D Programming Language

Walter

22nd January 2004

Contents

CHAPTER 1

Introduction

1.1. 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!

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

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

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.

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

Alternate versions of this document:

- Alexander Klinsky has prepared a pdffile, though it's a bit out of date.
- Kazuhiro Inaba has prepared a Japanesetranslation.

-Walter

1.2. Overview

1.2.1. 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.

1.2.2. Why D?

Why, indeed. Who needs another programming language?

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

and the C++ standard is about 750 pages! C++ is a difficult and costly language to implement, resulting in implementation variations that make it frustrating to write fully portable C++ code.

C++ programmers tend to program in particular islands of the language, i.e. getting very proficient using certain features while avoiding other feature sets. While the code is usually portable from compiler to compiler, it can be hard to port it from programmer to programmer. A great strength of C++ is that it can support many radically different styles of programming - but in long term use, the overlapping and contradictory styles are a hindrance.

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

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

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

Major Goals of D

- Reduce software development costs by at least 10% by adding in proven productivity enhancing features and by adjusting language features so that common, time-consuming bugs are eliminated from the start.
- Make it easier to write code that is portable from compiler to compiler, machine to machine, and operating system to operating system.
- Support multi-paradigm programming, i.e. at a minimum support imperative, structured, object oriented, and generic programming paradigms.
- Have a short learning curve for programmers comfortable with programming in C or C++.
- Provide low level bare metal access as required.
- Make D substantially easier to implement a compiler for than C++.
- Be compatible with the local C application binary interface.
- Have a context-free grammar.
- Easilly support writing internationalized applications.

- Incorporate Design by Contract and unit testing methodology.
- Be able to build lightweight, standalone programs.

Features To Keep From C/C++

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

Using D will not mean that the programmer will become restricted to a specialized runtime vm (virtual machine) like the Java vm or the Smalltalk vm. There is no D vm, it's a straightforward compiler that generates linkable object files. D connects to the operating system just like C does. The usual familiar tools like **make** will fit right in with D development.

- The general **look and feel** of C/C++ will be maintained. It will use the same algebraic syntax, most of the same expression and statement forms, and the general layout.
- D programs can be written either in C style **function-and-data** or in C++ style **object-oriented**, or any mix of the two.
- The **compile/link/debug** development model will be carried forward, although nothing precludes D from being compiled into bytecode and interpreted.
- **Exception handling**. More and more experience with exception handling shows it to be a superior way to handle errors than the C traditional method of using error codes and errno globals.
- **Runtime Type Identification**. This is partially implemented in C++; in D it is taken to its next logical step. Fully supporting it enables better garbage collection, better debugger support, more automated persistence, etc.
- D maintains function link compatibility with the **C calling conventions**. This makes it possible for D programs to access operating system API's directly. Programmers' knowledge and experience with existing programming API's and paradigms can be carried forward to D with minimal effort.
- **Operator overloading**. D programs can overload operators enabling extension of the basic types with user defined types.
- **Templates**. Templates are a way to implement generic programming. Other ways include using macros or having a variant data type. Using macros is out. Variants are straightforward, but inefficient and lack type checking. The difficulties with C++ templates are their complexity, they

don't fit well into the syntax of the language, all the various rules for conversions and overloading fitted on top of it, etc. D offers a much simpler way of doing templates.

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

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

Features To Drop

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

- Link compatibility with C++. The C++ runtime object model is just too complicated - properly supporting it would essentially imply making D a full C++ compiler too.

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

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

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

- Tag name space. This misfeature of C is where the tag names of struct's are in a separate but parallel symbol table. C++ attempted to merge the

tag name space with the regular name space, while retaining backward compatibility with legacy C code. The result is not printable.

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

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

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

- Trigraphs and digraphs. Unicode is the future.

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

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

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

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

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

- Compiler complexity. Reducing the complexity of an implementation makes it more likely that multiple, *correct* implementations are available.
- Distinction between. and ->. This distinction is really not necessary. The. operator serves just as well for pointer dereferencing.

Who D is For

- Programmers who routinely use lint or similar code analysis tools to eliminate bugs before the code is even compiled.
- People who compile with maximum warning levels turned on and who instruct the compiler to treat warnings as errors.
- Programming managers who are forced to rely on programming style guidelines to avoid common C bugs.
- Those who decide the promise of C++ object oriented programming is not fulfilled due to the complexity of it.
- Programmers who enjoy the expressive power of C++ but are frustrated by the need to expend much effort explicitly managing memory and finding pointer bugs.
- Projects that need built-in testing and verification.
- Teams who write apps with a million lines of code in it.
- Programmers who think the language should provide enough features to obviate the continual necessity to manipulate pointers directly.
- Numerical programmers. D has many features to directly support features needed by numerics programmers, like direct support for the complex data type and defined behavior for NaN's and infinities. (These are added in the new C99 standard, but not in C++.)
- D's lexical analyzer and parser are totally independent of each other and of the semantic analyzer. This means it is easy to write simple tools to manipulate D source perfectly without having to build a full compiler. It also means that source code can be transmitted in tokenized form for specialized applications.

Who D is Not For

- Realistically, nobody is going to convert million line C or C++ programs into D, and since D does not compile unmodified C/C++ source code, D is not for legacy apps. (However, D supports legacy C API's very well.)

- Very small programs - a scripting or interpreted language like Python, DMDScript, or Perl is likelymore suitable.
- As a first programming language - Basic or Java is more suitable for beginners. D makes an excellent second language for intermediate to advanced programmers.
- Language purists. D is a practical language, and each feature of it is evaluated in that light, rather than by an ideal. For example, D has constructs and semantics that virtually eliminate the need for pointers for ordinary tasks. But pointers are still there, because sometimes the rules need to be broken. Similary, casts are still there for those times when the typing system needs to be overridden.

1.2.3. 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 heirarchy, so all classes implement a common set of functionality. Classes are instantiated by reference, and so complex code to clean up after exceptions is not required.

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

Productivity

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

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

Example:

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

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

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

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

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

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

But the equivalent D code will work:

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

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

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

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

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

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

```
typedef int handle;
```

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

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

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

```
bit[] foo;
```

Functions

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

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

Function Literals Anonymous functions can be embedded directly into an expression.

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

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

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

Arrays

C arrays have several faults that can be corrected:

- Dimension information is not carried around with the array, and so has to be stored and passed separately. The classic example of this are the argc and argv parameters to `main(int argc, char *argv[])`. (In D, main is declared as `main(char[][] args)`.)
- Arrays are not first class objects. When an array is passed to a function, it is converted to a pointer,even though the prototype confusingly says it's an array. When this conversion happens, all array type information gets lost.
- C arrays cannot be resized. This means that even simple aggregates like a stack need to be constructed as a complex class.
- C arrays cannot be bounds checked, because they don't know what the array bounds are.
- Arrays are declared with the [] after the identifier. This leads to very clumsy syntax to declare things like a pointer to an array:

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

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

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

which is much simpler to understand.

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

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

Resource Management

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

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

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

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

For a fuller discussion of this, see garbage collection.

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

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

Performance

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

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

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

Reliability

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

Contracts 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 inadvertantly broken. The unit tests form part of the source code for a class. Creating them becomes a natural part of the class development process, as opposed to throwing the finished code over the wall to the testing group.

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

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

Along with 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.

1.2.4. Sample D Program (sieve.d)

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

bit[8191] flags;

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

    printf("10 iterations n");
    for (iter = 1; iter <= 10; iter++)
    {   count = 0;
        flags[] = 1;
        for (i = 0; i < flags.length; i++)
        {   if (flags[i])
            {   prime = i + i + 3;
                k = i + prime;
                while (k < flags.length)
                {
                    flags[k] = 0;
                    k += prime;
                }
                count += 1;
```

```
            }
        }
    }
    printf (" n%d primes", count);
    return 0;
}
```

CHAPTER 2

The Language

2.1. 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 readilly 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. **source character set**

   The source file is checked to see what character set it is, and the appropriate scanner is loaded. ASCII and UTF formats are accepted.

2. **lexical analysis**

   The source file is divided up into a sequence of tokens. Special tokens are processed and removed.

3. **syntax analysis**

   The sequence of tokens is parsed to form syntax trees.

4. **semantic analysis**

   The syntax trees are traversed to declare variables, load symbol tables, assign types, and in general determine the meaning of the program.

5. **optimization**

6. **code generation**

Source Text

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

- ASCII
- 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:
        IdentiferStart
        IdentiferStart  IdentifierChars

IdentifierChars:
        IdentiferChar
        IdentiferChar  IdentifierChars

IdentifierStart:
        _
        Letter
        UniversalAlpha

IdentifierChar:
        IdentiferStart
        Digit
```

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

String Literals

```
StringLiteral:
        WysiwygString
        AlternateWysiwygString
        DoubleQuotedString
        EscapeSequence
        HexString

WysiwygString:
        r"  WysiwygCharacters "

AlternateWysiwygString:
        `  WysiwygCharacters `

WysiwygCharacter:
        Character
```

```
        EndOfLine

DoubleQuotedString:
        "  DoubleQuotedCharacters "

DoubleQuotedCharacter:
        Character
        EscapeSequence
        EndOfLine

EscapeSequence:
```

```
        '
        "
        ?

        a
        b
        f
        n
        r
        t
        v
          EndOfFile
        x  HexDigit HexDigit
          OctalDigit
          OctalDigit OctalDigit
          OctalDigit OctalDigit OctalDigit
        u  HexDigit HexDigit HexDigit HexDigit
        U  HexDigit HexDigit HexDigit HexDigit HexDigit HexDigit HexDigit HexDigit

HexString:
        x"  HexStringChars "

HexStringChar
        HexDigit
        WhiteSpace
        EndOfLine
```

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

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

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

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

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

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

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

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

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

Escape sequences not listed above are errors.

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

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

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

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

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

The following are all equivalent:

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

Character Literals

```
CharacterLiteral:
        '  SingleQuotedCharacter '

SingleQuotedCharacter
        Character
        EscapeSequence
```

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

Integer Literals

```
IntegerLiteral:
        Integer
        Integer IntegerSuffix

Integer:
        Decimal
        Binary
        Octal
        Hexadecimal
        Integer _

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

Decimal:
        0
        NonZeroDigit
```

```
        NonZeroDigit Decimal
        NonZeroDigit _  Decimal

Binary:
        0b  BinaryDigits
        0B  BinaryDigits

Octal:
        0  OctalDigits

Hexadecimal:
        0x  HexDigits
        0X  HexDigits
```

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

Decimal integers are a sequence of decimal digits.

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

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

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

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

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

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

The type of the integer is resolved as follows:

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
        Float _

FloatSuffix:
        f
        F
        l
        L
        i
```


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

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

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

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

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

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

Examples:

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

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

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

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

Keywords

Keywords are reserved identifiers.

Keyword:

```
abstract
alias
align
asm
assert
auto

bit
body
break
byte

case
cast
catch
cent
char
class
cfloat
cdouble
creal
const
continue

dchar
debug
default
delegate
delete
deprecated
do
double

else
enum
export
extern

false
final
finally
float
for
foreach
function

goto
```

```
idouble
if
ifloat
import
in
inout
int
interface
invariant
ireal
is

long

module

new
null

out
override

pragma
private
protected
public

real
return

short
static
struct
super
switch
synchronized

template
this
throw
true
try
typedef
typeof

ubyte
ucent
uint
ulong
union
unittest
ushort

version
```

```
void
volatile

wchar
while
with
```

Tokens

Token:
Identifier
StringLiteral
IntegerLiteral
FloatLiteral
Keyword
/
/=
.
..
...
 &
 &=
 & &
|
|=
||
-
-=
--
+
+=
++
<
<=
<<
<<=
<>
<>=
>
>=
>>=
>>>=
>>
>>>
!
!=
!==
!<>
!<>=
!<
!<=
!>

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

Special Token Sequences

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

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

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

Filespec
        "  Characters "
```

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

For example:

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

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

2.2. 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 std.c.stdio;  // import module stdio  from the c  package
import foo, bar;     // import modules foo  and bar
```

Scope and Modules

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

For example, assume the following modules:

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

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

then:

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

and:

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

and:

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

and:

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

If the import is private, such as:

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

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

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

```
int x;

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

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

2.2.1. 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.

2.3. 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

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

IdentifierList
```

```
        Identifier
        Identifier . IdentifierList

Typeof
        typeof ( Expression )
```

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, read left to right:

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

Pointers to functions are declared 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; }
alias Foo2!(int) t1;
alias Foo2!(int).t t2;
alias t1.t t3;
alias t2 t4;

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

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

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

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

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

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

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

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

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

class D : C  {
}

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

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

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

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

The distinction is made in the semantic analysis pass.

typeof

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

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

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

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

2.4. Types

2.4.1. 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 UTF-8
wchar	unsigned 16 bit UTF-16
dchar	unsigned 32 bit UTF-32

2.4.2. Derived Data Types

- pointer
- array
- function

2.4.3. User Defined Types

- alias
- typedef

- enum
- struct
- union
- class

2.4.4. 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.

2.4.5. 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:
   (a) If both are the same type, no more conversions are done.
   (b) If both are signed or both are unsigned, the smaller type is converted to the larger.
   (c) If the signed type is larger than the unsigned type, the unsigned type is converted to the signed type.
   (d) The signed type is converted to the unsigned type.

2.4.6. 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)
```

2.5. Properties

Every type and expression has properties that can be queried:

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

2.5.1. Properties for Integral Data Types

```
.init           initializer (0)
.sizeof         size in bytes (equivalent to C's sizeof(type))
.size           alternative to.sizeof
.alignof        alignment size
.max            maximum value
.min            minimum value
.sign           should we do this?
```

2.5.2. Properties for Floating Point Types

```
.init           initializer (NaN)
.sizeof         size in bytes (equivalent to C's sizeof(type))
.size           alternative to.sizeof
.alignof        alignment size
.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
```

2.5.3. .init Property

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

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

int.init        // is 0
a.init          // is 0
b.init          // is 1
t.init          // is 2
c.init          // is 2
d.init          // is 3

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

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

Class and Struct Properties

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

A simple property would be:

```
struct Foo
{
```

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

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

  private:
    int m_data;
}
```

To use it:

```
int test()
{
    Foo f;

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

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

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

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

2.6. Attributes

```
AttributeSpecifier:
    Attribute :
    Attribute DeclDefBlock
    Pragma ;

AttributeElseSpecifier:
    AttributeElse :
    AttributeElse DeclDefBlock
    AttributeElse DeclDefBlock else  DeclDefBlock

Attribute:
    LinkageAttribute
    AlignAttribute
    Pragma
    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

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

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

For attributes with an optional else clause:

```
attribute
    declaration;
else
    declaration;

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

```
    declaration;
   ...
}
```

2.6.1. 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
    );
```

2.6.2. 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.

2.6.3. Deprecated Attribute

It is often necessary to deprecate a feature in a library, yet retain it for backwards compatiblity. 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.

2.6.4. 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.

2.6.5. 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;
}
```

2.6.6. 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
    }
}
```

2.6.7. 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.

2.6.8. Auto Attribute

```
auto
```

The auto attribute is used for local variables and for class declarations. For class declarations, the auto attribute creates an *auto* class. For local declarations, **auto** implements the RAII (Resource Acquisition Is Initialization) protocol. This means that the destructor for an object is automatically called when the auto reference to it goes out of scope. The destructor is called even if the scope is exited via a thrown exception, thus auto is used to guarantee cleanup.

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

2.7. Pragmas

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

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

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

pragma(ident) declaration; // influence one declaration

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

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

pragma(ident) statement; // influence one statement

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

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

2.7.1. Predefined Pragmas

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

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

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

2.7.2. Vendor Specific Pragmas

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

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

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

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

2.8. 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 is 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
        . Identifier
        this
        super
        null
        true
        false
        NumericLiteral
        CharacterLiteral
        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
```

2.8.1. 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.

2.8.2. 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.

2.8.3. 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.

2.8.4. 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.

2.8.5. 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.

2.8.6. 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.

2.8.7. 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.

2.8.8. 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(). If one or the other or both objects are null, an exception is raised.

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

2.8.9. Identity Expressions

```
RelExpression is RelExpression
```

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

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

For class objects, identity is defined as the object references are for the same object. Null class objects can be compared with **is**.

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

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

2.8.10. **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.

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:

<caption>Floating point comparison operators</caption>

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.

2.8.11. 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
```

2.8.12. 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.

2.8.13. **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.

2.8.14. **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 < type >(expression)
```

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

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

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

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

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

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

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)
```

2.8.15. **Postfix Expressions**

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

2.8.16. **Primary Expressions**

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

.Identifier

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

this

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

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.

Character Literals

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

Function Literals

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

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

2.9. Statements

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

```
Statement:
        LabeledStatement
        BlockStatement
        ExpressionStatement                DeclarationStatement                IfStatement
        DebugStatement
        VersionStatement
        WhileStatement
        DoWhileStatement
        ForStatement
        ForeachStatement
        SwitchStatement
        CaseStatement
        DefaultStatement
        ContinueStatement
        BreakStatement
        ReturnStatement
        GotoStatement
        WithStatement
        SynchronizeStatement               TryStatement
        ThrowStatement
        VolatileStatement
        AsmStatement
        PragmaStatement
```

2.9.1. 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.

2.9.2. Block Statement

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

```
BlockStatement:
        { }
        {  StatementList }

StatementList:
        Statement
        Statement  StatementList
```

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

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

void func2()
{
    int x;

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

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

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

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

2.9.3. Expression Statement

The expression is evaluated.

```
ExpressionStatement:
        Expression ;
```

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

2.9.4. 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.

2.9.5. 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.

2.9.6. 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 evaluationg *Expression* again.

2.9.7. Do-While Statement

Do-While statements implement simple loops.

```
DoStatement:
        do Statement while ( Expression )
```

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

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

2.9.8. 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.

2.9.9. Foreach Statement

A foreach statement loops over the contents of an aggregate.

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

ForeachTypeList:
        ForeachType
        ForeachType, ForeachTypeList

ForeachType:
        inout  Type Identifier
        Type Identifier
```

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

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

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

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

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

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

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

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

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

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

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

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

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

An example using this might be:

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

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

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

which would print:

```
73
82
```

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

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

which would print:

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

inout can be used to update the original elements:

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

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

which would print:

```
8
9
```

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

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

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

2.9.10. Switch Statement

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

```
SwitchStatement:
        switch ( Expression ) BlockStatement

CaseStatement:
        case ExpressionList : Statement

DefaultStatement:
        default: Statement
```

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

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

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

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

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

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

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

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

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

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

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

will set x to 4 if i is 1.

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

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

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

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

2.9.11. 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.

2.9.12. 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.

2.9.13. 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.

2.9.14. Goto Statement

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

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

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

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

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

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

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

2.9.15. 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.

2.9.16. 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.

2.9.17. 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.

2.9.18. 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.

2.9.19. 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.

2.9.20. 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 followthe 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...
}
```

2.10. Arrays

There are four kinds of arrays:

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

2.10.1. 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.

2.10.2. **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
```

2.10.3. **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];
```

2.10.4. 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.

2.10.5. 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
```

2.10.6. 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
```

2.10.7. 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
}
```

2.10.8. **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];
```

2.10.9. **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 concatenting 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.

2.10.10. 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.

2.10.11. 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.

2.10.12. Special Array Types

Arrays of Bits

Bit vectors can be constructed:

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

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

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

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

Strings

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

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

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

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

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

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

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

A pointer to a char can be generated:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Implicit Conversions

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

- *U** where *U* is a base class of *T*.
- `void*`

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

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

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

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

2.10.13. Associative Arrays

D goes one step further with arrays - adding associative arrays. Associative arrays have an index that is not necessarilly 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 std.file;            // D file I/O

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

    printf("   lines   words   bytes file n");
    for (int i = 1; i < args.length; ++i)        // program arguments
    {
        char[] input;               // input buffer
        int w_cnt, l_cnt, c_cnt;    // word, line, char counts
        int inword;
        int wstart;
```

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

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

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

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

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

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

2.11. Structs, Unions, Enums

2.11.1. 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.

Struct Properties

```
.sizeof                 Size in bytes of struct
.size                   Same as.sizeof
.alignof                Size boundary struct needs to be aligned on
```

2.11.2. 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

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

For example:

X.min                    is X.A
X.max                    is X.C
X.sizeof                 is same as int.sizeof
```

Initialization of Enums

In the absense 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
```

2.12. 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:
        {  ClassBodyDeclarations }

ClassBodyDeclaration:
        Declaration
        Constructor
        Destructor
        StaticConstructor
        StaticDestructor
        Invariant
        UnitTest
        ClassAllocator
        ClassDeallocator
```

Classes consist of:

- super class
- interfaces
- dynamic fields
- static fields
- types

- functions
  - static functions
  - dynamic functions
  - constructors
  - destructors
  - static constructors
  - static destructors
  - invariants
  - unit tests
  - allocators
  - deallocators

A class is defined:

```
class Foo
{
   ... members...
}
```

Note that there is no trailing ; after the closing } of the class definition. It is also not possible to declare a variable var like:

```
class Foo { }  var;
```

Instead:

```
class Foo { }
Foo var;
```

Fields

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

The D compiler is free to rearrange the order of fields in a class to optimally pack them in an implementation-defined manner. 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.

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

```
Constructor:
        this()  BlockStatement
```

Members are always initialized to the default initializer for their type, which is usually 0 for integer types and NAN for floating point types. This eliminates an entire class of obscure problems that come from neglecting to initialize a member in one of the constructors. In the class definition, there can be a static initializer to be used instead of the default:

```
class Abc
{
    int a;      // default initializer for a is 0
    long b = 7; // default initializer for b is 7
    float f;    // default initializer for f is NAN
}
```

This static initialization is done before any constructors are called.

Constructors are defined with a function name of **this** and having no return value:

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

Base class construction is done by calling the base class constructor by the name **super** :

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

class B : A
{
    int j;
    this()
```

```
    {
       ...
        super (3);        // call base constructor A.this(3)
       ...
    }
}
```

Constructors can also call other constructors for the same class in order to share common initializations:

```
class C
{
    int j;
    this()
    {
       ...
    }
    this(int i)
    {
        this ();
        j = i;
    }
}
```

If no call to constructors via **this** or **super** appear in a constructor, and the base class has a constructor, a call to **super** () is inserted at the beginning of the constructor.

If there is no constructor for a class, but there is a constructor for the base class, a default constructor of the form:

```
this() { }
```

is implicitly generated.

Class object construction is very flexible, but some restrictions apply:

1. It is illegal for constructors to mutually call each other:

```
this() { this(1); }
this(int i) { this(); }  // illegal, cyclic constructor calls
```

2. If any constructor call appears inside a constructor, any path through the constructor must make exactly one constructor call:

```
this()  { a || super(); }        // illegal

this() { this(1) || super(); }   // ok
```

```
this()
{
    for (...)
    {
        super();        // illegal, inside loop
    }
}
```

3. It is illegal to refer to **this** implicitly or explicitly prior to making a constructor call.

4. Constructor calls cannot appear after labels (in order to make it easy to check for the previous conditions in the presence of goto's).

Instances of class objects are created with *NewExpression* s:

```
A a = new A(3);
```

The following steps happen:

1. Storage is allocated for the object. If this fails, rather than return **null**, an **OutOfMemoryException** is thrown. Thus, tedious checks for null references are unnecessary.

2. The raw data is statically initialized using the values provided in the class definition. The pointer to the vtbl is assigned. This ensures that constructors are passed fully formed objects. This operation is equivalent to doing a memcpy() of a static version of the object onto the newly allocated one, although more advanced compilers may be able to optimize much of this away.

3. If there is a constructor defined for the class, the constructor matching the argument list is called.

4. If class invariant checking is turned on, the class invariant is called at the end of the constructor.

Destructors

```
Destructor:
        ~this()  BlockStatement
```

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

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

There can be only one destructor per class, the destructor does not have any parameters, and has no attributes. It is always virtual.

The destructor is expected to release any resources held by the object.

The program can explicitly inform the garbage collector that an object is no longer referred to (with the delete expression), and then the garbage collector calls the destructor immediately, and adds the object's memory to the free storage. The destructor is guaranteed to never be called twice.

The destructor for the super class automatically gets called when the destructor ends. There is no way to call the super destructor explicitly.

Static Constructors

```
StaticConstructor:
        static this()  BlockStatement
```

A static constructor is defined as a function that performs initializations before the main() function gets control. Static constructors are used to initialize static class members with values that cannot be computed at compile time.

Static constructors in other languages are built implicitly by using member initializers that can't be computed at compile time. The trouble with this stems from not having good control over exactly when the code is executed, for example:

```
class Foo
{
    static int a = b + 1;
    static int b = a * 2;
}
```

What values do a and b end up with, what order are the initializations executed in, what are the values of a and b before the initializations are run, is this a compile error, or is this a runtime error? Additional confusion comes from it not being obvious if an initializer is static or dynamic.

D makes this simple. All member initializations must be determinable by the compiler at compile time, hence there is no order-of-evaluation dependency for member initializations, and it is not possible to read a value that has not been initialized. Dynamic initialization is performed by a static constructor, defined with a special syntax `static this()`.

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

    static this()               // static constructor
    {
        a = b + 1;              // a is set to 2
        b = a * 2;              // b is set to 4
    }
}
```

`static this()` is called by the startup code before `main()` is called. If it returns normally (does not throw an exception), the static destructor is added to the list of 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

```
StaticDestructor:
        static ~this()  BlockStatement
```

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

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

A static 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

```
ClassInvariant:
        invariant  BlockStatement
```

Class invariants are used to specify characteristics of a class that always must be true (except while executing a member function). For example, a class representing a date might have an invariant that the day must be 1..31 and the hour must be 0..23:

```
class Date
{
    int day;
    int hour;

    invariant
    {
        assert(1 <= day  & & day <= 31);
        assert(0 <= hour  & & hour < 24);
    }
}
```

The class invariant is a contract saying that the asserts must hold true. The invariant is checked when a class constructor completes, at the start of the class destructor, before a public or exported member is run, and after a public or exported function finishes. The invariant can be checked when a class object is the argument to an `assert()` expression, as:

```
Date mydate;
...
assert(mydate);         // check that class Date invariant holds
```

If the invariant fails, it throws an `InvariantException`. Class invariants are inherited, that is, any class invariant is implicitly anded with the invariants of its base classes.

There can be only one *ClassInvariant* per class.

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

Unit Tests

```
UnitTest:
        unittest  BlockStatement
```

Unit tests are a series of test cases applied to a class to determine if it is working properly. Ideally, unit tests should be run every time a program is compiled. The best way to make sure that unit tests do get run, and that they are maintained along with the class code is to put the test code right in with the class implementation code.

D classes can have a special member function called:

```
unittest
{
   ...test code...
}
```

The test() functions for all the classes in the program get called after static initialization is done and before the main function is called. A compiler or linker switch will remove the test code from the final build.

For example, given a class Sum that is used to add two values:

```
class Sum
{
    int add(int x, int y) { return x + y; }

    unittest
    {
        assert(add(3,4) == 7);
        assert(add(-2,0) == -2);
    }
}
```

Class Allocators

```
ClassAllocator:
        new  ParameterList BlockStatement
```

A class member function of the form:

```
new(uint size)
{
   ...
}
```

is called a class allocator. The class allocator can have any number of parameters, provided the first one is of type uint. Any number can be defined for a class, the correct one is determined by the usual function overloading rules. When a new expression:

```
new Foo;
```

is executed, and Foo is a class that has an allocator, the allocator is called with the first argument set to the size in bytes of the memory to be allocated for the instance. The allocator must allocate the memory and return it as a `void*`. If the allocator fails, it must not return a **null**, but must throw an exception. If there is more than one parameter to the allocator, the additional arguments are specified within parentheses after the **new** in the *NewExpression*:

```
class Foo
{
    this(char[] a) {... }

    new(uint size, int x, int y)
    {
       ...
    }
}

...

new(1,2) Foo(a);        // calls new(Foo.size,1,2)
```

Derived classes inherit any allocator from their base class, if one is not specified.

See also Explicit Class Instance Allocation.

Class Deallocators

```
ClassDeallocator:
        delete  ParameterList  BlockStatement
```

A class member function of the form:

```
delete(void *p)
{
   ...
}
```

is called a class deallocator. The deallocator must have exactly one parameter of type `void*`. Only one can be specified for a class. When a delete expression:

```
delete f;
```

is executed, and f is a reference to a class instance that has a deallocator, the deallocator is called with a pointer to the class instance after the destructor (if any) for the class is called. It is the responsibility of the deallocator to free the memory.

Derived classes inherit any deallocator from their base class, if one is not specified.

See also Explicit Class Instance Allocation.

Auto Classes

An auto class is a class with the auto attribute, as in:

```
auto class Foo {... }
```

The auto characteristic is inherited, so if any classes derived from an auto class are also auto.

An auto class reference can only appear as a function local variable. It must be declared as being **auto** :

```
auto class Foo {... }

void func()
{
    Foo f;      // error, reference to auto class must be auto
    auto Foo g = new Foo();     // correct
}
```

When an auto class reference goes out of scope, the destructor (if any) for it is automatically called. This holds true even if the scope was exited via a thrown exception.

2.13. 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
```

2.14. Functions

Virtual Functions

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

Functions marked as `final` may not be overridden in a derived class, unless they are also `private`. For example:

```
class A
{
    int def() {... }
    final int foo() {... }
    final private int bar() {... }
    private int abc() {... }
}

class B : A
{
    int def() {... }    // ok, overrides A.def
    int bar() {... }    // error, A.bar is final
    int foo() {... }    // ok, A.foo is final private, but not virtual
    int abc() {... }    // ok, A.abc is not virtual, B.abc is virtual
}
```

```
void test(A a)
{
    a.def();    // calls B.def
    a.foo();    // calls A.foo
    a.bar();    // calls A.bar
    a.abc();    // calls A.abc
}

void func()
{   B b = new B();
    test(b);
}
```

Covariant return types are supported, which means that the overriding function in a derived class can return a type that is derived from the type returned by the overridden function:

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

class Foo
{
    A test() { return null; }
}

class Bar : Foo
{
    B test() { return null; }    // overrides and is covariant with Foo.test()
}
```

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.

2.14.1. **Nested Functions**

Functions may be nested within other functions:

```
int bar(int a)
{
    int foo(int b)
    {
        int abc() { return 1; }

        return b + abc();
    }
    return foo(a);
}

void test()
{
    int i = bar(3);     // i is assigned 4
}
```

Nested functions can only be accessed by the most nested lexically enclosing function, or by another nested function at the same nesting depth:

```
int bar(int a)
{
    int foo(int b) { return b + 1; }
    int abc(int b) { return foo(b); }    // ok
    return foo(a);
}

void test()
{
```

```
    int i = bar(3);     // ok
    int j = bar.foo(3); // error, bar.foo not visible
}
```

Nested functions have access to the variables and other symbols defined by the lexically enclosing function. This access includes both the ability to read and write them.

```
int bar(int a)
{   int c = 3;

    int foo(int b)
    {
        b += c;         // 4 is added to b
        c++;            // bar.c is now 5
        return b + c;   // 12 is returned
    }
    c = 4;
    int i = foo(a);     // i is set to 12
    return i + c;       // returns 17
}

void test()
{
    int i = bar(3);     // i is assigned 17
}
```

This access can span multiple nesting levels:

```
int bar(int a)
{   int c = 3;

    int foo(int b)
    {
        int abc()
        {
            return c;   // access bar.c
        }
        return b + c + abc();
    }
    return foo(3);
}
```

Static nested functions cannot access any stack variables of any lexically enclosing function, but can access static variables. This is analogous to how static member functions behave.

```
int bar(int a)
{   int c;
```

```
    static int d;

    static int foo(int b)
    {
        b = d;          // ok
        b = c;          // error, foo() cannot access frame of bar()
        return b + 1;
    }
    return foo(a);
}
```

Functions can be nested within member functions:

```
struct Foo
{   int a;

    int bar()
    {   int c;

        int foo()
        {
            return c + a;
        }
    }
}
```

Member functions of nested classes and structs do not have access to the stack variables of the enclosing function, but do have access to the other symbols:

```
void test()
{   int j;
    static int s;

    struct Foo
    {   int a;

        int bar()
        {   int c = s;          // ok, s is static
            int d = j;          // error, no access to frame of test()

            int foo()
            {
                int e = s;      // ok, s is static
                int f = j;      // error, no access to frame of test()
                return c + a;   // ok, frame of bar() is accessible,
                                // so are members of Foo accessible via
                                // the 'this' pointer to Foo.bar()
            }
        }
```

```
    }
}
```

Nested functions always have the D function linkage type.

Unlike module level declarations, declarations within function scope are processed in order. This means that two nested functions cannot mutually call each other:

```
void test()
{
    void foo() { bar(); }        // error, bar not defined
    void bar() { foo(); }        // ok
}
```

The solution is to use a delegate:

```
void test()
{
    void delegate() fp;
    void foo() { fp(); }
    void bar() { foo(); }
    fp =  &bar;
}
```

Future directions: This restriction may be removed.

Delegates, Function Pointers, and Dynamic Closures

A function pointer can point to a static nested function:

```
int function() fp;

void test()
{   static int a = 7;
    static int foo() { return a + 3; }

    fp =  &foo;
}

void bar()
{
    test();
    int i = fp();       // i is set to 10
}
```

A delegate can be set to a non-static nested function:

```
int delegate() dg;

void test()
{   int a = 7;
    int foo() { return a + 3; }

    dg =  &foo;
    int i = dg();       // i is set to 10
}
```

The stack variables, however, are not valid once the function declaring them has exited, in the same manner that pointers to stack variables are not valid upon exit from a function:

```
int* bar()
{   int b;
    test();
    int i = dg();       // error, test.a no longer exists
    return  &b;          // error, bar.b not valid after bar() exits
}
```

Delegates to non-static nested functions contain two pieces of data: the pointer to the stack frame of the lexically enclosing function (called the *frame pointer*) and the address of the function. This is analogous to struct/class non-static member function delegates consisting of a *this* pointer and the address of the member function. Both forms of delegates are interchangeable, and are actually the same type:

```
struct Foo
{   int a = 7;
    int bar() { return a; }
}

int foo(int delegate() dg)
{
    return dg() + 1;
}

void test()
{
    int x = 27;
    int abc() { return x; }
    Foo f;
    int i;

    i = foo( &abc);      // i is set to 28
    i = foo( &f.bar);    // i is set to 8
}
```

This combining of the environment and the function is called a *dynamic closure*.

Future directions: Function pointerss and delegates may merge into a common syntax and be interchangable with each other.

2.15. Operator Overloading

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

2.15.1. Unary Operator Overloading

Overloadable Unary Operators

op	opfunc
-	opNeg
~	opCom
e++	opPostInc
e–	opPostDec

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

```
op a
```

where *a* is a class or struct object reference, is interpreted as if it was written as:

```
a.opfunc()
```

Overloading ++*e* and –*e*

Since ++*e* is defined to be semantically equivalent to (*e* += 1), the expression ++*e* is rewritten as (*e* += 1), and then checking for operator overloading is done. The situation is analogous for –*e*.

Examples

1.
```
class A { int opNeg (); }
A a;
-a;     // equivalent to a.opNeg();
```

2.
```
class A { int opNeg (int i); }
A a;
-a;     // equivalent to a.opNeg(), which is an error
```

2.15.2. Binary Operator Overloading

Overloadable Binary Operators

op	commutative?	opfunc	opfunc_r
+	yes	opAdd	-
-	no	opSub	opSub_r
	yes	opMul	-
/	no	opDiv	opDiv_r
ampr	yes	opAnd	-
|	yes	opOr	-
dasz	yes	opXor	-
«	no	opShl	opShl_r
»	no	opShr	opShr_r
»>	no	opUShr	opUShr_r
	no	opCat	opCat_r
==	yes	opEquals	-
!=	yes	opEquals	-
<	yes	opCmp	-
<=	yes	opCmp	-
>	yes	opCmp	-
>=	yes	opCmp	-
+=	no	opAddAssign	-
-=	no	opSubAssign	-
=	no	opMulAssign	-
/=	no	opDivAssign	-
ampr=	no	opAndAssign	-
|=	no	opOrAssign	-
dasz=	no	opXorAssign	-
«=	no	opShlAssign	-
»=	no	opShrAssign	-
»>=	no	opUShrAssign	-
~=	no	opCatAssign	-

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

   ```
   a.opfunc(b)
   ```

2. 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:

   ```
   b.opfunc_r(a)
   ```

3. 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:

   ```
   b.opfunc(a)
   ```

4. If *a* or *b* is a struct or class object reference, it is an error.

Examples

1. ```
   class A { int opAdd (int i); }
   A a;
   a + 1;  // equivalent to a.opAdd(1)
   ```

2. ```
   1 + a;  // equivalent to a.opAdd(1)
   ```

3. ```
   class B { int opDiv_r (int i); }
   B b;
   1 / b;  // equivalent to b.opDiv_r(1)
   ```
   ```

Overloading == and !=

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

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

```
int opEquals (Object o);
```

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

If a struct has no **opEquals** () function declared for it, a bit compare of the contents of the two structs is done to determine equality or inequality.

Overloading <, <=, > and >=

These comparison operators all use the `opCmp ()` function. The expression `(a` *op* `b)` is rewritten as `(a.opCmp (b)` *op* `0)`. The commutative operation is rewritten as `(0` *op* `b.opCmp (a))`

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

```
int opCmp (Object o);
```

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

If a struct has no **opCmp** () function declared for it, attempting to compare two structs is an error.

Note: Comparing a reference to a class object against `null` should be done as:

```
if (a === null)
```

and not as:

```
if (a == null)
```

The latter is converted to:

```
if (a.opCmp (null))
```

which will fail if `opCmp ()` is a virtual function.

Rationale The reason for having both **opEquals** () and **opCmp** () is that:

- Testing for equality can sometimes be a much more efficient operation than testing for less or greater than.
- For some objects, testing for less or greater makes no sense. For these, override **opCmp** () with:

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

2.15.3. Function Call Operator Overloading *f*()

The function call operator, (), can be overloaded by declaring a function named **opCall** :

```
struct F
{
    int opCall ();
    int opCall (int x, int y, int z);
}

void test()
{   F f;
    int i;

    i = f() ;             // same as i = f.opCall();
    i = f( 3,4,5) ; // same as i = a.opCall(3,4,5);
}
```

In this way a struct or class object can behave as if it were a function.

2.15.4. Array Operator Overloading

Overloading Indexing *a*[*i*]

The array index operator, [], can be overloaded by declaring a function named **opIndex** with one or two parameters. The method with one parameter is used as the rvalue, the method with two parameters is the lvalue:

```
struct A
{
    int opIndex (int i);
    int opIndex (int i, int value);
}

void test()
{   A a;
    int i;

    i = a[ 5] ;             // same as i = a.opIndex(5);
    a[ i]  = 7;             // same as a.opIndex(i,7);
}
```

In this way a struct or class object can behave as if it were an array.

Note: Array index overloading currently does not work for the lvalue of an *op*=, ++, or – operator.

Overloading Slicing *a*[] and *a*[*i*.. *j*]

Overloading the slicing operator means overloading expressions like `a[]` and `a[i.. j]`.

```
class A
{
    int opSlice ();               // overloads a[]
    int opSlice (int x, int y);   // overloads a[i.. j]
}

void test()
{   A a = new A();
    int i;

    i = a[] ;             // same as i = a.opSlice();
    i = a[ 3..4] ;  // same as i = a.opSlice(3,4);
}
```

2.15.5. Future Directions

The operators., & &, ||, ?:, and a few others will likely never be overloadable. The names of the overloaded operators may change.

2.16. Templates

Templates are D's approach to generic programming. Templates are defined with a *TemplateDeclaration*:

```
TemplateDeclaration:
        template  TemplateIdentifier (  TemplateParameterList )
                {  DeclDefs }

TemplateIdentifier:
        Identifier

TemplateParameterList
        TemplateParameter
        TemplateParameter, TemplateParameterList

TemplateParameter:
        TypeParameter
        ValueParameter
        AliasParameter

TypeParameter:
        Identifier
        Identifier :  Type

ValueParameter:
        Declaration
        Declaration :  AssignExpression

AliasParameter:
        alias  Identifier
```

The body of the *TemplateDeclaration* must be syntactically correct even if never instantiated. Semantic analysis is not done until instantiated. A template forms its own scope, and the template body can contain classes, structs, types, enums, variables, functions, and other templates.

Template parameters can be types, values, or symbols. Types can be any type. Value parameters must be of an integral type, and specializations for them must resolve to an integral constant. Symbols can be any non-local symbol.

2.16.1. Template Instantiation

Templates are instantiated with:

```
TemplateInstance:
        TemplateIdentifer !(  TemplateArgumentList )

TemplateArgumentList:
        TemplateArgument
        TemplateArgument, TemplateArgumentList

TemplateArgument:
        Type
        AssignExpression
        Symbol
```

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

```
template TFoo(T) { alias T* t; }
...
TFoo!(int).t x; // declare x to be of type int*
```

A template instantiation can be aliased:

```
template TFoo(T) { alias T* t; }
alias TFoo!(int) abc;
abc.t x;        // declare x to be of type int*
```

Multiple instantiations of a *TemplateDeclaration* with the same *TemplateParameterList* all will refer to the same instantiation. For example:

```
template TFoo(T) { T f; }
alias TFoo(int) a;
alias TFoo(int) b;
...
a.f = 3;
assert(b.f == 3);       // a and b refer to the same instance of TFoo
```

This is true even if the *TemplateInstance*s are done in different modules.

If multiple templates with the same *TemplateIdentifier* are declared, they are distinct if they have a different number of arguments or are differently specialized.

For example, a simple generic copy template would be:

```
template TCopy(T)
{
    void copy(out T to, T from)
    {
        to = from;
    }
}
```

To use the template, it must first be instantiated with a specific type:

```
int i;
TCopy!(int).copy(i, 3);
```

2.16.2. **Instantiation Scope**

*TemplateInstantance*s are always performed in the scope of where the *TemplateDeclaration* is declared, with the addition of the template parameters being declared as aliases for their deduced types.

For example:

```
-------- module a ---------
template TFoo(T) { void bar() { func(); }  }

-------- module b ---------
import a;

void func() { }
alias TFoo!(int) f;     // error: func not defined in module a
```

and:

```
-------- module a ---------
template TFoo(T) { void bar() { func(1); }  }
void func(double d) { }

-------- module b ---------
import a;

void func(int i) { }
alias TFoo!(int) f;
...
f.bar();        // will call a.func(double)
```

2.16.3. **Argument Deduction**

The types of template parameters are deduced for a particular template instantiation by comparing the template argument with the corresponding template parameter.

For each template parameter, the following rules are applied in order until a type is deduced for each parameter:

1. If there is no type specialization for the parameter, the type of the parameter is set to the template argument.
2. If the type specialization is dependent on a type parameter, the type of that parameter is set to be the corresponding part of the type argument.
3. If after all the type arguments are examined there are any type parameters left with no type assigned, they are assigned types corresponding to the template argument in the same position in the *TemplateArgumentList*.

4. If applying the above rules does not result in exactly one type for each template parameter, then it is an error.

For example:

```
template TFoo(T) { }
alias TFoo!(int) Foo1;          // (1) T is deduced to be int
alias TFoo!(char*) Foo2;        // (1) T is deduced to be char*

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

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

template TBar(D : E*, E) { }
alias TBar!(int*, int) Bar3;    // (1) E is int
                                // (3) D is int*
```

When considering matches, a class is considered to be a match for any super classes or interfaces:

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

template TFoo(T : A) { }
alias TFoo!(B) Foo4;            // (3) T is B

template TBar(T : U*, U : A) { }
alias TBar!(B*, B) Foo5;        // (2) T is B*
                                // (3) U is B
```

2.16.4. **Value Parameters**

This example of template foo has a value parameter that is specialized for 10:

```
template foo(U : int, int T : 10)
{
    U x = T;
}

void main()
{
    assert(foo!(int, 10).x == 10);
}
```

2.16.5. Specialization

Templates may be specialized for particular types of arguments by following the template parameter identifier with a : and the specialized type. For example:

```
template TFoo(T)        {... }  // #1
template TFoo(T : T[])  {... }  // #2
template TFoo(T : char) {... }  // #3
template TFoo(T,U,V)    {... }  // #4

alias TFoo!(int) foo1;         // instantiates #1
alias TFoo!(double[]) foo2;    // instantiates #2 with T being double
alias TFoo!(char) foo3;        // instantiates #3
alias TFoo!(char, int) fooe;   // error, number of arguments mismatch
alias TFoo!(char, int, int) foo4; // instantiates #4
```

The template picked to instantiate is the one that is most specialized that fits the types of the *TemplateArgumentList*. Determine which is more specialized is done the same way as the C++ partial ordering rules. If the result is ambiguous, it is an error.

2.16.6. Alias Parameters

Alias parameters enable templates to be parameterized with any type of D symbol, including global names, type names, module names, template names, and template instance names. Local names may not be used as alias parameters. It is a superset of the uses of template template parameters in C++.

- Global names

```
int x;

template Foo(alias X)
{
    static int* p =  &X;
}

void test()
{
    alias Foo!(x) bar;
    *bar.p = 3;         // set x to 3
    int y;
    alias Foo!(y) abc; // error, y is local name
}
```

- Type names

```
class Foo
{
    static int p;
}

template Bar(alias T)
{
    alias T.p q;
}

void test()
{
    alias Bar!(Foo) bar;
    bar.q = 3;  // sets Foo.p to 3
}
```

- Module names

```
import std.string;

template Foo(alias X)
{
        alias X.toString y;
}

void test()
{
    alias Foo!(std.string) bar;
    bar.y(3);   // calls std.string.toString(3)
}
```

- Template names

```
int x;

template Foo(alias X)
{
    static int* p =  &X;
}

template Bar(alias T)
{
    alias T!(x) abc;
}

void test()
{
    alias Bar!(Foo) bar;
    *bar.abc.p = 3;     // sets x to 3
}
```

- Template alias names

```
int x;

template Foo(alias X)
{
    static int* p =  &X;
}

template Bar(alias T)
{
    alias T.p q;
}

void test()
{
    alias Foo!(x) foo;
    alias Bar!(foo) bar;
    *bar.q = 3;         // sets x to 3
}
```

2.16.7. 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.

2.17. Design by Contract

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.

2.17.1. Assert Contract

The most basic contract is the assert. An assert inserts acheckable 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.

2.17.2. 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 <dfn>contracts</dfn>. 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.

2.17.3. In, Out and Inheritance

If a function in a derived class overrides a function in its super class, then only one of the `in` contracts of the base functions must be satisified Overriding functions then becomes a process of *loosening* the `in` contracts.

Conversely, all of the `out` contracts needs to be satisified, so overriding functions becomes a processes of *tightening* the `out` contracts.

2.17.4. 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.

2.17.5. References

ContractsReading List

AddingContracts to Java

2.18. Debug, Version, and Static Assert

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

```
DebugSpecification
DebugAttribute
DebugStatement

VersionSpecification
VersionAttribute
VersionStatement

StaticAssert
```

2.18.1. 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
- **AMD64** AMD 64 bit processors
- **Windows** Microsoft Windows systems
- **Win32** Microsoft 32 bit Windows systems
- **Win64** Microsoft 64 bit Windows systems
- **linux** All linux systems
- **LittleEndian** Byte order, least significant first
- **BigEndian** Byte order, most significant first
- **D_InlineAsm** Inline assembler is implemented
- **none** Never defined; used to just disable a section of code

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.

2.18.2. **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...
}
```

2.18.3. **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(identifer) 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.

2.18.4. **Version Statement**

It is commonplace to conveniently support multiple versions of a module with a single source file. While the D way is to isolate all versioning into separate modules, that can get burdensome if it's just simple line change, or if the entire program would otherwise fit into one module.

```
VersionStatement:
        VersionPredicate Statement
        VersionPredicate Statement else  Statement

VersionPredicate
        version (  Integer )
        version (  Identifier )
```

The version statement conditionally compiles in its statement body based on the version specified by the *Integer* of *Identifier*. Both forms are set by the **-version** switch to the compiler. If *Statement* is a block statement, it does not introduce a new scope. For example:

```
int k;
version (Demo)  // compile in this code block for the demo version
{   int i;
    int k;      // error, k already defined

    i = 3;
}
x = i;          // uses the i declared above
```

The version statement works together with the version attribute for declarations.

Version statements can nest.

The optional else clause gets conditionally compiled in if the version predicate is false:

```
version (X86)
{
   ... // implement custom inline assembler version
}
else
{
   ... // use default, but slow, version
}
```

While the debug and version statements superficially behave the same, they are intended for very different purposes. Debug statements are for adding debug code that is removed for the release version. Version statements are to aid in portability and multiple release versions.

2.18.5. 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*.

2.18.6. **Version Attribute**

```
VersionAttribute:
    version (  Integer )
    version (  Identifier )
```

The version attribute is very similar to the debug attribute, and in many ways is functionally interchangable 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*.

2.18.7. Static Assert

```
StaticAssert:
    static assert (  Expression );
```

Expression is evaluated at compile time, and converted to a boolean value. If the value is true, the static assert is ignored. If the value is false, an error diagnostic is issued and the compile fails.

Unlike *AssertExpression*s, *StaticAssert*s are always checked and evaluted by the compiler unless they appear in a false debug or version conditional.

```
void foo()
{
    if (0)
    {
        assert(0);              // never trips
        static assert(0);       // always trips
    }
    version (BAR)
    {
        static assert(0);       // does not trip unless BAR is defined
    }
}
```

2.19. 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.

2.19.1. 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.

2.19.2. **The D Error Handling Solution**

Let's first make some observations and assumptions about errors:

- Errors are not part of the normal flow of a program. Errors are exceptional, unusual, and unexpected.
- Because errors are unusual, execution of error handling code is not performance critical.
- The normal flow of program logic is performance critical.
- All errors must be dealt with in some way, either by code explicitly written to handle them, or by some system default handling.
- The code that detects an error knows more about the error than the code that must recover from the error.

The solution is to use exception handling to report errors. All errors are objects derived from abstract class Error. class Error has a pure virtual function called toString() which produces a char[] with a human readable description of the error.

If code detects an error like "out of memory," then an Error is thrown with a message saying "Out of memory". The function call stack is unwound, looking for a handler for the Error. Finally blocks are executed as the stack is unwound. If an error handler is found, execution resumes there. If not, the default Error handler is run, which displays the message and terminates the program.

How does this meet our criteria?

- Standardized - consistent usage makes it more useful. This is the D way, and is used consistently in the D runtime library and examples.
- Produces a reasonable result even if the programmer fails to check for errors. If no catch handlers are there for the errors, then the program gracefully exits through the default error handler with an appropriate message.
- Allows old code to be reused with new code without having to modify the old code to be compatible with new error types. Old code can decide to catch all errors, or only specific ones, propagating the rest upwards. In any case, there is no more need to correlate error numbers with messages, the correct message is always supplied.
- No errors get inadvertently ignored. Error exceptions get handled one way or another. There is nothing like a NULL pointer return indicating an error, followed by trying to use that NULL pointer.

- Allows 'quick and dirty' utilities to be written that still correctly handle errors. Quick and dirty code need not write any error handling code at all, and don't need to check for errors. The errors will be caught, an appropriate message displayed, and the program gracefully shut down all by default.
- Easy to make the error handling source code look good. The try/catch/finally statements look a lot nicer than endless if (error) goto errorhandler; statements.

How does this meet our assumptions about errors?

- Errors are not part of the normal flow of a program. Errors are exceptional, unusual, and unexpected. D exception handling fits right in with that.
- Because errors are unusual, execution of error handling code is not performance critical. Exception handling stack unwinding is a relatively slow process.
- The normal flow of program logic is performance critical. Since the normal flow code does not have to check every function call for error returns, it can be realistically faster to use exception handling for the errors.
- All errors must be dealt with in some way, either by code explicitly written to handle them, or by some system default handling. If there's no handler for a particular error, it is handled by the runtime library default handler. If an error is ignored, it is because the programmer specifically added code to ignore an error, which presumably means it was intentional.
- The code that detects an error knows more about the error than the code that must recover from the error. There is no more need to translate error codes into human readable strings, the correct string is generated by the error detection code, not the error recovery code. This also leads to consistent error messages for the same error between applications.

2.20. 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 collecters 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 arbitrarilly 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.

2.20.1. How Garbage Collection Works

To be written...

2.20.2. Interfacing Garbage Collected Objects With Foreign Code

The garbage collector looks for roots in its static data segment, and the stacks and register contents of each thread. If the only root of an object is held outside of this, then the collecter will miss it and free the memory.

To avoid this from happening,

- Maintain a root to the object in an area the collector does scan for roots.
- Reallocate the object using the foreign code's storage allocator or using the C runtime library's malloc/free.

2.20.3. 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.

2.20.4. 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.

2.21. 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

2.21.1. **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 unnecessarilly consume time and memory for strings that are already all upper case.

The solution is to implement copy-on-write, which means that a copy is made only if the string needs to be modified. Some string processing languages do do this as the default behavior, but there is a huge cost to it. The string "abcdeF" will wind up being copied 5 times by the function. To get the maximum efficiency using the protocol, it'll have to be done explicitly in the code. Here's toupper() rewritten to implement copy-on-write in an efficient manner:

```
char[] toupper(char[] s)
{
    int changed;
    int i;

    changed = 0;
    for (i = 0; i < s.length; i++)
    {
        char c = s[i];
        if ('a' <= c  & & c <= 'z')
        {
            if (!changed)
            {   char[] r = new char[s.length];
                r[] = s;
                s = r;
                changed = 1;
            }
            s[i] = c - (cast(char)'a' - 'A');
        }
    }
    return s;
}
```

Copy-on-write is the protocol implemented by array processing functions in the D Phobos runtime library.

2.21.2. **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.

2.21.3. **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.

2.21.4. **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.

2.21.5. 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.

2.21.6. Explicit Class Instance Allocation

D provides a means of creating custom allocators and deallocators for class instances. Normally, these would be allocated on the garbage collected heap, and deallocated when the collector decides to run. For specialized purposes, this can be handled by creating *NewDeclaration*s and *DeleteDeclaration*s. For example, to allocate using the C runtime library's `malloc` and `free`:

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

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

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

    delete(void* p)
    {
        if (p)
        {   gc.removeRange(p);
            std.c.stdlib.free(p);
        }
    }
}
```

The critical features of new() are:

- new() does not have a return type specified, but it is defined to be void*. new() must return a void*.
- If new() cannot allocate memory, it must not return null, but must throw an exception.
- The pointer returned from new() must be to memory aligned to the default alignment. This is 8 on win32 systems.
- The *size* parameter is needed in case the allocator is called from a class derived from Foo and is a larger size than Foo.
- A null is not returned if storage cannot be allocated. Instead, an exception is thrown. Which exception gets thrown is up to the programmer, in this case, OutOfMemory() is.
- When scanning memory for root pointers into the garbage collected heap, the static data segment and the stack are scanned automatically. The C heap is not. Therefore, if Foo or any class derived from Foo using the allocator contains any references to data allocated by the garbage collector, the gc needs to be notified. This is done with the gc.addRange() method.
- No initialization of the memory is necessary, as code is automatically inserted after the call to new() to set the class instance members to their defaults and then the constructor (if any) is run.

The critical features of delete() are:

- The destructor (if any) has already been called on the argument p, so the data it points to should be assumed to be garbage.
- The pointer p may be null.
- If the gc was notified with gc.addRange(), a corresponding call to gc.removeRange() must happen in the deallocator.
- If there is a delete(), there should be a corresponding new().

If memory is allocated using class specific allocators and deallocators, careful coding practices must be followed to avoid memory leaks and dangling references. In the presence of exceptions, it is particularly important to practice RAII to prevent memory leaks.

2.21.7. **Mark/Release**

Mark/Release is equivalent to a stack method of allocating and freeing memory. A 'stack' is created in memory. Objects are allocated by simply moving a pointer down the stack. Various points are 'marked', and then whole sections of memory are released simply by resetting the stack pointer back to a marked point.

```
import std.c.stdlib;
import std.outofmemory;

class Foo
{
    static void[] buffer;
    static int bufindex;
    static const int bufsize = 100;

    static this()
    {   void *p;

        p = malloc(bufsize);
        if (!p)
            throw new OutOfMemory;
        gc.addRange(p, p + bufsize);
        buffer = p[0.. bufsize];
    }

    static ~this()
    {
        if (buffer.length)
        {
            gc.removeRange(buffer);
            free(buffer);
            buffer = null;
```

```
            }
        }

        new(uint sz)
        {   void *p;

            p =  &buffer[bufindex];
            bufindex += sz;
            if (bufindex > buffer.length)
                throw new OutOfMemory;
            return p;
        }

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

        static int mark()
        {
            return bufindex;
        }

        static void release(int i)
        {
            bufindex = i;
        }
    }

    void test()
    {
        int m = Foo.mark();
        Foo f1 = new Foo;           // allocate
        Foo f2 = new Foo;           // allocate
       ...
        Foo.release(m);             // deallocate f1 and f2
    }
```

- The allocation of buffer[] itself is added as a region to the gc, so there is no need for a separate call inside Foo.new() to do it.

2.21.8. 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.

2.21.9. 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 std.c.stdlib;

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

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

void test()
{
    Foo f = new(std.c.stdlib.alloca(Foo.classinfo.init.length)) Foo;
   ...
}
```

- There is no need to check for a failure of alloca() and throw an exception, since by definition alloca() will generate a stack overflow exception if it overflows.
- There is no need for a call to gc.addRange() or gc.removeRange() since the gc automatically scans the stack anyway.
- The dummy delete() function is to ensure that no attempts are made to delete a stack based object.

2.22. Floating Point

Floating Point Intermediate Values

On many computers, greater precision operations do not take any longer than lesser precision operations, so it makes numerical sense to use the greatest precision available for internal temporaries. The philosophy is not to dumb down the language to the lowest common hardware denominator, but to enable the exploitation of the best capabilities of target hardware.

For floating point operations and expression intermediate values, a greater precision can be used than the type of the expression. Only the minimum precision is set by the types of the operands, not the maximum. **Implementation**

Note: On Intel x86 machines, for example, it is expected (but not required) that the intermediate calculations be done to the full 80 bits of precision implemented by the hardware.

It's possible that, due to greater use of temporaries and common subexpressions, optimized code may produce a more accurate answer than unoptimized code.

Algorithms should be written to work based on the minimum precision of the calculation. They should not degrade or fail if the actual precision is greater. Float or double types, as opposed to the extended type, should only be used for:

- reducing memory consumption for large arrays
- data and function argument compatibility with C

Complex and Imaginary types

In existing languages, there is an astonishing amount of effort expended in trying to jam a complex type onto existing type definition facilities: templates, structs, operator overloading, etc., and it all usually ultimately fails. It fails because the semantics of complex operations can be subtle, and it fails because the compiler doesn't know what the programmer is trying to do, and so cannot optimize the semantic implementation.

This is all done to avoid adding a new type. Adding a new type means that the compiler can make all the semantics of complex work "right". The programmer then can rely on a correct (or at least fixable) implementation of complex.

Coming with the baggage of a complex type is the need for an imaginary type. An imaginary type eliminates some subtle semantic issues, and improves performance by not having to perform extra operations on the implied 0 real part.

Imaginary literals have an i suffix:

```
ireal j = 1.3i;
```

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

```
cdouble cd = 3.6 + 4i;
creal c = 4.5 + 2i;
```

Complex numbers have two properties:

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

For example:

```
cd.re           is 4.5 double
cd.im           is 2i idouble
c.re            is 4.5 real
c.im            is 2i ireal
```

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]
```

2.23. 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
```

2.23.1. 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
}
```

2.23.2. 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.

2.23.3. even

Causes the assembler to emit NOP instructions to align the next assembler instruction on an even boundary.

2.23.4. **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.

2.23.5. **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
}
```

2.23.6. **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
```

2.23.7. **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.

2.23.8. Opcodes Supported

aaa	aad	aam	aas	adc
add	addpd	addps	addsd	addss
and	andnpd	andnps	andpd	andps
arpl	bound	bsf	bsr	bswap
bt	btc	btr	bts	call
cbw	cdq	clc	cld	clflush

cli	clts	cmc	cmova	cmovae
cmovb	cmovbe	cmovc	cmove	cmovg
cmovge	cmovl	cmovle	cmovna	cmovnae
cmovnb	cmovnbe	cmovnc	cmovne	cmovng
cmovnge	cmovnl	cmovnle	cmovno	cmovnp
cmovns	cmovnz	cmovo	cmovp	cmovpe
cmovpo	cmovs	cmovz	cmp	cmppd
cmpps	cmps	cmpsb	cmpsd	cmpss
cmpsw	cmpxch8b	cmpxchg	comisd	comiss
cpuid	cvtdq2pd	cvtdq2ps	cvtpd2dq	cvtpd2pi
cvtpd2ps	cvtpi2pd	cvtpi2ps	cvtps2dq	cvtps2pd
cvtps2pi	cvtsd2si	cvtsd2ss	cvtsi2sd	cvtsi2ss
cvtss2sd	cvtss2si	cvttpd2dq	cvttpd2pi	cvttps2dq
cvttps2pi	cvttsd2si	cvttss2si	cwd	cwde
da	daa	das	db	dd
de	dec	df	di	div
divpd	divps	divsd	divss	dl
dq	ds	dt	dw	emms
enter	f2xm1	fabs	fadd	faddp
fbld	fbstp	fchs	fclex	fcmovb
fcmovbe	fcmove	fcmovnb	fcmovnbe	fcmovne
fcmovnu	fcmovu	fcom	fcomi	fcomip
fcomp	fcompp	fcos	fdecstp	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
pshuflw	pshufw	pslld	pslldq	psllq
psllw	psrad	psraw	psrld	psrldq
psrlq	psrlw	psubb	psubd	psubq
psubsb	psubsw	psubusb	psubusw	psubw
punpckhbw	punpckhdq	punpckhqdq	punpckhwd	punpcklbw
punpckldq	punpcklqdq	punpcklwd	push	pusha
pushad	pushf	pushfd	pxor	rcl
rcpps	rcpss	rcr	rdmsr	rdpmc
rdtsc	rep	repe	repne	repnz
repz	ret	retf	rol	ror
rsm	rsqrtps	rsqrtss	sahf	sal
sar	sbb	scas	scasb	scasd
scasw	seta	setae	setb	setbe
setc	sete	setg	setge	setl
setle	setna	setnae	setnb	setnbe

setnc	setne	setng	setnge	setnl
setnle	setno	setnp	setns	setnz
seto	setp	setpe	setpo	sets
setz	sfence	sgdt	shl	shld
shr	shrd	shufpd	shufps	sidt
sldt	smsw	sqrtpd	sqrtps	sqrtsd
sqrtss	stc	std	sti	stmxcsr
stos	stosb	stosd	stosw	str
sub	subpd	subps	subsd	subss
sysenter	sysexit	test	ucomisd	ucomiss
ud2	unpckhpd	unpckhps	unpcklpd	unpcklps
verr	verw	wait	wbinvd	wrmsr
xadd	xchg	xlat	xlatb	xor
xorpd	xorps			

AMD Opcodes Supported

pavgusb	pf2id	pfacc	pfadd	pfcmpeq
pfcmpge	pfcmpgt	pfmax	pfmin	pfmul
pfnacc	pfpnacc	pfrcp	pfrcpit1	pfrcpit2
pfrsqit1	pfrsqrt	pfsub	pfsubr	pi2fd
pmulhrw	pswapd			

CHAPTER 3

Appendices

3.1. 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.

3.1.1. Calling C Functions

C functions can be called directly from D. There is no need for wrapper functions, argument swizzling, and the C functions do not need to be put into a separate DLL.

The C function must be declared and given a calling convention, most likely the "C" calling convention, for example:

```
extern (C) int strcmp(char *string1, char *string2);
```

and then it can be called within D code in the obvious way:

```
import std.string;
int myDfunction(char[] s)
{
    return strcmp(std.string.toCharz(s), "foo 0");
}
```

There are several things going on here:

- D understands how C function names are "mangled" and the correct C function call/return sequence.
- C functions cannot be overloaded with another C function with the same name.
- There are no __cdecl, __far, __stdcall, __declspec, or other such C type modifiers in D. These are handled by attributes, such as `extern (C)`.
- There are no const or volatile type modifiers in D. To declare a C function that uses those type modifiers, just drop those keywords from the declaration.
- Strings are not 0 terminated in D. See "Data Type Compatibility" for more information about this.

C code can correspondingly call D functions, if the D functions use an attribute that is compatible with the C compiler, most likely the extern (C):

```
// myfunc() can be called from any C function
extern (C)
{
    void myfunc(int a, int b)
    {
       ...
    }
}
```

3.1.2. 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.

3.1.3. **Data Type Compatibility**

D type	C type
void	void
bit	no equivalent
byte	signed char
ubyte	unsigned char
char	char (chars are unsigned in D)
wchar	wchar_t
short	short
ushort	unsigned short
int	int
uint	unsigned
long	long long
ulong	unsigned long long
float	float
double	double
extended	long double
imaginary	long double _Imaginary
complex	long double _Complex
type*	type *
type[dim]	type[dim]
type[]	no equivalent
type[type]	no equivalent
"string0"	"string" or L"string"
class	no equivalent
type(*)(parameters)	type(*)(parameters)

These equivalents hold for most 32 bit C compilers. The C standard does not pin down the sizes of the types, so some care is needed.

3.1.4. **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.

3.1.5. 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.

3.2. 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.

3.3. Portability Guide

It's good software engineering practice to minimize gratuitous portability problems in the code. Techniques to minimize potential portability problems are:

- The integral and floating type sizes should be considered as minimums. Algorithms should be designed to continue to work properly if the type size increases.
- Floating point computations can be carried out at a higher precision than the size of the floating point variable can hold. Floating point algorithms should continue to work properly if precision is arbitrarilly 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);
```

3.3.1. 32 to 64 Bit Portability

64 bit processors and operating systems are coming. With that in mind:

- Integral types will remain the same sizes between 32 and 64 bit code.

- Pointers and object references will increase in size from 4 bytes to 8 bytes going from 32 to 64 bit code.

- Use **size_t** as an alias for an unsigned integral type that can span the address space.

- Use **ptrdiff_t** as an alias for a signed integral type that can span the address space.

- The **.length**, **.size**, **.sizeof**, and **.alignof** properties will be of type **size_t**.

3.3.2. 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.

3.4. 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 onepossibility. 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 `and` 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.

3.5. MISSING: model.html

3.6. MISSING: phobos.html

3.7. D for Win32

This describes the D implementation for 32 bit Windows systems. Naturally, Windows specific D features are not portable to other platforms.

Instead of the:

```
        #include <windows.h>
```

of C, in D there is:

```
import std.c.windows.windows;
```

3.7.1. 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.

3.7.2. 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:

```
import std.c.windows.windows;

extern (C) void gc_init(); extern (C) void gc_term(); extern (C)
void _minit(); extern (C) void _moduleCtor(); extern (C) void
_moduleUnitTests();

extern (Windows) int WinMain (HINSTANCE hInstance,
        HINSTANCE hPrevInstance,
        LPSTR lpCmdLine,
        int nCmdShow)
{
    int result;
```

```
    gc_init();                  // initialize garbage collector
    _minit();                   // initialize module constructor table

    try
    {
        _moduleCtor();          // call module constructors
        _moduleUnitTests();     // run unit tests (optional)

        result = doit();        // insert user code here
    }

    catch (Object o)            // catch any uncaught exceptions
    {
        MessageBoxA(null, (char *)o.toString(), "Error",
                    MB_OK | MB_ICONEXCLAMATION);
        result = 0;             // failed
    }

    gc_term();                  // run finalizers; terminate garbage collector
    return result;
}
```

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

3. A.def (Module DefinitionFile) with at least the following two lines in it:

```
EXETYPE NT SUBSYSTEM WINDOWS
```

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

4. The presence of `WinMain()` is recognized by the compiler causing it to emit a reference to __acrtused_dllland the phobos.lib runtime library.

3.7.3. DLLs (Dynamic Link Libraries)

DLLs can be created in D in roughly the same way as in C. A `DllMain()` is required, looking like:

```
import std.c.windows.windows;
HINSTANCE g_hInst;

extern (C)
{
        void gc_init();
        void gc_term();
        void _minit();
        void _moduleCtor();
```

```
        void _moduleUnitTests();
}

extern (Windows)
BOOL DllMain (HINSTANCE hInstance, ULONG ulReason, LPVOID pvReserved)
{
    switch (ulReason)
    {
        case DLL_PROCESS_ATTACH:
            gc_init();                  // initialize GC
            _minit();                   // initialize module list
            _moduleCtor();              // run module constructors
            _moduleUnitTests();         // run module unit tests
            break;

        case DLL_PROCESS_DETACH:
            gc_term();                  // shut down GC
            break;

        case DLL_THREAD_ATTACH:
        case DLL_THREAD_DETACH:
            // Multiple threads not supported yet
            return false;
    }
    g_hInst=hInstance;
    return true;
}
```

Notes:

- The _moduleUnitTests() call is optional.
- It's a little crude, I hope to improve it.
- The presence of `DllMain()` is recognized by the compiler causing it to emit a reference to __acrtused_dlland the phobos.lib runtime library.

Link with a.def (Module DefinitionFile) 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.

3.7.4. COM Programming

Many Windows API interfaces are in terms of COM (Common Object Model) objects (also called OLE or ActiveX objects). A COM object is an object who's first field is a pointer to a vtbl[], and the first 3 entries in that vtbl[] are for QueryInterface(), AddRef(), and Release().

COM objects are analogous to D interfaces. Any COM object can be expressed as a D interface, and every D object with an interface X can be exposed as a COM object X. This means that D is compatible with COM objects implemented in other languages.

While not strictly necessary, the Phobos library provides an Object useful as a super class for all D COM objects, called ComObject. ComObject provides a default implementation for QueryInterface(), AddRef(), and Release().

Windows COM objects use the Windows calling convention, which is not the default for D, so COM functions need to have the attribute extern (Windows). So, to write a COM object:

```
import std.c.windows.com;

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

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

3.8. Converting C .h Files to D Modules

While D cannot directly compile C source code, it can easilly interface to C code, be linked with C object files, and call C functions in DLLs. The interface to C code is normally found in C .h files. So, the trick to connecting with C code is in converting C .h files to D modules. This turns out to be difficult to do mechanically since inevitably some human judgement must be applied. This is a guide to doing such conversions.

Preprocessor .h files can sometimes be a bewildering morass of layers of macros, #include files, #ifdef 's, etc. D doesn't support a text preprocessor, so the first step is to remove the need for it by taking the preprocessed output. For DMC (the Digital Mars C/C++ compiler), the command:

```
dmc -c program.h -e -l
```

will create a file program.lst which is the source file after all text preprocessing.

Remove all the #if, #ifdef, #include , etc. statements.

Linkage Generally, surround the entire module with:

```
extern (C)
{
    ...file contents...
}
```

to give it C linkage.

Types A little global search and replace will take care of renaming the C types to D types. The following table shows a typical mapping for 32 bit C code:

C type	D type
long double	real
unsigned long long	ulong
long long	long
unsigned long	uint
long	int
unsigned	uint
unsigned short	ushort
signed char	byte
unsigned char	ubyte
wchar_t	wchar or dchar
bool	int

NULL Or `((void*)0)` should be replaced with `null`.

Numeric Literals Any 'L' or 'l' numeric literal suffixes should be removed, as a C `long` is (usually) the same size as a D `int`. Similarly, 'LL' suffixes should be replaced with a single 'L'. Any 'u' suffix will work the same in D.

String Literals In most cases, any 'L' prefix to a string can just be dropped, as D will implicitly convert strings to wide characters if necessary. However, one can also replace:

```
L"string"
```

with:

```
cast(wchar[])"string"
```

Macros Lists of macros like:

```
#define FOO     1
#define BAR     2
#define ABC     3
#define DEF     40
```

can be replaced with:

```
enum
{   FOO = 1,
```

```
    BAR = 2,
    ABC = 3,
    DEF = 40
}
```

or with:

```
const int FOO = 1;
const int BAR = 2;
const int ABC = 3;
const int DEF = 40;
```

Function style macros, such as:

```
#define MAX(a,b) ((a) < (b) ? (b) : (a))
```

can be replaced with functions:

```
int MAX(int a, int b) { return (a < b) ? b : a); }
```

Declaration Lists D doesn't allow declaration lists to change the type. Hence:

```
int *p, q, t[3], *s;
```

should be written as:

```
int* p, s;
int q;
int[3] t;
```

Void Parameter Lists Functions that take no parameters:

```
int foo(void);
```

are in D:

```
int foo();
```

Const Type Modifiers D has `const` as a storage class, not a type modifier. Hence, just drop any `const` used as a type modifier:

```
void foo(const int *p, char *const q);
```

becomes:

```
void foo(int* p, char* q);
```

Typedef `alias` is the D equivalent to the C `typedef` :

```
typedef int foo;
```

becomes:

```
alias int foo;
```

Structs Replace declarations like:

```
typedef struct Foo
{   int a;
    int b;
}  Foo, *pFoo, *lpFoo;
```

with:

```
struct Foo
{   int a;
    int b;
}
alias Foo* pFoo, lpFoo;
```

Struct Member Alignment A good D implementation by default will align struct members the same way as the C compiler it was designed to work with. But if the `.h` file has some `#pragma` 's to control alignment, they can be duplicated with the D `align` attribute:

```
#pragma pack(1)
struct Foo
{
    int a;
    int b;
} ;
#pragma pack()
```

becomes:

```
struct Foo
{
  align (1):
    int a;
    int b;
}
```

Nested Structs

```
struct Foo
{
    int a;
    struct Bar
    {
        int c;
    }  bar;
} ;

struct Abc
{
    int a;
    struct
    {
        int c;
    }  bar;
} ;
```

becomes:

```
struct Foo
{
    int a;
    struct Bar
    {
        int c;
    }
    Bar bar;
} ;

struct Abc
{
    int a;
    struct
    {
        int c;
    }
} ;
```

`__cdecl, __pascal, __stdcall`

```
int __cdecl x;
int __cdecl foo(int a);
int __pascal bar(int b);
int __stdcall abc(int c);
```

become:

```
extern (C) int x;
extern (C) int foo(int a);
extern (Pascal) int bar(int b);
extern (Windows) int abc(int c);
```

`__declspec(dllimport)`

```
__declspec(dllimport) int __stdcall foo(int a);
```

becomes:

```
export extern (Windows) int foo(int a);
```

`__fastcall` Unfortunately, D doesn't support the `__fastcall` convention. Therefore, a shim will be needed, either written in C:

```
int __fastcall foo(int a);

int myfoo(int a)
{
    return foo(int a);
}
```

and compiled with a C compiler that supports `__fastcall` and linked in, or compile the above, disassemble it with obj2asm andinsert it in a D `myfoo` shim with inline assembler.

3.9. The D Style

The D Style is a set of style conventions for writing D programs. The D Style is not enforced by the compiler, it is purely cosmetic and a matter of choice. Adhering to the D Style, however, will make it easier for others to work with your D code and easier for you to work with others' D code. The D Style can form the starting point for a D project style guide customized for your project team.

White Space

- One statement per line.
- Two or more spaces per indentation level.
- Operators are separated by single spaces from their operands.
- Two blank lines separating function bodies.
- One blank line separating variable declarations from statements in function bodies.

Comments

- Use // comments to document a single line:

```
        statement;  // comment
        statement;  // comment
```

- Use block comments to document a multiple line block of statements:

```
        /*
         * comment
         * comment
         */
         statement;
         statement;
```

- Use nesting comments to 'comment out' a piece of trial code:

```
    /+++++
        /*
         * comment
         * comment
         */
         statement;
         statement;
     +++++/
```

Naming Conventions

- General Names formed by joining multiple works should have each word other than the first capitalized.

```
    int myFunc();
```

- Module Module names are all lower case. This avoids problems dealing with case insensitive file systems.
- C Modules Modules that are interfaces to C functions go into the "c" package, for example:

```
import std.c.stdio;
```

  Module names should be all lower case.

- Class, Struct, Union, Enum names are capitalized.

```
class Foo;
class FooAndBar;
```

- Function names Function names are not capitalized.

```
int done();
int doneProcessing();
```

- Const names Are in all caps.
- Enum member names Are in all caps.

Meaningless Type Aliases

Things like:

```
alias void VOID;
alias int INT;
alias int* pint;
```

should be avoided.

Declaration Style

Since in D the declarations are left-associative, left justify them:

```
int[] x, y;     // makes it clear that x and y are the same type
int** p, q;     // makes it clear that p and q are the same type
```

to emphasize their relationship. Do not use the C style:

```
int []x, y;     // confusing since y is also an int[]
int **p, q;     // confusing since q is also an int**
```

Operator Overloading

Operator overloading is a powerful tool to extend the basic types supported by the language. But being powerful, it has great potential for creating obfuscated code. In particular, the existing D operators have conventional meanings, such as '+' means 'add' and '«' means 'shift left'. Overloading operator '+' with a meaning different from 'add' is arbitrarilly confusing and should be avoided.

Hungarian Notation

Just say no.

3.10. Example: wc

This program is the D version of the classic wc (wordcount) C program. It serves to demonstrate how to read files, slice arrays, and simple symbol table management with associative arrays.

```
import std.file;

int main (char[][] args) {
    int w_total;
    int l_total;
    int c_total;

    printf ("   lines   words   bytes file n");
    foreach (char[] arg; args[1.. args.length])
    {
        char[] input;
        int w_cnt, l_cnt, c_cnt;
        int inword;

        input = cast(char[])std.file.read(arg);

        foreach (char c; input)
        {
            if (c == ' n')
                ++l_cnt;
            if (c != ' ')
            {
                if (!inword)
                {
                    inword = 1;
                    ++w_cnt;
                }
            }
            else
                inword = 0;
```

```
            ++c_cnt;
        }
        printf ("%8lu%8lu%8lu %.*s n", l_cnt, w_cnt, c_cnt, arg);
        l_total += l_cnt;
        w_total += w_cnt;
        c_total += c_cnt;
    }
    if (args.length > 2)
    {
        printf ("-------------------------------------- n%8lu%8lu%8lu total",
            l_total, w_total, c_total);
    }
    return 0;
}
```

3.11. Compiler for D Programming Language

- D for Win32
- D for x86 Linux
- general

3.11.1. Files Common to Win32 and Linux

- `dmd`
  `src`
  `phobos`
  D runtime library source
- `dmd`
  `src`
  `dmd`
  D compiler front end source under dual (GPL and Artistic) license
- `dmd`
  `html`
  `d`
  Documentation
- `dmd`
  `samples`

`d`
Sample D programs

3.12. Win32 D Compiler

3.12.1. 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

3.12.2. Requirements

- 32 bit Windows operating system
- D compiler for Win32
- linker and utilities for Win32

3.12.3. 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].

3.12.4. Example

Run:

```
dmd bin shell all.sh
```

in the dmd samples d directory for several small examples.

3.12.5. Compiler Arguments and Switches

- **dmd** *files... -switch...*

- *files...*

Extension	File Type
none	D source files
.d	D source files
.obj	Object files to link in
.exe	Name output executable file
.def	module definition file
.res	resource file

- **-c** compile only, do not link
- **-d** allow deprecated features
- **-debug** compile in debug code
- **-debug=** *level* compile in debug code <= *level*
- **-debug=** *ident* compile in debug code identified by *ident*
- **-g** add symbolic debug info
- **-gt** add trace profiling hooks
- **-inline** inline expand functions
- **-I** *path* where to look for imports. *path* is a ; separated list of paths. Multiple **-I** 's can be used, and the paths are searched in the same order.
- **-L** *linkerflag* pass *linkerflag* to the linker, for example, `/ma/li`
- **-O** optimize
- **-od** *objdir* write object files relative to directory *objdir* instead of to the current directory
- **-of** *filename* set output file name to *filename* in the output directory
- **-op** normally the path for **.d** source files is stripped off when generating an object file name. **-op** will leave it on.
- **-release** compile release version
- **-unittest** compile in unittest code
- **-v** verbose

- **-version=** *level* compile in version code >= *level*
- **-version=** *ident* compile in version code identified by *ident*

3.12.6. Linking

Linking is done directly by the **dmd** compiler after a successful compile. To prevent **dmd** from running the linker, use the **-c** switch.

The programs must be linked with the D runtime library **phobos.lib**, followed by the C runtime library **snn.lib**. This is done automatically as long as the directories for the libraries are on the LIB environment variable path. A typical way to set LIB would be:

```
set LIB= dmd lib; dm lib
```

3.12.7. 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.

3.12.8. 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"
```

3.13. Linux D Compiler

3.13.1. Files

- `/dmd/bin/dmd` D compiler executable
- `/dmd/bin/dumpobj` Elf file dumper
- `/dmd/bin/obj2asm` Elf file disassembler
- `/dmd/bin/dmd.conf` Global compiler settings (copy to `/etc/dmd.conf`)
- `/dmd/lib/libphobos.a` D runtime library (copy to `/usr/lib/libphobos.a`)

3.13.2. Requirements

- 32 bit x86 Linux operating system
- D compiler for Linux
- Gnu C compiler (gcc)

3.13.3. Installation

1. Unzip the archive into your home directory. It will create a `~/dmd` directory with all the files in it. All the tools are command line tools, which means they are run from a console window.
2. Edit the file `~/dmd/bin/dmd.conf` to put the path in to where the phobos source files are.
3. Copy `dmd.conf` to `/etc` :

```
cp dmd/bin/dmd.conf /etc
```

4. Give execute permission to the following files:

```
chmod u+x dmd/bin/dmd dmd/bin/obj2asm dmd/bin/dumpobj
```

5. Put `dmd/bin` on your **PATH**, or copy the linux executables to `/usr/local/bin`
6. Copy the library to `/usr/lib` :

```
cp dmd/lib/libphobos.a /usr/lib
```

3.13.4. Compiler Arguments and Switches

- **dmd** *files... -switch...*

- *files...*

Extension	File Type
none	D source files
.d	D source files
.o	Object files to link in
.a	Library files to link in

- **-c** compile only, do not link
- **-d** allow deprecated features
- **-debug** compile in debug code
- **-debug=** *level* compile in debug code <= *level*
- **-debug=** *ident* compile in debug code identified by *ident*
- **-g** add symbolic debug info

- **-gt** add trace profiling hooks (not supported under linux)
- **-inline** inline expand functions
- **-I** *path* where to look for imports. *path* is a ; separated list of paths. Multiple **-I** 's can be used, and the paths are searched in the same order.
- **-L** *linkerflag* pass *linkerflag* to the linker, for example, `-M`
- **-O** optimize
- **-od** *objdir* write object files relative to directory *objdir* instead of to the current directory
- **-of** *filename* set output file name to *filename* in the output directory
- **-op** normally the path for **.d** source files is stripped off when generating an object file name. **-op** will leave it on.
- **-release** compile release version
- **-unittest** compile in unittest code
- **-v** verbose
- **-version=** *level* compile in version code >= *level*
- **-version=** *ident* compile in version code identified by *ident*

3.13.5. Linking

Linking is done directly by the **dmd** compiler after a successful compile. To prevent **dmd** from running the linker, use the **-c** switch.

The actual linking is done by running **gcc**. This ensures compatibility with modules compiled with **gcc**.

3.13.6. 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**.

3.13.7. dmd.conf Initialization File

dmd will look for the initialization file **dmd.conf** in the directory `/etc`. If found, environment variable settings in the file will override any existing settings. This is handy to make **dmd** independent of programs with conflicting use of environment variables.

Environment variables follow the `[Environment]` section heading, in name=value pairs. Comments are lines that start with ;. For example:

```
; dmd.conf file for dmd
; Names enclosed by %% are searched for in the existing environemnt
; and inserted. The special name %@P% is replaced with the path
; to this file.
[Environment]
DFLAGS="-I%@P%: src phobos"
```

3.13.8. Differences from Win32 version

- String literals are read-only. Attempting to write to them will cause a segment violation.
- The configuration file is `/etc/dmd.conf`

3.13.9. Linux Bugs

- **-g** is not implemented, because I haven't figured out how to do it yet. **gdb** still works, though, at the global symbol level.
- The code generator output has not been tuned yet, so it can be bloated.
- Shared libraries cannot be generated.
- The exception handling is not compatible with the way **g++** does it. I don't know if this is an issue or not.

3.14. General

3.14.1. 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 gettor/settor not implemented.
- In preconditions and out postconditions for member functions are not inherited.
- It cannot be run from the IDDE.

3.14.2. **Feedback**

We welcome all feedback - kudos, flames, bugs, suggestions, hints, and most especially donated code! Join the fray in the D forum.

3.15. 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, Helmut Leitner, LubomirLitchev, Pavel Minayev, Paul Nash, Pat Nelson, Burton Radons, Tim Rentsch, Fabio Riccardi, Bob Taniguchi, John Whited, Matthew Wilson, Peter Zatloukal