

D Programming Language

The 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 successor to 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!

The D newsgroup in news.digitalmars.com server is where discussions of this should go. Suggestions, criticism, kudos, flames, etc., are all welcome there.

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	11
What is D?	11
Why D?	11
Features To Keep From C/C++	12
Features To Drop	13
Who D is For	14
Who D is Not For	14
Major Features of D	15
Object Oriented Programming	15
Productivity	15
Functions	17
Arrays	17
Resource Management	18
Performance	18
Reliability	19
Compatibility	20
Project Management	21
Sample D Program (sieve.d)	21
Lexical	23
Phases of Compilation	23
Source Text	23
End of File	24
End of Line	24
White Space	24
Comments	24
Identifiers	25
String Literals	25
Integer Literals	27
Floating Literals	28
Keywords	29
Tokens	30
Pragmas	31
Modules	33
Module Declaration	33
Import Declaration	34
Scope and Modules	34
Static Construction and Destruction	35
Order of Static Construction	35
Order of Static Construction within a Module	35
Order of Static Destruction	35
Declarations	36
Declaration Syntax	36
Type Defining	37
Type Aliasing	37
Alias Declarations	37
Types	39
Basic Data Types	39
Derived Data Types	39

User Defined Types.....	40
Pointer Conversions	40
Implicit Conversions	40
Integer Promotions	40
Usual Arithmetic Conversions	40
Delegates	41
Properties.....	42
Properties for Integral Data Types	42
Properties for Floating Point Types	42
.init Property.....	42
Attributes	44
Linkage Attribute	45
Align Attribute	45
Deprecated Attribute	46
Protection Attribute	46
Const Attribute	46
Override Attribute	46
Static Attribute	47
Auto Attribute	47
Expressions.....	49
Evaluation Order	51
Expressions.....	51
Assign Expressions	51
Assignment Operator Expressions	51
Conditional Expressions.....	52
OrOr Expressions	52
AndAnd Expressions.....	52
Bitwise Expressions	53
Or Expressions	53
Xor Expressions	53
And Expressions.....	53
Equality Expressions	53
Identity Expressions	53
Relational Expressions	54
Integer comparisons	55
Floating point comparisons	55
In Expressions	56
Shift Expressions.....	56
Add Expressions.....	57
Mul Expressions.....	57
Unary Expressions.....	57
New Expressions	58
Cast Expressions	58
Postfix Expressions	59
Primary Expressions.....	59
this	59
super	59
null.....	59
true, false	59
Function Literals	59
Assert Expressions	60

Statements	61
Labelled Statements	62
Block Statement	62
Expression Statement	63
Declaration Statement	63
If Statement	63
While Statement	63
Do-While Statement	64
For Statement	64
Switch Statement	65
Continue Statement	66
Break Statement	67
Return Statement	67
Goto Statement	67
With Statement	68
Synchronize Statement	68
Try Statement	69
Throw Statement	69
Volatile Statement	69
Asm Statement	70
Arrays	72
Pointers	72
Static Arrays	72
Dynamic Arrays	72
Array Declarations	72
Usage	73
Slicing	73
Array Copying	74
Array Setting	74
Array Concatenation	74
Array Operations	75
Rectangular Arrays	76
Array Properties	76
Setting Dynamic Array Length	77
Array Bounds Checking	78
Array Initialization	78
Static Initialization of Static Arrays	78
Special Array Types	79
Arrays of Bits	79
Strings	79
Associative Arrays	80
Properties	81
Associative Array Example: word count	81
Structs, Unions, Enums	83
Structs, Unions	83
Static Initialization of Structs	83
Static Initialization of Unions	83
Enums	84
Enum Properties	85
Initialization of Enums	85
Classes	86

Fields	87
Super Class	87
Constructors	88
Destructors	89
Static Constructors	90
Static Destructor	91
Class Invariants	91
Unit Tests	92
Class Allocators.....	92
Class Deallocators	93
Auto Classes	93
Interfaces	93
Functions	97
Virtual Functions.....	97
Inline Functions.....	97
Function Overloading.....	97
Function Parameters.....	97
Local Variables	98
Nested Functions	98
Delegates, Function Pointers, and Dynamic Closures	101
Operator Overloading.....	103
Unary Operator Overloading.....	103
Overloadable Unary Operators.....	103
Overloading ++e and --e	103
Examples	103
Binary Operator Overloading.....	103
Overloadable Binary Operators.....	103
Overloading == and !=	105
Overloading <, <=, > and >=.....	105
Future Directions.....	106
Templates	107
Instantiation Scope	108
Argument Deduction	109
Value Parameters.....	110
Specialization	110
Limitations	110
Contracts.....	111
Assert Contract	111
Pre and Post Contracts.....	111
In, Out and Inheritance.....	113
Class Invariants	113
Debug and Version.....	114
Predefined Versions	114
Specification.....	115
Debug Statement	115
Version Statement	116
Debug Attribute.....	116
Version Attribute.....	117
Error Handling in D.....	119
The Error Handling Problem.....	119
The D Error Handling Solution.....	120

Garbage Collection.....	122
How Garbage Collection Works	123
Interfacing Garbage Collected Objects With Foreign Code	123
Pointers and the Garbage Collector.....	123
Working with the Garbage Collector	124
Memory Management	125
Strings (and Array) Copy-on-Write	125
Real Time	126
Smooth Operation	126
Free Lists	126
Reference Counting.....	127
Explicit Class Instance Allocation	127
Mark/Release.....	129
RAII (Resource Acquisition Is Initialization)	130
Allocating Class Instances On The Stack	130
Floating Point	131
Floating Point Intermediate Values.....	131
Complex and Imaginary types.....	131
Rounding Control.....	132
Exception Flags	132
Floating Point Comparisons	132
D x86 Inline Assembler.....	133
Labels	133
align IntegerExpression.....	133
even	134
naked	134
db, ds, di, dl, df, dd, de	134
Opcodes	134
Special Cases.....	135
Operands.....	135
Operand Types	136
Struct/Union/Class Member Offsets	137
Special Symbols	137
Opcodes Supported	137
AMD Opcodes Supported	141
Interfacing to C.....	142
Calling C Functions.....	142
Storage Allocation.....	143
Data Type Compatibility.....	143
Calling printf().....	144
Structs and Unions	144
Interfacing to C++	145
Portability Guide	146
OS Specific Code	146
Embedding D in HTML	147
D Runtime Model.....	148
Object Model.....	148
Array Model	148
Phobos	150
D Runtime Library	150
Philosophy.....	150

Imports	150
Core D: Available on all D implementations	151
Standard C: interface to C functions	151
Operating System and Hardware: platform specific	151
compiler.....	152
conv	152
ctype	152
date	153
file.....	154
gc	154
intrinsic.....	155
math.....	157
object	160
outbuffer	160
path	161
process	162
random.....	162
regexp.....	163
stdint.....	164
stream	165
Reading.....	165
Writing	166
Seeking	167
string.....	169
To copy or not to copy?.....	169
system.....	171
thread	171
zip	172
stdio	172
D for Win32	173
Calling Conventions.....	173
Windows Executables	173
DLLs (Dynamic Link Libraries)	174
Memory Allocation	175
COM Programming.....	176
D vs Other Languages	177
Notes.....	179
Programming in D for C Programmers	180
Getting the Size of a Type.....	181
Get the max and min values of a type	181
Primitive Types	181
Special Floating Point Values	182
Taking the Modulus of a floating point number	182
Dealing with NAN's in floating point compares	183
Assert's are a necessary part of any good defensive coding strategy.....	183
Initializing all elements of an array.....	184
Looping through an array	184
Creating an array of variable size.....	184
String Concatenation	185
Formatted printing.....	186
Forward referencing functions	186

Functions that have no arguments	186
Labelled break's and continue's.	187
Goto Statements	187
Struct tag name space	188
Looking up strings.....	188
Setting struct member alignment.....	189
Anonymous Structs and Unions	189
Declaring struct types and variables.....	190
Getting the offset of a struct member.....	190
Union initializations.	191
Struct initializations.....	191
Array initializations.....	191
Escaped String Literals.....	192
Ascii vs Wide Characters	192
Arrays that parallel an enum	193
Creating a new typedef'd type	193
Comparing structs	194
Comparing strings	195
Sorting arrays	196
Volatile memory access	196
String literals	196
Data Structure Traversal.....	197
Programming in D for C++ Programmers.....	199
Defining constructors	199
Base class initialization	200
Comparing structs	200
Creating a new typedef'd type	201
Friends	202
Operator overloading.....	203
Namespace using declarations.....	204
RAII (Resource Acquisition Is Initialization)	204
Dynamic Closures	205
The C Preprocessor Versus D	207
Header Files.....	207
#pragma once	207
#pragma pack	208
Macros	208
Conditional Compilation	212
Code Factoring	213
The D Style.....	215
White Space.....	215
Comments.....	215
Naming Conventions	215
Meaningless Type Aliases.....	216
Declaration Style	216
Operator Overloading.....	216
Hungarian Notation	216
Example: wc	217
Compiler for D Programming Language	219
Files	219
Requirements.....	219

Installation.....	219
Example.....	219
Compiler Arguments and Switches.....	219
Linking.....	220
Environment Variables.....	221
SC.INI Initialization File.....	221
Bugs.....	221
Feedback.....	221
Acknowledgements.....	222

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! The reality of the C++ compiler business is that few compilers effectively implement the entire standard.

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 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.

It's frustrating that such a powerful language does not do basic things like resizing arrays and concatenating strings. Yes, C++ does provide the meta programming ability to implement resizable arrays and strings like the vector type in the STL. Such fundamental features, however, ought to be part of the language. 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.

D aims to 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.

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 RAI 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. Similarly, 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[])`.
- 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

Design by Contract (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 inadvertently 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 design by contract, 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.

```
synchronize 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
- Explicit initialization required
- Unused local variables not allowed
- 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 design by contract 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 */

import c.stdio;

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 analyser. This separation of the passes makes language tools like syntax directed editors relatively easy to produce.

1. **ascii/wide char**
The source file is checked to see if it is in ASCII or wide characters, and the appropriate scanner is loaded.
2. **lexical analysis**
The source file is divided up into a sequence of tokens. Pragmas 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**
6. **code generation**

Source Text

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

- UTF-8
- UTF-16BE
- UTF-16LE
- UTF-32BE
- UTF-32LE

Note that UTF-8 is a superset of traditional 7-bit ASCII. The source text is assumed to be in UTF-8, unless one of the following BOMs (Byte Order Marks) is 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
UTF-8	none of the above

There are no digraphs or trigraphs in D. 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.

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

Identifiers

```
Identifier:
    IdentifierStart
    IdentifierStart IdentifierChars

IdentifierChars:
    IdentifierChar
    IdentifierChar IdentifierChars

IdentifierStart:
    _
    Letter

IdentifierChar:
    IdentifierStart
    Digit
```

Identifiers start with a letter or `_`, and are followed by any number of letters, `_` or digits. Identifiers can be arbitrarily long, and are case sensitive. Identifiers starting with `__` are reserved.

String Literals

```
StringLiteral:
    SingleQuotedString
    DoubleQuotedString
    EscapeSequence

SingleQuotedString:
    ' SingleQuotedCharacters '
```

```
SingleQuotedCharacter:
    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

```

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

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

```

'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
\r\n        carriage return, line feed

```

Escape sequences not listed above are errors.

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"
'ab' 'c'
'a' "bc"
"a" ~ "b" ~ "c"
\0x61"bc"

```

Integer Literals

```
IntegerLiteral:
    Integer
    Integer IntegerSuffix

Integer:
    Decimal
    Binary
    Octal
    Hexadecimal

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

Decimal:
    0
    NonZeroDigit
    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 be immediately followed by one 'l' or one 'u' or both.

The type of the integer is resolved as follows:

1. If it is decimal it is the last representable of `ulong`, `long`, or `int`.
2. If it is not decimal, it is the last representable of `ulong`, `long`, `uint`, or `int`.
3. If it has the 'u' suffix, it is the last representable of `ulong` or `uint`.
4. If it has the 'l' suffix, it is the last representable of `ulong` or `long`.
5. If it has the 'u' and 'l' suffixes, it is `ulong`.

Floating Literals

```

FloatLiteral:
    Float
    Float FloatSuffix
    Float ImaginarySuffix
    Float FloatSuffix ImaginarySuffix

Float:
    DecimalFloat
    HexFloat

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.

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.FFFFFFFFFFFFFFFFp1023    // 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
cent
char
class
cfloat
cdouble
creal
const
continue

debug
default
delegate
delete
deprecated
do
double

else
enum
export
extern

false
final
finally
float
for
function

super
null
new
short
int
long
ifloat
idouble
ireal
if
switch
synchronized
return
goto
struct
```

interface
import
static
override
in
out
inout
private
protected
public
invariant
real

this
throw
true
try
typedef

ubyte
ucent
uint
ulong
union
ushort

version
void
volatile

wchar
while
with

Tokens

Token:

Identifier
StringLiteral
IntegerLiteral
FloatLiteral
Keyword
/
/=
.
..
...
&
&=
&&
|
|=
||
-
-=
--
+
+=
++
<
<=
<<

```

<<=
<>
<>=
>
>=
>>=
>>>=
>>
>>>
!
!=
!==
!<>
!<>=
!<
!<=
!>
!>=
(
)
[
]
{
}
?
,
;
:
$
=
==
===
*
*=
%
%=
^
^=
~
~=

```

Pragmas

Pragmas are special token sequences that give instructions to the compiler. Pragmas are processed by the lexical analyzer, may appear between any other tokens, and do not affect the syntax parsing.

There is currently only one pragma, the `#line` pragma.

```

Pragma
    # 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.

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 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, 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
```

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:

```
const Decl
static Decl
final Decl
synchronized Decl
deprecated Decl
BasicType BasicType2 Declarators ;
BasicType BasicType2 FunctionDeclarator
```

Declarators:

```
Declarator
Declarator , Declarators
```

Declaration Syntax

Declaration syntax generally reads left to right:

```
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, when lexically next to each other, read right to left:

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

Pointers to functions are declared as subdeclarations:

```
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
```

C-style array declarations, where the [] appear to the right of the identifier, 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
```

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; }
instance Foo2(int) t1; // a TemplateAliasDeclaration
alias instance Foo2(int).t t2;
alias t1.t t3;
alias t2 t4;
alias instance Foo2(int) t5;

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
t5.t v5;           // v5 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;
```

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.

Types

Basic Data Types

void	no type
bit	single bit
byte	signed 8 bits
ubyte	unsigned 8 bits
short	signed 16 bits
ushort	unsigned 16 bits
int	signed 32 bits
uint	unsigned 32 bits
long	signed 64 bits
ulong	unsigned 64 bits
cent	signed 128 bits (reserved for future use)
ucent	unsigned 128 bits (reserved for future use)
float	32 bit floating point
double	64 bit floating point
real	largest hardware implemented floating point size (Implementation Note: 80 bits for Intel CPU's)
ireal	a floating point value with imaginary type
ifloat	imaginary float
idouble	imaginary double
creal	a complex number of two floating point values
cfloat	complex float
cdouble	complex double
char	unsigned 8 bit ASCII
wchar	unsigned Wide char (Implementation Note: 16 bits on Win32 systems, 32 bits on linux, corresponding to C's wchar_t type)

The bit data type is special. It means one binary bit. Pointers or references to a bit are not allowed.

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 not allowed in D. This is to prevent casual manipulation of pointers as integers, as these kinds of practices can play havoc with the garbage collector and in porting code from one machine to another. If it is really, absolutely, positively necessary to do this, use a union, and even then, be very careful that the garbage collector won't get botched by this.

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 = (myint)i;   // OK
```

Integer Promotions

The following types are implicitly converted to `int`:

```
bit
byte
ubyte
short
ushort
enum
```

Typedefs are converted to their underlying type.

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 extended, the other operand is converted to extended.
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:
 1. If both are the same type, no more conversions are done.

2. If both are signed or both are unsigned, the smaller type is converted to the larger.
3. If the signed type is larger than the unsigned type, the unsigned type is converted to the signed type.
4. 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.size           // yields
float.nan         // yields the floating point value
(float).nan       // yields the floating point nan value
(3).size          // yields 4 (because 3 is an int)
2.size           // syntax error, since "2." is a floating point
number
int.init          // default initializer for int's
```

Properties for Integral Data Types

```
.init             initializer (0)
.size            size in bytes
.max             maximum value
.min            minimum value
.sign           should we do this?
```

Properties for Floating Point Types

```
.init             initializer (NaN)
.size            size in bytes
.infinity        infinity value
.nan            NaN value
.sign           1 if -, 0 if +
.isnan          1 if nan, 0 if not
.isinfinite     1 if +-infinity, 0 if not
.isnormal       1 if not nan or infinity, 0 if
.digits         number of digits of precision
.epsilon        smallest increment
.mantissa       number of bits in mantissa
.maxExp maximum  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
```

Attributes

```

AttributeSpecifier:
    Attribute :
    Attribute DeclDefBlock

AttributeElseSpecifier:
    AttributeElse :
    AttributeElse DeclDefBlock
    AttributeElse DeclDefBlock else DeclDefBlock

Attribute:
    LinkageAttribute
    AlignAttribute
    deprecated
    private
    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

next }
attribute:
    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**, **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.

Protected means that only members of the enclosing class or any classes derived from that 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.

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

...

Foo f;
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 RAI (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.

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
AndAndExpression || AndAndExpression
```

AndAndExpression:

```
OrExpression
OrExpression && OrExpression
```

OrExpression:

```
XorExpression
XorExpression | XorExpression
```

XorExpression:

```
AndExpression
AndExpression ^ AndExpression
```

AndExpression:

```
EqualExpression
EqualExpression & EqualExpression
```

EqualExpression:

```
RelExpression
RelExpression == RelExpression
RelExpression != RelExpression
RelExpression === RelExpression
RelExpression !== RelExpression
```

RelExpression:

```
ShiftExpression
ShiftExpression < ShiftExpression
ShiftExpression <= ShiftExpression
ShiftExpression > ShiftExpression
ShiftExpression >= ShiftExpression
ShiftExpression !<>= ShiftExpression
ShiftExpression !<> ShiftExpression
ShiftExpression <> ShiftExpression
ShiftExpression <>= ShiftExpression
ShiftExpression !> ShiftExpression
ShiftExpression !>= ShiftExpression
ShiftExpression !< ShiftExpression
ShiftExpression !<= ShiftExpression
ShiftExpression in ShiftExpression
```

ShiftExpression:

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

AddExpression:

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

MulExpression:

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

UnaryExpression:

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

PostfixExpression:

```
PrimaryExpression
PostfixExpression . Identifier
PostfixExpression ++
PostfixExpression --
PostfixExpression ( ArgumentList )
PostfixExpression [ Expression ]
```

PrimaryExpression:

```
Identifier
this
super
null
true
false
```

```

NumericLiteral
StringLiteral
FunctionLiteral
AssertExpression
Type . Identifier

```

```

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

```
AndAndExpression || 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

```
OrExpression && 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

```
XorExpression | XorExpression
```

The operands are OR'd together.

Xor Expressions

```
AndExpression ^ AndExpression
```

The operands are XOR'd together.

And Expressions

```
EqualExpression & EqualExpression
```

The operands are AND'd together.

Equality Expressions

```
RelExpression == RelExpression  
RelExpression != 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(). Two null objects compare as equal, if only one is null they compare not equal.

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

Identity Expressions

```
RelExpression === RelExpression  
RelExpression !== RelExpression
```

The `===` compares for identity, and `!==` compares for not identity. 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.

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

Relational Expressions

```
ShiftExpression < ShiftExpression
ShiftExpression <= ShiftExpression
ShiftExpression > ShiftExpression
ShiftExpression >= ShiftExpression
ShiftExpression !<>= ShiftExpression
ShiftExpression !<> ShiftExpression
ShiftExpression <> ShiftExpression
ShiftExpression <>= ShiftExpression
ShiftExpression !> ShiftExpression
ShiftExpression !>= ShiftExpression
ShiftExpression !< ShiftExpression
ShiftExpression !<= ShiftExpression
ShiftExpression in 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

ShiftExpression 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.

Shift Expressions

AddExpression << AddExpression
AddExpression >> AddExpression
AddExpression >>> 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

```
MulExpression + MulExpression  
MulExpression - 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 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.

For the + operator, if both operands are arrays of a compatible type, the resulting type is an array of that compatible type, and the resulting value is the concatenation of the two arrays.

Mul Expressions

```
UnaryExpression * UnaryExpression  
UnaryExpression / UnaryExpression  
UnaryExpression % 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.

Unary Expressions

```
& UnaryExpression  
++ UnaryExpression  
-- UnaryExpression  
* UnaryExpression  
- UnaryExpression  
+ UnaryExpression  
! UnaryExpression
```

```

~ UnaryExpression
delete UnaryExpression
NewExpression
( Type ) UnaryExpression
( Type ) . Identifier

```

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(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
}

```

```
}

```

D does not have a Java style instanceof operator, because the cast operator performs the same function:

```
Java:
    if (a instanceof B)

D:
    if ((B) a)
```

Postfix Expressions

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

Primary Expressions

```
Identifier
this
super
null
true
false
NumericLiteral
StringLiteral
FunctionLiteral
AssertExpression
Type . Identifier
```

this

Within a non-static member function, **this** resolves to a reference to the object that called the function.

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.

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.

Function Literals

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

delegate *Type* (*ParameterList*) *FunctionBody*

FunctionLiterals enable embedding anonymous functions directly into expressions. For example:

```
int function(char c) fp;

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; });
}
```

If the *Type* is omitted, it is treated as **void**. 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.

Assert Expressions

AssertExpression:
assert (*Expression*)

Asserts evaluate the *expression*. If the result is false, an `AssertException` 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 Design by Contract support in D.

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*
- SwitchStatement*
- CaseStatement*
- DefaultStatement*
- ContinueStatement*
- BreakStatement*
- ReturnStatement*
- GotoStatement*
- WithStatement*
- SynchronizeStatement*
- TryStatement*
- ThrowStatement*
- VolatileStatement*
- AsmStatement*

- Labeled Statements
- Block Statement
- Expression Statement
- Declaration Statement
- If Statement
- Debug Statement
- Version Statement
- While Statement
- Do-While Statement
- For Statement
- Switch Statement
- Case Statement
- Default Statement
- Continue Statement
- Break Statement
- Return Statement
- Goto Statement
- With Statement
- Synchronize Statement
- Try Statement
- Throw Statement
- Volatile Statement
- Asm Statement

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 inadvertently 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 evaluating *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.

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 Expression : 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.

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 `SwitchException` 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.

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:
        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 illegal if the function specifies a void return type.

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 design by contract), 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 ;
```

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 ( TemplateInstance ) BlockStatement
```

where *Expression* evaluates to an Object reference. Within the with body the referenced Object is searched first for identifier symbols. The with statement

```
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.

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*.

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

Do not cache values across volatile statement boundaries.

```
VolatileStatement:
    volatile Statement
```

Statement is evaluated, and no common subexpressions or memory references cached in registers are propagated either into it or out of it. This is useful for accessing memory that can change asynchronously, such as memory mapped I/O or memory accessed by multiple threads.

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:

<code>int* p;</code>	Pointers to data
<code>int[3] s;</code>	Static arrays
<code>int[] a;</code>	Dynamic arrays
<code>int[char[]] x;</code>	Associative arrays (discussed later)

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 handles (references).

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 contain a length and a garbage collected pointer to 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. 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]
```

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];
```

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

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 Properties

Static array properties are:

size	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.
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:

size	Returns the size of the dynamic array reference, which is 8 on 32 bit machines.
length	Get/set number of elements in 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 efficient. 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";
```

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.append(0);
```

The type of a string is determined by the semantic phase of compilation. The type is one of: `ascii`, `wchar`, `ascii[]`, `wchar[]`, 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:

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

It is an error to implicitly convert a string containing non-ascii characters to an ascii string or an ascii constant.

```
(ascii)"\u1234" // error
```

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

```
char c;
wchar u;

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

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.append(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.

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 boolean result indicating if a key is in an associative array or not:

```
if ("hello" in b)
    ...
```


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, for example, the program is done loading up a symbol table and now needs fast lookups in it. Returns a reference to the reorganized array.

Associative Array Example: word count

```

import stdio;           // C printf()
import 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 = File.read(args[i]);           // read file into
input[]

        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]++;                // 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, Enums

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 0, and floating point values to NAN. 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.

```
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, d = 5
```

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.

Enums

```
EnumDeclaration:
    enum identifier { EnumMembers }
    enum { EnumMembers }
    enum identifier ;
```

```
EnumMembers:
    EnumMember
    EnumMember ,
    EnumMember , EnumMembers
```

```
EnumMember:
    Identifier
    Identifier = Expression
```

Enums replace the usual C use of #define macros to define constants. Enums can be either anonymous, in which case they simply define integral constants, or they can be named, in which case they introduce a new type.

```
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

<code>.min</code>	Smallest value of enum
<code>.max</code>	Largest value of enum
<code>.size</code>	Size of storage for an enumerated value

For example:

<code>X.min</code>	is <code>X.A</code>
<code>X.max</code>	is <code>X.C</code>
<code>X.size</code>	is same as <code>int.size</code>

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
```

Classes

The object-oriented features of D all come from classes. The class heirarchy 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:
    { Declarations }
```

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. Hence, alignment statements, anonymous structs, and anonymous unions are not allowed in classes because they are data layout mechanisms. 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.

In C++, it is common practice to define a field, along with "object-oriented" get and set functions for it:

```
class Abc
{
    int property;
    void setProperty(int newproperty) { property = newproperty; }
}

    int getProperty() { return 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. In D, get'ers and set'ers take advantage of the idea that an lvalue is a set'er, and an rvalue is a get'er:

```
class Abc
{
    int myprop;
    void property(int newproperty) { myprop = newproperty; } //
set'er
    int property() { return myprop; } // get'er
}
```

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 you can treat a property like it was a simple field name. A property can start out actually being a simple field name, but if later it becomes necessary to make getting and setting it function calls, no code needs to be modified other than the class definition.

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 heirarchy.

Constructors

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 = 3;
    }
}
```

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:

```
2.         this() { this(1); }
3.         this(int i) { this(); }           // illegal, cyclic
           constructor calls
```
4. If any constructor call appears inside a constructor, any path through the constructor must make exactly one constructor call:

```
5.         this() { a || super(); }         // illegal
6.
7.         this() { this(1) || super(); } // ok
8.
9.         this()
10.        {
11.            for (...)
12.            {
13.                super();                 // illegal, inside loop
14.            }
15.        }
```
16. It is illegal to refer to **this** implicitly or explicitly prior to making a constructor call.
17. 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 *NewExpressions*:

```
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

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.

Static Constructors

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 function 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

A static destructor is defined as a special static function with the syntax `static ~this()`.

```
class Foo
{
    static ~this()           // static destructor
    {
    }
}
```

A static constructor 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

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 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 added with the invariants of its base classes.

There can be only one `invariant()` function per class.

When compiling for release, the invariant code is not generated, and the compiled program runs at maximum speed.

Unit Tests

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);
    }
}
```

There can be only one `unittest` function per class.

Class Allocators

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

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
}
```


Functions

Virtual Functions

All non-static member functions are virtual. This may sound inefficient, but since the D compiler knows all of the class heirarchy 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.

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()
}
```

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
```

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()
            }
        }
    }
}

```

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*.

Operator Overloading

Overloading is accomplished by interpreting specially named member functions as being implementations of unary and binary operators. No additional syntax is used.

Unary Operator Overloading

Overloadable Unary Operators

<i>op</i>	<i>opfunc</i>
-	neg
~	com
<i>e</i> ++	postinc
<i>e</i> --	postdec

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 neg(); }
2.      A a;
3.      -a;      // equivalent to a.neg();
4.
5.      class A { int neg(int i); }
6.      A a;
7.      -a;      // equivalent to a.neg(), which is an error
8.
```

Binary Operator Overloading

Overloadable Binary Operators

<i>op</i>	commutative?	<i>opfunc</i>	<i>opfunc_r</i>
+	yes	add	-
-	no	sub	sub_r
*	yes	mul	-

/	no	div	div_r
%	no	mod	mod_r
&	yes	and	-
	yes	or	-
^	yes	xor	-
<<	no	shl	shl_r
>>	no	shr	shr_r
>>>	no	ushr	ushr_r
~	no	cat	cat_r
==	yes	eq	-
!=	yes	eq	-
<	yes	cmp	-
<=	yes	cmp	-
>	yes	cmp	-
>=	yes	cmp	-
+=	no	addass	-
-=	no	subass	-
*=	no	mulass	-
/=	no	divass	-
%=	no	modass	-
&=	no	andass	-
=	no	orass	-
^=	no	xorass	-
<<=	no	shlass	-
>>=	no	shrass	-
>>>=	no	ushrass	-
~=	no	catass	-

Given a binary overloadable operator *op* and its corresponding class or struct member function name *opfunc* and *opfunc_r*, the syntax:

a op b

is interpreted as if it was written as:

`a.opfunc(b)`

or:

`b.opfunc_r(a)`

The following sequence of rules is applied, in order, to determine which form is used:

1. If *a* is a struct or class object reference that contains a member named *opfunc*, the expression is rewritten as:
 2. `a.opfunc(b)`
3. If *b* is a struct or class object reference that contains a member named *opfunc_r* and the operator *op* is not commutative, the expression is rewritten as:
 4. `b.opfunc_r(a)`
5. If *b* is a struct or class object reference that contains a member named *opfunc* and the operator *op* is commutative, the expression is rewritten as:
 6. `b.opfunc(a)`
7. If *a* or *b* is a struct or class object reference, it is an error.

Examples

```

1.      class A { int add(int i); }
2.      A a;
3.      a + 1; // equivalent to a.add(1)
4.
5.      1 + a; // equivalent to a.add(1)
6.
7.      class B { int div_r(int i); }
8.      B b;
9.      1 / b; // equivalent to b.div_r(1)
10.

```

Overloading == and !=

Both operators use the `eq()` function. The expression `(a == b)` is rewritten as `a.eq(b)`, and `(a != b)` is rewritten as `!a.eq(b)`.

The member function `eq()` is defined as part of `Object` as:

```
int eq(Object o);
```

so that every class object has an `eq()`.

If a struct has no `eq()` 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 `cmp()` function. The expression `(a op b)` is rewritten as `(a.cmp(b) op 0)`. The commutative operation is rewritten as `(0 op b.cmp(a))`.

The member function `cmp()` is defined as part of `Object` as:

```
int cmp(Object o);
```

so that every class object has a `cmp()`.

If a struct has no `cmp()` 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.cmp(null))
```

which will fail if `cmp` is a virtual function.

Rationale

The reason for having both `eq()` and `cmp()` 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 `cmp()` with:

```
•         class A
•         {
•             int cmp(Object o)
•             {
•                 assert(0);    // comparison makes no sense
•                 return 0;
•             }
•         }
```

Future Directions

Likely many more operators will become overloadable. But the operators `..`, `&&`, `||`, `?:`, and `a`

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

TypeParameter:
    Identifier
    Identifier : Type

ValueParameter:
    Declaration
    Declaration : AssignExpression
```

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 either types or values. Value parameters must be of an integral type, and specializations for them must resolve to an integral constant.

Templates are instantiated with:

```
TemplateInstance:
    instance TemplateIdentifier ( TemplateArgumentList )

TemplateAliasDeclaration:
    TemplateInstance AliasIdentifier;

AliasIdentifier:
    Identifier

TemplateArgumentList:
    TemplateArgument
    TemplateArgument , TemplateArgumentList

TemplateArgument:
    Type
    AssignExpression
```

Once instantiated, the declarations inside the template, called the template members, are in the scope of the *AliasIdentifier*:

```
template TFoo(T) { alias T* t; }
```

```
instance TFoo(int) abc;
...
abc.t x;          // declare x to be of type int
```

Template members can also be accessed directly from the *TemplateInstance*:

```
template TFoo(T) { alias T* t; }
instance TFoo(int).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; }
instance TFoo(int) a;
instance 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 *TemplateInstances* 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:

```
instance TCopy(int) copyint;
```

And then the instance can be called:

```
int i;
copyint.copy(i, 3);
```

Instantiation Scope

TemplateInstantances 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() { }
instance 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) { }
instance TFoo(int) f;
...
f.bar();          // will call a.func(double)

```

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) { }
instance TFoo(int) Foo1;          // (1) T is deduced to be int
instance TFoo(char*) Foo2;      // (1) T is deduced to be char*

template TFoo(T : T*) { }
instance TFoo(char*) Foo3;      // (2) T is deduced to be char

template TBar(D, U : D[]) { }
instance TBar(int, int[]) Bar1;  // (2) D is deduced to be
int, U is int[]
instance TBar(char, int[]) Bar2; // (4) error, D is both char and
int

template TBar(D : E*, E) { }
instance TBar(int*, int);        // (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) { }
instance TFoo(B);              // (3) T is B

template TBar(T : U*, U : A) { }
instance TBar(B*, B);          // (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(instance 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

instance TFoo(int) foo1;          // instantiates #1
instance TFoo(double[]) foo2;    // instantiates #2 with T being
double
instance TFoo(char) foo3;        // instantiates #3
instance TFoo(char, int) foe;    // error, number of arguments
mismatch
instance 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.

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
    }
}
```

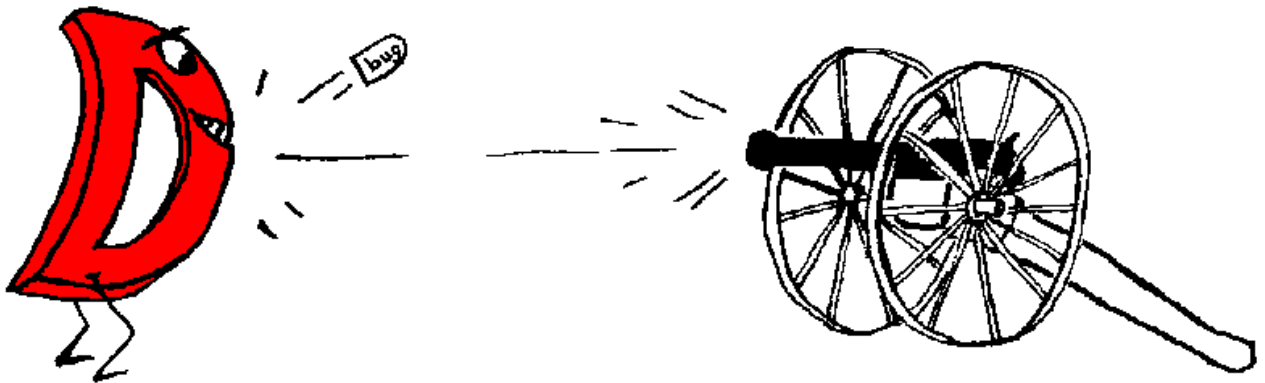
Templates cannot be declared inside functions.

Contracts

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 satisfied. Overriding functions then becomes a process of *loosening* the `in` contracts.

Conversely, all of the `out` contracts needs to be satisfied, 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`.

Debug and Version

D supports building multiple versions and various debug builds from the same source code using the features:

DebugSpecification
DebugAttribute
DebugStatement

VersionSpecification
VersionAttribute
VersionStatement

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.

DigitalMars

Digital Mars is the compiler vendor

X86

Intel and AMD 32 bit processors

Win32

Microsoft 32 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

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.

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.

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:
    debug Statement
    debug ( Integer ) Statement
    debug ( Identifier ) Statement
    
```

Debug statements are compiled in when the `-debug` switch is thrown on the compiler.

`debug(Integer)` statements are compiled in when the debug level `n` set by the `-debug(n)` switch is `<= Integer`.

`debug(Identifier)` statements are compiled in when the debug identifier set by 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

```

There is no else clause for a debug statement, as debug statements should add code, not subtract code.

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* or *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(n) { } // add in debug code if debug level is <= n
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`.

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`. `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 collector 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

The garbage collector's algorithms depend on pointers being pointers and not pointers being not pointers. To that end, the following practices that are not unusual in C should be carefully avoided in D:

- Do not hide pointers by xor'ing 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.
- Do not store pointers into int variables using casts and other tricks. The garbage collector does not scan non-pointer types for roots.
- Do not take advantage of alignment of pointers to store bit flags in the low order bits, do not store bit flags in the high order bits.
- Do not store integer values into pointers.
- Do not store magic values into pointers, other than `null`.

- If you *must* share the same storage location between pointers and non-pointer types, use a union to do it so the garbage collector knows about it.

In fact, avoid using pointers at all as much as possible. 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 wizard 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
- Real Time
- Smooth Operation
- Free Lists
- Reference Counting
- Explicit Class Instance Allocation
- Mark/Release
- RAII (Resource Acquisition Is Initialization)
- Allocating Class Instances On The Stack

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 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 RIAA 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 *NewDeclarations* and *DeleteDeclarations*. For example, to allocate using the C runtime library's `malloc` and `free`:

```
import c.stdlib;
import outofmemory;
import gc;

class Foo
{
    new(uint sz)
    {
        void* p;

        p = c.stdlib.malloc(sz);
        if (!p)
            throw new OutOfMemory();
        gc.addRange(p, p + sz);
        return p;
    }

    delete(void* p)
    {
        if (p)
        {
            gc.removeRange(p);
            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 c.stdlib;
import 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.

```

import c.stdlib;

class Foo
{
    new(uint sz, void *p)
    {
        return p;
    }

    delete(void* p)
    {
        assert(0);
    }
}

void test()
{
    Foo f = new(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.

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:

```
imaginary j = 1.3i;
```

There is no particular complex literal syntax, just add a real and imaginary type:

```
complex c = 4.5 + 2i;
```

Adding two new types to the language is enough, hence complex and imaginary have extended precision. There is no complex float or complex double type, and no imaginary float

or imaginary double. [NOTE: the door is open to adding them in the future, but I doubt there's a need]

Complex numbers have two properties:

```
.re    get real part as an extended
.im    get imaginary part as an imaginary
```

For example:

```
c.re          is 4.5
c.im          is 2i
```

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 [blah, blah, blah] and match the semantics for the NCEG extensions to C.

[insert table here]

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
cvts2sd	cvts2si	cvttpd2dq	cvttpd2pi	cvttps2dq
cvttps2pi	cvtt2si	cvtt2si	cwdd	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	fcmovbe	fcmovnb	fcmovnbe	fcmovne
fcmovnu	fcmovu	fcom	fcomi	fcomip
fcomp	fcompp	fcos	fdecstp	fdiv
fdivp	fdivr	fdivrp	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
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
pshufw	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	pfrsqr	pfsb	pfsbr	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 string;
int myDfunction(char[] s)
{
    return strcmp(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
<i>type*</i>	<i>type *</i>
<i>type[dim]</i>	<i>type[dim]</i>
<i>type[]</i>	no equivalent
<i>type[type]</i>	no equivalent
<i>"string\0"</i>	<i>"string" or L"string"</i>
class	no equivalent
<i>type(*)(parameters)</i>	<i>type(*)(parameters)</i>

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.
- The **wchar** type can be either 2 or 4 bytes wide in current implementations; future implementations can increase the size further.
- 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.

- 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);`

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 Object;
import stdio;

int main()
{
    printf("hello world\n");
    return 0;
}
```

D Runtime Model

Object Model

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 ClassLayout
4...    pointers to virtual member functions
```

Array Model

A dynamic array consists of:

```
0:      pointer to array data
4:      array dimension
```

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.

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
```

Class Model

The class definition:

```
class XXXX
{
    ....
};
```

Generates the following:

- o An instance of Class called ClassXXXX.
- o A type called StaticClassXXXX which defines all the static members.
- o An instance of StaticClassXXXX called StaticXXXX for the static members.

Phobos

D Runtime Library

Phobos is the standard runtime library that comes with the D language compiler.

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.

No user interface windowing classes

GUI styles, philosophies, etc., are not portable from machine to machine. A GUI Windows app should look like a Windows app when running on a Windows machine. It should not look and feel like a Mac app unless it is running on a Mac. Attempts to create a common GUI class library between Windows, Mac, and other GUI operating systems have all to my knowledge failed.

Java has a successful GUI class library, but does so by creating its own GUI with its own look and feel. This approach is fine for a web language, but not for a systems language like D is.

Windowing class libraries should be separate.

Class implementations should use DBC

This will prove that DBC (Design by Contract) 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

Each of these can be imported with the **import** statement. The categories are:

Core D: Available on all D implementations

compiler

Information about the D compiler implementation.

conv

Conversion of strings to integers.

ctype

Simple character classification

date

Date and time functions. Support locales.

file

Basic file operations like read, write, append.

gc

Control the garbage collector.

math

Include all the usual math functions like sin, cos, atan, etc.

object

The root class of the inheritance heirarchy

outbuffer

Assemble data into an array of bytes

path

Manipulate file names, path names, etc.

process

Create/destroy threads.

random

Random number generation.

regexp

The usual regular expression functions.

stdint

Integral types for various purposes.

stream

Stream I/O.

string

Basic string operations not covered by array ops.

system

Inquire about the CPU, operating system.

thread

One per thread. Operations to do on a thread.

zip

Manipulate zip files.

Standard C: interface to C functions

stdio

Interface to C stdio functions like printf().

Operating System and Hardware: platform specific

intrinsic

Compiler built in intrinsic functions

windows

Interface to Windows APIs

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.

conv

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)

ctype

int **isalnum**(char c)

Returns !=0 if c is a letter or a digit.

int **isalpha**(char c)

Returns !=0 if *c* is an upper or lower case letter.

int **isctrl**(char *c*)
Returns !=0 if *c* is a control character.

int **isdigit**(char *c*)
Returns !=0 if *c* is a digit.

int **isgraph**(char *c*)
Returns !=0 if *c* is a printing character except for the space character.

int **islower**(char *c*)
Returns !=0 if *c* is lower case.

int **isprint**(char *c*)
Returns !=0 if *c* is a printing character or a space.

int **ispunct**(char *c*)
Returns !=0 if *c* is a punctuation character.

int **isspace**(char *c*)
Returns !=0 if *c* is a space, tab, vertical tab, form feed, carriage return, or linefeed.

int **isupper**(char *c*)
Returns !=0 if *c* is an upper case character.

int **isxdigit**(char *c*)
Returns !=0 if *c* is a hex digit (0..9, a..f, A..F).

int **isascii**(uint *c*)
Returns !=0 if *c* is in the ascii character set.

char **tolower**(char *c*)
If *c* is upper case, return the lower case equivalent, otherwise return *c*.

char **toupper**(char *c*)
If *c* is lower case, return the upper case equivalent, otherwise return *c*.

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 fo *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.

`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.

file

class **FileException**
Exception thrown if file I/O errors.

`byte[] read(char[] name)`
Read file *name[]*, return array of bytes read.

`void write(char[] name, byte[] buffer)`
Write *buffer[]* to file *name[]*.

`void append(char[] name, byte[] 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.

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()

Temporarily 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.

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 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 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

```

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.

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:

<i>x</i>	return value	invalid?
±INFINITY	NAN	yes

real **sin**(real *x*)

Compute sine of *x*. *x* is in radians.

Special values:

<i>x</i>	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:

<i>x</i>	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:

$$value = x * 2^{exp}$$

$$.5 \leq |x| < 1.0$$

x has same sign as *value*.

Special values:

<i>value</i>	<i>x</i>	<i>exp</i>
+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:

<i>x</i>	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:

<i>x</i>	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)

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 :

$$\text{sqrt}(x^2 + y^2)$$

Note that $\text{hypot}(x,y)$, $\text{hypot}(y,x)$ and $\text{hypot}(x,-y)$ are equivalent.

Special values:

x	y	return value	invalid?
x	$+0.0$	$\text{fabs}(x)$	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.

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 **cmp**(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**.

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*.

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**(char c1, char 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

process

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.

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 (<i>string</i>)
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 <i>n</i> th parenthesized match, <i>n</i> is 1..9
\$nn	replace with the <i>nn</i> th parenthesized match, <i>nn</i> 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 <i>n</i> th parenthesized match, <i>n</i> is 1..9
\c	replace with char <i>c</i> .

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 signed
uint8_t	exactly 8 bits unsigned	uint_least8_t	at least 8 bits unsigned	uint_fast8_t	fast 8 bits unsigned
int16_t	exactly 16 bits signed	int_least16_t	at least 16 bits signed	int_fast16_t	fast 16 bits signed

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 signed
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 signed
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

stream

class **Stream**

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 **writable**

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 **ReadError** being thrown.

uint **readBlock**(void* buffer, uint size)

Read up to size bytes into the buffer and return the number of bytes actually read.

void **readExact**(void* buffer, uint size)

Read exactly size bytes into the buffer, throwing a **ReadError** 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 ireal x)

void **read**(out creal x)

void **read**(out char x)

void **read**(out wchar x)

void **read**(out char[] s)

void **read**(out wchar[] s)

Read a basic type or counted string, throwing a **ReadError** 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**()

wchar[] **readLineW**()

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.

char[] **readString**(uint length)

Read a string of the given length, throwing **ReadError** if there was a problem.

wchar[] **readStringW**(uint length)

Read a string of the given length, throwing **ReadError** 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**.

Writing

These methods require that the **writable** flag be set. Problems with writing result in a **WriteError** being thrown.

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 **writeExact**(void* buffer, uint size)

Write exactly *size* bytes from *buffer*, or throw a **WriteError** 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**(ireal x)

void **write**(creal x)

void **write**(char x)

void **write**(wchar 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 **WriteError** if it could not be fully written.

void **writeStringW**(wchar[] s)

Write a string of text, throwing **WriteError** 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 **copyFrom**(Stream s)

Copies all data from *s* into this stream. This may throw **ReadError** or **WriteError** 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 **ReadError** or **WriteError** 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 **SeekError** 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`.

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 **File : Stream**

This subclass is for file system streams.

this()

this(char[] filename)

this(char[] filename, FileMode mode)

Create the stream with no open file, an open file in read and write 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). If the file does not exist, it is created.

void **open**(char[] filename)

void **open**(char[] filename, FileMode mode)

Open a file for the stream, in an identical manner to the constructors.

void **create**(char[] filename)

void **create**(char[] filename, FileMode mode)

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 **MemoryStream : Stream**

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.
```

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* toCharz(char[] string)
 Converts a D array of chars to a C-style 0 terminated string.

int find(char[] s, char c)
 Find first occurrence of c in string s. Return index in s where it is found. Return -1 if not found.

int rfind(char[] s, char c)
 Find last occurrence 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 occurrence 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 occurrence of *sub* in string *s*. Return index in *s* where it is found. Return -1 if not found.

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.

system

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.

zip

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 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 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.     _m_init();                // initialize module constructor
    table
21.
22.     try
23.     {
24.         _moduleCtor();        // call module constructors
25.         _moduleUnitTests();   // run unit tests (optional)
26.
27.         result = doit();      // insert user code here
28.     }
29.
30.     catch (Object o)          // catch any uncaught exceptions
31.     {
32.         MessageBoxA(null, (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. }

```

The `doit()` function is where the user code goes, the rest of `WinMain` is boilerplate to initialize and shut down the D runtime system.

40. A `.def` (Module Definition File) with at least the following two lines in it:

```

41. EXETYPE NT
42. SUBSYSTEM WINDOWS

```

Without those, Win32 will open a text console window whenever the application is run.

43. The presence of `WinMain()` is recognized by the compiler causing it to emit a reference to `__acrtused` 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 windows;
HINSTANCE g_hInst;

extern (C)
{
    void gc_init();
    void gc_term();
    void _m_init();
    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`.

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 whose 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 com;

class MyCOMObject : ComObject
{
    extern (Windows):
        ...
}
```

The sample code includes an example COM client program and server DLL.

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 libraries, this table is for features built in to the language itself.

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
Covariant return types	Yes	No	Yes	No	No
Arrays					
Lightweight arrays	Yes	Yes	Yes	No	No
Resizable 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
Aliases	Yes	Yes	Yes	No	No
OOP					
Object Oriented	Yes	No	Yes	Yes	Yes
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	No	Yes
Inner classes	No	No	No	No	Yes

Feature	D	C	C++	C#	Java
Covariant return types	Yes	No	Yes	No	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
Templates	Yes	No	Yes	No	No
Reliability					
Design by Contract	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	Yes	No	Yes	Yes	No
Exception handling	Yes	No	Yes	Yes	Yes
try-catch-finally blocks	Yes	No	No	Yes	Yes
Thread synchronization primitives	Yes	No	No	Yes	Yes
Compatibility					
Algol-style syntax	Yes	Yes	Yes	Yes	Yes
Enumerated types	Yes	Yes	Yes	Yes	No
Support all C types	Yes	Yes	No	No	No
Long double 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	Yes	Yes	No	No
Generates standard object files	Yes	Yes	Yes	No	No
Macro preprocessor	No	Yes	Yes	No	No
Other					
Conditional compilation	Yes	Yes	Yes	Yes	No

Notes

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.

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.

Design by Contract

The Digital Mars C++ compiler supports Design by Contract as an extension.

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.

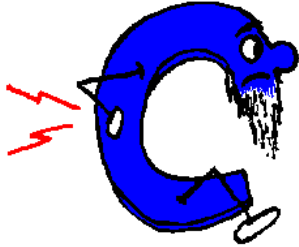
Struct member alignment control

Although many C/C++ compilers contain pragmas to specify struct alignment, these are nonstandard and incompatible from compiler to compiler.

Long double floating point

While the standard for C and C++ specify long doubles, few compilers (besides Digital Mars C/C++) actually implement 80 bit (or longer) floating point types.

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
- Get the max and min values of a type
- Primitive Types
- Special Floating Point Values
- Taking the Modulus of a floating point number
- Dealing with NAN's in floating point compares
- Asserts
- Initializing all elements of an array
- Looping through an array
- Creating an array of variable size
- String Concatenation
- Formatted printing
- Forward referencing functions
- Functions that have no arguments
- Labelled break's and continue's
- Goto Statements
- Struct tag name space
- Looking up strings
- Setting struct member alignment
- Anonymous Structs and Unions
- Declaring struct types and variables
- Getting the offset of a struct member
- Union initializations
- Struct initializations
- Array initializations
- Escaped String Literals
- Ascii vs Wide Characters
- Arrays that parallel an enum
- Creating a new typedef'd type
- Comparing structs
- Comparing strings

- Sorting arrays
 - Volatile memory access
 - String literals
 - Data Structure Traversal
-

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
```

Taking the Modulus of a floating point number

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 modulus ('%') 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]);
```

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[array.length++] = 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 is 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 a[3] = [ 3, 2, 0 ];
int a[3] = [ 3, 2 ];           // unsupplied initializers are 0,
just like in C
int a[3] = [ 2:0, 0:3, 1:2 ];
int a[3] = [ 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 c[3] = [ black:3, green:2, red:5 ];
```

Nested array initializations must be explicit:

```
int b[3][2] = [ [2,3], [6,5], [3,4] ];

int b[3][2] = [[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 specify 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[] = \" '[^\\]*(\\. [^\\]*)*' \";
```

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 cstring[COLORS.max + 1][] =
[
    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);           // syntax error
bar(h);           // ok
```

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 reimplement 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 *t2 = (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;

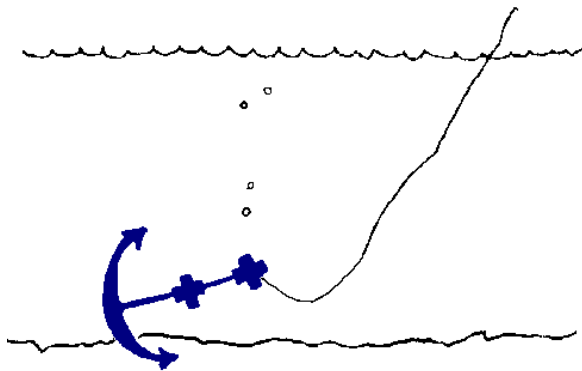
    void membersearchx(Symbol s)
    {
        while (s)
        {
            if (id == s.id)
            {
                if (sm)
                    error("ambiguous member %s\n", id);
                sm = s;
            }

            if (s.left)
                membersearchx(s.left);
            s = s.right;
        }
    }

    for (int i = 0; i < table.length; i++)
    {
        membersearchx(table[i]);
    }
    return sm;
}

```

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](#)
- [Comparing structs](#)
- [Creating a new typedef'd type](#)
- [Friends](#)
- [Operator overloading](#)
- [Namespace using declarations](#)
- [RAII \(Resource Acquisition Is Initialization\)](#)
- [Dynamic Closures](#)

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
}
```

Dynamic Closures

The C++ Way

Consider a reusable container class. 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.

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
- #pragma once
- #pragma pack
- Macros
- Conditional Compilation
- Code Factoring

Header Files

The C Preprocessor Way

C and C++ rely heavily 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 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 problematic to 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 the macros 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 capability, 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
```

The D Way

```
const int VALUE = 5;
```


2. Creating a list of values or flags:

The C Preprocessor Way

```
int flags:
#define FLAG_X 0x1
#define FLAG_Y 0x2
#define FLAG_Z 0x4
...
flags |= FLAGS_X;
```

The D Way

```
enum FLAGS { X = 0x1, Y = 0x2, Z = 0x4 };
FLAGS flags;
...
flags |= FLAGS.X;
```

3. Distinguishing between ascii chars and wchar chars:

The C Preprocessor Way

```
#if UNICODE
#define dchar      wchar_t
#define TEXT(s)   L##s
#else
#define dchar      char
#define TEXT(s)   s
#endif

...
dchar h[] = TEXT("hello");
```

The D Way

```
import dchar;           // contains appropriate typedef for dchar
...
dchar[] h = "hello";
```

D's optimizer will inline the function, and will do the conversion of the string constant at compile time.

4. Supporting legacy compilers:

The C Preprocessor Way

```
#if PROTOTYPES
#define P(p)    p
#else
#define P(p)    ()
#endif
int func P((int x, int y));
```

The D Way

By making the D compiler open source, it will largely avoid the problem of syntactical backwards compatibility.

5. Type aliasing:

The C Preprocessor Way

```
#define INT    int
```

The D Way

```
alias int INT;
```

6. Using one header file for both declaration and definition:

The C Preprocessor Way

```
#define EXTERN extern
#include "declarations.h"
#undef EXTERN
#define EXTERN
#include "declarations.h"
```

In declarations.h:

```
EXTERN int foo;
```

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.

7. Lightweight inline functions:

The C Preprocessor Way

```
#define X(i)    ((i) = (i) / 3)
```

The D Way

```
int X(inout int i) { return i = i / 3; }
```

The compiler optimizer will inline it; no efficiency is lost.

8. Assert function file and line number information:

The C Preprocessor Way

```
#define assert(e) ((e) || _assert(__LINE__, __FILE__))
```

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.

9. Setting function calling conventions:

The C Preprocessor Way

```
#ifndef _CRTAPI1
#define _CRTAPI1 __cdecl
#endif
#ifndef _CRTAPI2
#define _CRTAPI2 __cdecl
#endif

int _CRTAPI2 func();
```

The D Way

Calling conventions can be specified in blocks, so there's no need to change it for every function:

```
extern (Windows)
{
    int onefunc();
    int anotherfunc();
}
```

10. Hiding __near or __far pointer wierdness:

The C Preprocessor Way

```
#define LPSTR char FAR *
```

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.

11. Simple generic programming:

The C Preprocessor Way

Selecting which function to use based on text substitution:

```
#ifdef UNICODE
int getValueW(wchar_t *p);
#define getValue getValueW
#else
int getValueA(char *p);
#define getValue getValueA
#endif
```

The D Way

D enables declarations of symbols that are *aliases* of other symbols:

```
version (UNICODE)
{
    int getValueW(wchar[] p);
    alias getValueW getValue;
}
else
{
    int getValueA(char[] p);
    alias getValueA getValue;
}
```

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.

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 words should have each word other than the first capitalized.

```
int myFunc();
```

Module

Module names are all lower case.

C Modules

Modules that are interfaces to C functions go into the "c" package, for example:

```
import 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, do array slicing, and simple symbol table management with associative arrays.

```
import stdio;
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 = 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[] word = input[wstart .. input.length];
            dictionary[word]++;
        }
        printf("%8lu%8lu%8lu %s\n", l_cnt, w_cnt, c_cnt, (char *)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");
    char[][] keys = dictionary.keys;
    for (int i = 0; i < keys.length; i++)
    {
        char[] word;

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

Compiler for D Programming Language

This is the D compiler for Win32.

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
`\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

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
<i>none</i>	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 \leq *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
- o*objdir***
write object files to directory *objdir* instead of to the current directory
- release**
compile release version
- unittest**
compile in unittest code
- v**
verbose
- version=*level***
compile in version code \geq *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 **snm.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 environemnt
; 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"
```

Bugs

These are some of the major bugs:

- The compiler quits on the first error, and sometimes gets the line number wrong.
- 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.
- Property getter/setter not implemented.
- In preconditions and out postconditions for member functions are not inherited.
- It cannot be run from the IDDE.

Feedback

We welcome all feedback - kudos, flames, bugs, suggestions, hints, and most especially donated code!

Acknowledgements

The following people have contributed to the D language project; with ideas, code, expertise, marketing, inspiration and moral support.

Bruce Eckel, Eric Engstrom, Jan Knepper, Lubomir Litchev, Pavel Minayev, Paul Nash, Pat Nelson, Burton Radons, Tim Rentsch, Fabio Riccardi, Bob Taniguchi, John Whited, Peter Zatloukal