The D Programming Language

Why D?

The software industry has come a long way since the C language was invented. Many new concepts were added to the language with C++, but backwards compatibility with C was maintained. Additionally, C++ was very much designed by committee, and hence is loaded with multiple overlapping ways of doing the same thing, and compromises for political purposes. C++ has become so complicated that nobody really understands it, and no C++ compiler exists that properly implements 100% of the spec (even if someone did understand 100% of the spec).

Software has grown thousands of times more complex. What is needed is for the language to be forward looking to solve today's and future software programming needs, not looking backwards to bring along code written 30 years ago. For legacy code written 30 years ago, they can be ably compiled by existing C and C++ compilers.

A new language needs to be developed that takes the best features and capabilities of C++, adds in modern features that are impractical in C++, and puts it all in a package that is easy for compiler writers to implement, and which enables compilers to easily generate optimized code.

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.)

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 will be like C and C++. This will make 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 and will not have to learn a new environment.

o The compile/link/debug development model will be carried forward, although D will also fit well with interpreters and JIT development models.

- 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.

- D maintains function link compatibility with the C calling conventions. This makes it possible for D programs to access operating system APIs directly.

Features To Drop

- C source code compatibility. Extensions to C that maintain source compatiblity have already been done (C++ and ObjectiveC). That vein is pretty much played out.

- Link compatibility with C++. The C++ object model is just too complicated to be worth supporting.

- Multiple inheritance. It's a tremendously complex feature of dubious value. It's very difficult to implement in an efficient manner, and compilers are prone to many bugs in implementing it.

- Templates. A great idea in theory, but in practice leads to numbingly complex code to implement trivial things like a "next" pointer in a singly linked list.

- Namespaces. Primarilly a hodge podge of rules trying to get around C++'s legacy scoping rules.

- Include files. A major cause of slow compiles as each compilation unit must reparse enormous quantities of header files.

- Creating object instances on the stack. In D, all objects are by reference. This eliminates the need for copy constructors, assignment operators, complex destructor semantics, and interactions with exception handling stack unwinding.

- Trigraphs and digraphs. Better to just support unicode. A large class of problems just go away if the source text of the language can be unicode.

- Old style function prototypes. A thoroughly obsolete legacy feature.

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

- Operator overloading. Another feature that looks great on paper, but in

practice just doesn't work out very well. The only practical applications for operator overloading seem to be implementing a complex floating point type and a string class. D provides both of these natively.

o Object oriented gradualism. It should either be object oriented or not, not a buffet of aspects of object orientedness.

o Bit fields of arbitrary size. A complex, inefficient feature rarely used.

---------- Who D is For ----------

- People who wonder why lint is not part of the language.

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

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

o Those who look at C++ and think "there's gotta be an easier way to do Object Oriented Programming."

o Programmers who can't remember the byzantine rules for which overloaded function is selected.

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

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

o Numerics programming. D has many features to directly support features needed by numerics programmers, like direct support for the complex data type.

o 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 ----------

o Programmers who delight in the "C Puzzle Book."

o Winners of the "Obfuscated C Code Contest."

o Programmers who write macros 3 levels deep.

- People who understand what problem the “mutable” keyword attempts to solve.
-

o Real time programming where latency must be guaranteed.

- 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 APIs very well.)

---------- Language Specification ----------

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/unicode

   The source file is checked to see if it is in ascii or unicode, and the appropriate scanner is loaded.

2. lexical analysis

   The source file is divided up into a sequence of tokens.

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

Identifiers

Identifiers start with [_A-Za-z], and continue with [_A-Za-z0-9]. Identifiers starting with an '__' are reserved. There is no limit to the length of an identifier.

Strings

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

Single quoted strings are enclosed by '. All characters between the ' are part of the string, 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.

"hello"	
"c:\\root\\foo.exe"	
"ab\n"	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
\0123	octal
\x1A	hex
\u1234	unicode character
\r\n	carriage return, line feed

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"
```

The type of the string is determined by the semantic phase of compilation. The type is one of: ascii, unicode, ascii[], unicode[], 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 approprate:

(unicode [])"abc"	this is an array of unicode 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 unicode constant:

```
char c;
unicode u;

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

[NOTE: String syntax is patentable. Invented by Walter Bright, John Whited, Eric Engstrom]

Keywords

```
ascii
unicode
byte
short
ushort
int
uint
long
ulong
float
double
extended
imaginary
complex

asm
with
null
new
delete
in
out
body
```

instanceof
asm
typeof ?
import
export
synchronize

do
while
for
switch
try
catch
finally
break
continue
default
case

struct
union
enum
class

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
float	32 bit floating point
double	64 bit floating point
extended	largest hardware implemented floating point size (80 bits for Intel CPU's)
imaginary	an extended floating point value, but with imaginary type
complex	two extended floating point values, one real and the other imaginary
ascii	unsigned 8 bit ASCII

unicode	unsigned 16 bit Unicode

[NOTE: complex float, complex double, imaginary float, and imaginary double may wind up being added some time in the future.]

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

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
```

Properties for Integral Data Types

.size	size in bytes
.max	maximum value
.min	minimum value
.sign	should we do this?

Properties for Floating Point Types

.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

Nulls

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.

Pointer Casts

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.

Structs, Unions

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 not 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.

[Note: perhaps we should do away with unions entirely, or at least unions that contain pointers.]

Enums

Enums, of course, replace the usual 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 };	Define the constants A=1, B=2, C=3 in a manner equivalent to:
	const int A = 1;
	const int B = 2;
	const int C = 3;

enum X { A, B, C };	Define a new type X which has values X.A=1, X.B=2, X.C=3

Named enum members can be implicitly cast to integral types, but integral types cannot be implicitly cast to an enum type.

Enum Properties

.min	Smallest value of enum
.max	Largest value of enum
.size	Size of storage for an enumerated value

For example:

X.min	is X.A
X.max	is X.C
X.size	is same as int.size

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
```

[NOTE: Eric thinks an = should be used instead of a :. Pistols at dawn.]

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.

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 - 1] = [ 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..

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;
    }
    {   unicode 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.

Constructors

Members are always initialized to zero, except for floating point members which are initialized to NAN. This eliminatesan entire class of obscure problems that come from neglectingto initialize a member in one of the constructors. Additionally,the beauty of the NAN initialization is that any floating pointoperation with any NAN operand produces a NAN result.In the class definition, the programmer can supply a static initializer to be used instead of the default:

```
class Abc
{
        long bar = 7;           // set default initialization
}
```

[NOTE: should explicit static initialization be required for all members?]

D constructors are not defined by using the name of the class, but by using the this keyword:

```
class Foo
{
        this(int x)             // declare constructor for Foo
        { …
        }
        this()
        {…
        }
}
```

This eliminates the tedium of retyping long class names over, it matches the use of the constructor, and is analogous to using 'super' to call the base class constructor. Also, since D requires an explicit return type for functions, it eliminates the syntactical ambiguity of constructors having no explicit return type.

C++ constructors have a complex syntax to initialize the base class. This clumsiness becomes even more onerous and error prone when there are several constructors, each with quite a bit of parallel code in common. D eliminates this by allowing one constructor to call another at any point - the constructor is really just another function. The vptr initialization is performed before the constructor is ever called (by the new operator). The base class constructor is explicitly called by the name "super".

```
class A { this(int y) { } }
```

```
class B : A
{       int j;
        this()
        {
           ...blah...
           super(3);            // call base constructor A(3)
           ...blah...
           this(6);
        }
        this(int i)
        {
           super(4);
           j = 3;
        }
}
```

D objects are created with the new syntax:

```
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 vtbl pointer is assigned as part of this. 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. It is that constructor's responsibility to call any base class constructor.

Destructors

The garbage collector calls the destructor function when the object is deleted. The syntax is:

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

```
}
```

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.

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 as a static function with the name _staticCtor:

```
    class Foo
    {
        static int a;           // default initialized to 0          static int b =
1;
        static int c = b + a;   // error, not a constant initializer

        static void _staticCtor()
```

```
        {
                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 name _staticDtor:

```
class Foo
{
        static void _staticDtor()
        {
        }
}
```

A static constructor gets called on program termination, but only if the static constructor completed successfully. Static destructors have empty parameter lists.

Arrays

C arrays have several faults that can be corrected:

1. 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[]).

2. 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 informationgets lost.

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

4. C arrays cannot be bounds checked, because they don't know

what the array bounds are.

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

int (&array)[3];

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

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

which is much simpler to understand.

Mars arrays come in 4 varieties: pointers, static arrays, dynamic arrays, and associative arrays.

See Arrays.doc

Associative Arrays

D goes one step further with arrays - associative arrays are also fully supported.

```
int[char[]] b;          // associative array indexed by character string
b.length;               // number of elements in the array
b["hello"] = 3;         // set value associated with "hello" to 3
func(b["hello"]);       // pass 3 as parameter to func()
```

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

```
delete b[“hello”];
```

The in-expression yields a boolean result indicating if a key is in an associative array or not:

```
if (“hello” in b)
        ...
```

Associated arrays are supported for all following types.

Properties:

.length number of items in the array

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
}
```

Arrays of Bits

Bit vectors can be constructed:

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

The amount of storage used up is implementation dependent, but 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).

[NOTE: how do enums and bits work together?]

Strings

There's no more obvious failure of C++ being an object-oriented language than the classic String class problem. The goal is to create a class where

strings act like they do in string oriented languages like Javascript, where you can manipulate strings as a whole rather than being concerned with the implementation details. Much heated debate and proposals have been made to implement this in C++, using various schemes of reference counting, strict protocols, etc., but none succeed, and all require tedious attention to avoid corrupted memory, dangling pointers, etc.

Dynamic arrays being first class objects suggest an elegant solution - a String is simply an array of characters:

```
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 arraynot just a special String array.

A pointer to a char can be generated:

```
char *p = &str[3];      // pointer to 3rd element
char *p = str;          // pointer to 0th element
```

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

```
str.push(0);
```

Imports

Rather than text include files, Mars imports symbols symbolically with the import statement:

```
import xyz;
```

The top level scope in xyz is merged with the current scope.

Exports

A class can be exported, which means its name and all its non-private

members are exposed externally to the DLL or EXE.

Inline Assembler

Inline assembler is supported with the asm syntax:

```
asm
{
        // assembler statements go here
}
```

The format of the assembler statements matches the target CPU. For example, for the Intel Pentium:

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

D has no volatile storage type. Volatile is typically used to access hardware registers, so using inline assembler is appropriate for those cases, as in:

```
int gethardware()
{
   asm
   {
           mov EAX, dword ptr 0x1234;
   }
}
```

[NOTE: the inline assembler is not supported yet.]

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 declarationinformation 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;
```

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

Virtual Functions

In D, 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.

[NOTE: typelib support?]

Inline Functions

There is no inline keyword in D. 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 in D, either.)

Function Overloading

In C++, there are many arcane 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 Mars, 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.

[NOTE: Eric suggests making this even simpler, it matches exactly or it does not match.]

Functions defined with non-D linkage cannot be overloaded.

Function Parameters

D supports in, out, and inout parameters. 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:

1. The function declaration makes it clear what the inputs and outputs to the function are.
2. It eliminates the need for IDL as a separate language.
3. It provides more information to the compiler, enabling more error checking and possibly better code generation.
4. It (perhaps?) eliminates the need for reference (&) declarations.

Open for discussion is whether out parameters can only be set when a function returns, i.e. it will not be possible to set an out parameter and then throw an exception. The latter is probably the more robust way.

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 Mars, this is done with the typealias declaration:

```
typealias 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:

```
typealias int myint;

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

Type aliases are equivalent to the C typedef.

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)
```

Persistence

Objects can be persisted by writing them to a file. They can be read back from the file by simply mapping the file into memory and pointing to the object. This results in very fast file reads, and a robust way of creating file formats. Handles within objects are resolved automatically; pointers require user code to support.

[NOTE: this needs much work]

Classes

The object-oriented features of Mars 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.

Class members are always accessed with the . operator. There are no :: or -> operators as in C++.

Fields

The Mars 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 Mars, 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 Mars 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 if becomes necessary to make getting and setting it function calls, no code needs to be modified other than the class definition.

DECLARATION ATTRIBUTES

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

```
attribute declaration;          affects the declaration

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

```
        declaration;
        ...

attribute
{
        declaration;
        declaration;
        ...
}
```

The following attributes are supported:

```
        deprecated
        align
        linkage
        private
        protected
        public
        export
        static
        final
        overload
        abstract
        debug
```

Deprecated

It is often necessary to deprecate a feature in a library, yet retain it for backwards compatiblity. Mars allows such declarations to be marked as deprecated, which means that the programmer can be notified if any new code is written referring to deprecated declarations:

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

Function Linkage

D provides an easy way to call C functions and operating system API functions, as compatibility with both is essential. C function calling conventions are specified by:

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

Windows API conventions are:

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

Pascal conventions are:

```
extern Pascal:
```

And limited linkage to C++ functions is:

```
extern C++:
```

It is limited by the ability to express C++ types in D. For example, no C++ const, volatile, template, or object types can be expressed in D, so C++ functions that take those as parameters cannot be called.

Protection

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. Private members cannot be overridden.

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

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.

Debug

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 attribute conditionally compiles in code:

```
class Foo
{
        int a, b;
   debug:
        int flag;
}

void func(Foo f)
{
        debug int x;
        …
        debug
        {
                x = f.flag;
        }
}
```

FLOATING POINT

Precision

One of the best ways to reduce the problems associated with roundoff error in floating point computations is to use more precision. Isn't it strange, then, that so few languages support extended precision, even when the hardware to do it is installed in 95% of the computers out there? D exposes that feature to the programmer with the extended floating point type.

A D language implementation is also free to compute all intermediate values in the highest precision the hardware allows. D implementations are strongly encouraged to make this the default approach. Smaller precisions like float and double are useful only to conserve storage in massive arrays, and to be type-portable with data and functions from other, more primitive, languages.

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 <g>) 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]

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 matrix[3][3];
```

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];
```

Relational Operators

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:

Symbol	Relation
<	less
>	greater
<=	less or equal
>=	greater or equal
==	equal
!=	unordered, less, or greater
!<>=	unordered
<>	less or greater
<>=	less, equal, or greater
!<=	unordered or greater
!<	unordered, greater, or equal
!>=	unordered or less
!>	unordered, less, or equal
!<>	unordered or equal

It is important to note that, for relational operator op, (a !op b) is not the same as !(a op b).

OPERATOR OVERLOADING

There really appear to be only two significant uses for operator overloading: string manipulation and complex floating point numbers. String operations are handled by the char[] and unicode[] semantics. Complex numbers have their own, built in, native type.

Hence, there is no significant need left to justify adding the whole vast clumsy architecture of operator overloading into D.

EXPRESSIONS

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 tobe able to parse the syntax without reference to the symbol table. So, in order to distinguish a cast from a parenthesized subexpression, a different syntax is necessary.

C++ does this by introducing:

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

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

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

cast has the nice characteristic that it is easy to to 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)
```

Shift Expressions

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

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

In Expressions

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.

STATEMENTS

Goto Statement

Goto's are fully implemented in D.

Synchronization

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.

Null Statements

The null statement in C is represented by a ;. Unfortunately, this can lead to a difficult to see error:

```
while ((I = func()) != 0);
        test();
```

Note the extraneous and nearly invisible ; that crept in after the while(). D solves this by requiring that a null statement be represented by:

```
{}
```

A loop with an empty body is written:

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

DEBUG SUPPORT

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.

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 APIs

Runtime Checking

- assert() expressions
- array bounds checking
- undefined case in switch exception
- out of memory exception

THE C PREPROCESSOR VERSUS MARS

Back when C was invented, compiler technology was primitive. Tacking a text macro preprocessor onto the front end was a cheap and easy way to add many powerful features. The increasing size & complexity of programs have illustrated that these features come at a stiff price. D doesn't have a preprocessor; but D provides a more scalable means to solve the same problems.

Header Files

C and C++ rely heavilly on textual inclusion of header files. This frequently results in the compiler having to recompile tens of thousands of lines of code over and over again for every source file, an obvious source of slow compile times. What header files are normally used for is more appropriately done doing a symbolic, rather than textual, insertion. This is done with the import statement. Symbolic inclusion means the compiler just loads an already compiled symbol table. The needs for macro "wrappers" to prevent multiple #inclusion, funky #pragma once syntax, and incomprehensible fragile syntax for precompiled headers are simply unnecessary and irrelevant to D.

The syntax:

```
#include <stdio.h>
```

is expressed in D as:

```
import stdio;
```

#pragma once

Is rendered irrelevant by the import statement.

#pragma pack

This is used in C to adjust the alignment for structs. 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:

1. Macros have no concept of scope; they are valid from the point of definition to the end of the source. They cut a swath across .h files, nested code, etc. When #include'ing tens of thousands of lines of macro definitions, it becomes problematicalto avoid inadvertent macro expansions.

2. Macros are unknown to the debugger. Trying to debug a program with symbolic data is undermined by the debugger only knowing about macro expansions, not themacros themselves.

3. Macros make it impossible to tokenize source code, as an earlier macro change can arbitrarilly redo tokens.

4. 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++.)

5. Macros are still used to make up for deficits in the language's expressive capabiltiy, such as for "wrappers" around header files.

Here's an enumeration of the common uses for macros, and the corresponding feature in D:

1. Macros as constants:

```
#define VALUE        5
```

In D:

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

2. Macros to protect a .h file from multiple inclusions:

```
#ifndef STDIO_H
#define STDIO_H 1
        ...
#endif // STDIO_H
```

In D:

```
import stdio;
```

3. Macros as a list of values or flags:

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

In D:

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

4. Macros to distinguish between ascii chars and unicode chars:

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

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

In D:

```
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.

5. Macros to support legacy C compilers:

```
#if PROTOTYPES
```

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

D doesn't have legacy compilers, and so doesn't need legacy support. (yet!)

6. Macros as typedefs:

```
#define INT     int
```

In D:

```
typealias int INT;
```

7. Macros to change storage class for declaration vs definition:

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

In declarations.h:

EXTERN int foo;
```

In D, 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.

8. Macros as lightweight inline functions:

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

In D:

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

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

9. Macros to pass the assert function file and line number information:

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

In D, 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.

10. Macros to change function calling conventions:

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

int _CRTAPI2 func();
```

D determines the optimal calling convention for you, so there is no need to override the default or to fiddle with it with macros.

11. Macros to hide __near or __far pointer wierdness:

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

D doesn't support 16 bit code or mixed pointer sizes, and so the problem is just irrelevant. Good riddance. Of course, this problem may return with mixed 64 bit and 32 bit code.

FEATURES TO THINK ABOUT

1. Integration with HTML and XML source.
2. Support for microversioning.
3. Make strings edittable after the fact (so the program can be internationalized by others without requiring recompiling).
4. .
5. Compiler switch <ugh> to make it an error to mix ascii and unicode.
6. How to implement "copy on write" for arrays.
7. PeterZ's idea for functions that return multiple values:

```
(int,long) func(int x) { return (x,5); }
(i,j) = func(3);
```

8. JeffO's idea of a "runonce" modifier for functions that only get run once.
9. Allocate temporaries with alloca() rather than garbage collected heap.
10. Eric's idea for error returns from functions and error handlers.