D Specification

05/02/2007 snapshot

Contents

Lexical

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

Phases of Compilation

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

1. **source character set**
   The source file is checked to see what character set it is, and the appropriate scanner is loaded. ASCII and UTF formats are accepted.
2. **script line**
   If the first line starts with #! then the first line is ignored.
3. **lexical analysis**
   The source file is divided up into a sequence of tokens. Special Tokens are replaced with other tokens.Special token sequences Special token sequences are processed and removed.
4. **syntax analysis**
   The sequence of tokens is parsed to form syntax trees.
5. **semantic analysis**
   The syntax trees are traversed to declare variables, load symbol tables, assign types, and in general determine the meaning of the program.
6. **optimization**
   Optimization is an optional pass that tries to rewrite the program in a semantically equivalent, but faster executing, version.
7. **code generation**
   Instructions are selected from the target architecture to implement the semantics of the program. The typical result will be an object file, suitable for input to a linker.

Source Text

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

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

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

Format	BOM
UTF-8	EF BB BF

UTF-16BE	FE FF
UTF-16LE	FF FE
UTF-32BE	00 00 FE FF
UTF-32LE	FF FE 00 00
ASCII	no BOM

If the source file does not start with a BOM, then the first character must be less than or equal to U0000007F.

There are no digraphs or trigraphs in D.

The source text is decoded from its source representation into Unicode *Character*s. The *Character*s are further divided into: white space, end of lines, comments, special token sequences, tokens, all followed by end of file.

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

End of File

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

The source text is terminated by whichever comes first.

End of Line

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

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

White Space

```
WhiteSpace:
        Space
        Space WhiteSpace

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

Comments

```
Comment:
        /* Characters */
```

```
        // Characters EndOfLine
        NestingBlockComment

Characters:
        Character
        Character Characters

NestingBlockComment:
        /+ NestingBlockCommentCharacters +/

NestingBlockCommentCharacters:
        NestingBlockCommentCharacter
        NestingBlockCommentCharacter NestingBlockCommentCharacters

NestingBlockCommentCharacter:
        Character
        NestingBlockComment
```

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.

The contents of strings and comments are not tokenized. Consequently, comment openings occurring within a string do not begin a comment, and string delimiters within a comment do not affect the recognition of comment closings and nested "/+" comment openings. With the exception of "/+" occurring within a "/+" comment, comment openings within a comment are ignored.

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

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

Tokens

```
Token:
        Identifier
        StringLiteral
        CharacterLiteral
        IntegerLiteral
        FloatLiteral
        Keyword
        /
        /=
        .
        ..
        ...
        &
        &=
        &&
        |
        |=
        ||
        -
        -=
        --
```

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

Identifiers

Identifier:
        IdentiferStart
        IdentiferStart IdentifierChars

IdentifierChars:
        IdentiferChar
        IdentiferChar IdentifierChars

IdentifierStart:

```
        _
        Letter
        UniversalAlpha

IdentifierChar:
        IdentiferStart
        0
        NonZeroDigit
```

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

String Literals

```
StringLiteral:
        WysiwygString
        AlternateWysiwygString
        DoubleQuotedString
        EscapeSequence
        HexString

WysiwygString:
        r" WysiwygCharacters " Postfix_opt

AlternateWysiwygString:
        ` WysiwygCharacters ` Postfix_opt

WysiwygCharacters:
        WysiwygCharacter
        WysiwygCharacter WysiwygCharacters

WysiwygCharacter:
        Character
        EndOfLine

DoubleQuotedString:
        " DoubleQuotedCharacters " Postfix_opt

DoubleQuotedCharacters:
        DoubleQuotedCharacter
        DoubleQuotedCharacter 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
        \& NamedCharacterEntity ;

HexString:
        x" HexStringChars " Postfix_opt

HexStringChars:
        HexStringChar
        HexStringChar HexStringChars

HexStringChar
        HexDigit
        WhiteSpace
        EndOfLine

Postfix
        c
        w
        d
```

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
\&reg;                  ® dchar character
\r\n                    carriage return, line feed
```

Undefined escape sequences are errors. Although string literals are defined to be composed of UTF characters, the octal and hex escape sequences allow the insertion of arbitrary binary data. \u and \U escape sequences can only be used to insert valid UTF characters.

Hex strings allow string literals to be created using hex data. The hex data need not form valid UTF characters.

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

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

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

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

The following are all equivalent:

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

The optional *Postfix* character gives a specific type to the string, rather than it being inferred from the context. This is useful when the type cannot be unambiguously inferred, such as when overloading based on string type. The types corresponding to the postfix characters are:

Postfix	Type
c	char[ ]
w	wchar[ ]
d	dchar[ ]

```
"hello"c          // char[]
"hello"w          // wchar[]
"hello"d          // dchar[]
```

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
        u
        U
        Lu
        LU
        uL
        UL

Decimal:
        0
        NonZeroDigit
        NonZeroDigit DecimalDigits

Binary:
        0b BinaryDigits
        0B BinaryDigits

Octal:
        0 OctalDigits

Hexadecimal:
        0x HexDigits
        0X HexDigits

NonZeroDigit:
        1
        2
        3
        4
        5
        6
        7
        8
        9

DecimalDigits:
        DecimalDigit
        DecimalDigit DecimalDigits

DecimalDigit:
        0
```

```
        NonZeroDigit
        _

BinaryDigits:
        BinaryDigit
        BinaryDigit BinaryDigits

BinaryDigit:
        0
        1
        _

OctalDigits:
        OctalDigit
        OctalDigit OctalDigits

OctalDigit:
        0
        1
        2
        3
        4
        5
        6
        7
        _

HexDigits:
        HexDigit
        HexDigit HexDigits

HexDigit:
        DecimalDigit
        a
        b
        c
        d
        e
        f
        A
        B
        C
        D
        E
        F
```

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

Decimal integers are a sequence of decimal digits.

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

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

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

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

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

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

The type of the integer is resolved as follows:

Decimal Literal	Type
0 .. 2147483647	int
2147483648 .. 9223372036854775807	long
Decimal Literal, L Suffix	**Type**
0L .. 9223372036854775807L	long
Decimal Literal, U Suffix	**Type**
0U .. 4294967295U	uint
4294967296U .. 18446744073709551615U	ulong
Decimal Literal, UL Suffix	**Type**
0UL .. 18446744073709551615UL	ulong
Non-Decimal Literal	**Type**
0x0 .. 0x7FFFFFFF	int
0x80000000 .. 0xFFFFFFFF	uint
0x100000000 .. 0x7FFFFFFFFFFFFFFF	long
0x8000000000000000 .. 0xFFFFFFFFFFFFFFFF	ulong
Non-Decimal Literal, L Suffix	**Type**
0x0L .. 0x7FFFFFFFFFFFFFFFL	long
0x8000000000000000L .. 0xFFFFFFFFFFFFFFFFL	ulong
Non-Decimal Literal, U Suffix	**Type**
0x0U .. 0xFFFFFFFFU	uint
0x100000000UL .. 0xFFFFFFFFFFFFFFFFUL	ulong
Non-Decimal Literal, UL Suffix	**Type**
0x0UL .. 0xFFFFFFFFFFFFFFFFUL	ulong

Floating Literals

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

Float:
        DecimalFloat
        HexFloat

DecimalFloat:
        DecimalDigits .
```

```
        DecimalDigits . DecimalDigits
        DecimalDigits . DecimalDigits DecimalExponent
        . Decimal
        . Decimal DecimalExponent
        DecimalDigits DecimalExponent

DecimalExponent
        e DecimalDigits
        E DecimalDigits
        e+ DecimalDigits
        E+ DecimalDigits
        e- DecimalDigits
        E- DecimalDigits

HexFloat:
        HexPrefix HexDigits .
        HexPrefix HexDigits . HexDigits
        HexPrefix HexDigits . HexDigits HexExponent
        HexPrefix . HexDigits
        HexPrefix . HexDigits HexExponent
        HexPrefix HexDigits HexExponent

HexPrefix:
        0x
        0X

HexExponent
        p DecimalDigits
        P DecimalDigits
        p+ DecimalDigits
        P+ DecimalDigits
        p- DecimalDigits
        P- DecimalDigits

FloatSuffix:
        f
        F
        L

ImaginarySuffix:
        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 decimal number serving as the exponent of 2.

Floating literals 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
```

Floating literals with no suffix are of type double. Floats can be followed by one **f**, **F**, or **L** suffix. The **f** or **F** suffix means it is a float, and **L** means it is a real.

If a floating literal is followed by **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

        body
        bool
        break
        byte

        case
        cast
        catch
        cdouble
        cent
        cfloat
        char
        class
        const
        continue
        creal

        dchar
        debug
        default
        delegate
        delete
        deprecated
        do
        double

        else
        enum
        export
        extern

        false
        final
```

```
finally
float
for
foreach
foreach_reverse
function

goto

idouble
if
ifloat
import
in
inout
int
interface
invariant
ireal
is

lazy
long

mixin
module

new
null

out
override

package
pragma
private
protected
public

real
return

scope
short
static
struct
super
switch
synchronized

template
this
throw
true
try
typedef
typeid
typeof

ubyte
ucent
```

```
        uint
        ulong
        union
        unittest
        ushort

        version
        void
        volatile

        wchar
        while
        with
```

Special Tokens

These tokens are replaced with other tokens according to the following table:

Special Token	Replaced with...
__FILE__	string literal containing source file name
__LINE__	integer literal of the current source line number
__DATE__	string literal of the date of compilation "*mmm dd yyyy*"
__TIME__	string literal of the time of compilation "*hh:mm:ss*"
__TIMESTAMP__	string literal of the date and time of compilation "*www mmm dd hh:mm:ss yyyy*"

Special Token Sequences

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

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

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

Filespec
        " Characters "
```

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

For example:

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

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

Modules

```
Module:
        ModuleDeclaration DeclDefs
        DeclDefs

DeclDefs:
        DeclDef
        DeclDef DeclDefs

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

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

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

- There's only one instance of each module, and it is statically allocated.
- There is no virtual table.
- Modules do not inherit, they have no super modules, etc.
- Only one module per file.
- Module symbols can be imported.
- Modules are always compiled at global scope, and are unaffected by surrounding attributes or other modifiers.

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

Module Declaration

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

```
ModuleDeclaration:
        module ModuleName ;

ModuleName:
        Identifier
        ModuleName . Identifier
```

The *Identifier* preceding the rightmost are the *packages* that the module is in. The packages

correspond to directory names in the source file path.

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

Example:

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

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

Import Declaration

Symbols from one module are made available in another module by using the *ImportDeclaration*:

```
ImportDeclaration:
        import ImportList ;
        static import ImportList ;

ImportList:
        Import
        ImportBindings
        Import , ImportList

Import:
        ModuleName
        ModuleAliasIdentifier = ModuleName

ImportBindings:
        Import : ImportBindList

ImportBindList:
        ImportBind
        ImportBind , ImportBindList

ImportBind:
        Identifier
        Identifier =
```

There are several forms of the *ImportDeclaration*, from generalized to fine-grained importing.

The order in which *ImportDeclarations* occur has no significance.

*ModuleName*s in the *ImportDeclaration* must be fully qualified with whatever packages they are in. They are not considered to be relative to the module that imports them.

Basic Imports

The simplest form of importing is to just list the modules being imported:

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

void main()
{
    writefln("hello!\n");  // calls std.stdio.writefln
}
```

How basic imports work is that first a name is searched for in the current namespace. If it is not found, then it is looked for in the imports. If it is found uniquely among the imports, then that is used. If it is in more than one import, an error occurs.

```
module A;
void foo();
void bar();

module B;
void foo();
void bar();

module C;
import A;
void foo();
void test()
{ foo(); // C.foo() is called, it is found before imports are searched
  bar(); // A.bar() is called, since imports are searched
}

module D;
import A;
import B;
void test()
{ foo();   // error, A.foo() or B.foo() ?
  A.foo(); // ok, call A.foo()
  B.foo(); // ok, call B.foo()
}

module E;
import A;
import B;
alias B.foo foo;
void test()
{ foo();   // call B.foo()
  A.foo(); // call A.foo()
  B.foo(); // call B.foo()
}
```

Public Imports

By default, imports are *private*. This means that if module A imports module B, and module B imports module C, then C's names are not searched for. An import can be specifically declared *public*, when it will be treated as if any imports of the module with the *ImportDeclaration* also import the public imported modules.

```
module A;
void foo() { }

module B;
void bar() { }

module C;
import A;
public import B;
...
foo();  // call  A.foo()
bar();  // calls B.bar()
```

```
module D;
import C;
...
foo();  // error, foo() is undefined
bar();  // ok, calls B.bar()
```

Static Imports

Basic imports work well for programs with relatively few modules and imports. If there are a lot of imports, name collisions can start occurring between the names in the various imported modules. One way to stop this is by using static imports. A static import requires one to use a fully qualified name to reference the module's names:

```
static import std.stdio;

void main()
{
    writefln("hello!");            // error, writefln is undefined
    std.stdio.writefln("hello!");  // ok, writefln is fully qualified
}
```

Renamed Imports

A local name for an import can be given, through which all references to the module's symbols must be qualified with:

```
import io = std.stdio;

void main()
{
    io.writefln("hello!");         // ok, calls std.stdio.writefln
    std.stdio.writefln("hello!");  // error, std is undefined
    writefln("hello!");            // error, writefln is undefined
}
```

Renamed imports are handy when dealing with very long import names.

Selective Imports

Specific symbols can be exclusively imported from a module and bound into the current namespace:

```
import std.stdio : writefln, foo = writef;

void main()
{
    std.stdio.writefln("hello!");  // error, std is undefined
    writefln("hello!");            // ok, writefln bound into current namespace
    writef("world");               // error, writef is undefined
    foo("world");                  // ok, calls std.stdio.writef()
    fwritefln(stdout, "abc");      // error, fwritefln undefined
}
```

static cannot be used with selective imports.

Renamed and Selective Imports

When renaming and selective importing are combined:

```
import io = std.stdio : foo = writefln;

void main()
{
    writefln("bar");            // error, writefln is undefined
    std.stdio.foo("bar");       // error, foo is bound into current namespace
    std.stdio.writefln("bar"); // error, std is undefined
    foo("bar");                 // ok, foo is bound into current namespace,
                                // FQN not required
    io.writefln("bar");         // ok, io=std.stdio bound the name io in
                                // the current namespace to refer to the entire
module
    io.foo("bar");              // error, foo is bound into current namespace,
                                // foo is not a member of io
```

Module Scope Operator

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

```
int x;

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

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

Static Construction and Destruction

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

Order of Static Construction

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

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

Order of Static Construction within a Module

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

Order of Static Destruction

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

Order of Unit tests

Unit tests are run in the lexical order in which they appear within a module.

Declarations

Declaration:
        typedef *Decl*
        alias *Decl*
        Decl

Decl:
        StorageClasses Decl
        BasicType Declarators ;
        BasicType Declarator FunctionBody
        AutoDeclaration

Declarators:
        DeclaratorInitializer
        DeclaratorInitializer , *DeclaratorIdentifierList*

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

DeclaratorIdentifierList:
        DeclaratorIdentifier
        DeclaratorIdentifier , *DeclaratorIdentifierList*

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

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

BasicType2:
        *****
        [ ]

```
        [ Expression ]
        [ Type ]
        delegate Parameters
        function Parameters

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

DeclaratorSuffixes:
        DeclaratorSuffix
        DeclaratorSuffix DeclaratorSuffixes

DeclaratorSuffix:
        [ ]
        [ Expression ]
        [ Type ]
        Parameters

IdentifierList:
        Identifier
        Identifier . IdentifierList
        TemplateInstance
        TemplateInstance . IdentifierList

Typeof:
        typeof ( Expression )

StorageClasses:
        StorageClass
        StorageClass StorageClasses

StorageClass:
        abstract
        auto
        const
        deprecated
        extern
        final
        override
        scope
        static
        synchronized

Type:
        BasicType
        BasicType Declarator2

Declarator2:
        BasicType2 Declarator2
        ( Declarator2 )
        ( Declarator2 ) DeclaratorSuffixes

Parameters:
        ( ParameterList )
        ( )

ParameterList:
```

```
        Parameter
        Parameter , ParameterList
        Parameter ...
        ...

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

InOut:
        in
        out
        inout
        lazy

Initializer:
        void
        NonVoidInitializer

NonVoidInitializer:
        AssignExpression
        ArrayInitializer
        StructInitializer

ArrayInitializer:
        [ ]
        [ ArrayMemberInitializations ]

ArrayMemberInitializations:
        ArrayMemberInitialization
        ArrayMemberInitialization ,
        ArrayMemberInitialization , ArrayMemberInitializations

ArrayMemberInitialization:
        NonVoidInitializer
        AssignExpression : NonVoidInitializer

StructInitializer:
        {  }
        { StructMemberInitializers }

StructMemberInitializers:
        StructMemberInitializer
        StructMemberInitializer ,
        StructMemberInitializer , StructMemberInitializers

StructMemberInitializer:
        NonVoidInitializer
        Identifier : NonVoidInitializer

AutoDeclaration:
        StorageClasses Identifier = AssignExpression ;
```

Declaration Syntax

Declaration syntax generally reads right to left:

```
int x;          // x is an int
int* x;         // x is a pointer to int
```

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

Arrays read right to left as well:

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

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

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

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

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

In a declaration declaring multiple symbols, 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
```

Implicit Type Inference

If a declaration starts with a *StorageClass* and has a *NonVoidInitializer* from which the type can be inferred, the type on the declaration can be omitted.

```
static x = 3;      // x is type int
auto y = 4u;       // y is type uint
auto s = "string"; // s is type char[6]

class C { ... }

auto c = new C();  // c is a handle to an instance of class C
```

The *NonVoidInitializer* cannot contain forward references (this restriction may be removed in the future). The implicitly inferred type is statically bound to the declaration at compile time, not run time.

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.

Aliases cannot be used for expressions:

```
struct S { static int i; }
S s;

alias s.i a;    // illegal, s.i is an expression
alias S.i b;    // ok
b = 4;          // sets S.i to 4
```

Extern Declarations

Variable declarations with the storage class **extern** are not allocated storage within the module. They must be defined in some other object file with a matching name which is then linked in. The primary usefulness of this is to connect with global variable declarations in C files.

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').sizeof); // prints 1
    double c = cast(typeof(1.0))j;      // cast j to double
}
```

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

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

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

```
class A { }

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

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

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

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

Void Initializations

Normally, variables are initialized either with an explicit *Initializer* or are set to the default value for the type of the variable. If the *Initializer* is **void**, however, the variable is not initialized. If its value is used before it is set, undefined program behavior will result.

```
void foo()
{
```

```
    int x = void;
    writefln(x);        // will print garbage
}
```

Therefore, one should only use **void** initializers as a last resort when optimizing critical code.

Types

Basic Data Types

Keyword	Description	Default Initializer (.init)
`void`	no type	-
`bool`	boolean value	false
`byte`	signed 8 bits	0
`ubyte`	unsigned 8 bits	0
`short`	signed 16 bits	0
`ushort`	unsigned 16 bits	0
`int`	signed 32 bits	0
`uint`	unsigned 32 bits	0
`long`	signed 64 bits	0L
`ulong`	unsigned 64 bits	0L
`cent`	signed 128 bits (reserved for future use)	0
`ucent`	unsigned 128 bits (reserved for future use)	0
`float`	32 bit floating point	float.nan
`double`	64 bit floating point	double.nan
`real`	largest hardware implemented floating point size (**Implementation Note:** 80 bits for Intel CPUs)	real.nan
`ifloat`	imaginary float	float.nan * 1.0i
`idouble`	imaginary double	double.nan * 1.0i
`ireal`	imaginary real	real.nan * 1.0i
`cfloat`	a complex number of two float values	float.nan + float.nan * 1.0i
`cdouble`	complex double	double.nan + double.nan * 1.0i
`creal`	complex real	real.nan + real.nan * 1.0i
`char`	unsigned 8 bit UTF-8	0xFF
`wchar`	unsigned 16 bit UTF-16	0xFFFF
`dchar`	unsigned 32 bit UTF-32	0x0000FFFF

Derived Data Types

- pointer
- array
- associative array
- function
- delegate

User Defined Types

- alias
- typedef
- enum
- struct
- union
- class

Base Types

The *base type* of an enum is the type it is based on:

```
enum E : T { ... }      // T is the base type of E
```

The *base type* of a typedef is the type it is formed from:

```
typedef T U;            // T is the base type of U
```

Pointer Conversions

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

Implicit Conversions

Implicit conversions are used to automatically convert types as required.

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

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

Integer Promotions

Integer Promotions are conversions of the following types:

from	to

bool	int
byte	int
ubyte	int
short	int
ushort	int
char	int
wchar	int
dchar	uint

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

Usual Arithmetic Conversions

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

1. If either operand is real, the other operand is converted to real.
2. Else if either operand is double, the other operand is converted to double.
3. Else if either operand is float, the other operand is converted to float.
4. Else the integer promotions are done on each operand, followed by:
    1. If both are the same type, no more conversions are done.
5. If both are signed or both are unsigned, the smaller type is converted to the larger.
6. If the signed type is larger than the unsigned type, the unsigned type is converted to the signed type.
7. The signed type is converted to the unsigned type.

If one or both of the operand types is a typedef or enum after undergoing the above conversions, the result type is:

1. If the operands are the same type, the result will be the that type.
2. If one operand is a typedef or enum and the other is the base type of that typedef or enum, the result is the base type.
3. If the two operands are different typedefs or enums but of the same base type, then the result is that base type.

Integer values cannot be implicitly converted to another type that cannot represent the integer bit pattern after integral promotion. For example:

```
ubyte  u1 = cast(byte)-1;   // error, -1 cannot be represented in a ubyte
ushort u2 = cast(short)-1;  // error, -1 cannot be represented in a ushort
uint   u3 = cast(int)-1;    // ok, -1 can be represented in a uint
ulong  u4 = cast(ulong)-1;  // ok, -1 can be represented in a ulong
```

Floating point types cannot be implicitly converted to integral types.

Complex floating point types cannot be implicitly converted to non-complex floating point types.

Imaginary floating point types cannot be implicitly converted to float, double, or real types. Float, double, or real types cannot be implicitly converted to imaginary floating point types.

bool

The bool type is a 1 byte size type that can only hold the value true or false. The only operators that can accept operands of type bool are: & | ^ &= |= ^= ! && || ?:. A bool value can be implicitly converted to any integral type, with false becoming 0 and true becoming 1. The numeric literals 0 and 1 can be implicitly converted to the bool values false and true, respectively. Casting an expression to bool means testing for 0 or !=0 for arithmetic types, and null or !=null for pointers or references.

Delegates

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

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

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

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

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

A delegate is initialized analogously to function pointers:

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

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

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

Delegates are called analogously to function pointers:

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

Properties

Every type and expression has properties that can be queried:

Expression	Value
int.sizeof	yields 4
float.nan	yields the floating point nan (Not A Number) value
(float).nan	yields the floating point nan value
(3).sizeof	yields 4 (because 3 is an int)
2.sizeof	syntax error, since "2." is a floating point number
int.init	default initializer for int's
int.mangleof	yields the string "i"

Properties for All Types

Property	Description
.init	initializer
.sizeof	size in bytes (equivalent to C's sizeof(type))
.alignof	alignment size
.mangleof	string representing the 'mangled' representation of the type

Properties for Integral Types

Property	Description
.init	initializer (0)
.max	maximum value
.min	minimum value

Properties for Floating Point Types

Property	Description
.init	initializer (NaN)
.infinity	infinity value
.nan	NaN value
.dig	number of decimal digits of precision
.epsilon	smallest increment to the value 1

.mant_dig	number of bits in mantissa
.max_10_exp	maximum int value such that $10^{max\_10\_exp}$ is representable
.max_exp	maximum int value such that 2^{max_exp-1} is representable
.min_10_exp	minimum int value such that $10^{min\_10\_exp}$ is representable as a normalized value
.min_exp	minimum int value such that 2^{min_exp-1} is representable as a normalized value
.max	largest representable value that's not infinity
.min	smallest representable normalized value that's not 0
.re	real part
.im	imaginary part

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

A simple property would be:

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

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

  private:
    int m_data;
}
```

To use it:

```
int test()
{
    Foo f;

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

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

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

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

Attributes

```
AttributeSpecifier:
    Attribute :
    Attribute DeclarationBlock

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

DeclarationBlock
    DeclDef
    { }
    { DeclDefs }
```

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

```
attribute declaration;          affects the declaration

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

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

For attributes with an optional else clause:

```
attribute
    declaration;
else
    declaration;

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

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

Linkage Attribute

LinkageAttribute:
        extern
        extern (*LinkageType*)

LinkageType:
        C
        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. **C++** is reserved for future use. **Implementation Note:** for Win32 platforms, **Windows** and **Pascal** should exist.

C function calling conventions are specified by:

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

D conventions are:

```
extern (D):
```

or:

```
extern:
```

Windows API conventions are:

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

Align Attribute

AlignAttribute:
        align
        align (*Integer*)

Specifies the alignment of struct members. **align** by itself sets it to the default, which matches the default member alignment of the companion C compiler. *Integer* specifies the alignment which matches the behavior of the companion C compiler when non-default alignments are used.

Matching the behavior of the companion C compiler can have some surprising results, such as the following for Digital Mars C++:

```
struct S
{   align(4) byte a;    // placed at offset 0
    align(4) byte b;    // placed at offset 1
}
```

AlignAttribute is meant for C ABI compatiblity, which is not the same thing as binary compatibility across diverse platforms. For that, use packed structs:

```
align (1) struct S
{   byte a;     // placed at offset 0
    byte[3] filler1;
    byte b;     // placed at offset 4
    byte[3] filler2;
}
```

A value of 1 means that no alignment is done; members are packed together.

Do not align references or pointers that were allocated using *NewExpression* on boundaries that are not a multiple of `size_t`. The garbage collector assumes that pointers and references to gc allocated objects will be on `size_t` byte boundaries. If they are not, undefined behavior will result.

AlignAttribute is ignored when applied to declarations that are not structs or struct members.

Deprecated Attribute

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

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

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

Protection Attribute

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

Private means that only members of the enclosing class can access the member, or members and functions in the same module as the enclosing class. Private members cannot be overridden. Private module members are equivalent to **static** declarations in C programs.

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

Protected means that only members of the enclosing class or any classes derived from that class, or members and functions in the same module as the enclosing class, can access the member. If accessing a protected instance member through a derived class member function, that member can only be accessed for the object instance which is the 'this' object for the member function call.

Protected module members are illegal.

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

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

Const Attribute

const

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

```
const int foo = 7;

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

A const declaration without an initializer must be initialized in a constructor (for class fields) or in a static constructor (for static class members, or module variable declarations).

```
const int x;
const int y;

static this()
{
    x = 3;      // ok
    // error: y not initialized
}

void foo()
{
    x = 4;      // error, x is const and not in static constructor
}

class C
{
    const int a;
    const int b;
    static const int c;
    static const int d;

    this()
    {   a = 3;          // ok
        a = 4;          // ok, multiple initialization allowed
        C p = this;
        p.a = 4;        // error, only members of this instance
        c = 5;          // error, should initialize in static constructor
        // error, b is not initialized
    }

    this(int x)
    {
        this();         // ok, forwarding constructor
    }

    static this()
    {
```

```
        c = 3;          // ok
        // error, d is not initialized
    }
}
```

It is not an error to have const module variable declarations without initializers if there is no constructor. This is to support the practice of having modules serve only as declarations that are not linked in, the implementation of it will be in another module that is linked in.

Override Attribute

override

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

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

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

Static Attribute

static

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

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

...

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

Static functions are never virtual.

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

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

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

Auto Attribute

auto

The **auto** attribute is used when there are no other attributes and type inference is desired.

```
auto i = 6.8;   // declare i as a double
```

Scope Attribute

scope

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

If there is more than one **scope** variable going out of scope at the same point, then the destructors are called in the reverse order that the variables were constructed.

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

Abstract Attribute

If a class is abstract, it cannot be instantiated directly. It can only be instantiated as a base class of another, non-abstract, class.

Classes become abstract if they are defined within an abstract attribute, or if any of the virtual member functions within it are declared as abstract.

Non-virtual functions cannot be declared as abstract.

Functions declared as abstract can still have function bodies. This is so that even though they must be overridden, they can still provide 'base class functionality.'

Pragmas

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

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

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

pragma(ident) declaration; // influence one declaration

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

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

pragma(ident) statement;   // influence one statement

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

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

Predefined Pragmas

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

msg

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

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

lib

Inserts a directive in the object file to link in the library specified by the *AssignExpression*. The *AssignExpression*s must be a string literal:

```
pragma(lib, "foo.lib");
```

Vendor Specific Pragmas

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

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

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

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

Expressions

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

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

StringLiterals:
        StringLiteral
        StringLiterals *StringLiteral*

ArgumentList:
        AssignExpression
        AssignExpression , *ArgumentList*

Evaluation Order

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

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

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

Expressions

Expression:
        AssignExpression
        AssignExpression , *Expression*

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

Assign Expressions

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*

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

Assignment operator expressions, such as:

```
a op= b
```

are semantically equivalent to:

```
a = a op b
```

except that operand *a* is only evaluated once.

Conditional Expressions

```
ConditionalExpression:
        OrOrExpression
        OrOrExpression ? Expression : ConditionalExpression
```

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

OrOr Expressions

```
OrOrExpression:
        AndAndExpression
        OrOrExpression || AndAndExpression
```

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

The *OrOrExpression* 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 *OrOrExpression* 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 *OrOrExpression* is bool then the result of the expression is the right operand converted to type bool.

AndAnd Expressions

```
AndAndExpression:
        OrExpression
        AndAndExpression && OrExpression
```

The result type of an *AndAndExpression* is bool, unless the right operand has type void, when the

result is type void.

The *AndAndExpression* 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 *AndAndExpression* 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 *AndAndExpression* is bool then the result of the expression is the right operand converted to type bool.

Bitwise Expressions

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

Or Expressions

```
OrExpression:
        XorExpression
        OrExpression | XorExpression
```

The operands are OR'd together.

Xor Expressions

```
XorExpression:
        AndExpression
        XorExpression ^ AndExpression
```

The operands are XOR'd together.

And Expressions

```
AndExpression:
        EqualExpression
        AndExpression & EqualExpression
```

The operands are AND'd together.

Equality Expressions

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

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

If they are integral values or pointers, equality is defined as the bit pattern of the type matches exactly. Equality for struct objects means the bit patterns of the objects match exactly (the existence

of alignment holes in the objects is accounted for, usually by setting them all to 0 upon initialization). Equality for floating point types is more complicated. -0 and +0 compare as equal. If either or both operands are NAN, then both the == returns false and != returns true. 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 and struct objects, the expression `(a == b)` is rewritten as `a.opEquals(b)`, and `(a != b)` is rewritten as `!a.opEquals(b)`.

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

Identity Expressions

EqualExpression **is** *RelExpression*
EqualExpression **!is** *RelExpression*

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

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

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

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

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

Relational Expressions

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

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.opCmp() forms the left operand, and 0 forms the right operand. The result of the relational expression (o1 op o2) is:

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

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

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

Integer comparisons

Integer comparisons happen when both operands are integral types.

Integer comparison operators

Operator	Relation
<	less
>	greater
<=	less or equal
>=	greater or equal
==	equal
!=	not equal

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

Floating point comparisons

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

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

Floating point comparison operators

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

!<>=	F	F	F	T	no	unordered
<>	T	T	F	F	yes	less or greater
<>=	T	T	T	F	yes	less, equal, or greater
!<=	T	F	F	T	no	unordered or greater
!<	T	F	T	T	no	unordered, greater, or equal
!>=	F	T	F	T	no	unordered or less
!>	F	T	T	T	no	unordered, less, or equal
!<>	F	F	T	T	no	unordered or equal

Notes:

1. For floating point comparison operators, (a !op b) is not the same as !(a op b).
2. "Unordered" means one or both of the operands is a NAN.
3. "Exception" means the *Invalid Exception* is raised if one of the operands is a NAN. It does not mean an exception is thrown. The *Invalid Exception* can be checked using the functions in std.c.fenv.

In Expressions

InExpression:
        RelExpression **in** *ShiftExpression*

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

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

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

Shift Expressions

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

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

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

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

```
int c;
```

```
c << 33;        // error
```

Add Expressions

```
AddExpression:
        MulExpression
        AddExpression + MulExpression
        AddExpression - MulExpression
        CatExpression
```

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

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

If the operator is + or -, and the first operand is a pointer, and the second is an integral type, the resulting type is the type of the first operand, and the resulting value is the pointer plus (or minus) the second operand multiplied by the size of the type pointed to by the first operand.

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

Add expressions for floating point operands are not associative.

Cat Expressions

```
CatExpression:
        AddExpression ~ MulExpression
```

A *CatExpression* concatenates arrays, producing a dynmaic array with the result. The arrays must be arrays of the same element type. If one operand is an array and the other is of that array's element type, that element is converted to an array of length 1 of that element, and then the concatenation is performed.

Mul Expressions

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

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

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

For integral operands of the **%** operator, the sign of the result is positive if the operands are positive, otherwise the sign of the result is implementation defined.

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

operands.

Mul expressions for floating point operands are not associative.

Unary Expressions

UnaryExpression:
        PostfixExpression
        & *UnaryExpression*
        ++ *UnaryExpression*
        -- *UnaryExpression*
        ***** *UnaryExpression*
        - *UnaryExpression*
        + *UnaryExpression*
        ! *UnaryExpression*
        ~ *UnaryExpression*
        (*Type* **) .** *Identifier*
        NewExpression
        DeleteExpression
        CastExpression
        NewAnonClassExpression

New Expressions

NewExpression:
        NewArguments Type **[** *AssignExpression* **]**
        NewArguments Type **(** *ArgumentList* **)**
        NewArguments Type
        NewArguments ClassArguments BaseClasslist$_{opt}$ **{** *DeclDefs* **}**

NewArguments:
        new (*ArgumentList* **)**
        new ( )
        new

ClassArguments:
        class (*ArgumentList* **)**
        class ( )
        class

*NewExpression*s are used to allocate memory on the garbage collected heap (default) or using a class or struct 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 array of 30 strings
```

The above allocation can also be written as:

```
foo = new char[][](30); // allocate array of 30 strings
```

To allocate the nested arrays, multiple arguments can be used:

```
int[][][] bar;
...
bar = new int[][][](5,20,30);
```

Which is equivalent to:

```
bar = new int[][][5];
foreach (inout a; bar)
{
    a = new int[][20];
    foreach (inout b; a)
    {
        b = new int[30];
    }
}
```

If there is a **new (** *ArgumentList* **)**, then those arguments are passed to the class or struct specific allocator function after the size argument.

If a *NewExpression* is used as an initializer for a function local variable with **scope** storage class, and the *ArgumentList* to **new** is empty, then the instance is allocated on the stack rather than the heap or using the class specific allocator.

Delete Expressions

DeleteExpression:
        delete *UnaryExpression*

If the *UnaryExpression* is a class object reference, and there is a destructor for that class, the destructor is called for that object instance.

Next, if the *UnaryExpression* is a class object reference, or a pointer to a struct instance, and the class or struct has overloaded operator delete, then that operator delete is called for that class object instance or struct instance.

Otherwise, the garbage collector is called to immediately free the memory allocated for the class instance or struct instance. If the garbage collector was not used to allocate the memory for the instance, undefined behavior will result.

If the *UnaryExpression* is a pointer or a dynamic array, the garbage collector is called to immediately release the memory. If the garbage collector was not used to allocate the memory for the instance, undefined behavior will result.

The pointer, dynamic array, or reference is set to **null** after the delete is performed.

If *UnaryExpression* is a variable allocated on the stack, the class destructor (if any) is called for that instance. Neither the garbage collector nor any class deallocator is called.

Cast Expressions

CastExpression:
        cast (***Type* **) *UnaryExpression*

A *CastExpression* converts the *UnaryExpression* to *Type*.

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

Any casting of a class reference to a derived class reference is done with a runtime check to make sure it really is a downcast. **null** is the result if it isn't. **Note:** This is equivalent to the behavior of the dynamic_cast operator in C++.

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

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

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

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

Casting a floating point literal from one type to another changes its type, but internally it is retained at full precision for the purposes of constant folding.

```
void test()
{
    real a = 3.40483L;
    real b;
    b = 3.40483;         // literal is not truncated to double precision
    assert(a == b);
    assert(a == 3.40483);
    assert(a == 3.40483L);
    assert(a == 3.40483F);
    double d = 3.40483; // truncate literal when assigned to variable
    assert(d != a);     // so it is no longer the same
    const double x = 3.40483; // assignment to const is not
    assert(x == a);           // truncated if the initializer is visible
}
```

Casting a value *v* to a struct *S*, when value is not a struct of the same type, is equivalent to:

```
S(v)
```

Postfix Expressions

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

Index Expressions

```
IndexExpression:
        PostfixExpression [ ArgumentList ]
```

PostfixExpression is evaluated. If *PostfixExpression* is an expression of type static array or dynamic array, the variable **length** is declared and set to be the length of the array. if *PostfixExpression* is an *ExpressionTuple*, the variable **length** (and the special variable $) is declared and set to be the the number of elements in the tuple. A new declaration scope is created for the evaluation of the *ArgumentList* and **length** (and $) appears in that scope only.

If *PostfixExpression* is an *ExpressionTuple*, then the *ArgumentList* must consist of only one argument, and that must be statically evaluatable to an integral constant. That integral constant *n* then selects the *n*th expression in the *ExpressionTuple*, which is the result of the *IndexExpression*. It is an error if *n* is out of bounds of the *ExpressionTuple*.

Slice Expressions

```
SliceExpression:
        PostfixExpression [ ]
        PostfixExpression [ AssignExpression .. AssignExpression ]
```

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

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

If the **[ ]** form is used, the slice is of the entire array.

The type of the slice is a dynamic array of the element type of the *PostfixExpression*.

If *PostfixExpression* is an *ExpressionTuple*, then the result of the slice is a new *ExpressionTuple* formed from the upper and lower bounds, which must statically evaluate to integral constants. It is an error if those bounds are out of range.

Primary Expressions

```
PrimaryExpression:
        Identifier
        .Identifier
        this
        super
        null
        true
        false
        NumericLiteral
        CharacterLiteral
        StringLiterals
        ArrayLiteral
        FunctionLiteral
        AssertExpression
        BasicType . Identifier
```

```
typeid ( Type )
IsExpression
( Expression )
```

.Identifier

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

this

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

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

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

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

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

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

super

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

null

null represents the null value for pointers, pointers to functions, delegates, dynamic arrays, associative arrays, and class objects. If it has not already been cast to a type, it is given the type (void *) and it is an exact conversion to convert it to the null value for pointers, pointers to functions, delegates, etc. After it is cast to a type, such conversions are implicit, but no longer exact.

true, false

These are of type **bool** and when cast to another integral type become the 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.

Array Literals

ArrayLiteral:
        [*ArgumentList* **]**

Array literals are a comma-separated list of *AssignExpression*s between square brackets [and]. The *AssignExpression*s form the elements of a static array, the length of the array is the number of elements. The type of the first element is taken to be the type of all the elements, and all elements are implicitly converted to that type. If that type is a static array, it is converted to a dynamic array.

```
[1,2,3];        // type is int[3], with elements 1, 2 and 3
[1u,2,3];       // type is uint[3], with elements 1u, 2u, and 3u
```

If any of the arguments in the *ArgumentList* are an *ExpressionTuple*, then the elements of the *ExpressionTuple* are inserted as arguments in place of the tuple.

Function Literals

FunctionLiteral
        function *Type*$_{opt}$ **(** *ArgumentList* **)**$_{opt}$ *FunctionBody*
        delegate *Type*$_{opt}$ **(** *ArgumentList* **)**$_{opt}$ *FunctionBody*
        (*ArgumentList* **)** *FunctionBody*
        FunctionBody

*FunctionLiteral*s enable embedding anonymous functions and anonymous delegates directly into expressions. *Type* is the return type of the function or delegate, if omitted it is inferred from any *ReturnStatement*s in the *FunctionBody*. **(** *ArgumentList* **)** forms the arguments to the function. If omitted it defaults to the empty argument list **()**. The type of a function literal is pointer to function or pointer to delegate. If the keywords **function** or **delegate** are omitted, it defaults to being a delegate.

For example:

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

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

    fp = &foo;
}
```

is exactly equivalent to:

```
int function(char c) fp;

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

And:

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

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

    abc(&foo);
}
```

is exactly equivalent to:

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

void test()
{   int b = 3;

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

and the following where the return type **int** is inferred:

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

void test()
{   int b = 3;

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

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

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

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

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

    return d + f;
}
```

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

Assert Expressions

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

```
        assert ( Expression , Expression )
```

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

The expression `assert(0)` is a special case; it signifies that it is unreachable code. Either **AssertError** is thrown at runtime if it is reachable, or the execution is halted (on the x86 processor, a **HLT** instruction can be used to halt execution). The optimization and code generation phases of compilation may assume that it is unreachable code.

The second *Expression*, if present, must be implicitly convertible to type `char[]`. It is evaluated if the result is false, and the string result is appended to the **AssertError**'s message.

```
void main()
{
    assert(0, "an" ~ " error message");
}
```

When compiled and run, it will produce the message:

```
Error: AssertError Failure test.d(3) an error message
```

Typeid Expressions

```
TypeidExpression:
    typeid ( Type )
```

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

IsExpression

```
IsExpression:
        is ( Type )
        is ( Type : TypeSpecialization )
        is ( Type == TypeSpecialization )
        is ( Type Identifier )
        is ( Type Identifier : TypeSpecialization )
        is ( Type Identifier == TypeSpecialization )

TypeSpecialization:
        Type
        typedef
        struct
        union
        class
        interface
        enum
        function
        delegate
        super
```

*IsExpression*s are evaluated at compile time and are used for checking for valid types, comparing types for equivalence, determining if one type can be implicitly converted to another, and deducing the subtypes of a type. The result of an *IsExpression* is an int of type 0 if the condition is not

satisified, 1 if it is.

Type is the type being tested. It must be syntactically correct, but it need not be semantically correct. If it is not semantically correct, the condition is not satisfied.

Identifier is declared to be an alias of the resulting type if the condition is satisfied. The *Identifier* forms can only be used if the *IsExpression* appears in a *StaticIfCondition*.

TypeSpecialization is the type that *Type* is being compared against.

The forms of the *IsExpression* are:

1. **is (** *Type* **)**
   The condition is satisfied if *Type* is semantically correct (it must be syntactically correct regardless).

```
alias int func(int);    // func is a alias to a function type
void foo()
{
    if ( is(func[]) )   // not satisfied because arrays of
                        // functions are not allowed
        printf("satisfied\n");
    else
        printf("not satisfied\n");

    if (is([][]))       // error, [][] is not a syntactically valid type
        ...
}
```

2. **is (** *Type* **:** *TypeSpecialization* **)**
   The condition is satisfied if *Type* is semantically correct and it is the same as or can be implicitly converted to *TypeSpecialization*. *TypeSpecialization* is only allowed to be a *Type*.

```
alias short bar;
void foo(bar x)
{
    if ( is(bar : int) )        // satisfied because short can be
                                // implicitly converted to int
        printf("satisfied\n");
    else
        printf("not satisfied\n");
}
```

3. **is (** *Type* == *TypeSpecialization* **)**
   The condition is satisfied if *Type* is semantically correct and is the same type as *TypeSpecialization*.

   If *TypeSpecialization* is one of **typedef struct union class interface enum function delegate** then the condition is satisifed if *Type* is one of those.

```
alias short bar;
typedef char foo;
void test(bar x)
{
    if ( is(bar == int) )       // not satisfied because short is not
                                // the same type as int
        printf("satisfied\n");
    else
        printf("not satisfied\n");

    if ( is(foo == typedef) ) // satisfied because foo is a typedef
```

```
        printf("satisfied\n");
    else
        printf("not satisfied\n");
}
```

4. **is (** *Type Identifier* **)**
   The condition is satisfied if *Type* is semantically correct. If so, *Identifier* is declared to be an alias of *Type*.

```
alias short bar;
void foo(bar x)
{
    static if ( is(bar T) )
        alias T S;
    else
        alias long S;
    writefln(typeid(S));   // prints "short"

    if ( is(bar T) )  // error, Identifier T form can
                      // only be in StaticIfConditions
        ...
}
```

5. **is (** *Type Identifier* **:** *TypeSpecialization* **)**
   The condition is satisfied if *Type* is the same as or can be implicitly converted to *TypeSpecialization*. The *Identifier* is declared to be either an alias of the *TypeSpecialization* or, if *TypeSpecialization* is dependent on *Identifier*, the deduced type. *TypeSpecialization* is only allowed to be a *Type*.

```
alias short bar;
alias long* abc;
void foo(bar x, abc a)
{
    static if ( is(bar T : int) )
        alias T S;
    else
        alias long S;

    writefln(typeid(S));        // prints "int"

    static if ( is(abc U : U*) )
        U u;

    writefln(typeid(typeof(u)));        // prints "long"
}
```

   The way the type of *Identifier* is determined is analogous to the way template parameter types are determined by *TemplateTypeParameterSpecialization*.

6. **is (** *Type Identifier* == *TypeSpecialization* **)**
   The condition is satisfied if *Type* is semantically correct and is the same as *TypeSpecialization*. The *Identifier* is declared to be either an alias of the *TypeSpecialization* or, if *TypeSpecialization* is dependent on *Identifier*, the deduced type.

   If *TypeSpecialization* is one of **typedef struct union class interface enum function delegate** then the condition is satisifed if *Type* is one of those. Furthermore, *Identifier* is set to be an alias of the type:

keyword	alias type for *Identifier*
typedef	the type that *Type* is a typedef of
struct	*Type*
union	*Type*
class	*Type*
interface	*Type*
super	*TypeTuple* of base classes and interfaces
enum	the base type of the enum
function	*TypeTuple* of the function parameter types
delegate	the function type of the delegate
return	the return type of the function, delegate, or function pointer

```
alias short bar;
enum E : byte { Emember }
void foo(bar x)
{
    static if ( is(bar T == int) ) // not satisfied, short is not int
        alias T S;
    alias T U;                     // error, T is not defined

    static if ( is(E V == enum) )  // satisified, E is an enum
        V v;                       // v is declared to be a byte
}
```

Statements

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

```
Statement:
    ;
    NonEmptyStatement
    ScopeBlockStatement

NoScopeNonEmptyStatement:
    NonEmptyStatement
    BlockStatement

NoScopeStatement:
    ;
    NonEmptyStatement
    BlockStatement

NonEmptyOrScopeBlockStatement:
    NonEmptyStatement
    ScopeBlockStatement

NonEmptyStatement:
    LabeledStatement
    ExpressionStatement
    DeclarationStatement
    IfStatement
    ConditionalStatement
    WhileStatement
    DoStatement
    ForStatement
    ForeachStatement
    SwitchStatement
    CaseStatement
    DefaultStatement
    ContinueStatement
    BreakStatement
    ReturnStatement
    GotoStatement
    WithStatement
    SynchronizedStatement
    TryStatement
    ScopeGuardStatement
    ThrowStatement
    VolatileStatement
    AsmStatement
    PragmaStatement
```

Scope Statements

```
ScopeStatement:
    NonEmptyStatement
    BlockStatement
```

A new scope for local symbols is introduced for the *NonEmptyStatement* or *BlockStatement.*

Even though a new scope is introduced, local symbol declarations cannot shadow (hide) other local symbol declarations in the same function.

```
void func1(int x)
{   int x;      // illegal, x shadows parameter x

    int y;

    { int y; }  // illegal, y shadows enclosing scope's y

    void delegate() dg;
    dg = { int y; };    // ok, this y is not in the same function

    struct S
    {
        int y;          // ok, this y is a member, not a local
    }

    { int z; }
    { int z; }  // ok, this z is not shadowing the other z

    { int t; }
    { t++;   }  // illegal, t is undefined
}
```

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

Scope Block Statements

ScopeBlockStatement:
    BlockStatement

A scope block statement introduces a new scope for the *BlockStatement*.

Labeled Statements

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

LabelledStatement:
    Identifier ':' *NoScopeStatement*

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

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

Block Statement

BlockStatement:
    { }
    { *StatementList* **}**

StatementList:
    Statement
    Statement *StatementList*

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

Expression Statement

ExpressionStatement:
    Expression ;

The expression is evaluated.

Expressions that have no effect, like `(x + x)`, are illegal in expression statements. If such an expression is needed, casting it to void will make it legal.

```
int x;
x++;                // ok
x;                  // illegal
1+1;                // illegal
cast(void)(x + x);  // ok
```

Declaration Statement

Declaration statements declare variables and types.

DeclarationStatement:
    Declaration

Some declaration statements:

```
int a;           // declare a as type int and initialize it to 0
struct S { }     // declare struct s
alias int myint;
```

If Statement

If statements provide simple conditional execution of statements.

IfStatement:
        if (*IfCondition*) *ThenStatement*
        if (*IfCondition*) *ThenStatement* **else** *ElseStatement*

IfCondition:
        Expression
        auto *Identifier* **=** *Expression*
        Declarator **=** *Expression*

ThenStatement:
        ScopeStatement

ElseStatement:
        ScopeStatement

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

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

If an **auto** *Identifier* is provided, it is declared and initialized to the value and type of the

Expression. Its scope extends from when it is initialized to the end of the *ThenStatement*.

If a *Declarator* is provided, it is declared and initialized to the value of the *Expression*. Its scope extends from when it is initialized to the end of the *ThenStatement*.

```
import std.regexp;
...
if (auto m = std.regexp.search("abcdef", "b(c)d"))
{
    writefln("[%s]", m.pre);      // prints [a]
    writefln("[%s]", m.post);     // prints [ef]
    writefln("[%s]", m.match(0)); // prints [bcd]
    writefln("[%s]", m.match(1)); // prints [c]
    writefln("[%s]", m.match(2)); // prints []
}
else
{
    writefln(m.post);    // error, m undefined
}
writefln(m.pre);         // error, m undefined
```

While Statement

WhileStatement:
    while (*Expression*) *ScopeStatement*

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

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

A *BreakStatement* will exit the loop. A #ContinueStatementwill transfer directly to evaluating *Expression* again.

Do Statement

DoStatement:
    do *ScopeStatement* **while** (*Expression*)

Do while statements implement simple loops. *ScopeStatement* 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.

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

A *BreakStatement* will exit the loop. A *ContinueStatement* will transfer directly to evaluating

Expression again.

For Statement

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

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

Initialize:
        ;
        NoScopeNonEmptyStatement

Test:
        empty
        Expression

Increment:
        empty
        Expression
```

Initialize 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 *BreakStatement* will exit the loop. A *ContinueStatement* will transfer directly to the *Increment*.

A *ForStatement* creates a new scope. If *Initialize* declares a variable, that variable's scope extends through the end of the for 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 *Initialize* may be omitted. *Test* may also be omitted, and if so, it is treated as if it evaluated to true.

Foreach Statement

A foreach statement loops over the contents of an aggregate.

```
ForeachStatement:
    Foreach (ForeachTypeList; Aggregate) ScopeStatement

Foreach:
    foreach
    foreach_reverse

ForeachTypeList:
    ForeachType
    ForeachType , ForeachTypeList

ForeachType:
    inout Type Identifier
    Type Identifier
    inout Identifier
    Identifier

Aggregate:
    Expression
    Tuple
```

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

The aggregate must be loop invariant, meaning that elements to the aggregate cannot be added or removed from it in the *NoScopeNonEmptyStatement.*

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

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

For **foreach**, the elements for the array are iterated over starting at index 0 and continuing to the maximum of the array. For **foreach_reverse**, the array elements are visited in the reverse order.

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

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

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

dchar[] b = "\u2260";
```

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

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'
```

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. The order in which the elements of the array is unspecified for **foreach**. **foreach_reverse** for associative arrays is illegal.

```
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 it is a struct or class object, the **foreach** is defined by the special *opApply* member function. The **foreach_reverse** behavior is defined by the special *opApplyReverse* member function. These special functions must be defined by the type in order to use the corresponding foreach statement. The functions have the type:

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

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

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

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
```

If *Aggregate* is a delegate, the type signature of the delegate is of the same as for **opApply**. This enables many different named looping strategies to coexist in the same class or struct.

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
```

inout can not be applied to the index values.

If not specified, the *Type*s in the *ForeachType* can be inferred from the type of the *Aggregate*.

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 = a.length + 10;   // error
    a = b;                      // error
}
a = null;                       // ok
```

If the aggregate is a tuple, 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 tuple, one by one. If the type of the variable is given, it must match the type of the tuple contents. If it is not given, the type of the variable is set to the type of the tuple element, which may change from iteration to iteration. 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 tuple element.

If the tuple is a list of types, then the foreach statement is executed once for each type, and the value is aliased to that type.

```
import std.stdio;
import std.typetuple;   // for TypeTuple

void main()
{
    alias TypeTuple!(int, long, double) TL;

    foreach (T; TL)
    {
        writefln(typeid(T));
    }
}
```

Prints:

```
int
long
double
```

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

Switch Statement

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

SwitchStatement:
        switch (*Expression*) *ScopeStatement*

```
CaseStatement:
        case ExpressionList : Statement

DefaultStatement:
        default: Statement
```

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

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

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

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

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

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

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

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

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

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

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

will set x to 4 if i is 1.

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

```
char[] name;
...
switch (name)
```

```
{
    case "fred":
    case "sally":
        ...
}
```

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

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

Continue Statement

ContinueStatement:
    continue;
    continue *Identifier* **;**

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

```
for (i = 0; i < 10; i++)
{
    if (foo(i))
        continue;
    bar();
}
```

Break Statement

BreakStatement:
    break;
    break *Identifier* **;**

A break exits the enclosing statement. 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.

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

Return Statement

ReturnStatement:
    return;
    return *Expression* **;**

A return exits the current function and supplies its return value. *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, throw statement, or assert(0) expression is required if the function specifies a return type that is not void.

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

Before the function actually returns, any objects with auto storage duration are destroyed, any enclosing finally clauses are executed, any scope(exit) statements are executed, any scope(success) statements are executed, and any enclosing synchronization objects are released.

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

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

```
int foo(int x)
{
    return x + 3;
}
```

Goto Statement

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

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

```
    if (foo)
        goto L1;
    x = 3;
L1:
    x++;
```

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*.

```
switch (x)
{
    case 3:
        goto case;
    case 4:
        goto default;
    case 5:
        goto case 4;
    default:
        x = 4;
        break;
}
```

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

It is illegal for a *GotoStatement* to be used to skip initializations.

With Statement

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

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

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

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

is semantically equivalent to:

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

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

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

```
struct Foo
{
    typedef int Y;
}
```

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

Synchronized Statement

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

SynchronizedStatement:
    synchronized *ScopeStatement*
    synchronized **(** *Expression* **)** *ScopeStatement*

Synchronized allows only one thread at a time to execute *ScopeStatement*.

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

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

Example:

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

This implements a standard critical section.

Try Statement

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

TryStatement:
        try *ScopeStatement* *Catches*
        try *ScopeStatement* *Catches FinallyStatement*
        try *ScopeStatement* *FinallyStatement*

Catches:
        LastCatch
        Catch
        Catch Catches

LastCatch:
        catch *NoScopeNonEmptyStatement*

Catch:
        catch **(** *CatchParameter* **)** *NoScopeNonEmptyStatement*

FinallyStatement:
        finally *NoScopeNonEmptyStatement*

CatchParameter 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 *CatchParameter* 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.

The *FinallyStatement* is always executed, whether the **try** *ScopeStatement* exits with a goto, break, continue, return, exception, or fall-through.

If an exception is raised in the *FinallyStatement* and is not caught before the *FinallyStatement* is executed, the new exception replaces any existing exception:

```
int main()
{
    try
    {
        try
        {
            throw new Exception("first");
        }
        finally
        {
            printf("finally\n");
            throw new Exception("second");
        }
    }
    catch(Exception e)
    {
        printf("catch %.*s\n", e.msg);
    }
    printf("done\n");
    return 0;
}
```

prints:

```
finally
catch second
done
```

A *FinallyStatement* may not exit with a goto, break, continue, or return; nor may it be entered with a goto.

A *FinallyStatement* may not contain any *Catches*. This restriction may be relaxed in future versions.

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.

```
throw new Exception("message");
```

Scope Guard Statement

ScopeGuardStatement:
        scope(exit) *#NonEmptyOrScopeBlockStatement*
        scope(success) *#NonEmptyOrScopeBlockStatement*
        scope(failure) *#NonEmptyOrScopeBlockStatement*

The *ScopeGuardStatement* executes *NonEmptyOrScopeBlockStatement* at the close of the current scope, rather than at the point where the *ScopeGuardStatement* appears. **scope(exit)** executes *NonEmptyOrScopeBlockStatement* when the scope exits normally or when it exits due to exception unwinding. **scope(failure)** executes *NonEmptyOrScopeBlockStatement* when the scope exits due to exception unwinding. **scope(success)** executes *NonEmptyOrScopeBlockStatement* when the scope exits normally.

If there are multiple *ScopeGuardStatement*s in a scope, they are executed in the reverse lexical order in which they appear. If any auto instances are to be destructed upon the close of the scope, they also are interleaved with the *ScopeGuardStatement*s in the reverse lexical order in which they appear.

```
writef("1");
{
    writef("2");
    scope(exit) writef("3");
    scope(exit) writef("4");
    writef("5");
}
writefln();
```

writes:

```
12543
```

```
{
    scope(exit) writef("1");
    scope(success) writef("2");
    scope(exit) writef("3");
    scope(success) writef("4");
}
writefln();
```

writes:

```
4321
```

```
class Foo
{
    this() { writef("0"); }
    ~this() { writef("1"); }
}

try
{
    scope(exit) writef("2");
    scope(success) writef("3");
    auto Foo f = new Foo();
    scope(failure) writef("4");
    throw new Exception("msg");
    scope(exit) writef("5");
    scope(success) writef("6");
```

```
    scope(failure) writef("7");
}
catch (Exception e)
{
}
writefln();
```

writes:

```
0412
```

A **scope(exit)** or **scope(success)** statement may not exit with a throw, goto, break, continue, or return; nor may it be entered with a goto.

Volatile Statement

No code motion occurs across volatile statement boundaries.

```
VolatileStatement:
        volatile Statement
        volatile ;
```

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

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

Asm Statement

Inline assembler is supported with the asm statement:

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

AsmInstructionList:
        AsmInstruction ;
        AsmInstruction ; AsmInstructionList
```

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

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

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

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

For example, for the Intel Pentium:

```
int x = 3;
```

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

Inline assembler can be used to access hardware directly:

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

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

```
version (D_InlineAsm_X86)
{
    asm
    {
        ...
    }
}
else
{
    /* ... some workaround ... */
}
```

Pragma Statement

PragmaStatement:
    Pragma *NoScopeStatement*

Arrays

There are four kinds of arrays:

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

Pointers

```
int* p;
```

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

Static Arrays

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

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

The total size of a static array cannot exceed 16Mb. A dynamic array should be used instead for such large arrays.

A static array with a dimension of 0 is allowed, but no space is allocated for it. It's useful as the last member of a variable length struct, or as the degenerate case of a template expansion.

Dynamic Arrays

```
int[] a;
```

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

Array Declarations

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

Prefix Array Declarations

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

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

Postfix Array Declarations

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

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

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

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

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

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

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

Usage

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

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

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

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

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

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

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

Slicing

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

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

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

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

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

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

are all semantically equivalent.

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

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

Array Copying

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

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

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

Overlapping copies are an error:

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

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

Array Setting

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

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

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

Array Concatenation

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

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

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

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

```
"10" + 3
```

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

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

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

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

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

Pointer Arithmetic

```
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 3rd element
}

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

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

Rectangular Arrays

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

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

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

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

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

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

Array Length

Within the [ ] of a static or a dynamic array, the variable **length** is implicitly declared and set to the length of the array. The symbol **$** can also be so used.

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

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

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

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

Array Properties

Static array properties are:

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

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

Dynamic array properties are:

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

For the **.sort** property to work on arrays of class objects, the class definition must define the function: `int opCmp(Object)`. This is used to determine the ordering of the class objects. Note that the parameter is of type `Object`, not the type of the class.

For the **.sort** property to work on arrays of structs or unions, the struct or union definition must define the function: `int opCmp(S)` or `int opCmp(S*)`. The type `S` is the type of the struct or union. This function will determine the sort ordering.

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[11] and c[1] are also affected

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

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

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

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

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

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

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

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

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

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

Functions as Array Properties

If the first parameter to a function is an array, the function can be called as if it were a property of the array:

```
int[] array;
void foo(int[] a, int x);

foo(array, 3);
array.foo(3);   // means the same thing
```

Array Bounds Checking

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

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

The loop is correctly written:

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

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

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

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

Array Initialization

Default 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.

Void Initialization

Void initialization happens when the *Initializer* for an array is **void**. What it means is that no initialization is done, i.e. the contents of the array will be undefined. This is most useful as an

efficiency optimization. Void initializations are an advanced technique and should only be used when profiling indicates that it matters.

Static Initialization of Static Arrays

Static initalizations are supplied by a list of array element values enclosed in [ ]. The values can be optionally preceded by an index and a :. If an index is not supplied, it is set to the previous index plus 1, or 0 if it is the first value.

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

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

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

int value[Color.max + 1] = [ Color.blue:6, Color.green:2, Color.red:5 ];
```

These arrays are static when they appear in global scope. Otherwise, they need to be marked with **const** or **static** storage classes to make them static arrays.

Special Array Types

Strings

A string is an array of characters. String literals are just an easy way to write character arrays.

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

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

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

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

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

A pointer to a char can be generated:

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

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

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

or use the function `std.string.toStringz`.

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 or a postfix of **c**, **w** or

d can be used:

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

String literals that do not have a postfix character and that have not been cast can be implicitly converted between char[], wchar[], and dchar[] as necessary.

```
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[0];              // w is assigned the carriage return wchar character
d = 'd';                // d is assigned the character 'd'
```

C's 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);
```

or:

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

String literals already have a 0 appended to them, so can be used directly:

```
printf("the string is '%s'\n", "string literal");
```

which is, of course, why the format string to printf was working.

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);
```

The best way is to use std.stdio.writefln, which can handle D strings:

```
import std.stdio;
writefln("the string is '%s'", str);
```

Implicit Conversions

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

- `void*`

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

- `T[]`
- `U[]`

- `void[]`

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

- *U*`[]`
- `void[]`

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

Associative Arrays

Associative arrays have an index that is not necessarily an integer, and can be sparsely populated. The index for an associative array is called the *key*, and its type is called the *KeyType*.

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

```
int[char[]] b;          // associative array b of ints that are
                        // indexed by an array of characters.
                        // The KeyType is char[]
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 remove function:

```
b.remove("hello");
```

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

```
int* p;
p = ("hello" in b);
if (p != null)
        ...
```

*KeyType*s cannot be functions or voids.

If the *KeyType* is a struct type, a default mechanism is used to compute the hash and comparisons of it based on the binary data within the struct value. A custom mechanism can be used by providing the following functions as struct members:

```
uint toHash();
int opCmp(KeyType* s);
```

For example:

```
import std.string;

struct MyString
{
    char[] str;

    uint toHash()
    {   uint hash;
        foreach (char c; s)
            hash = (hash * 9) + c;
        return hash;
    }

    int opCmp(MyString* s)
    {
        return std.string.cmp(this.str, s.str);
    }
}
```

Using Classes as the KeyType

Classes can be used as the *KeyType*. For this to work, the class definition must override the following member functions of class `Object`:

- `hash_t toHash()`
- `int opEquals(Object)`
- `int opCmp(Object)`

Note that the parameter to `opCmp` and `opEquals` is of type `Object`, not the type of the class in which it is defined.

For example:

```
class Foo
{
    int a, b;

    hash_t toHash() { return a + b; }

    int opEquals(Object o)
    {   Foo f = cast(Foo) o;
        return f && a == foo.a && b == foo.b;
    }

    int opCmp(Object o)
    {   Foo f = cast(Foo) o;
        if (!f)
            return -1;
        if (a == foo.a)
            return b - foo.b;
        return a - foo.a;
    }
}
```

The implementation may use either `opEquals` or `opCmp` or both. Care should be taken so that the results of `opEquals` and `opCmp` are consistent with each other when the class objects are the same or not.

Using Structs or Unions as the KeyType

Structs or unions can be used as the *KeyType*. For this to work, the struct or union definition must define the following member functions:

- `hash_t toHash()`
- `int opEquals(S)` or `int opEquals(S*)`
- `int opCmp(S)` or `int opCmp(S*)`

Note that the parameter to `opCmp` and `opEquals` can be either the struct or union type, or a pointer to the struct or union type.

For example:

```
struct S
{
    int a, b;

    hash_t toHash() { return a + b; }

    int opEquals(S s)
    {
        return a == s.a && b == s.b;
    }
```

```
    int opCmp(S* s)
    {
        if (a == s.a)
            return b - s.b;
        return a - s.a;
    }
}
```

The implementation may use either `opEquals` or `opCmp` or both. Care should be taken so that the results of `opEquals` and `opCmp` are consistent with each other when the struct/union objects are the same or not.

Properties

Properties for associative arrays are:

.sizeof	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
import std.stdio;

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

    writefln("   lines   words   bytes file");
    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;

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

        foreach (j, 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]++;
        }
        writefln("%8d%8d%8d %s", l_cnt, w_cnt, c_cnt, args[i]);
        line_total += l_cnt;
        word_total += w_cnt;
        char_total += c_cnt;
    }

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

    writefln("--------------------------------------");
    foreach (word; dictionary.keys.sort)
    {
        writefln("%3d %s", dictionary[word], word);
    }
    return 0;
}
```

Structs & Unions

Whereas classes are reference types, structs are value types. Any C struct can be exactly represented as a D struct. In C++ parlance, a D struct is a POD (Plain Old Data) type, with a trivial constructors and destructors. 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.

A struct is defined to not have an identity; that is, the implementation is free to make bit copies of the struct as convenient.

Struct, Class Comparison Table

Feature	struct	class	C struct	C++ struct	C++ class
value type	X		X	X	X
reference type		X			
data members	X	X	X	X	X
hidden members		X		X	X
static members	X	X		X	X
default member initializers	X	X			
bit fields			X	X	X
non-virtual member functions	X	X		X	X
virtual member functions		X		X	X
constructors		X		X	X
destructors		X		X	X
RAII		X		X	X
operator overloading	X	X		X	X
inheritance		X		X	X
invariants	X	X			
unit tests	X	X			
synchronizable		X			
parameterizable	X	X		X	X
alignment control	X	X			
member protection	X	X		X	X
default public	X	X	X	X	
tag name space			X	X	X
anonymous	X		X	X	X

AggregateDeclaration:
        Tag **{** *DeclDefs* **}**

```
        Tag Identifier StructBody
        Tag Identifier ;

Tag:
        struct
        union

StructBody:
        { }
        { StructBodyDeclarations }

StructBodyDeclarations:
        StructBodyDeclaration
        StructBodyDeclaration StructBodyDeclarations

StructBodyDeclaration:
        Declaration
        StaticConstructor
        StaticDestructor
        Invariant
        UnitTest
        StructAllocator
        StructDeallocator

StructAllocator:
        ClassAllocator

StructDeallocator:
        ClassDeallocator
```

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.

Static Initialization of Structs

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

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

C-style initialization, based on the order of the members in the struct declaration, is also supported:

```
static X q = { 1, 2 };    // q.a = 1, q.b = 2, q.c = 0, q.d = 7
```

Static Initialization of Unions

Unions are initialized explicitly.

```
union U { int a; double b; }
static U u = { b : 5.0 };               // u.b = 5.0
```

Other members of the union that overlay the initializer, but occupy more storage, have the extra storage initialized to zero.

Dynamic Initialization of Structs

Structs can be dynamically initialized from another value of the same type:

```
struct S { int a; }
S t;      // default initialized
t.a = 3;
S s = t;  // s.a is set to 3
```

If `opCall` is overridden for the struct, and the struct is initialized with a value that is of a different type, then the `opCall` operator is called:

```
struct S
{   int a;

    static S opCall(int v)
    {   S s;
        s.a = v;
        return s;
    }

    static S opCall(S v)
    {   S s;
        s.a = v.a + 1;
        return s;
    }
}

S s = 3;        // sets s.a to 3
S t = s;        // sets t.a to 3, S.opCall(s) is not called
```

Struct Properties

```
.sizeof                 Size in bytes of struct
.alignof                Size boundary struct needs to be aligned on
.tupleof                Gets type tuple of fields
```

Struct Field Properties

```
.offsetof               Offset in bytes of field from beginning
                        of struct
```

Classes

The object-oriented features of D all come from classes. The class hierarchy has as its root the class Object. Object defines a minimum level of functionality that each derived class has, and a default implementation for that functionality.

Classes are programmer defined types. Support for classes are what make D an object oriented language, giving it encapsulation, inheritance, and polymorphism. D classes support the single inheritance paradigm, extended by adding support for interfaces. Class objects are instantiated by reference only.

A class can be exported, which means its name and all its non-private members are exposed externally to the DLL or EXE.

A class declaration is defined:

ClassDeclaration:
        class *Identifier BaseClassList*$_{opt}$ *ClassBody*

BaseClassList:
        : *SuperClass*
        : *SuperClass InterfaceClasses*
        : *InterfaceClass*

SuperClass:
        Identifier
        Protection Identifier

InterfaceClasses:
        InterfaceClass
        InterfaceClass InterfaceClasses

InterfaceClass:
        Identifier
        Protection Identifier

Protection:
        private
        package
        public
        export

ClassBody:
        { }
        { *ClassBodyDeclarations* **}**

ClassBodyDeclarations:
        ClassBodyDeclaration
        ClassBodyDeclaration ClassBodyDeclarations

ClassBodyDeclaration:
        Declaration
        Constructor
        Destructor
        StaticConstructor
        StaticDestructor
        Invariant
        UnitTest
        ClassAllocator

ClassDeallocator

Classes consist of:

a super class
interfaces
dynamic fields
static fields
types
functions
    static functions
    dynamic functions
    constructors
    destructors
    static constructors
    static destructors
    invariants
    unit tests
    allocators
    deallocators

A class is defined:

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

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

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

Instead:

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

Fields

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

The D compiler is free to rearrange the order of fields in a class to optimally pack them in an implementation-defined manner. Consider the fields much like the local variables in a function - the compiler assigns some to registers and shuffles others around all to get the optimal stack frame layout. This frees the code designer to organize the fields in a manner that makes the code more readable rather than being forced to organize it according to machine optimization rules. Explicit control of field layout is provided by struct/union types, not classes.

Field Properties

The **.offsetof** property gives the offset in bytes of the field from the beginning of the class instantiation. **.offsetof** can only be applied to fields qualified with the type of the class, not expressions which produce the type of the field itself:

```
class Foo
{
    int x;
}
...
void test(Foo foo)
{
    size_t o;

    o = Foo.x.offsetof;   // yields 8
    o = foo.x.offsetof;   // error, .offsetof an int type
}
```

Class Properties

The **.tupleof** property returns an *ExpressionTuple* of all the fields in the class, excluding the hidden fields and the fields in the base class.

```
class Foo { int x; long y; }
void test(Foo foo)
{
    foo.tupleof[0] = 1;         // set foo.x to 1
    foo.tupleof[1] = 2;         // set foo.y to 2
    foreach (x; foo.tupleof)
        writef(x);              // prints 12
}
```

Super Class

All classes inherit from a super class. If one is not specified, it inherits from Object. Object forms the root of the D class inheritance hierarchy.

Constructors

Constructor:
        this *Parameters* *FunctionBody*

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() { (a) ? 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[] (the array of pointers to virtual functions) is assigned. This ensures that constructors are passed fully formed objects for which virtual functions can be called. This operation is equivalent to doing a memory copy 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() *FunctionBody*

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

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

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

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

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

The destructor for the super class automatically gets called when the destructor ends. There is no

way to call the super destructor explicitly.

When the garbage collector calls a destructor for an object of a class that has members that are references to garbage collected objects, those references are no longer valid. This means that destructors cannot reference sub objects. This rule does not apply to auto objects or objects deleted with the *DeleteExpression*.

The garbage collector is not guaranteed to run the destructor for all unreferenced objects. Furthermore, the order in which the garbage collector calls destructors for unreference objects is not specified.

Objects referenced from the data segment never get collected by the gc.

Static Constructors

StaticConstructor:
        static this() *FunctionBody*

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

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

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

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

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

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

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

`static this()` is called by the startup code before `main()` is called. If it returns normally (does not throw an exception), the static destructor is added to the list of functions to be called on program termination. Static constructors have empty parameter lists.

Static constructors within a module are executed in the lexical order in which they appear. All the static constructors for modules that are directly or indirectly imported are executed before the static constructors for the importer.

The **static** in the static constructor declaration is not an attribute, it must appear immediately before the **this**:

```
class Foo
{
    static this() { ... }       // a static constructor
    static private this() { ... } // not a static constructor
    static
    {
        this() { ... }          // not a static constructor
    }
    static:
        this() { ... }          // not a static constructor
}
```

Static Destructor

StaticDestructor:
        static ~this() *FunctionBody*

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

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

A static destructor gets called on program termination, but only if the static constructor completed successfully. Static destructors have empty parameter lists. Static destructors get called in the reverse order that the static constructors were called in.

The **static** in the static destructor declaration is not an attribute, it must appear immediately before the **~this**:

```
class Foo
{
    static ~this() { ... }      // a static destructor
    static private ~this() { ... } // not a static destructor
    static
    {
        ~this() { ... }         // not a static destructor
    }
    static:
        ~this() { ... }         // not a static destructor
}
```

Class Invariants

ClassInvariant:
        invariant *BlockStatement*

Class invariants are used to specify characteristics of a class that always must be true (except while

executing a member function). For example, a class representing a date might have an invariant that the day must be 1..31 and the hour must be 0..23:

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

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

The class invariant is a contract saying that the asserts must hold true. The invariant is checked when a class constructor completes, at the start of the class destructor, before a public or exported member is run, and after a public or exported function finishes.

The code in the invariant may not call any public non-static members of the class, either directly or indirectly. Doing so will result in a stack overflow, as the invariant will wind up being called in an infinitely recursive manner.

```
class Foo
{
    public void f() { }
    private void g() { }

    invariant
    {
        f();  // error, cannot call public member function from invariant
        g();  // ok, g() is not public
    }
}
```

The invariant can be checked when a class object is the argument to an `assert()` expression, as:

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

Invariants contain assert expressions, and so when they fail, they throw a `AssertErrors`. 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 *FunctionBody*

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.

Classes can have a special member function called:

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

A compiler switch, such as **-unittest** for **dmd**, will cause the unittest test code to be compiled and incorporated into the resulting executable. The unittest code gets run after static initialization is run and before the `main()` function is called.

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
    {
        Sum sum = new Sum;
        assert(sum.add(3,4) == 7);
        assert(sum.add(-2,0) == -2);
    }
}
```

Class Allocators

ClassAllocator:
        new *Parameters* *FunctionBody*

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.sizeof,1,2)
```

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

The class allocator is not called if the instance is created on the stack.

See also Explicit Class Instance Allocation.

Class Deallocators

ClassDeallocator:
        delete *Parameters* *FunctionBody*

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.

The class allocator is not called if the instance is created on the stack.

See also Explicit Class Instance Allocation.

Scope Classes

A scope class is a class with the **scope** attribute, as in:

```
scope class Foo { ... }
```

The scope characteristic is inherited, so if any classes derived from a scope class are also scope.

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

```
scope class Foo { ... }

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

When an scope 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.

Nested Classes

A *nested class* is a class that is declared inside the scope of a function or another class. A nested class has access to the variables and other symbols of the classes and functions it is nested inside:

```
class Outer
{
    int m;

    class Inner
    {
        int foo()
        {
            return m;   // Ok to access member of Outer
        }
    }
}

void func()
{   int m;

    class Inner
    {
        int foo()
        {
            return m;   // Ok to access local variable m of func()
        }
    }
}
```

If a nested class has the **static** attribute, then it can not access variables of the enclosing scope that are local to the stack or need a **this**:

```
class Outer
{
    int m;
    static int n;

    static class Inner
    {
        int foo()
        {
            return m;   // Error, Inner is static and m needs a this
            return n;   // Ok, n is static
        }
    }
}

void func()
{   int m;
    static int n;

    static class Inner
    {
        int foo()
        {
            return m;   // Error, Inner is static and m is local to the stack
            return n;   // Ok, n is static
```

```
        }
    }
}
```

Non-static nested classes work by containing an extra hidden member (called the context pointer) that is the frame pointer of the enclosing function if it is nested inside a function, or the **this** of the enclosing class's instance if it is nested inside a class.

When a non-static nested class is instantiated, the context pointer is assigned before the class's constructor is called, therefore the constructor has full access to the enclosing variables. A non-static nested class can only be instantiated when the necessary context pointer information is available:

```
class Outer
{
    class Inner { }

    static class SInner { }
}

void func()
{
    class Nested { }

    Outer o = new Outer;         // Ok
    Outer.Inner oi = new Outer.Inner;   // Error, no 'this' for Outer
    Outer.SInner os = new Outer.SInner; // Ok

    Nested n = new Nested;       // Ok
}
```

While a non-static nested class can access the stack variables of its enclosing function, that access becomes invalid once the enclosing function exits:

```
class Base
{
    int foo() { return 1; }
}

Base func()
{   int m = 3;

    class Nested : Base
    {
        int foo() { return m; }
    }

    Base b = new Nested;

    assert(b.foo() == 3);        // Ok, func() is still active
    return b;
}

int test()
{
    Base b = func();
    return b.foo();              // Error, func().m is undefined
}
```

If this kind of functionality is needed, the way to make it work is to make copies of the needed

variables within the nested class's constructor:

```
class Base
{
    int foo() { return 1; }
}

Base func()
{   int m = 3;

    class Nested : Base
    {   int m_;

        this() { m_ = m; }
        int foo() { return m_; }
    }

    Base b = new Nested;

    assert(b.foo() == 3);       // Ok, func() is still active
    return b;
}

int test()
{
    Base b = func();
    return b.foo();             // Ok, using cached copy of func().m
}
```

A *this* can be supplied to the creation of an inner class instance by prefixing it to the *NewExpression*:

```
class Outer
{   int a;

    class Inner
    {
        int foo()
        {
            return a;
        }
    }
}

int bar()
{
    Outer o = new Outer;
    o.a = 3;
    Outer.Inner oi = o.new Inner;
    return oi.foo();    // returns 3
}
```

Here **o** supplies the *this* to the outer class instance of **Outer**.

The property **.outer** used in a nested class gives the **this** pointer to its enclosing class. If the enclosing context is not a class, the **.outer** will give the pointer to it as a **void*** type.

```
class Outer
{
    class Inner
    {
```

```
        Outer foo()
        {
            return this.outer;
        }
    }

    void bar()
    {
        Inner i = new Inner;
        assert(this == i.foo());
    }
}

void test()
{
    Outer o = new Outer;
    o.bar();
}
```

Anonymous Nested Classes

An anonymous nested class is both defined and instantiated with a *NewAnonClassExpression*:

NewAnonClassExpression:
    new (*ArgumentList*)$_{opt}$ **class** (*ArgumentList*)$_{opt}$ *SuperClass*$_{opt}$
InterfaceClasses$_{opt}$ *ClassBody*

which is equivalent to:

```
class Identifier : SuperClass InterfaceClasses
        ClassBody

new (ArgumentList) Identifier (ArgumentList);
```

where *Identifier* is the name generated for the anonymous nested class.

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 = cast(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 = cast(D) b;
d.foo();                // returns 2
A a = cast(A) b;
D d2 = cast(D) a;
d2.foo();               // returns 2, even though it is A's D, not B's D
```

A reimplemented interface must implement all the interface functions, it does not inherit them from a super class:

```
interface D
```

```
{
    int foo();
}

class A : D
{
    int foo() { return 1; }
}

class B : A, D
{
}               // error, no foo() for interface D
```

COM Interfaces

A variant on interfaces is the COM interface. A COM interface is designed to map directly onto a Windows COM object. Any COM object can be represented by a COM interface, and any D object with a COM interface can be used by external COM clients.

A COM interface is defined as one that derives from the interface `std.c.windows.com.IUnknown`. A COM interface differs from a regular D interface in that:

- It derives from the interface `std.c.windows.com.IUnknown`.
- It cannot be the argument of a *DeleteExpression*.
- References cannot be upcast to the enclosing class object, nor can they be downcast to a derived interface. To accomplish this, an appropriate `QueryInterface()` would have to be implemented for that interface in standard COM fashion.

Enums - Enumerated Types

```
EnumDeclaration:
        enum Identifier EnumBody
        enum EnumBody
        enum Identifier : EnumBaseType EnumBody
        enum : EnumBaseType EnumBody

EnumBaseType:
        Type

EnumBody:
        ;
        { EnumMembers }

EnumMembers:
        EnumMember
        EnumMember ,
        EnumMember , EnumMembers

EnumMember:
        Identifier
        Identifier = AssignExpression
```

Enums are used to define a group of related integral constants.

If the enum *Identifier* is present, the *EnumMembers* are declared in the scope of the enum *Identifier*. The enum *Identifier* declares a new type.

If the enum *Identifier* is not present, then the enum is an *anonymous enum*, and the *EnumMembers* are declared in the scope the *EnumDeclaration* appears in. No new type is created; the *EnumMembers* have the type of the *EnumBaseType*.

The *EnumBaseType* is the underlying type of the enum. It must be an integral type. If omitted, it defaults to **int**.

```
enum { A, B, C }        // anonymous enum
```

Defines the constants A=0, B=1, C=2 in a manner equivalent to:

```
const int A = 0;
const int B = 1;
const int C = 2;
```

Whereas:

```
enum X { A, B, C }      // named enum
```

Define a new type X which has values X.A=0, X.B=1, X.C=2

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

Enums must have at least one member.

If an *Expression* is supplied for an enum member, the value of the member is set to the result of the *Expression*. The *Expression* must be resolvable at compile time. Subsequent enum members with no *Expression* are set to the value of the previous member plus one:

```
enum { A, B = 5+7, C, D = 8, E }
```

Sets A=0, B=12, C=13, D=8, and E=9.

Enum Properties

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

For example:

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

Initialization of Enums

In the absence of an explicit initializer, an enum variable is initialized to the first enum value.

```
enum X { A=3, B, C }
X x;            // x is initialized to 3
```

Functions

FunctionBody:
        BlockStatement
        BodyStatement
        InStatement BodyStatement
        OutStatement BodyStatement
        InStatement OutStatement BodyStatement
        OutStatement InStatement BodyStatement

InStatement:
        in *BlockStatement*

OutStatement:
        out *BlockStatement*
        out (*Identifier*) *BlockStatement*

BodyStatement:
        body *BlockStatement*

Virtual Functions

Virtual functions are functions that are called indirectly through a function pointer table, called a vtbl[], rather than directly. All non-static non-private non-template member functions are virtual. This may sound inefficient, but since the D compiler knows all of the class hierarchy when generating code, all functions that are not overridden can be optimized to be non-virtual. In fact, since C++ programmers tend to "when in doubt, make it virtual", the D way of "make it virtual unless we can prove it can be made non-virtual" results, on average, in many 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.

Member template functions cannot be virtual, and hence cannot be overridden.

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

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

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

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

```
    a.abc();    // calls A.abc
}

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

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

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

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

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

Function Inheritance and Overriding

A functions in a derived class with the same name and parameter types as a function in a base class overrides that function:

```
class A
{
    int foo(int x) { ... }
}

class B : A
{
    override int foo(int x) { ... }
}

void test()
{
    B b = new B();
    bar(b);
}

void bar(A a)
{
    a.foo(1);   // calls B.foo(int)
}
```

However, when doing overload resolution, the functions in the base class are not considered:

```
class A
{
    int foo(int x) { ... }
    int foo(long y) { ... }
}

class B : A
{
```

```
    override int foo(long x) { ... }
}

void test()
{
    B b = new B();
    b.foo(1);           // calls B.foo(long), since A.foo(int) not considered
    A a = b;
    a.foo(1);           // calls A.foo(int)
}
```

To consider the base class's functions in the overload resolution process, use an *AliasDeclaration*:

```
class A
{
    int foo(int x) { ... }
    int foo(long y) { ... }
}

class B : A
{
    alias A.foo foo;
    override int foo(long x) { ... }
}

void test()
{
    B b = new B();
    bar(b);
}

void bar(A a)
{
    a.foo(1);           // calls A.foo(int)
    B b = new B();
    b.foo(1);           // calls A.foo(int)
}
```

A function parameter's default value is not inherited:

```
class A
{
    void foo(int x = 5) { ... }
}

class B : A
{
    void foo(int x = 7) { ... }
}

class C : B
{
    void foo(int x) { ... }
}


void test()
{
    A a = new A();
    a.foo();            // calls A.foo(5)
```

```
    B b = new B();
    b.foo();            // calls B.foo(7)

    C c = new C();
    c.foo();            // error, need an argument for C.foo
}
```

Inline Functions

There is no inline keyword. The compiler makes the decision whether to inline a function or not, analogously to the register keyword no longer being relevant to a compiler's decisions on enregistering variables. (There is no register keyword either.)

Function Overloading

In C++, there are many complex levels of function overloading, with some defined as "better" matches than others. If the code designer takes advantage of the more subtle behaviors of overload function selection, the code can become difficult to maintain. Not only will it take a C++ expert to understand why one function is selected over another, but different C++ compilers can implement this tricky feature differently, producing subtly disastrous results.

In D, function overloading is simple. It matches exactly, it matches with implicit conversions, or it does not match. If there is more than one match, it is an error.

Functions defined with non-D linkage cannot be overloaded.

Function Parameters

Parameters are **in**, **out**, **inout** or **lazy**. **in** is the default; the others 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 x)
{
    // x is set to 0 at start of foo()
}

int a = 3;
foo(a);
// a is now 0


void abc(out int x)
```

```
{
    x = 2;
}

int y = 3;
abc(y);
// y is now 2


void def(inout int x)
{
    x += 1;
}

int z = 3;
def(z);
// z is now 4
```

For dynamic array and object parameters, which are passed by reference, in/out/inout apply only to the reference and not the contents.

Lazy arguments are evaluated not when the function is called, but when the parameter is evaluated within the function. Hence, a lazy argument can be executed 0 or more times. A lazy parameter cannot be an lvalue.

```
void dotimes(int n, lazy void exp)
{
    while (n--)
        exp();
}

void test()
{   int x;
    dotimes(3, writefln(x++));
}
```

prints to the console:

```
0
1
2
```

A lazy parameter of type `void` can accept an argument of any type.

Variadic Functions

Functions taking a variable number of arguments are called variadic functions. A variadic function can take one of three forms:

1. C-style variadic functions
2. Variadic functions with type info
3. Typesafe variadic functions

C-style Variadic Functions

A C-style variadic function is declared as taking a parameter of ... after the required function parameters. It has non-D linkage, such as `extern (C)`:

```
extern (C) int foo(int x, int y, ...);

foo(3, 4);      // ok
foo(3, 4, 6.8); // ok, one variadic argument
foo(2);         // error, y is a required argument
```

There must be at least one non-variadic parameter declared.

```
extern (C) int def(...); // error, must have at least one parameter
```

C-style variadic functions match the C calling convention for variadic functions, and is most useful for calling C library functions like `printf`. The implementiations of these variadic functions have a special local variable declared for them, **_argptr**, which is a `void*` pointer to the first of the variadic arguments. To access the arguments, **_argptr** must be cast to a pointer to the expected argument type:

```
foo(3, 4, 5);   // first variadic argument is 5

int foo(int x, int y, ...)
{   int z;

    z = *cast(int*)_argptr;     // z is set to 5
}
```

To protect against the vagaries of stack layouts on different CPU architectures, use **std.c.stdarg** to access the variadic arguments:

```
import std.c.stdarg;
```

D-style Variadic Functions

Variadic functions with argument and type info are declared as taking a parameter of ... after the required function parameters. It has D linkage, and need not have any non-variadic parameters declared:

```
int abc(char c, ...);   // one required parameter: c
int def(...);           // ok
```

These variadic functions have a special local variable declared for them, **_argptr**, which is a `void*` pointer to the first of the variadic arguments. To access the arguments, **_argptr** must be cast to a pointer to the expected argument type:

```
foo(3, 4, 5);   // first variadic argument is 5

int foo(int x, int y, ...)
{   int z;

    z = *cast(int*)_argptr;     // z is set to 5
}
```

An additional hidden argument with the name **_arguments** and type `TypeInfo[]` is passed to the function. **_arguments** gives the number of arguments and the type of each, enabling the creation of typesafe variadic functions.

```
class Foo { int x = 3; }
class Bar { long y = 4; }

void printargs(int x, ...)
```

```
{
    printf("%d arguments\n", _arguments.length);
    for (int i = 0; i < _arguments.length; i++)
    {   _arguments[i].print();

        if (_arguments[i] == typeid(int))
        {
            int j = *cast(int *)_argptr;
            _argptr += int.sizeof;
            printf("\t%d\n", j);
        }
        else if (_arguments[i] == typeid(long))
        {
            long j = *cast(long *)_argptr;
            _argptr += long.sizeof;
            printf("\t%lld\n", j);
        }
        else if (_arguments[i] == typeid(double))
        {
            double d = *cast(double *)_argptr;
            _argptr += double.sizeof;
            printf("\t%g\n", d);
        }
        else if (_arguments[i] == typeid(Foo))
        {
            Foo f = *cast(Foo*)_argptr;
            _argptr += Foo.sizeof;
            printf("\t%p\n", f);
        }
        else if (_arguments[i] == typeid(Bar))
        {
            Bar b = *cast(Bar*)_argptr;
            _argptr += Bar.sizeof;
            printf("\t%p\n", b);
        }
        else
            assert(0);
    }
}

void main()
{
    Foo f = new Foo();
    Bar b = new Bar();

    printf("%p\n", f);
    printargs(1, 2, 3L, 4.5, f, b);
}
```

which prints:

```
00870FE0
5 arguments
int
        2
long
        3
double
        4.5
Foo
        00870FE0
```

```
Bar
        00870FD0
```

To protect against the vagaries of stack layouts on different CPU architectures, use **std.stdarg** to access the variadic arguments:

```
import std.stdarg;

void foo(int x, ...)
{
    printf("%d arguments\n", _arguments.length);
    for (int i = 0; i < _arguments.length; i++)
    {   _arguments[i].print();

        if (_arguments[i] == typeid(int))
        {
            int j = va_arg!(int)(_argptr);
            printf("\t%d\n", j);
        }
        else if (_arguments[i] == typeid(long))
        {
            long j = va_arg!(long)(_argptr);
            printf("\t%lld\n", j);
        }
        else if (_arguments[i] == typeid(double))
        {
            double d = va_arg!(double)(_argptr);
            printf("\t%g\n", d);
        }
        else if (_arguments[i] == typeid(FOO))
        {
            FOO f = va_arg!(FOO)(_argptr);
            printf("\t%p\n", f);
        }
        else
            assert(0);
    }
}
```

Typesafe Variadic Functions

Typesafe variadic functions are used when the variable argument portion of the arguments are used to construct an array or class object.

For arrays:

```
int test()
{
    return sum(1, 2, 3) + sum(); // returns 6+0
}

int func()
{
    int[3] ii = [4, 5, 6];
    return sum(ii);             // returns 15
}

int sum(int[] ar ...)
{
    int s;
```

```
    foreach (int x; ar)
        s += x;
    return s;
}
```

For static arrays:

```
int test()
{
    return sum(2, 3);   // error, need 3 values for array
    return sum(1, 2, 3); // returns 6
}

int func()
{
    int[3] ii = [4, 5, 6];
    int[] jj = ii;
    return sum(ii);             // returns 15
    return sum(jj);             // error, type mismatch
}

int sum(int[3] ar ...)
{
    int s;
    foreach (int x; ar)
        s += x;
    return s;
}
```

For class objects:

```
class Foo
{
    int x;
    char[] s;

    this(int x, char[] s)
    {
        this.x = x;
        this.s = s;
    }
}

void test(int x, Foo f ...);

...

Foo g = new Foo(3, "abc");
test(1, g);             // ok, since g is an instance of Foo
test(1, 4, "def");      // ok
test(1, 5);             // error, no matching constructor for Foo
```

An implementation may construct the object or array instance on the stack. Therefore, it is an error to refer to that instance after the variadic function has returned:

```
Foo test(Foo f ...)
{
    return f;   // error, f instance contents invalid after return
}

int[] test(int[] a ...)
```

```
{
    return a;           // error, array contents invalid after return
    return a[0..1];     // error, array contents invalid after return
    return a.dup;       // ok, since copy is made
}
```

For other types, the argument is built with itself, as in:

```
int test(int i ...)
{
    return i;
}

...
test(3);        // returns 3
test(3, 4);     // error, too many arguments
int[] x;
test(x);        // error, type mismatch
```

Lazy Variadic Functions

If the variadic parameter is an array of delegates with no parameters:

```
void foo(int delegate()[] dgs ...);
```

Then each of the arguments whose type does not match that of the delegate is converted to a delegate.

```
int delegate() dg;
foo(1, 3+x, dg, cast(int delegate())null);
```

is the same as:

```
foo( { return 1; }, { return 3+x; }, dg, null );
```

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, are just a source of confusion for maintenance programmers.

It is an error to declare a local variable that hides another local variable in the same function:

```
void func(int x)
{   int x;              error, hides previous definition of x
     double y;
     ...
     {   char y;        error, hides previous definition of y
          int z;
     }
     {   wchar z;       legal, previous z is out of scope
     }
}
```

While this might look unreasonable, in practice whenever this is done it either is a bug or at least

looks like a bug.

It is an error to return the address of or a reference to a local variable.

It is an error to have a local variable and a label with the same name.

Nested Functions

Functions may be nested within other functions:

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

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

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

Nested functions can be accessed only if the name is in scope.

```
void foo()
{
   void A()
   {
     B();   // error, B() is forward referenced
     C();   // error, C undefined
   }
   void B()
   {
       A();     // ok, in scope
       void C()
       {
           void D()
           {
               A();      // ok
               B();      // ok
               C();      // ok
               D();      // ok
           }
       }
   }
   A(); // ok
   B(); // ok
   C(); // error, C undefined
}
```

and:

```
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;
        }
        return 0;
    }
}
```

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()
            }

            return 0;
        }
    }
}
```

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*.

The **.ptr** property of a delegate will return the *frame pointer* value as a `void*`.

The **.funcptr** property of a delegate will return the *function pointer* value as a function type.

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

Anonymous Functions and Anonymous Delegates

See Function Literals.

main() Function

For console programs, `main()` serves as the entry point. It gets called after all the module initializers are run, and after any unittests are run. After it returns, all the module destructors are run. `main()` must be declared using one of the following forms:

```
void main() { ... }
void main(char[][] args) { ... }
int main() { ... }
int main(char[][] args) { ... }
```

Operator Overloading

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

Unary Operator Overloading

Overloadable Unary Operators

op	*opfunc*
-*e*	`opNeg`
+*e*	`opPos`
~*e*	`opCom`
e++	`opPostInc`
e--	`opPostDec`
cast(*type*) *e*	`opCast`

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

```
op a
```

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

```
a.opfunc()
```

Overloading ++*e* and --*e*

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

Examples

```
1. class A { int opNeg(); }
   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
```

Overloading cast(*type*)*e*

The member function *e*.**opCast()** is called, and the return value of **opCast()** is implicitly converted to *type*. Since functions cannot be overloaded based on return value, there can be only one **opCast** per struct or class. Overloading the cast operator does not affect implicit casts, it only applies to explicit casts.

```
struct A
```

```
{
    int opCast() { return 28; }
}

void test()
{
    A a;

    long i = cast(long)a;   // i is set to 28L
    void* p = cast(void*)a; // error, cannot implicitly
                            // convert int to void*
    int j = a;              // error, cannot implicitly convert
                            // A to int
}
```

Binary Operator Overloading

Overloadable Binary Operators

op	commutative?	*opfunc*	*opfunc_r*
+	yes	opAdd	opAdd_r
-	no	opSub	opSub_r
*	yes	opMul	opMul_r
/	no	opDiv	opDiv_r
%	no	opMod	opMod_r
&	yes	opAnd	opAnd_r
|	yes	opOr	opOr_r
^	yes	opXor	opXor_r
<<	no	opShl	opShl_r
>>	no	opShr	opShr_r
>>>	no	opUShr	opUShr_r
~	no	opCat	opCat_r
==	yes	opEquals	-
!=	yes	opEquals	-
<	yes	opCmp	-
<=	yes	opCmp	-
>	yes	opCmp	-
>=	yes	opCmp	-
=	no	opAssign	-
+=	no	opAddAssign	-
-=	no	opSubAssign	-

*=	no	opMulAssign	-
/=	no	opDivAssign	-
%=	no	opModAssign	-
&=	no	opAndAssign	-
|=	no	opOrAssign	-
^=	no	opXorAssign	-
<<=	no	opShlAssign	-
>>=	no	opShrAssign	-
>>>=	no	opUShrAssign	-
~=	no	opCatAssign	-
in	no	opIn	opIn_r

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

```
a op b
```

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

1. The expression is rewritten as both:

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

   If any a.opfunc or b.opfunc_r functions exist, then overloading is applied across all of them and the best match is used. If either exist, and there is no argument match, then it is an error.

2. If the operator is commutative, then the following forms are tried:

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

3. 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)
   1 + a;  // equivalent to a.opAdd(1)

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

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

```
4. class A { int opAdd(B b);  int opAdd_r(B b); }
   class B { }
   A a;
   B b;
   a + b;  // equivalent to a.opAdd(b)
   b + a;  // equivalent to a.opAdd_r(b)

5. class A { int opAdd(B b);  int opAdd_r(B b); }
   class B { int opAdd_r(A a); }
   A a;
   B b;
   a + b;  // ambiguous: a.opAdd(b) or b.opAdd_r(a)
   b + a;  // equivalent to a.opAdd_r(b)
```

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 a default **opEquals**(). But every class definition which will be using == or != should expect to need to override opEquals. The parameter to the overriding function must be of type Object, not the type for the class.

Structs and unions (hereafter just called structs) can provide a member function:

```
int opEquals(S s)
```

or:

```
int opEquals(S* s)
```

where S is the struct name, to define how equality is determined.

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.

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

```
if (a is null)
```

and not as:

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

The latter is converted to:

```
if (a.opEquals(null))
```

which will fail if **opEquals**() is a virtual function.

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.

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;
    }
}
```

The parameter to **opEquals** and **opCmp** for class definitions must be of type Object, rather than the type of the particular class, in order to override the Object.**opEquals** and Object.**opCmp** functions properly.

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 = f.opCall(3,4,5);
}
```

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

Array Operator Overloading

Overloading Indexing *a*[*i*]

The array index operator, [], can be overloaded by declaring a function named **opIndex** with one or more parameters. Assignment to an array can be overloaded with a function named **opIndexAssign**

with two or more parameters. The first parameter is the rvalue of the assignment expression.

```
struct A
{
    int opIndex(size_t i1, size_t i2, size_t i3);
    int opIndexAssign(int value, size_t i1, size_t i2);
}

void test()
{   A a;
    int i;

    i = a[5,6,7];       // same as i = a.opIndex(5,6,7);
    a[i,3] = 7;         // same as a.opIndexAssign(7,i,3);
}
```

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

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

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

Overloading the slicing operator means overloading expressions like `a[]` and `a[i .. j]`. This can be done by declaring a function named **opSlice**. Assignment to a slice can be done by declaring **opSliceAssign**.

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

    int opSliceAssign(int v);                       // overloads a[] = v
    int opSliceAssign(int v, size_t x, size_t y); // overloads a[i .. j] = v
}

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

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

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

Assignment Operator Overloading

The assignment operator = can be overloaded if the lvalue is a struct or class aggregate, and `opAssign` is a member function of that aggregate.

The assignment operator cannot be overloaded for rvalues that can be implicitly cast to the lvalue type. Furthermore, the following parameter signatures for `opAssign` are not allowed:

```
opAssign(...)
opAssign(T)
opAssign(T, ...)
opAssign(T ...)
```

```
opAssign(T, U = defaultValue, etc.)
```

where *T* is the same type as the aggregate type *A*, is implicitly convertible to *A*, or if *A* is a struct and *T* is a pointer to a type that is implicitly convertible to *A*.

Future Directions

The operators !, ., &&, ||, ?:, and a few others will likely never be overloadable.

Templates

> I think that I can safely say that nobody understands template mechanics. -- Richard Deyman

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:
        TemplateTypeParameter
        TemplateValueParameter
        TemplateAliasParameter
        TemplateTupleParameter

TemplateTypeParameter:
        Identifier
        Identifier TemplateTypeParameterSpecialization
        Identifier TemplateTypeParameterDefault
        Identifier TemplateTypeParameterSpecialization
TemplateTypeParameterDefault

TemplateTypeParameterSpecialization:
         : Type

TemplateTypeParameterDefault:
         = Type

TemplateValueParameter:
        Declaration
        Declaration TemplateValueParameterSpecialization
        Declaration TemplateValueParameterDefault
        Declaration TemplateValueParameterSpecialization
TemplateValueParameterDefault

TemplateValueParameterSpecialization:
         : ConditionalExpression

TemplateValueParameterDefault:
         = ConditionalExpression

TemplateAliasParameter:
        alias Identifier
        alias Identifier TemplateAliasParameterSpecialization
        alias Identifier TemplateAliasParameterDefault
        alias Identifier TemplateAliasParameterSpecialization
TemplateAliasParameterDefault

TemplateAliasParameterSpecialization:
```

```
        : Type

TemplateAliasParameterDefault:
        = Type

TemplateTupleParameter:
        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, symbols, or tuples. Types can be any type. Value parameters must be of an integral type, floating point type, or string type and specializations for them must resolve to an integral constant, floating point constant, null, or a string literal. Symbols can be any non-local symbol. Tuples are a sequence of 0 or more types, values or symbols.

Template parameter specializations constrain the values or types the *TemplateParameter* can accept.

Template parameter defaults are the value or type to use for the *TemplateParameter* in case one is not supplied.

Explicit Template Instantiation

Templates are explicitly 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 *TemplateArgumentList*, before implicit conversions, 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.

Even if template arguments are implicitly converted to the same template parameter type, they still refer to different instances:

```
struct TFoo(int x) { }
static assert(is(TFoo!(3) == TFoo!(2 + 1)));   // 3 and 2+1 are both 3 of type
int
static assert(!is(TFoo!(3) == TFoo!(3u)));     // 3u and 3 are different types
```

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

For example, a simple generic copy template would be:

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

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

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

Instantiation Scope

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

For example:

module a

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

module b

```
import a;

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

and:

module a

```
template TFoo(T) { void bar() { func(1); } }
```

```
void func(double d) { }
```

module b

```
import a;

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

TemplateParameter specializations and default values are evaluated in the scope of the *TemplateDeclaration*.

Argument Deduction

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

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

1. If there is no type specialization for the parameter, the type of the parameter is set to the template argument.
2. If the type specialization is dependent on a type parameter, the type of that parameter is set to be the corresponding part of the type argument.
3. If after all the type arguments are examined there are any type parameters left with no type assigned, they are assigned types corresponding to the template argument in the same position in the *TemplateArgumentList*.
4. If applying the above rules does not result in exactly one type for each template parameter, then it is an error.

For example:

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

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

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

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

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

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

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

template TBar(T : U*, U : A) { }
```

```
alias TBar!(B*, B) Foo5;        // (2) T is B*
                                // (3) U is B
```

Value Parameters

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

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

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

Specialization

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

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

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

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

Alias Parameters

Alias parameters enable templates to be parameterized with any type of D symbol, including global names, local names, type names, module names, template names, and template instance names. 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
```

```
    static int y;
    alias Foo!(y) abc;
    *abc.p = 3;         // set y to 3
}
```

- Type names

```
class Foo
{
    static int p;
}

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

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

- Module names

```
import std.string;

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

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

- Template names

```
int x;

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

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

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

- Template alias names

```
int x;
```

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

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

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

Template Parameter Default Values

Trailing template parameters can be given default values:

```
template Foo(T, U = int) { ... }
Foo!(uint,long); // instantiate Foo with T as uint, and U as long
Foo!(uint);      // instantiate Foo with T as uint, and U as int

template Foo(T, U = T*) { ... }
Foo!(uint);      // instantiate Foo with T as uint, and U as uint*
```

Implicit Template Properties

If a template has exactly one member in it, and the name of that member is the same as the template name, that member is assumed to be referred to in a template instantiation:

```
template Foo(T)
{
    T Foo;      // declare variable Foo of type T
}

void test()
{
    Foo!(int) = 6;      // instead of Foo!(int).Foo
}
```

Tuple Parameters

If the last template parameter in the *TemplateParameterList* is declared as a *TemplateTupleParameter*, it is a match with any trailing template arguments. The sequence of arguments form a *Tuple*. A *Tuple* is not a type, an expression, or a symbol. It is a sequence of any mix of types, expressions or symbols.

A *Tuple* whose elements consist entirely of types is called a *TypeTuple*. A *Tuple* whose elements consist entirely of expressions is called an *ExpressionTuple*.

A *Tuple* can be used as an argument list to instantiate another template, or as the list of parameters for a function.

```
template Print(A ...)
{
    void print()
    {
        writefln("args are ", A);
    }
}

template Write(A ...)
{
    void write(A a)     // A is a TypeTuple
                        // a is an ExpressionTuple
    {
        writefln("args are ", a);
    }
}

void main()
{
    Print!(1,'a',6.8).print();                    // prints: args are 1a6.8
    Write!(int, char, double).write(1, 'a', 6.8); // prints: args are 1a6.8
}
```

Template tuples can be deduced from the types of the trailing parameters of an implicitly instantiated function template:

```
template Foo(T, R...)
{
    void Foo(T t, R r)
    {
        writefln(t);
        static if (r.length)    // if more arguments
            Foo(r);             // do the rest of the arguments
    }
}

void main()
{
    Foo(1, 'a', 6.8);
}
```

prints:

```
1
a
6.8
```

The tuple can also be deduced from the type of a delegate or function parameter list passed as a function argument:

```
/* R is return type
 * A is first argument type
 * U is TypeTuple of rest of argument types
 */
R delegate(U) Curry(R, A, U...)(R delegate(A, U) dg, A arg)
{
    struct Foo
    {
        typeof(dg) dg_m;
        typeof(arg) arg_m;
```

```
        R bar(U u)
        {
            return dg_m(arg_m, u);
        }
    }

    Foo* f = new Foo;
    f.dg_m = dg;
    f.arg_m = arg;
    return &f.bar;
}

void main()
{
    int plus(int x, int y, int z)
    {
        return x + y + z;
    }

    auto plus_two = Curry(&plus, 2);
    printf("%d\n", plus_two(6, 8));     // prints 16
}
```

The number of elements in a *Tuple* can be retrieved with the **.length** property. The *n*th element can be retrieved by indexing the *Tuple* with [*n*], and sub tuples can be created with the slicing syntax.

*Tuple*s are static compile time entities, there is no way to dynamically change, add, or remove elements.

If both a template with a tuple parameter and a template without a tuple parameter exactly match a template instantiation, the template without a *TemplateTupleParameter* is selected.

Class Templates

ClassTemplateDeclaration:
    class *Identifier* **(** *TemplateParameterList* **)** [*SuperClass* {, *InterfaceClass* }]
ClassBody

If a template declares exactly one member, and that member is a class with the same name as the template:

```
template Bar(T)
{
    class Bar
    {
        T member;
    }
}
```

then the semantic equivalent, called a *ClassTemplateDeclaration* can be written as:

```
class Bar(T)
{
    T member;
}
```

Function Templates

If a template declares exactly one member, and that member is a function with the same name as the template:

FunctionTemplateDeclaration:
    Type *Identifier* **(** *TemplateParameterList* **)** **(** *FunctionParameterList* **)**
FunctionBody

A function template to compute the square of type *T* is:

```
T Square(T)(T t)
{
    return t * t;
}
```

Function templates can be explicitly instantiated with a !(*TemplateArgumentList*):

```
writefln("The square of %s is %s", 3, Square!(int)(3));
```

or implicitly, where the *TemplateArgumentList* is deduced from the types of the function arguments:

```
writefln("The square of %s is %s", 3, Square(3));  // T is deduced to be int
```

Function template type parameters that are to be implicitly deduced may not have specializations:

```
void Foo(T : T*)(T t) { ... }

int x,y;
Foo!(int*)(x);   // ok, T is not deduced from function argument
Foo(&y);         // error, T has specialization
```

Template arguments not implicitly deduced can have default values:

```
void Foo(T, U=T*)(T t) { U p; ... }

int x;
Foo(&x);    // T is int, U is int*
```

Recursive Templates

Template features can be combined to produce some interesting effects, such as compile time evaluation of non-trivial functions. For example, a factorial template can be written:

```
template factorial(int n : 1)
{
    enum { factorial = 1 }
}

template factorial(int n)
{
    enum { factorial = n* factorial!(n-1) }
}

void test()
{
    writefln("%s", factorial!(4));      // prints 24
}
```

Limitations

Templates cannot be used to add non-static members or functions to classes. For example:

```
class Foo
{
    template TBar(T)
    {
        T xx;                   // Error
        int func(T) { ... }     // Error

        static T yy;                            // Ok
        static int func(T t, int y) { ... }     // Ok
    }
}
```

Templates cannot be declared inside functions.

Mixins

Mixins mean different things in different programming languages. In D, a mixin takes an arbitrary set of declarations from the body of a *TemplateDeclaration* and inserts them into the current context.

```
TemplateMixin:
        mixin TemplateIdentifier ;
        mixin TemplateIdentifier MixinIdentifier ;
        mixin TemplateIdentifier !( TemplateArgumentList ) ;
        mixin TemplateIdentifier !( TemplateArgumentList ) MixinIdentifier ;

MixinIdentifier:
        Identifier
```

A *TemplateMixin* can occur in declaration lists of modules, classes, structs, unions, and as a statement. The *TemplateIdentifier* refers to a *TemplateDeclaration*. If the *TemplateDeclaration* has no parameters, the mixin form that has no !(*TemplateArgumentList*) can be used.

Unlike a template instantiation, a template mixin's body is evaluated within the scope where the mixin appears, not where the template declaration is defined. It is analogous to cutting and pasting the body of the template into the location of the mixin. It is useful for injecting parameterized 'boilerplate' code, as well as for creating templated nested functions, which is not possible with template instantiations.

```
template Foo()
{
    int x = 5;
}

mixin Foo;

struct Bar
{
    mixin Foo;
}

void test()
{
    printf("x = %d\n", x);              // prints 5
    {   Bar b;
        int x = 3;

        printf("b.x = %d\n", b.x);      // prints 5
        printf("x = %d\n", x);          // prints 3
        {
            mixin Foo;
            printf("x = %d\n", x);      // prints 5
            x = 4;
            printf("x = %d\n", x);      // prints 4
        }
        printf("x = %d\n", x);          // prints 3
    }
    printf("x = %d\n", x);              // prints 5
}
```

Mixins can be parameterized:

```
template Foo(T)
```

```
{
    T x = 5;
}

mixin Foo!(int);                // create x of type int
```

Mixins can add virtual functions to a class:

```
template Foo()
{
    void func() { printf("Foo.func()\n"); }
}

class Bar
{
    mixin Foo;
}

class Code : Bar
{
    void func() { printf("Code.func()\n"); }
}

void test()
{
    Bar b = new Bar();
    b.func();           // calls Foo.func()

    b = new Code();
    b.func();           // calls Code.func()
}
```

Mixins are evaluated in the scope of where they appear, not the scope of the template declaration:

```
int y = 3;

template Foo()
{
    int abc() { return y; }
}

void test()
{
    int y = 8;
    mixin Foo;  // local y is picked up, not global y
    assert(abc() == 8);
}
```

Mixins can parameterize symbols using alias parameters:

```
template Foo(alias b)
{
    int abc() { return b; }
}

void test()
{
    int y = 8;
    mixin Foo!(y);
    assert(abc() == 8);
}
```

This example uses a mixin to implement a generic Duff's device for an arbitrary statement (in this case, the arbitrary statement is in bold). A nested function is generated as well as a delegate literal, these can be inlined by the compiler:

```
template duffs_device(alias id1, alias id2, alias s)
{
    void duff_loop()
    {
        if (id1 < id2)
        {
            typeof(id1) n = (id2 - id1 + 7) / 8;
            switch ((id2 - id1) % 8)
            {
                case 0:        do {  s();
                case 7:              s();
                case 6:              s();
                case 5:              s();
                case 4:              s();
                case 3:              s();
                case 2:              s();
                case 1:              s();
                                  } while (--n > 0);
            }
        }
    }
}

void foo() { printf("foo\n"); }

void test()
{
    int i = 1;
    int j = 11;

    mixin duffs_device!(i, j, delegate { foo(); } );
    duff_loop();        // executes foo() 10 times
}
```

Mixin Scope

The declarations in a mixin are 'imported' into the surrounding scope. If the name of a declaration in a mixin is the same as a declaration in the surrounding scope, the surrounding declaration overrides the mixin one:

```
int x = 3;

template Foo()
{
    int x = 5;
    int y = 5;
}

mixin Foo;
int y = 3;

void test()
{
    printf("x = %d\n", x);      // prints 3
    printf("y = %d\n", y);      // prints 3
```

```
}
```

If two different mixins are put in the same scope, and each define a declaration with the same name, there is an ambiguity error when the declaration is referenced:

```
template Foo()
{
    int x = 5;
    void func(int x) { }
}

template Bar()
{
    int x = 4;
    void func() { }
}

mixin Foo;
mixin Bar;

void test()
{
    printf("x = %d\n", x);      // error, x is ambiguous
    func();             // error, func is ambiguous
}
```

The call to **func()** is ambiguous because Foo.func and Bar.func are in different scopes.

If a mixin has a *MixinIdentifier*, it can be used to disambiguate:

```
int x = 6;

template Foo()
{
    int x = 5;
    int y = 7;
    void func() { }
}

template Bar()
{
    int x = 4;
    void func() { }
}

mixin Foo F;
mixin Bar B;

void test()
{
    printf("y = %d\n", y);      // prints 7
    printf("x = %d\n", x);      // prints 6
    printf("F.x = %d\n", F.x);  // prints 5
    printf("B.x = %d\n", B.x);  // prints 4
    F.func();                   // calls Foo.func
    B.func();                   // calls Bar.func
}
```

Alias declarations can be used to overload together functions declared in different mixins:

```
template Foo()
```

```
{
    void func(int x) {  }
}

template Bar()
{
    void func() {  }
}

mixin Foo!() F;
mixin Bar!() B;

alias F.func func;
alias B.func func;

void main()
{
    func();     // calls B.func
    func(1);    // calls F.func
}
```

A mixin has its own scope, even if a declaration is overridden by the enclosing one:

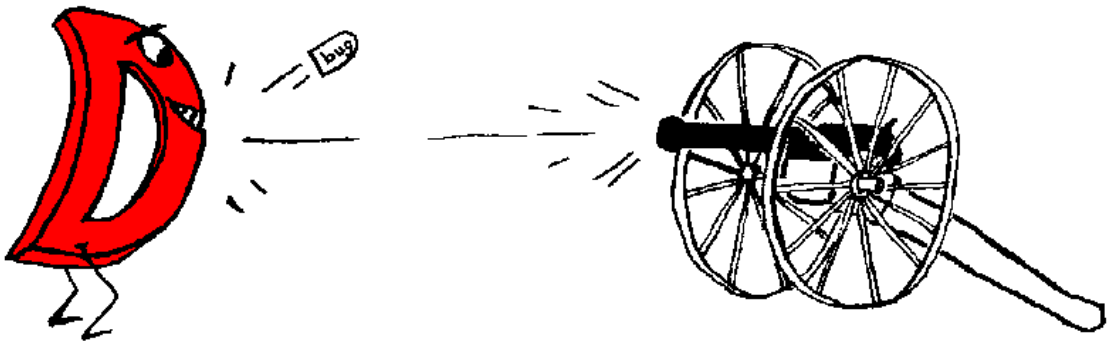
```
int x = 4;

template Foo()
{
    int x = 5;
    int bar() { return x; }
}

mixin Foo;

void test()
{
    printf("x = %d\n", x);              // prints 4
    printf("bar() = %d\n", bar());      // prints 5
}
```

Contract Programming

Contracts are a breakthrough technique to reduce the programming effort for large projects. Contracts are the concept of preconditions, postconditions, errors, and invariants. Contracts can be done in C++ without modification to the language, but the result is clumsy and inconsistent.

Building contract support into the language makes for:

1. a consistent look and feel for the contracts
2. tool support
3. it's possible the compiler can generate better code using information gathered from the contracts
4. easier management and enforcement of contracts
5. handling of contract inheritance



The idea of a contract is simple - it's just an expression that must evaluate to true. If it does not, the contract is broken, and by definition, the program has a bug in it. Contracts form part of the specification for a program, moving it from the documentation to the code itself. And as every programmer knows, documentation tends to be incomplete, out of date, wrong, or non-existent. Moving the contracts into the code makes them verifiable against the program.

Assert Contract

The most basic contract is the assert. An assert inserts a checkable expression into the code, and that expression must evaluate to true:

```
assert(expression);
```

C programmers will find it familiar. Unlike C, however, an `assert` in function bodies works by throwing an `AssertError`, which can be caught and handled. Catching the contract violation is useful when the code must deal with errant uses by other code, when it must be failure proof, and as a useful tool for debugging.

Pre and Post Contracts

The pre contracts specify the preconditions before a statement is executed. The most typical use of this would be in validating the parameters to a function. The post contracts validate the result of the statement. The most typical use of this would be in validating the return value of a function and of any side effects it has. The syntax is:

```
in
{
```

```
    ...contract preconditions...
}
out (result)
{
    ...contract postconditions...
}
body
{
    ...code...
}
```

By definition, if a pre contract fails, then the body received bad parameters. An AssertError is thrown. If a post contract fails, then there is a bug in the body. An AssertError 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 && (result+1) * (result+1) >= x);
    }
    body
    {
        return cast(long)std.math.sqrt(cast(real)x);
    }
```

The assert's in the in and out bodies are called contracts. Any other D statement or expression is allowed in the bodies, but it is important to ensure that the code has no side effects, and that the release version of the code will not depend on any effects of the code. For a release build of the code, the in and out code is not inserted.

If the function returns a void, there is no result, and so there can be no result declaration in the out clause. In that case, use:

```
void func()
   out
   {
        ...contracts...
   }
   body
   {
        ...
   }
```

In an out statement, *result* is initialized and set to the return value of the function.

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 function and its base functions must be satisfied. Overriding functions then becomes a process of *loosening* the `in` contracts.

A function without an `in` contract means that any values of the function parameters are allowed. This implies that if any function in an inheritance hierarchy has no `in` contract, then `in` contracts on functions overriding it have no useful effect.

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

Class Invariants

Class invariants are used to specify characteristics of a class that always must be true (except while executing a member function). They are described in Classes.

Conditional Compilation

Conditional compilation is the process of selecting which code to compile and which code to not compile. (In C and C++, conditional compilation is done with the preprocessor directives #if / #else / #endif.)

```
ConditionalDeclaration:
    Condition DeclarationBlock
    Condition DeclarationBlock else DeclarationBlock
    Condition : Declarations

DeclarationBlock:
    Declaration
    { Declarations }
    { }

Declarations:
    Declaration
    Declaration Declarations

ConditionalStatement:
    Condition NoScopeNonEmptyStatement
    Condition NoScopeNonEmptyStatement else NoScopeNonEmptyStatement
```

If the *Condition* is satisfied, then the following *DeclarationBlock* or *Statement* is compiled in. If it is not satisfied, the *DeclarationBlock* or *Statement* after the optional **else** is compiled in.

Any *DeclarationBlock* or *Statement* that is not compiled in still must be syntactically correct.

No new scope is introduced, even if the *DeclarationBlock* or *Statement* is enclosed by **{ }**.

*ConditionalDeclaration*s and *ConditionalStatement*s can be nested.

The *StaticAssert* can be used to issue errors at compilation time for branches of the conditional compilation that are errors. *Condition* comes in the following forms:

```
Condition:
    VersionCondition
    DebugCondition
    StaticIfCondition
```

Version Condition

Versions enable multiple versions of a module to be implemented with a single source file.

```
VersionCondition:
        version ( Integer )
        version ( Identifier )
```

The *VersionCondition* is satisfied if the *Integer* is greater than or equal to the current *version level*, or if *Identifier* matches a *version identifier*.

The *version level* and *version identifier* can be set on the command line by the **-version** switch or in the module itself with a *VersionSpecification*, or they can be predefined by the compiler.

Version identifiers are in their own unique name space, they do not conflict with debug identifiers or other symbols in the module. Version identifiers defined in one module have no influence over other imported modules.

```
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

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

Version Specification

VersionSpecification
    version = *Identifier* ;
    version = *Integer* ;

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 ...
}
```

Version identifiers or levels may not be forward referenced:

```
version (Foo)
{
    int x;
}
version = Foo;  // error, Foo already used
```

While the debug and version conditions 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.

Here's an example of 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 code ...
}

version(identifier) // add in version code if version
                        // keyword is identifier
{
   ... version code ...
}
```

These are presumably set by the command line as `-version=`*`n`* and `-version=`*`identifier`*.

Predefined Versions

Several environmental version identifiers and identifier name spaces are predefined for consistent usage. Version identifiers do not conflict with other identifiers in the code, they are in a separate name space. Predefined version identifiers are global, i.e. they apply to all modules being compiled and imported.

DigitalMars
: Digital Mars is the compiler vendor

X86
: Intel and AMD 32 bit processors

X86_64
: AMD and Intel 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_Coverage
Coverage analyser is implemented and the **-cov** switch is thrown

D_InlineAsm_X86
Inline assembler for X86 is implemented

none
Never defined; used to just disable a section of code

all
Always defined; used as the opposite of **none**

Others will be added as they make sense and new implementations appear.

It is inevitable that the D language will evolve over time. Therefore, the version identifier namespace beginning with "D_" is reserved for identifiers indicating D language specification or new feature conformance.

Furthermore, predefined version identifiers from this list cannot be set from the command line or from version statements. (This prevents things like both **Windows** and **linux** being simultaneously set.)

Compiler vendor specific versions can be predefined if the trademarked vendor identifier prefixes it, as in:

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

It is important to use the right version identifier for the right purpose. For example, use the vendor identifier when using a vendor specific feature. Use the operating system identifier when using an operating system specific feature, etc.

Debug Condition

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 statement conditionally compiles in its statement body. It is D's way of what in C is done with #ifdef DEBUG / #endif pairs.

DebugCondition:
    debug
    debug **(** *Integer* **)**
    debug **(** *Identifier* **)**

The **debug** condition is satisfied when the **-debug** switch is thrown on the compiler.

The **debug (** *Integer* **)** condition is satisfied when the debug level is >= *Integer*.

The **debug (** *Identifier* **)** condition is satisfied when the debug identifier matches *Identifier*.

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

```
}
```

Debug Specification

DebugSpecification
    debug = *Identifier* **;**
    debug = *Integer* **;**

Debug identifiers and levels are set either by the command line switch **-debug** or by a *DebugSpecification*.

Debug specifications only affect the module they appear in, they do not affect any imported modules. Debug identifiers are in their own namespace, independent from version identifiers and other symbols.

It is illegal to forward reference a debug specification:

```
debug (foo) printf("Foo\n");
debug = foo;    // error, foo used before set
```

Various different debug builds can be built with a parameter to debug:

```
debug(Integer) { }    // add in debug code if debug level is >= Integer
debug(identifier) { } // add in debug code if debug keyword is identifier
```

These are presumably set by the command line as `-debug=`*`n`* and `-debug=`*`identifier`*.

Static If Condition

StaticIfCondition:
    static if (*AssignExpression* **)**

AssignExpression is implicitly converted to a boolean type, and is evaluated at compile time. The condition is satisfied if it evaluates to **true**. It is not satisfied if it evaluates to **false**.

It is an error if *AssignExpression* cannot be implicitly converted to a boolean type or if it cannot be evaluated at compile time.

*StaticIfCondition*s can appear in module, class, template, struct, union, or function scope. In function scope, the symbols referred to in the *AssignExpression* can be any that can normally be referenced by an expression at that point.

```
const int i = 3;
int j = 4;

static if (i == 3)      // ok, at module scope
    int x;

class C
{   const int k = 5;

    static if (i == 3)  // ok
        int x;
    else
        long x;

    static if (j == 3)  // error, j is not a constant
        int y;
```

```
    static if (k == 5)  // ok, k is in current scope
        int z;
}

template INT(int i)
{
    static if (i == 32)
        alias int INT;
    else static if (i == 16)
        alias short INT;
    else
        static assert(0);       // not supported
}

INT!(32) a;     // a is an int
INT!(16) b;     // b is a short
INT!(17) c;     // error, static assert trips
```

A *StaticIfConditional* condition differs from an *IfStatement* in the following ways:

1. It can be used to conditionally compile declarations, not just statements.
2. It does not introduce a new scope even if **{ }** are used for conditionally compiled statements.
3. For unsatisfied conditions, the conditionally compiled code need only be syntactically correct. It does not have to be semantically correct.
4. It must be evaluatable at compile time.

Static Assert

StaticAssert:
    static assert (*AssignExpression*);
    static assert (*AssignExpression* , *AssignExpression*);

AssignExpression 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 an unsatisfied conditional.

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

StaticAssert is useful tool for drawing attention to conditional configurations not supported in the code.

The optional second *AssertExpression* can be used to supply additional information, such as a text string, that will be printed out along with the error diagnostic.

Error Handling in D

I came, I coded, I crashed. -- Julius C'ster

All programs have to deal with errors. Errors are unexpected conditions that are not part of the normal operation of a program. Examples of common errors are:

- Out of memory.
- Out of disk space.
- Invalid file name.
- Attempting to write to a read-only file.
- Attempting to read a non-existent file.
- Requesting a system service that is not supported.

The Error Handling Problem

The traditional C way of detecting and reporting errors is not traditional, it is ad-hoc and varies from function to function, including:

- Returning a NULL pointer.
- Returning a 0 value.
- Returning a non-zero error code.
- Requiring errno to be checked.
- Requiring that a function be called to check if the previous function failed.

To deal with these possible errors, tedious error handling code must be added to each function call. If an error happened, code must be written to recover from the error, and the error must be reported to the user in some user friendly fashion. If an error cannot be handled locally, it must be explicitly propagated back to its caller. The long list of errno values needs to be converted into appropriate text to be displayed. Adding all the code to do this can consume a large part of the time spent coding a project - and still, if a new errno value is added to the runtime system, the old code can not properly display a meaningful error message.

Good error handling code tends to clutter up what otherwise would be a neat and clean looking implementation.

Even worse, good error handling code is itself error prone, tends to be the least tested (and therefore buggy) part of the project, and is frequently simply omitted. The end result is likely a "blue screen of death" as the program failed to deal with some unanticipated error.

Quick and dirty programs are not worth writing tedious error handling code for, and so such utilities tend to be like using a table saw with no blade guards.

What's needed is an error handling philosophy and methodology such that:

- It is standardized - consistent usage makes it more useful.
- The result is reasonable even if the programmer fails to check for errors.
- Old code can be reused with new code without having to modify the old code to be compatible with new error types.
- No errors get inadvertently ignored.
- 'Quick and dirty' utilities can be written that still correctly handle errors.
- It is easy to make the error handling source code look good.

The D Error Handling Solution

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

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

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

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

How does this meet our criteria?

It is standardized - consistent usage makes it more useful.
: This is the D way, and is used consistently in the D runtime library and examples.

The result is 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.

Old code can 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.

'Quick and dirty' utilities can 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.

It is 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.

Using exceptions to handle errors leads to another issue - how to write exception safe programs. Here's how.

Garbage Collection

D is a fully garbage collected language. That means that it is never necessary to free memory. Just allocate as needed, and the garbage collector will periodically return all unused memory to the pool of available memory.

C and C++ programmers accustomed to explicitly managing memory allocation and deallocation will likely be skeptical of the benefits and efficacy of garbage collection. Experience both with new projects written with garbage collection in mind, and converting existing projects to garbage collection shows that:

- Garbage collected programs are faster. This is counterintuitive, but the reasons are:
  - Reference counting is a common solution to solve explicit memory allocation problems. The code to implement the increment and decrement operations whenever assignments are made is one source of slowdown. Hiding it behind smart pointer classes doesn't help the speed. (Reference counting methods are not a general solution anyway, as circular references never get deleted.)
- Destructors are used to deallocate resources acquired by an object. For most classes, this resource is allocated memory. With garbage collection, most destructors then become empty and can be discarded entirely.
- All those destructors freeing memory can become significant when objects are allocated on the stack. For each one, some mechanism must be established so that if an exception happens, the destructors all get called in each frame to release any memory they hold. If the destructors become irrelevant, then there's no need to set up special stack frames to handle exceptions, and the code runs faster.
- All the code necessary to manage memory can add up to quite a bit. The larger a program is, the less in the cache it is, the more paging it does, and the slower it runs.
- Garbage collection kicks in only when memory gets tight. When memory is not tight, the program runs at full speed and does not spend any time freeing memory.
- Modern garbage collectors are far more advanced now than the older, slower ones. Generational, copying collectors eliminate much of the inefficiency of early mark and sweep algorithms.
- Modern garbage collectors do heap compaction. Heap compaction tends to reduce the number of pages actively referenced by a program, which means that memory accesses are more likely to be cache hits and less swapping.
- Garbage collected programs do not suffer from gradual deterioration due to an accumulation of memory leaks.
- Garbage collectors reclaim unused memory, therefore they do not suffer from "memory leaks" which can cause long running applications to gradually consume more and more memory until they bring down the system. GC programs have longer term stability.
- Garbage collected programs have fewer hard-to-find pointer bugs. This is because there are no dangling references to freed memory. There is no code to explicitly manage memory, hence no bugs in such code.
- Garbage collected programs are faster to develop and debug, because there's no need for developing, debugging, testing, or maintaining the explicit deallocation code.
- Garbage collected programs can be significantly smaller, because there is no code to manage deallocation, and there is no need for exception handlers to deallocate memory.

Garbage collection is not a panacea. There are some downsides:

- It is not predictable when a collection gets run, so the program can arbitrarily pause.
- The time it takes for a collection to run is not bounded. While in practice it is very quick,

this cannot be guaranteed.
- All threads other than the collector thread must be halted while the collection is in progress.
- Garbage collectors can keep around some memory that an explicit deallocator would not. In practice, this is not much of an issue since explicit deallocators usually have memory leaks causing them to eventually use far more memory, and because explicit deallocators do not normally return deallocated memory to the operating system anyway, instead just returning it to its own internal pool.
- Garbage collection should be implemented as a basic operating system kernel service. But since they are not, garbage collecting programs must carry around with them the garbage collection implementation. While this can be a shared DLL, it is still there.

These constraints are addressed by techniques outlined in Memory Management.

How Garbage Collection Works

The GC works by:

1. Looking for all the pointer 'roots' into GC allocated memory.
2. Recursively scanning all allocated memory pointed to by roots looking for more pointers into GC allocated memory.
3. Freeing all GC allocated memory that has no active pointers to it.
4. Possibly compacting the remaining used memory by copying the allocated objects (called a copying collector).

Interfacing Garbage Collected Objects With Foreign Code

The garbage collector looks for roots in:

1. its static data segment
2. the stacks and register contents of each thread
3. any roots added by std.gc.addRoot() or std.gc.addRange()

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.
- Add a root to the object using std.gc.addRoot() or std.gc.addRange().
- Reallocate and copy the object using the foreign code's storage allocator or using the C runtime library's malloc/free.

Pointers and the Garbage Collector

Pointers in D can be broadly divided into two categories: those that point to garbage collected memory, and those that do not. Examples of the latter are pointers created by calls to C's malloc(), pointers received from C library routines, pointers to static data, pointers to objects on the stack, etc. For those pointers, anything that is legal in C can be done with them.

For garbage collected pointers and references, however, there are some restrictions. These restrictions are minor, but they are intended to enable the maximum flexibility in garbage collector design.

Undefined behavior:

- Do not xor pointers with other values, like the xor pointer linked list trick used in C.
- Do not use the xor trick to swap two pointer values.
- Do not store pointers into non-pointer variables using casts and other tricks.

```
void* p;
...
int x = cast(int)p;   // error: undefined behavior
```

  The garbage collector does not scan non-pointer types for roots.
- Do not take advantage of alignment of pointers to store bit flags in the low order bits:

```
p = cast(void*)(cast(int)p | 1);  // error: undefined behavior
```

- Do not store into pointers values that may point into the garbage collected heap:

```
p = cast(void*)12345678;   // error: undefined behavior
```

  A copying garbage collector may change this value.
- Do not store magic values into pointers, other than `null`.
- Do not write pointer values out to disk and read them back in again.
- Do not use pointer values to compute a hash function. A copying garbage collector can arbitrarily move objects around in memory, thus invalidating the computed hash value.
- Do not depend on the ordering of pointers:

```
if (p1 < p2)            // error: undefined behavior
    ...
```

  since, again, the garbage collector can move objects around in memory.
- Do not add or subtract an offset to a pointer such that the result points outside of the bounds of the garbage collected object originally allocated.

```
char* p = new char[10];
char* q = p + 6;        // ok
q = p + 11;             // error: undefined behavior
q = p - 1;              // error: undefined behavior
```

- Do not misalign pointers if those pointers may point into the gc heap, such as:

```
align (1) struct Foo
{   byte b;
    char* p;    // misaligned pointer
}
```

  Misaligned pointers may be used if the underlying hardware supports them **and** the pointer is never used to point into the gc heap.
- Do not use byte-by-byte memory copies to copy pointer values. This may result in intermediate conditions where there is not a valid pointer, and if the gc pauses the thread in such a condition, it can corrupt memory. Most implementations of `memcpy()` will work since the internal implementation of it does the copy in aligned chunks greater than or equal to a pointer size, but since this kind of implementation is not guaranteed by the C standard, use `memcpy()` only with extreme caution.

Things that are reliable and can be done:

- Use a union to share storage with a pointer:

```
union U { void* ptr; int value }
```

- A pointer to the start of a garbage collected object need not be maintained if a pointer to the interior of the object exists.

```
char[] p = new char[10];
char[] q = p[3..6];
// q is enough to hold on to the object, don't need to keep
// p as well.
```

One can avoid using pointers anyway for most tasks. D provides features rendering most explicit pointer uses obsolete, such as reference objects, dynamic arrays, and garbage collection. Pointers are provided in order to interface successfully with C APIs and for some low level work.

Working with the Garbage Collector

Garbage collection doesn't solve every memory deallocation problem. For example, if a root to a large data structure is kept, the garbage collector cannot reclaim it, even if it is never referred to again. To eliminate this problem, it is good practice to set a reference or pointer to an object to null when no longer needed.

This advice applies only to static references or references embedded inside other objects. There is not much point for such stored on the stack to be nulled, since the collector doesn't scan for roots past the top of the stack, and because new stack frames are initialized anyway.

References

- Wikipedia
- GC FAQ
- Uniprocessor Garbage Collector Techniques
- Garbage Collection : Algorithms for Automatic Dynamic Memory Management

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 real (extended) type, should only be used for:

- reducing memory consumption for large arrays
- when speed is more important than accuracy
- data and function argument compatibility with C

Floating Point Constant Folding

Regardless of the type of the operands, floating point constant folding is done in **real** or greater precision. It is always done following IEEE 754 rules and round-to-nearest is used.

Floating point constants are internally represented in the implementation in at least **real** precision, regardless of the constant's type. The extra precision is available for constant folding. Committing to the precision of the result is done as late as possible in the compilation process. For example:

```
const float f = 0.2f;
writefln(f - 0.2);
```

will print 0. A non-const static variable's value cannot be propagated at compile time, so:

```
static float f = 0.2f;
writefln(f - 0.2);
```

will print 2.98023e-09. Hex floating point constants can also be used when specific floating point bit patterns are needed that are unaffected by rounding. To find the hex value of 0.2f:

```
import std.stdio;

void main()
{
    writefln("%a", 0.2f);
}
```

which is 0x1.99999ap-3. Using the hex constant:

```
const float f = 0x1.99999ap-3f;
```

```
writefln(f - 0.2);
```

prints 2.98023e-09.

Different compiler settings, optimization settings, and inlining settings can affect opportunities for constant folding, therefore the results of floating point calculations may differ depending on those settings.

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, real and imaginary numbers have two properties:

```
.re     get real part (0 for imaginary numbers)
.im     get imaginary part as a real (0 for real numbers)
```

For example:

```
cd.re           is 4.5 double
cd.im           is 2 double
c.re            is 4.5 real
c.im            is 2 real
j.im            is 1.3 real
j.re            is 0 real
```

Rounding Control

IEEE 754 floating point arithmetic includes the ability to set 4 different rounding modes. These are accessible via the functions in std.c.fenv.

Exception Flags

IEEE 754 floating point arithmetic can set several flags based on what happened with a computation:

FE_INVALID
FE_DENORMAL
FE_DIVBYZERO
FE_OVERFLOW
FE_UNDERFLOW
FE_INEXACT
These flags can be set/reset via the functions in std.c.fenv.

Floating Point Comparisons

In addition to the usual < <= > >= == != comparison operators, D adds more that are specific to floating point. These are !<>= <> <>= !<= !< !>= !> !<> and match the semantics for the NCEG extensions to C. See Floating point comparisons.

D x86 Inline Assembler

D, being a systems programming language, provides an inline assembler. The inline assembler is standardized for D implementations across the same CPU family, for example, the Intel Pentium inline assembler for a Win32 D compiler will be syntax compatible with the inline assembler for Linux running on an Intel Pentium.

Differing D implementations, however, are free to innovate upon the memory model, function call/return conventions, argument passing conventions, etc.

This document describes the x86 implementation of the inline assembler.

```
AsmInstruction:
        Identifier : AsmInstruction
        align IntegerExpression
        even
        naked
        db Operands
        ds Operands
        di Operands
        dl Operands
        df Operands
        dd Operands
        de Operands
        Opcode
        Opcode Operands

Operands
        Operand
        Operand , Operands
```

Labels

Assembler instructions can be labeled just like other statements. They can be the target of goto statements. For example:

```
void *pc;
asm
{
    call L1             ;
 L1:                    ;
    pop EBX             ;
    mov pc[EBP],EBX     ;       // pc now points to code at L1
}
```

align *IntegerExpression*

Causes the assembler to emit NOP instructions to align the next assembler instruction on an *IntegerExpression* boundary. *IntegerExpression* must evaluate to an integer that is a power of 2.

Aligning the start of a loop body can sometimes have a dramatic effect on the execution speed.

even

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

naked

Causes the compiler to not generate the function prolog and epilog sequences. This means such is the responsibility of inline assembly programmer, and is normally used when the entire function is to be written in assembler.

db, ds, di, dl, df, dd, de

These pseudo ops are for inserting raw data directly into the code. **db** is for bytes, **ds** is for 16 bit words, **di** is for 32 bit words, **dl** is for 64 bit words, **df** is for 32 bit floats, **dd** is for 64 bit doubles, and **de** is for 80 bit extended reals. Each can have multiple operands. If an operand is a string literal, it is as if there were *length* operands, where *length* is the number of characters in the string. One character is used per operand. For example:

```
asm
{
    db 5,6,0x83;   // insert bytes 0x05, 0x06, and 0x83 into code
    ds 0x1234;     // insert bytes 0x34, 0x12
    di 0x1234;     // insert bytes 0x34, 0x12, 0x00, 0x00
    dl 0x1234;     // insert bytes 0x34, 0x12, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00
    df 1.234;      // insert float 1.234
    dd 1.234;      // insert double 1.234
    de 1.234;      // insert real 1.234
    db "abc";      // insert bytes 0x61, 0x62, and 0x63
    ds "abc";      // insert bytes 0x61, 0x00, 0x62, 0x00, 0x63, 0x00
}
```

Opcodes

A list of supported opcodes is at the end.

The following registers are supported. Register names are always in upper case.

AL, **AH**, **AX**, **EAX**
BL, **BH**, **BX**, **EBX**
CL, **CH**, **CX**, **ECX**
DL, **DH**, **DX**, **EDX**
BP, **EBP**
SP, **ESP**
DI, **EDI**
SI, **ESI**
ES, **CS**, **SS**, **DS**, **GS**, **FS**
CR0, **CR2**, **CR3**, **CR4**
DR0, **DR1**, **DR2**, **DR3**, **DR6**, **DR7**
TR3, **TR4**, **TR5**, **TR6**, **TR7**
ST
ST(0), **ST(1)**, **ST(2)**, **ST(3)**, **ST(4)**, **ST(5)**, **ST(6)**, **ST(7)**

MM0, **MM1**, **MM2**, **MM3**, **MM4**, **MM5**, **MM6**, **MM7**
XMM0, **XMM1**, **XMM2**, **XMM3**, **XMM4**, **XMM5**, **XMM6**, **XMM7**

Special Cases

lock, **rep**, **repe**, **repne**, **repnz**, **repz**
These prefix instructions do not appear in the same statement as the instructions they prefix; they appear in their own statement. For example:

```
asm
{
    rep   ;
    movsb ;
}
```

pause
This opcode is not supported by the assembler, instead use

```
{
    rep  ;
    nop  ;
}
```

which produces the same result.

floating point ops
Use the two operand form of the instruction format;

```
fdiv ST(1);     // wrong
fmul ST;        // wrong
fdiv ST,ST(1);  // right
fmul ST,ST(0);  // right
```

Operands

Operand:
    AsmExp

AsmExp:
    AsmLogOrExp
    AsmLogOrExp **?** *AsmExp* **:** *AsmExp*

AsmLogOrExp:
    AsmLogAndExp
    AsmLogAndExp **||** *AsmLogAndExp*

AsmLogAndExp:
    AsmOrExp
    AsmOrExp **&&** *AsmOrExp*

AsmOrExp:
    AsmXorExp
    AsmXorExp **|** *AsmXorExp*

AsmXorExp:
    AsmAndExp

```
    AsmAndExp ^ AsmAndExp

AsmAndExp:
    AsmEqualExp
    AsmEqualExp & AsmEqualExp

AsmEqualExp:
    AsmRelExp
    AsmRelExp == AsmRelExp
    AsmRelExp != AsmRelExp

AsmRelExp:
    AsmShiftExp
    AsmShiftExp < AsmShiftExp
    AsmShiftExp <= AsmShiftExp
    AsmShiftExp > AsmShiftExp
    AsmShiftExp >= AsmShiftExp

AsmShiftExp:
    AsmAddExp
    AsmAddExp << AsmAddExp
    AsmAddExp >> AsmAddExp
    AsmAddExp >>> AsmAddExp

AsmAddExp:
    AsmMulExp
    AsmMulExp + AsmMulExp
    AsmMulExp - AsmMulExp

AsmMulExp:
    AsmBrExp
    AsmBrExp * AsmBrExp
    AsmBrExp / AsmBrExp
    AsmBrExp % AsmBrExp

AsmBrExp:
    AsmUnaExp
    AsmBrExp [ AsmExp ]

AsmUnaExp:
    AsmTypePrefix AsmExp
    offset AsmExp
    seg AsmExp
    + AsmUnaExp
    - AsmUnaExp
    ! AsmUnaExp
    ~ AsmUnaExp
    AsmPrimaryExp

AsmPrimaryExp
    IntegerConstant
    FloatConstant
    __LOCAL_SIZE
    $
    Register
    DotIdentifier

DotIdentifier
    Identifier
    Identifier . DotIdentifier
```

The operand syntax more or less follows the Intel CPU documentation conventions. In particular, the convention is that for two operand instructions the source is the right operand and the destination is the left operand. The syntax differs from that of Intel's in order to be compatible with the D language tokenizer and to simplify parsing.

Operand Types

AsmTypePrefix:
```
        near ptr
        far ptr
        byte ptr
        short ptr
        int ptr
        word ptr
        dword ptr
        float ptr
        double ptr
        real ptr
```

In cases where the operand size is ambiguous, as in:

```
add     [EAX],3         ;
```

it can be disambiguated by using an *AsmTypePrefix*:

```
add     byte ptr [EAX],3        ;
add     int ptr [EAX],7         ;
```

Struct/Union/Class Member Offsets

To access members of an aggregate, given a pointer to the aggregate is in a register, use the qualified name of the member:

```
struct Foo { int a,b,c; }
int bar(Foo *f)
{
    asm
    {   mov     EBX,f           ;
        mov     EAX,Foo.b[EBX]  ;
    }
}
```

Special Symbols

$$

Represents the program counter of the start of the next instruction. So,

```
jmp     $  ;
```

branches to the instruction following the jmp instruction.

__LOCAL_SIZE

This gets replaced by the number of local bytes in the local stack frame. It is most handy when the **naked** is invoked and a custom stack frame is programmed.

Opcodes Supported

aaa	aad	aam	aas	adc
add	addpd	addps	addsd	addss
and	andnpd	andnps	andpd	andps
arpl	bound	bsf	bsr	bswap
bt	btc	btr	bts	call
cbw	cdq	clc	cld	clflush
cli	clts	cmc	cmova	cmovae
cmovb	cmovbe	cmovc	cmove	cmovg
cmovge	cmovl	cmovle	cmovna	cmovnae
cmovnb	cmovnbe	cmovnc	cmovne	cmovng
cmovnge	cmovnl	cmovnle	cmovno	cmovnp
cmovns	cmovnz	cmovo	cmovp	cmovpe
cmovpo	cmovs	cmovz	cmp	cmppd
cmpps	cmps	cmpsb	cmpsd	cmpss
cmpsw	cmpxch8b	cmpxchg	comisd	comiss
cpuid	cvtdq2pd	cvtdq2ps	cvtpd2dq	cvtpd2pi
cvtpd2ps	cvtpi2pd	cvtpi2ps	cvtps2dq	cvtps2pd
cvtps2pi	cvtsd2si	cvtsd2ss	cvtsi2sd	cvtsi2ss
cvtss2sd	cvtss2si	cvttpd2dq	cvttpd2pi	cvttps2dq
cvttps2pi	cvttsd2si	cvttss2si	cwd	cwde
da	daa	das	db	dd
de	dec	df	di	div
divpd	divps	divsd	divss	dl
dq	ds	dt	dw	emms
enter	f2xm1	fabs	fadd	faddp
fbld	fbstp	fchs	fclex	fcmovb
fcmovbe	fcmove	fcmovnb	fcmovnbe	fcmovne
fcmovnu	fcmovu	fcom	fcomi	fcomip
fcomp	fcompp	fcos	fdecstp	fdisi
fdiv	fdivp	fdivr	fdivrp	feni
ffree	fiadd	ficom	ficomp	fidiv
fidivr	fild	fimul	fincstp	finit

fist	fistp	fisub	fisubr	fld
fld1	fldcw	fldenv	fldl2e	fldl2t
fldlg2	fldln2	fldpi	fldz	fmul
fmulp	fnclex	fndisi	fneni	fninit
fnop	fnsave	fnstcw	fnstenv	fnstsw
fpatan	fprem	fprem1	fptan	frndint
frstor	fsave	fscale	fsetpm	fsin
fsincos	fsqrt	fst	fstcw	fstenv
fstp	fstsw	fsub	fsubp	fsubr
fsubrp	ftst	fucom	fucomi	fucomip
fucomp	fucompp	fwait	fxam	fxch
fxrstor	fxsave	fxtract	fyl2x	fyl2xp1
hlt	idiv	imul	in	inc
ins	insb	insd	insw	int
into	invd	invlpg	iret	iretd
ja	jae	jb	jbe	jc
jcxz	je	jecxz	jg	jge
jl	jle	jmp	jna	jnae
jnb	jnbe	jnc	jne	jng
jnge	jnl	jnle	jno	jnp
jns	jnz	jo	jp	jpe
jpo	js	jz	lahf	lar
ldmxcsr	lds	lea	leave	les
lfence	lfs	lgdt	lgs	lidt
lldt	lmsw	lock	lods	lodsb
lodsd	lodsw	loop	loope	loopne
loopnz	loopz	lsl	lss	ltr
maskmovdqu	maskmovq	maxpd	maxps	maxsd
maxss	mfence	minpd	minps	minsd
minss	mov	movapd	movaps	movd
movdq2q	movdqa	movdqu	movhlps	movhpd
movhps	movlhps	movlpd	movlps	movmskpd
movmskps	movntdq	movnti	movntpd	movntps

movntq	movq	movq2dq	movs	movsb
movsd	movss	movsw	movsx	movupd
movups	movzx	mul	mulpd	mulps
mulsd	mulss	neg	nop	not
or	orpd	orps	out	outs
outsb	outsd	outsw	packssdw	packsswb
packuswb	paddb	paddd	paddq	paddsb
paddsw	paddusb	paddusw	paddw	pand
pandn	pavgb	pavgw	pcmpeqb	pcmpeqd
pcmpeqw	pcmpgtb	pcmpgtd	pcmpgtw	pextrw
pinsrw	pmaddwd	pmaxsw	pmaxub	pminsw
pminub	pmovmskb	pmulhuw	pmulhw	pmullw
pmuludq	pop	popa	popad	popf
popfd	por	prefetchnta	prefetcht0	prefetcht1
prefetcht2	psadbw	pshufd	pshufhw	pshuflw
pshufw	pslld	pslldq	psllq	psllw
psrad	psraw	psrld	psrldq	psrlq
psrlw	psubb	psubd	psubq	psubsb
psubsw	psubusb	psubusw	psubw	punpckhbw
punpckhdq	punpckhqdq	punpckhwd	punpcklbw	punpckldq
punpcklqdq	punpcklwd	push	pusha	pushad
pushf	pushfd	pxor	rcl	rcpps
rcpss	rcr	rdmsr	rdpmc	rdtsc
rep	repe	repne	repnz	repz
ret	retf	rol	ror	rsm
rsqrtps	rsqrtss	sahf	sal	sar
sbb	scas	scasb	scasd	scasw
seta	setae	setb	setbe	setc
sete	setg	setge	setl	setle
setna	setnae	setnb	setnbe	setnc
setne	setng	setnge	setnl	setnle
setno	setnp	setns	setnz	seto
setp	setpe	setpo	sets	setz

sfence	sgdt	shl	shld	shr
shrd	shufpd	shufps	sidt	sldt
smsw	sqrtpd	sqrtps	sqrtsd	sqrtss
stc	std	sti	stmxcsr	stos
stosb	stosd	stosw	str	sub
subpd	subps	subsd	subss	sysenter
sysexit	test	ucomisd	ucomiss	ud2
unpckhpd	unpckhps	unpcklpd	unpcklps	verr
verw	wait	wbinvd	wrmsr	xadd
xchg	xlat	xlatb	xor	xorpd
xorps				

Pentium 4 (Prescott) Opcodes Supported

addsubpd	addsubps	fisttp	haddpd	haddps
hsubpd	hsubps	lddqu	monitor	movddup
movshdup	movsldup	mwait		

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			

Embedded Documentation

The D programming language enables embedding both contracts and test code along side the actual code, which helps to keep them all consistent with each other. One thing lacking is the documentation, as ordinary comments are usually unsuitable for automated extraction and formatting into manual pages. Embedding the user documentation into the source code has important advantages, such as not having to write the documentation twice, and the likelihood of the documentation staying consistent with the code.

Some existing approaches to this are:

- Doxygen which already has some support for D
- Java's Javadoc, probably the most well-known
- C#'s embedded XML
- Other documentation tools

D's goals for embedded documentation are:

1. It looks good as embedded documentation, not just after it is extracted and processed.
2. It's easy and natural to write, i.e. minimal reliance on <tags> and other clumsy forms one would never see in a finished document.
3. It does not repeat information that the compiler already knows from parsing the code.
4. It doesn't rely on embedded HTML, as such will impede extraction and formatting for other purposes.
5. It's based on existing D comment forms, so it is completely independent of parsers only interested in D code.
6. It should look and feel different from code, so it won't be visually confused with code.
7. It should be possible for the user to use Doxygen or other documentation extractor if desired.

Specification

The specification for the form of embedded documentation comments only specifies how information is to be presented to the compiler. It is implementation-defined how that information is used and the form of the final presentation. Whether the final presentation form is an HTML web page, a man page, a PDF file, etc. is not specified as part of the D Programming Language.

Phases of Processing

Embedded documentation comments are processed in a series of phases:

1. Lexical - documentation comments are identified and attached to tokens.
2. Parsing - documentation comments are associated with specific declarations and combined.
3. Sections - each documentation comment is divided up into a sequence of sections.
4. Special sections are processed.
5. Highlighting of non-special sections is done.
6. All sections for the module are combined.
7. Macro text substitution is performed to produce the final result.

Lexical

Embedded documentation comments are one of the following forms:

1. /** ... */ The two *'s after the opening /
2. /++ ... +/ The two +'s after the opening /
3. /// The three slashes

The following are all embedded documentation comments:

```
/// This is a one line documentation comment.

/** So is this. */

/++ And this. +/

/**
   This is a brief documentation comment.
 */

/**
 * The leading * on this line is not part of the documentation comment.
 */

/*********************************
   The extra *'s immediately following the /** are not
   part of the documentation comment.
 */

/++
   This is a brief documentation comment.
 +/

/++
 + The leading + on this line is not part of the documentation comment.
 +/

/+++++++++++++++++++++++++++++++++
   The extra +'s immediately following the / ++ are not
   part of the documentation comment.
 +/

/*************** Closing *'s are not part *****************/
```

The extra *'s and +'s on the comment opening, closing and left margin are ignored and are not part of the embedded documentation. Comments not following one of those forms are not documentation comments.

Parsing

Each documentation comment is associated with a declaration. If the documentation comment is on a line by itself or with only whitespace to the left, it refers to the next declaration. Multiple documentation comments applying to the same declaration are concatenated. Documentation comments not associated with a declaration are ignored. Documentation comments preceding the *ModuleDeclaration* apply to the entire module. If the documentation comment appears on the same line to the right of a declaration, it applies to that.

If a documentation comment for a declaration consists only of the identifier `ditto` then the documentation comment for the previous declaration at the same declaration scope is applied to this declaration as well.

If there is no documentation comment for a declaration, that declaration may not appear in the

output. To ensure it does appear in the output, put an empty declaration comment for it.

```
int a;  /// documentation for a; b has no documentation
int b;

/** documentation for c and d */
/** more documentation for c and d */
int c;
/** ditto */
int d;

/** documentation for e and f */ int e;
int f;  /// ditto

/** documentation for g */
int g; /// more documentation for g

/// documentation for C and D
class C
{
    int x;    /// documentation for C.x

    /** documentation for C.y and C.z */
    int y;
    int z;    /// ditto
}

/// ditto
class D
{
}
```

Sections

The document comment is a series of *Section*s. A *Section* is a name that is the first non-blank character on a line immediately followed by a ':'. This name forms the section name. The section name is not case sensitive.

Summary

The first section is the *Summary*, and does not have a section name. It is first paragraph, up to a blank line or a section name. While the summary can be any length, try to keep it to one line. The *Summary* section is optional.

Description

The next unnamed section is the *Description*. It consists of all the paragraphs following the *Summary* until a section name is encountered or the end of the comment.

While the *Description* section is optional, there cannot be a *Description* without a *Summary* section.

```
/**********************************
 * Brief summary of what
 * myfunc does, forming the summary section.
 *
 * First paragraph of synopsis description.
 *
 * Second paragraph of
 * synopsis description.
```

```
 */

void myfunc() { }
```

Named sections follow the *Summary* and *Description* unnamed sections.

Standard Sections

For consistency and predictability, there are several standard sections. None of these are required to be present.

Authors:
Lists the author(s) of the declaration.

```
/**
 * Authors: Melvin D. Nerd, melvin@mailinator.com
 */
```

Bugs:
Lists any known bugs.

```
/**
 * Bugs: Doesn't work for negative values.
 */
```

Date:
Specifies the date of the current revision. The date should be in a form parseable by std.date.

```
/**
 * Date: March 14, 2003
 */
```

Deprecated:
Provides an explanation for and corrective action to take if the associated declaration is marked as deprecated.

```
/**
 * Deprecated: superseded by function bar().
 */

deprecated void foo() { ... }
```

Examples:
Any usage examples

```
/**
 * Examples:
 * --------------------
 * writefln("3"); // writes '3' to stdout
 * --------------------
 */
```

History:
Revision history.

```
/**
 * History:
 *      V1 is initial version
 *
 *      V2 added feature X
 */
```

License:

Any license information for copyrighted code.

```
/**
 * License: use freely for any purpose
 */

void bar() { ... }
```

Returns:

Explains the return value of the function. If the function returns **void**, don't redundantly document it.

```
/**
 * Read the file.
 * Returns: The contents of the file.
 */

void[] readFile(char[] filename) { ... }
```

See_Also:

List of other symbols and URL's to related items.

```
/**
 * See_Also:
 *    foo, bar, http://www.digitalmars.com/d/phobos/index.html
 */
```

Standards:

If this declaration is compliant with any particular standard, the description of it goes here.

```
/**
 * Standards: Conforms to DSPEC-1234
 */
```

Throws:

Lists exceptions thrown and under what circumstances they are thrown.

```
/**
 * Write the file.
 * Throws: WriteException on failure.
 */

void writeFile(char[] filename) { ... }
```

Version:

Specifies the current version of the declaration.

```
/**
 * Version: 1.6a
 */
```

Special Sections

Some sections have specialized meanings and syntax.

Copyright:

This contains the copyright notice. The macro COPYRIGHT is set to the contents of the section when it documents the module declaration. The copyright section only gets this special treatment when it is for the module declaration.

```
/** Copyright: Public Domain */

module foo;
```

Params:

Function parameters can be documented by listing them in a params section. Each line that starts with an identifier followed by an '=' starts a new parameter description. A description can span multiple lines.

```
/**********************************
 * foo does this.
 * Params:
 *      x =     is for this
 *              and not for that
 *      y =     is for that
 */

void foo(int x, int y)
{
}
```

Macros:

The macros section follows the same syntax as the **Params:** section. It's a series of *NAME*=*value* pairs. The *NAME* is the macro name, and *value* is the replacement text.

```
/**
 * Macros:
 *      FOO =   now is the time for
 *              all good men
 *      BAR =   bar
 *      MAGENTA =   &lt;font color=magenta&gt;&lt;/font&gt;
 */
```

Highlighting

Embedded Comments

The documentation comments can themselves be commented using the $(DDOC_COMMENT comment text) syntax. These comments do not nest.

Embedded Code

D code can be embedded using lines with at least three hyphens in them to delineate the code section:

```
/++++++++++++++++++++++++
 + Our function.
 + Example:
 + -------------------------
 +  import std.stdio;
 +
 +  void foo()
 +  {
 +     writefln("foo!");  /* print the string */
 +  }
 + -------------------------
 +/
```

Note that the documentation comment uses the /++ ... +/ form so that /* ... */ can be used inside the code section.

Embedded HTML

HTML can be embedded into the documentation comments, and it will be passed through to the HTML output unchanged. However, since it is not necessarily true that HTML will be the desired output format of the embedded documentation comment extractor, it is best to avoid using it where practical.

```
/** Example of embedded HTML:
 *
 *      <li> <a href="www.digitalmars.com">Digital Mars</a>
 *      <li> <a href="www.classicempire.com">Empire</a>
 *
 */
```

Emphasis

Identifiers in documentation comments that are function parameters or are names that are in scope at the associated declaration are emphasized in the output. This emphasis can take the form of italics, boldface, a hyperlink, etc. How it is emphasized depends on what it is - a function parameter, type, D keyword, etc. To prevent unintended emphasis of an identifier, it can be preceded by an underscore (_). The underscore will be stripped from the output.

Character Entities

Some characters have special meaning to the documentation processor, to avoid confusion it can be best to replace them with their corresponding character entities:

Character	Entity
<	&lt;
>	&gt;
&	&amp;

It is not necessary to do this inside a code section, or if the special character is not immediately followed by a # or a letter.

Macros

The documentation comment processor includes a simple macro text preprocessor. When a $(*NAME*) appears in section text it is replaced with *NAME*'s corresponding replacement text. The replacement text is then recursively scanned for more macros. If a macro is recursively encountered, with no argument or with the same argument text as the enclosing macro, it is replaced with no text. Macro invocations that cut across replacement text boundaries are not expanded. If the macro name is undefined, the replacement text has no characters in it. If a $(NAME) is desired to exist in the output without being macro expanded, the $ should be replaced with $.

Macros can have arguments. Any text from the end of the identifier to the closing ')' is the $0 argument. A $0 in the replacement text is replaced with the argument text. If there are commas in the argument text, $1 will represent the argument text up to the first comma, $2 from the first comma to the second comma, etc., up to $9. $+ represents the text from the first comma to the closing ')'. The argument text can contain nested parentheses, "" or '' strings, comments, or tags. If stray, unnested parentheses are used, they can be replaced with the entity (for (and) for).

Macro definitions come from the following sources, in the specified order:

1. Predefined macros.
2. Definitions from file specified by sc.ini's DDOCFILE setting.
3. Definitions from *.ddoc files specified on the command line.
4. Runtime definitions generated by Ddoc.
5. Definitions from any Macros: sections.

Macro redefinitions replace previous definitions of the same name. This means that the sequence of macro definitions from the various sources forms a hierarchy.

Macro names beginning with "D_" and "DDOC_" are reserved.

Predefined Macros

These are hardwired into Ddoc, and represent the minimal definitions needed by Ddoc to format and highlight the presentation. The definitions are for simple HTML.

```
B =      <b>$0</b>
I =      <i>$0</i>
U =      <u>$0</u>
P =      <p>$0</p>
DL =     <dl>$0</dl>
DT =     <dt>$0</dt>
DD =     <dd>$0</dd>
TABLE =  <table>$0</table>
TR =     <tr>$0</tr>
TH =     <th>$0</th>
TD =     <td>$0</td>
OL =     <ol>$0</ol>
UL =     <ul>$0</ul>
LI =     <li>$0</li>
BIG =    <big>$0</big>
SMALL =  <small>$0</small>
BR =     <br>
LINK =   <a href="$0">$0</a>
LINK2 =  <a href="$1">$+</a>

RED =    <font color=red>$0</font>
BLUE =   <font color=blue>$0</font>
```

```
GREEN = <font color=green>$0</font>
YELLOW =<font color=yellow>$0</font>
BLACK = <font color=black>$0</font>
WHITE = <font color=white>$0</font>

D_CODE = <pre class="d_code">$0</pre>
D_COMMENT = $(GREEN $0)
D_STRING  = $(RED $0)
D_KEYWORD = $(BLUE $0)
D_PSYMBOL = $(U $0)
D_PARAM   = $(I $0)

DDOC =  <html><head>
        <META http-equiv="content-type" content="text/html; charset=utf-8">
        <title>$(TITLE)</title>
        </head><body>
        <h1>$(TITLE)</h1>
        $(BODY)
        </body></html>

DDOC_COMMENT   = <!-- $0 -->
DDOC_DECL      = $(DT $(BIG $0))
DDOC_DECL_DD   = $(DD $0)
DDOC_DITTO     = $(BR)$0
DDOC_SECTIONS  = $0
DDOC_SUMMARY   = $0$(BR)$(BR)
DDOC_DESCRIPTION = $0$(BR)$(BR)
DDOC_AUTHORS   = $(B Authors:)$(BR)
                 $0$(BR)$(BR)
DDOC_BUGS      = $(RED BUGS:)$(BR)
                 $0$(BR)$(BR)
DDOC_COPYRIGHT = $(B Copyright:)$(BR)
                 $0$(BR)$(BR)
DDOC_DATE      = $(B Date:)$(BR)
                 $0$(BR)$(BR)
DDOC_DEPRECATED = $(RED Deprecated:)$(BR)
                 $0$(BR)$(BR)
DDOC_EXAMPLES  = $(B Examples:)$(BR)
                 $0$(BR)$(BR)
DDOC_HISTORY   = $(B History:)$(BR)
                 $0$(BR)$(BR)
DDOC_LICENSE   = $(B License:)$(BR)
                 $0$(BR)$(BR)
DDOC_RETURNS   = $(B Returns:)$(BR)
                 $0$(BR)$(BR)
DDOC_SEE_ALSO  = $(B See Also:)$(BR)
                 $0$(BR)$(BR)
DDOC_STANDARDS = $(B Standards:)$(BR)
                 $0$(BR)$(BR)
DDOC_THROWS    = $(B Throws:)$(BR)
                 $0$(BR)$(BR)
DDOC_VERSION   = $(B Version:)$(BR)
                 $0$(BR)$(BR)
DDOC_SECTION_H = $(B $0)$(BR)$(BR)
DDOC_SECTION   = $0$(BR)$(BR)
DDOC_MEMBERS   = $(DL $0)
DDOC_MODULE_MEMBERS   = $(DDOC_MEMBERS $0)
DDOC_CLASS_MEMBERS    = $(DDOC_MEMBERS $0)
DDOC_STRUCT_MEMBERS   = $(DDOC_MEMBERS $0)
DDOC_ENUM_MEMBERS     = $(DDOC_MEMBERS $0)
DDOC_TEMPLATE_MEMBERS = $(DDOC_MEMBERS $0)
```

```
DDOC_PARAMS    = $(B Params:)$(BR)\n$(TABLE $0)$(BR)
DDOC_PARAM_ROW = $(TR $0)
DDOC_PARAM_ID  = $(TD $0)
DDOC_PARAM_DESC  = $(TD $0)
DDOC_BLANKLINE  = $(BR)$(BR)

DDOC_PSYMBOL    = $(U $0)
DDOC_KEYWORD    = $(B $0)
DDOC_PARAM      = $(I $0)
```

Ddoc does not generate HTML code. It formats into the basic formatting macros, which (in their predefined form) are then expanded into HTML. If output other than HTML is desired, then these macros need to be redefined.

Basic Formatting Macros

B	boldface the argument
I	italicize the argument
U	underline the argument
P	argument is a paragraph
DL	argument is a definition list
DT	argument is a definition in a definition list
DD	argument is a description of a definition
TABLE	argument is a table
TR	argument is a row in a table
TH	argument is a header entry in a row
TD	argument is a data entry in a row
OL	argument is an ordered list
UL	argument is an unordered list
LI	argument is an item in a list
BIG	argument is one font size bigger
SMALL	argument is one font size smaller
BR	start new line
LINK	generate clickable link on argument
LINK2	generate clickable link, first arg is address
RED	argument is set to be red
BLUE	argument is set to be blue
GREEN	argument is set to be green
YELLOW	argument is set to be yellow
BLACK	argument is set to be black
WHITE	argument is set to be white
D_CODE	argument is D code
DDOC	overall template for output

DDOC is special in that it specifies the boilerplate into which the entire generated text is inserted (represented by the Ddoc generated macro **BODY**). For example, in order to use a style sheet, **DDOC** would be redefined as:

```
DDOC =  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
```

```
<html><head>
<META http-equiv="content-type" content="text/html; charset=utf-8">
<title>$(TITLE)</title>
<link rel="stylesheet" type="text/css" href="style.css">
</head><body>
<h1>$(TITLE)</h1>
$(BODY)
</body></html>
```

DDOC_COMMENT is used to insert comments into the output file.

Highlighting of D code is performed by the following macros:

D Code Formatting Macros

D_COMMENT	Highlighting of comments
D_STRING	Highlighting of string literals
D_KEYWORD	Highlighting of D keywords
D_PSYMBOL	Highlighting of current declaration name
D_PARAM	Highlighting of current function declaration parameters

The highlighting macros start with **DDOC_**. They control the formatting of individual parts of the presentation.

Ddoc Section Formatting Macros

DDOC_DECL	Highlighting of the declaration.
DDOC_DECL_DD	Highlighting of the description of a declaration.
DDOC_DITTO	Highlighting of ditto declarations.
DDOC_SECTIONS	Highlighting of all the sections.
DDOC_SUMMARY	Highlighting of the summary section.
DDOC_DESCRIPTION	Highlighting of the description section.
DDOC_AUTHORS .. DDOC_VERSION	Highlighting of the corresponding standard section.
DDOC_SECTION_H	Highlighting of the section name of a non-standard section.
DDOC_SECTION	Highlighting of the contents of a non-standard section.
DDOC_MEMBERS	Default highlighting of all the members of a class, struct, etc.
DDOC_MODULE_MEMBERS	Highlighting of all the members of a module.
DDOC_CLASS_MEMBERS	Highlighting of all the members of a class.
DDOC_STRUCT_MEMBERS	Highlighting of all the members of a struct.
DDOC_ENUM_MEMBERS	Highlighting of all the members of an enum.
DDOC_TEMPLATE_MEMBERS	Highlighting of all the members of a template.
DDOC_PARAMS	Highlighting of a function parameter section.
DDOC_PARAM_ROW	Highlighting of a name=value function parameter.
DDOC_PARAM_ID	Highlighting of the parameter name.
DDOC_PARAM_DESC	Highlighting of the parameter value.
DDOC_PSYMBOL	Highlighting of declaration name to which a particular section is referring.
DDOC_KEYWORD	Highlighting of D keywords.

DDOC_PARAM	Highlighting of function parameters.
DDOC_BLANKLINE	Inserts a blank line.

For example, one could redefine **DDOC_SUMMARY**:

```
DDOC_SUMMARY = $(GREEN $0)
```

And all the summary sections will now be green.

Macro Definitions from sc.ini's DDOCFILE

A text file of macro definitions can be created, and specified in sc.ini:

```
DDOCFILE=myproject.ddoc
```

Macro Definitions from .ddoc Files on the Command Line

File names on the DMD command line with the extension .ddoc are text files that are read and processed in order.

Macro Definitions Generated by Ddoc

BODY	Set to the generated document text.
TITLE	Set to the module name.
DATETIME	Set to the current date and time.
YEAR	Set to the current year.
COPYRIGHT	Set to the contents of any **Copyright:** section that is part of the module comment.

Using Ddoc for other Documentation

Ddoc is primarily designed for use in producing documentation from embedded comments. It can also, however, be used for processing other general documentation. The reason for doing this would be to take advantage of the macro capability of Ddoc and the D code syntax highlighting capability.

If the .d source file starts with the string "Ddoc" then it is treated as general purpose documentation, not as a D code source file. From immediately after the "Ddoc" string to the end of the file or any "Macros:" section forms the document. No automatic highlighting is done to that text, other than highlighting of D code embedded between lines delineated with --- lines. Only macro processing is done.

Much of the D documentation itself is generated this way, including this page. Such documentation is marked at the bottom as being generated by Ddoc.

References

CandyDoc is a very nice example of how one can customize the Ddoc results with macros and style sheets.

Interfacing to C

D is designed to fit comfortably with a C compiler for the target system. D makes up for not having its own VM by relying on the target environment's C runtime library. It would be senseless to attempt to port to D or write D wrappers for the vast array of C APIs available. How much easier it is to just call them directly.

This is done by matching the C compiler's data types, layouts, and function call/return sequences.

Calling C Functions

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

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

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

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

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

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. However, string literals in D are 0 terminated.

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

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

Storage Allocation

C code explicitly manages memory with calls to malloc() and free(). D allocates memory using the D garbage collector, so no explicit free's are necessary.

D can still explicitly allocate memory using c.stdlib.malloc() and c.stdlib.free(), these are useful for connecting to C functions that expect malloc'd buffers, etc.

If pointers to D garbage collector allocated memory are passed to C functions, it's critical to ensure that that memory will not be collected by the garbage collector before the C function is done with it. This is accomplished by:

- Making a copy of the data using c.stdlib.malloc() and passing the copy instead.
- Leaving a pointer to it on the stack (as a parameter or automatic variable), as the garbage collector will scan the stack.
- Leaving a pointer to it in the static data segment, as the garbage collector will scan the static data segment.
- Registering the pointer with the garbage collector with the gc.addRoot() or gc.addRange() calls.

An interior pointer to the allocated memory block is sufficient to let the GC know the object is in use; i.e. it is not necessary to maintain a pointer to the beginning of the allocated memory.

The garbage collector does not scan the stacks of threads not created by the D Thread interface. Nor does it scan the data segments of other DLL's, etc.

Data Type Compatibility

D type	C type
void	**void**
bit	no equivalent
byte	**signed char**
ubyte	**unsigned char**
char	**char** (chars are unsigned in D)
wchar	**wchar_t** (when sizeof(wchar_t) is 2)
dchar	**wchar_t** (when sizeof(wchar_t) is 4)
short	**short**
ushort	**unsigned short**
int	**int**
uint	**unsigned**
long	**long long**
ulong	**unsigned long long**
float	**float**
double	**double**
real	**long double**

ifloat	**float _Imaginary**
idouble	**double _Imaginary**
ireal	**long double _Imaginary**
cfloat	**float _Complex**
cdouble	**double _Complex**
creal	**long double _Complex**
struct	**struct**
union	**union**
enum	**enum**
class	no equivalent
*type******	*type* *****
*type***[**dim**]**	*type***[**dim**]**
*type***[**dim**]***	*type***(*)[**dim**]**
*type***[]**	no equivalent
*type***[**type**]**	no equivalent
type **function(**parameters**)**	*type***(*)(**parameters**)**
type **delegate(**parameters**)**	no equivalent

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

Calling printf()

This mostly means checking that the printf format specifier matches the corresponding D data type. Although printf is designed to handle 0 terminated strings, not D dynamic arrays of chars, it turns out that since D dynamic arrays are a length followed by a pointer to the data, the `%.*s` format works perfectly:

```
void foo(char[] string)
{
    printf("my string is: %.*s\n", string);
}
```

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.

An improved D function for formatted output is `std.stdio.writef()`.

Structs and Unions

D structs and unions are analogous to C's.

C code often adjusts the alignment and packing of struct members with a command line switch or with various implementation specific #pragma's. D supports explicit alignment attributes that correspond to the C compiler's rules. Check what alignment the C code is using, and explicitly set it for the D struct declaration.

D does not support bit fields. If needed, they can be emulated with shift and mask operations.

Interfacing to C++

D does not provide an interface to C++. Since D, however, interfaces directly to C, it can interface directly to C++ code if it is declared as having C linkage.

D class objects are incompatible with C++ class objects.

Portability Guide

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

- The integral and floating type sizes should be considered as minimums. Algorithms should be designed to continue to work properly if the type size increases.
- Floating point computations can be carried out at a higher precision than the size of the floating point variable can hold. Floating point algorithms should continue to work properly if precision is arbitrarily increased.
- Avoid depending on the order of side effects in a computation that may get reordered by the compiler. For example:

  ```
  a + b + c
  ```

  can be evaluated as (a + b) + c, a + (b + c), (a + c) + b, (c + b) + a, etc. Parentheses control operator precedence, parentheses do not control order of evaluation.

  Function parameters can be evaluated either left to right or right to left, depending on the particular calling conventions used.

  If the operands of an associative operator + or * are floating point values, the expression is not reordered.
- Avoid dependence on byte order; i.e. whether the CPU is big-endian or little-endian.
- Avoid dependence on the size of a pointer or reference being the same size as a particular integral type.
- If size dependencies are inevitable, put an `assert` in the code to verify it:

  ```
  assert(int.sizeof == (int*).sizeof);
  ```

32 to 64 Bit Portability

64 bit processors and operating systems are here. 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. Array indices should be of type **size_t**.
- Use **ptrdiff_t** as an alias for a signed integral type that can span the address space. A type representing the difference between two pointers should be of type **ptrdiff_t**.
- The **.length**, **.size**, **.sizeof**, and **.alignof** properties will be of type **size_t**.

Endianness

Endianness refers to the order in which multibyte types are stored. The two main orders are *big endian* and *little endian*. The compiler predefines the version identifier **BigEndian** or **LittleEndian** depending on the order of the target system. The x86 systems are all little endian.

The times when endianness matters are:

- When reading data from an external source (like a file) written in a different endian format.
- When reading or writing individual bytes of a multibyte type like **long**s or **double**s.

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 *IfStatement* or *StaticIfStatement*.

Embedding D in HTML

The D compiler is designed to be able to extract and compile D code embedded within HTML files. This capability means that D code can be written to be displayed within a browser utilizing the full formatting and display capability of HTML.

For example, it is possible to make all uses of a class name actually be hyperlinks to where the class is defined. There's nothing new to learn for the person browsing the code, he just uses the normal features of an HTML browser. Strings can be displayed in green, comments in red, and keywords in **boldface**, for one possibility. It is even possible to embed pictures in the code, as normal HTML image tags.

Embedding D in HTML makes it possible to put the documentation for code and the code itself all together in one file. It is no longer necessary to relegate documentation in comments, to be extracted later by a tech writer. The code and the documentation for it can be maintained simultaneously, with no duplication of effort.

How it works is straightforward. If the source file to the compiler ends in .htm or .html, the code is assumed to be embedded in HTML. The source is then preprocessed by stripping all text outside of <code> and </code> tags. Then, all other HTML tags are stripped, and embedded character encodings are converted to ASCII. The processing does not attempt to diagnose errors in the HTML itself. All newlines in the original HTML remain in their corresponding positions in the preprocessed text, so the debug line numbers remain consistent. The resulting text is then fed to the D compiler.

Here's an example of the D program "hello world" embedded in this very HTML file. This file can be compiled and run.

```
import std.c.stdio;

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

Named Character Entities

These are the character entity names supported by D.

Note: Not all will display properly in the **Symbol** column in all browsers.

Named Character Entities

Name	Value	Symbol
quot	34	"
amp	38	&
lt	60	<
gt	62	>
OElig	338	Œ
oelig	339	œ
Scaron	352	Š
scaron	353	š
Yuml	376	Ÿ
circ	710	ˆ
tilde	732	˜
ensp	8194	
emsp	8195	
thinsp	8201	
zwnj	8204	
zwj	8205	
lrm	8206	
rlm	8207	
ndash	8211	–
mdash	8212	—
lsquo	8216	‘
rsquo	8217	’
sbquo	8218	‚
ldquo	8220	“
rdquo	8221	”
bdquo	8222	„
dagger	8224	†
Dagger	8225	‡

permil	8240	‰
lsaquo	8249	‹
rsaquo	8250	›
euro	8364	€
Latin-1 (ISO-8859-1) Entities		
nbsp	160	
iexcl	161	¡
cent	162	¢
pound	163	£
curren	164	¤
yen	165	¥
brvbar	166	¦
sect	167	§
uml	168	¨
copy	169	©
ordf	170	ª
laquo	171	«
not	172	¬
shy	173	
reg	174	®
macr	175	¯
deg	176	°
plusmn	177	±
sup2	178	²
sup3	179	³
acute	180	´
micro	181	µ
para	182	¶
middot	183	·
cedil	184	¸
sup1	185	¹
ordm	186	º
raquo	187	»

frac14	188	¼
frac12	189	½
frac34	190	¾
iquest	191	¿
Agrave	192	À
Aacute	193	Á
Acirc	194	Â
Atilde	195	Ã
Auml	196	Ä
Aring	197	Å
AElig	198	Æ
Ccedil	199	Ç
Egrave	200	È
Eacute	201	É
Ecirc	202	Ê
Euml	203	Ë
Igrave	204	Ì
Iacute	205	Í
Icirc	206	Î
Iuml	207	Ï
ETH	208	Đ
Ntilde	209	Ñ
Ograve	210	Ò
Oacute	211	Ó
Ocirc	212	Ô
Otilde	213	Õ
Ouml	214	Ö
times	215	×
Oslash	216	Ø
Ugrave	217	Ù
Uacute	218	Ú
Ucirc	219	Û
Uuml	220	Ü

Yacute	221	Ý
THORN	222	Þ
szlig	223	ß
agrave	224	à
aacute	225	á
acirc	226	â
atilde	227	ã
auml	228	ä
aring	229	å
aelig	230	æ
ccedil	231	ç
egrave	232	è
eacute	233	é
ecirc	234	ê
euml	235	ë
igrave	236	ì
iacute	237	í
icirc	238	î
iuml	239	ï
eth	240	ð
ntilde	241	ñ
ograve	242	ò
oacute	243	ó
ocirc	244	ô
otilde	245	õ
ouml	246	ö
divide	247	÷
oslash	248	ø
ugrave	249	ù
uacute	250	ú
ucirc	251	û
uuml	252	ü
yacute	253	ý

thorn	254	þ
yuml	255	ÿ
Symbols and Greek letter entities		
fnof	402	*f*
Alpha	913	Α
Beta	914	Β
Gamma	915	Γ
Delta	916	Δ
Epsilon	917	Ε
Zeta	918	Ζ
Eta	919	Η
Theta	920	Θ
Iota	921	Ι
Kappa	922	Κ
Lambda	923	Λ
Mu	924	Μ
Nu	925	Ν
Xi	926	Ξ
Omicron	927	Ο
Pi	928	Π
Rho	929	Ρ
Sigma	931	Σ
Tau	932	Τ
Upsilon	933	Υ
Phi	934	Φ
Chi	935	Χ
Psi	936	Ψ
Omega	937	Ω
alpha	945	α
beta	946	β
gamma	947	γ
delta	948	δ
epsilon	949	ε

zeta	950	ζ
eta	951	η
theta	952	θ
iota	953	ι
kappa	954	κ
lambda	955	λ
mu	956	μ
nu	957	ν
xi	958	ξ
omicron	959	ο
pi	960	π
rho	961	ρ
sigmaf	962	ς
sigma	963	σ
tau	964	τ
upsilon	965	υ
phi	966	φ
chi	967	χ
psi	968	ψ
omega	969	ω
thetasym	977	ϑ
upsih	978	ϒ
piv	982	ϖ
bull	8226	•
hellip	8230	…
prime	8242	′
Prime	8243	″
oline	8254	‾
frasl	8260	⁄
weierp	8472	℘
image	8465	ℑ
real	8476	ℜ
trade	8482	™

alefsym	8501	ℵ
larr	8592	←
uarr	8593	↑
rarr	8594	→
darr	8595	↓
harr	8596	↔
crarr	8629	↵
lArr	8656	⇐
uArr	8657	⇑
rArr	8658	⇒
dArr	8659	⇓
hArr	8660	⇔
forall	8704	∀
part	8706	∂
exist	8707	∃
empty	8709	∅
nabla	8711	∇
isin	8712	∈
notin	8713	∉
ni	8715	∋
prod	8719	∏
sum	8721	∑
minus	8722	−
lowast	8727	*
radic	8730	√
prop	8733	∝
infin	8734	∞
ang	8736	∠
and	8743	∧
or	8744	∨
cap	8745	∩
cup	8746	∪
int	8747	∫

there4	8756	∴
sim	8764	~
cong	8773	≅
asymp	8776	≈
ne	8800	≠
equiv	8801	≡
le	8804	≤
ge	8805	≥
sub	8834	⊂
sup	8835	⊃
nsub	8836	⊄
sube	8838	⊆
supe	8839	⊇
oplus	8853	⊕
otimes	8855	⊗
perp	8869	⊥
sdot	8901	⋅
lceil	8968	⌈
rceil	8969	⌉
lfloor	8970	⌊
rfloor	8971	⌋
lang	9001	〈
rang	9002	〉
loz	9674	◊
spades	9824	♠
clubs	9827	♣
hearts	9829	♥
diams	9830	♦

D Application Binary Interface

A D implementation that conforms to the D ABI (Application Binary Interface) will be able to generate libraries, DLL's, etc., that can interoperate with D binaries built by other implementations.

Most of this specification remains TBD (To Be Defined).

C ABI

The C ABI referred to in this specification means the C Application Binary Interface of the target system. C and D code should be freely linkable together, in particular, D code shall have access to the entire C ABI runtime library.

Basic Types

TBD

Structs

Conforms to the target's C ABI struct layout.

Classes

An object consists of:

offset	contents
0	pointer to vtable
ptrsize	monitor
*ptrsize*2...*	non-static members

The vtable consists of:

offset	contents
0	pointer to instance of ClassInfo
ptrsize...	pointers to virtual member functions

The class definition:

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

Generates the following:

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

Interfaces

TBD

Arrays

A dynamic array consists of:

offset	contents
0	array dimension
size_t	pointer to array data

A dynamic array is declared as:

```
type[] array;
```

whereas a static array is declared as:

```
type[dimension] array;
```

Thus, a static array always has the dimension statically available as part of the type, and so it is implemented like in C. Static array's and Dynamic arrays can be easily converted back and forth to each other.

Associative Arrays

Associative arrays consist of a pointer to an opaque, implementation defined type. The current implementation is contained in phobos/internal/aaA.d.

Reference Types

D has reference types, but they are implicit. For example, classes are always referred to by reference; this means that class instances can never reside on the stack or be passed as function parameters.

When passing a static array to a function, the result, although declared as a static array, will actually be a reference to a static array. For example:

```
int[3] abc;
```

Passing abc to functions results in these implicit conversions:

```
void func(int[3] array); // actually <reference to><array[3] of><int>
void func(int* p);       // abc is converted to a pointer
                         // to the first element
void func(int[] array);  // abc is converted to a dynamic array
```

Name Mangling

D accomplishes typesafe linking by *mangling* a D identifier to include scope and type information.

```
MangledName:
    _D QualifiedName Type
    _D QualifiedName M Type

QualifiedName:
```

```
    SymbolName
    SymbolName QualifiedName

SymbolName:
    LName
    TemplateInstanceName
```

The **M** means that the symbol is a function that requires a `this` pointer.

Template Instance Names have the types and values of its parameters encoded into it:

```
TemplateInstanceName:
     __T LName TemplateArgs Z

TemplateArgs:
    TemplateArg
    TemplateArg TemplateArgs

TemplateArg:
    T Type
    V Type Value
    S LName

Value:
    n
    Number
    N Number
    e HexFloat
    c HexFloat c HexFloat
    A Number Value...

HexFloat:
    NAN
    INF
    NINF
    N HexDigits P Exponent
    HexDigits P Exponent

Exponent:
    N Number
    Number

HexDigits:
    HexDigit
    HexDigit HexDigits

HexDigit:
    Digit
    A
    B
    C
    D
    E
    F
```

n
: is for **null** arguments.

Number
: is for positive numeric literals (including character literals).

N *Number*

is for negative numeric literals.

e *HexFloat*
is for real and imaginary floating point literals.

c *HexFloat* **c** *HexFloat*
is for complex floating point literals.

Width Number **_** *HexDigits*
Width is whether the characters are 1 byte (**a**), 2 bytes (**w**) or 4 bytes (**d**) in size. *Number* is the number of characters in the string. The *HexDigits* are the hex data for the string.

A *Number Value...*
An array literal. *Value* is repeated *Number* times.

```
Name:
    Namestart
    Namestart Namechars

Namestart:
    _
    Alpha

Namechar:
    Namestart
    Digit

Namechars:
    Namechar
    Namechar Namechars
```

A *Name* is a standard D identifier.

```
LName:
    Number Name

Number:
    Digit
    Digit Number

Digit:
    0
    1
    2
    3
    4
    5
    6
    7
    8
    9
```

An *LName* is a name preceded by a *Number* giving the number of characters in the *Name*.

Type Mangling

Types are mangled using a simple linear scheme:

```
Type:
    TypeArray
    TypeSarray
    TypeAarray
```

```
    TypePointer
    TypeFunction
    TypeIdent
    TypeClass
    TypeStruct
    TypeEnum
    TypeTypedef
    TypeDelegate
    TypeNone
    TypeVoid
    TypeByte
    TypeUbyte
    TypeShort
    TypeUshort
    TypeInt
    TypeUint
    TypeLong
    TypeUlong
    TypeFloat
    TypeDouble
    TypeReal
    TypeIfloat
    TypeIdouble
    TypeIreal
    TypeCfloat
    TypeCdouble
    TypeCreal
    TypeBool
    TypeChar
    TypeWchar
    TypeDchar
    TypeTuple

TypeArray:
    A Type

TypeSarray:
    G Number Type

TypeAarray:
    H Type Type

TypePointer:
    P Type

TypeFunction:
    CallConvention Arguments ArgClose Type

CallConvention:
    F
    U
    W
    V
    R

Arguments:
    Argument
    Argument Arguments

Argument:
    Type
```

```
    J Type
    K Type
    L Type

ArgClose
    X
    Y
    Z

TypeIdent:
    I LName

TypeClass:
    C LName

TypeStruct:
    S LName

TypeEnum:
    E LName

TypeTypedef:
    T LName

TypeDelegate:
    D TypeFunction

TypeNone:
    n

TypeVoid:
    v

TypeByte:
    g

TypeUbyte:
    h

TypeShort:
    s

TypeUshort:
    t

TypeInt:
    i

TypeUint:
    k

TypeLong:
    l

TypeUlong:
    m

TypeFloat:
    f

TypeDouble:
```

```
    d

TypeReal:
    e

TypeIfloat:
    o

TypeIdouble:
    p

TypeIreal:
    j

TypeCfloat:
    q

TypeCdouble:
    r

TypeCreal:
    c

TypeBool:
    b

TypeChar:
    a

TypeWchar:
    u

TypeDchar:
    w

TypeTuple:
    B Number Arguments
```

Function Calling Conventions

The extern (C) calling convention matches the C calling convention used by the supported C compiler on the host system. The extern (D) calling convention for x86 is described here.

Register Conventions

- EAX, ECX, EDX are scratch registers and can be destroyed by a function.
- EBX, ESI, EDI, EBP must be preserved across function calls.
- EFLAGS is assumed destroyed across function calls, except for the direction flag which must be forward.
- The FPU stack must be empty when calling a function.
- The FPU control word must be preserved across function calls.
- Floating point return values are returned on the FPU stack. These must be cleaned off by the caller, even if they are not used.

Return Value

- The types bool, byte, ubyte, short, ushort, int, uint, pointer, Object, and interfaces are

returned in EAX.
- long and ulong are returned in EDX,EAX, where EDX gets the most significant half.
- float, double, real, ifloat, idouble, ireal are returned in ST0.
- cfloat, cdouble, creal are returned in ST1,ST0 where ST1 is the real part and ST0 is the imaginary part.
- Dynamic arrays are returned with the pointer in EDX and the length in EAX.
- Associative arrays are returned in EAX with garbage returned in EDX. The EDX value will probably be removed in the future; it's there for backwards compatibility with an earlier implementation of AA's.
- Delegates are returned with the pointer to the function in EDX and the context pointer in EAX.
- For Windows, 1, 2 and 4 byte structs are returned in EAX.
- For Windows, 8 byte structs are returned in EDX,EAX, where EDX gets the most significant half.
- For other struct sizes, and for all structs on Linux, the return value is stored through a hidden pointer passed as an argument to the function.
- Constructors return the this pointer in EAX.

Parameters

The parameters to the non-variadic function:

```
foo(a1, a2, ..., an);
```

are passed as follows:

a1

a2

...

an

hidden

this

where *hidden* is present if needed to return a struct value, and *this* is present if needed as the this pointer for a member function or the context pointer for a nested function.

The last parameter is passed in EAX rather than being pushed on the stack if the following conditions are met:

- It fits in EAX.
- It is not a 3 byte struct.

Parameters are always pushed as multiples of 4 bytes, rounding upwards, so the stack is always aligned on 4 byte boundaries. They are pushed most significant first. **out** and **inout** are passed as pointers. Static arrays are passed as pointers to their first element. On Windows, a real is pushed as a 10 byte quantity, a creal is pushed as a 20 byte quantity. On Linux, a real is pushed as a 12 byte quantity, a creal is pushed as two 12 byte quantities. The extra two bytes of pad occupy the 'most significant' position.

The callee cleans the stack.

The parameters to the variadic function:

```
void foo(int p1, int p2, int[] p3...)
foo(a1, a2, ..., an);
```

are passed as follows:

p1
p2
a3
hidden
this

The variadic part is converted to a dynamic array and the rest is the same as for non-variadic functions.

The parameters to the variadic function:

```
void foo(int p1, int p2, ...)
foo(a1, a2, a3, ..., an);
```

are passed as follows:

an
...
a3
a2
a1
_arguments
hidden
this

The caller is expected to clean the stack. **_argptr** is not passed, it is computed by the callee.

Exception Handling

Windows

Conforms to the Microsoft Windows Structured Exception Handling conventions.

Linux

Uses static address range/handler tables. TBD

Garbage Collection

The interface to this is found in `phobos/internal/gc`.

Runtime Helper Functions

These are found in `phobos/internal`.

Module Initialization and Termination

TBD

Unit Testing

TBD