**.

void **add**(Socket s)
Add a **Socket** to the collection. Adding more than the maximum has dangerous side affects.

void **remove**(Socket s)
Remove this **Socket** from the collection.

int **isSet**(Socket s)
Returns nonzero if this **Socket** is in the collection.

void **reset**()
Reset the **SocketSet** so that there are 0 **Socket**s in the collection.

Notes

For Win32 systems, link with `ws2_32.lib`.

Example

See `/dmd/samples/d/listener.d`.

---

std.socketstream

class **SocketStream** : **std.stream.Stream**
SocketStream is a stream for a blocking, connected **Socket**.

this(Socket sock, std.stream.FileMode mode)
Constructs a **SocketStream** with the specified **Socket** and **FileMode** flags.
this(Socket sock)
Uses mode **FileMode.In | FileMode.Out**.

Socket **socket**
Property to get the **Socket** that is being streamed.

uint **readBlock**(void* buffer, uint size)
Attempts to read the entire block, waiting if necessary.

char[] **readLine**()
wchar[] **readLineW**()
Read a line. Safely does not use **ungetc**/**ungetcw**.

uint **writeBlock**(void* buffer, uint size)
Attempts to write the entire block, waiting if necessary.

bit **eof**()
Returns true if a remote disconnection has been detected.

char[] **toString**()
Does not return the entire stream because that would require the remote connection to be closed.

void **close**()
Close the **Socket**.

Notes

For Win32 systems, link with `ws2_32.lib`.

Example

See `/dmd/samples/d/htmlget.d`.

---

std.stdint

D constrains integral types to specific sizes. But efficiency of different sizes varies from machine to machine, pointer sizes vary, and the maximum integer size varies. **stdint** offers a portable way of trading off size vs efficiency, in a manner compatible with the `stdint.h` definitions in C.

The exact aliases are types of exactly the specified number of bits. The at least aliases are at least the specified number of bits large, and can be larger. The fast aliases are the fastest integral type supported by the processor that is at least as wide as the specified number of bits.

The aliases are:

Exact Alias	Description	At Least Alias	Description	Fast Alias	Description
int8_t	exactly 8 bits signed	int_least8_t	at least 8 bits signed	int_fast8_t	fast 8 bits sign
uint8_t	exactly 8 bits unsigned	uint_least8_t	at least 8 bits unsigned	uint_fast8_t	fast 8 bits unsi
int16_t	exactly 16 bits signed	int_least16_t	at least 16 bits signed	int_fast16_t	fast 16 bits sig
uint16_t	exactly 16 bits unsigned	uint_least16_t	at least 16 bits unsigned	uint_fast16_t	fast 16 bits unsigned
int32_t	exactly 32 bits signed	int_least32_t	at least 32 bits signed	int_fast32_t	fast 32 bits sig
uint32_t	exactly 32 bits unsigned	uint_least32_t	at least 32 bits unsigned	uint_fast32_t	fast 32 bits unsigned
int64_t	exactly 64 bits signed	int_least64_t	at least 64 bits signed	int_fast64_t	fast 64 bits sig
uint64_t	exactly 64 bits unsigned	uint_least64_t	at least 64 bits unsigned	uint_fast64_t	fast 64 bits unsigned

The ptr aliases are integral types guaranteed to be large enough to hold a pointer without losing bits:

Alias	Description
intptr_t	signed integral type large enough to hold a pointer
uintptr_t	unsigned integral type large enough to hold a pointer

The max aliases are the largest integral types:

Alias	Description
intmax_t	the largest signed integral type
uintmax_t	the largest unsigned integral type

---

std.stdio

Standard I/O functions that extend **std.c.stdio**. **std.c.stdio** is automatically imported when importing **std.stdio**.

void **writef**(...);
Arguments are formatted per the format strings and written to **stdout**.

void **writefln**(...);
Same as **writef**, but a newline is appended to the output.

void **fwritef**(FILE* *fp*, ...);
Same as **writef**, but output is sent to the stream *fp* instead of **stdout**.

void **fwritefln**(FILE* *fp*, ...);
Same as **writefln**, but output is sent to the stream *fp* instead of **stdout**.

---

std.stream

interface **InputStream**
InputStream is the interface for readable streams.

void **readExact**(void* buffer, uint size)
Read exactly size bytes into the buffer, throwing a **ReadException** if it is not

correct.

uint **read**(ubyte[] buffer)
Read a block of data big enough to fill the given array and return the actual number of bytes read. Unfilled bytes are not modified.

void **read**(out byte x)
void **read**(out ubyte x)
void **read**(out short x)
void **read**(out ushort x)
void **read**(out int x)
void **read**(out uint x)
void **read**(out long x)
void **read**(out ulong x)
void **read**(out float x)
void **read**(out double x)
void **read**(out real x)
void **read**(out ifloat x)
void **read**(out idouble x)
void **read**(out ireal x)
void **read**(out cfloat x)
void **read**(out cdouble x)
void **read**(out creal x)
void **read**(out char x)
void **read**(out wchar x)
void **read**(out dchar x)
void **read**(out char[] s)
void **read**(out wchar[] s)
Read a basic type or counted string, throwing a **ReadException** if it could not be read. Outside of byte, ubyte, and char, the format is implementation-specific and should not be used except as opposite actions to **write**.

char[] **readLine**()
char[] **readLine**(char[] buffer)
wchar[] **readLineW**()
wchar[] **readLineW**(wchar[] buffer)
Read a line that is terminated with some combination of carriage return and line feed or end-of-file. The terminators are not included. The wchar version is identical. When a buffer is supplied as a parameter it is filled unless the content does not fit in the buffer, in which case a new buffer is allocated, filled and returned.

char[] **readString**(uint length)
Read a string of the given length, throwing **ReadException** if there was a problem.

wchar[] **readStringW**(uint length)

Read a string of the given length, throwing **ReadException** if there was a problem. The file format is implementation-specific and should not be used except as opposite actions to **write**.

char **getc**()
wchar **getcw**()
Read and return the next character in the stream. This is the only method that will handle **ungetc** properly. **getcw**'s format is implementation-specific.

char **ungetc**(char c)
wchar **ungetcw**(wchar c)
Push a character back onto the stream. They will be returned in first-in last-out order from **getc**/**getcw**.

int **scanf**(char[] fmt, ...)
int **vscanf**(char[] fmt, va_list args)
Scan a string from the input using a similar form to C's scanf.

uint **available**()
Retrieve the nubmer of bytes available for immediate reading.

---

interface **OutputStream**
OutputStream is the interface for writable streams.

void **writeExact**(void* buffer, uint size)
Write exactly *size* bytes from *buffer*, or throw a **WriteException** if that could not be done.

uint **write**(ubyte[] buffer)
Write as much of the *buffer* as possible, returning the number of bytes written.

void **write**(byte x)
void **write**(ubyte x)
void **write**(short x)
void **write**(ushort x)
void **write**(int x)
void **write**(uint x)
void **write**(long x)
void **write**(ulong x)
void **write**(float x)
void **write**(double x)
void **write**(real x)
void **write**(ifloat x)
void **write**(idouble x)
void **write**(ireal x)

void **write**(cfloat x)
void **write**(cdouble x)
void **write**(creal x)
void **write**(char x)
void **write**(wchar x)
void **write**(dchar x)
void **write**(char[] s)
void **write**(wchar[] s)
Write a basic type or counted string. Outside of byte, ubyte, and char, the format is implementation-specific and should only be used in conjunction with **read**.

void **writeLine**(char[] s)
Write a line of text, appending the line with an operating-system-specific line ending.

void **writeLineW**(wchar[] s)
Write a line of text, appending the line with an operating-system-specific line ending. The format is implementation-specific.

void **writeString**(char[] s)
Write a string of text, throwing **WriteException** if it could not be fully written.

void **writeStringW**(wchar[] s)
Write a string of text, throwing **WriteException** if it could not be fully written. The format is implementation-dependent.

uint **printf**(char[] format, ...)
uint **vprintf**(char[] format, va_list args)
Print a formatted string into the stream using printf-style syntax, returning the number of bytes written.

void **writef**(...)
void **writefln**(...)
Print a formatted string into the stream using writef-style syntax. See **std.format**

---

class **Stream** : **InputStream**,**OutputStream**
Stream is the base abstract class from which the other stream classes derive. **Stream**'s byte order is the format native to the computer.

bit **readable**
Indicates whether this stream can be read from.

bit **writeable**

Indicates whether this stream can be written to.

bit **seekable**
Indicates whether this stream can be seeked within.

Reading

These methods require that the **readable** flag be set. Problems with reading result in a **ReadException** being thrown. Stream implements the InputStream interface in addition to the following methods.

uint **readBlock**(void* buffer, uint size)
Read up to size bytes into the buffer and return the number of bytes actually read.

Writing

These methods require that the **writeable** flag be set. Problems with writing result in a **WriteException** being thrown. Stream implements the OutputStream interface in addition to the following methods.

uint **writeBlock**(void* buffer, uint size)
Write up to *size* bytes from *buffer* in the stream, returning the actual number of bytes that were written.

void **copyFrom**(Stream s)
Copies all data from *s* into this stream. This may throw **ReadException** or **WriteException** on failure. This restores the file position of *s* so that it is unchanged.
void **copyFrom**(Stream s, uint count)
Copy a specified number of bytes from the given stream into this one. This may throw **ReadException** or **WriteException** on failure. Unlike the previous form, this doesn't restore the file position of *s*.

Seeking

These methods require that the **seekable** flag be set. Problems with seeking result in a **SeekException** being thrown.

ulong **seek**(long offset, SeekPos whence)
Change the current position of the stream. *whence* is either **SeekPos.Set**, in which case the offset is an absolute index from the beginning of the stream, **SeekPos.Current**, in which case the offset is a delta from the current position,

or **SeekPos.End**, in which case the offset is a delta from the end of the stream (negative or zero offsets only make sense in that case). This returns the new file position.

ulong **seekSet**(long offset)
ulong **seekCur**(long offset)
ulong **seekEnd**(long offset)
Aliases for their normal seek counterparts.

ulong **position**()
void **position**(ulong pos)
Retrieve or set the file position, identical to calling **`seek`**`(0, SeekPos.Current)` or **`seek`**`(pos, SeekPos.Set)` respectively.

ulong **size**()
Retrieve the size of the stream in bytes.

bit **eof**()
Return whether the current file position is the same as the end of the file. This does not require actually reading past the end of the file, as with stdio.

bit **isOpen**()
Return true if the stream is currently open.

void **flush**()
Flush pending output if appropriate.

void **close**()
Close the stream, flushing output if appropriate.

char[] **toString**()
Read the entire stream and return it as a string.

uint **toHash**()
Get a hash of the stream by reading each byte and using it in a CRC-32 checksum.

---

class **BufferedStream** : **Stream**
This subclass is for buffering a source stream. A buffered stream must be closed explicitly to ensure the final buffer content is written to the source stream.

this(Stream source, uint bufferSize = 8192)
Create a buffered stream for the stream *source* with the buffer size *bufferSize*.

---

class **File** : **Stream**

This subclass is for file system streams.

this()
this(char[] filename, FileMode mode = FileMode.In)

Create the stream with no open file, an open file in read mode, or an open file with explicit file mode. *mode*, if given, is a combination of **FileMode.In** (indicating a file that can be read) and **FileMode.Out** (indicating a file that can be written). Opening a file for reading that doesn't exist will error. Opening a file for writing that doesn't exist will create the file. The **FileMode.OutNew** mode will open the file for writing and reset the legnth to zero. The **FileMode.Append** mode will open the file for writing and move the file position to the end of the file.

void **open**(char[] filename, FileMode mode = FileMode.In)

Open a file for the stream, in an identical manner to the constructors.

void **create**(char[] filename, FileMode mode = FileMode.OutNew)

Create a file for the stream.

void **close**()

Close the current file if it is open; otherwise it does nothing.

uint **readBlock**(void* buffer, uint size)
uint **writeBlock**(void* buffer, uint size)
ulong **seek**(long offset, SeekPos rel)

Overrides of the **Stream** methods.

---

class **BufferedFile** : **BufferedStream**

This subclass is for buffered file system streams. It is a convenience class for wrapping a File in a BufferedStream. A buffered stream must be closed explicitly to ensure the final buffer content is written to the file.

this()
this(char[] filename, FileMode mode = FileMode.In, uint buffersize = 8192)
this(File file, uint buffersize = 8192)
void **open**(char[] filename, FileMode mode = FileMode.In)
void **create**(char[] filename, FileMode mode = FileMode.OutNew)
void **close**()
uint **readBlock**(void* buffer, uint size)
uint **writeBlock**(void* buffer, uint size)
ulong **seek**(long offset, SeekPos rel)

Overrides of the **Stream** methods.

---

enum **BOM**

UTF byte-order-mark signatures

UTF8

UTF-8

UTF16LE

UTF-16 Little Endian

UTF16BE

UTF-16 Big Endian

UTF32LE

UTF-32 Little Endian

UTF32BE

UTF-32 Big Endian

class **EndianStream** : **Stream**

This subclass wraps a stream with big-endian or little-endian byte order swapping. UTF Byte-Order-Mark (BOM) signatures can be read and deduced or written.

this(Stream source, Endian end = std.system.endian)

Create the endian stream for the source stream *source* with endianness *end*. The default endianness is the native byte order. The Endian type is defined in the std.system module.

Endian **endian**

property for endianness of the source stream

int **readBOM**(int ungetCharSize = 1)

Return -1 if no BOM and otherwise read the BOM and return it. If there is no BOM or if bytes beyond the BOM are read then the bytes read are pushed back onto the ungetc buffer or ungetcw buffer. Pass ungetCharSize == 2 to use ungetcw instead of ungetc when no BOM is present.

void **writeBOM**(BOM b)

Write the BOM *b* to the source stream

final void **fixBO**(void* buffer, uint size)

fix the byte order of the given buffer to match the native order

---

class **TArrayStream**(**Buffer**) : **Stream**

This subclass wraps an array-like buffer with a stream interface. The type *Buffer* must support the length property and reading ubyte slices.

this(Buffer buf)

Create the stream for the the buffer *buf*.

uint **readBlock**(void* buffer, uint size)

uint **writeBlock**(void* buffer, uint size)

ulong **seek**(long offset, SeekPos rel)

char[] **toString**()

Overrides of **Stream** methods.

---

class **MemoryStream** : **TArrayStream!(ubyte[])**

This subclass reads and constructs an array of bytes in memory.

this()
this(ubyte[] data)

Create the output buffer and setup for reading, writing, and seeking. The second constructor loads it with specific input data.

ubyte[] **data**()

Get the current memory data in total.

uint **readBlock**(void* buffer, uint size)
uint **writeBlock**(void* buffer, uint size)
ulong **seek**(long offset, SeekPos rel)
char[] **toString**()

Overrides of **Stream** methods.

---

class **SliceStream** : **Stream**

This subclass slices off a portion of another stream, making seeking relative to the boundaries of the slice. It could be used to section a large file into a set of smaller files, such as with tar archives.

this(Stream base, int low)

Indicate both the base stream to use for reading from and the low part of the slice. The high part of the slice is dependent upon the end of the base stream, so that if you write beyond the end it resizes the stream normally.

this(Stream base, int low, int high)

Indicate the high index as well. Attempting to read or write past the high index results in the end being clipped off.

uint **readBlock**(void* buffer, uint size)
uint **writeBlock**(void* buffer, uint size)
ulong **seek**(long offset, SeekPos rel)

Overrides of **Stream** methods.

---

std.string

To copy or not to copy?

When a function takes a string as a parameter, and returns a string, is that string the same as the input string, modified in place, or is it a modified copy of the input string? The D array convention is "copy-on-write". This means that if no modifications are done, the original string (or slices of it) can be returned. If any modifications are done, the returned string is a copy.

class **StringException**
Thrown on errors in string functions.

const char[] **hexdigits**;
"0123456789ABCDEF"

const char[] **digits**;
"0123456789"

const char[] **octdigits**;
"01234567"

const char[] **lowercase**;
"abcdefghijklmnopqrstuvwxyz"

const char[] **uppercase**;
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"

const char[] **letters**;
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"

const char[] **whitespace**;
" \t\v\r\n\f"

long **atoi**(char[] s)
Convert string to integer.

real **atof**(char[] s)
Convert string to real.

int **cmp**(char[] s1, char[] s2)
Compare two strings. Returns:
<0 for (s1 < s2)
=0 for (s1 == s2)
>0 for (s1 > s2)

int **icmp**(char[] s1, char[] s2)
Same as **cmp**() but case insensitive.

char* **toStringz**(char[] string)
Converts a D array of chars to a C-style 0 terminated string.

int **find**(char[] s, dchar c)
Find first occurrance of c in string s. Return index in s where it is found. Return -1 if not found.

int **rfind**(char[] s, dchar c)
Find last occurrance of c in string s. Return index in s where it is found. Return -1 if not found.

int **find**(char[] s, char[] sub)
Find first occurrance of *sub[]* in string *s[]*. Return index in *s[]* where it is found. Return -1 if not found.

int **rfind**(char[] s, char[] sub)
Find last occurrance of *sub* in string *s*. Return index in *s* where it is found. Return -1 if not found.

int **ifind**(char[] s, dchar c)
int **irfind**(char[] s, dchar c)
int **ifind**(char[] s, char[] sub)
int **irfind**(char[] s, char[] sub)
Case insensitive versions.

char[] **tolower**(char[] s)
Convert string to lower case.

char[] **toupper**(char[] s)
Convert string to upper case.

char[] **capitalize**(char[] s)
Capitalize first character of string.

char[] **capwords**(char[] s)
Capitalize all words in string. Remove leading and trailing whitespace. Replace all sequences of whitespace with a single space.

char[] **join**(char[][] words, char[] sep)
Concatenate all the strings together into one string; use *sep[]* as the separator.

char[][] **split**(char[] s)
Split *s[]* into an array of words, using whitespace as the delimiter.

char[][] **split**(char[] s, char[] delim)
Split *s[]* into an array of words, using *delim[]* as the delimiter.

char[][] **splitlines**(char[] s)
Split *s[]* into an array of lines, using CR, LF, or CR-LF as the delimiter.

char[] **stripl**(char[] s)
char[] **stripr**(char[] s)
char[] **strip**(char[] s)
Strips leading or trailing whitespace, or both.

char[] **ljustify**(char[] s, int width)
char[] **rjustify**(char[] s, int width)
char[] **center**(char[] s, int width)
Left justify, right justify, or center string in field *width* chars wide.

char[] **zfill**(char[] s, int width)
Same as **rjustify**(), but fill with '0's.

char[] **replace**(char[] s, char[] from, char[] to)
Replace occurrences of *from[]* with *to[]* in *s[]*.

char[] **replaceSlice**(char[] string, char[] slice, char[] replacement)
Given a *string[]* with a *slice[]* into it, replace *slice[]* with *replacement[]*.

char[] **insert**(char[] s, int index, char[] sub)
Insert *sub*[] into *s*[] at location *index*.

int **count**(char[] s, char[] sub)
Count up all instances of *sub[]* in *s[]*.

char[] **expandtabs**(char[] s, int tabsize)
Replace tabs with the appropriate number of spaces. *tabsize* is the distance between tab stops.

char[] **maketrans**(char[] from, char[] to)
Construct translation table for **translate**().

char[] **translate**(char[] s, char[] transtab, char[] delchars)
Translate characters in *s[]* using table created by **maketrans**(). Delete chars in *delchars[]*.

char[] **toString**(uint u)
Convert uint to string.

char[] **toString**(char* s)

Convert C-style 0 terminated string to D string.

---

std.system

enum **Endian**
Byte order endianness
BigEndian
big endian byte order
LittleEndian
little endian byte order

Endian **endian**
Native system endianness

---

std.thread

The thread module defines the class **Thread**. **Thread** is the basis for writing multithreaded applications. Each thread has a unique instance of class **Thread** associated with it. It is important to use the **Thread** class to create and manage threads as the garbage collector needs to know about all the threads.

typedef ... **thread_hdl**
The type of the thread handle used by the operating system.

class **Thread**
One for each thread.

class **ThreadError**
Thrown for errors.

The members of **Thread** are:

this()
Constructor used by classes derived from **Thread** that override **main**().

this(int (*fp)(void*), void* arg)
Constructor used by classes derived from **Thread** that override **run**().

this(int delegate() dg)
Constructor used by classes derived from **Thread** that override **run**().

thread_hdl **hdl**;
The handle to this thread assigned by the operating system. This is set to

thread_id.init if the thread hasn't been started yet.

void **start**();
Create a new thread and start it running. The new thread initializes itself and then calls **run**(). **start**() can only be called once.

int **run**(void* p);
Entry point for a thread. If not overridden, it calls the function pointer *fp* and argument *arg* passed in the constructor, or the delegate *dg*. The return value is the thread exit code, which is normally 0.

void **wait**();
Wait for this thread to terminate. Throws **ThreadError** if the thread hasn't begun yet or has already terminated or is called on itself.

void **wait**(unsigned milliseconds);
Wait for this thread to terminate or until milliseconds time has elapsed, whichever occurs first. Throws **ThreadError** if the thread hasn't begun yet or has already terminated or is called on itself.

TS **getState**();
Returns the state of the thread. The state is one of the following:

TS	Description
INITIAL	The thread hasn't been started yet.
RUNNING	The thread is running or paused.
TERMINATED	The thread has ended.

void **setPriority**(PRIORITY* p);
Adjust the priority of this thread.

PRIORITY	Description
INCREASE	Increase thread priority
DECREASE	Decrease thread priority
IDLE	Assign thread low priority
CRITICAL	Assign thread high priority

static Thread **getThis**();
Returns a reference to the **Thread** for the thread that called the function.

static Thread[] **getAll**();
Returns an array of all the threads currently running.

void **pause**();

Suspend execution of this thread.

void **resume**();
Resume execution of this thread.

static void **pauseAll**();
Suspend execution of all threads but this thread.

static void **resumeAll**();
Resume execution of all paused threads.

static void **yield**();
Give up the remainder of this thread's time slice.

---

std.uri

Encode and decode Uniform Resource Identifiers (URIs). URIs are used in internet transfer protocols. Valid URI characters consist of letters, digits, and the characters ;/?:@&=+$,-_.!~*'(). Escape sequences consist of '%' followed by two hex digits.

char[] **decode**(char[] encodedURI)
Decodes the URI string *encodedURI* into a UTF-8 string and returns it. Escape sequences that resolve to valid URI characters are not replaced. Escape sequences that resolve to the '#' character are not replaced.

char[] **decodeComponent**(char[] encodedURIComponent)
Decodes the URI string *encodedURI* into a UTF-8 string and returns it. All escape sequences are decoded.

char[] **encode**(char[] uri)
Encodes the UTF-8 string *uri* into a URI and returns that URI. Any character not a valid URI character is escaped. The '#' character is not escaped.

char[] **encodeComponent**(char[] uriComponent)
Encodes the UTF-8 string *uri* into a URI and returns that URI. Any character not a letter, digit, or one of -_.!~*'() is escaped.

---

std.utf

Encode and decode UTF-8, UTF-16 and UTF-32 strings. For more information on UTF-8, see http://www.cl.cam.ac.uk/~mgk25/unicode.html#utf-8.

Note: For Win32 systems, the C **wchar_t** type is UTF-16 and corresponds to the D **wchar** type.

For linux systems, the C **wchar_t** type is UTF-32 and corresponds to the D **utf.dchar** type.

UTF character support is restricted to (0 <= character <= 0x10FFFF).

class **UtfError**

Exception class that is thrown upon any errors. The members are:

idx

Set to the index of the start of the offending UTF sequence.

alias ... **dchar**

An alias for a single UTF-32 character. This may become a D basic type in the future.

bit **isValidDchar**(dchar c)

Test if *c* is a valid UTF-32 character. Returns **true** if it is, **false** if not.

dchar **decode**(char[] s, inout uint idx)
dchar **decode**(wchar[] s, inout uint idx)
dchar **decode**(dchar[] s, inout uint idx)

Decodes and returns character starting at *s[idx]*. *idx* is advanced past the decoded character. If the character is not well formed, a **UriError** is thrown and *idx* remains unchanged.

void **encode**(inout char[] s, dchar c)
void **encode**(inout wchar[] s, dchar c)
void **encode**(inout dchar[] s, dchar c)

Encodes character *c* and appends it to array *s*.

void **validate**(char[] s)
void **validate**(wchar[] s)
void **validate**(dchar[] s)

Checks to see if string is well formed or not. Throws a **UtfError** if it is not. Use to check all untrusted input for correctness.

char[] **toUTF8**(char[] s)
char[] **toUTF8**(wchar[] s)
char[] **toUTF8**(dchar[] s)

Encodes string *s* into UTF-8 and returns the encoded string.

wchar[] **toUTF16**(char[] s)
wchar* **toUTF16z**(char[] s)
wchar[] **toUTF16**(wchar[] s)
wchar[] **toUTF16**(dchar[] s)

Encodes string *s* into UTF-16 and returns the encoded string. **toUTF16z** is suitable for calling the 'W' functions in the Win32 API that take an LPWSTR or LPCWSTR argument.

dchar[] **toUTF32**(char[] s)

dchar[] **toUTF32**(wchar[] s)
dchar[] **toUTF32**(dchar[] s)
Encodes string *s* into UTF-32 and returns the encoded string.

---

std.zip

Read/write data in the zip archive format. Makes use of the zlib compression library.

class **ZipException**
Thrown on error.

class **ZipArchive**
Object representing the entire archive. **ZipArchive**s are collections of **ArchiveMember**s.

this()
Constructor used when creating a new archive.

void **addMember**(ArchiveMember de)
Add *de* to the archive.

void **deleteMember**(ArchiveMember de)
Delete *de* from the archive.

void[] **build**()
Construct an archive out of the current members of the archive. Fills in the properties **data[]**, **diskNumber**, **diskStartDir**, **numEntries**, **totalEntries**, and **directory[]**. For each **ArchiveMember**, fills in properties **crc32**, **compressedSize**, **compressedData[]**. Return array representing the entire archive.

this(void[] data)
Constructor used when reading an existing archive. *data*[] is the entire contents of the archive. Fills in the properties **data[]**, **diskNumber**, **diskStartDir**, **numEntries**, **totalEntries**, **comment[]**, and **directory[]**. For each **ArchiveMember**, fills in properties **madeVersion**, **extractVersion**, **flags**, **compressionMethod**, **time**, **crc32**, **compressedSize**, **expandedSize**, **compressedData[]**, **diskNumber**, **internalAttributes**, **externalAttributes**, **name[]**, **extra[]**, **comment[]**. Use **expand()** to get the expanded data for each **ArchiveMember**.

ubyte[] **expand**(ArchiveMember de)
Decompress the contents of archive member *de* and return the expanded data. Fills in properties **extractVersion**, **flags**, **compressionMethod**, **time**, **crc32**, **compressedSize**, **expandedSize**, **expandedData[]**, **name[]**,

extra[].

ubyte[] **data**
Read Only: array representing the entire contents of the archive.

uint **diskNumber**
Read Only: 0 since multi-disk zip archives are not supported.

uint **diskStartDir**
Read Only: 0 since multi-disk zip archives are not supported.

uint **numEntries**
Read Only: number of **ArchiveMember**s in the directory.

uint **totalEntries**
Read Only: same as **totalEntries**.

char[] **comment**
Read/Write: the archive comment. Must be less than 65536 bytes in length.

ArchiveMember[char[]] **directory**
Read Only: array indexed by the name of each member of the archive. All the members of the archive can be accessed with a foreach loop:

```
ZipArchive archive = new ZipArchive(data);
foreach (ArchiveMember am; archive)
{
    printf("member name is '%.*s'\n", am.name);
}
```

class **ArchiveMember**
A member of the **ZipArchive**.

ushort **madeVersion**
Read Only

ushort **extractVersion**
Read Only

ushort **flags**
Read/Write: normally set to 0

ushort **compressionMethod**
Read/Write: the only supported values are 0 (no compression) and 8 (deflate).

date.DosFileTime **time**
Read/Write: Last modified time of the member. It's in the DOS date/time format.

uint **crc32**
Read Only: cyclic redundancy check (CRC) value

uint **compressedSize**
Read Only: size of data of member in compressed form.

uint **expandedSize**
Read Only: size of data of member in expanded form.

ushort **diskNumber**
Read Only: should be 0.

ushort **internalAttributes**
Read/Write

uint **externalAttributes**
Read/Write

char[] **name**
Read/Write: Usually the file name of the archive member; it is used to index the archive directory for the member. Each member must have a unique **name[]**. Do not change without removing member from the directory first.

ubyte[] **extra**
Read/Write: extra data for this member.

char[] **comment**
Read/Write: comment associated with this member.

ubyte[] **compressedData**
Read Only: data of member in compressed form.

ubyte[] **expandedData**
Read/Write: data of member in uncompressed form.

Bugs:

Multi-disk zips not supported.
Only Zip version 20 formats are supported.
Only supports compression modes 0 (no compression) and 8 (deflate).
Does not support encryption.

std.zlib

Compress / decompress data using the zlib library.

uint **adler32**(uint *adler*, void[] *buf*)
Compute the Adler32 checksum of the data in *buf*[]. *adler* is the starting value when computing a cumulative checksum.

uint **crc32**(uint *crc*, void[] *buf*)
Compute the CRC32 checksum of the data in *buf*[]. *crc* is the starting value when computing a cumulative checksum.

class **ZlibException**
Thrown in the case of errors occurring in the following:

void[] **compress**(void[] *buf*)
void[] **compress**(void[] *buf*, int *level*)
Compresses the data in *buf*[] using compression level *level*. The default value for *level* is 6, legal values are 1..9, with 1 being the least compression and 9 being the most. Returns the compressed data.

void[] **uncompress**(void[] *buf*)
void[] **uncompress**(void[] *buf*, uint *destbufsize*)
Decompresses the data in *buf*[]. *destbufsize* is the size of the uncompressed data. It need not be accurate, but the decompression will be faster if the exact size is supplied. Returns the decompressed data.

class **Compress**
Used when the data to be compressed is not all in one buffer.

this()
this(int *level*)
Construct. *level* is the same as for **D.zlib.compress**().

void[] **compress**(void[] *buf*)
Compress the data in *buf* and return the compressed data. The buffers returned from successive calls to this should be concatenated together.

void[] **flush**()
void[] **flush**(int *mode*)
Compress and return any remaining data. The returned data should be appended to that returned by **compress**(). *mode* is one of the following:
Z_SYNC_FLUSH
Syncs up flushing to the next byte boundary. Used when more data is to be compressed later on.
Z_FULL_FLUSH

Syncs up flushing to the next byte boundary. Used when more data is to be compressed later on, and the decompressor needs to be restartable at this point.

Z_FINISH (default)
Used when finished compressing the data.

class **UnCompress**
Used when the data to be decompressed is not all in one buffer.

this()
this(uint *destbufsize*)
Construct. *destbufsize* is the same as for **D.zlib.uncompress**().

void[] **uncompress**(void[] *buf*)
Decompress the data in *buf* and return the decompressed data. The buffers returned from successive calls to this should be concatenated together.

void[] **flush**()
Decompress and return any remaining data. The returned data should be appended to that returned by **uncompress**(). The **UnCompress** object cannot be used further.

---

std.c.stdio

int **printf**(char* format, ...)
C printf() function.

D for Win32

This describes the D implementation for 32 bit Windows systems. Naturally, Windows specific D features are not portable to other platforms.

Instead of the:

```
#include <windows.h>
```

of C, in D there is:

```
import std.c.windows.windows;
```

Calling Conventions

In C, the Windows API calling conventions are __stdcall. In D, it is simply:

```
extern (Windows)
{
        ... function declarations ...
}
```

The Windows linkage attribute sets both the calling convention and the name mangling scheme to be compatible with Windows.

For functions that in C would be __declspec(dllimport) or __declspec(dllexport), use the export attribute:

```
export void func(int foo);
```

If no function body is given, it's imported. If a function body is given, it's exported.

Windows Executables

Windows GUI applications can be written with D. A sample such can be found in `\dmd\samples\d\winsamp.d`

These are required:

1. Instead of a `main` function serving as the entry point, a `WinMain` function is needed.
2. `WinMain` must follow this form:

```
3. import std.c.windows.windows;
4.
5. extern (C) void gc_init();
6. extern (C) void gc_term();
7. extern (C) void _minit();
8. extern (C) void _moduleCtor();
9. extern (C) void _moduleUnitTests();
10.
11.extern (Windows)
12.int WinMain(HINSTANCE hInstance,
13.        HINSTANCE hPrevInstance,
14.        LPSTR lpCmdLine,
15.        int nCmdShow)
16.{
17.    int result;
18.
19.    gc_init();                  // initialize garbage collector
20.    _minit();                   // initialize module constructor table
21.
22.    try
23.    {
24.        _moduleCtor();          // call module constructors
25.        _moduleUnitTests();     // run unit tests (optional)
```

```
26.
27.         result = myWinMain(hInstance, hPrevInstance, lpCmdLine,
  nCmdShow);
28.     }
29.
30.     catch (Object o)            // catch any uncaught exceptions
31.     {
32.         MessageBoxA(null, cast(char *)o.toString(), "Error",
33.                     MB_OK | MB_ICONEXCLAMATION);
34.         result = 0;             // failed
35.     }
36.
37.     gc_term();                  // run finalizers; terminate garbage
  collector
38.     return result;
39.}
40.
41.int myWinMain(HINSTANCE hInstance,
42.         HINSTANCE hPrevInstance,
43.         LPSTR lpCmdLine,
44.         int nCmdShow)
45.{
46.     ... insert user code here ...
47.}
48.
```

49. The `myWinMain()` function is where the user code goes, the rest of `WinMain` is boilerplate to initialize and shut down the D runtime system.
50. A .def (Module Definition File) with at least the following two lines in it:

```
51.EXETYPE NT
52.SUBSYSTEM WINDOWS
53.
```

54. Without those, Win32 will open a text console window whenever the application is run.
55. The presence of `WinMain()` is recognized by the compiler causing it to emit a reference to __acrtused_dll and the phobos.lib runtime library.

DLLs (Dynamic Link Libraries)

DLLs can be created in D in roughly the same way as in C. A `DllMain()` is required, looking like:

```
import std.c.windows.windows;
HINSTANCE g_hInst;

extern (C)
{
        void gc_init();
        void gc_term();
        void _minit();
        void _moduleCtor();
        void _moduleUnitTests();
}
```

```
extern (Windows)
BOOL DllMain(HINSTANCE hInstance, ULONG ulReason, LPVOID pvReserved)
{
    switch (ulReason)
    {
        case DLL_PROCESS_ATTACH:
            gc_init();                  // initialize GC
            _minit();                   // initialize module list
            _moduleCtor();              // run module constructors
            _moduleUnitTests();         // run module unit tests
            break;

        case DLL_PROCESS_DETACH:
            gc_term();                  // shut down GC
            break;

        case DLL_THREAD_ATTACH:
        case DLL_THREAD_DETACH:
            // Multiple threads not supported yet
            return false;
    }
    g_hInst=hInstance;
    return true;
}
```

Notes:

The _moduleUnitTests() call is optional.
It's a little crude, I hope to improve it.
The presence of `DllMain()` is recognized by the compiler causing it to emit a reference to __acrtused_dll and the phobos.lib runtime library.

Link with a .def (Module Definition File) along the lines of:

```
LIBRARY         MYDLL
DESCRIPTION     'My DLL written in D'

EXETYPE         NT
CODE            PRELOAD DISCARDABLE
DATA            PRELOAD SINGLE

EXPORTS
                DllGetClassObject       @2
                DllCanUnloadNow         @3
                DllRegisterServer       @4
                DllUnregisterServer     @5
```

The functions in the EXPORTS list are for illustration. Replace them with the actual exported functions from MYDLL. Alternatively, use implib. Here's an example of a simple DLL with a function print() which prints a string:

mydll2.d:

```
module mydll;
export void dllprint() { printf("hello dll world\n"); }
```

mydll.def:

```
LIBRARY "mydll.dll"
EXETYPE NT
SUBSYSTEM WINDOWS
CODE SHARED EXECUTE
DATA WRITE
```

Put the code above that contains DllMain() into a file dll.d. Compile and link the dll with the following command:

```
dmd -ofmydll.dll mydll2.d dll.d mydll.def
implib/system mydll.lib mydll.dll
```

which will create mydll.dll and mydll.lib. Now for a program, test.d, which will use the dll:

test.d:

```
import mydll;

int main()
{
   mydll.dllprint();
   return 0;
}
```

Create a clone of mydll2.d that doesn't have the function bodies:

mydll.d:

```
export void dllprint();
```

Compile and link with the command:

```
dmd test.d mydll.lib
```

and run:

```
C:>test
hello dll world
```

```
C:>
```

Memory Allocation

D DLLs use garbage collected memory management. The question is what happens when pointers to allocated data cross DLL boundaries? Other DLLs, or callers to a D DLL, may even be written in another language and may have no idea how to interface with D's garbage collector.

There are many approaches to solving this problem. The most practical approaches are to assume that other DLLs have no idea about D. To that end, one of these should work:

Do not return pointers to D gc allocated memory to the caller of the DLL. Instead, have the caller allocate a buffer, and have the DLL fill in that buffer.

Retain a pointer to the data within the D DLL so the GC will not free it. Establish a protocol where the caller informs the D DLL when it is safe to free the data.

Use operating system primitives like VirtualAlloc() to allocate memory to be transferred between DLLs.

Use COM interfaces, rather than D class objects. D supports the AddRef()/Release() protocol for COM interfaces. Most languages implemented on Win32 have support for COM, making it a good choice.

COM Programming

Many Windows API interfaces are in terms of COM (Common Object Model) objects (also called OLE or ActiveX objects). A COM object is an object who's first field is a pointer to a vtbl[], and the first 3 entries in that vtbl[] are for QueryInterface(), AddRef(), and Release().

COM objects are analogous to D interfaces. Any COM object can be expressed as a D interface, and every D object with an interface X can be exposed as a COM object X. This means that D is compatible with COM objects implemented in other languages.

While not strictly necessary, the Phobos library provides an Object useful as a super class for all D COM objects, called ComObject. ComObject provides a default implementation for QueryInterface(), AddRef(), and Release().

Windows COM objects use the Windows calling convention, which is not the default for D, so COM functions need to have the attribute extern (Windows). So, to write a COM object:

```
import std.c.windows.com;

class MyCOMobject : ComObject
{
    extern (Windows):
        ...
}
```

The sample code includes an example COM client program and server DLL.

Converting C .h Files to D Modules

While D cannot directly compile C source code, it can easily interface to C code, be linked with C object files, and call C functions in DLLs. The interface to C code is normally found in C `.h` files. So, the trick to connecting with C code is in converting C `.h` files to D modules. This turns out to be difficult to do mechanically since inevitably some human judgement must be applied. This is a guide to doing such conversions.

Preprocessor

`.h` files can sometimes be a bewildering morass of layers of macros, `#include` files, `#ifdef`'s, etc. D doesn't support a text preprocessor, so the first step is to remove the need for it by taking the preprocessed output. For DMC (the Digital Mars C/C++ compiler), the command:

```
dmc -c program.h -e -l
```

will create a file `program.lst` which is the source file after all text preprocessing.

Remove all the `#if`, `#ifdef`, `#include`, etc. statements.

Linkage

Generally, surround the entire module with:

```
extern (C)
{
     ...file contents...
}
```

to give it C linkage.

Types

A little global search and replace will take care of renaming the C types to D types. The following table shows a typical mapping for 32 bit C code:

C type	D type
long double	real
unsigned long long	ulong
long long	long
unsigned long	uint
long	int

unsigned	uint
unsigned short	ushort
signed char	byte
unsigned char	ubyte
wchar_t	wchar or dchar
bool	int

NULL

Or `((void*)0)` should be replaced with `null`.

Numeric Literals

Any 'L' or 'l' numeric literal suffixes should be removed, as a C `long` is (usually) the same size as a D `int`. Similarly, 'LL' suffixes should be replaced with a single 'L'. Any 'u' suffix will work the same in D.

String Literals

In most cases, any 'L' prefix to a string can just be dropped, as D will implicitly convert strings to wide characters if necessary. However, one can also replace:

```
L"string"
```

with:

```
cast(wchar[])"string"
```

Macros

Lists of macros like:

```
#define FOO     1
#define BAR     2
#define ABC     3
#define DEF     40
```

can be replaced with:

```
enum
{   FOO = 1,
    BAR = 2,
    ABC = 3,
    DEF = 40
```

```
}
```

or with:

```
const int FOO = 1;
const int BAR = 2;
const int ABC = 3;
const int DEF = 40;
```

Function style macros, such as:

```
#define MAX(a,b) ((a) < (b) ? (b) : (a))
```

can be replaced with functions:

```
int MAX(int a, int b) { return (a < b) ? b : a); }
```

Declaration Lists

D doesn't allow declaration lists to change the type. Hence:

```
int *p, q, t[3], *s;
```

should be written as:

```
int* p, s;
int q;
int[3] t;
```

Void Parameter Lists

Functions that take no parameters:

```
int foo(void);
```

are in D:

```
int foo();
```

Const Type Modifiers

D has `const` as a storage class, not a type modifier. Hence, just drop any `const` used as a type modifier:

```
void foo(const int *p, char *const q);
```

becomes:

```
void foo(int* p, char* q);
```

Typedef

`alias` is the D equivalent to the C `typedef`:

```
typedef int foo;
```

becomes:

```
alias int foo;
```

Structs

Replace declarations like:

```
typedef struct Foo
{   int a;
    int b;
} Foo, *pFoo, *lpFoo;
```

with:

```
struct Foo
{   int a;
    int b;
}
alias Foo* pFoo, lpFoo;
```

Struct Member Alignment

A good D implementation by default will align struct members the same way as the C compiler it was designed to work with. But if the `.h` file has some `#pragma`'s to control alignment, they can be duplicated with the D `align` attribute:

```
#pragma pack(1)
struct Foo
{
    int a;
    int b;
};
#pragma pack()
```

becomes:

```
struct Foo
{
  align (1):
    int a;
    int b;
}
```

Nested Structs

```
struct Foo
{
    int a;
    struct Bar
    {
        int c;
    } bar;
};

struct Abc
{
    int a;
    struct
    {
        int c;
    } bar;
};
```

becomes:

```
struct Foo
{
    int a;
    struct Bar
    {
        int c;
    }
    Bar bar;
};

struct Abc
{
    int a;
    struct
    {
        int c;
    }
};
```

__cdecl, __pascal, __stdcall

```
int __cdecl x;
int __cdecl foo(int a);
int __pascal bar(int b);
int __stdcall abc(int c);
```

become:

```
extern (C) int x;
extern (C) int foo(int a);
extern (Pascal) int bar(int b);
extern (Windows) int abc(int c);
```

__declspec(dllimport)

```
__declspec(dllimport) int __stdcall foo(int a);
```

becomes:

```
export extern (Windows) int foo(int a);
```

__fastcall

Unfortunately, D doesn't support the `__fastcall` convention. Therefore, a shim will be needed, either written in C:

```
int __fastcall foo(int a);

int myfoo(int a)
{
    return foo(int a);
}
```

and compiled with a C compiler that supports `__fastcall` and linked in, or compile the above, disassemble it with obj2asm and insert it in a D `myfoo` shim with inline assembler.

FAQ

The same questions keep cropping up, so the obvious thing to do is prepare a FAQ.

The D FAQ wiki with many more questions answered.

Why the name D?

Why the name D?

The original name was the Mars Programming Language. But my friends kept calling it D, and I found myself starting to call it D. The idea of D being a successor to C goes back at least as far as 1988, as in this thread.

Where can I get a D compiler?

Right here.

Is there a linux port of D?

Yes, the D compiler includes a linux version.

Is there a GNU version of D?

Yes, David Friedman has integrated the D frontend with GCC.

How do I write my own D compiler for CPU X?

Burton Radons has written a back end. you can use as a guide.

Where can I get a GUI for D?

DUI is a graphical user interface based on the GTK+ graphical toolkit.

Where can I get an IDE for D?

LEDS is a D language editor for Linux.

What about templates?

D now supports advanced templates.

Why emphasize implementation ease?

Isn't ease of use for the user of the language more important? Yes, it is. But a vaporware language is useless to everyone. The easier a language is to implement, the more robust implementations there will be. In C's heyday, there were 30 different commericial C compilers for the IBM PC. Not many made the transition to C++. In looking at the C++ compilers on the market today, how many years of development went into each? At least 10 years? Programmers waited years for the various pieces of C++ to get implemented after they were specified. If C++ was not so enormously popular, it's doubtful that very complex features like multiple inheritance, templates, etc., would ever have been implemented.

I suggest that if a language is easier to implement, then it is likely also easier to understand. Isn't it better to spend time learning to write better programs than language arcana? If a language can capture 90% of the power of C++ with 10% of its complexity, I argue that is a worthwhile tradeoff.

Why is [expletive deleted] printf in D?

printf is not typesafe. It's old fashioned. It's not object-oriented. It's not usable with user-defined types. **printf** is guilty as charged. But it's just so darned useful. Nothing beats it for banging out a quick dump of a value when debugging.

Note: printf is actually not really part of D anyway, but since D provides easy access to C's runtime library, D gets it when needed.

Will D be open source?

The front end for D is open source, and the source comes with the compiler. There is a SourceForge project underway to create a Gnu implementation of D from this.

Why fall through on switch statements?

Many people have asked for a requirement that there be a break between cases in a switch statement, that C's behavior of silently falling through is the cause of many bugs.

The reason D doesn't change this is for the same reason that integral promotion rules and operator precedence rules were kept the same - to make code that looks the same as in C operate the same. If it had subtly different semantics, it will cause frustratingly subtle bugs.

Why should I use D instead of Java?

D is distinct from Java in purpose, philosophy and reality. See this comparison.

Java is designed to be write once, run everywhere. D is designed for writing efficient native system apps. Although D and Java share the notion that garbage collection is good and multiple inheritance is bad <g>, their different design goals mean the languages have very different feels.

Doesn't C++ support strings, bit arrays, etc. with STL?

In the C++ standard library are mechanisms for doing strings, bit arrays, dynamic arrays, associative arrays, bounds checked arrays, and complex numbers.

Sure, all this stuff can be done with libraries, following certain coding disciplines, etc. But you can also do object oriented programming in C (I've seen it done). Isn't it incongruous that something like strings, supported by the simplest BASIC interpreter, requires a very large and complicated infrastructure to support? Just the implementation of a string type in STL is over two thousand lines of code, using every advanced feature of templates. How much confidence can you have that this is all working correctly, how do you fix it if it is not, what do you do with

the notoriously inscrutable error messages when there's an error using it, how can you be sure you are using it correctly (so there are no memory leaks, etc.)?

D's implementation of strings is simple and straightforward. There's little doubt how to use it, no worries about memory leaks, error messages are to the point, and it isn't hard to see if it is working as expected or not.

Can't garbage collection be done in C++ with an add-on library?

Yes, I use one myself. It isn't part of the language, though, and requires some subverting of the language to make it work. Using gc with C++ isn't for the standard or casual C++ programmer. Building it into the language, like in D, makes it practical for everyday programming chores.

GC isn't that hard to implement, either, unless you're building one of the more advanced ones. But a more advanced one is like building a better optimizer - the language still works 100% correctly even with a simple, basic one. The programming community is better served by multiple implementations competing on quality of code generated rather than by which corners of the spec are implemented at all.

Can't unit testing be done in C++ with an add-on library?

Sure. Try one out and then compare it with how D does it. It'll be quickly obvious what an improvement building it into the language is.

Why have an asm statement in a portable language?

An asm statement allows assembly code to be inserted directly into a D function. Assembler code will obviously be inherently non-portable. D is intended, however, to be a useful language for developing systems apps. Systems apps almost invariably wind up with system dependent code in them anyway, inline asm isn't much different. Inline asm will be useful for things like accessing special CPU instructions, accessing flag bits, special computational situations, and super optimizing a piece of code.

Before the C compiler had an inline assembler, I used external assemblers. There was constant grief because many, many different versions of the assembler were out there, the vendors kept changing the syntax of the assemblers, there were many different bugs in different versions, and even the command line syntax kept changing. What it all meant was that users could not reliably rebuild any code that needed assembler. An inline assembler provided reliability and consistency.

Is there a GUI library for D?

Since D can call C functions, any GUI library with a C interface is accessible from D. Various D GUI libraries and ports can be found at AvailableGuiLibraries.

What is the point of 80 bit reals?

More precision enables more accurate floating point computations to be done, especially when adding together large numbers of small real numbers. Prof. Kahan, who designed the Intel floating point unit, has an eloquent paper on the subject.

How do I do anonymous struct/unions in D?

```
struct Foo
{
    union { int a; int b; }
    struct { int c; int d; }
}

void main()
{
    Foo f;

    printf("Foo.size = %d, a.offset = %d, b.offset = %d, c.offset = %d,
d.offset = %d\n",
        f.size,
        0,
        &f.b - &f.a,
        &f.c - &f.a,
        &f.d - &f.a);
}
```

How do I get printf() to work with strings?

In C, the normal way to printf a string is to use the **%s** format:

```
char s[8];
strcpy(s, "foo");
printf("string = '%s'\n", s);
```

Attempting this in D, as in:

```
char[] s;
s = "foo";
printf("string = '%s'\n", s);
```

usually results in garbage being printed, or an access violation. The cause is that in C, strings are terminated by a 0 character. The **%s** format prints until a 0 is encountered. In D, strings are not 0 terminated, the size is determined by a separate length value. So, strings are printf'd using the **%.*s** format:

```
char[] s;
s = "foo";
printf("string = '%.*s'\n", s);
```

which will behave as expected. Remember, though, that printf's **%.*s** will print until the length is reached or a 0 is encountered, so D strings with embedded 0's will only print up to the first 0.

Why are floating point values default initialized to nan rather than 0?

A floating point value, if no explicit initializer is given, is initialized to nan (Not A Number):

```
double d;       // d is set to double.nan
```

Nan's have the interesting property in that whenever a nan is used as an operand in a computation, the result is a nan. Therefore, nan's will propagate and appear in the output whenever a computation made use of one. This implies that a nan appearing in the output is an unambiguous indication of the use of an uninitialized variable.

If 0.0 was used as the default initializer for floating point values, its effect can easily be unnoticed in the output, and so if the default initializer was unintended, the bug may go unrecognized.

The default initializer value is not meant to be a useful value, it is meant to expose bugs. Nan fills that role well.

But surely the compiler can detect and issue an error message for variables used that are not initialized? Most of the time, it can, but not always, and what it can do is dependent on the sophistication of the compiler's internal data flow analysis. Hence, relying on such is unportable and unreliable.

Because of the way CPUs are designed, there is no nan value for integers, so D uses 0 instead. It doesn't have the advantages of error detection that nan has, but at least errors resulting from unintended default initializations will be consistent and therefore more debuggable.

Why is overload the assignment operator not supported?

Most of the assignment operator overloading in C++ seems to be needed to just keep track of who owns the memory. So by using reference types coupled with GC, most of this just gets replaced with copying the reference itself. For example, given an array of class objects, the

array's contents can be moved, sorted, shifted, etc., all without any need for overloaded assignments. Ditto for function parameters and return values. The references themselves just get moved about. There just doesn't seem to be any need for copying the entire contents of one class object into another pre-existing class object.

Sometimes, one does need to create a copy of a class object, and for that one can still write a copy constructor in D, but they just don't seem to be needed remotely as much as in C++.

Structs, being value objects, do get copied about. A copy is defined in D to be a bit copy. I've never been comfortable with any object in C++ that does something other than a bit copy when copied. Most of this other behavior stems from that old problem of trying to manage memory. Absent that, there doesn't seem to be a compelling rationale for having anything other than a bit copy.

The '~' is not on my keyboard?

On PC keyboards, hold down the [Alt] key and press the 1, 2, and 6 keys in sequence on the numeric pad. That will generate a '~' character. Single inheritance may be easier to implement, but you are losing >something. It's a little concerning how often folks here take the >opinion that "Feature X has problems and I never use it anyway, so no >body else 'really' needs it." I'm not specificly blaming you, but i've >lost track of how many time if seen that reasoning tonight. I'm afraid >I'll see it a lot in the 275 I still have to read. Your reasoning has merit. The counterargument (and I've discussed this at length with my colleagues) is that C++ gives you a dozen ways and styles to do X. Programmers tend to develop specific styles and do things in certain ways. This leads to one programmer's use of C++ to be radically different than another's, almost to the point where they are different languages. C++ is a huge language, and C++ programmers tend to learn particular "islands" in the language and not be too familiar with the rest of it. Hence one idea behind D is to *reduce* the number of ways X can be accomplished, and reduce the balkanization of programmer expertise. Then, one programmer's coding style will look more like another's, with the intended result that legacy D code will be more maintainable. For example, over the years I've seen dozens of different ways that debug code was inserted into a program, all very different. D has one way - with the debug attribute/statement. C++ has a dozen string classes plus the native C way of doing strings. D has one way of doing strings. I intend to further help this along by writing a D style guide, "The D Way". There's a start on it all ready with the document on how to do error handling: www.digitalmars.com/d/errors.html -->

The D Style

The D Style is a set of style conventions for writing D programs. The D Style is not enforced by the compiler, it is purely cosmetic and a matter of choice. Adhering to the D Style, however, will make it easier for others to work with your D code and easier for you to work with others' D code. The D Style can form the starting point for a D project style guide customized for your project team.

White Space

One statement per line.
Two or more spaces per indentation level.
Operators are separated by single spaces from their operands.
Two blank lines separating function bodies.
One blank line separating variable declarations from statements in function bodies.

Comments

Use // comments to document a single line:

```
statement;  // comment
statement;  // comment
```

Use block comments to document a multiple line block of statements:

```
/*
 * comment
 * comment
 */
 statement;
 statement;
```

Use nesting comments to 'comment out' a piece of trial code:

```
/+++++
    /*
     * comment
     * comment
     */
     statement;
     statement;
 +++++/
```

Naming Conventions

General

Names formed by joining multiple works should have each word other than the first capitalized.

```
int myFunc();
```

Module
: Module names are all lower case. This avoids problems dealing with case insensitive file systems.

C Modules
: Modules that are interfaces to C functions go into the "c" package, for example:

```
import std.c.stdio;
```

: Module names should be all lower case.

Class, Struct, Union, Enum names
: are capitalized.

```
class Foo;
class FooAndBar;
```

Function names
: Function names are not capitalized.

```
int done();
int doneProcessing();
```

Const names
: Are in all caps.

Enum member names
: Are in all caps.

Meaningless Type Aliases

Things like:

```
alias void VOID;
alias int INT;
alias int* pint;
```

should be avoided.

Declaration Style

Since in D the declarations are left-associative, left justify them:

```
int[] x, y;     // makes it clear that x and y are the same type
int** p, q;     // makes it clear that p and q are the same type
```

to emphasize their relationship. Do not use the C style:

```
int []x, y;     // confusing since y is also an int[]
int **p, q;     // confusing since q is also an int**
```

Operator Overloading

Operator overloading is a powerful tool to extend the basic types supported by the language. But being powerful, it has great potential for creating obfuscated code. In particular, the existing D operators have conventional meanings, such as '+' means 'add' and '<<' means 'shift left'. Overloading operator '+' with a meaning different from 'add' is arbitrarily confusing and should be avoided.

Hungarian Notation

Just say no.

Example: wc

This program is the D version of the classic wc (wordcount) C program. It serves to demonstrate how to read files, slice arrays, and simple symbol table management with associative arrays.

```
import std.file;

int main (char[][] args)
{
    int w_total;
    int l_total;
    int c_total;

    printf ("   lines   words   bytes file\n");
    foreach (char[] arg; args[1 .. args.length])
    {
        char[] input;
        int w_cnt, l_cnt, c_cnt;
        int inword;

        input = cast(char[])std.file.read(arg);

        foreach (char c; input)
        {
            if (c == '\n')
                ++l_cnt;
            if (c != ' ')
```

```
            {
                if (!inword)
                {
                    inword = 1;
                    ++w_cnt;
                }
            }
            else
                inword = 0;
            ++c_cnt;
        }
        printf ("%8lu%8lu%8lu %.*s\n", l_cnt, w_cnt, c_cnt, arg);
        l_total += l_cnt;
        w_total += w_cnt;
        c_total += c_cnt;
    }
    if (args.length > 2)
    {
        printf ("--------------------------------------\n%8lu%8lu%8lu total",
            l_total, w_total, c_total);
    }
    return 0;
}
```

Compiler for D Programming Language

D for Win32
D for x86 Linux
general

Files Common to Win32 and Linux

`\dmd\src\phobos\`
: D runtime library source

`\dmd\src\dmd\`
: D compiler front end source under dual (GPL and Artistic) license

`\dmd\html\d\`
: Documentation

`\dmd\samples\d\`
: Sample D programs

Win32 D Compiler

Files

`\dmd\bin\dmd.exe`
: D compiler executable

`\dmd\bin\shell.exe`
: Simple command line shell

`\dmd\bin\sc.ini`
: Global compiler settings

`\dmd\lib\phobos.lib`
: D runtime library

Requirements

32 bit Windows operating system
D compiler for Win32
linker and utilities for Win32

Installation

Unzip the files in the root directory. It will create a \dmd directory with all the files in it. All the tools are command line tools, which means they are run from a console window. Create a console window in Windows XP by clicking on [Start][Command Prompt].

Example

Run:

```
\dmd\bin\shell all.sh
```

in the \dmd\samples\d directory for several small examples.

Compiler Arguments and Switches

dmd *files... -switch...*

files...

Extension	File Type

none	D source files
.d	D source files
.obj	Object files to link in
.exe	Name output executable file
.def	module definition file
.res	resource file

-c
compile only, do not link

-d
allow deprecated features

-debug
compile in debug code

-debug=*level*
compile in debug code <= *level*

-debug=*ident*
compile in debug code identified by *ident*

-g
add symbolic debug info

-gt
add trace profiling hooks

-inline
inline expand functions

-I*path*
where to look for imports. *path* is a ; separated list of paths. Multiple **-I**'s can be used, and the paths are searched in the same order.

-L*linkerflag*
pass *linkerflag* to the linker, for example, `/ma/li`

-O
optimize

-od*objdir*
write object files relative to directory *objdir* instead of to the current directory

-of*filename*
set output file name to *filename* in the output directory

-op
normally the path for **.d** source files is stripped off when generating an object file name. **-op** will leave it on.

-release
compile release version

-unittest
compile in unittest code

-v
verbose

-version=*level*

compile in version code >= *level*

-version=*ident*

compile in version code identified by *ident*

Linking

Linking is done directly by the **dmd** compiler after a successful compile. To prevent **dmd** from running the linker, use the **-c** switch.

The programs must be linked with the D runtime library **phobos.lib**, followed by the C runtime library **snn.lib**. This is done automatically as long as the directories for the libraries are on the LIB environment variable path. A typical way to set LIB would be:

```
set LIB=\dmd\lib;\dm\lib
```

Environment Variables

The D compiler dmd uses the following environment variables:

DFLAGS

The value of **DFLAGS** is treated as if it were appended to the command line to **dmd.exe**.

LIB

The linker uses LIB to search for library files. For D, it will normally be set to:

```
set LIB=\dmd\lib;\dm\lib
```

LINKCMD

dmd normally runs the linker by looking for **link.exe** along the **PATH**. To use a specific linker instead, set the **LINKCMD** environment variable to it. For example:

```
set LINKCMD=\dm\bin\link
```

PATH

If the linker is not found in the same directory as **dmd.exe** is in, the **PATH** is searched for it. **Note:** other linkers named **link.exe** will likely not work. Make sure the Digital Mars **link.exe** is found first in the **PATH** before other **link.exe**'s, or use **LINKCMD** to specifically identify which linker to use.

SC.INI Initialization File

dmd will look for the initialization file **sc.ini** in the same directory **dmd.exe** resides in. If found, environment variable settings in the file will override any existing settings. This is handy to make **dmd** independent of programs with conflicting use of environment variables.

Environment variables follow the `[Environment]` section heading, in name=value pairs. Comments are lines that start with ;. For example:

```
; sc.ini file for dmd
; Names enclosed by %% are searched for in the existing environment
; and inserted. The special name %@P% is replaced with the path
; to this file.
[Environment]
LIB="%@P%\..\lib";\dm\lib
DFLAGS="-I%@P%\..\src\phobos"
LINKCMD="%@P%\..\..\dm\bin"
```

Linux D Compiler

Files

`/dmd/bin/dmd`
: D compiler executable

`/dmd/bin/dumpobj`
: Elf file dumper

`/dmd/bin/obj2asm`
: Elf file disassembler

`/dmd/bin/dmd.conf`
: Global compiler settings (copy to `/etc/dmd.conf`)

`/dmd/lib/libphobos.a`
: D runtime library (copy to `/usr/lib/libphobos.a`)

Requirements

32 bit x86 Linux operating system
D compiler for Linux
Gnu C compiler (gcc)

Installation

1. Unzip the archive into your home directory. It will create a `~/dmd` directory with all the files in it. All the tools are command line tools, which means they are run from a console window.
2. Edit the file `~/dmd/bin/dmd.conf` to put the path in to where the phobos source files are.

3. Copy `dmd.conf` to `/etc`:

```
4.          cp dmd/bin/dmd.conf /etc
5.
```

6. Give execute permission to the following files:

```
7.          chmod u+x dmd/bin/dmd dmd/bin/obj2asm dmd/bin/dumpobj
8.
```

9. Put `dmd/bin` on your **PATH**, or copy the linux executables to `/usr/local/bin`
10. Copy the library to `/usr/lib`:

```
11.          cp dmd/lib/libphobos.a /usr/lib
12.
```

Compiler Arguments and Switches

dmd *files... -switch...*

files...

Extension	File Type
none	D source files
.d	D source files
.o	Object files to link in
.a	Library files to link in

-c
compile only, do not link

-d
allow deprecated features

-debug
compile in debug code

-debug=*level*
compile in debug code <= *level*

-debug=*ident*
compile in debug code identified by *ident*

-g
add symbolic debug info

-gt
add trace profiling hooks (not supported under linux)

-inline
inline expand functions

-I*path*

where to look for imports. *path* is a ; separated list of paths. Multiple **-I**'s can be used, and the paths are searched in the same order.

-L*linkerflag*
pass *linkerflag* to the linker, for example, `-M`

-O
optimize

-od*objdir*
write object files relative to directory *objdir* instead of to the current directory

-of*filename*
set output file name to *filename* in the output directory

-op
normally the path for **.d** source files is stripped off when generating an object file name. **-op** will leave it on.

-release
compile release version

-unittest
compile in unittest code

-v
verbose

-version=*level*
compile in version code >= *level*

-version=*ident*
compile in version code identified by *ident*

Linking

Linking is done directly by the **dmd** compiler after a successful compile. To prevent **dmd** from running the linker, use the **-c** switch.

The actual linking is done by running **gcc**. This ensures compatibility with modules compiled with **gcc**.

Environment Variables

The D compiler dmd uses the following environment variables:

DFLAGS
The value of **DFLAGS** is treated as if it were appended to the command line to **dmd**.

dmd.conf Initialization File

dmd will look for the initialization file **dmd.conf** in the directory `/etc`. If found, environment variable settings in the file will override any existing settings. This is handy to make **dmd** independent of programs with conflicting use of environment variables.

Environment variables follow the `[Environment]` section heading, in name=value pairs. Comments are lines that start with ;. For example:

```
; dmd.conf file for dmd
; Names enclosed by %% are searched for in the existing environment
; and inserted. The special name %@P% is replaced with the path
; to this file.
[Environment]
DFLAGS="-I%@P%\..\src\phobos"
```

Differences from Win32 version

String literals are read-only. Attempting to write to them will cause a segment violation.
The configuration file is `/etc/dmd.conf`

Linux Bugs

-g is not implemented, because I haven't figured out how to do it yet. **gdb** still works, though, at the global symbol level.
The code generator output has not been tuned yet, so it can be bloated.
Shared libraries cannot be generated.
The exception handling is not compatible with the way **g++** does it. I don't know if this is an issue or not.

General

Bugs

These are some of the major bugs:

The compiler sometimes gets the line number wrong on an error.
The phobos D runtime library is inadequate.
Need to write a tool to convert C .h files into D imports.
Array op= operations are not implemented.
In preconditions and out postconditions for member functions are not inherited.
It cannot be run from the IDDE.

Questions?

We welcome all feedback - kudos, flames, bugs, suggestions, hints, and most especially donated code! Join the fray in the D forum.

Feedback and Comments for this Page

Add feedback and comments regarding this page.

Future Directions

The following new features for D are planned, but the details have not been worked out:

1. Mixins.
2. Template inheritance.
3. Array literal expressions.

D Change Log

What's new for D 0.81
What's new for D 0.80
What's new for D 0.79
What's new for D 0.78
What's new for D 0.77
What's new for D 0.76
What's new for D 0.75
What's new for D 0.74
What's new for D 0.73
What's new for D 0.72
What's new for D 0.71
What's new for D 0.70
What's new for D 0.69
What's new for D 0.68
What's new for D 0.67
What's new for D 0.66
What's new for D 0.65
What's new for D 0.64
What's new for D 0.63
What's new for D 0.61
What's new for D 0.59
What's new for D 0.58
What's new for D 0.57
What's new for D 0.56
What's new for D 0.55
What's new for D 0.54
What's new for D 0.53
What's new for D 0.52
What's new for D 0.51
What's new for D 0.50
What's new for D 0.49
What's new for D 0.48
What's new for D 0.46
What's new for D 0.45
What's new for D 0.44
What's new for D 0.43
Download D compiler for Win32 and x86 linux
tech support

What's New for D 0.109

Dec 5, 2004

New/Changed Features

super.id and this.id now work if id does not need a 'this' pointer.

Bugs Fixed

Fixed constant folding problem with complex numbers.
Fixed class member initialize bug.
Fixed compiler gpf on bit sizeof.
Now correctly diagnoses some forward reference errors.
Fixed error recovery on minusing template as a variable.
Fixed internal error e2ir 1158
Fixed Assertion failure: '0' on line 661 in file 'expression.c'
Fixed synchronized property call omitting ().

What's New for D 0.108

Nov 30, 2004

Bugs Fixed

Fixed mango build break.

What's New for D 0.107

Nov 29, 2004

New/Changed Features

Improved speed of writef().
Improved single thread performance of gc allocation per Dave Fladebo's suggestions.
*InExpression*s now, instead of returning a **bit**, return a pointer to the associative array element if the key is present, **null** if it is not. This obviates the need for many double lookups.
.offset property is now deprecated, use **.offsetof** instead. This makes for better compatibility with C and fewer conflicts with `offset` as a field name.
Added **.ptr** property to arrays, which is handier and more typesafe than casting an array to a pointer.
Added Ben Hinkle's changes to std.stream:
adds EndianStream for BOM and endian support.
removes the two public imports std.string and std.file from end of file

adds read/write for ifloat, idouble, cfloat, cdouble and dchar to interfaces InputStream and OutputStream and add implementations to Stream and EndianStream
Added std.c.stddef for definition of C wchar_t.

Bugs Fixed

Fixed internal error e2ir 814
Fixed protection of implicit constructor.
Fixed deprecated attribute overriding static.
Tightened up detection of constants being implicitly converted to a type that cannot hold it.
Now detects duplicate case strings in switch statements.
Added support for switch(dchar[]) statements.
Fixed bug reading source files without B.O.M.
Fixed initialization of anonymous structs in classes.
Anonymous structs/unions can now only be a member of an aggregate.
Assert expressions are no longer evaluated for side effects even if asserts are turned off. It is not legal to depend on the side effect of an assert.
Fixed _init vs __init prefix for TypeInfo classes.
Adjusted arithmetic conversion rules to match C99.

What's New for D 0.106

Nov 9, 2004

New/Changed Features

Changed **std.c.time.CLOCKS_PER_SEC** from 1000 to 1000000 for linux.
Added version identifier **all** being always set.
Setting one of the list of predefined version identifiers is now an error.
Changed TypeInfoClass to TypeInfo_Class.
Changed TypeInfoTypedef to TypeInfo_Typedef.
Incorporated Matthew Wilson's changes to std.process.

Bugs Fixed

Fixed internal error s2ir 458
Fixed Assertion failure: 64 tocvdebug.c
Fixed codegen error with EAX parameter.
Version switch syntax checking fixed.
Fixed cast of constant to bit.
Multiple declarations of the same type alias now detected.
Access check now done for private constructor in new expression.

Anonymous structs/unions can now have non-field members.
Non-top level classes now added to module class list.
Fixed bugs in std.format with class objets, typedefs, %o and %b formats.
Fixed optimizer bug with loop invariants.
Fixed synchronized statements on Linux.

What's New for D 0.105

Oct 28, 2004

New/Changed Features

Changed integral literal type determination to match C99 6.4.4.1.

Bugs Fixed

Incorporated David Friedman's fixes for static initializers.
Incorporated David Friedman's fixes for bit array duping and array appending.
Incorporated David Friedman's fix for cast of integral constant to bit.
Incorporated Stewart Gordon's fix for switcherr.d and array.d.
Fixed several bugs with dmd command line parsing.
Fixed Assertion failure: 'ei' line 2576 file 'mtype.c'
Fixed Ivan Senji's two compiler crashes.
Fixed Internal error: ..\ztc\cod3.c 736
Fixed diagnostic on forward referenced templates.
Fixed diagnostic on missing identifiers in foreach declarations.
Fixed bug with mixin constructors.
Internal use of "_invariant" changed to "__invariant" to avoid conflicts with user code.
Fixed dmd hang on transcoding some utf strings.
Interfaces used as an argument to synchronized statements are now automatically cast to Object.

What's New for D 0.104

Oct 21, 2004

Bugs Fixed

Fixed linker bug with unknown CV version.

What's New for D 0.103

Oct 20, 2004

New/Changed Features

Improved symbolic debug info generation for Win32.
Added Dwarf2 line number debug info for linux.
Added Dave Fladebo's speedups to associative arrays.

Bugs Fixed

Removed redundant declaration of MAP_FAILED from Phobos.
Fixed codegen scheduling bug.
Incorporated Dave Fladebo's linux phobos makefile improvements.
Incorporated David Friedman's std.thread pause fix.

What's New for D 0.102

Sep 20, 2004

New/Changed Features

Upgraded etc.c.zlib to 1.2.1 (thanks to Sean Kelly).
Improved performance of array append (thanks to Dave Fladebo).
Added \dmd\bin\make.exe.
Improved Phobos makefiles.

Bugs Fixed

Fixed private module construction order.
Folded in Dave Fladebo's fix to std.zlib.
Fixed array append seg fault under linux.
Fixed problem where COMDATs could not exist in multiple modules.
Fixed seg fault on const fields as lvalues.
Fixed Internal error: ../ztc/cod1.c 2498
Fixed Internal error: e2ir.c 447
Fixed seg fault with empty debug statement.
Fixed seg fault with undefined qualified types.
Fixed error recovery with void in arithmetic expressions.
Fixed assertion failure 1147 in expression.c.
Fixed Internal error: ../ztc/cod2.c 4207

Eliminated trailing 0 in returned array from std.file.getcwd()
Fixed "cd" command in make.exe.

What's New for D 0.101

Aug 30, 2004

New/Changed Features

Added Ben Hinkle's improvements to std.stream. Ben writes: "I've attached a modified version of std.stream and phobos.html that include the bug fixes and behavior changes outlined in the newsgroup post yesterday. The bug fixes are almost entirely backwards compatible. The biggest difference is the change that opening a file on Windows with FileMode.In errors when the file doesn't exist. That makes it the same behavior as Linux and it just seems to make sense. I also did things like added writef and changed the Error classes to Exception."
Kris pointed out that version identifiers have global effect, and are dependent on which order modules are imported. This is clearly wrong. Now, only version and debug definitions on the command line have global effect. Others only influence the module they are declared in. Furthermore, version and debug definitions can only be done at module scope.

Bugs Fixed

Fixed "super" bug introduced by 0.99.
Missing function body after 'in' now diagnosed.
Function literals can now have same type signature.
Fixed problem with 'out' parameters in nested functions.
Fixed array initialization causing seg fault.
Fixed compiler gpf on invalid enum syntax.
Fixed Internal error: ..\ztc\cod1.c 3251
Fixed FPU stack overflow bug.
Fixed float optimization bug.
Fixed bug in conversion of string literals to UTF-16.

What's New for D 0.100

Aug 20, 2004

Bugs Fixed

Fixed Internal error: ..\ztc\cgcs.c 213 introduced by 0.99.

Hopefully fixed linux seg fault ENTER problem.

What's New for D 0.99

Aug 19, 2004

New/Changed Features

Added std.string functions ifind() and irfind() (thanks to David L. Davis).
Slicing of bit arrays now allowed if lower bound lands on a byte boundary.
Bit pointer offsets now are in number of bits, however, they must be a multiple of 8.
Improved performance of stack array initialization.
Idiom of using std.c.stdlib.alloca() with a constant value now recognized.
Removed implicit cast to void. What was I thinking?
Moved setErrno() from std.c.linux.linux to std.c.stdlib.
Added setErrno() to std.c.stdlib.
Improved code gen on cast of 'this' to interface.
Added implicit declaration of **length** to index and slice expressions. This will **break existing code** if length is used as a variable within [ ].

Bugs Fixed

Some overloading errors fixed.
~ now works on bit operands.
Copy of non-byte aligned upper bound of bit array copies no longer copies too many bits.
Fixed cast of bit to byte.
Fixed MI bug with interface vtbl dispatch.
Fixed Internal error: ..\ztc\el.c 723
Fixed gpf on forward reference of mixin.
Fixed Internal error: ..\ztc\dt.c 104
Fixed Internal error: ..\ztc\cgcod.c 552
Fixed error recovery on bad initializers.
Diagnoses error on arithmetic on class references.
Fixed Internal error: ..\ztc\cgcod.c 1464
Fixed internal error: ..\ztc\cgcs.c 353
Fixed unresolved external when expanding nested templates in imports.
Fixed mysterious TOK881 message.
Fixed compiler gpf with using const char[] as enum value.
Fixed compiler gpf with error recovery.

What's New for D 0.98

Aug 5, 2004

New/Changed Features

One can now 'new' a scalar type, for example:

```
int* p = new int;
```

Bugs Fixed

Fixed bug where typedef/alias members of a class weren't found when looked up via an instance of that class.
Added Berin Loritsch's fixes to pi.d.
Better detection of recursive template instantiation errors.
Improved error message for implicit conversion errors.
Fixed gpf on erroneous template instantiations.
Fixed name mangling of D __import__ symbols from DLLs.
Fixed Assertion failure: 267 'declaration.c'
Fixed Internal error: ..\ztc\cgcod.c 1464
Fixed Assertion failure: 1204 'expression.c'
Fixed some error recovery for undefined identifiers.
Fixed Internal error: ..\ztc\cgcs.c 350
Fixed Internal error: ..\ztc\cod1.c 2244
Access checking now done after overload resolution rather than before.
Fixed gpf on static initializers for function local structs.
Fixed error recovery on undefined identifiers.
Fixed gpf on const arrays without initializers.
Fixed Assertion failure: 3154 'mtype.c'
Fixed Assertion failure: 1203 'func.c'
Fixed Internal error: ..\ztc\cg87.c 1968

What's New for D 0.97

Jul 26, 2004

New/Changed Features

Added missing overload (opPos) for unary plus.
B.O.M. no longer required for UTF-16 and UTF-32 source text (thanks to Arcane Jill).

Bugs Fixed

Fixed linux bug where sometimes global symbols were not made public in .o file.
Fixed linux bug with function returns of small structs from C functions.
Implemented listdir() for linux (thanks to Christopher E. Miller and Andre Fornacon)
Deprecated functions can now call other deprecated functions without causing an error.
Check for deprecated functions now done after overloading rather than before.
Fixed some forward reference problems with interfaces.
Fixed compiler GPF with va_arg.
Fixes for std.system (thanks to Thomas Kuehne)
Fixed method alias ordering conflict in derived class.
Fixed std.utf bugs (thanks to Stewart Gordon)

What's New for D 0.96

Jul 21, 2004

New/Changed Features

Added exec* functions to std.process. (thanks to Matthew Wilson)
Added std.perf (thanks to Matthew Wilson)

Bugs Fixed

Fixed compiler crash on circular class derivations.
Invalid UTF characters in string literals now diagnosed.
Date parsing can now handle time zone offsets that are not in hourly increments.
Fixed incorrect handling of precision for strings in std.format.
Fixed auto bug inside template bodies.
Fixed static initializations of the form int[2]x=3;
Added some missing line numbers in error messages.
Fixed an alias resolution problem.
Break/continue in try and syncrhonized blocks now matches documentation.
Function out parameters are now initialized at the start of the function.
Fixed mishandling of out parameters in variadic functions.
Fixed problem of undefined symbols at link time when templates are expanded in interface declarations.
Template default arguments for parameters now are 'lazilly' semantically analyzed, which means they can refer to previous template argument types.
Fixed DMD GPF when trying to index a mixin with [].
time_t definition removed from std.c.linux.linux, as it conflicted with definition in std.c.time.

What's New for D 0.95

Jul 6, 2004

New/Changed Features

Added printf replacement, std.stdio.writef and related functions.
Added std.format for formatting strings.
Rewrote std.string.format() for full functionality.

Bugs Fixed

Fixed Internal error: ..\ztc\cgcod.c 614
Fixed overflow detection on static array dimension.
Fixed Internal error: ..\ztc\cod4.c 352
Now detects attempts to use non-identifer character filenames as module names.
Can now overload static member functions with non-static ones within a static one.
Now detects attempts to overload destructors.
Fixed compiler exit status under linux (thanks to Sam McCall).
Fixed poor error recover from undefined template identifier.
Better forward reference handling.
Fixed problems with local statics having name collisions.

What's New for D 0.94

Jun 27, 2004

Bugs Fixed

Missing TypeInfo for bit added.
Incorrect integral type promotion for variadic arguments fixed.
Fixed std.string.toStringz() 0 termination bug.
Fixed some .size => .sizeof deprecations in library.
Fixed std.stream.File.open() and std.stream.File.create() to handle UTF filenames (thanks to Carlos Santander Bernal).
Fixed std.uri.encode() bug with Octets.
Fixed codegen error with typedef'd static arrays.
Correct error message now generated when using an array initializer as an expression.
Fixed "frame" error when compiling.
Fixed compiler GPF on getting boolean value of module name.
Fixed compiler GPF with function with all default arguments.
Fixed bug with associative array typedef'd indices.
Fixed TypeInfo crash with arrays of null Objects.

Fixed Internal error: ..\ztc\blockopt.c 1941
Fixed Internal error: ..\ztc\cod1.c 2244
Fixed Assertion failure: '0' on line 80 in file 'mtype.c'

What's New for D 0.93

Jun 22, 2004

New/Changed Features

char.init is now 0xFF, wchar.init is 0xFFFF, dchar.init is 0x0000FFFF. Thanks to Arcane Jill.
Array operator assignment overloading with opIndex(index, value) is now deprecated. Use opIndexAssign(value, index) instead.
Multiple indices are now allowed with array operator overloading.
Added **_arguments[]** and **_argptr** to variadic functions.
Added *TypeidExpression*s.
.typeinfo property is now deprecated (use *TypeidExpression* instead).
Replaced **std.c.stdarg** with **std.stdarg**.

Bugs Fixed

Some minor error message improvements.
Fixed bug where local class static constructor did not get called.
Fixed Assertion failure: 'f' on line 2695 in file 'expression.c'
Fixed Internal error: ..\ztc\cgcs.c 353
Fixed dmd GPF when module used as array.

What's New for D 0.92

Jun 7, 2004

New/Changed Features

Modified Object.toString() and Object.print() to print out the name of the object instance's class.
The *Expression* within an array's brackets is now an *AssignExpression* (meaning that commas are no longer allowed).
Added default arguments to function parameters. Semantics are like C++.
Added **package** attribute for package level access.
Added operator overloads opAdd_r, opMul_r, opAnd_r, opOr_r, and opXor_r.

Modified operator overloading rules so functionality can be added to the lvalue of an operator overload for user defined rvalues without needing to modify the lvalue's class.

Bugs Fixed

Fixed parsing bug in typeof(this).func();
Fixed nested multiple inheritance bug with interfaces.
Fixed compare() bug in typeinfo's for int and uint. Thanks to Stewart Gordon.

What's New for D 0.91

May 27, 2004

New/Changed Features

Allow functions that return void to return expressions.
Added support for **typeof(this)** and **typeof(super)**. outside of non-static member functions.
Added cast operator overloading.
typeof(this).member() now does non-virtual call to member().
Mixin qualifier names now work for class members.

Bugs Fixed

Fixed std.file.listdir unicode issue with Win95.
Fixed incorrect diamond inheritance of interface classes.
std.string.capwords() now converts non-first characters of a word to lower case.
Fixed problem with **super** in mixins.
Fixed assert failure mtype.c 2575
Fixed bad formatting of typedef* error messages.
Fixed std.string.toString(char).
Fixed bug accessing private static from inlined function.
Fixed compiler gpf on synchronized blank for statement inits.

What's New for D 0.90

May 20, 2004

Bugs Fixed

Fixed problem with mixins overriding same function more than once.

Added ability for invariants and other special functions to be in mixins.
Fixed error diagnostic for delegate literals in class declarations.
Fixed extended comparison operators so they work on integral types when constant folding.
Fixed bug with mixin functions for interfaces.
Fixed Internal error: ..\ztc\cod1.c 1641
Fixed name lookups for special functions in mixins.

What's New for D 0.89

May 17, 2004

New/Changed Features

Allow void() to be omitted from function literals.
Mixins added.
A *WithStatement* now works on struct instances.

Bugs Fixed

Added Stewart Gordon's windows.d function pointer fixes.
Added Stewart Gordon's std.asserterror fixes.
Added Ben Hinkle's std.stream fixes.
Fixed bug with template function alias arguments.

What's New for D 0.88

May 5, 2004

New/Changed Features

Added **std.c.stdarg** (thanks to Hauke Duden).
C style casts deprecated (use **-d** to compile them for now).
instance style template instantiation deprecated (use **-d** to compile them for now).

Bugs Fixed

Fixed problem with template classes and opCall.
Fixed problems with implicit this and member template functions.
Fixed problem with property in [] of *NewExp*.
Now correctly diagnoses problem with using local as template parameter alias.

Fixed bug with no identifier for declarator in template type.
Fixed const folding with ?: operator.
Fixed std.utf.toUTF32() with J C Calvarese's fix.
Included David Friedman's fixes for linux monitors.
Fixed bug with overloading alias template parameters.
Fixed internal error cg87 1235.
Superclass dtors now called for class deallocator.
Fixed _d_delmemory() taking the wrong argument.
Fixed auto super dtors.
Fixed multi argument struct new's with custom new.
Fixed custom deallocator struct delete.
Monitors now deleted when object is deleted.

What's New for D 0.86

Apr 23, 2004

New/Changed Features

Added template default parameters.
Added casting ability to non-COM interfaces.
Added Christopher E. Miller's std.socket and std.socketstream.
Added std.mmfile (thanks to Matthew Wilson!)

Bugs Fixed

An error is now issued when the argument to `delete` is a COM interface object.
Incorporated Antonio Monteiro's fixes for std.date and std.zip.
Error now diagnosed when *EnumBaseType* is not integral.
Fixed delete on non-class objects.

What's New for D 0.82

Mar 28, 2004

Bugs Fixed

Some cases of forward referenced classes are now handled.
Illegal recursive alias declarations now correctly diagnosed.
Erroneous anonymous struct member offsets within unions fixed.
Fixed problem with array initializers for consts.

Fixed bug with intrinsics.
Can now slice a void[].
std.date: fixed dst calculation for zones with no dst.
Fixed bug with nested delegate literals.
Fixed overload inheritance bug with function aliases from base classes.
Added files gnuc.h, mem.h, mem.c, port.h, enum.h, and enum.c to the \dmd\src\dmd.
Fixed seg fault on forward reference to template.
Fixed std.date.getUTCtime() for linux.

What's New for D 0.81

Mar 7, 2004

Bugs Fixed

Fixed problem with class and modules having the same name.
Fixed access problem with protected members of base class.

What's New for D 0.80

Mar 5, 2004

New/Changed Features

Added overloads for basic types for std.string.toString().
Modified front end source to be more compatible with GDMD project.
Implicit conversions of floating point values to integer values is now disallowed.

Bugs Fixed

Fixed dmd crash with module and class name conflict.
Fixed bug where f(x)(y); was interpreted as declaring x as a function taking parameter type y and returning type f, instead of f being an instance of a class with () overloaded. There's still the problem of f(*p)(y), though, the solution is to probably just deprecate the C function pointer declaration syntax.
Fixed using **super** in nested function.
Function members without bodies in non-abstract classes now reference an external symbol rather than inserting a 0 in the vtbl[].
Fixed Internal error: ../ztc/cgcod.c 1459
Fixed problems with template aliases.
Improved semantic analysis with member templates.

Adding some missing file/line numbers to error messages.
Fixed bug with property syntax in return expression.
Fixed problem with template expansion contexts.

What's New for D 0.79

Feb 2, 2004

New/Changed Features

Added utf decoding capability to foreach statement.
Added Christopher Miller's std.base64.
Fixed gc per http://www.digitalmars.com/drn-bin/wwwnews?D/21217

Bugs Fixed

Fixed std.file failure on Win95 (hopefully got it right this time!).
Fixed code gen error with unsigned % 10.
Fixed problem with std.regexp.test and multiple calls.
Fixed Internal error: ..\ztc\cgcod.c 2241
Fixed chained assignments to bit arrays.
Fixed bugs with foreach over a bit array.
Fixed many problems with recursive templates.
Fixed Assertion failure on line 67 in file 'template.c'
Fixed problem with optimization of short registers.

What's New for D 0.78

Jan 14, 2004

New/Changed Features

std.string.find() and .rfind() upgraded to support dchar searches within char[] strings.
Fixed std.ctype functions to accept/return dchar types instead of char types.
// comments can now be terminated by end of file rather than a newline.

Bugs Fixed

Fixed Internal error e2ir.c 133
Fixed Internal error: ..\ztc\cod1.c 2240
Fixed Internal error: ..\ztc\cod2.c 4116

Fixed crash on invalid template syntax.
std.file functions now fall back to "A" functions on Win9x systems.
Fixed problem with static arrays of structs with non-zero default initializations.
Fixed crasher with structs declared but not defined.
Fixed critical section bug in Win95.

What's New for D 0.77

Jan 2, 2004

New/Changed Features

Changed std.file so unicode filenames work.
Added toUTF16z to std.utf.
Added std.md5.
Hex strings must now contain an even number of hex digits.
Added **.alignof** property.
Added **.sizeof** property which has same value as **.size**, but won't conflict with C/C++ struct member names.
Global main must be main() or main(char[][] args).
Added size_t, ptrdiff_t predefined aliases. Use size_t as an alias for an unsigned integral type spanning the address space, and ptrdiff_t as an alias for a signed integral type spanning the address space. This will make the code portable from 32 to 64 bits.
.length, .size, .sizeof, and .alignof now return a value of type size_t.
For win32, converts '/' in source file names passed to dmd to '\' so that back end routines work consistently.
Added typeof.
Added pragmas.
Added expression lists to case statements.
Added `goto default;` and `goto case;` and `goto case Expression;` for use in switch statements.
Added template alias parameters.
Added new template instance syntax.
Added class template syntax.
Added template member injection rule.
Added Matthew Wilson's std.recls - for recursive file-system searching.

Bugs Fixed

Better error recovery in parser.
Sizes of bit arrays fixed.
Fixed conflict between **std.intrinsic** and **std.math**.
Fixed reset problem with **std.regexp.match**().

Fixed **std.string.rfind**().
Fixed several problems with **-inline**.

What's New for D 0.76

Nov 21, 2003

New/Changed Features

std.assert changed to std.asserterror.
Added keyword **is**, which has same function as ===.
Renamed operator overloading names.
Renamed std.assert to std.asserterror.
Added **system** to std.process.
Changes to std.file:
Added several functions to std.file.
Changed byte[] types for data in std.file to void[].
std.file.getSize() now returns a ulong.

Bugs Fixed

Library modules are all correctly named.
Library modules now have private imports.

What's New for D 0.75

Nov 4, 2003

New/Changed Features

Changed to new standard library package layout.
Changed **apply** to **opApply**.
Fixed foreach so it can have multiple parameters.
Added etc.c.zlib, a D interface to the C zlib compression library.
Added std.zlib, a compression module.
Added std.zip, a zip archive module.

What's New for D 0.74

Oct 15, 2003

New/Changed Features

Added support for Unicode characters in identifiers.
Version declarations can now wrap attribute : and they'll stick.
-of*filename* switch now accepts any extension.
Added D.win32.registry.

Bugs Fixed

Problems fixed with with abstract member function definitions.
Fixed problems with getting the wrong .classinfo with inherited classes.
Fixed internal error cgcod.c 1459
Fixed internal error e2ir.c 721
Fixed ICE for empty foreach bodies.
Fixed name mangling for out/inout parameters to extern (Windows) functions.
Fixed bug in ?: of string literals.
Fixed order-of-evaluation bug with &((new foo()).bar)
Can now assign null to delegates.
Fixed problem diagnosing error of slices as out parameters.
Fixed problem diagnosing error of const field initializations.
Fixed gc and multithread deadlock (thanks to jhenzie).

What's New for D 0.73

Sep 18, 2003

New/Changed Features

Added static asserts.
Added **bswap** as a compiler intrinsic function.
Added operator overloading for array index [], array slice [..] and function call () operators.
Added properties.
To take the address of a function, the & operator is required now, rather than being implicit.

Bugs Fixed

Implicit conversions of B[] to A[] are no longer allowed if B is derived from A. Gaping type safety holes are the reason.

What's New for D 0.72

Sep 14, 2003

New/Changed Features

Implicit conversions of B[] to A[] are now allowed if B is derived from A.
Functions in abstract classes can now have function bodies.

Bugs Fixed

Fixed bug with **in** and **out** instructions in inline assembler.
Fixed speed problem with `%.*s` printf format.
Fixed problem with **foreach** over array of arrays or structs.
Fixed compiler error with array **rehash**.
Now correctly issues error on self-initializations like:
```
int a = a;
```
Fixed problem converting "string" to char[], it should be an exact conversion, not an implicit conversion.

Linux Bugs Fixed

Occaisional segfault during gc collection fixed.
Empty static arrays now placed in BSS segment.
Conversion of uint to real now works.

What's New for D 0.71

Sep 3, 2003

New/Changed Features

Added foreach statement.

Bugs Fixed

Fixed bug with nested functions.
Fixed bug with linux file.write() and file.append() functions.
Fixed bug with linux processing of /etc/dmd.conf.

What's New for D 0.70

Aug 24, 2003

New/Changed Features

Added wprintf() to object.d.
Extended alias to work for overloaded functions from another scope.

Bugs Fixed

Fixed incorrect handling of leading UTF-8 BOM mark.
Fixed 2 compiler GPF's.
Fixed problem with downcasting to interfaces.
Fixed linux seg fault with %@P% in DFLAGS.
Fixed inoperative -L switch for linux.

What's New for D 0.69

Aug 11, 2003

New/Changed Features

Added **dchar** keyword for UTF-32 characters.
' ' strings are now character literals, not strings.
wysiwyg strings are not r"*string*", not '*string*'.
`string` (backquotes) are also wysiwyg strings.
Added x"0a AA BF" style hex strings.
Implicit conversion of single character strings to character literals no longer happens.
Implicit conversions of string literals between UTF-8, UTF-16 and UTF-32 now happens.
Deleted command line switch **-o**, replaced it with **-od**, **-of** and **-op**. This should remove confusion and add flexibility.
Added **bool** as an alias for **bit**.
Integer and floating point literals can now have embedded _ for formatting purposes.

Bugs Fixed

Bit variables can now be **out** or **inout** function parameters.
Package names now part of mangled names.
Mangled names are now reversible unambiguously.
Fixed problem returning 8 byte structs from functions.
Fixed bug with typedef'd associative arrays.
Fixed bug with typedef'd return values.

Fixed error recovery from 'typedef struct'.
Fixed problems with Linux exception handling.
0b... numeric literals now work under Linux.

What's New for D 0.68

Jul 8, 2003

Added TypeInfo's for classes and other basic types.
new'ing arrays of types with non-zero initializers now works.
new'ing bit arrays now works.
new'ing structs now works.
Fixed bug with typedef'd associative array index.
Falling off the end of a function that has a non-void return type now generates a runtime exception if a syntax error is not already generated.
Fixed parse bug with inout keyword in nested functions.
Fixed cgcs 350 error.
Fixed error where const objects could be assigned to.
Fixed crash with non-existent struct member names in initializers.
Fixed crash with const objects with no initializers.
Fixed cgobj 3115 error.
Correctly errors out now when slicing voids.

What's New for D 0.67

Jun 17, 2003

Fixed some bugs.

What's New for D 0.66

Jun 8, 2003

Added uri module.
Added utf module.
Added constant folding for ?: operator.
Zero initialized structs no longer need to link to the module the struct was defined in.
Private imports now supported.
Class names common to multiple modules no longer collide, but this required a change to how class names are mangled. Hence, existing libraries will need to be recompiled.
templates of the form:

```
template foo(T:T[])
```

now correctly resolve T in foo(int[]) to int rather than int[].

What's New for D 0.65

May 13, 2003

linux version

Another bad section name bug is hopefully fixed.

What's New for D 0.64

May 12, 2003

linux version

`dmd.conf` now needs to be installed as `/etc/dmd.conf`.
`phobos.a` has been renamed as `libphobos.a` and been placed in the `/usr/lib` directory.
dmd will now do the link step automatically.
The bad section name bug is hopefully fixed.

What's New for D 0.63

May 10, 2003

Added linux version. Thanks to Burton Radons and Pat Nelson for the help.

What's New for D 0.61

Mar 30, 2003

Incorporated Burton Radon's stream.d fixes.
Improved performance of string switches and string concatenation.
Phobos library is now built optimized.
Fixed numerous reported bugs.

What's New for D 0.59

Mar 6, 2003

Fixed bugs in nested functions.
Refactored code so it compiles with gnuc.

What's New for D 0.58

Mar 3, 2003

Added covariant function return types.
Fixed several bugs in nested functions, template argument deduction, access protections, constant folding, etc.

What's New for D 0.57

Feb 25, 2003

Added keyword **function** and new function pointer syntax analogous to the way the delegate declaration syntax works.
Added function literals.
Added nested functions.
Added closures.
Fixed struct/class definitions nested within functions.
Many thanks to Burton Radons for his help with this.

What's New for D 0.56

Feb 20, 2003

Fixed a couple bad bugs with template typedefs and arrays.

What's New for D 0.55

Feb 17, 2003

Added several new math functions to math.d.
Changed behavior of new expressions to match change in array declaration order:

```
char[][] foo;
foo = new char[][45];   // new, correct way to allocate array of
45 strings
//foo = new char[45][]; // old, now wrong, way
```

Fixed some problems with error recovery.
The module name is now in its own scope enclosing the scope of the contents of the module (before it was simply inaccessible).

What's New for D 0.54

Feb 14, 2003

Fixed some parsing problems with template instances.
Introduced new types **cfloat**, **cdouble**, **ifloat**, **idouble**.
Introduced new constant types **ifloat**, **idouble**.
Renamed **extended** to **real**.
Renamed **imaginary** to **ireal**.
Renamed **complex** to **creal**.
Reversed order of array declarations when they appear to the left of the identifier being declared. Postfix array declaration syntax remains the same (and equivalent to C).

```
int[3][4] a;    // a is 4 arrays of 3 arrays of ints
int b[4][3];    // b is 4 arrays of 3 arrays of ints
```

These changes have been needed for a while, and it's time to put them in before D gets too constrained by legacy code. Fortunately, using grep and global/search/replace can easily take care of the type renames. The array declaration issue can be fixed by grepping for "][", and then manually fixing each.

What's New for D 0.53

Feb 8, 2003

Added ability for explicit memory allocation/deallocation.

What's New for D 0.52

Feb 5, 2003

The argument to a with statement can now be a template instance.
The inline asm for `FCOMI/FCOMIP/FUCOMI/FUCOMIP` can now accept the `ST,ST(i)` form of the instruction to match the Intel documentation.
Fixed numerous minor bugs.

What's New for D 0.51

Jan 27, 2003

Added template value parameters (as opposed to just type parameters).
Fixed several problems with templates.
Added **#line** pragma support.
D can now accept source files in various UTF formats.

What's New for D 0.50

Nov 20, 2002

To convert to type **bit** now requires an explicit cast, rather than implicit. The conversion `(cast(bit)i)` is now performed as `(i?true:false)`.
Added library functions string.toString().
Fixed many template bugs.
Changed implicit conversion rules for integral types; implicit conversions are not allowed if the value would change. For example:

```
byte  b = 0x10;         // ok
ubyte c = 0x100;        // error
byte  d = 0x80;         // error
ubyte e = 0x80;         // ok
```

What's New for D 0.49

Nov 18, 2002

Added **LINKCMD** to sc.ini to specify which linker to use.
Multiple **-I** switches can now be used, and the paths are searched in order.
Fixed bug in regexp with blank substitutions.
Removed "reference to this before super()" error.
% and %= floating point operations are now handled by code generator instead of library.
Fixed GCStats linking problem.
Fixed many other minor problems.

What's New for D 0.48

Oct 25, 2002

Added conv to phobos library.
Fixed a number of bugs that were blocking people.

What's New for D 0.46

Oct 22, 2002

Fixed problem with auto class constructors.
Fixed problem with calling class invariants with -O.
Redid lib.exe command syntax to simplify makefiles.
String literals can now span multiple lines.
Fixed bugs in handling access violations and other windows exceptions.
Revamped support for generating Windows GUI apps.
Fixed some code gen bugs.

What's New for D 0.45

Oct 8, 2002

Redid constructor semantics to improve reliability.
Expanded inlining capability.
Redid interface semantics.
Fixed problems with ~ string concatenation.

What's New for D 0.44

Oct 1, 2002

Clarified interface semantics and fixed problems with it per Joe Battelle's suggestions.

What's New for D 0.43

Sep 28, 2002

Added new *VolatileStatement*.
Added **auto** storage class and auto classes to implement RAII. See here and here.

Acknowledgements

The following people have contributed to the D language project; with ideas, code, expertise, marketing, inspiration and moral support.

Kris Bell, Hauke Duden, Bruce Eckel, Eric Engstrom, Dave Fladebo, David Friedman, Stewart Gordon, Ben Hinkle, Jan Knepper, Helmut Leitner, Lubomir Litchev, Christopher E. Miller,

Pavel Minayev, Antonio Monteiro, Paul Nash, Pat Nelson, Burton Radons, Tim Rentsch, Fabio Riccardi, Bob Taniguchi, John Whited, Matthew Wilson, Peter Zatloukal

---

D vs Other Languages

This table is a quick and rough comparison of various features of **D** with other languages it is frequently compared with. While many capabilities are available with standard libraries, this table is for features built in to the core language itself. Only official standardized features are considered, not proposed features, betas, or extensions. And, like all language comparisons, it is biased in terms of what features are mentioned, omitted, and my interpretation of those features.

Feature	D	C	C++	C#	Java
Garbage Collection	Yes	No	No	Yes	Yes
Functions					
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
Arrays					
Lightweight arrays	Yes	Yes	Yes	No	No
Resizeable arrays	Yes	No	No	No	No
Arrays of bits	Yes	No	No	No	No
Built-in strings	Yes	No	No	Yes	Yes
Array slicing	Yes	No	No	No	No
Array bounds checking	Yes	No	No	Yes	Yes
Associative arrays	Yes	No	No	No	No
Strong typedefs	Yes	No	No	No	No
String switches	Yes	No	No	Yes	No
Aliases	Yes	Yes	Yes	No	No
OOP					
Object Oriented	Yes	No	Yes	Yes	Yes

Feature	D	C	C++	C#	Java
Multiple Inheritance	No	No	Yes	No	No
Interfaces	Yes	No	Yes	Yes	Yes
Operator overloading	Yes	No	Yes	Yes	No
Modules	Yes	No	Yes	Yes	Yes
Dynamic class loading	No	No	No	Yes	Yes
Nested classes	Yes	Yes	Yes	Yes	Yes
Inner (adaptor) classes	No	No	No	No	Yes
Covariant return types	Yes	No	Yes	No	No
Properties	Yes	No	No	Yes	No
Performance					
Inline assembler	Yes	Yes	Yes	No	No
Direct access to hardware	Yes	Yes	Yes	No	No
Lightweight objects	Yes	Yes	Yes	Yes	No
Explicit memory allocation control	Yes	Yes	Yes	No	No
Independent of VM	Yes	Yes	Yes	No	No
Direct native code gen	Yes	Yes	Yes	No	No
Generic Programming					
Templates	Yes	No	Yes	No	No
Mixins	Yes	No	No	No	No
typeof	Yes	No	No	Yes	No
foreach	Yes	No	No	Yes	No
Constraints	Yes	No	No	No	No
Reliability					
Contract Programming	Yes	No	No	No	No
Unit testing	Yes	No	No	No	No
Static construction order	Yes	No	No	Yes	Yes
Guaranteed initialization	Yes	No	No	Yes	Yes
RAII (automatic destructors)	Yes	No	Yes	Yes	No
Exception handling	Yes	No	Yes	Yes	Yes

Feature	D	C	C++	C#	Java
try-catch-finally blocks	Yes	No	No	Yes	Yes
Thread synchronization primitives	Yes	No	No	Yes	Yes
Compatibility					
C-style syntax	Yes	Yes	Yes	Yes	Yes
Enumerated types	Yes	Yes	Yes	Yes	No
Support all C types	Yes	Yes	No	No	No
80 bit floating point	Yes	Yes	Yes	No	No
Complex and Imaginary	Yes	Yes	No	No	No
Direct access to C	Yes	Yes	Yes	No	No
Use existing debuggers	Yes	Yes	Yes	No	No
Struct member alignment control	Yes	No	No	No	No
Generates standard object files	Yes	Yes	Yes	No	No
Macro text preprocessor	No	Yes	Yes	No	No
Other					
Conditional compilation	Yes	Yes	Yes	Yes	No
Unicode source text	Yes	Yes	Yes	Yes	Yes

Notes

C Language Specification
: ANSI/ISO/IEC 9899-1999 (a.k.a. C99)

C++ Language Specification
: ISO/IEC 14882-1998 (a.k.a. C++98)

C# Language Specification
: ECMA-334

Java Language Specification
: "The Java Language Specification" by Gosling, Joy, and Steele, Addison-Wesley, ISBN 0-201-63451-1

Object Oriented
: This means support for classes, member functions, inheritance, and virtual function

dispatch.

Inline assembler
: Many C and C++ compilers support an inline assembler, but this is not a standard part of the language, and implementations vary widely in syntax and quality.

Interfaces
: Support in C++ for interfaces is weak enough that an IDL (Interface Description Language) was invented to compensate.

Modules
: Many correctly argue that C++ doesn't really have modules. But C++ namespaces coupled with header files share many features with modules.

Garbage Collection
: The Hans-Boehm garbage collector can be successfully used with C and C++, but it is not a standard part of the language.

Contract Programming
: The Digital Mars C++ compiler supports Contract Programming as an extension. Compare some C++ techniques for doing Contract Programming with D.

Resizeable arrays
: Part of the standard library for C++ implements resizeable arrays, however, they are not part of the core language. A conforming freestanding implementation of C++ (C++98 17.4.1.3) does not need to provide these libraries.

Built-in Strings
: Part of the standard library for C++ implements strings, however, they are not part of the core language. A conforming freestanding implementation of C++ (C++98 17.4.1.3) does not need to provide these libraries. Here's a comparison of C++ strings and D built-in strings.

Strong typedefs
: Strong typedefs can be emulated in C/C++ by wrapping a type in a struct. Getting this to work right requires much tedious programming, and so is considered as not supported.

Use existing debuggers
: By this is meant using common debuggers that can operate using debug data in common formats embedded in the executable. A specialized debugger useful only with that language is not required.

Struct member alignment control
: Although many C/C++ compilers contain pragmas to specify struct alignment, these are nonstandard and incompatible from compiler to compiler.

The C# standard ECMA-334 25.5.8 says only this about struct member alignment: *"The order in which members are packed into a struct is unspecified. For alignment purposes, there may be unnamed padding at the beginning of a struct, within a struct, and at the end of the struct. The contents of the bits used as padding are indeterminate."* Therefore, although Microsoft may have extensions to support specific member alignment, they are not an official part of standard C#.

Support all C types
: C99 adds many new types not supported by C++.

80 bit floating point
: While the standards for C and C++ specify long doubles, few compilers (besides Digital Mars C/C++) actually implement 80 bit (or longer) floating point types.

Mixins
: Mixins have many different meanings in different programming languages. D mixins mean taking an arbitrary sequence of declarations and inserting (mixing) them into the current scope. Mixins can be done at the global, class, struct, or local level.

C++ Mixins
: C++ mixins refer to a couple different techniques. The first is analogous to D's interface classes. The second is to create a template of the form:

```
template <class Base> class Mixin : public Base
{
    ... mixin body ...
}
```

D mixins are different.

Comparison with Ada
: James S. Rogers has written a comparison chart with Ada.

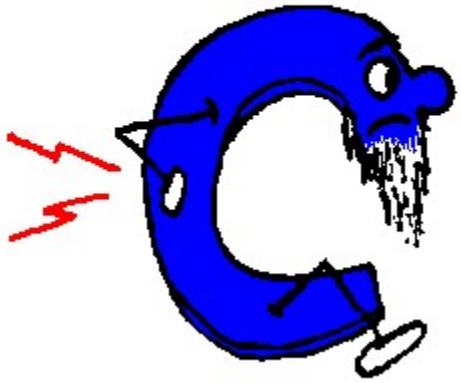
Inner (adaptor) classes
: A *nested class* is one whose definition is within the scope of another class. An *inner class* is a nested class that can also reference the members and fields of the lexically enclosing class; one can think of it as if it contained a 'this' pointer to the enclosing class.

Errors

If I've made any errors in this table, please contact me so I can correct them:

Programming in D for C Programmers

Every experienced C programmer accumulates a series of idioms and techniques which become second nature. Sometimes, when learning a new language, those idioms can be so comfortable it's hard to see how to do the equivalent in the new language. So here's a collection of common C techniques, and how to do the corresponding task in D.



Since C does not have object-oriented features, there's a separate section for object-oriented issues Programming in D for C++ Programmers.

The C preprocessor is covered in The C Preprocessor vs D.

Getting the Size of a Type

The C Way

```
sizeof(int)
sizeof(char *)
sizeof(double)
sizeof(struct Foo)
```

The D Way

Use the size property:

```
int.size
(char *).size
double.size
Foo.size
```

Get the max and min values of a type

The C Way

```
#include <limits.h>
#include <math.h>

CHAR_MAX
CHAR_MIN
ULONG_MAX
DBL_MIN
```

The D Way

```
char.max
```

```
char.min
ulong.max
double.min
```

Primitive Types

C to D types

```
bool                => bit
char                => char
signed char         => byte
unsigned char       => ubyte
short               => short
unsigned short      => ushort
wchar_t             => wchar
int                 => int
unsigned            => uint
long                => int
unsigned long       => uint
long long           => long
unsigned long long  => ulong
float               => float
double              => double
long double         => extended
_Imaginary long double => imaginary
_Complex long double   => complex
```

Although char is an unsigned 8 bit type, and wchar is an unsigned 16 bit type, they have their own separate types in order to aid overloading and type safety.

Ints and unsigneds in C are of varying size; not so in D.

Special Floating Point Values

The C Way

```
#include <fp.h>

NAN
INFINITY

#include <float.h>

DBL_DIG
DBL_EPSILON
DBL_MANT_DIG
```

```
DBL_MAX_10_EXP
DBL_MAX_EXP
DBL_MIN_10_EXP
DBL_MIN_EXP
```

The D Way

```
double.nan
double.infinity
double.dig
double.epsilon
double.mant_dig
double.max_10_exp
double.max_exp
double.min_10_exp
double.min_exp
```

Remainder after division of floating point numbers

The C Way

```
#include <math.h>

float f = fmodf(x,y);
double d = fmod(x,y);
long double e = fmodl(x,y);
```

The D Way

D supports the remainder ('%') operator on floating point operands:

```
float f = x % y;
double d = x % y;
extended e = x % y;
```

Dealing with NAN's in floating point compares

The C Way

C doesn't define what happens if an operand to a compare is NAN, and few C compilers check for it (the Digital Mars C compiler is an exception, DM's compilers do check for NAN operands).

```
#include <math.h>

if (isnan(x) || isnan(y))
    result = FALSE;
else
    result = (x < y);
```

The D Way

D offers a full complement of comparisons and operators that work with NAN arguments.

```
result = (x < y);        // false if x or y is nan
```

Assert's are a necessary part of any good defensive coding strategy.

The C Way

C doesn't directly support assert, but does support __FILE__ and __LINE__ from which an assert macro can be built. In fact, there appears to be practically no other use for __FILE__ and __LINE__.

```
#include <assert.h>

assert(e == 0);
```

The D Way

D simply builds assert into the language:

```
assert(e == 0);
```

[NOTE: trace functions?]

Initializing all elements of an array

The C Way

```
#define ARRAY_LENGTH        17
int array[ARRAY_LENGTH];
for (i = 0; i < ARRAY_LENGTH; i++)
    array[i] = value;
```

The D Way

```
int array[17];
array[] = value;
```

Looping through an array

The C Way

The array length is defined separately, or a clumsy sizeof() expression is used to get the length.

```
#define ARRAY_LENGTH        17
int array[ARRAY_LENGTH];
for (i = 0; i < ARRAY_LENGTH; i++)
    func(array[i]);
```

or:

```
int array[17];
for (i = 0; i < sizeof(array) / sizeof(array[0]); i++)
    func(array[i]);
```

The D Way

The length of an array is accessible the property "length".

```
int array[17];
for (i = 0; i < array.length; i++)
    func(array[i]);
```

or even better:

```
int array[17];
foreach (int value; array)
    func(value);
```

Creating an array of variable size

The C Way

C cannot do this with arrays. It is necessary to create a separate variable for the length, and then explicitly manage the size of the array:

```
#include <stdlib.h>
```

```
int array_length;
int *array;
int *newarray;

newarray = (int *) realloc(array, (array_length + 1) *
sizeof(int));
if (!newarray)
    error("out of memory");
array = newarray;
array[array_length++] = x;
```

The D Way

D supports dynamic arrays, which can be easily resized. D supports all the requisite memory management.

```
int[] array;

array.length = array.length + 1;
array[array.length - 1] = x;
```

String Concatenation

The C Way

There are several difficulties to be resolved, like when can storage be free'd, dealing with null pointers, finding the length of the strings, and memory allocation:

```
#include <string.h>

char *s1;
char *s2;
char *s;

// Concatenate s1 and s2, and put result in s
free(s);
s = (char *)malloc((s1 ? strlen(s1) : 0) +
                   (s2 ? strlen(s2) : 0) + 1);
if (!s)
    error("out of memory");
if (s1)
    strcpy(s, s1);
else
    *s = 0;
if (s2)
    strcpy(s + strlen(s), s2);

// Append "hello" to s
```

```
                char hello[] = "hello";
                char *news;
                size_t lens = s ? strlen(s) : 0;
                news = (char *)realloc(s, (lens + sizeof(hello) + 1) *
sizeof(char));
                if (!news)
                    error("out of memory");
                s = news;
                memcpy(s + lens, hello, sizeof(hello));
```

The D Way

D overloads the operators ~ and ~= for char and wchar arrays to mean concatenate and append, respectively:

```
                char[] s1;
                char[] s2;
                char[] s;

                s = s1 ~ s2;
                s ~= "hello";
```

Formatted printing

The C Way

printf() is the general purpose formatted print routine:

```
                #include <stdio.h>

                printf("Calling all cars %d times!\n", ntimes);
```

The D Way

What can we say? printf() rules:

```
                import stdio;

                printf("Calling all cars %d times!\n", ntimes);
```

Forward referencing functions

The C Way

Functions cannot be forward referenced. Hence, to call a function not yet encountered in the source file, it is necessary to insert a function declaration lexically preceding the call.

```
void forwardfunc();

void myfunc()
{
    forwardfunc();
}

void forwardfunc()
{
    ...
}
```

The D Way

The program is looked at as a whole, and so not only is it not necessary to code forward declarations, it is not even allowed! D avoids the tedium and errors associated with writing forward referenced function declarations twice. Functions can be defined in any order.

```
void myfunc()
{
    forwardfunc();
}

void forwardfunc()
{
    ...
}
```

Functions that have no arguments

The C Way

```
void function(void);
```

The D Way

D is a strongly typed language, so there is no need to explicitly say a function takes no arguments, just don't declare it has having arguments.

```
void function()
{
    ...
}
```

Labelled break's and continue's.

The C Way

Break's and continue's only apply to the innermost nested loop or switch, so a multilevel break must use a goto:

```
for (i = 0; i < 10; i++)
{
    for (j = 0; j < 10; j++)
    {
        if (j == 3)
            goto Louter;
        if (j == 4)
            goto L2;
    }
  L2:
    ;
}
Louter:
    ;
```

The D Way

Break and continue statements can be followed by a label. The label is the label for an enclosing loop or switch, and the break applies to that loop.

```
Louter:
  for (i = 0; i < 10; i++)
  {
      for (j = 0; j < 10; j++)
      {
          if (j == 3)
              break Louter;
          if (j == 4)
              continue Louter;
      }
  }
  // break Louter goes here
```

Goto Statements

The C Way

The much maligned goto statement is a staple for professional C coders. It's necessary to make up for sometimes inadequate control flow statements.

The D Way

Many C-way goto statements can be eliminated with the D feature of labelled break and continue statements. But D is a practical language for practical programmers who know when the rules need to be broken. So of course D supports the goto!

Struct tag name space

The C Way

It's annoying to have to put the struct keyword every time a type is specified, so a common idiom is to use:

```
typedef struct ABC { ... } ABC;
```

The D Way

Struct tag names are not in a separate name space, they are in the same name space as ordinary names. Hence:

```
struct ABC { ... };
```

Looking up strings

The C Way

Given a string, compare the string against a list of possible values and take action based on which one it is. A typical use for this might be command line argument processing.

```
#include <string.h>
void dostring(char *s)
{
    enum Strings { Hello, Goodbye, Maybe, Max };
    static char *table[] = { "hello", "goodbye", "maybe" };
```

```
    int i;

    for (i = 0; i < Max; i++)
    {
        if (strcmp(s, table[i]) == 0)
            break;
    }
    switch (i)
    {
        case Hello:         ...
        case Goodbye:         ...
        case Maybe:         ...
        default:         ...
    }
}
```

The problem with this is trying to maintain 3 parallel data structures, the enum, the table, and the switch cases. If there are a lot of values, the connection between the 3 may not be so obvious when doing maintenance, and so the situation is ripe for bugs. Additionally, if the number of values becomes large, a binary or hash lookup will yield a considerable performance increase over a simple linear search. But coding these can be time consuming, and they need to be debugged. It's typical that such just never gets done.

The D Way

D extends the concept of switch statements to be able to handle strings as well as numbers. Then, the way to code the string lookup becomes straightforward:

```
void dostring(char[] s)
{
    switch (s)
    {
        case "hello":         ...
        case "goodbye":         ...
        case "maybe":         ...
        default:         ...
    }
}
```

Adding new cases becomes easy. The compiler can be relied on to generate a fast lookup scheme for it, eliminating the bugs and time required in hand-coding one.

Setting struct member alignment

The C Way

It's done through a command line switch which affects the entire program, and woe results if any modules or libraries didn't get recompiled. To address this, #pragma's are used:

```
#pragma pack(1)
struct ABC
{
    ...
};
#pragma pack()
```

But #pragmas are nonportable both in theory and in practice from compiler to compiler.

The D Way

Clearly, since much of the point to setting alignment is for portability of data, a portable means of expressing it is necessary.

```
struct ABC
{
    int z;                      // z is aligned to the default

  align (1) int x;              // x is byte aligned
  align (4)
  {
    ...                         // declarations in {} are dword
aligned
  }
  align (2):                    // switch to word alignment from
here on

    int y;                      // y is word aligned
}
```

Anonymous Structs and Unions

Sometimes, it's nice to control the layout of a struct with nested structs and unions.

The C Way

C doesn't allow anonymous structs or unions, which means that dummy tag names and dummy members are necessary:

```
struct Foo
{   int i;
    union Bar
    {
        struct Abc { int x; long y; } _abc;
        char *p;
    } _bar;
};

#define x _bar._abc.x
#define y _bar._abc.y
```

```
#define p _bar.p

struct Foo f;

f.i;
f.x;
f.y;
f.p;
```

Not only is it clumsy, but using macros means a symbolic debugger won't understand what is being done, and the macros have global scope instead of struct scope.

The D Way

Anonymous structs and unions are used to control the layout in a more natural manner:

```
struct Foo
{   int i;
    union
    {
        struct { int x; long y; }
        char* p;
    }
}

Foo f;

f.i;
f.x;
f.y;
f.p;
```

Declaring struct types and variables.

The C Way

Is to do it in one statement ending with a semicolon:

```
struct Foo { int x; int y; } foo;
```

Or to separate the two:

```
struct Foo { int x; int y; };        // note terminating ;
struct Foo foo;
```

The D Way

Struct definitions and declarations can't be done in the same statement:

```
            struct Foo { int x; int y; }        // note there is no
terminating ;
            Foo foo;
```

which means that the terminating ; can be dispensed with, eliminating the confusing difference between struct {} and function & block {} in how semicolons are used.

Getting the offset of a struct member.

The C Way

Naturally, another macro is used:

```
            #include <stddef>
            struct Foo { int x; int y; };

            off = offsetof(Foo, y);
```

The D Way

An offset is just another property:

```
            struct Foo { int x; int y; }

            off = Foo.y.offset;
```

Union initializations.

The C Way

Unions are initialized using the "first member" rule:

```
            union U { int a; long b; };
            union U x = { 5 };                  // initialize member 'a' to 5
```

Adding union members or rearranging them can have disastrous consequences for any initializers.

The D Way

In D, which member is being initialized is mentioned explicitly:

```
            union U { int a; long b; }
            U x = { a:5 }
```

avoiding the confusion and maintenance problems.

Struct initializations.

The C Way

Members are initialized by their position within the {}'s:

```
struct S { int a; int b; };
struct S x = { 5, 3 };
```

This isn't much of a problem with small structs, but when there are numerous members, it becomes tedious to get the initializers carefully lined up with the field declarations. Then, if members are added or rearranged, all the initializations have to be found and modified appropriately. This is a minefield for bugs.

The D Way

Member initialization can be done explicitly:

```
struct S { int a; int b; }
S x = { b:3, a:5 }
```

The meaning is clear, and there no longer is a positional dependence.

Array initializations.

The C Way

C initializes array by positional dependence:

```
int a[3] = { 3,2,2 };
```

Nested arrays may or may not have the { }:

```
int b[3][2] = { 2,3, {6,5}, 3,4 };
```

The D Way

D does it by positional dependence too, but an index can be used as well: The following all produce the same result:

```
    int[3] a = [ 3, 2, 0 ];
    int[3] a = [ 3, 2 ];              // unsupplied initializers are 0, just
like in C
```

```
    int[3] a = [ 2:0, 0:3, 1:2 ];
    int[3] a = [ 2:0, 0:3, 2 ];       // if not supplied, the index is the
previous
                                      // one plus one.
```

This can be handy if the array will be indexed by an enum, and the order of enums may be changed or added to:

```
    enum color { black, red, green }
    int[3] c = [ black:3, green:2, red:5 ];
```

Nested array initializations must be explicit:

```
    int[2][3] b = [ [2,3], [6,5], [3,4] ];

    int[2][3] b = [[2,6,3],[3,5,4]];            // error
```

Escaped String Literals

The C Way

C has problems with the DOS file system because a \ is an escape in a string. To specifiy file c:\root\file.c:

```
    char file[] = "c:\\root\\file.c";
```

This gets even more unpleasant with regular expressions. Consider the escape sequence to match a quoted string:

```
    /"[^\\]*(\\.[^\\]*)*"/
```

In C, this horror is expressed as:

```
    char quoteString[] = "\"[^\\\\]*(\\\\.[^\\\\]*)*\"";
```

The D Way

Within strings, it is WYSIWYG (what you see is what you get). Escapes are in separate strings. So:

```
    char[] file = 'c:\root\file.c';
    char[] quoteString = \"  r"[^\\]*(\\.[^\\]*)*"  \";
```

The famous hello world string becomes:

```
    char[] hello = "hello world" \n;
```

Ascii vs Wide Characters

Modern programming requires that wchar strings be supported in an easy way, for internationalization of the programs.

The C Way

C uses the wchar_t and the L prefix on strings:

```
#include <wchar.h>
char foo_ascii[] = "hello";
wchar_t foo_wchar[] = L"hello";
```

Things get worse if code is written to be both ascii and wchar compatible. A macro is used to switch strings from ascii to wchar:

```
#include <tchar.h>
tchar string[] = TEXT("hello");
```

The D Way

The type of a string is determined by semantic analysis, so there is no need to wrap strings in a macro call:

```
char[] foo_ascii = "hello";            // string is taken to be ascii
wchar[] foo_wchar = "hello";       // string is taken to be wchar
```

Arrays that parallel an enum

The C Way

Consider:

```
enum COLORS { red, blue, green, max };
char *cstring[max] = {"red", "blue", "green" };
```

This is fairly easy to get right because the number of entries is small. But suppose it gets to be fairly large. Then it can get difficult to maintain correctly when new entries are added.

The D Way

```
enum COLORS { red, blue, green }

 char[][COLORS.max + 1] cstring =
 [
     COLORS.red   : "red",
```

```
        COLORS.blue  : "blue",
        COLORS.green : "green",
    ];
```

Not perfect, but better.

Creating a new typedef'd type

The C Way

Typedef's in C are weak, that is, they really do not introduce a new type. The compiler doesn't distinguish between a typedef and its underlying type.

```
typedef void *Handle;
void foo(void *);
void bar(Handle);

Handle h;
foo(h);                 // coding bug not caught
bar(h);                 // ok
```

The C solution is to create a dummy struct whose sole purpose is to get type checking and overloading on the new type.

```
struct Handle__ { void *value; }
typedef struct Handle__ *Handle;
void foo(void *);
void bar(Handle);

Handle h;
foo(h);                 // syntax error
bar(h);                 // ok
```

Having a default value for the type involves defining a macro, a naming convention, and then pedantically following that convention:

```
#define HANDLE_INIT ((Handle)-1)

Handle h = HANDLE_INIT;
h = func();
if (h != HANDLE_INIT)
    ...
```

For the struct solution, things get even more complex:

```
struct Handle__ HANDLE_INIT;

void init_handle()      // call this function upon startup
{
```

```
    HANDLE_INIT.value = (void *)-1;
}

Handle h = HANDLE_INIT;
h = func();
if (memcmp(&h,&HANDLE_INIT,sizeof(Handle)) != 0)
    ...
```

There are 4 names to remember: `Handle, HANDLE_INIT, struct Handle__, value`.

The D Way

No need for idiomatic constructions like the above. Just write:

```
typedef void* Handle;
void foo(void*);
void bar(Handle);

Handle h;
foo(h);
bar(h);
```

To handle a default value, add an initializer to the typedef, and refer to it with the `.init` property:

```
typedef void* Handle = cast(void*)(-1);
Handle h;
h = func();
if (h != Handle.init)
    ...
```

There's only one name to remember: `Handle`.

Comparing structs

The C Way

While C defines struct assignment in a simple, convenient manner:

```
struct A x, y;
...
x = y;
```

it does not for struct comparisons. Hence, to compare two struct instances for equality:

```
#include <string.h>

struct A x, y;
...
if (memcmp(&x, &y, sizeof(struct A)) == 0)
    ...
```

Note the obtuseness of this, coupled with the lack of any kind of help from the language with type checking.

There's a nasty bug lurking in the memcmp(). The layout of a struct, due to alignment, can have 'holes' in it. C does not guarantee those holes are assigned any values, and so two different struct instances can have the same value for each member, but compare different because the holes contain different garbage.

The D Way

D does it the obvious, straightforward way:

```
A x, y;
...
if (x == y)
    ...
```

Comparing strings

The C Way

The library function strcmp() is used:

```
char string[] = "hello";

if (strcmp(string, "betty") == 0)       // do strings match?
    ...
```

C uses 0 terminated strings, so the C way has an inherent inefficiency in constantly scanning for the terminating 0.

The D Way

Why not use the == operator?

```
char[] string = "hello";

if (string == "betty")
```

```
    ...
```

D strings have the length stored separately from the string. Thus, the implementation of string compares can be much faster than in C (the difference being equivalent to the difference in speed between the C memcmp() and strcmp()).

D supports comparison operators on strings, too:

```
char[] string = "hello";

if (string < "betty")
    ...
```

which is useful for sorting/searching.

Sorting arrays

The C Way

Although many C programmers tend to reimplmement bubble sorts over and over, the right way to sort in C is to use qsort():

```
int compare(const void *p1, const void *p2)
{
    type *t1 = (type *)p1;
    type *t1 = (type *)p2;

    return *t1 - *t2;
}

type array[10];
...
qsort(array, sizeof(array)/sizeof(array[0]), sizeof(array[0]),
compare);
```

A compare() must be written for each type, and much careful typo-prone code needs to be written to make it work.

The D Way

Sorting couldn't be easier:

```
type[] array;
...
array.sort;             // sort array in-place
```

Volatile memory access

The C Way

To access volatile memory, such as shared memory or memory mapped I/O, a pointer to volatile is created:

```
volatile int *p = address;

i = *p;
```

The D Way

D has volatile as a statement type, not as a type modifier:

```
int* p = address;

volatile { i = *p; }
```

String literals

The C Way

String literals in C cannot span multiple lines, so to have a block of text it is necessary to use \ line splicing:

```
"This text spans\n\
multiple\n\
lines\n"
```

If there is a lot of text, this can wind up being tedious.

The D Way

String literals can span multiple lines, as in:

```
"This text spans
multiple
lines
"
```

So blocks of text can just be cut and pasted into the D source.

Data Structure Traversal

The C Way

Consider a function to traverse a recursive data structure. In this example, there's a simple symbol table of strings. The data structure is an array of binary trees. The code needs to do an exhaustive search of it to find a particular string in it, and determine if it is a unique instance.

To make this work, a helper function `membersearchx` is needed to recursively walk the trees. The helper function needs to read and write some context outside of the trees, so a custom `struct Paramblock` is created and a pointer to it is used to maximize efficiency.

```
    struct Symbol
    {   char *id;
        struct Symbol *left;
        struct Symbol *right;
    };

    struct Paramblock
    {   char *id;
        struct Symbol *sm;
    };

    static void membersearchx(struct Paramblock *p, struct Symbol *s)
    {
        while (s)
        {
            if (strcmp(p->id,s->id) == 0)
            {
                if (p->sm)
                    error("ambiguous member %s\n",p->id);
                p->sm = s;
            }

            if (s->left)
                membersearchx(p,s->left);
            s = s->right;
        }
    }

    struct Symbol *symbol_membersearch(Symbol *table[], int tablemax, char
*id)
    {
        struct Paramblock pb;
        int i;

        pb.id = id;
        pb.sm = NULL;
        for (i = 0; i < tablemax; i++)
        {
            membersearchx(pb, table[i]);
```

```
        }
        return pb.sm;
    }
```

The D Way

This is the same algorithm in D, and it shrinks dramatically. Since nested functions have access to the lexically enclosing function's variables, there's no need for a Paramblock or to deal with its bookkeeping details. The nested helper function is contained wholly within the function that needs it, improving locality and maintainability.

The performance of the two versions is indistinguishable.

```
class Symbol
{   char[] id;
    Symbol left;
    Symbol right;
}

Symbol symbol_membersearch(Symbol[] table, char[] id)
{   Symbol sm;

    void membersearchx(Symbol s)
    {
        while (s)
        {
            if (id == s.id)
            {
                if (sm)
                    error("ambiguous member %s\n", id);
                sm = s;
            }

            if (s.left)
                membersearchx(s.left);
            s = s.right;
        }
    }

    for (int i = 0; i < table.length; i++)
    {
        membersearchx(table[i]);
    }
    return sm;
}
```

Unsigned Right Shift

The C Way

The right shift operators >> and >>= are signed shifts if the left operand is a signed integral type, and are unsigned right shifts if the left operand is an unsigned integral type. To produce an unsigned right shift on an int, a cast is necessary:

```
int i, j;
...
j = (unsigned)i >> 3;
```

If `i` is an `int`, this works fine. But if `i` is of a typedef'd type,

```
myint i, j;
...
j = (unsigned)i >> 3;
```

and `myint` happens to be a `long int`, then the cast to unsigned will silently throw away the most significant bits, corrupting the answer.

The D Way

D has the right shift operators >> and >>= which behave as they do in C. But D also has explicitly unsigned right shift operators >>> and >>>= which will do an unsigned right shift regardless of the sign of the left operand. Hence,

```
myint i, j;
...
j = i >>> 3;
```

avoids the unsafe cast and will work as expected with any integral type.

Dynamic Closures

The C Way

Consider a reusable container type. In order to be reusable, it must support a way to apply arbitrary code to each element of the container. This is done by creating an *apply* function that accepts a function pointer to which is passed each element of the container contents.

A generic context pointer is also needed, represented here by `void *p`. The example here is of a trivial container class that holds an array of int's, and a user of that container that computes the maximum of those int's.

```
struct Collection
{
    int array[10];

    void apply(void *p, void (*fp)(void *, int))
    {
        for (int i = 0; i < sizeof(array)/sizeof(array[0]); i++)
            fp(p, array[i]);
    }
};

void comp_max(void *p, int i)
{
    int *pmax = (int *)p;

    if (i > *pmax)
        *pmax = i;
}

void func(Collection *c)
{
    int max = INT_MIN;

    c->apply(&max, comp_max);
}
```

The C way makes heavy use of pointers and casting. The casting is tedious, error prone, and loses all type safety.

The D Way

The D version makes use of *delegates* to transmit context information for the *apply* function, and *nested functions* both to capture context information and to improve locality.

```
class Collection
{
    int[10] array;

    void apply(void delegate(int) fp)
    {
        for (int i = 0; i < array.length; i++)
            fp(array[i]);
    }
}

void func(Collection c)
{
    int max = int.min;

    void comp_max(int i)
    {
        if (i > max)
            max = i;
    }

    c.apply(comp_max);
}
```

Pointers are eliminated, as well as casting and generic pointers. The D version is fully type safe. An alternate method in D makes use of *function literals*:

```
void func(Collection c)
{
    int max = int.min;

    c.apply(delegate(int i) { if (i > max) max = i; } );
}
```

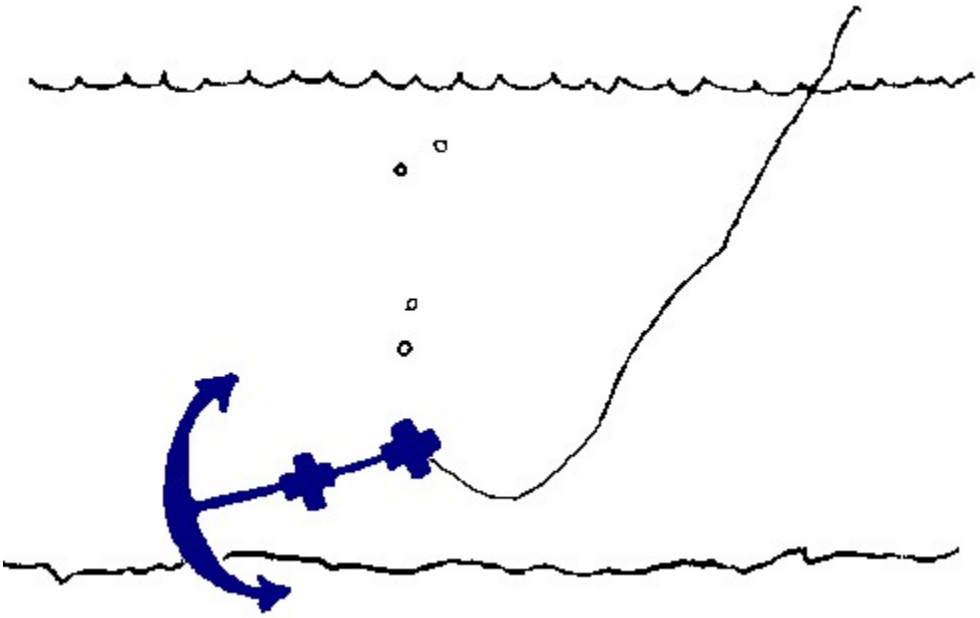
eliminating the need to create irrelevant function names.

Programming in D for C++ Programmers

Every experienced C++ programmer accumulates a series of idioms and techniques which become second nature. Sometimes, when learning a new language, those idioms can be so comfortable it's hard to see how to do the equivalent in the new language. So here's a collection of common C++ techniques, and how to do the corresponding task in D.

See also: Programming in D for C Programmers

- Defining Constructors
- Base class initialization



Defining constructors

The C++ Way

Constructors have the same name as the class:

```
class Foo
{
        Foo(int x);
};
```

The D Way

Constructors are defined with the this keyword:

```
class Foo
{
        this(int x) { }
```

```
}
```

which reflects how they are used in D.

Base class initialization

The C++ Way

Base constructors are called using the base initializer syntax.

```
class A { A() {... } };
class B : A
{
     B(int x)
        : A()           // call base constructor
     {  ...
     }
};
```

The D Way

The base class constructor is called with the super syntax:

```
class A { this() { ... } }
class B : A
{
     this(int x)
     {  ...
        super();                // call base constructor
        ...
     }
}
```

It's superior to C++ in that the base constructor call can be flexibly placed anywhere in the derived constructor. D can also have one constructor call another one:

```
class A
{       int a;
        int b;
        this() { a = 7; b = foo(); }
        this(int x)
        {
            this();
            a = x;
        }
}
```

Members can also be initialized to constants before the constructor is ever called, so the above example is equivalently written as:

```
class A
{       int a = 7;
        int b;
        this() { b = foo(); }
        this(int x)
        {
            this();
            a = x;
        }
}
```

Comparing structs

The C++ Way

While C++ defines struct assignment in a simple, convenient manner:

```
struct A x, y;
...
x = y;
```

it does not for struct comparisons. Hence, to compare two struct instances for equality:

```
#include <string.h>

struct A x, y;

inline bool operator==(const A& x, const A& y)
{
    return (memcmp(&x, &y, sizeof(struct A)) == 0);
}
...
if (x == y)
    ...
```

Note that the operator overload must be done for every struct needing to be compared, and the implementation of that overloaded operator is free of any language help with type checking. The C++ way has an additional problem in that just inspecting the (x == y) does not give a clue what is actually happening, you have to go and find the particular overloaded operator==() that applies to verify what it really does.

There's a nasty bug lurking in the memcmp() implementation of operator==(). The layout of a struct, due to alignment, can have 'holes' in it. C++ does not guarantee those holes are assigned any values, and so two different struct instances can have the same value for each member, but

compare different because the holes contain different garbage.

To address this, the operator==() can be implemented to do a memberwise compare. Unfortunately, this is unreliable because (1) if a member is added to the struct definition one may forget to add it to operator==(), and (2) floating point nan values compare unequal even if their bit patterns match.

There just is no robust solution in C++.

The D Way

D does it the obvious, straightforward way:

```
A x, y;
...
if (x == y)
    ...
```

Creating a new typedef'd type

The C++ Way

Typedef's in C++ are weak, that is, they really do not introduce a new type. The compiler doesn't distinguish between a typedef and its underlying type.

```
#define HANDLE_INIT     ((Handle)(-1))
typedef void *Handle;
void foo(void *);
void bar(Handle);

Handle h = HANDLE_INIT;
foo(h);                 // coding bug not caught
bar(h);                 // ok
```

The C++ solution is to create a dummy struct whose sole purpose is to get type checking and overloading on the new type.

```
#define HANDLE_INIT     ((void *)(-1))
struct Handle
{   void *ptr;
    Handle() { ptr = HANDLE_INIT; }     // default initializer
    Handle(int i) { ptr = (void *)i; }
    operator void*() { return ptr; }    // conversion to underlying
type
};
void bar(Handle);

Handle h;
```

```
bar(h);
h = func();
if (h != HANDLE_INIT)
    ...
```

The D Way

No need for idiomatic constructions like the above. Just write:

```
typedef void *Handle = cast(void *)-1;
void bar(Handle);

Handle h;
bar(h);
h = func();
if (h != Handle.init)
    ...
```

Note how a default initializer can be supplied for the typedef as a value of the underlying type.

Friends

The C++ Way

Sometimes two classes are tightly related but not by inheritance, but need to access each other's private members. This is done using `friend` declarations:

```
class A
{
    private:
        int a;

    public:
        int foo(B *j);
        friend class B;
        friend int abc(A *);
};

class B
{
    private:
        int b;

    public:
        int bar(A *j);
        friend class A;
};

int A::foo(B *j) { return j->b; }
```

```
int B::bar(A *j) { return j->a; }

int abc(A *p) { return p->a; }
```

The D Way

In D, friend access is implicit in being a member of the same module. It makes sense that tightly related classes should be in the same module, so implicitly granting friend access to other module members solves the problem neatly:

```
module X;

class A
{
    private:
        static int a;

    public:
        int foo(B j) { return j.b; }
}

class B
{
    private:
        static int b;

    public:
        int bar(A j) { return j.a; }
}

int abc(A p) { return p.a; }
```

The `private` attribute prevents other modules from accessing the members.

Operator overloading

The C++ Way

Given a struct that creates a new arithmetic data type, it's convenient to overload the comparison operators so it can be compared against integers:

```
struct A
{
        virtual int operator <  (int i);
        virtual int operator <= (int i);
        virtual int operator >  (int i);
        virtual int operator >= (int i);
```

```
        static int operator <  (int i, A *a) { return a >  i; }
        static int operator <= (int i, A *a) { return a >= i; }
        static int operator >  (int i, A *a) { return a <  i; }
        static int operator >= (int i, A *a) { return a <= i; }
};
```

A total of 8 functions are necessary, and all the latter 4 do is just rewrite the expression so the virtual functions can be used. Note the asymmetry between the virtual functions, which have `(a < i)` as the left operand, and the non-virtual static function necessary to handle `(i < a)` operations.

The D Way

D recognizes that the comparison operators are all fundamentally related to each other. So only one function is necessary:

```
struct A
{
        int cmp(int i);
}
```

The compiler automatically interprets all the <, <=, > and >= operators in terms of the `cmp` function, as well as handling the cases where the left operand is not an object reference.

Similar sensible rules hold for other operator overloads, making using operator overloading in D much less tedious and less error prone. Far less code needs to be written to accomplish the same effect.

Namespace using declarations

The C++ Way

A *using-declaration* in C++ is used to bring a name from a namespace scope into the current scope:

```
namespace Foo
{
    int x;
}
using Foo::x;
```

The D Way

D uses modules instead of namespaces and #include files, and alias declarations take the place of

using declarations:

```
---- Module Foo.d ------
module Foo;
int x;

---- Another module ----
import Foo;
alias Foo.x x;
```

Alias is a much more flexible than the single purpose using declaration. Alias can be used to rename symbols, refer to template members, refer to nested class types, etc.

RAII (Resource Acquisition Is Initialization)

The C++ Way

In C++, resources like memory, etc., all need to be handled explicitly. Since destructors automatically get called when leaving a scope, RAII is implemented by putting the resource release code into the destructor:

```
class File
{   Handle *h;

    ~File()
    {
        h->release();
    }
};
```

The D Way

The bulk of resource release problems are simply keeping track of and freeing memory. This is handled automatically in D by the garbage collector. The second common resources used are semaphores and locks, handled automatically with D's `synchronized` declarations and statements.

The few RAII issues left are handled by *auto* classes. Auto classes get their destructors run when they go out of scope.

```
auto class File
{   Handle h;

    ~this()
    {
        h.release();
    }
}
```

```
void test()
{
    if (...)
    {   auto File f = new File();
        ...
    } // f.~this() gets run at closing brace, even if
      // scope was exited via a thrown exception
}
```

Properties

The C++ Way

It is common practice to define a field, along with object-oriented get and set functions for it:

```
class Abc
{
  public:
    void setProperty(int newproperty) { property = newproperty; }
    int getProperty() { return property; }

  private:
    int property;
};

Abc a;
a.setProperty(3);
int x = a.getProperty();
```

All this is quite a bit of typing, and it tends to make code unreadable by filling it with getProperty() and setProperty() calls.

The D Way

Properties can be get and set using the normal field syntax, yet the get and set will invoke methods instead.

```
class Abc
{
    void property(int newproperty) { myprop = newproperty; } // set
    int property() { return myprop; }   // get

  private:
    int myprop;
}
```

which is used as:

```
Abc a;
a.property = 3;          // equivalent to a.property(3)
int x = a.property;      // equivalent to int x = a.property()
```

Thus, in D a property can be treated like it was a simple field name. A property can start out actually being a simple field name, but if later if becomes necessary to make getting and setting it function calls, no code needs to be modified other than the class definition. It obviates the wordy practice of defining get and set properties 'just in case' a derived class should need to override them. It's also a way to have interface classes, which do not have data fields, behave syntactically as if they did.

Recursive Templates

The C++ Way

An advanced use of templates is to recursively expand them, relying on specialization to end it. A template to compute a factorial would be:

```
template<int n> class factorial
{
    public:
        enum { result = n * factorial<n - 1>::result };
};

template<> class factorial<1>
{
    public:
        enum { result = 1 };
};

void test()
{
    printf("%d\n", factorial<4>::result); // prints 24
}
```

The D Way

The D version is analogous, though a little simpler, taking advantage of promotion of single template members to the enclosing name space:

```
template factorial(int n)
{
    enum { factorial = n* .factorial!(n-1) }
}

template factorial(int n : 1)
```

```
{
    enum { factorial = 1 }
}

void test()
{
    printf("%d\n", factorial!(4));      // prints 24
}
```

The C Preprocessor Versus D

Back when C was invented, compiler technology was primitive. Installing a text macro preprocessor onto the front end was a straightforward and easy way to add many powerful features. The increasing size & complexity of programs have illustrated that these features come with many inherent problems. D doesn't have a preprocessor; but D provides a more scalable means to solve the same problems.

Header Files

The C Preprocessor Way

C and C++ rely heavilly on textual inclusion of header files. This frequently results in the compiler having to recompile tens of thousands of lines of code over and over again for every source file, an obvious source of slow compile times. What header files are normally used for is more appropriately done doing a symbolic, rather than textual, insertion. This is done with the import statement. Symbolic inclusion means the compiler just loads an already compiled symbol table. The needs for macro "wrappers" to prevent multiple #inclusion, funky #pragma once syntax, and incomprehensible fragile syntax for precompiled headers are simply unnecessary and irrelevant to D.

```
#include <stdio.h>
```

The D Way

D uses symbolic imports:

```
import std.c.stdio;
```

#pragma once

The C Preprocessor Way

C header files frequently need to be protected against being #include'd multiple times. To do it, a header file will contain the line:

```
#pragma once
```

or the more portable:

```
#ifndef __STDIO_INCLUDE
#define __STDIO_INCLUDE
... header file contents
#endif
```

The D Way

Completely unnecessary since D does a symbolic include of import files; they only get imported once no matter how many times the import declaration appears.

#pragma pack

The C Preprocessor Way

This is used in C to adjust the alignment for structs.

The D Way

For D classes, there is no need to adjust the alignment (in fact, the compiler is free to rearrange the data fields to get the optimum layout, much as the compiler will rearrange local variables on the stack frame). For D structs that get mapped onto externally defined data structures, there is a need, and it is handled with:

```
struct Foo
{
        align (4):      // use 4 byte alignment
        ...
}
```

Macros

Preprocessor macros add powerful features and flexibility to C. But they have a downside:

- Macros have no concept of scope; they are valid from the point of definition to the end of the source. They cut a swath across .h files, nested code, etc. When #include'ing tens of thousands of lines of macro definitions, it becomes problematicalto avoid inadvertent macro expansions.
- Macros are unknown to the debugger. Trying to debug a program with symbolic data is undermined by the debugger only knowing about macro expansions, not themacros themselves.
- Macros make it impossible to tokenize source code, as an earlier macro change can arbitrarily redo tokens.
- The purely textual basis of macros leads to arbitrary and inconsistent usage, making code using macros error prone. (Some attempt to resolve this was introduced with templates in C++.)
- Macros are still used to make up for deficits in the language's expressive capabiltiy, such as for "wrappers" around header files.

Here's an enumeration of the common uses for macros, and the corresponding feature in D:

1. Defining literal constants:

The C Preprocessor Way

```
    #define VALUE   5
2.
```

The D Way

```
    const int VALUE = 5;
3.
```

4. Creating a list of values or flags:

The C Preprocessor Way

```
    int flags:
5.      #define FLAG_X  0x1
6.      #define FLAG_Y  0x2
```

```
7.          #define FLAG_Z  0x4
8.          ...
9.          flags |= FLAG_X;
10.
```

The D Way

```
    enum FLAGS { X = 0x1, Y = 0x2, Z = 0x4 };
11.         FLAGS flags;
12.         ...
13.         flags |= FLAGS.X;
14.
```

15. Distinguishing between ascii chars and wchar chars:

The C Preprocessor Way

```
    #if UNICODE
16.             #define dchar       wchar_t
17.             #define TEXT(s)     L##s
18.         #else
19.             #define dchar       char
20.             #define TEXT(s)     s
21.         #endif
22.
23.         ...
24.         dchar h[] = TEXT("hello");
25.
```

The D Way

```
    dchar[] h = "hello";
26.
```

27. D's optimizer will inline the function, and will do the conversion of the string constant at compile time.

28. Supporting legacy compilers:

The C Preprocessor Way

```
    #if PROTOTYPES
29.         #define P(p)    p
30.         #else
31.         #define P(p)    ()
32.         #endif
33.         int func P((int x, int y));
34.
```

The D Way

By making the D compiler open source, it will largely avoid the problem of syntactical backwards compatibility.

35.Type aliasing:

The C Preprocessor Way

```
    #define INT     int
36.
```

The D Way

```
    alias int INT;
37.
```

38.Using one header file for both declaration and definition:

The C Preprocessor Way

```
    #define EXTERN extern
39.         #include "declations.h"
40.         #undef EXTERN
41.         #define EXTERN
42.         #include "declations.h"
43.
```

44.In declarations.h:

```
45.         EXTERN int foo;
46.
```

The D Way

The declaration and the definition are the same, so there is no need to muck with the storage class to generate both a declaration and a definition from the same source.

47.Lightweight inline functions:

The C Preprocessor Way

```
    #define X(i)    ((i) = (i) / 3)
48.
```

The D Way

```
    int X(inout int i) { return i = i / 3; }
```

```
49.
```

50.The compiler optimizer will inline it; no efficiency is lost.
51.Assert function file and line number information:

The C Preprocessor Way

```
    #define assert(e)       ((e) || _assert(__LINE__, __FILE__))
52.
```

The D Way

assert() is a built-in expression primitive. Giving the compiler such knowledge of assert() also enables the optimizer to know about things like the _assert() function never returns.
53.Setting function calling conventions:

The C Preprocessor Way

```
    #ifndef _CRTAPI1
54.         #define _CRTAPI1 __cdecl
55.         #endif
56.         #ifndef _CRTAPI2
57.         #define _CRTAPI2 __cdecl
58.         #endif
59.
60.         int _CRTAPI2 func();
61.
```

The D Way

Calling conventions can be specified in blocks, so there's no need to change it for every function:

```
62.         extern (Windows)
63.         {
64.             int onefunc();
65.             int anotherfunc();
66.         }
67.
```

68.Hiding __near or __far pointer wierdness:

The C Preprocessor Way

```
    #define LPSTR   char FAR *
69.
```

The D Way

D doesn't support 16 bit code, mixed pointer sizes, and different kinds of pointers, and so the problem is just irrelevant.

70.Simple generic programming:

The C Preprocessor Way

Selecting which function to use based on text substitution:

```
71.         #ifdef UNICODE
72.         int getValueW(wchar_t *p);
73.         #define getValue getValueW
74.         #else
75.         int getValueA(char *p);
76.         #define getValue getValueA
77.         #endif
78.
```

The D Way

D enables declarations of symbols that are *aliases* of other symbols:

```
79.         version (UNICODE)
80.         {
81.             int getValueW(wchar[] p);
82.             alias getValueW getValue;
83.         }
84.         else
85.         {
86.             int getValueA(char[] p);
87.             alias getValueA getValue;
88.         }
89.
```

Conditional Compilation

The C Preprocessor Way

Conditional compilation is a powerful feature of the C preprocessor, but it has its downside:

The preprocessor has no concept of scope. #if/#endif can be interleaved with code in a completely unstructured and disorganized fashion, making things difficult to follow.

Conditional compilation triggers off of macros - macros that can conflict with identifiers used in the program.

#if expressions are evaluated in subtly different ways than C expressions are.

The preprocessor language is fundamentally different in concept than C, for example,

whitespace and line terminators mean things to the preprocessor that they do not in C.

The D Way

D supports conditional compilation:

1. Separating version specific functionality into separate modules.
2. The debug statement for enabling/disabling debug harnesses, extra printing, etc.
3. The version statement for dealing with multiple versions of the program generated from a single set of sources.
4. The if (0) statement.
5. The /+ +/ nesting comment can be used to comment out blocks of code.

Code Factoring

The C Preprocessor Way

It's common in a function to have a repetitive sequence of code to be executed in multiple places. Performance considerations preclude factoring it out into a separate function, so it is implemented as a macro. For example, consider this fragment from a byte code interpreter:

```
unsigned char *ip;      // byte code instruction pointer
int *stack;
int spi;                // stack pointer
...
#define pop()           (stack[--spi])
#define push(i)         (stack[spi++] = (i))
while (1)
{
    switch (*ip++)
    {
        case ADD:
            op1 = pop();
            op2 = pop();
            result = op1 + op2;
            push(result);
            break;

        case SUB:
            ...
    }
}
```

This suffers from numerous problems:

1. The macros must evaluate to expressions and cannot declare any variables. Consider the difficulty of extending them to check for stack overflow/underflow.
2. The macros exist outside of the semantic symbol table, so remain in scope even outside of the function they are declared in.

3. Parameters to macros are passed textually, not by value, meaning that the macro implementation needs to be careful to not use the parameter more than once, and must protect it with ().
4. Macros are invisible to the debugger, which sees only the expanded expressions.

The D Way

D neatly addresses this with nested functions:

```
ubyte* ip;              // byte code instruction pointer
int[] stack;            // operand stack
int spi;                // stack pointer
...

int pop()         { return stack[--spi]; }
void push(int i) { stack[spi++] = i; }

while (1)
{
    switch (*ip++)
    {
        case ADD:
            op1 = pop();
            op2 = pop();
            push(op1 + op2);
            break;

        case SUB:
        ...
    }
}
```

The problems addressed are:

1. The nested functions have available the full expressive power of D functions. The array accesses already are bounds checked (adjustable by compile time switch).
2. Nested function names are scoped just like any other name.
3. Parameters are passed by value, so need to worry about side effects in the parameter expressions.
4. Nested functions are visible to the debugger.

Additionally, nested functions can be inlined by the implementation resulting in the same high performance that the C macro version exhibits.

#error and Static Asserts

Static asserts are user defined checks made at compile time; if the check fails the compile issues an error and fails.

The C Preprocessor Way

The first way is to use the `#error` preprocessing directive:

```
#if FOO || BAR
    ... code to compile ...
#else
#error "there must be either FOO or BAR"
#endif
```

This has the limitations inherent in preprocessor expressions (i.e. integer constant expressions only, no casts, no `sizeof`, no symbolic constants, etc.).

These problems can be circumvented to some extent by defining a `static_assert` macro (thanks to M. Wilson):

```
#define static_assert(_x) do { typedef int ai[(_x) ? 1 : 0]; }
while(0)
```

and using it like:

```
void foo(T t)
{
    static_assert(sizeof(T) < 4);
    ...
}
```

This works by causing a compile time semantic error if the condition evaluates to false. The limitations of this technique are a sometimes very confusing error message from the compiler, along with an inability to use a `static_assert` outside of a function body.

The D Way

D has the static assert, which can be used anywhere a declaration or a statement can be used. For example:

```
version (FOO)
{
    class Bar
    {
        const int x = 5;
        static assert(Bar.x == 5 || Bar.x == 6);

        void foo(T t)
        {
            static assert(T.size < 4);
            ...
        }
    }
}
else version (BAR)
```

```
        {
            ...
        }
        else
        {
            static assert(0);   // unsupported version
        }
```

Mixins

D mixins superficially look just like using C's preprocessor to insert blocks of code and parse them in the scope of where they are instantiated. But the advantages of mixins over macros are:

1. Mixins substitute in parsed declaration trees that pass muster with the language syntax, macros substitute in arbitrary preprocessor tokens that have no organization.
2. Mixins are in the same language. Macros are a separate and distinct language layered on top of C++, with its own expression rules, its own types, its distinct symbol table, its own scoping rules, etc.
3. Mixins are selected based on partial specialization rules, macros have no overloading.
4. Mixins create a scope, macros do not.
5. Mixins are compatible with syntax parsing tools, macros are not.
6. Mixin semantic information and symbol tables are passed through to the debugger, macros are lost in translation.
7. Mixins have override conflict resolution rules, macros just collide.
8. Mixins automatically create unique identifiers as required using a standard algorithm, macros have to do it manually with kludgy token pasting.
9. Mixin value arguments with side effects are evaluated once, macro value arguments get evaluated each time they are used in the expansion (leading to weird bugs).
10. Mixin argument replacements don't need to be 'protected' with parentheses to avoid operator precedence regrouping.
11. Mixins can be typed as normal D code of arbitrary length, multiline macros have to be backslash line-spliced, can't use // to end of line comments, etc.
12. Mixins can define other mixins. Macros cannot create other macros.

D Strings vs C++ Strings

Why have strings built-in to the core language of D rather than entirely in a library as in C++ Strings? What's the point? Where's the improvement?

Concatenation Operator

C++ Strings are stuck with overloading existing operators. The obvious choice for concatenation is += and +. But someone just looking at the code will see + and think "addition". He'll have to

look up the types (and types are frequently buried behind multiple typedef's) to see that it's a string type, and it's not adding strings but concatenating them.

Additionally, if one has an array of floats, is '+' overloaded to be the same as a vector addition, or an array concatenation?

In D, these problems are avoided by introducing a new binary operator ~ as the concatenation operator. It works with arrays (of which strings are a subset). ~= is the corresponding append operator. ~ on arrays of floats would concatenate them, + would imply a vector add. Adding a new operator makes it possible for orthogonality and consistency in the treatment of arrays. (In D, strings are simply arrays of characters, not a special type.)

Interoperability With C String Syntax

Overloading of operators only really works if one of the operands is overloadable. So the C++ string class cannot consistently handle arbitrary expressions containing strings. Consider:

```
const char abc[5] = "world";
string str = "hello" + abc;
```

That isn't going to work. But it does work when the core language knows about strings:

```
const char[5] abc = "world";
char[] str = "hello" ~ abc;
```

Consistency With C String Syntax

There are three ways to find the length of a string in C++:

```
const char abc[] = "world";     :       sizeof(abc)/sizeof(abc[0])-1
                                :       strlen(abc)
string str                      :       str.length()
```

That kind of inconsistency makes it hard to write generic templates. Consider D:

```
char[5] abc = "world";  :       abc.length
char[] str              :       str.length
```

Checking For Empty Strings

C++ strings use a function to determine if a string is empty:

```
string str;
if (str.empty())
        // string is empty
```

In D, an empty string is just null:

```
char[] str;
if (!str)
        // string is empty
```

Resizing Existing String

C++ handles this with the resize() member function:

```
string str;
str.resize(newsize);
```

D takes advantage of knowing that str is a string, and so resizing it is just changing the length property:

```
char[] str;
str.length = newsize;
```

Slicing a String

C++ slices an existing string using a special constructor:

```
string s1 = "hello world";
string s2(s1, 6, 5);            // s2 is "world"
```

D has the array slice syntax, not possible with C++:

```
char[] s1 = "hello world";
char[] s2 = s1[6 .. 11];        // s2 is "world"
```

Slicing, of course, works with any array in D, not just strings.

Copying a String

C++ copies strings with the replace function:

```
string s1 = "hello world";
string s2 = "goodbye      ";
s2.replace(8, 5, s1, 6, 5);     // s2 is "goodbye world"
```

D uses the slice syntax as an lvalue:

```
char[] s1 = "hello world";
char[] s2 = "goodbye      ";
s2[8..13] = s1[6..11];          // s2 is "goodbye world"
```

Conversions to C Strings

This is needed for compatibility with C API's. In C++, this uses the c_str() member function:

```
void foo(const char *);
string s1;
foo(s1.c_str());
```

In D, strings can be implicitly converted to char*:

```
void foo(char *);
```

```
char[] s1;
foo(s1);
```

Note: some will argue that it is a mistake in D to have an implicit conversion from char[] to char*.

Array Bounds Checking

In C++, string array bounds checking for [] is not done. In D, array bounds checking is on by default and it can be turned off with a compiler switch after the program is debugged.

String Switch Statements

Are not possible in C++, nor is there any way to add them by adding more to the library. In D, they take the obvious syntactical forms:

```
switch (str)
{
    case "hello":
    case "world":
        ...
}
```

where str can be any of literal "string"s, fixed string arrays like char[10], or dynamic strings like char[]. A quality implementation can, of course, explore many strategies of efficiently implementing this based on the contents of the case strings.

Filling a String

In C++, this is done with the replace() member function:

```
string str = "hello";
str.replace(1,2,2,'?');         // str is "h??lo"
```

In D, use the array slicing syntax in the natural manner:

```
char[] str = "hello";
str[1..2] = '?';                // str is "h??lo"
```

Value vs Reference

C++ strings, as implemented by STLport, are by value and are 0-terminated. [The latter is an implementation choice, but STLport seems to be the most popular implementation.] This, coupled with no garbage collection, has some consequences. First of all, any string created must make its own copy of the string data. The 'owner' of the string data must be kept track of, because when the owner is deleted all references become invalid. If one tries to avoid the dangling reference problem by treating strings as value types, there will be a lot of overhead of memory allocation, data copying, and memory deallocation. Next, the 0-termination implies that

strings cannot refer to other strings. String data in the data segment, stack, etc., cannot be referred to.

D strings are reference types, and the memory is garbage collected. This means that only references need to be copied, not the string data. D strings can refer to data in the static data segment, data on the stack, data inside other strings, objects, file buffers, etc. There's no need to keep track of the 'owner' of the string data.

The obvious question is if multiple D strings refer to the same string data, what happens if the data is modified? All the references will now point to the modified data. This can have its own consequences, which can be avoided if the copy-on-write convention is followed. All copy-on-write is is that if a string is written to, an actual copy of the string data is made first.

The result of D strings being reference only and garbage collected is that code that does a lot of string manipulating, such as an lzw compressor, can be a lot more efficient in terms of both memory consumption and speed.

Benchmark

Let's take a look at a small utility, wordcount, that counts up the frequency of each word in a text file. In D, it looks like this:

```
import file;

int main (char[][] args)
{
    int w_total;
    int l_total;
    int c_total;
    int[char[]] dictionary;

    printf("   lines   words   bytes file\n");
    for (int i = 1; i < args.length; ++i)
    {
        char[] input;
        int w_cnt, l_cnt, c_cnt;
        int inword;
        int wstart;

        input = cast(char[])file.read(args[i]);

        for (int j = 0; j < input.length; j++)
        {   char c;

            c = input[j];
            if (c == '\n')
                ++l_cnt;
            if (c >= '0' && c <= '9')
            {
            }
            else if (c >= 'a' && c <= 'z' ||
                c >= 'A' && c <= 'Z')
            {
                if (!inword)
```

```
                        {
                            wstart = j;
                            inword = 1;
                            ++w_cnt;
                        }
                    }
                    else if (inword)
                    {   char[] word = input[wstart .. j];

                        dictionary[word]++;
                        inword = 0;
                    }
                    ++c_cnt;
                }
                if (inword)
                {   char[] w = input[wstart .. input.length];
                    dictionary[w]++;
                }
                printf("%8lu%8lu%8lu %.*s\n", l_cnt, w_cnt, c_cnt, args[i]);
                l_total += l_cnt;
                w_total += w_cnt;
                c_total += c_cnt;
            }

            if (args.length > 2)
            {
                printf("--------------------------------------\n%8lu%8lu%8lu
total",
                    l_total, w_total, c_total);
            }

            printf("--------------------------------------\n");

            foreach (char[] word1; dictionary.keys.sort)
            {
                printf("%3d %.*s\n", dictionary[word1], word1);
            }
            return 0;
        }
```

Two people have written C++ implementations using the C++ standard template library, wccpp1 and wccpp2. The input file alice30.txt is the text of "Alice in Wonderland." The D compiler, dmd, and the C++ compiler, dmc, share the same optimizer and code generator, which provides a more apples to apples comparison of the efficiency of the semantics of the languages rather than the optimization and code generator sophistication. Tests were run on a Win XP machine. dmc uses STLport for the template implementation.

Program	Compile	Compile Time	Run	Run Time
D wc	dmd wc -O -release	0.0719	wc alice30.txt >log	0.0326
C++ wccpp1	dmc wccpp1 -o -I \dm\stlport\stlport	2.1917	wccpp1 alice30.txt >log	0.0944

C++ wccpp2	dmc wccpp2 -o -I \dm\stlport\stlport	2.0463	wccpp2 alice30.txt >log	0.1012

The following tests were run on linux, again comparing a D compiler (gdc) and a C++ compiler (**g++**) that share a common optimizer and code generator. The system is Pentium III 800MHz running RedHat Linux 8.0 and gcc 3.4.2. The Digital Mars D compiler for linux (**dmd**) is included for comparison.

Program	Compile	Compile Time	Run	Run Time
D wc	gdc -O2 -frelease -o wc wc.d	0.326	wc alice30.txt > /dev/null	0.041
D wc	dmd wc -O -release	0.235	wc alice30.txt > /dev/null	0.041
C++ wccpp1	g++ -O2 -o wccpp1 wccpp1.cc	2.874	wccpp1 alice30.txt > /dev/null	0.086
C++ wccpp2	g++ -O2 -o wccpp2 wccpp2.cc	2.886	wccpp2 alice30.txt > /dev/null	0.095

These tests compare gdc with g++ on a PowerMac G5 2x2.0GHz running MacOS X 10.3.5 and gcc 3.4.2. (Timings are a little less accurate.)

Program	Compile	Compile Time	Run	Run Time
D wc	gdc -O2 -frelease -o wc wc.d	0.28	wc alice30.txt > /dev/null	0.03
C++ wccpp1	g++ -O2 -o wccpp1 wccpp1.cc	1.90	wccpp1 alice30.txt > /dev/null	0.07
C++ wccpp2	g++ -O2 -o wccpp2 wccpp2.cc	1.88	wccpp2 alice30.txt > /dev/null	0.08

---

wccpp2 by Allan Odgaard

```
#include <algorithm>
#include <cstdio>
#include <fstream>
#include <iterator>
#include <map>
#include <vector>

bool isWordStartChar (char c)   { return isalpha(c); }
bool isWordEndChar (char c)     { return !isalnum(c); }

int main (int argc, char const* argv[])
{
    using namespace std;
    printf("Lines Words Bytes File:\n");

    map<string, int> dict;
    int tLines = 0, tWords = 0, tBytes = 0;
    for(int i = 1; i < argc; i++)
```

```
            {
                ifstream file(argv[i]);
                istreambuf_iterator<char> from(file.rdbuf()), to;
                vector<char> v(from, to);
                vector<char>::iterator first = v.begin(), last = v.end(),
bow, eow;

                int numLines = count(first, last, '\n');
                int numWords = 0;
                int numBytes = last - first;

                for(eow = first; eow != last; )
                {
                    bow = find_if(eow, last, isWordStartChar);
                    eow = find_if(bow, last, isWordEndChar);
                    if(bow != eow)
                        ++dict[string(bow, eow)], ++numWords;
                }

                printf("%5d %5d %5d %s\n", numLines, numWords, numBytes,
argv[i]);

                tLines += numLines;
                tWords += numWords;
                tBytes += numBytes;
            }

            if(argc > 2)
                    printf("----------------------\n%5d %5d %5d\n", tLines,
tWords, tBytes);
            printf("----------------------\n\n");

            for(map<string, int>::const_iterator it = dict.begin(); it !=
dict.end(); ++it)
                    printf("%5d %s\n", it->second, it->first.c_str());

            return 0;
        }
```

D Complex Types and C++ std::complex

How do D's complex numbers compare with C++'s std::complex class?

Syntactical Aesthetics

In C++, the complex types are:

```
complex<float>
complex<double>
complex<long double>
```

C++ has no distinct imaginary type. D has 3 complex types and 3 imaginary types:

```
cfloat
cdouble
creal
ifloat
idouble
ireal
```

A C++ complex number can interact with an arithmetic literal, but since there is no imaginary type, imaginary numbers can only be created with the constructor syntax:

```
complex<long double> a = 5;              // a = 5 + 0i
complex<long double> b(0,7);             // b = 0 + 7i
c = a + b + complex<long double>(0,7);   // c = 5 + 14i
```

In D, an imaginary numeric literal has the 'i' suffix. The corresponding code would be the more natural:

```
creal a = 5;            // a = 5 + 0i
ireal b = 7i;           // b = 7i
c = a + b + 7i;         // c = 5 + 14i
```

For more involved expressions involving constants:

```
c = (6 + 2i - 1 + 3i) / 3i;
```

In C++, this would be:

```
c = (complex<double>(6,2) + complex<double>(-1,3)) /
complex<double>(0,3);
```

or if an imaginary class were added to C++ it might be:

```
c = (6 + imaginary<double>(2) - 1 + imaginary<double>(3))) /
imaginary<double>(3);
```

In other words, an imaginary number *nn* can be represented with just *nn*i rather than writing a constructor call complex<long double>(0,*nn*).

Efficiency

The lack of an imaginary type in C++ means that operations on imaginary numbers wind up with a lot of extra computations done on the 0 real part. For example, adding two imaginary numbers in D is one add:

```
ireal a, b, c;
c = a + b;
```

In C++, it is two adds, as the real parts get added too:

```
c.re = a.re + b.re;
c.im = a.im + b.im;
```

Multiply is worse, as 4 multiplies and two adds are done instead of one multiply:

```
c.re = a.re * b.re - a.im * b.im;
c.im = a.im * b.re + a.re * b.im;
```

Divide is the worst - D has one divide, whereas C++ implements complex division with typically one comparison, 3 divides, 3 multiplies and 3 additions:

```
if (fabs(b.re) < fabs(b.im))
{
    r = b.re / b.im;
    den = b.im + r * b.re;
    c.re = (a.re * r + a.im) / den;
    c.im = (a.im * r - a.re) / den;
}
else
{
    r = b.im / b.re;
    den = b.re + r * b.im;
    c.re = (a.re + r * a.im) / den;
    c.im = (a.im - r * a.re) / den;
}
```

To avoid these efficiency concerns in C++, one could simulate an imaginary number using a double. For example, given the D:

```
cdouble c;
idouble im;
c *= im;
```

it could be written in C++ as:

```
complex<double> c;
double im;
c = complex<double>(-c.imag() * im, c.real() * im);
```

but then the advantages of complex being a library type integrated in with the arithmetic operators have been lost.

Semantics

Worst of all, the lack of an imaginary type can cause the wrong answer to be inadvertently produced. To quote Prof. Kahan:

> "A streamline goes astray when the complex functions SQRT and LOG are implemented, as is necessary in Fortran and in libraries currently distributed with C/C++ compilers, in a way that disregards the sign of 0.0 in IEEE 754 arithmetic and consequently violates identities like SQRT(CONJ(Z)) = CONJ(SQRT(Z)) and LOG(CONJ(Z)) = CONJ(LOG(Z)) whenever the COMPLEX variable Z takes negative real values. Such anomalies are unavoidable if Complex Arithmetic operates on pairs (x, y) instead of notional sums x + i*y of real and imaginary variables. The language of pairs is *incorrect* for Complex Arithmetic; it needs the Imaginary type."

The semantic problems are:

> Consider the formula (1 - infinity*i) * i which should produce (infinity + i). However, if instead the second factor is (0 + i) rather than just i, the result is (infinity + NaN*i), a spurious NaN was generated.
>
> A distinct imaginary type preserves the sign of 0, necessary for calculations involving branch cuts.

Appendix G of the C99 standard has recommendations for dealing with this problem. However, those recommendations are not part of the C++98 standard, and so cannot be portably relied upon.

References

How Java's Floating-Point Hurts Everyone Everywhere Prof. W. Kahan and Joseph D. Darcy

The Numerical Analyst as Computer Science Curmudgeon by Prof. W. Kahan

"Branch Cuts for Complex Elementary Functions, or Much Ado About Nothing's Sign Bit" by W. Kahan, ch.
7 in The State of the Art in Numerical Analysis (1987) ed. by M. Powell and A. Iserles for Oxford U.P.

D's Contract Programming vs C++'s

Many people have written me saying that D's Contract Programming (DbC) does not add anything that C++ does not already support. They go on to illustrate their point with a technique for doing DbC in C++.

It makes sense to review what DbC is, how it is done in D, and stack that up with what each of the various C++ DbC techniques can do.

Digital Mars C++ adds extensions to C++ to support DbC, but they are not covered here because

they are not part of standard C++ and are not supported by any other C++ compiler.

Contract Programming in D

This is more fully documented in the D Contract Programming document. To sum up, DbC in D has the following characteristics:

1. The *assert* is the basic contract.
2. When an assert contract fails, it throws an exception. Such exceptions can be caught and handled, or allowed to terminate the program.
3. Classes can have *class invariants* which are checked upon entry and exit of each public class member function, the exit of each constructor, and the entry of the destructor.
4. Assert contracts on object references check the class invariant for that object.
5. Class invariants are inherited, that means that a derived class invariant will implicitly call the base class invariant.
6. Functions can have *preconditions* and *postconditions*.
7. For member functions in a class inheritance hierarchy, the precondition of a derived class function are OR'd together with the preconditions of all the functions it overrides. The postconditions are AND'd together.
8. By throwing a compiler switch, DbC code can be enabled or can be withdrawn from the compiled code.
9. Code works semantically the same with or without DbC checking enabled.

Contract Programming in C++

The `assert` Macro

C++ does have the basic `assert` macro, which tests its argument and if it fails, aborts the program. `assert` can be turned on and off with the `NDEBUG` macro.

`assert` does not know anything about class invariants, and does not throw an exception when it fails. It just aborts the program after writing a message. `assert` relies on a macro text preprocessor to work.

`assert` is where explicit support for DbC in Standard C++ begins and ends.

Class Invariants

Consider a class invariant in D:

```
class A
{
    invariant() { ...contracts... }

    this() { ... }      // constructor
    ~this() { ... }     // destructor
```

```
    void foo() { ... }  // public member function
}

class B : A
{
    invariant() { ...contracts... }
    ...
}
```

To accomplish the equivalent in C++ (thanks to Bob Bell for providing this):

```
template
inline void check_invariant(T& iX)
{
#ifdef DBC
    iX.invariant();
#endif
}

// A.h:

class A {
    public:
#ifdef DBG
       virtual void invariant() { ...contracts... }
#endif
       void foo();
};

// A.cpp:

void A::foo()
{
    check_invariant(*this);
    ...
    check_invariant(*this);
}

// B.h:

#include "A.h"

class B : public A {
    public:
#ifdef DBG
        virtual void invariant()
        { ...contracts...
           A::invariant();
        }
#endif
       void bar();
};

// B.cpp:

void B::barG()
```

```
{
    check_invariant(*this);
    ...
    check_invariant(*this);
}
```

There's an additional complication with `A::foo()`. Upon every normal exit from the function, the `invariant()` should be called. This means that code that looks like:

```
int A::foo()
{
    ...
    if (...)
        return bar();
    return 3;
}
```

would need to be written as:

```
int A::foo()
{
    int result;
    check_invariant(*this);
    ...
    if (...)
    {
        result = bar();
        check_invariant(*this);
        return result;
    }
    check_invariant(*this);
    return 3;
}
```

Or recode the function so it has a single exit point. One possibility to mitigate this is to use RAII techniques:

```
int A::foo()
{
#if DBC
    struct Sentry {
       Sentry(A& iA) : mA(iA) { check_invariants(iA); }
       ~Sentry() { check_invariants(mA); }
       A& mA;
    } sentry(*this);
#endif
    ...
    if (...)
        return bar();
    return 3;
}
```

The #if DBC is still there because some compilers may not optimize the whole thing away if check_invariants compiles to nothing.

Preconditions and Postconditions

Consider the following in D:

```
void foo()
    in { ...preconditions... }
    out { ...postconditions... }
    body
    {
        ...implementation...
    }
```

This is nicely handled in C++ with the nested Sentry struct:

```
void foo()
{
    struct Sentry
    {   Sentry() { ...preconditions... }
        ~Sentry() { ...postconditions... }
    } sentry;
    ...implementation...
}
```

If the preconditions and postconditions consist of nothing more than `assert` macros, the whole doesn't need to be wrapped in a `#ifdef` pair, since a good C++ compiler will optimize the whole thing away if the `asserts` are turned off.

But suppose `foo()` sorts an array, and the postcondition needs to walk the array and verify that it really is sorted. Now the shebang needs to be wrapped in `#ifdef`:

```
void foo()
{
#ifdef DBC
    struct Sentry
    {   Sentry() { ...preconditions... }
        ~Sentry() { ...postconditions... }
    } sentry;
#endif
    ...implementation...
}
```

(One can make use of the C++ rule that templates are only instantiated when used can be used to avoid the `#ifdef`, by putting the conditions into a template function referenced by the `assert`.)

Let's add a return value to `foo()` that needs to be checked in the postconditions. In D:

```
int foo()
```

```
    in { ...preconditions... }
    out (result) { ...postconditions... }
    body
    {
        ...implementation...
        if (...)
            return bar();
        return 3;
    }
```

In C++:

```
int foo()
{
#ifdef DBC
    struct Sentry
    {   int result;
        Sentry() { ...preconditions... }
        ~Sentry() { ...postconditions... }
    } sentry;
#endif
    ...implementation...
    if (...)
    {   int i = bar();
#ifdef DBC
        sentry.result = i;
#endif
        return i;
    }
#ifdef DBC
    sentry.result = 3;
#endif
    return 3;
}
```

Now add a couple parameters to `foo()`. In D:

```
int foo(int a, int b)
    in { ...preconditions... }
    out (result) { ...postconditions... }
    body
    {
        ...implementation...
        if (...)
            return bar();
        return 3;
    }
```

In C++:

```
int foo(int a, int b)
{
#ifdef DBC
    struct Sentry
```

```
        {   int a, b;
            int result;
            Sentry(int a, int b)
            {   this->a = a;
                this->b = b;
                ...preconditions...
            }
            ~Sentry() { ...postconditions... }
        } sentry(a,b);
    #endif
        ...implementation...
            if (...)
            {   int i = bar();
    #ifdef DBC
                sentry.result = i;
    #endif
                return i;
            }
    #ifdef DBC
            sentry.result = 3;
    #endif
            return 3;
    }
```

Preconditions and Postconditions for Member Functions

Consider the use of preconditions and postconditions for a polymorphic function in D:

```
class A
{
    void foo()
        in { ...Apreconditions... }
        out { ...Apostconditions... }
        body
        {
            ...implementation...
        }
}

class B : A
{
    void foo()
        in { ...Bpreconditions... }
        out { ...Bpostconditions... }
        body
        {
            ...implementation...
        }
}
```

The semantics for a call to `B.foo()` are:

Either Apreconditions or Bpreconditions must be satisfied.

Both Apostconditions and Bpostconditions must be satisfied.
Let's get this to work in C++:

```
class A
{
protected:
    #if DBC
    int foo_preconditions() { ...Apreconditions... }
    void foo_postconditions() { ...Apostconditions... }
    #else
    int foo_preconditions() { return 1; }
    void foo_postconditions() { }
    #endif

    void foo_internal()
    {
        ...implementation...
    }

public:
    virtual void foo()
    {
        foo_preconditions();
        foo_internal();
        foo_postconditions();
    }
};

class B : A
{
protected:
    #if DBC
    int foo_preconditions() { ...Bpreconditions... }
    void foo_postconditions() { ...Bpostconditions... }
    #else
    int foo_preconditions() { return 1; }
    void foo_postconditions() { }
    #endif

    void foo_internal()
    {
        ...implementation...
    }

public:
    virtual void foo()
    {
        assert(foo_preconditions() || A::foo_preconditions());
        foo_internal();
        A::foo_postconditions();
        foo_postconditions();
    }
};
```

Something interesting has happened here. The preconditions can no longer be done using

`assert`, since the results need to be OR'd together. I'll leave as a reader exercise adding in a class invariant, function return values for `foo()`, and parameters for `foo()`.

Conclusion

These C++ techniques can work up to a point. But, aside from `assert`, they are not standardized and so will vary from project to project. Furthermore, they require much tedious adhesion to a particular convention, and add significant clutter to the code. Perhaps that's why it's rarely seen in practice.

By adding support for DbC into the language, D offers an easy way to use DbC and get it right. Being in the language standardizes the way it will be used from project to project.

References

Chapter C.11 introduces the theory and rationale of Contract Programming in Object-Oriented Software Construction
Bertrand Meyer, Prentice Hall

Chapters 24.3.7.1 to 24.3.7.3 discuss Contract Programming in C++ in The C++ Programming Language Special Edition
Bjarne Stroustrup, Addison-Wesley

D Links

These are links to other **D** resources!

Japanese Language D sites
The Digital Mars D site translated to Japanese.
A tutorial of the D language.
D wiki site
Using using Direct3D9 in the D language.
Windows header files.
d_cpp, a tool help to access C++ libraries from D.
Some import libraries.
barrage 360 deg., a viewer of BulletML. A sample of d_cpp.
TUMIKI Fighters game implemented in D.
A small sample game.
Shinchiro.h's Japanese D resource page.
Open Source Development for D at dsource.org. Project hosting, discussion forums, tutorials.
Check for a D user's group in your community.
Manfred Hansen's D MySQL Binding for Linux.

cURL for D at www.atari-soldiers.com/libcurl.html.
Extend or embed Python using D at http://www.scratch-ware.net/D/.
Old school chase action game A7Xpg, written in D.
Andy Friesen has taken SWIG and modified it to generate code for D.
Windows Developer Network's Fall 2003 issue compares **D** with C, C++, C# and Java.
Ant has released the **D** graphical **U**ser **I**nterface (DUI) toolkit.
The latest version of Zeus for Windows is finally out. This release comes with D compiler (*) and keywords predefined and it also has a ctags program (xtags.exe) that supports the D language. (*) For the D compiler to work within Zeus the user will need to download and install the compiler from www.digitalmars.com/d/. If it is already installed it should be automatic.
KDE Linux Support for D
Mike Wynn's D Win32 COM libraries.
The wiki for the **D** programming language: Wiki4D.
The wiki for the D's runtime library Phobos.
The DIDE from Atari-Soldiers.
Program editors customized for use with **D**.
Simple URL loading library by Burton Radons. Requires the DIG library to be installed, although it doesn't use it, just digc. Comes with documentation. It has the functions:
urlopen: Open a URL as a stream (http, file, and ftp schema supported).
urlread: Open a URL and read its contents.
urllistdir: List a directory, return an array of URLStat (file and ftp schema supported).
urlencode, urldecode: Encode and decode the URL. The above functions expect a decoded URL.
Why **D** isn't Java? By Daniel Yokomiso.
Burton Radons has prepared a linux port of **D**. This is a very useful guide for anyone wanting to build a new code generator for D for a different processor.
minddrome networks.
Pavel's DedicateD site. Lots of **D** projects with source code, FAQs, **D** news, etc.
Dr. Dobb's February 2002 has a cover story on **D**.
D's appearance on slashdot.
A page on **D** on the wikiwiki, where computer languages get discussed. See Dee Language.
The Pragmatic Programmers
Gnu **D** compiler project on SourceForge. Come join and make Gnu **D** a reality!
The **D** Journal (coming soon!)
The Code Moon
The OpenD project.
99 Bottles.
OO Programming Newsletter #30 from Bruce Eckel.
Open Directory: Computers: Programming: Languages: D.
DDevil's language shootout D benchmarks.
DServicesAPI
Lua API for D.
Andy's D Page

SynSoft's D Page provides a number of free D libraries. SynSoft is an imprint of Synesis Software, which provides free D, Java, .NET, Perl and Python libraries. SynSoft has contributed several modules to the D standard library, and is currently working on a standard template library - the DTL.
Dprogramming.com
DML, embedding D in HTML.
Felix writes: "This archive contains a Windows shell for the DMD, DMC, BCC compilers. It is also extensible for all kinds of compilers. The configuration file syntax is very strict, but it works. A kind of Kriate's shell, but even more advanced."
Justin's D Stuff
Joel Anderson's D Page contains dig modified to support .81 and a GLee port which supports around 300 openGL extensions.
Stephan Wienczny's D Programming page.
Sam McCall's class-based string library and documentation.
MKoD - D Programming Language (MKoD == Magikal Kingdom of Dreams)
Always expanding, currently the site contains projects (ex: like the Financial Pkg), examples (in plain D code, or a mix of D with Windows APIs), converted Win32 APIs from C to D (ex: ODBC32 and WinCon), technical references (D datatypes / size chart), general information (ex: how to install D and D programming 101), and D site banners.
Deja's D.NET PreAlpha Release.

If you have any **D** code, documents, or web pages of interest to **D** programmers, please email the links to:

Cut & paste the following free images on your web pages of interest to **D** programmers:

Corto's **D** button images: 

If you'd like to contribute more images, please email them to Digital Mars!