D Programming Language Specification

This is the specification for the D Programming Language. For more information see dlang.org.

Contents

Chapter 1

Introduction

D is a general-purpose systems programming language with a C-like syntax that compiles to native code. It is statically typed and supports both automatic (garbage collected) and manual memory management. D programs are structured as modules that can be compiled separately and linked with external libraries to create native libraries or executables.

This document is the reference manual for the D Programming Language. For more information and other documents, see The D Language Website.

1.1 Phases of Compilation

The process of compiling is divided into multiple phases. Each phase is independent of subsequent phases. For example, the scanner is not affected by the semantic analyzer. This separation of passes makes language tools like syntax directed editors relatively easy to create. It is also possible to compress D source by storing it in 'tokenized' form.

1. **source character set**
   The source file is checked to see which encoding it is using, and the appropriate scanner is loaded. 7-bit ASCII and UTF encodings are accepted.
2. **script line**
   If the first line starts with "#!", then that line is ignored.
3. **lexical analysis**
   The source file is divided up into a sequence of tokens. Special tokens are replaced with other tokens. *SpecialTokenSequence*s 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.

1.2 Memory Model

A *byte* is the fundamental unit of storage. Each byte has 8 bits, and is stored at a unique address. A *memory location* is a sequence of one or more bytes of the exact size required to hold a scalar type. Multiple threads can access separate memory locations without interference.

Memory locations come in three groups:

1. *Thread local memory locations* are accessible from only one thread at a time.
2. *Immutable memory locations* cannot be written to during their lifetime. Immutable memory locations can be read from by multiple threads without synchronization.
3. *Shared memory locations* are accessible from multiple threads.

Undefined Behavior: Allowing multiple threads to access a thread local memory location results in undefined behavior.

Undefined Behavior: Writing to an immutable memory location during its lifetime results in undefined behavior.

Undefined Behavior: Unless synchronized, writing to a shared memory location in one thread while other threads read or write to it results in undefined behavior.

Execution of a single thread on thread local and immutable memory locations is *sequentially consistent*. This means the result of the operations is the same as if they were executed in the same order that the operations appear in the program.

A memory location can be transferred from thread local to immutable or shared if there is only one reference to the location.

A memory location can be transferred from shared to immutable or thread local if there is only one reference to the location.

A memory location can be temporarily transferred from shared to local if synchronization is used to prevent any other threads from accessing the memory location during the operation.

1.3 Object Model

An *object* is created when one of the following is executed:

- a definition
- a *NewExpression*
- a temporary is created
- changing which field of a union is active

An object spans a sequence of memory locations which may or may not be contiguous. It exists during construction, its lifetime, and destruction. Each object has a type, which is determined either statically or by runtime type information. The memory locations may include any combination of thread local, immutable, or shared memory locations.

Objects can be composed into a *composed object*. The objects that make up the composed object are *subobjects*. An object that is not the subobject of another object is a *complete object*. The lifetime of a subobject is always within the lifetime of its complete object.

An object's address is the address of the first byte of the first memory location for that object. Object addresses are distinct from each other unless one of the objects is nested within the other.

Chapter 2

Lexical

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 the syntax of those tokens. The grammar is designed to be suitable for high speed scanning and to make it easy to write a correct scanner for it. It has a minimum of special case rules and there is only one phase of translation.

2.1 Source Text

Source text can be in one of the following formats:

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

One of the following UTF BOMs (Byte Order Marks) can be present at the beginning of the source text:

UTF Byte Order Marks

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 U+0000007F.

The source text is decoded from its source representation into Unicode *Character*s. The *Character*s are further divided into: *WhiteSpace*, *EndOfLine*, *Comment*s, *SpecialTokenSequence*s, *Token*s, all followed by *EndOfFile*.

The source text is split into tokens using the maximal munch technique, i.e., the lexical analyzer makes the longest token it can. For example `>>` is a right shift token, not two greater than tokens. There are two exceptions to this rule:

- A `..` embedded inside what looks like two floating point literals, as in `1..2`, is interpreted as if the `..` was separated by a space from the first integer.
- A `1.a` is interpreted as the three tokens `1`, `.`, and `a`, whereas `1. a` is interpreted as the two tokens `1.` and `a`.

2.2 Character Set

```
Character:
    any Unicode character
```

2.3 End of File

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

The source text is terminated by whichever comes first.

2.4 End of Line

EndOfLine:
```
\u000D
\u000A
\u000D \u000A
\u2028
\u2029
```
EndOfFile

2.5 White Space

WhiteSpace:
Space
Space WhiteSpace

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

2.6 Comments

Comment:
BlockComment
LineComment
NestingBlockComment

BlockComment:
/* *Characters* */

LineComment:
// *Characters EndOfLine*

NestingBlockComment:
  `/+` *NestingBlockCommentCharacters* `+/`

NestingBlockCommentCharacters:
  NestingBlockCommentCharacter
  NestingBlockCommentCharacter *NestingBlockCommentCharacters*

NestingBlockCommentCharacter:
  Character
  NestingBlockComment

Characters:
  Character
  Character *Characters*

There are 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 block 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.

2.7 Tokens

Token:

```
Identifier          ||       >>>     *
StringLiteral       -        !       *=
CharacterLiteral    -=       !=      %
IntegerLiteral      --       (       %=
FloatLiteral        +        )       ^
Keyword             +=       [       ^=
/                   ++       ]       ^^
/=                  <        {       ^^=
.                   <=       }       ~
..                  <<       ?       ~=
...                 <<=      ,       @
&                   >        ;       =>
&=                  >=       :       #
&&                  >>=      $
|                   >>>=     =
=|                  >>       ==
```

2.8 Identifiers

Identifier:
  IdentifierStart
  IdentifierStart IdentifierChars

IdentifierChars:
  IdentifierChar
  IdentifierChar IdentifierChars

IdentifierStart:
  _
  Letter
  UniversalAlpha

IdentifierChar:
  IdentifierStart
  `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 of the C99 Standard. Identifiers can be arbitrarily long, and are case sensitive.

Implementation Defined: Identifiers starting with `__` (two underscores) are reserved.

2.9 String Literals

StringLiteral:
  WysiwygString
  AlternateWysiwygString
  DoubleQuotedString

  HexString
  DelimitedString
  TokenString

WysiwygString:
  `r"` *WysiwygCharacters* `"` *StringPostfix*$_{\text{opt}}$

AlternateWysiwygString:
  `` ` `` *WysiwygCharacters* `` ` `` *StringPostfix*$_{\text{opt}}$

WysiwygCharacters:
  WysiwygCharacter
  WysiwygCharacter *WysiwygCharacters*

WysiwygCharacter:
  Character

EndOfLine

DoubleQuotedString:
" *DoubleQuotedCharacters* " *StringPostfix*$_{\text{opt}}$

DoubleQuotedCharacters:
DoubleQuotedCharacter
DoubleQuotedCharacter DoubleQuotedCharacters

DoubleQuotedCharacter:
Character
EscapeSequence
EndOfLine

EscapeSequence:
\'
\"
\?
\\
\0
\a
\b
\f
\n
\r
\t
\v
\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 " StringPostfix_opt

HexStringChars:
    HexStringChar
    HexStringChar HexStringChars

HexStringChar:
    HexDigit
    WhiteSpace
    EndOfLine

StringPostfix:
    c
    w
    d

DelimitedString:
    q" Delimiter WysiwygCharacters MatchingDelimiter "

TokenString:
    q{ Tokens }
```

A string literal is either a double quoted string, a wysiwyg quoted string, a delimited string, a token string, or a hex string.

In all string literal forms, an *EndOfLine* is regarded as a single `\n` character.

String literals are read only.

Undefined Behavior: Writes to string literals cannot always be detected, but cause undefined behavior.

Wysiwyg Strings

Wysiwyg ("what you see is what you get") quoted strings are enclosed by `r"` and `"`. All characters between the `r"` and `"` are part of the string. There are no escape sequences inside wysiwyg strings.

```
r"I␣am␣Oz"
r"c:\games\Sudoku.exe"
r"ab\n" // string is 4 characters,
        // 'a', 'b', '\', 'n'
```

An alternate form of wysiwyg strings are enclosed by backquotes, the ` character.

```
`the Great and Powerful.`
`c:\games\Empire.exe`
`The "lazy" dog`
`a"b\n`␣␣//␣string␣is␣5␣characters,
␣␣␣␣␣␣␣␣␣//␣'a',␣'"',␣'b',␣'\',␣'n'
```

Double Quoted Strings

Double quoted strings are enclosed by "". *EscapeSequence*s can be embedded into them.

```
"Who␣are␣you?"
"c:\\games\\Doom.exe"
"ab\n"   // string is 3 characters,
         // 'a', 'b', and a linefeed
"ab
"        // string is 3 characters,
         // 'a', 'b', and a linefeed
```

Hex Strings

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.

Note: Hex Strings are **deprecated**. Please use `std.conv.hexString` instead.

Delimited Strings

Delimited strings use various forms of delimiters. The delimiter, whether a character or identifer, must immediately follow the " without any intervening whitespace. The terminating delimiter must immediately precede the closing " without any intervening whitespace. A *nesting delimiter* nests, and is one of the following characters:

Nesting Delimiters

Delimiter	Matching Delimiter
[	]
(	)
<	>
{	}

```
q"(foo(xxx))"   // "foo(xxx)"
q"[foo{]"       // "foo{"
```

If the delimiter is an identifier, the identifier must be immediately followed by a newline, and the matching delimiter is the same identifier starting at the beginning of the line:

```
writeln(q"EOS
This
is␣a␣multi-line
heredoc␣string
EOS"
);
```

The newline following the opening identifier is not part of the string, but the last newline before the closing identifier is part of the string. The closing identifier must be placed on its own line at the leftmost column.

Otherwise, the matching delimiter is the same as the delimiter character:

```
q"/foo]/"          // "foo]"
// q"/abc/def/"    // error
```

Token Strings

Token strings open with the characters `q{` and close with the token `}`. In between must be valid D tokens. The `{` and `}` tokens nest. The string is formed of all the characters between the opening and closing of the token string, including comments.

```
q{this is the voice of} // "this is the voice of"
q{/*}*/ }               // "/*}*/ "
q{ world(q{control}); } // " world(q{control}); "
q{ __TIME__ }           // " __TIME__ "
                        // i.e. it is not replaced with the time
// q{ __EOF__ }         // error
                        // __EOF__ is not a token, it's end of file
```

String Postfix

The optional *StringPostfix* character gives a specific type to the string, rather than it being inferred from the context. The types corresponding to the postfix characters are:

String Literal Postfix Characters

Postfix	Type	Aka
c	`immutable(char)[]`	`string`
w	`immutable(wchar)[]`	`wstring`
d	`immutable(dchar)[]`	`dstring`

```
"hello"c  // string
"hello"w  // wstring
"hello"d  // dstring
```

The string literals are assembled as UTF-8 char arrays, and the postfix is applied to convert to wchar or dchar as necessary as a final step.

2.10 Escape Sequences

The escape sequences listed in *EscapeSequence* are:

Table 2.1: Escape Sequences

Sequence	Meaning
`\'`	Literal single-quote: `'`
`\"`	Literal double-quote: `"`
`\?`	Literal question mark: `?`
`\\`	Literal backslash: `\`
`\0`	Binary zero (NUL, U+0000).

↩

(continued)

Sequence	Meaning
`\a`	BEL (alarm) character (U+0007).
`\b`	Backspace (U+0008).
`\f`	Form feed (FF) (U+000C).
`\n`	End-of-line (U+000A).
`\r`	Carriage return (U+000D).
`\t`	Horizontal tab (U+0009).
`\v`	Vertical tab (U+000B).
`\x`*nn*	Byte value in hexadecimal, where *nn* is specified as two hexadecimal digits.For example: `\xFF` represents the character with the value 255.
*n**nn**nnn*	Byte value in octal.For example: `\101` represents the character with the value 65 (`'A'`). Analogous to hexadecimal characters, the largest byte value is `\377` (= `\xFF` in hexadecimal or `255` in decimal)
`\u`*nnnn*	Unicode character U+*nnnn*, where *nnnn* are four hexadecimal digits.For example, `\u03B3` represents the Unicode character γ (U+03B3 - GREEK SMALL LETTER GAMMA).
`\U`*nnnnnnnn*	Unicode character U+*nnnnnnnn*, where *nnnnnnnn* are 8 hexadecimal digits.For example, `\U0001F603` represents the Unicode character U+1F603 (SMILING FACE WITH OPEN MOUTH).
name	Named character entity from the HTML5 specification. See *NamedCharacterEntity*.

2.11 Character Literals

CharacterLiteral:
  ' *SingleQuotedCharacter* '

SingleQuotedCharacter:
  Character
  EscapeSequence

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

```
'h'   // the letter h
'\n'  // newline
'\\'  // the backslash character
```

2.12 Integer Literals

IntegerLiteral:
  Integer
  Integer IntegerSuffix

Integer:
  DecimalInteger
  BinaryInteger
  HexadecimalInteger

IntegerSuffix:
  `L`
  `u`
  `U`
  `Lu`
  `LU`
  `uL`
  `UL`

DecimalInteger:
  `0`
  NonZeroDigit
  NonZeroDigit DecimalDigitsUS

BinaryInteger:
  BinPrefix BinaryDigitsNoSingleUS

BinPrefix:
  `0b`
  `0B`

HexadecimalInteger:
  HexPrefix HexDigitsNoSingleUS

NonZeroDigit:
    1
    2
    3
    4
    5
    6
    7
    8
    9

DecimalDigits:
    DecimalDigit
    DecimalDigit DecimalDigits

DecimalDigitsUS:
    DecimalDigitUS
    DecimalDigitUS DecimalDigitsUS

DecimalDigitsNoSingleUS:
    DecimalDigit
    DecimalDigit DecimalDigitsUS
    DecimalDigitsUS DecimalDigit

DecimalDigitsNoStartingUS:
    DecimalDigit
    DecimalDigit DecimalDigitsUS

DecimalDigit:
    0
    NonZeroDigit

DecimalDigitUS:
  DecimalDigit
  `_`

BinaryDigitsNoSingleUS:
  BinaryDigit
  BinaryDigit BinaryDigitsUS
  BinaryDigitsUS BinaryDigit
  BinaryDigitsUS BinaryDigit BinaryDigitsUS

BinaryDigitsUS:
  BinaryDigitUS
  BinaryDigitUS BinaryDigitsUS

BinaryDigit:
  `0`
  `1`

BinaryDigitUS:
  BinaryDigit
  `_`

OctalDigit:
  `0`
  `1`
  `2`
  `3`
  `4`
  `5`
  `6`
  `7`

HexDigits:
  HexDigit

HexDigit HexDigits

HexDigitsUS:
HexDigitUS
HexDigitUS HexDigitsUS

HexDigitsNoSingleUS:
HexDigit
HexDigit HexDigitsUS
HexDigitsUS HexDigit

HexDigitsNoStartingUS:
HexDigit
HexDigit HexDigitsUS

HexDigit:
DecimalDigit
HexLetter

HexDigitUS:
HexDigit
`_`

HexLetter:
`a`
`b`
`c`
`d`
`e`
`f`
`A`
`B`
`C`
`D`

```
E
F
```

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

Decimal integers are a sequence of decimal digits.

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

C-style octal integer notation was deemed too easy to mix up with decimal notation; it is only fully supported in string literals. D still supports octal integer literals interpreted at compile time through the `std.conv.octal` template, as in `octal!167`.

Hexadecimal integers are a sequence of hexadecimal digits preceded by a '0x' or '0X'.

Integers can have embedded '_' characters, which are ignored.

```
20_000        // leagues under the sea
867_5309      // number on the wall
1_522_000     // thrust of F1 engine (lbf sea level)
```

Integers can be immediately followed by one 'L' or one of 'u' or 'U' or both. Note that there is no 'l' suffix.

The type of the integer is resolved as follows:

Decimal Literal Types

Literal	Type
Usual decimal notation	
`0 .. 2_147_483_647`	`int`
`2_147_483_648 .. 9_223_372_036_854_775_807`	`long`
`9_223_372_036_854_775_808 .. 18_446_744_073_709_551_615`	`ulong`
Explicit suffixes	
`0L .. 9_223_372_036_854_775_807L`	`long`
`0U .. 4_294_967_295U`	`uint`
`4_294_967_296U .. 18_446_744_073_709_551_615U`	`ulong`
`0UL .. 18_446_744_073_709_551_615UL`	`ulong`
Hexadecimal notation	
`0x0 .. 0x7FFF_FFFF`	`int`
`0x8000_0000 .. 0xFFFF_FFFF`	`uint`
`0x1_0000_0000 .. 0x7FFF_FFFF_FFFF_FFFF`	`long`
`0x8000_0000_0000_0000 .. 0xFFFF_FFFF_FFFF_FFFF`	`ulong`
Hexadecimal notation with explicit suffixes	
`0x0L .. 0x7FFF_FFFF_FFFF_FFFFL`	`long`
`0x8000_0000_0000_0000L .. 0xFFFF_FFFF_FFFF_FFFFL`	`ulong`
`0x0U .. 0xFFFF_FFFFU`	`uint`
`0x1_0000_0000U .. 0xFFFF_FFFF_FFFF_FFFFU`	`ulong`
`0x0UL .. 0xFFFF_FFFF_FFFF_FFFFUL`	`ulong`

An integer literal may not exceed those values.

Best Practices: Octal integer notation is not supported in integer literals. However, octal integer literals can be interpreted at compile time through the `std.conv.octal` template, as in `octal!167`.

2.13 Floating Point Literals

FloatLiteral:
    Float
    Float Suffix
    Integer FloatSuffix
    Integer ImaginarySuffix
    Integer FloatSuffix ImaginarySuffix
    Integer RealSuffix ImaginarySuffix

Float:
  DecimalFloat
  HexFloat

DecimalFloat:
  LeadingDecimal .
  LeadingDecimal . *DecimalDigits*
  DecimalDigits . *DecimalDigitsNoStartingUS* *DecimalExponent*
  . *DecimalInteger*
  . *DecimalInteger* *DecimalExponent*
  LeadingDecimal *DecimalExponent*

DecimalExponent
  DecimalExponentStart *DecimalDigitsNoSingleUS*

DecimalExponentStart
  e
  E
  e+
  E+
  e-
  E-

HexFloat:
  HexPrefix *HexDigitsNoSingleUS* . *HexDigitsNoStartingUS* *HexExponent*
  HexPrefix . *HexDigitsNoStartingUS* *HexExponent*
  HexPrefix *HexDigitsNoSingleUS* *HexExponent*

HexPrefix:
  0x
  0X

HexExponent:

HexExponentStart DecimalDigitsNoSingleUS

HexExponentStart:
`p`
`P`
`p+`
`P+`
`p-`
`P-`

Suffix:
FloatSuffix
RealSuffix
ImaginarySuffix
FloatSuffix ImaginarySuffix
RealSuffix ImaginarySuffix

FloatSuffix:
`f`
`F`

RealSuffix:
`L`

ImaginarySuffix:
`i`

LeadingDecimal:
DecimalInteger
`0` *DecimalDigitsNoSingleUS*

Floats can be in decimal or hexadecimal format.

Hexadecimal floats are preceded by a **0x** or **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.

```
2.645_751
6.022140857E+23
6_022.140857E+20
6_022_.140_857E+20_
```

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

The literal may not exceed the range of the type. The literal is rounded to fit into the significant digits of the type.

If a floating literal has a **.** and a type suffix, at least one digit must be in-between:

```
1f;  // OK, float
1.f; // error
1.;  // OK, double
```

2.14 Keywords

Keywords are reserved identifiers.

Keyword:

`abstract`	`asm`		`bool`
`alias`	`assert`		`break`
`align`	`auto`	`body`	`byte`

	`for`		`typeof`
	`foreach`		
`case`	`foreach_reverse`	`out`	
`cast`	`function`	`override`	`ubyte`
`catch`			`ucent`
`cdouble`			`uint`
`cent`	`goto`	`package`	`ulong`
`cfloat`		`pragma`	`union`
`char`		`private`	`unittest`
`class`	`idouble`	`protected`	`ushort`
`const`	`if`	`public`	
`continue`	`ifloat`	`pure`	
`creal`	`immutable`		`version`
	`import`		`void`
	`in`	`real`	
`dchar`	`inout`	`ref`	
`debug`	`int`	`return`	`wchar`
`default`	`interface`		`while`
`delegate`	`invariant`		`with`
`delete` (deprecated)	`ireal`	`scope`	
`deprecated`	`is`	`shared`	
`do`		`short`	`__FILE__`
`double`		`static`	`__FILE_FULL_PATH__`
	`lazy`	`struct`	`__MODULE__`
	`long`	`super`	`__LINE__`
`else`		`switch`	`__FUNCTION__`
`enum`		`synchronized`	`__PRETTY_FUNCTION__`
`export`	`macro` (reserved)		
`extern`	`mixin`		
	`module`	`template`	`__gshared`
		`this`	`__traits`
`false`		`throw`	`__vector`
`final`	`new`	`true`	`__parameters`
`finally`	`nothrow`	`try`	
`float`	`null`	`typeid`	

2.15 Special Tokens

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

Special Tokens

Special Token	Replaced with
`__DATE__`	string literal of the date of compilation "*mmm dd yyyy*"
`__EOF__`	sets the scanner to the end of the file
`__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*"
`__VENDOR__`	Compiler vendor string
`__VERSION__`	Compiler version as an integer

Implementation Defined: The replacement string literal for `__VENDOR__` and the replacement integer value for `__VERSION__`.

2.16 Special Token Sequences

```
SpecialTokenSequence:
    # line IntegerLiteral EndOfLine
    # line IntegerLiteral Filespec EndOfLine
```

```
Filespec:
    " Characters "
```

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`.

This sets the current source line number to *IntegerLiteral*, and optionally the current source file name to *Filespec*, beginning with the next line of source text.

The backslash character is not treated specially inside *Filespec* strings.

For example:

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

Implementation Defined: The source file and line number is typically used for printing error messages and for mapping generated code back to the source for the symbolic debugging output.

Chapter 3

Grammar

3.1 Lexical Syntax

Refer to the page for lexical syntax.

3.2 Modules

Module:
    ModuleDeclaration DeclDefs
    DeclDefs

DeclDefs:
    DeclDef
    DeclDef DeclDefs

DeclDef:
    AttributeSpecifier
    Declaration
    Constructor
    Destructor
    Postblit
    Allocator
    Deallocator
    ClassInvariant
    StructInvariant

```
    UnitTest
    AliasThis
    StaticConstructor
    StaticDestructor
    SharedStaticConstructor
    SharedStaticDestructor
    ConditionalDeclaration
    DebugSpecification
    VersionSpecification
    StaticAssert
    TemplateDeclaration
    TemplateMixinDeclaration
    TemplateMixin
    MixinDeclaration
    ;


ModuleDeclaration:
    ModuleAttributes(opt) module ModuleFullyQualifiedName ;


ModuleAttributes:
    ModuleAttribute
    ModuleAttribute ModuleAttributes


ModuleAttribute:
    DeprecatedAttribute
    UserDefinedAttribute


ModuleFullyQualifiedName:
    ModuleName
    Packages . ModuleName


ModuleName:
    Identifier
```

```
Packages:
    PackageName
    Packages . PackageName

PackageName:
    Identifier

ImportDeclaration:
    import ImportList ;
    static import ImportList ;

ImportList:
    Import
    ImportBindings
    Import , ImportList

Import:
    ModuleFullyQualifiedName
    ModuleAliasIdentifier = ModuleFullyQualifiedName

ImportBindings:
    Import : ImportBindList

ImportBindList:
    ImportBind
    ImportBind , ImportBindList

ImportBind:
    Identifier
    Identifier = Identifier

ModuleAliasIdentifier:
```

```
    Identifier

MixinDeclaration:
    mixin ( ArgumentList ) ;
```

3.3 Declarations

```
Declaration:
    FuncDeclaration
    VarDeclarations
    AliasDeclaration
    AggregateDeclaration
    EnumDeclaration
    ImportDeclaration
    ConditionalDeclaration
    StaticForeachDeclaration
    StaticAssert

VarDeclarations:
    StorageClasses opt BasicType Declarators ;
    AutoDeclaration

Declarators:
    DeclaratorInitializer
    DeclaratorInitializer , DeclaratorIdentifierList

DeclaratorInitializer:
    VarDeclarator
    VarDeclarator TemplateParameters opt = Initializer
    AltDeclarator
    AltDeclarator = Initializer

DeclaratorIdentifierList:
    DeclaratorIdentifier
```

DeclaratorIdentifier , *DeclaratorIdentifierList*

DeclaratorIdentifier:
VarDeclaratorIdentifier
AltDeclaratorIdentifier

VarDeclaratorIdentifier:
Identifier
Identifier *TemplateParameters*$_{\text{opt}}$ = *Initializer*

AltDeclaratorIdentifier:
BasicType2 *Identifier* *AltDeclaratorSuffixes*$_{\text{opt}}$
BasicType2 *Identifier* *AltDeclaratorSuffixes*$_{\text{opt}}$ = *Initializer*
BasicType2$_{\text{opt}}$ *Identifier* *AltDeclaratorSuffixes*
BasicType2$_{\text{opt}}$ *Identifier* *AltDeclaratorSuffixes* = *Initializer*

Declarator:
VarDeclarator
AltDeclarator

VarDeclarator:
BasicType2$_{\text{opt}}$ *Identifier*

AltDeclarator:
BasicType2$_{\text{opt}}$ *Identifier* *AltDeclaratorSuffixes*
BasicType2$_{\text{opt}}$ (*AltDeclaratorX*)
BasicType2$_{\text{opt}}$ (*AltDeclaratorX*) *AltFuncDeclaratorSuffix*
BasicType2$_{\text{opt}}$ (*AltDeclaratorX*) *AltDeclaratorSuffixes*

AltDeclaratorX:
BasicType2$_{\text{opt}}$ *Identifier*
BasicType2$_{\text{opt}}$ *Identifier* *AltFuncDeclaratorSuffix*
AltDeclarator

```
AltDeclaratorSuffixes:
    AltDeclaratorSuffix
    AltDeclaratorSuffix AltDeclaratorSuffixes

AltDeclaratorSuffix:
    [ ]
    [ AssignExpression ]
    [ Type ]

AltFuncDeclaratorSuffix:
    Parameters MemberFunctionAttributes opt

Type:
    TypeCtors opt BasicType BasicType2 opt

TypeCtors:
    TypeCtor
    TypeCtor TypeCtors

TypeCtor:
    const
    immutable
    inout
    shared

BasicType:
    BasicTypeX
    . IdentifierList
    IdentifierList
    Typeof
    Typeof . IdentifierList
    TypeCtor ( Type )
```

Vector
Traits
MixinType

Vector:
`__vector` (*VectorBaseType*)

VectorBaseType:
Type

BasicTypeX:

`bool`	`int`	`ucent`	`double`	`cfloat`
`byte`	`uint`	`char`	`real`	`cdouble`
`ubyte`	`long`	`wchar`	`ifloat`	`creal`
`short`	`ulong`	`dchar`	`idouble`	`void`
`ushort`	`cent`	`float`	`ireal`	

BasicType2:
BasicType2X *BasicType2*$_{\text{opt}}$

BasicType2X:
*
[ ]
[*AssignExpression*]
[*AssignExpression* .. *AssignExpression*]
[*Type*]
`delegate` *Parameters* *MemberFunctionAttributes*$_{\text{opt}}$
`function` *Parameters* *FunctionAttributes*$_{\text{opt}}$

```
IdentifierList:
    Identifier
    Identifier . IdentifierList
    TemplateInstance
    TemplateInstance . IdentifierList
    Identifier [ AssignExpression ]
    Identifier [ AssignExpression ]. IdentifierList
```

```
StorageClasses:
    StorageClass
    StorageClass StorageClasses
```

```
StorageClass:
```

LinkageAttribute	extern	auto	shared	ref
AlignAttribute	abstract	scope	__gshared	
deprecated	final	const	*Property*	
enum	override	immutable	nothrow	
static	synchronized	inout	pure	

```
Initializer:
    VoidInitializer
    NonVoidInitializer
```

```
NonVoidInitializer:
    ExpInitializer
    ArrayInitializer
    StructInitializer
```

ExpInitializer:
    AssignExpression

ArrayInitializer:
    [*ArrayMemberInitializations*$_{\text{opt}}$]

ArrayMemberInitializations:
    ArrayMemberInitialization
    ArrayMemberInitialization ,
    ArrayMemberInitialization , *ArrayMemberInitializations*

ArrayMemberInitialization:
    NonVoidInitializer
    AssignExpression : *NonVoidInitializer*

StructInitializer:
    { *StructMemberInitializers*$_{\text{opt}}$ }

StructMemberInitializers:
    StructMemberInitializer
    StructMemberInitializer ,
    StructMemberInitializer , *StructMemberInitializers*

StructMemberInitializer:
    NonVoidInitializer
    Identifier : *NonVoidInitializer*

AutoDeclaration:
    StorageClasses *AutoDeclarationX* ;

AutoDeclarationX:
    AutoDeclarationY

AutoDeclarationX , *AutoDeclarationY*

AutoDeclarationY:
Identifier *TemplateParameters*$_{\text{opt}}$ = *Initializer*

AliasDeclaration:
`alias` *StorageClasses*$_{\text{opt}}$ *BasicType* *Declarators* ;
`alias` *StorageClasses*$_{\text{opt}}$ *BasicType* *FuncDeclarator* ;
`alias` *AliasDeclarationX* ;

AliasDeclarationX:
AliasDeclarationY
AliasDeclarationX , *AliasDeclarationY*

AliasDeclarationY:
Identifier *TemplateParameters*$_{\text{opt}}$ = *StorageClasses*$_{\text{opt}}$ *Type*
Identifier *TemplateParameters*$_{\text{opt}}$ = *FunctionLiteral*
Identifier *TemplateParameters*$_{\text{opt}}$ = *StorageClasses*$_{\text{opt}}$ *BasicType* *Parameters* *MemberFunctionAttri*

Typeof:
`typeof` (*Expression*)
`typeof` (`return`)

VoidInitializer:
`void`

3.4 Types

MixinType:
`mixin` (*ArgumentList*)

3.5 Attributes

```
AttributeSpecifier:
    Attribute :
    Attribute DeclarationBlock
```

```
Attribute:
    LinkageAttribute
    AlignAttribute
    DeprecatedAttribute
    VisibilityAttribute
    Pragma
    static
    extern
    abstract
    final
    override
    synchronized
    auto
    scope
    const
    immutable
    inout
    shared
    __gshared
    AtAttribute
    nothrow
    pure
    ref
    return
```

```
AtAttribute:
    @ disable
    @ nogc
    Property
    @ safe
    @ system
    @ trusted
```

```
    UserDefinedAttribute

Property:
    @ property

DeclarationBlock:
    DeclDef
    { DeclDefs opt }

LinkageAttribute:
    extern ( LinkageType )
    extern ( C++, IdentifierList )

LinkageType:
    C
    C++
    D
    Windows
    System
    Objective-C

AlignAttribute:
    align
    align ( AssignExpression )

DeprecatedAttribute:
    deprecated
    deprecated ( AssignExpression )

VisibilityAttribute:
    private
    package
    package ( IdentifierList )
```

```
    protected
    public
    export
```

```
UserDefinedAttribute:
    @ ( ArgumentList )
    @ Identifier
    @ Identifier ( ArgumentList opt )
    @ TemplateInstance
    @ TemplateInstance ( ArgumentList opt )
```

3.6 Pragmas

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

3.7 Expressions

```
Expression:
    CommaExpression
```

```
CommaExpression:
    AssignExpression
    AssignExpression , CommaExpression
```

```
AssignExpression:
    ConditionalExpression
    ConditionalExpression = AssignExpression
    ConditionalExpression += AssignExpression
    ConditionalExpression -= AssignExpression
    ConditionalExpression *= AssignExpression
    ConditionalExpression /= AssignExpression
    ConditionalExpression %= AssignExpression
```

ConditionalExpression &= AssignExpression
ConditionalExpression =| AssignExpression
ConditionalExpression ^= AssignExpression
ConditionalExpression ~= AssignExpression
ConditionalExpression <<= AssignExpression
ConditionalExpression >>= AssignExpression
ConditionalExpression >>>= AssignExpression
ConditionalExpression ^^= AssignExpression

ConditionalExpression:
OrOrExpression
OrOrExpression ? Expression : ConditionalExpression

OrOrExpression:
AndAndExpression
OrOrExpression || AndAndExpression

AndAndExpression:
OrExpression
AndAndExpression && OrExpression

OrExpression:
XorExpression
OrExpression | XorExpression

XorExpression:
AndExpression
XorExpression ^ AndExpression

AndExpression:
CmpExpression
AndExpression & CmpExpression

```
CmpExpression:
    ShiftExpression
    EqualExpression
    IdentityExpression
    RelExpression
    InExpression

EqualExpression:
    ShiftExpression == ShiftExpression
    ShiftExpression != ShiftExpression

IdentityExpression:
    ShiftExpression is ShiftExpression
    ShiftExpression !is ShiftExpression

RelExpression:
    ShiftExpression < ShiftExpression
    ShiftExpression <= ShiftExpression
    ShiftExpression > ShiftExpression
    ShiftExpression >= ShiftExpression

InExpression:
    ShiftExpression in ShiftExpression
    ShiftExpression !in ShiftExpression

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

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

```
    CatExpression

CatExpression:
    AddExpression ~ MulExpression

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

UnaryExpression:
    & UnaryExpression
    ++ UnaryExpression
    -- UnaryExpression
    * UnaryExpression
    - UnaryExpression
    + UnaryExpression
    ! UnaryExpression
    ComplementExpression
    ( Type ) . Identifier
    ( Type ) . TemplateInstance
    DeleteExpression
    CastExpression
    PowExpression

ComplementExpression:
    ~ UnaryExpression

NewExpression:
    new AllocatorArguments_opt Type
    NewExpressionWithArgs

NewExpressionWithArgs:
    new AllocatorArguments_opt Type [ AssignExpression ]
```

```
    new AllocatorArguments opt Type ( ArgumentList opt )
    NewAnonClassExpression


AllocatorArguments:
    ( ArgumentList opt )


ArgumentList:
    AssignExpression
    AssignExpression ,
    AssignExpression , ArgumentList


DeleteExpression:
    delete UnaryExpression


CastExpression:
    cast ( Type ) UnaryExpression
    cast ( TypeCtors opt ) UnaryExpression


PowExpression:
    PostfixExpression
    PostfixExpression ^^ UnaryExpression


PostfixExpression:
    PrimaryExpression
    PostfixExpression . Identifier
    PostfixExpression . TemplateInstance
    PostfixExpression . NewExpression
    PostfixExpression ++
    PostfixExpression --
    PostfixExpression ( ArgumentList opt )
    TypeCtors opt BasicType ( ArgumentList opt )
    IndexExpression
    SliceExpression
```

IndexExpression:
  PostfixExpression [*ArgumentList*]

SliceExpression:
  PostfixExpression [ ]
  PostfixExpression [*Slice* ,$_{\text{opt}}$]
Slice:
  AssignExpression
  AssignExpression , *Slice*
  AssignExpression .. *AssignExpression*
  AssignExpression .. *AssignExpression* , *Slice*

PrimaryExpression:
  Identifier
  . *Identifier*
  TemplateInstance
  . *TemplateInstance*
  this
  super
  null
  true
  false
  $
  IntegerLiteral
  FloatLiteral
  CharacterLiteral
  StringLiterals
  ArrayLiteral
  AssocArrayLiteral
  FunctionLiteral
  AssertExpression
  MixinExpression
  ImportExpression
  NewExpressionWithArgs
  BasicTypeX . *Identifier*
  BasicTypeX (*ArgumentList*$_{\text{opt}}$)
  TypeCtor (*Type*) . *Identifier*
  TypeCtor (*Type*) (*ArgumentList*$_{\text{opt}}$)

Typeof
TypeidExpression
IsExpression
(*Expression*)
SpecialKeyword
TraitsExpression

StringLiterals:
StringLiteral
StringLiterals StringLiteral

ArrayLiteral:
[*ArgumentList*$_{\text{opt}}$]

AssocArrayLiteral:
[*KeyValuePairs*]

KeyValuePairs:
KeyValuePair
KeyValuePair , *KeyValuePairs*

KeyValuePair:
KeyExpression : *ValueExpression*

KeyExpression:
AssignExpression

ValueExpression:
AssignExpression

FunctionLiteral:
`function` `ref`$_{\text{opt}}$ *Type*$_{\text{opt}}$ *ParameterWithAttributes*$_{\text{opt}}$ *FunctionLiteralBody*

```
    delegate ref_opt Type_opt ParameterWithMemberAttributes_opt FunctionLiteralBody
    ref_opt ParameterWithMemberAttributes FunctionLiteralBody
    FunctionLiteralBody
    Lambda

ParameterWithAttributes:
    Parameters FunctionAttributes_opt

ParameterWithMemberAttributes:
    Parameters MemberFunctionAttributes_opt

FunctionLiteralBody:
    BlockStatement
    FunctionContracts_opt BodyStatement

Lambda:
    function ref_opt Type_opt ParameterWithAttributes => AssignExpression
    delegate ref_opt Type_opt ParameterWithMemberAttributes => AssignExpression
    ref_opt ParameterWithMemberAttributes => AssignExpression
    Identifier => AssignExpression

AssertExpression:
    assert ( AssertArguments )

AssertArguments:
    AssignExpression ,_opt
    AssignExpression , AssignExpression ,_opt

MixinExpression:
    mixin ( ArgumentList )

ImportExpression:
```

```
    import ( AssignExpression )

TypeidExpression:
    typeid ( Type )
    typeid ( Expression )

IsExpression:
    is ( Type )
    is ( Type : TypeSpecialization )
    is ( Type == TypeSpecialization )
    is ( Type : TypeSpecialization , TemplateParameterList )
    is ( Type == TypeSpecialization , TemplateParameterList )
    is ( Type Identifier )
    is ( Type Identifier : TypeSpecialization )
    is ( Type Identifier == TypeSpecialization )
    is ( Type Identifier : TypeSpecialization , TemplateParameterList )
    is ( Type Identifier == TypeSpecialization , TemplateParameterList )

TypeSpecialization:
    Type
    struct
    union
    class
    interface
    enum
    __vector
    function
    delegate
    super
    const
    immutable
    inout
    shared
    return
    __parameters
```

```
    module
    package
```

SpecialKeyword:
```
    __FILE__
    __FILE_FULL_PATH__
    __MODULE__
    __LINE__
    __FUNCTION__
    __PRETTY_FUNCTION__
```

3.8 Statements

Statement:
```
    ;
    NonEmptyStatement
    ScopeBlockStatement
```

NoScopeNonEmptyStatement:
```
    NonEmptyStatement
    BlockStatement
```

NoScopeStatement:
```
    ;
    NonEmptyStatement
    BlockStatement
```

NonEmptyOrScopeBlockStatement:
```
    NonEmptyStatement
    ScopeBlockStatement
```

NonEmptyStatement:
```
    NonEmptyStatementNoCaseNoDefault
    CaseStatement
```

CaseRangeStatement
DefaultStatement

NonEmptyStatementNoCaseNoDefault:
LabeledStatement
ExpressionStatement
DeclarationStatement
IfStatement
WhileStatement
DoStatement
ForStatement
ForeachStatement
SwitchStatement
FinalSwitchStatement
ContinueStatement
BreakStatement
ReturnStatement
GotoStatement
WithStatement
SynchronizedStatement
TryStatement
ScopeGuardStatement
ThrowStatement
AsmStatement
PragmaStatement
MixinStatement
ForeachRangeStatement
ConditionalStatement
StaticForeachStatement
StaticAssert
TemplateMixin
ImportDeclaration

ScopeStatement:
NonEmptyStatement
BlockStatement

```
ScopeBlockStatement:
    BlockStatement


LabeledStatement:
    Identifier :
    Identifier : NoScopeStatement
    Identifier : Statement


BlockStatement:
    { }
    { StatementList }


StatementList:
    Statement
    Statement StatementList


ExpressionStatement:
    Expression ;


DeclarationStatement:
    StorageClasses(opt) Declaration


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


IfCondition:
    Expression
    auto Identifier = Expression
    TypeCtors Identifier = Expression
    TypeCtors(opt) BasicType Declarator = Expression


ThenStatement:
```

ScopeStatement

ElseStatement:
ScopeStatement

WhileStatement:
`while` (*Expression*) *ScopeStatement*

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

ForStatement:
`for` (*Initialize* *Test*$_{\text{opt}}$; *Increment*$_{\text{opt}}$) *ScopeStatement*

Initialize:
;
NoScopeNonEmptyStatement

Test:
Expression

Increment:
Expression

AggregateForeach:
Foreach (*ForeachTypeList* ; *ForeachAggregate*)

ForeachStatement:
AggregateForeach *NoScopeNonEmptyStatement*

Foreach:
  `foreach`
  `foreach_reverse`

ForeachTypeList:
  ForeachType
  ForeachType `,` *ForeachTypeList*

ForeachType:
  ForeachTypeAttributes$_{\text{opt}}$ *BasicType Declarator*
  ForeachTypeAttributes$_{\text{opt}}$ *Identifier*
  ForeachTypeAttributes$_{\text{opt}}$ `alias` *Identifier*

ForeachTypeAttributes
  ForeachTypeAttribute
  ForeachTypeAttribute ForeachTypeAttributes$_{\text{opt}}$

ForeachTypeAttribute:
  `ref`
  TypeCtor
  `enum`

ForeachAggregate:
  Expression

RangeForeach:
  Foreach `(` *ForeachType* `;` *LwrExpression* `..` *UprExpression* `)`

LwrExpression:
  Expression

UprExpression:

Expression

ForeachRangeStatement:
RangeForeach ScopeStatement

SwitchStatement:
`switch` (*Expression*) *ScopeStatement*

CaseStatement:
`case` *ArgumentList* : *ScopeStatementList*

CaseRangeStatement:
`case` *FirstExp* : .. `case` *LastExp* : *ScopeStatementList*

FirstExp:
AssignExpression

LastExp:
AssignExpression

DefaultStatement:
`default` : *ScopeStatementList*

ScopeStatementList:
StatementListNoCaseNoDefault

StatementListNoCaseNoDefault:
StatementNoCaseNoDefault
StatementNoCaseNoDefault StatementListNoCaseNoDefault

```
StatementNoCaseNoDefault:
    ;
    NonEmptyStatementNoCaseNoDefault
    ScopeBlockStatement
```

```
FinalSwitchStatement:
    final switch ( Expression ) ScopeStatement
```

```
ContinueStatement:
    continue Identifier_opt ;
```

```
BreakStatement:
    break Identifier_opt ;
```

```
ReturnStatement:
    return Expression_opt ;
```

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

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

```
SynchronizedStatement:
    synchronized ScopeStatement
    synchronized ( Expression ) ScopeStatement
```

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

Catches:
  Catch
  Catch Catches

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

CatchParameter:
  BasicType Identifier$_{\text{opt}}$

FinallyStatement:
  `finally` *NoScopeNonEmptyStatement*

ThrowStatement:
  `throw` *Expression* ;

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

AsmStatement:
  `asm` *FunctionAttributes*$_{\text{opt}}$ { *AsmInstructionList*$_{\text{opt}}$ }

AsmInstructionList:
  AsmInstruction ;
  AsmInstruction ; *AsmInstructionList*

```
PragmaStatement:
    Pragma NoScopeStatement
```

```
MixinStatement:
    mixin ( ArgumentList ) ;
```

3.9 Structs and Unions

```
AggregateDeclaration:
    ClassDeclaration
    InterfaceDeclaration
    StructDeclaration
    UnionDeclaration
```

```
StructDeclaration:
    struct Identifier ;
    struct Identifier AggregateBody
    StructTemplateDeclaration
    AnonStructDeclaration
```

```
AnonStructDeclaration:
    struct AggregateBody
```

```
UnionDeclaration:
    union Identifier ;
    union Identifier AggregateBody
    UnionTemplateDeclaration
    AnonUnionDeclaration
```

```
AnonUnionDeclaration:
    union AggregateBody
```

```
AggregateBody:
```

{ *DeclDefs*$_{\text{opt}}$ }

Postblit:
this (this) *MemberFunctionAttributes*$_{\text{opt}}$;
this (this) *MemberFunctionAttributes*$_{\text{opt}}$ *FunctionBody*

StructInvariant:
invariant ( ) *BlockStatement*
invariant *BlockStatement*
invariant (*AssertArguments*) ;

3.10 Classes

ClassDeclaration:
class *Identifier* ;
class *Identifier* *BaseClassList*$_{\text{opt}}$ *AggregateBody*
ClassTemplateDeclaration

BaseClassList:
: *SuperClass*
: *SuperClass* , *Interfaces*
: *Interfaces*

SuperClass:
BasicType

Interfaces:
Interface
Interface , *Interfaces*

Interface:
BasicType

```
Constructor:
    this Parameters MemberFunctionAttributes_opt ;
    this Parameters MemberFunctionAttributes_opt FunctionBody
    ConstructorTemplate

Destructor:
    ~ this ( ) MemberFunctionAttributes_opt ;
    ~ this ( ) MemberFunctionAttributes_opt FunctionBody

StaticConstructor:
    static this ( ) MemberFunctionAttributes_opt ;
    static this ( ) MemberFunctionAttributes_opt FunctionBody

StaticDestructor:
    static ~ this ( ) MemberFunctionAttributes_opt ;
    static ~ this ( ) MemberFunctionAttributes_opt FunctionBody

SharedStaticConstructor:
    shared static this ( ) MemberFunctionAttributes_opt ;
    shared static this ( ) MemberFunctionAttributes_opt FunctionBody

SharedStaticDestructor:
    shared static ~ this ( ) MemberFunctionAttributes_opt ;
    shared static ~ this ( ) MemberFunctionAttributes_opt FunctionBody

ClassInvariant:
    invariant ( ) BlockStatement
    invariant BlockStatement
    invariant ( AssertArguments ) ;

Allocator:
    new Parameters ;
    new Parameters FunctionBody
```

```
Deallocator:
    delete Parameters ;
    delete Parameters FunctionBody
```

```
AliasThis:
    alias Identifier this ;
```

```
NewAnonClassExpression:
    new AllocatorArguments_opt class ClassArguments_opt SuperClass_opt Interfaces_opt AggregateBody
```

```
ClassArguments:
    ( ArgumentList_opt )
```

```
class Identifier : SuperClass Interfaces AggregateBody
new (ArgumentList) Identifier (ArgumentList);
```

3.11 Interfaces

```
InterfaceDeclaration:
    interface Identifier ;
    interface Identifier BaseInterfaceList_opt AggregateBody
    InterfaceTemplateDeclaration
```

```
BaseInterfaceList:
    : Interfaces
```

3.12 Enums

```
EnumDeclaration:
    enum Identifier EnumBody
    enum Identifier : EnumBaseType EnumBody
    AnonymousEnumDeclaration
```

```
EnumBaseType:
    Type

EnumBody:
    { EnumMembers }
    ;

EnumMembers:
    EnumMember
    EnumMember ,
    EnumMember , EnumMembers

EnumMemberAttributes:
    EnumMemberAttribute
    EnumMemberAttribute EnumMemberAttributes

EnumMemberAttribute:
    DeprecatedAttribute
    UserDefinedAttribute
    @disable

EnumMember:
    EnumMemberAttributes_opt Identifier
    EnumMemberAttributes_opt Identifier = AssignExpression

AnonymousEnumDeclaration:
    enum : EnumBaseType { EnumMembers }
    enum { EnumMembers }
    enum { AnonymousEnumMembers }

AnonymousEnumMembers:
    AnonymousEnumMember
```

```
    AnonymousEnumMember ,
    AnonymousEnumMember , AnonymousEnumMembers
```

```
AnonymousEnumMember:
    EnumMember
    Type Identifier = AssignExpression
```

3.13 Functions

```
FuncDeclaration:
    StorageClasses opt BasicType FuncDeclarator FunctionBody
    AutoFuncDeclaration
```

```
AutoFuncDeclaration:
    StorageClasses Identifier FuncDeclaratorSuffix FunctionBody
```

```
FuncDeclarator:
    BasicType2 opt Identifier FuncDeclaratorSuffix
```

```
FuncDeclaratorSuffix:
    Parameters MemberFunctionAttributes opt
    TemplateParameters Parameters MemberFunctionAttributes opt Constraint opt
```

```
Parameters:
    ( ParameterList opt )
```

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

```
Parameter:
```

ParameterAttributes$_{\mathrm{opt}}$ *BasicType Declarator*
ParameterAttributes$_{\mathrm{opt}}$ *BasicType Declarator* ...
ParameterAttributes$_{\mathrm{opt}}$ *BasicType Declarator* = *AssignExpression*
ParameterAttributes$_{\mathrm{opt}}$ *Type*
ParameterAttributes$_{\mathrm{opt}}$ *Type* ...

ParameterAttributes:
InOut
UserDefinedAttribute
ParameterAttributes InOut
ParameterAttributes UserDefinedAttribute
ParameterAttributes

InOut:
`auto`
TypeCtor
`final`
`in`
`lazy`
`out`
`ref`
`return ref`
`scope`

FunctionAttributes:
FunctionAttribute
FunctionAttribute FunctionAttributes

FunctionAttribute:
`nothrow`
`pure`
Property

MemberFunctionAttributes:
MemberFunctionAttribute

MemberFunctionAttribute MemberFunctionAttributes

MemberFunctionAttribute:
`const`
`immutable`
`inout`
`return`
`shared`
FunctionAttribute

FunctionBody:
SpecifiedFunctionBody
MissingFunctionBody

FunctionLiteralBody:
SpecifiedFunctionBody

SpecifiedFunctionBody:
`do`$_{\text{opt}}$ *BlockStatement*
FunctionContracts$_{\text{opt}}$ *InOutContractExpression* `do`$_{\text{opt}}$ *BlockStatement*
FunctionContracts$_{\text{opt}}$ *InOutStatement* `do` *BlockStatement*

MissingFunctionBody:
;
FunctionContracts$_{\text{opt}}$ *InOutContractExpression* ;
FunctionContracts$_{\text{opt}}$ *InOutStatement*

FunctionContracts:
FunctionContract
FunctionContract FunctionContracts

FunctionContract:
InOutContractExpression

```
    InOutStatement

InOutContractExpression:
    InContractExpression
    OutContractExpression

InOutStatement:
    InStatement
    OutStatement

InContractExpression:
    in ( AssertArguments )

OutContractExpression:
    out ( ; AssertArguments )
    out ( Identifier ; AssertArguments )

InStatement:
    in BlockStatement

OutStatement:
    out BlockStatement
    out ( Identifier ) BlockStatement
```

3.14 Templates

```
TemplateDeclaration:
    template Identifier TemplateParameters Constraint opt { DeclDefs opt }

TemplateParameters:
    ( TemplateParameterList opt )
```

TemplateParameterList:
TemplateParameter
TemplateParameter ,
TemplateParameter , *TemplateParameterList*

TemplateParameter:
TemplateTypeParameter
TemplateValueParameter
TemplateAliasParameter
TemplateSequenceParameter
TemplateThisParameter

TemplateInstance:
Identifier TemplateArguments

TemplateArguments:
! (*TemplateArgumentList*$_{\text{opt}}$)
! *TemplateSingleArgument*

TemplateArgumentList:
TemplateArgument
TemplateArgument ,
TemplateArgument , *TemplateArgumentList*

TemplateArgument:
Type
AssignExpression
Symbol

Symbol:
SymbolTail
. *SymbolTail*

```
SymbolTail:
    Identifier
    Identifier . SymbolTail
    TemplateInstance
    TemplateInstance . SymbolTail
```

```
TemplateSingleArgument:
    Identifier
    BasicTypeX
    CharacterLiteral
    StringLiteral
    IntegerLiteral
    FloatLiteral
    true
    false
    null
    this
    SpecialKeyword
```

```
TemplateTypeParameter:
    Identifier
    Identifier TemplateTypeParameterSpecialization
    Identifier TemplateTypeParameterDefault
    Identifier TemplateTypeParameterSpecialization TemplateTypeParameterDefault
```

```
TemplateTypeParameterSpecialization:
    : Type
```

```
TemplateTypeParameterDefault:
    = Type
```

```
TemplateThisParameter:
    this TemplateTypeParameter
```

TemplateValueParameter:
BasicType Declarator
BasicType Declarator TemplateValueParameterSpecialization
BasicType Declarator TemplateValueParameterDefault
BasicType Declarator TemplateValueParameterSpecialization TemplateValueParameterDefault

TemplateValueParameterSpecialization:
: *ConditionalExpression*

TemplateValueParameterDefault:
= *AssignExpression*
= *SpecialKeyword*

TemplateAliasParameter:
alias *Identifier TemplateAliasParameterSpecialization*$_{\text{opt}}$ *TemplateAliasParameterDefault*$_{\text{opt}}$
alias *BasicType Declarator TemplateAliasParameterSpecialization*$_{\text{opt}}$ *TemplateAliasParameterD*

TemplateAliasParameterSpecialization:
: *Type*
: *ConditionalExpression*

TemplateAliasParameterDefault:
= *Type*
= *ConditionalExpression*

TemplateSequenceParameter:
Identifier ...

ConstructorTemplate:
this *TemplateParameters Parameters MemberFunctionAttributes*$_{\text{opt}}$ *Constraint*$_{\text{opt}}$:
this *TemplateParameters Parameters MemberFunctionAttributes*$_{\text{opt}}$ *Constraint*$_{\text{opt}}$ *FunctionBody*

ClassTemplateDeclaration:
    `class` *Identifier* *TemplateParameters* ;
    `class` *Identifier* *TemplateParameters* *Constraint*$_{\text{opt}}$ *BaseClassList*$_{\text{opt}}$ *AggregateBody*
    `class` *Identifier* *TemplateParameters* *BaseClassList*$_{\text{opt}}$ *Constraint*$_{\text{opt}}$ *AggregateBody*

InterfaceTemplateDeclaration:
    `interface` *Identifier* *TemplateParameters* ;
    `interface` *Identifier* *TemplateParameters* *Constraint*$_{\text{opt}}$ *BaseInterfaceList*$_{\text{opt}}$ *AggregateBody*
    `interface` *Identifier* *TemplateParameters* *BaseInterfaceList* *Constraint* *AggregateBody*

StructTemplateDeclaration:
    `struct` *Identifier* *TemplateParameters* ;
    `struct` *Identifier* *TemplateParameters* *Constraint*$_{\text{opt}}$ *AggregateBody*

UnionTemplateDeclaration:
    `union` *Identifier* *TemplateParameters* ;
    `union` *Identifier* *TemplateParameters* *Constraint*$_{\text{opt}}$ *AggregateBody*

Constraint:
    `if` (*Expression*)

3.15 Template Mixins

TemplateMixinDeclaration:
    `mixin` `template` *Identifier* *TemplateParameters* *Constraint*$_{\text{opt}}$ { *DeclDefs*$_{\text{opt}}$ }

TemplateMixin:
    `mixin` *MixinTemplateName* *TemplateArguments*$_{\text{opt}}$ *Identifier*$_{\text{opt}}$;

MixinTemplateName:
    . *QualifiedIdentifierList*
    QualifiedIdentifierList
    Typeof . *QualifiedIdentifierList*

```
QualifiedIdentifierList:
    Identifier
    Identifier . QualifiedIdentifierList
    TemplateInstance . QualifiedIdentifierList
```

3.16 Conditional Compilation

```
ConditionalDeclaration:
    Condition DeclarationBlock
    Condition DeclarationBlock else DeclarationBlock
    Condition : DeclDefs opt
    Condition DeclarationBlock else : DeclDefs opt
```

```
ConditionalStatement:
    Condition NoScopeNonEmptyStatement
    Condition NoScopeNonEmptyStatement else NoScopeNonEmptyStatement
```

```
Condition:
    VersionCondition
    DebugCondition
    StaticIfCondition
```

```
VersionCondition:
    version ( IntegerLiteral )
    version ( Identifier )
    version ( unittest )
    version ( assert )
```

```
VersionSpecification:
    version = Identifier ;
    version = IntegerLiteral ;
```

DebugCondition:
  `debug`
  `debug` (*IntegerLiteral*)
  `debug` (*Identifier*)

DebugSpecification:
  `debug` = *Identifier* ;
  `debug` = *IntegerLiteral* ;

StaticIfCondition:
  `static if` (*AssignExpression*)

StaticForeach:
  `static` *AggregateForeach*
  `static` *RangeForeach*

StaticForeachDeclaration:
  StaticForeach DeclarationBlock
  StaticForeach : *DeclDefs* $_{\text{opt}}$

StaticForeachStatement:
  StaticForeach NoScopeNonEmptyStatement

StaticAssert:
  `static assert` (*AssertArguments*);

3.17 Traits

TraitsExpression:
  `__traits` (*TraitsKeyword* , *TraitsArguments*)

TraitsKeyword:

```
isAbstractClass
isArithmetic
isAssociativeArray
isFinalClass
isPOD
isNested
isFuture
isDeprecated
isFloating
isIntegral
isScalar
isStaticArray
isUnsigned
isDisabled
isVirtualFunction
isVirtualMethod
isAbstractFunction
isFinalFunction
isStaticFunction
isOverrideFunction
isTemplate
isRef
isOut
isLazy
isReturnOnStack
isZeroInit
isModule
isPackage
hasMember
hasCopyConstructor
hasPostblit
identifier
getAliasThis
getAttributes
getFunctionAttributes
getFunctionVariadicStyle
getLinkage
getLocation
getMember
getOverloads
```

getParameterStorageClasses
getPointerBitmap
getProtection
getTargetInfo
getVirtualFunctions
getVirtualMethods
getUnitTests
parent
classInstanceSize
getVirtualIndex
allMembers
derivedMembers
isSame
compiles

TraitsArguments:
TraitsArgument
TraitsArgument , *TraitsArguments*

TraitsArgument:
AssignExpression
Type

3.18 Unit Tests

UnitTest:
`unittest` *BlockStatement*

3.19 D x86 Inline Assembler

AsmStatement:
`asm` *FunctionAttributes*$_{\text{opt}}$ { *AsmInstructionList*$_{\text{opt}}$ }

AsmInstructionList:

```
    AsmInstruction ;
    AsmInstruction ; AsmInstructionList
```

```
AsmInstruction:
    Identifier : AsmInstruction
    align IntegerExpression
    even
    naked
    db Operands
    ds Operands
    di Operands
    dl Operands
    df Operands
    dd Operands
    de Operands
    db StringLiteral
    ds StringLiteral
    di StringLiteral
    dl StringLiteral
    dw StringLiteral
    dq StringLiteral
    Opcode
    Opcode Operands
```

```
Operands:
    Operand
    Operand , Operands
```

```
IntegerExpression:
    IntegerLiteral
    Identifier
```

```
Register:
    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
```

Register64:

```
    RAX  RBX  RCX  RDX
    BPL  RBP
    SPL  RSP
    DIL  RDI
    SIL  RSI
    R8B  R8W  R8D  R8
    R9B  R9W  R9D  R9
    R10B R10W R10D R10
    R11B R11W R11D R11
    R12B R12W R12D R12
    R13B R13W R13D R13
    R14B R14W R14D R14
    R15B R15W R15D R15
    XMM8 XMM9 XMM10 XMM11 XMM12 XMM13 XMM14 XMM15
    YMM0 YMM1 YMM2  YMM3  YMM4  YMM5  YMM6  YMM7
    YMM8 YMM9 YMM10 YMM11 YMM12 YMM13 YMM14 YMM15
```

Operand:
    AsmExp

AsmExp:
    AsmLogOrExp

```
    AsmLogOrExp ? AsmExp : AsmExp

AsmLogOrExp:
    AsmLogAndExp
    AsmLogOrExp || AsmLogAndExp

AsmLogAndExp:
    AsmOrExp
    AsmLogAndExp && AsmOrExp

AsmOrExp:
    AsmXorExp
    AsmOrExp | AsmXorExp

AsmXorExp:
    AsmAndExp
    AsmXorExp ^ AsmAndExp

AsmAndExp:
    AsmEqualExp
    AsmAndExp & AsmEqualExp

AsmEqualExp:
    AsmRelExp
    AsmEqualExp == AsmRelExp
    AsmEqualExp != AsmRelExp

AsmRelExp:
    AsmShiftExp
    AsmRelExp < AsmShiftExp
    AsmRelExp <= AsmShiftExp
    AsmRelExp > AsmShiftExp
    AsmRelExp >= AsmShiftExp
```

```
AsmShiftExp:
   AsmAddExp
   AsmShiftExp << AsmAddExp
   AsmShiftExp >> AsmAddExp
   AsmShiftExp >>> AsmAddExp

AsmAddExp:
   AsmMulExp
   AsmAddExp + AsmMulExp
   AsmAddExp - AsmMulExp

AsmMulExp:
   AsmBrExp
   AsmMulExp * AsmBrExp
   AsmMulExp / AsmBrExp
   AsmMulExp % AsmBrExp

AsmBrExp:
   AsmUnaExp
   AsmBrExp [ AsmExp ]

AsmUnaExp:
   AsmTypePrefix AsmExp
   offsetof AsmExp
   seg AsmExp
   + AsmUnaExp
   - AsmUnaExp
   ! AsmUnaExp
   ~ AsmUnaExp
   AsmPrimaryExp

AsmPrimaryExp:
   IntegerLiteral
```

```
    FloatLiteral
    __LOCAL_SIZE
    $
    Register
    Register : AsmExp
    Register64
    Register64 : AsmExp
    DotIdentifier
    this
```

```
DotIdentifier:
    Identifier
    Identifier . DotIdentifier
    BasicTypeX . Identifier
```

```
AsmTypePrefix:
    near ptr
    far ptr
    word ptr
    dword ptr
    qword ptr
    BasicTypeX ptr
```

3.20 Named Character Entities

```
NamedCharacterEntity:
    & Identifier ;
```

3.21 Application Binary Interface

```
MangledName:
    _D QualifiedName Type
    _D QualifiedName Z        // Internal
```

```
QualifiedName:
```

```
   SymbolFunctionName
   SymbolFunctionName QualifiedName


SymbolFunctionName:
   SymbolName
   SymbolName TypeFunctionNoReturn
   SymbolName M TypeModifiers_opt TypeFunctionNoReturn


SymbolName:
   LName
   TemplateInstanceName
   IdentifierBackRef
   0                         // anonymous symbols


TemplateInstanceName:
   TemplateID LName TemplateArgs Z


TemplateID:
   __T
   __U        // for symbols declared inside template constraint


TemplateArgs:
   TemplateArg
   TemplateArg TemplateArgs


TemplateArg:
   TemplateArgX
   H TemplateArgX


TemplateArgX:
   T Type
   V Type Value
   S QualifiedName
```

X *Number ExternallyMangledName*

Value:
n
i *Number*
N *Number*
e *HexFloat*
c *HexFloat* c *HexFloat*
CharWidth Number _ *HexDigits*
A *Number Value...*
S *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

CharWidth:
    a
    w
    d

Name:
    Namestart
    Namestart Namechars

Namestart:
    _
    Alpha

Namechar:
    Namestart
    Digit

Namechars:
    Namechar
    Namechar Namechars

LName:
    Number Name

Number:
    Digit
    Digit Number

Digit:
    0
    1

2
3
4
5
6
7
8
9

TypeBackRef:
Q *NumberBackRef*

IdentifierBackRef:
Q *NumberBackRef*

NumberBackRef:
lower-case-letter
upper-case-letter *NumberBackRef*

Type:
TypeModifiers$_{\text{opt}}$ *TypeX*
TypeBackRef

TypeX:
TypeArray
TypeStaticArray
TypeAssocArray
TypePointer
TypeFunction
TypeIdent
TypeClass
TypeStruct
TypeEnum
TypeTypedef
TypeDelegate

TypeVoid
TypeByte
TypeUbyte
TypeShort
TypeUshort
TypeInt
TypeUint
TypeLong
TypeUlong
TypeCent
TypeUcent
TypeFloat
TypeDouble
TypeReal
TypeIfloat
TypeIdouble
TypeIreal
TypeCfloat
TypeCdouble
TypeCreal
TypeBool
TypeChar
TypeWchar
TypeDchar
TypeNull
TypeTuple
TypeVector

TypeModifiers:
Const
Wild
Wild Const
Shared
Shared Const
Shared Wild
Shared Wild Const
Immutable

Shared:
    `O`

Const:
    `x`

Immutable:
    `y`

Wild:
    `Ng`

TypeArray:
    `A` *Type*

TypeStaticArray:
    `G` *Number Type*

TypeAssocArray:
    `H` *Type Type*

TypePointer:
    `P` *Type*

TypeVector:
    `Nh` *Type*

TypeFunction:
    TypeFunctionNoReturn Type

TypeFunctionNoReturn:
    CallConvention *FuncAttrs*$_{\text{opt}}$ *Parameters*$_{\text{opt}}$ *ParamClose*

CallConvention:
```
F       // D
U       // C
W       // Windows
R       // C++
Y       // Objective-C
```

FuncAttrs:
    FuncAttr
    FuncAttr *FuncAttrs*

FuncAttr:
    FuncAttrPure
    FuncAttrNothrow
    FuncAttrRef
    FuncAttrProperty
    FuncAttrNogc
    FuncAttrReturn
    FuncAttrScope
    FuncAttrTrusted
    FuncAttrSafe
    FuncAttrLive

FuncAttrPure:
    `Na`

FuncAttrNogc:
    `Ni`

FuncAttrNothrow:
    `Nb`

```
FuncAttrProperty:
   Nd

FuncAttrRef:
   Nc

FuncAttrReturn:
   Nj

FuncAttrScope:
   Nl

FuncAttrTrusted:
   Ne

FuncAttrSafe:
   Nf

FuncAttrLive:
   Nm

Parameters:
   Parameter
   Parameter Parameters

Parameter:
   Parameter2
   M Parameter2      // scope
   Nk Parameter2     // return
```

```
Parameter2:
    Type
    J Type      // out
    K Type      // ref
    L Type      // lazy
```

```
ParamClose
    X     // variadic T t...) style
    Y     // variadic T t,...) style
    Z     // not variadic
```

```
TypeIdent:
    I QualifiedName
```

```
TypeClass:
    C QualifiedName
```

```
TypeStruct:
    S QualifiedName
```

```
TypeEnum:
    E QualifiedName
```

```
TypeTypedef:
    T QualifiedName
```

TypeDelegate:
  D *TypeModifiers*$_{\mathrm{opt}}$ *TypeFunction*

```
TypeVoid:
    v
```

TypeByte:
`g`

TypeUbyte:
`h`

TypeShort:
`s`

TypeUshort:
`t`

TypeInt:
`i`

TypeUint:
`k`

TypeLong:
`l`

TypeUlong:
`m`

TypeCent:
`zi`

TypeUcent:
`zk`

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`

TypeNull:
`n`

TypeTuple:
`B` *Parameters* `Z`

Chapter 4

Modules

```
Module:
    ModuleDeclaration DeclDefs
    DeclDefs
```

```
DeclDefs:
    DeclDef
    DeclDef DeclDefs
```

```
DeclDef:
    AttributeSpecifier
    Declaration
    Constructor
    Destructor
    Postblit
    Allocator
    Deallocator
    ClassInvariant
    StructInvariant
    UnitTest
    AliasThis
    StaticConstructor
    StaticDestructor
    SharedStaticConstructor
    SharedStaticDestructor
```

ConditionalDeclaration
DebugSpecification
VersionSpecification
StaticAssert
TemplateDeclaration
TemplateMixinDeclaration
TemplateMixin
MixinDeclaration
;

Modules have a one-to-one correspondence with source files. The module name is, by default, the file name with the path and extension stripped off, and can be set explicitly with the module declaration.

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.
- A file may contain only one module.
- 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*.

Modules offer several guarantees:

- The order in which modules are imported does not affect the semantics.
- The semantics of a module are not affected by what imports it.
- If a module C imports modules A and B, any modifications to B will not silently change code in C that is dependent on A.

4.1 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:
    ModuleAttributes$_{\mathrm{opt}}$ `module` *ModuleFullyQualifiedName* `;`

ModuleAttributes:
    ModuleAttribute
    ModuleAttribute *ModuleAttributes*

ModuleAttribute:
    DeprecatedAttribute
    UserDefinedAttribute

ModuleFullyQualifiedName:
    ModuleName
    Packages `.` *ModuleName*

ModuleName:
    Identifier

Packages:
    PackageName
    Packages `.` *PackageName*

PackageName:
    Identifier

The *Identifier*s preceding the rightmost *Identifier* are the *Packages* that the module is in. The packages correspond to directory names in the source file path. Package and module names cannot be *Keyword*s.

If present, the *ModuleDeclaration* must be the first and only such declaration in the source file, and may be preceded only by comments and `#line` directives.

Example:

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

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

If the file name of a module is an invalid module name (e.g. `foo-bar.d`), you may use a module declaration to set a valid module name:

```
module foo_bar;
```

A *ModuleDeclaration* can have an optional *DeprecatedAttribute*. The compiler will produce a message when the deprecated module is imported.

```
deprecated module foo;

module bar;
import foo;  // Deprecated: module foo is deprecated
```

A *DeprecatedAttribute* can have an optional *AssignExpression* argument to provide a more informative message. An *AssignExpression* must evaluate to a string at compile time.

```
deprecated("Please␣use␣foo2␣instead.")
module foo;

module bar;
import foo;  // Deprecated: module foo is deprecated - Please use foo2 instead.
```

Implementation Defined:

1. The mapping of package and module identifiers to directory and file names.
2. How the deprecation messages are presented to the user.

Best Practices:

1. *PackageName*s and *ModuleName*s should be composed of the ASCII characters lower case letters, digits or _ to ensure maximum portability and compatibility with various file systems.
2. The file names for packages and modules should be composed only of the ASCII lower case letters, digits, and _s, and should not be a *Keyword*.

4.2 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:
    ModuleFullyQualifiedName
    ModuleAliasIdentifier = ModuleFullyQualifiedName
```

```
ImportBindings:
    Import : ImportBindList
```

```
ImportBindList:
    ImportBind
    ImportBind , ImportBindList
```

```
ImportBind:
    Identifier
    Identifier = Identifier
```

```
ModuleAliasIdentifier:
    Identifier
```

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

The order in which *ImportDeclaration*s occur has no significance.

*ModuleFullyQualifiedName*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.

4.3 Symbol Name Lookup

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

```
module myapp.main;

import std.stdio; // import module stdio from package std

class Foo : BaseClass
{
    import myapp.foo;  // import module myapp.foo in this class' scope
    void bar ()
    {
        import myapp.bar;  // import module myapp.bar in this function' scope
        writeln("hello!");  // calls std.stdio.writeln
    }
}
```

When a symbol name is used unqualified, a two-phase lookup is used. First, the module scope is searched, starting from the innermost scope. For example, in the previous example, while looking for `writeln`, the order will be:

- Declarations inside `bar`.
- Declarations inside `Foo`.
- Declarations inside `BaseClass`.
- Declarations at module scope.

If the first lookup isn't successful, a second one is performed on imports. In the second lookup phase inherited scopes are ignored. This includes the scope of base classes and interfaces (in this example, `BaseClass`'s imports would be ignored), as well as imports in mixed-in `template`.

Symbol lookup stops as soon as a matching symbol is found. If two symbols with the same name are found at the same lookup phase, this ambiguity will result in a compilation error.

```
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 foo = B.foo;
void test()
{
    foo();   // call B.foo()
    A.foo(); // call A.foo()
    B.foo(); // call B.foo()
}
```

4.4 Public Imports

By default, imports are *private*. This means that if module A imports module B, and module B imports module C, then names inside C are visible only inside B and not inside A.

An import can be explicitly declared *public*, which will cause names from the imported module to be visible to further imports. So in the above example where module A imports module B, if module B *publicly* imports module C, names from C will be visible in A as well.

All symbols from a publicly imported module are also aliased in the importing module. Thus in the above example if C contains the name foo, it will be accessible in A as `foo`, `B.foo` and `C.foo`.

For another example:

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

```
module X;
void bar() { }

module Y;
import W;
public import X;
...
foo();  // calls W.foo()
bar();  // calls X.bar()

module Z;
import Y;
...
foo();   // error, foo() is undefined
bar();   // ok, calls X.bar()
X.bar(); // ditto
Y.bar(); // ok, Y.bar() is an alias to X.bar()
```

4.5 Static Imports

A static import requires the use of a fully qualified name to reference the module's names:

```
static import std.stdio;

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

4.6 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.writeln("hello!");        // ok, calls std.stdio.writeln
```

```
    std.stdio.writeln("hello!"); // error, std is undefined
    writeln("hello!");           // error, writeln is undefined
}
```

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

4.7 Selective Imports

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

```
import std.stdio : writeln, foo = write;

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

`static` cannot be used with selective imports.

4.8 Renamed and Selective Imports

When renaming and selective importing are combined:

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

void main()
{
    writeln("bar");           // error, writeln is undefined
    std.stdio.foo("bar");     // error, foo is bound into current namespace
    std.stdio.writeln("bar"); // error, std is undefined
    foo("bar");               // ok, foo is bound into current namespace,
                              // FQN not required
    io.writeln("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
}
```

4.9 Scoped Imports

Import declarations may be used at any scope. For example:

```
void main()
{
    import std.stdio;
    writeln("bar");
}
```

The imports are looked up to satisfy any unresolved symbols at that scope. Imported symbols may hide symbols from outer scopes.

In function scopes, imported symbols only become visible after the import declaration lexically appears in the function body. In other words, imported symbols at function scope cannot be forward referenced.

```
void main()
{
    void writeln(string) {}
    void foo()
    {
        writeln("bar"); // calls main.writeln
        import std.stdio;
        writeln("bar"); // calls std.stdio.writeln
        void writeln(string) {}
        writeln("bar"); // calls main.foo.writeln
    }
    writeln("bar"); // calls main.writeln
    std.stdio.writeln("bar");  // error, std is undefined
}
```

4.10 Module Scope Operator

A leading `.` causes the identifer to be looked up in the module scope.

```
int x;

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

```
        return .x; // returns global x
}
```

4.11 Static Construction and Destruction

Static constructors are executed to initialize a module's state. Static destructors terminate a module's state.

A module may have multiple static constructors and static destructors. The static constructors are run in lexical order, the static destructors are run in reverse lexical order.

Non-shared static constructors and destructors are run whenever threads are created or destroyed, including the main thread.

Shared static constructors are run once before `main()` is called. Shared static destructors are run after the `main()` function returns.

```
import resource;

Resource x;
shared Resource y;
__gshared Resource z;

static this()  // non-shared static constructor
{
    x = acquireResource();
}

shared static this()  // shared static constructor
{
    y = acquireSharedResource();
    z = acquireSharedResource();
}

static ~this()  // non-shared static destructor
{
    releaseResource(x);
}

shared static ~this()   // shared static destructor
{
    releaseSharedResource(y);
    releaseSharedResource(z);
```

```
}
```

Best Practices:

1. Shared static constructors and destructors are used to initialize and terminate shared global data.
2. Non-shared static constructors and destructors are used to initialize and terminate thread local data.

4.12 Order of Static Construction

Shared static constructors on all modules are run before any non-shared static constructors.

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. There is no other order imposed on the execution of module static constructors.

Cycles (circular dependencies) in the import declarations are allowed so long as neither, or one, but not both, of the modules, contains static constructors or static destructors. Violation of this rule will result in a runtime exception.

Implementation Defined:

1. An implementation may provide a means of overriding the cycle detection abort. A typical method uses the D Runtime switch `--DRT-oncycle=...` where the following behaviors are supported:
   a) `abort` The default behavior. The normal behavior as described in the previous section.
   b) `deprecate` This works just like `abort`, but upon cycle detection the runtime will use a flawed pre-2.072 algorithm to determine if the cycle was previously detected. If no cycles are detected in the old algorithm, execution continues, but a deprecation message is printed.
   c) `print` Print all cycles detected, but do not abort execution. When cycles are present, the order of static construction is implementation defined, and not guaranteed to be valid.
   d) `ignore` Do not abort execution or print any cycles. When cycles are present, the order of static construction is implementation defined, and not guaranteed to be valid.

Best Practices:

1. Avoid cyclical imports where practical. They can be an indication of poor decomposition of a program's structure into independent modules. Two modules that import each other can often be reorganized into three modules without cycles, where the third contains the declarations needed by the other two.

4.13 Order of Static Construction within a Module

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

4.14 Order of Static Destruction

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

Shared static destructors are executed after static destructors.

4.15 Order of Unit tests

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

4.16 Mixin Declaration

```
MixinDeclaration:
    mixin ( ArgumentList ) ;
```

Each *AssignExpression* in the *ArgumentList* is evaluated at compile time, and the result must be representable as a string. The resulting strings are concatenated to form a string. The text contents of the string must be compilable as a valid *DeclDefs*, and is compiled as such.

4.17 Package Module

A package module can be used to publicly import other modules, while providing a simpler import syntax. This enables the conversion of a module into a package of modules, without breaking existing code which uses that module. Example of a set of library modules:

libweb/client.d:

```
module libweb.client;

void runClient() { }
```

libweb/server.d:

```
module libweb.server;

void runServer() { }
```

libweb/package.d:

```
module libweb;

public import libweb.client;
public import libweb.server;
```

The package module's file name must be `package.d`. The module name is declared to be the fully qualified name of the package. Package modules can be imported just like any other modules:

test.d:

```
module test;

// import the package module
import libweb;

void main()
{
    runClient();
    runServer();
}
```

A package module can be nested inside of a sub-package:

libweb/utils/package.d:

```
// must be declared as the fully qualified name of the package, not just 'utils'
module libweb.utils;

// publicly import modules from within the 'libweb.utils' package.
public import libweb.utils.conv;
public import libweb.utils.text;
```

The package module can then be imported with the standard module import declaration:

test.d:

```
module test;

// import the package module
import libweb.utils;

void main() { }
```

Chapter 5

Declarations

Declaration:
    FuncDeclaration
    VarDeclarations
    AliasDeclaration
    AggregateDeclaration
    EnumDeclaration
    ImportDeclaration
    ConditionalDeclaration
    StaticForeachDeclaration
    StaticAssert

VarDeclarations:
    StorageClasses$_{\text{opt}}$ *BasicType* *Declarators* ;
    AutoDeclaration

Declarators:
    DeclaratorInitializer
    DeclaratorInitializer , *DeclaratorIdentifierList*

DeclaratorInitializer:
    VarDeclarator
    VarDeclarator *TemplateParameters*$_{\text{opt}}$ = *Initializer*
    AltDeclarator
    AltDeclarator = *Initializer*

DeclaratorIdentifierList:
  DeclaratorIdentifier
  DeclaratorIdentifier , *DeclaratorIdentifierList*

DeclaratorIdentifier:
  VarDeclaratorIdentifier
  AltDeclaratorIdentifier

VarDeclaratorIdentifier:
  Identifier
  Identifier *TemplateParameters*$_{\text{opt}}$ = *Initializer*

AltDeclaratorIdentifier:
  BasicType2 *Identifier* *AltDeclaratorSuffixes*$_{\text{opt}}$
  BasicType2 *Identifier* *AltDeclaratorSuffixes*$_{\text{opt}}$ = *Initializer*
  BasicType2$_{\text{opt}}$ *Identifier* *AltDeclaratorSuffixes*
  BasicType2$_{\text{opt}}$ *Identifier* *AltDeclaratorSuffixes* = *Initializer*

Declarator:
  VarDeclarator
  AltDeclarator

VarDeclarator:
  BasicType2$_{\text{opt}}$ *Identifier*

AltDeclarator:
  BasicType2$_{\text{opt}}$ *Identifier* *AltDeclaratorSuffixes*
  BasicType2$_{\text{opt}}$ (*AltDeclaratorX*)
  BasicType2$_{\text{opt}}$ (*AltDeclaratorX*) *AltFuncDeclaratorSuffix*
  BasicType2$_{\text{opt}}$ (*AltDeclaratorX*) *AltDeclaratorSuffixes*

AltDeclaratorX:
  BasicType2$_{\text{opt}}$ *Identifier*
  BasicType2$_{\text{opt}}$ *Identifier* *AltFuncDeclaratorSuffix*
  AltDeclarator

AltDeclaratorSuffixes:
  AltDeclaratorSuffix
  AltDeclaratorSuffix *AltDeclaratorSuffixes*

AltDeclaratorSuffix:
  `[ ]`
  `[` *AssignExpression* `]`
  `[` *Type* `]`

AltFuncDeclaratorSuffix:
  Parameters *MemberFunctionAttributes*$_{\text{opt}}$

Type:
  TypeCtors$_{\text{opt}}$ *BasicType* *BasicType2*$_{\text{opt}}$

TypeCtors:
  TypeCtor
  TypeCtor *TypeCtors*

TypeCtor:
  `const`
  `immutable`
  `inout`
  `shared`

BasicType:
  BasicTypeX
  `.` *IdentifierList*

```
    IdentifierList
    Typeof
    Typeof . IdentifierList
    TypeCtor ( Type )
    Vector
    Traits
    MixinType

Vector:
    __vector ( VectorBaseType )

VectorBaseType:
    Type

BasicTypeX:
```

bool	int	ucent	double	cfloat
byte	uint	char	real	cdouble
ubyte	long	wchar	ifloat	creal
short	ulong	dchar	idouble	void
ushort	cent	float	ireal	

```
BasicType2:
    BasicType2X BasicType2opt

BasicType2X:
    *
    [ ]
    [ AssignExpression ]
    [ AssignExpression .. AssignExpression ]
```

[*Type*]
delegate *Parameters* *MemberFunctionAttributes*$_{\text{opt}}$
function *Parameters* *FunctionAttributes*$_{\text{opt}}$

IdentifierList:
Identifier
Identifier . *IdentifierList*
TemplateInstance
TemplateInstance . *IdentifierList*
Identifier [*AssignExpression*]
Identifier [*AssignExpression*]. *IdentifierList*

StorageClasses:
StorageClass
StorageClass *StorageClasses*

StorageClass:

LinkageAttribute	extern	auto	shared	ref
AlignAttribute	abstract	scope	__gshared	
deprecated	final	const	*Property*	
enum	override	immutable	nothrow	
static	synchronized	inout	pure	

Initializer:
VoidInitializer
NonVoidInitializer

NonVoidInitializer:
ExpInitializer

ArrayInitializer
StructInitializer

ExpInitializer:
AssignExpression

ArrayInitializer:
[*ArrayMemberInitializations*$_{\text{opt}}$]

ArrayMemberInitializations:
ArrayMemberInitialization
ArrayMemberInitialization ,
ArrayMemberInitialization , *ArrayMemberInitializations*

ArrayMemberInitialization:
NonVoidInitializer
AssignExpression : *NonVoidInitializer*

StructInitializer:
{ *StructMemberInitializers*$_{\text{opt}}$ }

StructMemberInitializers:
StructMemberInitializer
StructMemberInitializer ,
StructMemberInitializer , *StructMemberInitializers*

StructMemberInitializer:
NonVoidInitializer
Identifier : *NonVoidInitializer*

5.1 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, function pointer and pointer to array declarations are deprecated:

```
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
```

5.2 Implicit Type Inference

AutoDeclaration:
  StorageClasses *AutoDeclarationX* ;

AutoDeclarationX:
  AutoDeclarationY
  AutoDeclarationX , *AutoDeclarationY*

AutoDeclarationY:
  Identifier *TemplateParameters*$_{\text{opt}}$ = *Initializer*

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 = "Apollo"; // s is type immutable(char)[] i.e., string

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.

An *ArrayLiteral* is inferred to be a dynamic array type rather than a static array:

```
auto v = ["resistance", "is", "useless"]; // type is string[], not string[3]
```

5.3 Alias Declarations

AliasDeclaration:
  alias *StorageClasses*$_{\text{opt}}$ *BasicType* *Declarators* ;
  alias *StorageClasses*$_{\text{opt}}$ *BasicType* *FuncDeclarator* ;
  alias *AliasDeclarationX* ;

```
AliasDeclarationX:
    AliasDeclarationY
    AliasDeclarationX , AliasDeclarationY
```

```
AliasDeclarationY:
    Identifier TemplateParameters opt = StorageClasses opt Type
    Identifier TemplateParameters opt = FunctionLiteral
    Identifier TemplateParameters opt = StorageClasses opt BasicType Parameters MemberFunctionAt
```

AliasDeclarations create a symbol that is an alias for another type, and can be used anywhere that other type may appear.

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

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 myint = int;

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

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

```
import planets;

alias myAlbedo = planets.albedo;
...
int len = myAlbedo("Saturn"); // actually calls planets.albedo()
```

The following alias declarations are valid:

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

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

alias Fun = int(string p);
int fun(string){return 0;}
static assert(is(typeof(fun) == Fun));

alias MemberFun1 = int() const;
alias MemberFun2 = const int();
// leading attributes apply to the func, not the return type
static assert(is(MemberFun1 == MemberFun2));
```

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

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

```
alias strlen = string.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 foo = B.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 abc = foo.bar; // 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;
    static int j;
}

alias a = S.i; // OK, `S.i` is a symbol
alias b = S.j; // OK. `S.j` is also a symbol
alias c = a + b; // illegal, `a + b` is an expression
a = 2;         // sets `S.i` to `2`
b = 4;         // sets `S.j` to `4`
```

5.4 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.

An `extern` declaration can optionally be followed by an `extern` linkage attribute. If there is no linkage attribute it defaults to `extern(D)`:

```
// variable allocated and initialized in this module with C linkage
extern(C) int foo;
// variable allocated outside this module with C linkage
// (e.g. in a statically linked C library or another module)
extern extern(C) int bar;
```

Best Practices:

1. The primary usefulness of *Extern Declarations* is to connect with global variables declarations and functions in C or C++ files.

5.5 typeof

```
Typeof:
    typeof ( Expression )
    typeof ( return )
```

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*]

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

The *Expression* is not evaluated, it is used purely to generate the type:

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

Special cases:

1. `typeof(this)` will generate the type of what `this` would be in a non-static member function, even if not in a member function.

2. Analogously, `typeof(super)` will generate the type of what `super` would be in a non-static member function.
3. `typeof(return)` will, when inside a function scope, give the return type of that 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
{
    static 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
```

If the expression is a Property Function, `typeof` gives its return type.

```
struct S
{
    @property int foo() { return 1; }
}
typeof(S.foo) n;  // n is declared to be an int
```

Best Practices:

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

5.6 Void Initializations

```
VoidInitializer:
    void
```

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.

Undefined Behavior: If a void initialized variable's value is used before it is set, the behavior is undefined.

```
void foo()
{
    int x = void;
    writeln(x);  // will print garbage
}
```

Best Practices:

1. Void initializers are useful when a static array is on the stack, but may only be partially used, such as as a temporary buffer. Void initializers will potentially speed up the code, but they introduce risk, since one must ensure that array elements are always set before read.
2. The same is true for structs.
3. Use of void initializers is rarely useful for individual local variables, as a modern optimizer will remove the dead store of its initialization if it is initialized later.
4. For hot code paths, it is worth profiling to see if the void initializer actually improves results.

5.7 Global and Static Initializers

The *Initializer* for a global or static variable must be evaluatable at compile time. Runtime initialization is done with static constructors.

Implementation Defined:

1. Whether some pointers can be initialized with the addresses of other functions or data.

5.8 Type Qualifiers vs. Storage Classes

Type qualifer and storage classes are distinct.

A *type qualifier* creates a derived type from an existing base type, and the resulting type may be used to create multiple instances of that type.

For example, the `immutable` type qualifier can be used to create variables of immutable type, such as:

```
immutable(int)   x; // typeof(x) == immutable(int)
immutable(int)[] y; // typeof(y) == immutable(int)[]
                    // typeof(y[0]) == immutable(int)

// Type constructors create new types that can be aliased:
alias ImmutableInt = immutable(int);
ImmutableInt z;     // typeof(z) == immutable(int)
```

A *storage class*, on the other hand, does not create a new type, but describes only the kind of storage used by the variable or function being declared. For example, a member function can be declared with the `const` storage class to indicate that it does not modify its implicit `this` argument:

```
struct S
{
    int x;
    int method() const
    {
        //x++;    // Error: this method is const and cannot modify this.x
        return x; // OK: we can still read this.x
    }
}
```

Although some keywords can be used both as a type qualifier and a storage class, there are some storage classes that cannot be used to construct new types, such as `ref`:

```
// ref declares the parameter x to be passed by reference
void func(ref int x)
{
    x++; // so modifications to x will be visible in the caller
}

void main()
{
    auto x = 1;
    func(x);
    assert(x == 2);

    // However, ref is not a type qualifier, so the following is illegal:
    ref(int) y; // Error: ref is not a type qualifier.
}
```

Functions can also be declared as `ref`, meaning their return value is passed by reference:

```
ref int func2()
{
    static int y = 0;
    return y;
}

void main()
{
```

```
    func2() = 2; // The return value of func2() can be modified.
    assert(func2() == 2);

    // However, the reference returned by func2() does not propagate to
    // variables, because the 'ref' only applies to the return value itself,
    // not to any subsequent variable created from it:
    auto x = func2();
    static assert(is(typeof(x) == int)); // N.B.: *not* ref(int);
                                         // there is no such type as ref(int).
    x++;
    assert(x == 3);
    assert(func2() == 2); // x is not a reference to what func2() returned; it
                          // does not inherit the ref storage class from func2().
}
```

Some keywords, such as `const`, can be used both as a type qualifier and a storage class. The distinction is determined by the syntax where it appears.

```
struct S
{
    /* Is const here a type qualifier or a storage class?
     * Is the return value const(int), or is this a const function that returns
     * (mutable) int?
     */
    const int* func() // a const function
    {
        ++p;          // error, this.p is const
        return p;     // error, cannot convert const(int)* to int*
    }

    const(int)* func() // a function returning a pointer to a const int
    {
        ++p;          // ok, this.p is mutable
        return p;     // ok, int* can be implicitly converted to const(int)*
    }

    int* p;
}
```

Best Practices: To avoid confusion, the type qualifier syntax with parentheses should be used for return types, and function storage classes should be written on the right-hand side of the declaration instead of the left-hand side where it may be visually confused with the return type:

```
struct S
{
    // Now it is clear that the 'const' here applies to the return type:
    const(int) func1() { return 1; }

    // And it is clear that the 'const' here applies to the function:
    int func2() const { return 1; }
}
```

Chapter 6

Types

D is statically typed. Every expression has a type. Types constrain the values an expression can hold, and determine the semantics of operations on those values.

Basic data types are leaf types. Derived data types build on leaf types. User defined types are aggregates of basic and derived types.

6.1 Basic Data Types

Basic Data Types

Keyword	Default Initializer (`.init`)	Description
`void`	-	no type
`bool`	`false`	boolean value
`byte`	`0`	signed 8 bits
`ubyte`	`0u`	unsigned 8 bits
`short`	`0`	signed 16 bits
`ushort`	`0u`	unsigned 16 bits
`int`	`0`	signed 32 bits
`uint`	`0u`	unsigned 32 bits
`long`	`0L`	signed 64 bits
`ulong`	`0uL`	unsigned 64 bits
`cent`	`0`	signed 128 bits (reserved for future use)
`ucent`	`0u`	unsigned 128 bits (reserved for future use)
`float`	`float.nan`	32 bit floating point
`double`	`double.nan`	64 bit floating point
`real`	`real.nan`	largest FP size implemented in hardware[1]
`ifloat`	`float.nan*1.0i`	imaginary float
`idouble`	`double.nan*1.0i`	imaginary double
`ireal`	`real.nan*1.0i`	imaginary real
`cfloat`	`float.nan+float.nan*1.0i`	a complex number of two float values
`cdouble`	`double.nan+double.nan*1.0i`	complex double
`creal`	`real.nan+real.nan*1.0i`	complex real
`char`	`'xFF'`	unsigned 8 bit (UTF-8 code unit)
`wchar`	`'uFFFF'`	unsigned 16 bit (UTF-16 code unit)
`dchar`	`'U0000FFFF'`	unsigned 32 bit (UTF-32 code unit)

6.2 Derived Data Types

- pointer
- array
- associative array
- function
- delegate

Strings are a special case of arrays.

[1]Implementation Note: 80 bits for x86 CPUs or `double` size, whichever is larger

6.3 User-Defined Types

- enum
- struct
- union
- class

6.4 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
```

6.5 Pointer Conversions

Casting pointers to non-pointers and vice versa is allowed.

Best Practices: do not do this for any pointers that point to data allocated by the garbage collector.

6.6 Implicit Conversions

Implicit conversions are used to automatically convert types as required.

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

```
int i;

enum Foo { E }
Foo f;
i = f;           // OK
f = i;           // error
f = cast(Foo)i;  // OK
f = 0;           // error
f = Foo.E;       // OK
```

6.7 Integer Promotions

Integer Promotions are conversions of the following types:

Integer Promotions

from	to
`bool`	`int`
`byte`	`int`
`ubyte`	`int`
`short`	`int`
`ushort`	`int`
`char`	`int`
`wchar`	`int`
`dchar`	`uint`

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

6.8 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:
   a) If both are the same type, no more conversions are done.
   b) If both are signed or both are unsigned, the smaller type is converted to the larger.
   c) If the signed type is larger than the unsigned type, the unsigned type is converted to the signed type.
   d) The signed type is converted to the unsigned type.

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

1. If the operands are the same type, the result will be of that type.
2. If one operand is an enum and the other is the base type of that enum, the result is the base type.
3. If the two operands are different enums, the result is the closest base type common to both. A base type being closer means there is a shorter sequence of conversions to base type to get there from the original type.

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

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

Floating point types cannot be implicitly converted to integral types. Complex or imaginary floating point types cannot be implicitly converted to non-complex floating point types. Non-complex floating point types cannot be implicitly converted to imaginary floating point types.

6.9 `bool`

The bool type is a 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.

6.10 Delegates

Delegates are an aggregate of two pieces of data: an object reference and a pointer to a non-static member function, or a pointer to a closure and a pointer to a nested function. The object reference forms the `this` pointer when the function is called.

Delegates are declared similarly to function pointers:

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

The equivalent of member function pointers can be constructed using anonymous lambda functions:

```
class C
{
    int a;
    int foo(int i) { return i + a; }
}

// mfp is the member function pointer
auto mfp = function(C self, int i) { return self.foo(i); };
auto c = new C();  // create an instance of C
mfp(c, 1);  // and call c.foo(1)
```

The C style syntax for declaring pointers to functions is deprecated:

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

6.11 Mixin Types

```
MixinType:
    mixin ( ArgumentList )
```

Each *AssignExpression* in the *ArgumentList* is evaluated at compile time, and the result must be representable as a string. The resulting strings are concatenated to form a string. The text contents of the string must be compilable as a valid *Type*, and is compiled as such.

```
void test(mixin("int")* p) // int* p
{
    mixin("int")[] a;      // int[] a;
    mixin("int[]") b;      // int[] b;
}
```

6.12 size_t

`size_t` is an alias to one of the unsigned integral basic types, and represents a type that is large enough to represent an offset into all addressable memory.

6.13 ptrdiff_t

`ptrdiff_t` is an alias to the signed integral basic type the same size as `size_t`.

Chapter 7

Properties

Every symbol, type, and expression has properties that can be queried:

Property Examples

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)
`int.init`	default initializer for ints
`int.mangleof`	yields the string "i"
`int.stringof`	yields the string "int"
`(1+2).stringof`	yields the string "1 + 2"

Properties for All Types

Property	Description
`.init`	initializer
`.sizeof`	size in bytes
`.alignof`	alignment size
`.mangleof`	string representing the 'mangled' representation of the type
`.stringof`	string representing the source representation of the type

Properties for Integral Types

Property	Description
`.init`	initializer
`.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_normal`	smallest representable normalized value that's not 0
`.re`	real part
`.im`	imaginary part

Properties for Class Types

Property	Description
`.classinfo`	Information about the dynamic type of the class

7.1 .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's type.

```
int a;
int b = 1;
```

```
int.init // is 0
a.init   // is 0
b.init   // is 0

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

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

Note: `.init` produces a default initialized object, not default constructed. If there is a default constructor for an object, it may produce a different value.

1. If `T` is a nested struct, the context pointer in `T.init` is `null`.

   ```
   void main()
   {
       int x;
       struct S
       {
           void foo() { x = 1; }  // access x in enclosing scope via context pointer
       }
       S s1;           // OK. S() correctly initialize its context pointer.
       S s2 = S();     // OK. same as s1
       S s3 = S.init;  // Bad. the context pointer in s3 is null
       s3.foo();       // Access violation
   }
   ```

2. If `T` is a struct which has `@disable this();`, `T.init` might return a logically incorrect object.

   ```
   struct S
   {
       int x;
       @disable this();
       this(int n) { x = n; }
       invariant { assert(x > 0); }
       void check() {}
   }
   ```

```
void main()
{
  //S s1;           // Error: variable s1 initializer required for type S
  //S s2 = S();     // Error: constructor S.this is not callable
                    // because it is annotated with @disable
    S s3 = S.init;  // Bad. s3.x == 0, and it violates the invariant of S
    s3.check();     // Assertion failure
}
```

7.2 .stringof Property

`.stringof` produces a constant string that is the source representation of its prefix. If applied to a type, it is the string for that type. If applied to an expression, it is the source representation of that expression. The expression will not be evaluated.

```
module test;
import std.stdio;

struct Dog { }

enum Color { Red }

int b = 4;

void main()
{
    writeln((1+2).stringof);       // "1 + 2"
    writeln(Dog.stringof);         // "Dog"
    writeln(test.Dog.stringof);    // "Dog"
    writeln(int.stringof);         // "int"
    writeln((int*[5][]).stringof); // "int*[5][]"
    writeln(Color.Red.stringof);   // "Red"
    writeln((5).stringof);         // "5"

    writeln((++i).stringof);       // "i += 1"
    writeln(i);                    // 4
}
```

Implementation Defined: The string representation for a type or expression can vary.

Best Practices: Do not use `.stringof` for code generation. Instead use the identifier trait, or one of the Phobos helper functions such as `std.traits.fullyQualifiedName`.

7.3 .sizeof Property

`e.sizeof` gives the size in bytes of the expression `e`.

When getting the size of a member, it is not necessary for there to be a *this* object:

```
struct S
{
    int a;
    static int foo()
    {
        return a.sizeof; // returns 4
    }
}

void test()
{
    int x = S.a.sizeof; // sets x to 4
}
```

`.sizeof` applied to a class object returns the size of the class reference, not the class instantiation.

7.4 .alignof Property

`.alignof` gives the aligned size of an expression or type. For example, an aligned size of 1 means that it is aligned on a byte boundary, 4 means it is aligned on a 32 bit boundary.

Implementation Defined: the actual aligned size.

Best Practices: Be particularly careful when laying out an object that must line up with an externally imposed layout. Data misalignment can result in particularly pernicious bugs. It's often worth putting in an `assert` to assure it is correct.

7.5 .mangleof Property

Mangling refers to how a symbol is represented in text form in the generated object file. `.mangleof` returns a string literal of the representation of the type or symbol it is applied to. The mangling of types and symbols with D linkage is defined by Name Mangling.

Implementation Defined:

1. whether a leading underscore is added to a symbol
2. the mangling of types and symbols with non-D linkage. For C and C++ linkage, this will typically match what the associated C or C++ compiler does.

7.6 .classinfo Property

`.classinfo` provides information about the dynamic type of a class object. It returns a reference to type `object.TypeInfo_Class`.

`.classinfo` applied to an interface gives the information for the interface, not the class it might be an instance of.

7.7 User-Defined Properties

User-defined properties can be created using Property Functions.

Chapter 8

Attributes

```
AttributeSpecifier:
    Attribute :
    Attribute DeclarationBlock

Attribute:
    LinkageAttribute
    AlignAttribute
    DeprecatedAttribute
    VisibilityAttribute
    Pragma
    static
    extern
    abstract
    final
    override
    synchronized
    auto
    scope
    const
    immutable
    inout
    shared
    __gshared
    AtAttribute
    nothrow
```

```
    pure
    ref
    return
```

AtAttribute:
```
    @ disable
    @ nogc
    Property
    @ safe
    @ system
    @ trusted
    UserDefinedAttribute
```

Property:
```
    @ property
```

DeclarationBlock:
```
    DeclDef
    { DeclDefs opt }
```

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 end of
               // the current scope
  declaration;
  declaration;
  ...

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

8.1 Linkage Attribute

```
LinkageAttribute:
    extern ( LinkageType )
    extern ( C++, IdentifierList )


LinkageType:
    C
    C++
    D
    Windows
    System
    Objective-C
```

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++` offers limited compatibility with C++. `Objective-C` offers limited compatibility with Objective-C, see the Interfacing to Objective-C documentation for more information. `System` is the same as `Windows` on Windows platforms, and `C` on other platforms. `Implementation Note:` for Win32 platforms, `Windows` should exist.

C function calling conventions are specified by:

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

Note that `extern(C)` can be provided for all types of declarations, including `struct` or `class`, even though there is no corresponding match on the `C` side. In that case, the attribute is ignored. This behavior applies for nested functions and nested variables as well. However, for `static` member methods and `static` nested functions, adding `extern(C)` will change the calling convention, but not the mangling.

D conventions are:

```
extern (D):
```

Windows API conventions are:

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

```
        uint flAllocationType,
        uint flProtect
    );
```

The Windows convention is distinct from the C convention only on Win32 platforms, where it is equivalent to the stdcall convention.

Note that a lone `extern` declaration is used as a storage class.

C++ Namespaces

The linkage form `extern (C++,` *IdentifierList*`)` creates C++ declarations that reside in C++ namespaces. The *IdentifierList* specifies the namespaces.

```
extern (C++, N) { void foo(); }
```

refers to the C++ declaration:

```
namespace N { void foo(); }
```

and can be referred to with or without qualification:

```
foo();
N.foo();
```

Namespaces create a new named scope that is imported into its enclosing scope.

```
extern (C++, N) { void foo(); void bar(); }
extern (C++, M) { void foo(); }

bar();   // ok
foo();   // error - N.foo() or M.foo() ?
M.foo(); // ok
```

Multiple identifiers in the *IdentifierList* create nested namespaces:

```
extern (C++, N.M) { extern (C++) { extern (C++, R) { void foo(); } } }
N.M.R.foo();
```

refers to the C++ declaration:

```
namespace N { namespace M { namespace R { void foo(); } } }
```

8.2 `align` Attribute

```
AlignAttribute:
    align
    align ( AssignExpression )
```

Specifies the alignment of:

1. variables
2. struct fields
3. union fields
4. class fields
5. struct, union, and class types

`align` by itself sets it to the default, which matches the default member alignment of the companion C compiler.

```
struct S
{
  align:
    byte a;   // placed at offset 0
    int b;    // placed at offset 4
    long c;   // placed at offset 8
}
auto sz = S.sizeof;  // 16
```

AssignExpression specifies the alignment which matches the behavior of the companion C compiler when non-default alignments are used. It must be a positive power of 2.

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

```
struct S
{
  align (1):
    byte a;   // placed at offset 0
    int b;    // placed at offset 1
    long c;   // placed at offset 5
}
auto sz = S.sizeof;  // 16
```

The alignment for the fields of an aggregate does not affect the alignment of the aggregate itself - that is affected by the alignment setting outside of the aggregate.

```
align (2) struct S
{
  align (1):
    byte a;   // placed at offset 0
    int b;    // placed at offset 1
    long c;   // placed at offset 5
}
```

```
auto sz = S.sizeof;  // 14
```

Setting the alignment of a field aligns it to that power of 2, regardless of the size of the field.

```
struct S
{
  align (4):
    byte a;   // placed at offset 0
    byte b;   // placed at offset 4
    short c;  // placed at offset 8
}
auto sz = S.sizeof;  // 12
```

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.

Undefined Behavior: If any pointers and references to GC allocated objects are not aligned on `size_t` byte boundaries.

The *AlignAttribute* is reset to the default when entering a function scope or a non-anonymous struct, union, class, and restored when exiting that scope. It is not inherited from a base class.

8.3 `deprecated` Attribute

```
DeprecatedAttribute:
    deprecated
    deprecated ( AssignExpression )
```

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 instructed to produce an error if any code refers to deprecated declarations:

```
deprecated
{
    void oldFoo();
}

oldFoo();   // Deprecated: function test.oldFoo is deprecated
```

Optionally a string literal or manifest constant can be used to provide additional information in the deprecation message.

```
deprecated("Don't use bar") void oldBar();
oldBar();   // Deprecated: function test.oldBar is deprecated - Don't use bar
```

Calling CTFE-able functions or using manifest constants is also possible.

```
import std.format;
enum Message = format("%s␣and␣all␣its␣members␣are␣obsolete", Foobar.stringof);
deprecated(Message) class Foobar {}
auto f = new Foobar();   // Deprecated: class test.Foobar is deprecated - Foobar
     // and all its members are obsolete
deprecated(format("%s␣is␣also␣obsolete", "This␣class")) class BarFoo {}
auto bf = new BarFoo();  // Deprecated: class test.BarFoo is deprecated - This
     // class is also obsolete
```

`Implementation Note:` The compiler should have a switch specifying if `deprecated` should be ignored, cause a warning, or cause an error during compilation.

8.4 Visibility Attribute

```
VisibilityAttribute:
    private
    package
    package ( IdentifierList )
    protected
    public
    export
```

Visibility is an attribute that is one of `private`, `package`, `protected`, `public`, or `export`. They may be referred to as protection attributes in documents predating DIP22.

Symbols with `private` visibility can only be accessed from within the same module. Private member functions are implicitly final and cannot be overridden.

`package` extends private so that package members can be accessed from code in other modules that are in the same package. If no identifier is provided, this applies to the innermost package only, or defaults to `private` if a module is not nested in a package.

`package` may have an optional parameter in the form of a dot-separated identifier list which is resolved as the qualified package name. The package must be either the module's parent package or one of its anscestors. If this optional parameter is present, the symbol will be visible in the specified package and all of its descendants.

`protected` only applies inside classes (and templates as they can be mixed in) and means that a symbol can only be seen by members of the same module, or by a derived class. If accessing a protected instance member through a derived class member function, that member can only be accessed for the object instance which can be implicitly cast to the same type as 'this'. `protected` module members are illegal.

`public` means that any code within the executable can see the member. It is the default visibility attribute.

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

Visibility participates in symbol name lookup.

8.5 `const` Attribute

The `const` attribute changes the type of the declared symbol from `T` to `const(T)`, where `T` is the type specified (or inferred) for the introduced symbol in the absence of `const`.

```
const int foo = 7;
static assert(is(typeof(foo) == const(int)));

const
{
    double bar = foo + 6;
}
static assert(is(typeof(bar) == const(double)));

class C
{
    const void foo();
    const
    {
        void bar();
    }
    void baz() const;
}
pragma(msg, typeof(C.foo)); // const void()
pragma(msg, typeof(C.bar)); // const void()
pragma(msg, typeof(C.baz)); // const void()
static assert(is(typeof(C.foo) == typeof(C.bar)) &&
              is(typeof(C.bar) == typeof(C.baz)));
```

8.6 `immutable` Attribute

The `immutable` attribute modifies the type from `T` to `immutable(T)`, the same way as `const` does.

8.7 inout Attribute

The `inout` attribute modifies the type from `T` to `inout(T)`, the same way as `const` does.

8.8 shared Attribute

The `shared` attribute modifies the type from `T` to `shared(T)`, the same way as `const` does.

8.9 __gshared Attribute

By default, non-immutable global declarations reside in thread local storage. When a global variable is marked with the `__gshared` attribute, its value is shared across all threads.

```
int foo;            // Each thread has its own exclusive copy of foo.
__gshared int bar;  // bar is shared by all threads.
```

`__gshared` may also be applied to member variables and local variables. In these cases, `__gshared` is equivalent to `static`, except that the variable is shared by all threads rather than being thread local.

```
class Foo
{
    __gshared int bar;
}

int foo()
{
    __gshared int bar = 0;
    return bar++; // Not thread safe.
}
```

Unlike the `shared` attribute, `__gshared` provides no safe-guards against data races or other multi-threaded synchronization issues. It is the responsibility of the programmer to ensure that access to variables marked `__gshared` is synchronized correctly.

`__gshared` is disallowed in safe mode.

8.10 @disable Attribute

A reference to a declaration marked with the `@disable` attribute causes a compile time error. This can be used to explicitly disallow certain operations or overloads at compile time rather than relying on generating a runtime error.

```
@disable void foo() { }

void main() { foo(); /* error, foo is disabled */ }
```

Disabling struct no-arg constructor disallows default construction of the struct.

Disabling struct postblit makes the struct not copyable.

8.11 `@safe`, `@trusted`, and `@system` Attribute

See Function Safety.

8.12 `@nogc` Attribute

See No-GC Functions.

8.13 `@property` Attribute

See Property Functions.

8.14 `nothrow` Attribute

See Nothrow Functions.

8.15 `pure` Attribute

See Pure Functions.

8.16 `ref` Attribute

See Ref Functions.

8.17 `return` Attribute

See Return Ref Parameters.

8.18 override Attribute

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
    }
}
```

8.19 static Attribute

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 one instance per thread, not one 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
```

8.20 `auto` Attribute

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
```

For functions, the `auto` attribute means return type inference. See Auto Functions.

8.21 `scope` Attribute

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, ref 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.

8.22 `abstract` Attribute

An abstract member function must be overridden by a derived class. Only virtual member functions may be declared abstract; non-virtual member functions and free-standing functions cannot be declared abstract.

Classes become abstract if any of its virtual member functions are declared abstract or if they are defined within an abstract attribute. Note that an abstract class may also contain non-virtual member functions.

Classes defined within an abstract attribute or with abstract member functions cannot be instantiated directly. They can only be instantiated as a base class of another, non-abstract, class.

Member 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', e.g. through `super.foo()` in a derived class. Note that the class is still abstract and cannot be instantiated directly.

8.23 User-Defined Attributes

UserDefinedAttribute:
  `@ (` *ArgumentList* `)`
  `@` *Identifier*
  `@` *Identifier* `(` *ArgumentList* $_{\text{opt}}$ `)`
  `@` *TemplateInstance*
  `@` *TemplateInstance* `(` *ArgumentList* $_{\text{opt}}$ `)`

User-Defined Attributes (UDA) are compile-time expressions that can be attached to a declaration. These attributes can then be queried, extracted, and manipulated at compile time. There is no runtime component to them.

A user-defined attribute looks like:

```
@(3) int a;

@("string", 7) int b;

enum Foo;
@Foo int c;

struct Bar
{
    int x;
}

@Bar(3) int d;
```

If there are multiple UDAs in scope for a declaration, they are concatenated:

```
@(1)
{
    @(2) int a;         // has UDAs (1, 2)
    @("string") int b;  // has UDAs (1, "string")
}
```

UDAs can be extracted into an expression tuple using `__traits`:

```
@('c') string s;
pragma(msg, __traits(getAttributes, s)); // prints tuple('c')
```

If there are no user-defined attributes for the symbol, an empty tuple is returned. The expression tuple can be turned into a manipulatable tuple:

```
enum EEE = 7;
@("hello") struct SSS { }
@(3) { @(4) @EEE @SSS int foo; }

alias TP = __traits(getAttributes, foo);

pragma(msg, TP); // prints tuple(3, 4, 7, (SSS))
pragma(msg, TP[2]); // prints 7
```

Of course the tuple types can be used to declare things:

```
TP[3] a; // a is declared as an SSS
```

The attribute of the type name is not the same as the attribute of the variable:

```
pragma(msg, __traits(getAttributes, typeof(a))); // prints tuple("hello")
```

Of course, the real value of UDAs is to be able to create user-defined types with specific values. Having attribute values of basic types does not scale. The attribute tuples can be manipulated like any other tuple, and can be passed as the argument list to a template.

Whether the attributes are values or types is up to the user, and whether later attributes accumulate or override earlier ones is also up to how the user interprets them.

UDAs cannot be attached to template parameters.

Chapter 9

Pragmas

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

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.

Pragmas can appear as either declarations, *Pragma DeclarationBlock*, or as statements, *PragmaStatement*.

```
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 determined by the *Identifier*. *ArgumentList* is a comma-separated list of *AssignExpression*s. The *AssignExpression*s must be parsable as expressions, but their meaning is up to the individual pragma semantics.

9.1 Predefined Pragmas

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

- pragma crt_constructor
- pragma crt_destructor
- pragma inline
- pragma lib
- pragma linkerDirective
- pragma mangle
- pragma msg
- pragma printf
- pragma scanf
- pragma startaddress

Implementation Defined: An implementation may ignore these pragmas.

`pragma crt_constructor`

This pragma must directly precede an `extern(C)` function declaration that must take no argument, even default ones. The function this pragma applies to will be inserted in `.init_array` or `.ctors`, depending on the target and compiler implementation. It is equivalent to GCC's `__attribute__((constructor))`.

```
__gshared int initCount;

pragma(crt_constructor)
extern(C) void initializer() { initCount += 1; }
```

It is useful for system programming and interfacing with C/C++, for example to allow for initialization of the runtime when loading a DSO, or as a simple replacement for `shared static this` in betterC mode.

A module may contain any number of functions annotated with `crt_constructor`. The order in which functions are called is undefined and shouldn't be relied upon. A function can be annotated as both `crt_constructor` and `crt_destructor` (see below). The runtime is not initialized when the function is called. This pragma does not take any argument and can only be applied to a single declaration, so using them in an *AttributeSpecifier* is disallowed.

`crt_constructor` and `crt_destructor` were implemented in v2.078.0 (2018-01-01). Some compilers exposed non-standard, compiler-specific mechanism before.

pragma crt_destructor

Similarly, `pragma(crt_destructor)` must also directly precede an `extern(C)` function declaration that must take no argument, even default ones. The function this pragma applies to will be inserted in `.fini_array` or `.dtors`, depending on the target and compiler implementation. It is equivalent to GCC's `__attribute__((destructor))`.

```
__gshared int initCount;

pragma(crt_constructor)
extern(C) void initialize() { initCount += 1; }

pragma(crt_destructor)
extern(C) void deinitialize() { initCount -= 1; }

pragma(crt_constructor)
pragma(crt_destructor)
extern(C) void innuendo() { printf("Inside a constructor... Or destructor?\n"); }
```

The runtime might have been terminated and not be usable anymore when the destructors are called. Otherwise, usage and requirements of `crt_destructor` are similar to those of `crt_constructor`.

pragma inline

Affects whether functions are inlined or not. If at the declaration level, it affects the functions declared in the block it controls. If inside a function, it affects the function it is enclosed by.

It takes three forms:

1. `pragma(inline)`

   Sets the behavior to match the implementation's default behavior.

2. `pragma(inline, false)`

Functions are never inlined.

3. `pragma(inline, true)`

   Always inline the functions.

There can be only zero or one *AssignExpression*s. If one is there, it must be `true`, `false`, or an integer value. An integer value is implicitly converted to a bool.

If there are multiple pragma inlines in a function, the lexically last one takes effect.

```
pragma(inline):
int foo(int x) // foo() is never inlined
{
    pragma(inline, true);
    ++x;
    pragma(inline, false); // supercedes the others
    return x + 3;
}
```

Implementation Defined:

1. The default inline behavior is typically selectable with a compiler switch such as `-inline`.
2. Whether a particular function can be inlined or not is implementation defined.
3. What happens for `pragma(inline, true)` if the function cannot be inlined. An error message is typical.

`pragma lib`

There must be one *AssignExpression* and it must evaluate at compile time to a string literal.

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

Implementation Defined: Typically, the string literal specifies the file name of a library file. This name is inserted into the generated object file, or otherwise is passed to the linker, so the linker automatically links in that library.

`pragma linkerDirective`

There must be one *AssignExpression* and it must evaluate at compile time to a string literal.

```
pragma(linkerDirective, "/FAILIFMISMATCH:_ITERATOR_DEBUG_LEVEL=2");
```

Implementation Defined: The string literal specifies a linker directive to be embedded in the generated object file.

Linker directives are only supported for MS-COFF output.

`pragma mangle`

Overrides the default mangling for a symbol.

There must be one *AssignExpression* and it must evaluate at compile time to a string literal.

Implementation Defined: On macOS and Win32, an extra underscore (`_`) is prepended to the string since 2.079, as is done by the C/C++ toolchain. This allows using the same `pragma(mangle)` for all compatible (POSIX in one case, win64 in another) platforms instead of having to special-case.

Implementation Defined: It's only effective when the symbol is a function declaration or a variable declaration. For example this allows linking to a symbol which is a D keyword, which would normally be disallowed as a symbol name:

```
pragma(mangle, "body")
extern(C) void body_func();
```

`pragma msg`

Constructs a message from the *ArgumentList*.

```
pragma(msg, "compiling...", 1, 1.0);
```

Implementation Defined: The arguments are typically presented to the user during compilation, such as by printing them to the standard error stream.

`pragma printf`

`pragma(printf)` specifies that a function declaration is a printf-like function, meaning it is an `extern (C)` or `extern (C++)` function with a `format` parameter accepting a pointer to a 0-terminated `char` string conforming to the C99 Standard 7.19.6.1, immediately followed by either a `...` variadic argument list or a parameter of type `va_list` as the last parameter.

If the `format` argument is a string literal, it is verified to be a valid format string per the C99 Standard. If the `format` parameter is followed by `...`, the number and types of the variadic arguments are checked against the format string.

Diagnosed incompatibilities are:

- incompatible sizes which may cause argument misalignment
- deferencing arguments that are not pointers
- insufficient number of arguments
- struct arguments
- array and slice arguments
- non-pointer arguments to `s` specifier
- non-standard formats
- undefined behavior per C99

Per the C99 Standard, extra arguments are ignored.

Ignored mismatches are:

- sign mismatches, such as printing an `int` with a `%u` format
- integral promotion mismatches, where the format specifies a smaller integral type than `int` or `uint`, such as printing a `short` with the `%d` format rather than `%hd`

```
printf("%k\n", value); // error: non-Standard format k
printf("%d\n");        // error: not enough arguments
printf("%d\n", 1, 2);  // ok, extra arguments ignored
```

Best Practices: In order to use non-Standard printf/scanf formats, an easy workaround is:

```
const format = "%k\n";
printf(format.ptr, value);  // no error
```

Best Practices: Most of the errors detected are portability issues. For instance,

```
string s;
printf("%.*s\n", s.length, s.ptr);
printf("%d\n", s.sizeof);
ulong u;
scanf("%lld%*c\n", &u);
```

should be replaced with:

```
string s;
printf("%.*s\n", cast(int) s.length, s.ptr);
printf("%zd\n", s.sizeof);
ulong u;
scanf("%llu%*c\n", &u);
```

`pragma(printf)` applied to declarations that are not functions are ignored. In particular, it has no effect on the declaration of a pointer to function type.

`pragma scanf`

`pragma(scanf)` specifies that a function declaration is a scanf-like function, meaning it is an `extern (C)` or `extern (C++)` function with a `format` parameter accepting a pointer to a 0-terminated `char` string conforming to the C99 Standard 7.19.6.2, immediately followed by either a `...` variadic argument list or a parameter of type `va_list` as the last parameter.

If the `format` argument is a string literal, it is verified to be a valid format string per the C99 Standard. If the `format` parameter is followed by `...`, the number and types of the variadic arguments are checked against the format string.

Diagnosed incompatibilities are:

- argument is not a pointer to the format specified type
- insufficient number of arguments
- non-standard formats
- undefined behavior per C99

Per the C99 Standard, extra arguments are ignored.

`pragma(scanf)` applied to declarations that are not functions are ignored. In particular, it has no effect on the declaration of a pointer to function type.

`pragma startaddress`

There must be one *AssignExpression* and it must evaluate at compile time to a function symbol.

Implementation Defined: The function symbol specifies the start address for the program. The symbol is inserted into the object file or is otherwise presented to the linker to set the start address. This is not normally used for application level programming, but is for specialized systems work. For applications code, the start address is taken care of by the runtime library.

```
void foo() { ... }
pragma(startaddress, foo);
```

9.2 Vendor Specific Pragmas

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

```
pragma(DigitalMars_extension) { ... }
```

Implementations must diagnose an error for unrecognized *Pragma*s, even if they are vendor specific ones.

Implementation Defined: Vendor specific pragmas.

Best Practices: vendor specific pragmas should be wrapped in version statements

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

Chapter 10

Expressions

An expression is a sequence of operators and operands that specifies an evaluation. The syntax, order of evaluation, and semantics of expressions are as follows.

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.

10.1 Definitions and Terms

<div class="quickindex" id="quickindex.define-full-expression"></div>**Definition** ("Full expression"): For any expression *expr*, the full expression of *expr* is defined as follows. If *expr* parses as a subexpression of another expression *expr₁*, then the full expression of *expr* is the full expression of *expr₁*. Otherwise, *expr* is its own full expression.

Each expression has a unique full expression.

Example: in the statement `return f() + g() * 2;`, the full expression of `g() * 2` is `f() + g() * 2`, but not the full expression of `f() + g()` because the latter is not parsed as a subexpression.

Note: Although the definition is straightforward, a few subtleties exist related to function literals. In the statement `return (() => x + f())() * g();`, the full expression of `f()` is `x + f()`, not the expression passed to `return`. This is because the parent of `x + f()` has function literal type, not expression type.

<div class="quickindex" id="quickindex.define-lvalue"></div>**Definition** ("Lvalue"): The following expressions, and no others, are called lvalue expressions or lvalues:

1. `this` inside `struct` and `union` member functions;

2. a variable or the result of the *DotIdentifier* grammatical construct `.` (left side may be missing) when the rightmost side of the dot is a variable, field (direct or `static`), function name, or invocation of a function that returns by reference;
3. the result of the following expressions:

   - built-in unary operators `+` (when applied to an lvalue), `*`, `++` (prefix only), `--` (prefix only);
   - built-in indexing operator `[]` (but not the slicing operator);
   - built-in assignment binary operators, i.e. `=`, `+=`, `*=`, `/=`, `%=`, `&=`, =|, `^=`, `~=`, `<<=`, `>>=`, `>>>=`, and `^^=`;
   - the ternary operator *e* `?` e_1 `:` e_2 under the following circumstances:
     a) e_1 and e_2 are lvalues of the same type; OR
     b) One of e_1 and e_2 is an lvalue of type `T` and the other has and `alias this` converting it to `ref T`;
   - user-defined operators if and only if the function called as a result of lowering returns by reference;
   - `mixin` expressions if and only if the compilation of the expression resulting from compiling the argument(s) to `mixin` is an lvalue;
   - `cast(U)` expressions applied to lvalues of type `T` when `T*` is implicitly convertible to `U*`;
   - `cast()` and `cast(`*qualifier list*`)` when applied to an lvalue.

<div class="quickindex" id="quickindex.define-rvalue"></div>**Definition** ("Rvalue"): Expressions that are not lvalues are rvalues.

Note: Rvalues include all literals, special value keywords such as `__FILE__` and `__LINE__`, `enum` values, and the result of expressions not defined as lvalues above.

The built-in address-of operator (unary `&`) may only be applied to lvalues.

<div class="quickindex" id="quickindex.define-smallest-short-circuit"></div>**Definition** ("Smallest short-circuit expression"): Given an expression *expr* that is a subexpression of a full expression *fullexpr*, the smallest short-circuit expression, if any, is the shortest subexpression *scexpr* of *fullexpr* that is an *AndAndExpression* (`&&`) or an *OrOrExpression* (||), such that *expr* is a subexpression of *scexpr*.

Example: in the expression `((f() * 2 && g()) + 1)` | h()|, the smallest short-circuit expression of the subexpression `f() * 2` is `f() * 2 && g()`. In the expression `(f() && g()) + h()`, the subexpression `h()` has no smallest short-circuit expression.

10.2 Order Of Evaluation

Built-in prefix unary expressions `++` and `--` are evaluated as if lowered (rewritten) to assignments as follows: `++expr` becomes `((expr) += 1)`, and `--expr` becomes `((expr) -= 1)`. Therefore, the result of prefix `++` and `--` is the lvalue after the side effect has been effected.

Built-in postfix unary expressions `++` and `--` are evaluated as if lowered (rewritten) to lambda invocations as follows: `expr++` becomes `(ref T x){auto t = x; ++x; return t;}(expr)`, and `expr--` becomes `(ref T x){auto t = x; --x; return t;}(expr)`. Therefore, the result of postfix `++` and `--` is an rvalue just before the side effect has been effected.

Binary expressions except for *AssignExpression*, *OrOrExpression*, and *AndAndExpression* are evaluated in lexical order (left-to-right). Example:

```
int i = 2;
i = ++i * i++ + i;
assert(i == 3 * 3 + 4);
```

OrOrExpression and *AndAndExpression* evaluate their left-hand side argument first. Then, *OrOrExpression* evaluates its right-hand side if and only if its left-hand side does not evaluate to nonzero. *AndAndExpression* evaluates its right-hand side if and only if its left-hand side evaluates to nonzero.

ConditionalExpression evaluates its left-hand side argument first. Then, if the result is nonzero, the second operand is evaluated. Otherwise, the third operand is evaluated.

Calls to functions with `extern(D)` linkage (which is the default linkage) are evaluated in the following order: first, if necessary, the address of the function to call is evaluated (e.g. in the case of a computed function pointer or delegate). Then, arguments are evaluated left to right. Finally, transfer is passed to the function. Example:

```
import std.stdio;
void function(int a, int b, int c) fun()
{
    writeln("fun()␣called");
    static void r(int a, int b, int c) { writeln("callee␣called"); }
    return &r;
}
int f1() { writeln("f1()␣called"); return 1; }
int f2() { writeln("f2()␣called"); return 2; }
int f3(int x) { writeln("f3()␣called"); return x + 3; }
int f4() { writeln("f4()␣called"); return 4; }
// evaluates fun() then f1() then f2() then f3() then f4()
// after which control is transferred to the callee
fun()(f1(), f3(f2()), f4());
```

Implementation Defined:

1. The order of evaluation of the operands of *AssignExpression*.
2. The order of evaluation of function arguments for functions with linkage other than `extern (D)`.

Best Practices: Even though the order of evaluation is well-defined, writing code that depends on it is rarely recommended.

10.3 Lifetime of Temporaries

Expressions and statements may create and/or consume rvalues. Such values are called *temporaries* and do not have a name or a visible scope. Their lifetime is managed automatically as defined in this section.

For each evaluation that yields a temporary value, the lifetime of that temporary begins at the evaluation point, similarly to creation of a usual named value initialized with an expression.

Termination of lifetime of temporaries does not obey the customary scoping rules and is defined as follows:

- If:
    1. the full expression has a smallest short-circuit expression *expr*; and
    2. the temporary is created on the right-hand side of the `&&` or || operator; and
    3. the right-hand side is evaluated,

  then temporary destructors are evaluated right after the right-hand side expression has been evaluated and converted to `bool`. Evaluation of destructors proceeds in reverse order of construction.
- For all other cases, the temporaries generated for the purpose of invoking functions are deferred to the end of the full expression. The order of destruction is inverse to the order of construction.

If a subexpression of an expression throws an exception, all temporaries created up to the evaluation of that subexpression will be destroyed per the rules above. No destructor calls will be issued for temporaries not yet constructed.

Note: An intuition behind these rules is that destructors of temporaries are deferred to the end of full expression and in reverse order of construction, with the exception that the right-hand side of `&&` and || are considered their own full expressions even when part of larger expressions.

Note: The ternary expression e_1 *?* e_2 *:* e_3 is not a special case although it evaluates expressions conditionally: e_1 and one of e_2 and e_3 may create temporaries. Their destructors are inserted to the end of the full expression in the reverse order of creation.

Example:

```
import std.stdio;
struct S
{
    int x;
    this(int n) { x = n; writefln("S(%s)", x); }
    ~this() { writefln("~S(%s)", x); }
}
bool b = (S(1) == S(2) || S(3) != S(4)) && S(5) == S(6);
```

The output of the code above is:

```
S(1)
S(2)
S(3)
S(4)
~S(4)
~S(3)
S(5)
S(6)
~S(6)
~S(5)
~S(2)
~S(1)
```

First, `S(1)` and `S(2)` are evaluated in lexical order. Per the rules, they will be destroyed at the end of the full expression and in reverse order. The comparison `S(1) == S(2)` yields `false`, so the right-hand side of the || is evaluated causing `S(3)` and `S(4)` to be evaluated, also in lexical order. However, their destruction is not deferred to the end of the full expression. Instead, `S(4)` and then `S(3)` are destroyed at the end of the || expression. Following their destruction, `S(5)` and `S(6)` are constructed in lexical order. Again they are not destroyed at the end of the full expression, but right at the end of the `&&` expression. Consequently, the destruction of `S(6)` and `S(5)` is carried before that of `S(2)` and `S(1)`.

10.4 Expressions

```
Expression:
    CommaExpression
```

```
CommaExpression:
    AssignExpression
```

```
AssignExpression , CommaExpression
```

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. Using the result of comma expressions isn't allowed.

10.5 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
    ConditionalExpression ^^= AssignExpression
```

For all assign expressions, the left operand must be a modifiable lvalue. The type of the assign expression is the type of the left operand, and the value is the value of the left operand after assignment occurs. The resulting expression is a modifiable lvalue.

Undefined Behavior: If either operand is a reference type and one of the following:

1. the operands have partially overlapping storage
2. the operands' storage overlaps exactly but the types are different

Implementation Defined: If neither operand is a reference type and one of the following:

1. the operands have partially overlapping storage
2. the operands' storage overlaps exactly but the types are different

Simple Assignment Expression

If the operator is `=` then it is simple assignment. The right operand is implicitly converted to the type of the left operand, and assigned to it.

If the left and right operands are of the same struct type, and the struct type has a *Postblit*, then the copy operation is as described in Struct Postblit.

If the lvalue is the `.length` property of a dynamic array, the behavior is as described in Setting Dynamic Array Length.

If the lvalue is a static array or a slice, the behavior is as described in Array Copying and Array Setting.

If the lvalue is a user-defined property, the behavior is as described in Property Functions.

Assignment Operator Expressions

For arguments of built-in types, assignment operator expressions such as

```
a op= b
```

are semantically equivalent to:

```
a = cast(typeof(a))(a op b)
```

except that

- operand `a` is only evaluated once,
- overloading *op* uses a different function than overloading *op*= does, and
- the left operand of `>>>=` does not undergo Integer Promotions before shifting.

For user-defined types, assignment operator expressions are overloaded separately from the binary operator. Still the left operand must be an lvalue.

10.6 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.

Note: When a conditional expression is the left operand of an assign expression, parentheses are required for disambiguation:

```
bool test;
int a, b, c;
...
test ? a = b : c = 2;   // Deprecated
(test ? a = b : c) = 2; // Equivalent
```

This makes the intent clearer, because the first statement can easily be misread as the following code:

```
test ? a = b : (c = 2);
```

10.7 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`.

10.8 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`.

10.9 Bitwise Expressions

Bit wise expressions perform a bitwise operation on their operands. Their operands must be integral types. First, the Usual Arithmetic Conversions 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:
    CmpExpression
    AndExpression & CmpExpression
```

The operands are AND'd together.

10.10 Compare Expressions

```
CmpExpression:
    ShiftExpression
    EqualExpression
    IdentityExpression
    RelExpression
    InExpression
```

10.11 Equality Expressions

```
EqualExpression:
    ShiftExpression == ShiftExpression
    ShiftExpression != ShiftExpression
```

Equality expressions compare the two operands for equality (`==`) or inequality (`!=`). The type of the result is `bool`.

Inequality is defined as the logical negation of equality.

If the operands are integral values, the Usual Arithmetic Conversions are applied to bring them to a common type before comparison. Equality is defined as the bit patterns of the common type match exactly.

If the operands are pointers, equality is defined as the bit patterns of the operands match exactly.

For float, double, and real values, the Usual Arithmetic Conversions are applied to bring them to a common type before comparison. The values `-0` and `+0` are considered equal. If either or both operands are NAN, then `==` returns false and `!=` returns `true`. Otherwise, the bit patterns of the common type are compared for equality.

For complex numbers, equality is defined as equivalent to:

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

For struct objects, equality means the result of the `opEquals()` member function. If an `opEquals()` is not provided, equality is defined as the logical product of all equality results of the corresponding object fields.

Implementation Defined: The contents of any alignment gaps in the struct object.

Best Practices: If there are overlapping fields, which happens with unions, the default equality will compare each of the overlapping fields. An `opEquals()` can account for which of the overlapping fields contains valid data. An `opEquals()` can override the default behavior of floating point NaN values always comparing as unequal. Be careful using `memcmp()` to implement `opEquals()` if:

- there are any alignment gaps
- if any fields have an `opEquals()`
- there are any floating point fields that may contain NaN or `-0` values

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 class objects, the `==` and `!=` operators are intended to compare the contents of the objects, however an appropriate `opEquals` override must be defined for this to work. The default `opEquals` provided by the root `Object` class is equivalent to the `is` operator. Comparing against `null` is invalid, as `null` has no contents. Use the `is` and `!is` operators instead.

```
class C;
C c;
if (c == null)  // error
    ...
if (c is null)  // ok
    ...
```

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

Identity Expressions

```
IdentityExpression:
    ShiftExpression is ShiftExpression
    ShiftExpression !is ShiftExpression
```

The `is` compares for identity. To compare for nonidentity, use `e1 !is e2`. The type of the result is `bool`. The operands undergo the Usual Arithmetic Conversions to bring them to a common type before comparison.

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

For struct objects and floating point values, identity is defined as the bits in the operands being identical.

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

For other operand types, identity is defined as being the same as equality.

The identity operator `is` cannot be overloaded.

10.12 Relational Expressions

```
RelExpression:
    ShiftExpression < ShiftExpression
    ShiftExpression <= ShiftExpression
    ShiftExpression > ShiftExpression
    ShiftExpression >= ShiftExpression
```

First, the Usual Arithmetic Conversions 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.

A relational operator can have `NaN` operands. If either or both operands is `NaN`, the floating point comparison operation returns as follows:

Floating point comparison operators

Operator	Relation	Returns
`<`	less	`false`
`>`	greater	`false`
`<=`	less or equal	`false`
`>=`	greater or equal	`false`
`==`	equal	`false`
`!=`	unordered, less, or greater	`true`

Class comparisons

For class objects, the relational operators compare the contents of the objects. Therefore, comparing against `null` is invalid, as `null` has no contents.

```
class C;
C c;
if (c < null)  // error
    ...
```

10.13 In Expressions

```
InExpression:
    ShiftExpression in ShiftExpression
    ShiftExpression !in ShiftExpression
```

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

```
int foo[string];
...
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.

The `!in` expression is the logical negation of the `in` operation.

10.14 Shift Expressions

```
ShiftExpression:
```

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

The operands must be integral types, and undergo the Integer 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 the same or more bits than the size of the quantity being shifted:

```
int c;
auto x = c << 33;        // error
```

10.15 Add Expressions

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

If the operands are of integral types, they undergo the Usual Arithmetic Conversions, 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.

If both operands are pointers, and the operator is `+`, then it is illegal.

If both operands are pointers, and the operator is `-`, the pointers are subtracted and the result is divided by the size of the type pointed to by the operands. In this calculation the assumed size of `void` is one byte. It is an error if the pointers point to different types. The type of the result is `ptrdiff_t`.

If both operands are of integral types and an overflow or underflow occurs in the computation, wrapping will happen. For example, `uint.max + 1 == uint.min`, `uint.min - 1 == uint.max`, `int.max + 1 == int.min`, and `int.min - 1 == int.max`.

Add expressions for floating point operands are not associative.

10.16 Cat Expressions

```
CatExpression:
    AddExpression ~ MulExpression
```

A *CatExpression* concatenates arrays, producing a dynamic 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.

10.17 Mul Expressions

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

The operands must be arithmetic types. They undergo 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.

For integral operands of the `/` and `%` operators, the quotient rounds towards zero and the remainder has the same sign as the dividend. If the divisor is zero, an Exception is thrown.

For floating point operands, the * and / operations correspond to the IEEE 754 floating point equivalents. the IEEE 754 remainder. For example, 15.0 for IEEE 754, remainder(15.0,10.0) == -5.0.

Mul expressions for floating point operands are not associative.

10.18 Unary Expressions

```
UnaryExpression:
    & UnaryExpression
```

```
    ++ UnaryExpression
    -- UnaryExpression
    * UnaryExpression
    - UnaryExpression
    + UnaryExpression
    ! UnaryExpression
    ComplementExpression
    ( Type ) . Identifier
    ( Type ) . TemplateInstance
    DeleteExpression
    CastExpression
    PowExpression
```

Complement Expressions

```
ComplementExpression:
    ~ UnaryExpression
```

ComplementExpressions work on integral types (except `bool`). All the bits in the value are complemented.

Note: the usual Integer Promotions are not performed prior to the complement operation.

New Expressions

```
NewExpression:
    new AllocatorArguments_opt Type
    NewExpressionWithArgs
```

```
NewExpressionWithArgs:
    new AllocatorArguments_opt Type [ AssignExpression ]
    new AllocatorArguments_opt Type ( ArgumentList_opt )
    NewAnonClassExpression
```

```
AllocatorArguments:
    ( ArgumentList_opt )
```

```
ArgumentList:
    AssignExpression
    AssignExpression ,
    AssignExpression , ArgumentList
```

NewExpressions 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);
```

The code above is equivalent to:

```
bar = new int[][][5];
foreach (ref a; bar)
{
    a = new int[][20];
    foreach (ref 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
```

NOTE: `delete` has been deprecated. Instead, please use `destroy` if feasible, or `core.memory.__delete` as a last resort.

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 *UnaryExpression* is a pointer or a dynamic array, the garbage collector is called to immediately release the memory.

The pointer, dynamic array, or reference is set to `null` after the delete is performed. Any attempt to reference the data after the deletion via another reference to it will result in undefined behavior.

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.

Undefined Behavior:

1. Using `delete` to free memory not allocated by the garbage collector.
2. Referring to data that has been the operand of `delete`.

Cast Expressions

```
CastExpression:
    cast ( Type ) UnaryExpression
    cast ( TypeCtors_opt ) 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.

```
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 pointer type to and from a class type is done as a type paint (i.e. a reinterpret cast).

Casting a dynamic array to another dynamic array is done only if the array lengths multiplied by the element sizes match. The cast is done as a type paint, with the array length adjusted to match any change in element size. If there's not a match, a runtime error is generated.

```
import std.stdio;

int main()
{
    byte[] a = [1,2,3];
    auto b = cast(int[])a; // runtime array cast misalignment

    int[] c = [1, 2, 3];
    auto d = cast(byte[])c; // ok
    // prints:
    // [1, 0, 0, 0, 2, 0, 0, 0, 3, 0, 0, 0]
    writeln(d);
    return 0;
}
```

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 floating point value to an integral type is the equivalent of converting to an integer using truncation.

```
void main()
{
    int a = cast(int) 0.8f;
    assert(a == 0);
    long b = cast(long) 1.5;
    assert(b == 1L);
    long c = cast(long) -1.5;
    assert(c == -1);
}
```

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

```
S(v)
```

Casting to a *CastQual* replaces the qualifiers to the type of the *UnaryExpression*.

```
shared int x;
assert(is(typeof(cast(const)x) == const int));
```

Casting with no *Type* or *CastQual* removes any top level `const`, `immutable`, `shared` or `inout` type modifiers from the type of the *UnaryExpression*.

```
shared int x;
assert(is(typeof(cast()x) == int));
```

Casting an expression to `void` type is allowed to mark that the result is unused. On *ExpressionStatement*, it could be used properly to avoid "has no effect" error.

```
void foo(lazy void exp) {}
void main()
{
    foo(10);            // NG - has no effect in expression '10'
    foo(cast(void)10);  // OK
}
```

10.19 Pow Expressions

```
PowExpression:
    PostfixExpression
    PostfixExpression ^^ UnaryExpression
```

PowExpression raises its left operand to the power of its right operand.

10.20 Postfix Expressions

```
PostfixExpression:
    PrimaryExpression
    PostfixExpression . Identifier
    PostfixExpression . TemplateInstance
    PostfixExpression . NewExpression
    PostfixExpression ++
    PostfixExpression --
    PostfixExpression ( ArgumentList opt )
    TypeCtors opt BasicType ( ArgumentList opt )
    IndexExpression
    SliceExpression
```

10.21 Index Expressions

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

PostfixExpression is evaluated. If *PostfixExpression* is an expression of type static array or dynamic array, the symbol $ is set to be the number of elements in the array. If *PostfixExpression* is a *ValueSeq*, the symbol $ is set to be the number of elements in the sequence. A new declaration scope is created for the evaluation of the *ArgumentList* and $ appears in that scope only.

If *PostfixExpression* is a *ValueSeq*, 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 nth expression in the *ValueSeq*, which is the result of the *IndexExpression*. It is an error if n is out of bounds of the *ValueSeq*.

10.22 Slice Expressions

```
SliceExpression:
    PostfixExpression [ ]
    PostfixExpression [ Slice ,opt ]
Slice:
    AssignExpression
    AssignExpression , Slice
    AssignExpression .. AssignExpression
    AssignExpression .. AssignExpression , Slice
```

PostfixExpression is evaluated. if *PostfixExpression* is an expression of type static array or dynamic array, 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 $ 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*.

A *SliceExpression* is not a modifiable lvalue.

If the slice bounds can be known at compile time, the slice expression is implicitly convertible to an lvalue of static array. For example:

```
arr[a .. b]     // typed T[]
```

If both `a` and `b` are integers (may be constant-folded), the slice expression can be converted to a static array type `T[b - a]`.

```
void foo(int[2] a)
{
```

```
    assert(a == [2, 3]);
}
void bar(ref int[2] a)
{
    assert(a == [2, 3]);
    a[0] = 4;
    a[1] = 5;
    assert(a == [4, 5]);
}
void baz(int[3] a) {}

void main()
{
    int[] arr = [1, 2, 3];

    foo(arr[1 .. 3]);
    assert(arr == [1, 2, 3]);

    bar(arr[1 .. 3]);
    assert(arr == [1, 4, 5]);

  //baz(arr[1 .. 3]); // cannot match length
}
```

The following forms of slice expression can be convertible to a static array type:

`e` An expression that contains no side effects.

`a`, `b` Integers (that may be constant-folded).

Computing array lengths during compilation

Form	The length calculated at compile time
`arr[]`	The compile time length of `arr` if it's known.
`arr[a .. b]`	`b - a`
`arr[e-a .. e]`	`a`
`arr[e .. e+b]`	`b`
`arr[e-a .. e+b]`	`a + b`
`arr[e+a .. e+b]`	`b - a` *if* `a <= b`
`arr[e-a .. e-b]`	`a - b` *if* `a >= b`

If *PostfixExpression* is a *ValueSeq*, then the result of the slice is a new *ValueSeq* 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.

10.23 Primary Expressions

```
PrimaryExpression:
    Identifier
    . Identifier
    TemplateInstance
    . TemplateInstance
    this
    super
    null
    true
    false
    $
    IntegerLiteral
    FloatLiteral
    CharacterLiteral
    StringLiterals
    ArrayLiteral
    AssocArrayLiteral
    FunctionLiteral
    AssertExpression
    MixinExpression
    ImportExpression
    NewExpressionWithArgs
    BasicTypeX . Identifier
    BasicTypeX ( ArgumentList opt )
    TypeCtor ( Type ) . Identifier
    TypeCtor ( Type ) ( ArgumentList opt )
    Typeof
    TypeidExpression
    IsExpression
    ( Expression )
    SpecialKeyword
    TraitsExpression
```

.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 for which the function was called. If the object is an instance of a struct, `this` will be a pointer to that instance. 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
{
    override char get() { return 'B'; }
}

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

    assert(b.foo() == 'A');
    assert(b.bar() == 'B');
}
```

Assignment to `this` is not allowed.

super

`super` is identical to `this`, except that it is cast to `this`'s base class. It is an error if there is no base class. It is an error to use `super` within a struct member function. (Only class `Object` has no base class.) If a member function is called with an explicit reference to `super`, a non-virtual call is made.

Assignment to `super` is not allowed.

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 singular type `typeof(null)` 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.

String Literals

```
StringLiterals:
    StringLiteral
    StringLiterals StringLiteral
```

String literals can implicitly convert to any of the following types, they have equal weight:

```
immutable(char)*
immutable(wchar)*
immutable(dchar)*
immutable(char)[]
immutable(wchar)[]
immutable(dchar)[]
```

By default, a string literal is typed as a dynamic array, but the element count is known at compile time. So all string literals can be implicitly converted to static array types.

```
void foo(char[2] a)
{
    assert(a == "bc");
}
void bar(ref const char[2] a)
{
```

```
    assert(a == "bc");
}
void baz(const char[3] a) {}

void main()
{
    string str = "abc";
    foo(str[1 .. 3]);
    bar(str[1 .. 3]);
  //baz(str[1 .. 3]); // cannot match length
}
```

String literals have a 0 appended to them, which makes them easy to pass to C or C++ functions expecting a `const char*` string. The 0 is not included in the `.length` property of the string literal.

Array Literals

```
ArrayLiteral:
    [ ArgumentList opt ]
```

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

```
auto a1 = [1,2,3];  // type is int[], with elements 1, 2 and 3
auto a2 = [1u,2,3]; // type is uint[], with elements 1u, 2u, and 3u
```

By default, an array literal is typed as a dynamic array, but the element count is known at compile time. So all array literals can be implicitly converted to static array types.

```
void foo(long[2] a)
{
    assert(a == [2, 3]);
}
void bar(ref long[2] a)
{
    assert(a == [2, 3]);
    a[0] = 4;
    a[1] = 5;
    assert(a == [4, 5]);
}
```

```
void baz(const char[3] a) {}

void main()
{
    long[] arr = [1, 2, 3];

    foo(arr[1 .. 3]);
    assert(arr == [1, 2, 3]);

    bar(arr[1 .. 3]);
    assert(arr == [1, 4, 5]);

  //baz(arr[1 .. 3]); // cannot match length
}
```

If any of the arguments in the *ArgumentList* are a *ValueSeq*, then the elements of the *ValueSeq* are inserted as arguments in place of the sequence.

Array literals are allocated on the memory managed heap. Thus, they can be returned safely from functions:

```
int[] foo()
{
    return [1, 2, 3];
}
```

When array literals are cast to another array type, each element of the array is cast to the new element type. When arrays that are not literals are cast, the array is reinterpreted as the new type, and the length is recomputed:

```
import std.stdio;

void main()
{
    // cast array literal
    const short[] ct = cast(short[]) [cast(byte)1, 1];
    // this is equivalent with:
    // const short[] ct = [cast(short)1, cast(short)1];
    writeln(ct);  // writes [1, 1]

    // cast other array expression
    // --> normal behavior of CastExpression
    byte[] arr = [cast(byte)1, cast(byte)1];
```

```
    short[] rt = cast(short[]) arr;
    writeln(rt);  // writes [257]
}
```

In other words, casting literal expression will change the literal type.

Associative Array Literals

```
AssocArrayLiteral:
    [ KeyValuePairs ]


KeyValuePairs:
    KeyValuePair
    KeyValuePair , KeyValuePairs


KeyValuePair:
    KeyExpression : ValueExpression


KeyExpression:
    AssignExpression


ValueExpression:
    AssignExpression
```

Associative array literals are a comma-separated list of *key*:*value* pairs between square brackets `[` and `]`. The list cannot be empty. The common type of the all keys is taken to be the key type of the associative array, and all keys are implicitly converted to that type. The common type of the all values is taken to be the value type of the associative array, and all values are implicitly converted to that type. An *AssocArrayLiteral* cannot be used to statically initialize anything.

```
[21u:"he", 38:"ho", 2:"hi"];  // type is string[uint],
                              // with keys 21u, 38u and 2u
                              // and values "he", "ho", and "hi"
```

If any of the keys or values in the *KeyValuePairs* are a *ValueSeq*, then the elements of the *ValueSeq* are inserted as arguments in place of the sequence.

Function Literals

FunctionLiteral:
  `function` `ref`$_{opt}$ *Type*$_{opt}$ *ParameterWithAttributes* $_{opt}$ *FunctionLiteralBody*
  `delegate` `ref`$_{opt}$ *Type*$_{opt}$ *ParameterWithMemberAttributes* $_{opt}$ *FunctionLiteralBody*
  `ref`$_{opt}$ *ParameterWithMemberAttributes* *FunctionLiteralBody*
  FunctionLiteralBody
  Lambda

ParameterWithAttributes:
  Parameters *FunctionAttributes*$_{opt}$

ParameterWithMemberAttributes:
  Parameters *MemberFunctionAttributes*$_{opt}$

FunctionLiteralBody:
  BlockStatement
  FunctionContracts$_{opt}$ *BodyStatement*

*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 *FunctionLiteralBody*. `(` *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 is inferred from whether *FunctionLiteralBody* is actually accessing to the outer context.

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;} ;
}
```

Also:

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

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

    abc(&foo);
}
```

is exactly equivalent to:

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

void test()
{
    int b = 3;

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

and the following where the return type `int` and `function`/`delegate` are inferred:

```
int abc(int delegate(int i));
int def(int function(int s));

void test()
{
    int b = 3;

    abc( (int c) { return 6 + b; } );  // inferred to delegate
    def( (int c) { return c * 2; } );  // inferred to function
  //def( (int c) { return c * b; } );  // error!
    // Because the FunctionLiteralBody accesses b, then the function literal type
```

```
    // is inferred to delegate. But def cannot receive delegate.
}
```

If the type of a function literal can be uniquely determined from its context, the parameter type inference is possible.

```
void foo(int function(int) fp);

void test()
{
    int function(int) fp = (n) { return n * 2; };
    // The type of parameter n is inferred to int.

    foo((n) { return n * 2; });
    // The type of parameter n is inferred to int.
}
```

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.

Lambdas

```
Lambda:
    function ref_opt Type_opt ParameterWithAttributes => AssignExpression
    delegate ref_opt Type_opt ParameterWithMemberAttributes => AssignExpression
    ref_opt ParameterWithMemberAttributes => AssignExpression
    Identifier => AssignExpression
```

*Lambda*s are a shorthand syntax for *FunctionLiteral*s.

1. Just one *Identifier* is rewritten to *Parameters*:
   `(` *Identifier* `)`
2. The following part `=>` *AssignExpression* is rewritten to *FunctionLiteralBody*:
   `{ return` *AssignExpression* `; }`

Example usage:

```
import std.stdio;

void main()
{
    auto i = 3;
    auto twice  = function (int x) => x * 2;
    auto square = delegate (int x) => x * x;

    auto n = 5;
    auto mul_n = (int x) => x * n;

    writeln(twice(i));   // prints 6
    writeln(square(i));  // prints 9
    writeln(mul_n(i));   // prints 15
}
```

Uniform construction syntax for built-in scalar types

The implicit conversions of built-in scalar types can be explicitly represented by using function call syntax. For example:

```
auto a = short(1);  // implicitly convert an integer literal '1' to short
auto b = double(a); // implicitly convert a short variable 'a' to double
auto c = byte(128); // error, 128 cannot be represented in a byte
```

If the argument is omitted, it means default construction of the scalar type:

```
auto a = ushort();  // same as: ushort.init
auto b = wchar();   // same as: wchar.init
auto c = creal();   // same as: creal.init
```

Assert Expressions

```
AssertExpression:
    assert ( AssertArguments )


AssertArguments:
    AssignExpression ,opt
    AssignExpression , AssignExpression ,opt
```

The first *AssignExpression* must evaluate to true. If it does not, an *Assert Failure* has occurred and the program enters an *Invalid State*.

If the first *AssignExpression* consists entirely of compile time constants, and evaluates to false, it is a special case; it signifies that it is unreachable code. Compile Time Function Execution (CTFE) is not attempted.

AssertExpression has different semantics if it is in a `unittest` or `in` contract.

The second *AssignExpression*, if present, must be implicitly convertible to type `const(char)[]`.

If the first *AssignExpression* is a reference to a class instance for which a Class Invariant exists, the *Class Invariant* must hold.

If the first *AssignExpression* is a pointer to a struct instance for which a *Struct Invariant* exists, the *Struct Invariant* must hold.

The type of an *AssertExpression* is `void`.

Undefined Behavior: Once in an *Invalid State* the behavior of the continuing execution of the program is undefined.

Implementation Defined: Whether the first *AssertExpression* is evaluated or not at runtime is typically set with a compiler switch. If it is not evaluated, any side effects specified by the *AssertExpression* may not occur. The behavior if the first *AssertExpression* is evaluated and is false is also typically set with a compiler switch and may include these options:

1. continuing execution
2. immediately halting via execution of a special CPU instruction
3. aborting the program
4. calling the assert failure function in the corresponding C runtime library

5. throwing the `AssertError` exception in the D runtime library

If the optional second *AssignExpression* is provided, the implementation may evaluate it and print the resulting message upon assert failure:

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

When compiled and run, it will produce the message:

```
core.exception.AssertError@test.d(3) an error message
```

The implementation may handle the case of the first *AssignExpression* evaluating at compile time to false differently in that in release mode it may simply generate a `HLT` instruction or equivalent.

Best Practices:

1. Do not have side effects in either *AssignExpression* that subsequent code depends on.
2. *AssertExpressions* are intended to detect bugs in the program, do not use for detecting input or environmental errors.
3. Do not attempt to resume normal execution after an *Assert Failure*.

Mixin Expressions

```
MixinExpression:
    mixin ( ArgumentList )
```

Each *AssignExpression* in the *ArgumentList* is evaluated at compile time, and the result must be representable as a string. The resulting strings are concatenated to form a string. The text contents of the string must be compilable as a valid *Expression*, and is compiled as such.

```
int foo(int x)
{
    return mixin("x␣+", 1) * 7;  // same as ((x + 1) * 7)
}
```

Import Expressions

```
ImportExpression:
    import ( AssignExpression )
```

The *AssignExpression* must evaluate at compile time to a constant string. The text contents of the string are interpreted as a file name. The file is read, and the exact contents of the file become a string literal.

Implementations may restrict the file name in order to avoid directory traversal security vulnerabilities. A possible restriction might be to disallow any path components in the file name.

Note that by default an import expression will not compile unless you pass one or more paths via the **-J** switch. This tells the compiler where it should look for the files to import. This is a security feature.

```
void foo()
{
    // Prints contents of file foo.txt
    writeln(import("foo.txt"));
}
```

Typeid Expressions

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

If *Type*, returns an instance of class `TypeInfo` corresponding to *Type*.

If *Expression*, returns an instance of class `TypeInfo` corresponding to the type of the *Expression*. If the type is a class, it returns the `TypeInfo` of the dynamic type (i.e. the most derived type). The *Expression* is always executed.

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

void main()
{
    writeln(typeid(int));        // int
    uint i;
    writeln(typeid(i++));        // uint
    writeln(i);                  // 1
    A a = new B();
    writeln(typeid(a));          // B
    writeln(typeid(typeof(a)));  // A
}
```

IsExpression

IsExpression:
  `is` (*Type*)
  `is` (*Type* : *TypeSpecialization*)
  `is` (*Type* == *TypeSpecialization*)
  `is` (*Type* : *TypeSpecialization* , *TemplateParameterList*)
  `is` (*Type* == *TypeSpecialization* , *TemplateParameterList*)
  `is` (*Type* *Identifier*)
  `is` (*Type* *Identifier* : *TypeSpecialization*)
  `is` (*Type* *Identifier* == *TypeSpecialization*)
  `is` (*Type* *Identifier* : *TypeSpecialization* , *TemplateParameterList*)
  `is` (*Type* *Identifier* == *TypeSpecialization* , *TemplateParameterList*)

TypeSpecialization:
  Type
  `struct`
  `union`
  `class`
  `interface`
  `enum`
  `__vector`
  `function`
  `delegate`
  `super`
  `const`
  `immutable`
  `inout`
  `shared`
  `return`
  `__parameters`
  `module`
  `package`

*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 a boolean of value `true` if the condition is satisfied. If the condition is not satisfied, the result is a boolean of value `false`.

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
           writeln("satisfied");
       else
           writeln("not␣satisfied");

       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 Bar = short;
   void foo()
   {
       if (is(Bar : int))   // satisfied because short can be
                            // implicitly converted to int
           writeln("satisfied");
       else
           writeln("not␣satisfied");
   }
   ```

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
`struct union class interface enum function delegate const immutable shared module package`
then the condition is satisfied if *Type* is one of those. Package modules are considered to be both packages and modules.

```
alias Bar = short;
void foo()
{
    if (is(Bar == int))   // not satisfied because short is not
                          // the same type as int
        writeln("satisfied");
    else
        writeln("not␣satisfied");
}
```

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 Bar = short;
void foo()
{
    static if (is(Bar T))
        alias S = T;
    else
        alias S = long;

    writeln(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 *TypeSpecialization*, or if *Type* is a class and *TypeSpecialization* is a base class or base interface of it. The *Identifier* is declared to be either an alias of the *TypeSpecialization* or, if *TypeSpecialization* is dependent on *Identifier*, the deduced type.

```
alias Bar = int;
alias Abc = long*;
void foo()
{
```

```
    static if (is(Bar T : int))
        alias S = T;
    else
        alias S = long;

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

    static if (is(Abc U : U*))
    {
        U u;
        writeln(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 `struct union class interface enum function delegate const immutable shared`
   then the condition is satisfied if *Type* is one of those. Furthermore, *Identifier* is set to be an alias of the type:

keyword	alias type for *Identifier*
`struct`	*Type*
`union`	*Type*
`class`	*Type*
`interface`	*Type*
`super`	*TypeSeq* of base classes and interfaces
`enum`	the base type of the enum
`function`	*TypeSeq* of the function parameter types. For C- and D-style variadic functions, only the non-variadic parameters are included. For typesafe variadic functions, the ... is ignored.
`delegate`	the function type of the delegate
`return`	the return type of the function, delegate, or function pointer
`__parameters`	the parameter sequence of a function, delegate, or function pointer. This includes the parameter types, names, and default values.
`const`	*Type*
`immutable`	*Type*
`shared`	*Type*

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

7. `is` (*Type* : *TypeSpecialization* , *TemplateParameterList*) `is` (*Type* == *TypeSpecialization* , *TemplateParameterList*) `is` (*Type Identifier* : *TypeSpecialization* , *TemplateParameterList*) `is` (*Type Identifier* == *TypeSpecialization* , *TemplateParameterList*)
   More complex types can be pattern matched; the *TemplateParameterList* declares symbols based on the parts of the pattern that are matched, analogously to the way implied template parameters are matched.

```
import std.stdio, std.typecons;
```

```
void main()
{
    alias Tup = Tuple!(int, string);
    alias AA = long[string];

    static if (is(Tup : Template!Args, alias Template, Args...))
    {
        writeln(__traits(isSame, Template, Tuple)); // true
        writeln(is(Template!(int, string) == Tup));  // true
        writeln(typeid(Args[0]));  // int
        writeln(typeid(Args[1]));  // immutable(char)[]
    }

    static if (is(AA T : T[U], U : string))
    {
        writeln(typeid(T));  // long
        writeln(typeid(U));  // string
    }

    static if (is(AA A : A[B], B : int))
    {
        assert(0);  // should not match, as B is not an int
    }

    static if (is(int[10] W : W[len], int len))
    {
        writeln(typeid(W));  // int
        writeln(len);        // 10
    }

    static if (is(int[10] X : X[len], int len : 5))
    {
        assert(0);  // should not match, len should be 10
    }
}
```

10.24 Special Keywords

SpecialKeyword:

```
__FILE__
__FILE_FULL_PATH__
__MODULE__
__LINE__
__FUNCTION__
__PRETTY_FUNCTION__
```

`__FILE__` and `__LINE__` expand to the source file name and line number at the point of instantiation. The path of the source file is left up to the compiler.

`__FILE_FULL_PATH__` expands to the absolute source file name at the point of instantiation.

`__MODULE__` expands to the module name at the point of instantiation.

`__FUNCTION__` expands to the fully qualified name of the function at the point of instantiation.

`__PRETTY_FUNCTION__` is similar to `__FUNCTION__`, but also expands the function return type, its parameter types, and its attributes.

Example:

```
module test;
import std.stdio;

void test(string file = __FILE__, size_t line = __LINE__,
        string mod = __MODULE__, string func = __FUNCTION__,
        string pretty = __PRETTY_FUNCTION__,
        string fileFullPath = __FILE_FULL_PATH__)
{
    writefln("file: '%s', line: '%s', module: '%s',\nfunction: '%s', " ~
        "pretty function: '%s',\nfile full path: '%s'",
        file, line, mod, func, pretty, fileFullPath);
}

int main(string[] args)
{
    test();
    return 0;
}
```

Assuming the file was at /example/test.d, this will output:

```
file: 'test.d', line: '13', module: 'test',
function: 'test.main', pretty function: 'int test.main(string[] args)',
file full path: '/example/test.d'
```

10.25 Associativity and Commutativity

An implementation may rearrange the evaluation of expressions according to arithmetic associativity and commutativity rules as long as, within that thread of execution, no observable difference is possible.

This rule precludes any associative or commutative reordering of floating point expressions.

Chapter 11

Statements

The order of execution within a function is controlled by *Statements*. A function's body consists of a sequence of zero or more *Statements*. Execution occurs in lexical order, though certain statements may have deferred effects. A *Statement* has no value; it is executed for its effects.

```
Statement:
    ;
    NonEmptyStatement
    ScopeBlockStatement


NoScopeNonEmptyStatement:
    NonEmptyStatement
    BlockStatement


NoScopeStatement:
    ;
    NonEmptyStatement
    BlockStatement


NonEmptyOrScopeBlockStatement:
    NonEmptyStatement
    ScopeBlockStatement


NonEmptyStatement:
```

```
    NonEmptyStatementNoCaseNoDefault
    CaseStatement
    CaseRangeStatement
    DefaultStatement
```

```
NonEmptyStatementNoCaseNoDefault:
    LabeledStatement
    ExpressionStatement
    DeclarationStatement
    IfStatement
    WhileStatement
    DoStatement
    ForStatement
    ForeachStatement
    SwitchStatement
    FinalSwitchStatement
    ContinueStatement
    BreakStatement
    ReturnStatement
    GotoStatement
    WithStatement
    SynchronizedStatement
    TryStatement
    ScopeGuardStatement
    ThrowStatement
    AsmStatement
    PragmaStatement
    MixinStatement
    ForeachRangeStatement
    ConditionalStatement
    StaticForeachStatement
    StaticAssert
    TemplateMixin
    ImportDeclaration
```

Any ambiguities in the grammar between *Statement*s and *Declaration*s are resolved by the declarations taking precedence. Wrapping such a statement in parentheses will disambiguate it in favor of being a *Statement*.

11.1 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
}
```

Best Practices: Local declarations within a function should all have unique names, even if they are in non-overlapping scopes.

11.2 Scope Block Statements

```
ScopeBlockStatement:
    BlockStatement
```

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

11.3 Labeled Statements

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

```
LabeledStatement:
    Identifier :
    Identifier : NoScopeStatement
    Identifier : Statement
```

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

A label can appear without a following statement at the end of a block.

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.

Labels in one function cannot be referenced from another function.

11.4 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, until the end of the block is reached or a statement transfers control elsewhere.

11.5 Expression Statement

ExpressionStatement:
    Expression ;

The expression is evaluated.

Expressions that have no effect, like `(x + x)`, are illegal as expression statements unless the are cast to void.

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

11.6 Declaration Statement

Declaration statements define variables, and declare types, templates, functions, imports, conditionals, static foreaches, and static asserts.

DeclarationStatement:
    StorageClasses$_{\text{opt}}$ *Declaration*

Some declaration statements:

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

11.7 If Statement

If statements provide simple conditional execution of statements.

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

IfCondition:
    Expression
    `auto` *Identifier* = *Expression*
    TypeCtors *Identifier* = *Expression*
    TypeCtors$_{\text{opt}}$ *BasicType* *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 *ElseStatement*is associated with the innermost if statement which does not already have an associated *ElseStatement*.

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 *TypeCtors Identifier* is provided, it is declared to be of the type specified by *TypeCtors* and is initialized with the value 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.regex;
...
if (auto m = std.regex.matchFirst("abcdef", "b(c)d"))
{
    writefln("[%s]", m.pre);    // prints [a]
    writefln("[%s]", m.post);   // prints [ef]
    writefln("[%s]", m[0]);     // prints [bcd]
    writefln("[%s]", m[1]);     // prints [c]
}
else
{
    writeln(m.post); // Error: undefined identifier 'm'
}
writeln(m.pre);      // Error: undefined identifier 'm'
```

11.8 While Statement

```
WhileStatement:
    while ( Expression ) ScopeStatement
```

A *While Statement* implements a simple loop.

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 *ContinueStatement* will transfer directly to evaluating *Expression* again.

A *While Statement* is equivalent to:

```
for (; Expression; ) ScopeStatement
```

11.9 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.

11.10 For Statement

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

```
ForStatement:
    for ( Initialize Test_opt ; Increment_opt ) ScopeStatement
```

```
Initialize:
    ;
    NoScopeNonEmptyStatement
```

```
Test:
    Expression
```

```
Increment:
    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 (although the trailing `;` is still required). *Test* may also be omitted, and if so, it is treated as if it evaluated to true.

Best Practices: Consider replacing *ForStatements* with Foreach Statements or Foreach Range Statements. Foreach loops are easier to understand, less prone to error, and easier to refactor.

11.11 Foreach Statement

A `foreach` statement loops over the contents of an aggregate.

```
AggregateForeach:
    Foreach ( ForeachTypeList ; ForeachAggregate )
```

```
ForeachStatement:
    AggregateForeach NoScopeNonEmptyStatement
```

```
Foreach:
    foreach
    foreach_reverse
```

```
ForeachTypeList:
    ForeachType
    ForeachType , ForeachTypeList
```

```
ForeachType:
    ForeachTypeAttributes opt BasicType Declarator
    ForeachTypeAttributes opt Identifier
    ForeachTypeAttributes opt alias Identifier
```

ForeachTypeAttributes
  ForeachTypeAttribute
  ForeachTypeAttribute *ForeachTypeAttributes*$_{\text{opt}}$

ForeachTypeAttribute:
  `ref`
  TypeCtor
  `enum`

ForeachAggregate:
  Expression

ForeachAggregate is evaluated. It must evaluate to an expression of type static array, dynamic array, associative array, struct, class, delegate, or sequence. 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 `ref`, 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*.

Foreach over Arrays

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 type `size_t` for dynamic arrays. Static arrays may use any integral type that spans the length of the array. *index* cannot be `ref`. It is set to be the index of the array element.

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

Note: The *ForeachTypeAttribute* is implicit, and when a type is not specified, it is inferred. In that case, auto is implied, and it is not necessary (and actually forbidden) to use it.

```
int[] arr;
...
foreach (n; arr) // ok, n is an int
    writeln(n);

foreach (auto n; arr) // error, auto is redundant
    writeln(n);
```

Foreach over Arrays of Characters

If the aggregate expression is a static or dynamic array of chars, wchars, or dchars, 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".dup;  // \u2260 encoded as 3 UTF-8 bytes

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

dchar[] b = "\u2260"d.dup;

foreach (char c; b)
{
    writef("%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")
    {
        writefln("'%s'", c);
    }
```

```
    foreach (wchar w; "xy")
    {
        writefln("'%s'", w);
    }
}
```

which would print:

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

Foreach over Associative Arrays

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 `ref`, and it is set to be the index of the array element. The order in which the elements of the array are iterated over is unspecified for `foreach`. `foreach_reverse` for associative arrays is illegal.

```
double[string] a; // index type is string, value type is double
...
foreach (string s, double d; a)
{
    writefln("a['%s']␣=␣%g", s, d);
}
```

Foreach over Structs and Classes with opApply

If the aggregate expression is a struct or class object, the `foreach` is defined by the special `opApply` member function, and the `foreach_reverse` behavior is defined by the special `opApplyReverse` member function. These functions have the type:

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

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

where *Type* matches the *Type* used in the *ForeachType* declaration of *Identifier*. Multiple *ForeachTypes* correspond with multiple *Types* 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 *ForeachTypes* 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[2] array;

    int opApply(scope int delegate(ref 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)
    {
        writefln("%d", u);
    }
}
```

which would print:

```
73
82
```

The `scope` storage class on the *dg* parameter means that the parameter's value does not escape the scope of the *opApply* function (an example would be assigning *dg* to a global). If it cannot be statically guaranteed that *dg* does not escape, a closure may be allocated for it on the heap instead of the stack. Best practice is to annotate delegate parameters with `scope` when possible.

opApply can also be a templated function, which will infer the types of parameters based on the *ForeachStatement*.

For example:

```
struct S
{
    import std.traits : ParameterTypeTuple;  // introspection template

    int opApply(Dg)(scope Dg dg)
    if (ParameterTypeTuple!Dg.length == 2) // foreach with 2 parameters
    {
        return 0;
    }

    int opApply(Dg)(scope Dg dg)
    if (ParameterTypeTuple!Dg.length == 3) // foreach with takes 3 parameters
    {
        return 0;
    }
}

void main()
{
    foreach (int a, int b; S()) { }  // calls first opApply function
    foreach (int a, int b, float c; S()) { }  // calls second opApply function
}
```

It is important to make sure that, if `opApply` catches any exceptions, that those exceptions did not originate from the delegate passed to *opApply*. The user would expect exceptions thrown from a `foreach` body to both terminate the loop, and propagate outside the `foreach` body.

Foreach over Structs and Classes with Ranges

If the aggregate expression is a struct or class object, but the `opApply` for `foreach`, or `opApplyReverse` `foreach_reverse` do not exist, then iteration over struct and class objects can be done with range primitives. For foreach, this means the following properties and methods must be defined:

Foreach Range Properties

Property	Purpose
`.empty`	returns true if no more elements
`.front`	return the leftmost element of the range

Foreach Range Methods

Method	Purpose
`.popFront()`	move the left edge of the range right by one

Meaning:

```
foreach (e; range) { ... }
```

translates to:

```
for (auto __r = range; !__r.empty; __r.popFront())
{
    auto e = __r.front;
    ...
}
```

Similarly, for foreach_reverse, the following properties and methods must be defined:

Foreach_reverse Range Properties

Property	Purpose
`.empty`	returns true if no more elements
`.back`	return the rightmost element of the range

Foreach_reverse Range Methods

Method	Purpose
`.popBack()`	move the right edge of the range left by one

Meaning:

```
foreach_reverse (e; range) { ... }
```

translates to:

```
for (auto __r = range; !__r.empty; __r.popBack())
{
    auto e = __r.back;
    ...
}
```

Foreach over Delegates

If *ForeachAggregate* 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.

For example:

```
void main()
{
    // Custom loop implementation, that iterates over powers of 2 with
    // alternating sign. The loop body is passed in dg.
    int myLoop(int delegate(ref int) dg)
    {
        for (int z = 1; z < 128; z *= -2)
        {
            auto ret = dg(z);

            // If the loop body contains a break, ret will be non-zero.
            if (ret != 0)
                return ret;
        }
        return 0;
    }

    // This example loop simply collects the loop index values into an array.
    int[] result;
    foreach (ref x; &myLoop)
    {
        result ~= x;
    }
    assert(result == [1, -2, 4, -8, 16, -32, 64, -128]);
}
```

Note: When *ForeachAggregate* is a delegate, the compiler does not try to implement reverse traversal of the results returned by the delegate when `foreach_reverse` is used. This may result in code that is confusing to read. Therefore, using `foreach_reverse` with a delegate is now deprecated, and will be rejected in the future.

Foreach over Sequences

If the aggregate expression is a sequence, there can be one or two iteration symbols declared. If one, then the symbol is an *element alias* of each element in the sequence in turn.

If the sequence is a *TypeSeq*, then the foreach statement is executed once for each type, and the element alias is set to each type.

When the sequence is a *ValueSeq*, the element alias is a variable and is set to each value in the sequence. If the type of the variable is given, it must match the type of the sequence contents. If no type is given, the type of the variable is set to the type of the sequence element, which may change from iteration to iteration.

If there are two symbols declared, the first is the *index variable* and the second is the *element alias*. The index must be of `int`, `uint`, `long` or `ulong` type, it cannot be `ref`, and it is set to the index of each sequence element.

Example:

```
import std.meta : AliasSeq;

void main()
{
    alias Seq = AliasSeq!(int, "literal", main);

    foreach (sym; Seq)
    {
        pragma(msg, sym.stringof);
    }
}
```

Output:

```
int
"literal"
main()
```

See also: Static Foreach.

Foreach Ref Parameters

`ref` can be used to update the original elements:

```
void test()
```

```
{
    static uint[2] a = [7, 8];

    foreach (ref uint u; a)
    {
        u++;
    }
    foreach (uint u; a)
    {
        writefln("%d", u);
    }
}
```

which would print:

```
8
9
```

`ref` can not be applied to the index values.

If not specified, the *Types* in the *ForeachType* can be inferred from the type of the *ForeachAggregate*.

Foreach Restrictions

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

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

11.12 Foreach Range Statement

A foreach range statement loops over the specified range.

RangeForeach:

```
Foreach ( ForeachType ; LwrExpression .. UprExpression )
```

```
LwrExpression:
    Expression
```

```
UprExpression:
    Expression
```

```
ForeachRangeStatement:
    RangeForeach ScopeStatement
```

ForeachType declares a variable with either an explicit type, or a type inferred from *LwrExpression* and *UprExpression*. The *ScopeStatement* is then executed n times, where n is the result of *UprExpression* - *LwrExpression*. If *UprExpression* is less than or equal to *LwrExpression*, the *ScopeStatement* is executed zero times. If *Foreach* is `foreach`, then the variable is set to *LwrExpression*, then incremented at the end of each iteration. If *Foreach* is `foreach_reverse`, then the variable is set to *UprExpression*, then decremented before each iteration. *LwrExpression* and *UprExpression* are each evaluated exactly once, regardless of how many times the *ScopeStatement* is executed.

```
import std.stdio;

int foo()
{
    write("foo");
    return 10;
}

void main()
{
    foreach (i; 0 .. foo())
    {
        write(i);
    }
}
```

prints:

```
foo0123456789
```

Break and Continue out of Foreach

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

11.13 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 ArgumentList : ScopeStatementList
```

```
CaseRangeStatement:
    case FirstExp : .. case LastExp : ScopeStatementList
```

```
FirstExp:
    AssignExpression
```

```
LastExp:
    AssignExpression
```

```
DefaultStatement:
    default : ScopeStatementList
```

```
ScopeStatementList:
    StatementListNoCaseNoDefault
```

StatementListNoCaseNoDefault:
    StatementNoCaseNoDefault
    StatementNoCaseNoDefault StatementListNoCaseNoDefault

StatementNoCaseNoDefault:
    ;
    NonEmptyStatementNoCaseNoDefault
    ScopeBlockStatement

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, *ArgumentList*, are a comma separated list of expressions.

A *CaseRangeStatement* is a shorthand for listing a series of case statements from *FirstExp* to *LastExp*.

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

A switch statement must have a default statement.

The case expressions must all evaluate to a constant value or array, or a runtime initialized const or immutable variable of integral type. They must be implicitly convertible to the type of the switch *Expression*.

Case expressions must all evaluate to distinct values. Const or immutable variables must all have different names. If they share a value, the first case statement with that value gets control. There must be exactly one default statement.

The *ScopeStatementList* introduces a new scope.

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;
}
```

A *ScopeStatementList* must either be empty, or be ended with a *ContinueStatement*, *BreakStatement*, *ReturnStatement*, *GotoStatement*, *ThrowStatement* or assert(0) expression unless this is the last case. This is to set apart with C's error-prone implicit fall-through behavior. `goto case;` could be used for explicit fall-through:

```
int number;
string message;
switch (number)
{
    default:    // valid: ends with 'throw'
        throw new Exception("unknown number");

    case 3:     // valid: ends with 'break' (break out of the 'switch' only)
        message ~= "three ";
        break;

    case 4:     // valid: ends with 'continue' (continue the enclosing loop)
        message ~= "four ";
        continue;

    case 5:     // valid: ends with 'goto' (explicit fall-through to next case.)
        message ~= "five ";
        goto case;

    case 6:     // ERROR: implicit fall-through
        message ~= "six ";

    case 1:     // valid: the body is empty
    case 2:     // valid: this is the last case in the switch statement.
        message = "one or two";
}
```

A break statement will exit the switch *BlockStatement*.

Strings can be used in switch expressions. For example:

```
string 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. char, wchar and dchar 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.

11.14 Final Switch Statement

```
FinalSwitchStatement:
    final switch ( Expression ) ScopeStatement
```

A final switch statement is just like a switch statement, except that:

- No *DefaultStatement* is allowed.
- No *CaseRangeStatement*s are allowed.
- If the switch *Expression* is of enum type, all the enum members must appear in the *CaseStatement*s.
- The case expressions cannot evaluate to a run time initialized value.

11.15 Continue Statement

```
ContinueStatement:
    continue Identifier_opt ;
```

`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, foreach, 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 throw out of the finally clause, the continue target is never reached.

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

11.16 Break Statement

BreakStatement:
    `break` *Identifier*$_{\text{opt}}$;

`break` exits the innermost enclosing while, for, foreach, 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 throw out of the finally clause, the break target is never reached.

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

11.17 Return Statement

ReturnStatement:
    `return` *Expression*$_{\text{opt}}$;

`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, unless the function contains inline assembler code.

Before the function actually returns, any objects with scope 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;
}
```

11.18 Goto Statement

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

`goto` transfers to the statement labeled 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.

11.19 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 and is not copied. 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
{
    alias Y = int;
}
...
Y y;        // error, Y undefined
with (Foo)
{
    Y y;    // same as Foo.Y y;
}
```

Use of with object symbols that shadow local symbols with the same identifier are not allowed. This is to reduce the risk of inadvertent breakage of with statements when new members are added to the object declaration.

```
struct S
{
    float x;
}

void main()
{
    int x;
    S s;
    with (s)
    {
        x++;  // error, shadows the int x declaration
    }
}
```

In nested *WithStatement*s, the inner-most scope takes precedence. If a symbol cannot be resolved at the inner-most scope, resolution is forwarded incrementally up the scope hierarchy.

```
import std.stdio;

struct Foo
{
    void f() { writeln("Foo.f"); }
```

```
}

struct Bar
{
    void f() { writeln("Bar.f"); }
}

struct Baz
{
    // f() is not implemented
}

void f()
{
    writeln("f");
}

void main()
{
    Foo foo;
    Bar bar;
    Baz baz;

    f();               // prints "f"

    with(foo)
    {
        f();           // prints "Foo.f"

        with(bar)
        {
            f();       // prints "Bar.f"

            with(baz)
            {
                f();   // prints "Bar.f".  `Baz` does not implement `f()` so
                       // resolution is forwarded to `with(bar)`'s scope
            }
        }
        with(baz)
```

```
        {
            f();        // prints "Foo.f".  `Baz` does not implement `f()` so
                        // resolution is forwarded to `with(foo)`'s scope
        }
    }
    with(baz)
    {
        f();            // prints "f".  `Baz` does not implement `f()` so
                        // resolution is forwarded to `main`'s scope. `f()` is
                        // not implemented in `main`'s scope, so resolution is
                        // subsequently forward to module scope.
    }
}
```

11.20 Synchronized Statement

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

```
SynchronizedStatement:
    synchronized ScopeStatement
    synchronized ( Expression ) ScopeStatement
```

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

What mutex is used is determined by the *Expression*. If there is no *Expression*, then a global mutex is created, one per such synchronized statement. Different synchronized statements will have different global mutexes.

If there is an *Expression*, it must evaluate to either an Object or an instance of an *Interface*, in which case it is cast to the Object instance that implemented that *Interface*. The mutex used is specific to that Object instance, and is shared by all synchronized statements referring to that instance.

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

Example:

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

This implements a standard critical section.

Synchronized statements support recursive locking; that is, a function wrapped in synchronized is allowed to recursively call itself and the behavior will be as expected: The mutex will be locked and unlocked as many times as there is recursion.

11.21 Try Statement

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

```
TryStatement:
    try ScopeStatement Catches
    try ScopeStatement Catches FinallyStatement
    try ScopeStatement FinallyStatement
```

```
Catches:
    Catch
    Catch Catches
```

```
Catch:
    catch ( CatchParameter ) NoScopeNonEmptyStatement
```

```
CatchParameter:
    BasicType Identifier_opt
```

```
FinallyStatement:
    finally NoScopeNonEmptyStatement
```

CatchParameter declares a variable v of type T, where T is Throwable or derived from Throwable. 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.

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 original exception is caught, it is chained to the previous exception via the *next* member of *Throwable*. Note that, in contrast to most other programming languages, the new exception does not replace the original exception. Instead, later exceptions are regarded as 'collateral damage' caused by the first exception. The original exception must be caught, and this results in the capture of the entire chain.

Thrown objects derived from *Error* are treated differently. They bypass the normal chaining mechanism, such that the chain can only be caught by catching the first *Error*. In addition to the list of subsequent exceptions, *Error* also contains a pointer that points to the original exception (the head of the chain) if a bypass occurred, so that the entire exception history is retained.

```
import std.stdio;

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

prints:

```
finally
catch first
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.

11.22 Throw Statement

Throws an exception.

```
ThrowStatement:
    throw Expression ;
```

Expression is evaluated and must be a `Throwable` reference. The `Throwable` reference is thrown as an exception.

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

11.23 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 will be executed in the reverse lexical order in which they appear. If any scope instances are to be destroyed upon the close of the scope, their destructions will be interleaved with the *ScopeGuardStatement*s in the reverse lexical order in which they appear.

```
write("1");
{
    write("2");
    scope(exit) write("3");
    scope(exit) write("4");
```

```
    write("5");
}
writeln();
```

writes:

```
12543
```

```
{
    scope(exit) write("1");
    scope(success) write("2");
    scope(exit) write("3");
    scope(success) write("4");
}
writeln();
```

writes:

```
4321
```

```
struct Foo
{
    this(string s) { write(s); }
    ~this() { write("1"); }
}

try
{
    scope(exit) write("2");
    scope(success) write("3");
    Foo f = Foo("0");
    scope(failure) write("4");
    throw new Exception("msg");
    scope(exit) write("5");
    scope(success) write("6");
    scope(failure) write("7");
}
catch (Exception e)
{
}
writeln();
```

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.

Catching C++ Class Objects

On many platforms, catching C++ class objects is supported. Catching C++ objects and D objects cannot both be done in the same *TryStatement*. Upon exit from the *Catch*, any destructors for the C++ object will be run and the storage used for it reclaimed. C++ objects cannot be caught in `@safe` code.

11.24 Asm Statement

Inline assembler is supported with the asm statement:

```
AsmStatement:
    asm FunctionAttributes opt { AsmInstructionList opt }


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 ... */
}
```

Semantically consecutive *AsmStatement*s shall not have any other instructions (such as register save or restores) inserted between them by the compiler.

11.25 Pragma Statement

```
PragmaStatement:
    Pragma NoScopeStatement
```

11.26 Mixin Statement

```
MixinStatement:
    mixin ( ArgumentList ) ;
```

Each *AssignExpression* in the *ArgumentList* is evaluated at compile time, and the result must be representable as a string. The resulting strings are concatenated to form a string. The text contents of the string must be compilable as a valid *StatementList*, and is compiled as such.

```
import std.stdio;

void main()
{
    int j;
    mixin("
␣␣␣␣␣␣␣␣int␣x␣=␣3;
␣␣␣␣␣␣␣␣for␣(int␣i␣=␣0;␣i␣<␣3;␣i++)
␣␣␣␣␣␣␣␣␣␣␣␣writeln(x␣+␣i,␣++j);
␣␣␣␣␣␣␣␣");    // ok

    string s = "int␣y;";
    mixin(s);  // ok
    y = 4;     // ok, mixin declared y

    string t = "y␣=␣3;";
    mixin(t);  // error, t is not evaluatable at compile time

    mixin("y␣=") 4; // error, string must be complete statement

    mixin("y␣=" ~ "4;");  // ok

    mixin("y␣=", 2+2, ";");  // ok
}
```

Chapter 12

Arrays

12.1 Kinds

There are four kinds of arrays:

Kinds of Arrays

Syntax	Description
*type**	Pointers to data
type[*integer*]	Static arrays
type[]	Dynamic arrays
type[*type*]	Associative arrays

Pointers

```
int* p;
```

A pointer to type T has a value which is a reference (address) to another object of type T. It is commonly called a *pointer to T*.

If a pointer contains a *null* value, it is not pointing to a valid object.

When a pointer to T is dereferenced, it must either contain a *null* value, or point to a valid object of type T.

Implementation Defined:

1. The behavior when a *null* pointer is dereferenced. Typically the program will be aborted.

Undefined Behavior:

1. Dereferencing a pointer that is not *null* and does not point to a valid object of type T.

Best Practices: These are simple pointers to data. 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 `ref` parameters, and reference types.

Static Arrays

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

Static arrays have a length fixed at compile time.

The total size of a static array cannot exceed 16Mb.

A static array with a dimension of 0 is allowed, but no space is allocated for it.

Static arrays are value types. They are passed to and returned by functions by value.

Best Practices:

1. Use dynamic arrays for larger arrays.
2. Static arrays with 0 elements are useful as the last member of a variable length struct, or as the degenerate case of a template expansion.
3. Because static arrays are passed to functions by value, a larger array can consume a lot of stack space. Use dynamic arrays instead.

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.

Best Practices:

1. Use dynamic arrays instead of pointers to arrays as much as practical. Indexing of dynamic arrays are bounds checked, avoiding buffer underflow and overflow problems.

12.2 Array Declarations

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
```

12.3 Array 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.

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[] b;

p = s.ptr; // p points to the first element of the array s.
p = a.ptr; // p points to the first element of the array a.

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
```

12.4 Indexing

See also *IndexExpression*.

12.5 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:

```
void foo(int value)
{
  writeln("value=", value);
}

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 = [ 1,2,3,4,5,6,7,8,9,10 ];
int[] b1, b2, b3, b4;

b1 = a;
b2 = a[];
b3 = a[0 .. a.length];
b4 = a[0 .. $];
writeln(b1);
writeln(b2);
writeln(b3);
writeln(b4);
```

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[10] a = [ 1,2,3,4,5,6,7,8,9,10 ];

int* p = &a[2];
int[] b = p[0..8];
writeln(b);
writeln(p[7]);       // 10
writeln(p[8]);       // undefined behaviour
writeln(b[7]);       // 10
//writeln(b[8]);     // range error
```

See also *SliceExpression*.

12.6 Array Copying

When the slice operator appears as the left-hand side 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 left-hand side is a slice, and the right-hand side is an array of or pointer to the same type.

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

```
s = t;             // the 3 elements of t are copied into s
s[] = t;           // the 3 elements of t are copied into s
s[] = t[];         // the 3 elements of t are copied into s
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, operands have different lengths
```

Overlapping Copying

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.

If overlapping is required, use `std.algorithm.mutation.copy`:

```
import std.algorithm;
int[] s = [1, 2, 3, 4];

copy(s[1..3], s[0..2]);
assert(s == [2, 3, 3, 4]);
```

12.7 Array Setting

If a slice operator appears as the left-hand side of an assignment expression, and the type of the right-hand side is the same as the element type of the left-hand side, then the array contents of the left-hand side are set to the right-hand side.

```
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
```

12.8 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 + 4
```

produce the number 17, the string "1034" or the string "107" 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
```

Appending does not always create a copy, see setting dynamic array length for details.

12.9 Array Operations

Many array operations, also known as vector operations, can be expressed at a high level rather than as a loop. For example, the loop:

```
T[] a, b;
...
for (size_t i = 0; i < a.length; i++)
    a[i] = b[i] + 4;
```

assigns to the elements of `a` the elements of `b` with `4` added to each. This can also be expressed in vector notation as:

```
T[] a, b;
...
a[] = b[] + 4;
```

A vector operation is indicated by the slice operator appearing as the left-hand side of an assignment or an op-assignment expression. The right-hand side can be an expression consisting either of an array slice of the same length and type as the left-hand side or a scalar expression of the element type of the left-hand side, in any combination.

```
T[] a, b, c;
...
a[] -= (b[] + 4) * c[];
```

The slice on the left and any slices on the right must not overlap. All operands are evaluated exactly once, even if the array slice has zero elements in it.

The order in which the array elements are computed is implementation defined, and may even occur in parallel. An application must not depend on this order.

Implementation note: Many vector operations are expected to take advantage of any vector math instructions available on the target computer.

12.10 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
}
```

12.11 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 in a contiguous block of memory:

```
double[6][3] matrix = 0; // Sets all elements to 0.
writeln(matrix); // [[0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0]]
```

Note that dimensions and indices appear in opposite orders. Dimensions in the declaration are read right to left whereas indices are read left to right:

```
double[6][3] matrix = 0;
matrix[2][5] = 3.14; // Assignment to bottom right element.
writeln(matrix); // [[0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 3.14]]

static assert(!__traits(compiles, matrix[5][2])); // Array index out of bounds.
```

12.12 Array Length

Within the [ ] of a static or a dynamic array, the symbol `$` represents the length of the array.

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

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


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

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

12.13 Array Properties

Static array properties are:

Static Array Properties

Property	Description
`.init`	Returns an array literal with each element of the literal being the `.init` property of the array element type.
`.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. It is of type `size_t`.
`.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. The copy will have any immutability or const stripped. If this conversion is invalid the call will not compile.
`.idup`	Create a dynamic array of the same size and copy the contents of the array into it. The copy is typed as being immutable. If this conversion is invalid the call will not compile.

Dynamic array properties are:

Dynamic Array Properties

Property	Description
`.init`	Returns `null`.
`.sizeof`	Returns the size of the dynamic array reference, which is 8 in 32-bit builds and 16 on 64-bit builds.
`.length`	Get/set number of elements in the array. It is of type `size_t`.
`.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. The copy will have any immutability or const stripped. If this conversion is invalid the call will not compile.
`.idup`	Create a dynamic array of the same size and copy the contents of the array into it. The copy is typed as being immutable. If this conversion is invalid the call will not compile.

Examples:

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

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 left-hand side 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, the array is not reallocated, and no data is copied. It is equivalent to slicing the array:

```
array = array[0..7];
```

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 resizing in place would overwrite valid data in the array. For example:

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

b.length = 15; // always reallocates because extending in place would
               // overwrite other data in a.
b[11] = 'x';   // a[11] and c[1] are not affected

d.length = 1;
d.length = 20; // also reallocates, because doing this will overwrite a and
               // c
```

```
c.length = 12; // may reallocate in place if space allows, because nothing
               // was allocated after c.
c[5] = 'y';    // may affect contents of a, but not b or d because those
               // were reallocated.

a.length = 25; // This always reallocates because if c extended in place,
               // then extending a would overwrite c.  If c didn't
               // reallocate in place, it means there was not enough space,
               // which will still be true for a.
a[15] = 'z';   // does not affect c, because either a or c has reallocated.
```

To guarantee copying behavior, use the .dup property to ensure a unique array that can be resized. Also, one may use the `.capacity` property to determine how many elements can be appended to the array without reallocating.

These issues also apply to appending arrays with the = operator. Concatenation using the operator is not affected since it always reallocates.

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

```
int[] array;
while (1)
{
    import core.stdc.stdio : getchar;
    auto c = getchar;
    if (!c)
        break;
    ++array.length;
    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
int i;
for (i = 0; ; i++)
{
    import core.stdc.stdio : getchar;
    auto c = getchar;
    if (!c)
```

```
        break;
    if (i == 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 99For example, when gathering user input from the console - it's unlikely to be longer than 80.

Also, you may wish to utilize the `reserve` function to pre-allocate array data to use with the append operator.

```
int[] array;
size_t cap = array.reserve(10); // request
array ~= [1, 2, 3, 4, 5];
assert(cap >= 10); // allocated may be more than request
assert(cap == array.capacity);
```

Functions as Array Properties

See Uniform Function Call Syntax (UFCS).

12.14 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, a RangeError 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:

```
import core.exception;
try
{
    auto array = [1, 2];
    for (auto i = 0; ; i++)
    {
        array[i] = 5;
    }
}
catch (RangeError)
{
    // terminate loop
```

```
}
```

The loop is correctly written:

```
auto array = [1, 2];
for (auto 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.

Undefined Behavior: An out of bounds memory access will cause undefined behavior, therefore array bounds check is normally enabled in `@safe` functions. The runtime behavior is part of the language semantics.

See also Safe Functions.

Disabling Array Bounds Checking

Insertion of array bounds checking code at runtime may be turned off with a compiler switch `-boundscheck`.

If the bounds check in `@system` or `@trusted` code is disabled, the code correctness must still be guaranteed by the code author.

On the other hand, disabling the bounds check in `@safe` code will break the guaranteed memory safety by compiler. It's not recommended unless motivated by speed measurements.

12.15 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.

to void initialise the elements of dynamic array use `std.array.uninitializedArray`.

Static Initialization of Statically Allocated 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[Color.max + 1] value =
  [ Color.blue :6,
    Color.green:2,
    Color.red  :5 ];
```

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

12.16 Special Array Types

Strings

A string is an array of characters. String literals are just an easy way to write character arrays. String literals are immutable (read only).

```
char[] str1 = "abc";                // error, "abc" is not mutable
char[] str2 = "abc".dup;            // ok, make mutable copy
immutable(char)[] str3 = "abc";     // ok
immutable(char)[] str4 = str1;      // error, str4 is not mutable
immutable(char)[] str5 = str1.idup; // ok, make immutable copy
```

The name `string` is aliased to `immutable(char)[]`, so the above declarations could be equivalently written as:

```
char[] str1 = "abc";     // error, "abc" is not mutable
char[] str2 = "abc".dup; // ok, make mutable copy
string str3 = "abc";     // ok
string str4 = str1;      // error, str4 is not mutable
string str5 = str1.idup; // ok, make immutable copy
```

`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(immutable(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 string, wstring, and dstring 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'
```

Strings and Unicode

Note that built-in comparison operators operate on a code unit basis. The end result for valid strings is the same as that of code point for code point comparison as long as both strings are in the same normalization form. Since normalization is a costly operation not suitable for language primitives it's assumed to be enforced by the user.

The standard library lends a hand for comparing strings with mixed encodings (by transparently decoding, see `std.algorithm.cmp`), case-insensitive comparison and normalization.

Last but not least, a desired string sorting order differs by culture and language and is usually nothing like code point for code point comparison. The natural order of strings is obtained by applying the Unicode collation algorithm that should be implemented in the standard library.

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", str.ptr);
```

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".ptr);
```

So, why does the first string literal to printf not need the `.ptr`? The first parameter is prototyped as a `const(char)*`, and a string literal can be implicitly `cast` to a `const(char)*`. The rest of the arguments to printf, however, are variadic (specified by ...), and a string literal typed `immutable(char)[]` cannot pass to variadic parameters.

The second way is to use the precision specifier. The length comes first, followed by the pointer:

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

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

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

Void Arrays

There is a special type of array which acts as a wildcard that can hold arrays of any kind, declared as `void[]`. Void arrays are used for low-level operations where some kind of array data is being handled, but the exact type of the array elements are unimportant. The `.length` of a void array is the length of the data in bytes, rather than the number of elements in its original type. Array indices in indexing and slicing operations are interpreted as byte indices.

Arrays of any type can be implicitly converted to a void array; the compiler inserts the appropriate calculations so that the `.length` of the resulting array's size is in bytes rather than number of elements. Void arrays cannot be converted back to the original type without using a cast, and it is an error to convert to an array type whose element size does not evenly divide the length of the void array.

```
void main()
{
    int[] data1 = [1,2,3];
    long[] data2;

    void[] arr = data1;            // OK, int[] implicit converts to void[].
    assert(data1.length == 3);
    assert(arr.length == 12);      // length is implicitly converted to bytes.

    //data1 = arr;                 // Illegal: void[] does not implicitly
                                   // convert to int[].
    int[] data3 = cast(int[]) arr; // OK, can convert with explicit cast.
    data2 = cast(long[]) arr;      // Runtime error: long.sizeof == 8, which
                                   // does not divide arr.length, which is 12
                                   // bytes.
}
```

Void arrays can also be static if their length is known at compile-time. The length is specified in bytes:

```
void main()
{
    byte[2] x;
    int[2] y;

    void[2] a = x; // OK, lengths match
    void[2] b = y; // Error: int[2] is 8 bytes long, doesn't fit in 2 bytes.
}
```

While it may seem that void arrays are just fancy syntax for `ubyte[]`, there is a subtle distinction. The garbage collector generally will not scan `ubyte[]` arrays for pointers, `ubyte[]` being presumed to contain only pure byte data, not pointers. However, it *will* scan `void[]` arrays for pointers, since such an array may have been implicitly converted from an array of pointers or an array of elements that contain pointers. Allocating an array that contains pointers as `ubyte[]` may run the risk of the GC collecting live memory if these pointers are the only remaining references to their targets.

12.17 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[]`
- `const(U)[]`
- `const(U[])`
- `void[]`

A dynamic array `T[]` can be implicitly converted to one of the following (`U` is a base class of `T`):

- `const(U)[]`
- `const(U[])`
- `void[]`

Chapter 13

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[string] aa;   // Associative array of ints that are
                  // indexed by string keys.
                  // The KeyType is string.
aa["hello"] = 3;  // set value associated with key "hello" to 3
int value = aa["hello"];  // lookup value from a key
assert(value == 3);
```

Neither the *KeyTypes* nor the element types of an associative array can be function types or `void`.

Implementation Defined: The built-in associative arrays do not preserve the order of the keys inserted into the array. In particular, in a `foreach` loop the order in which the elements are iterated is typically unspecified.

13.1 Removing Keys

Particular keys in an associative array can be removed with the `remove` function:

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

`remove(key)` does nothing if the given *key* does not exist and returns `false`. If the given *key* does exist, it removes it from the AA and returns `true`.

All keys can be removed by using the method `clear`.

13.2 Testing Membership

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 aa);
if (p !is null)
{
    *p = 4;  // update value associated with key
    assert(aa["hello"] == 4);
}
```

Undefined Behavior: Adjusting the pointer to point before or after the element whose address is returned, and then dereferencing it.

13.3 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`:

- `size_t toHash() @trusted nothrow`
- `bool opEquals(Object)`

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

For example:

```
class Foo
{
    int a, b;

    override size_t toHash() { return a + b; }

    override bool opEquals(Object o)
    {
        Foo foo = cast(Foo) o;
        return foo && a == foo.a && b == foo.b;
    }
}
```

Implementation Defined: `opCmp` is not used to check for equality by the associative array. However, since the actual `opEquals` or `opCmp` called is not decided until runtime, the compiler cannot always detect mismatched functions. Because of legacy issues, the compiler may reject an associative array key type that overrides `opCmp` but not `opEquals`. This restriction may be removed in future versions.

Undefined Behavior:

1. If `toHash` must consistently be the same value when `opEquals` returns true. In other words, two objects that are considered equal should always have the same hash value. Otherwise, undefined behavior will result.

Best Practices:

1. Use the attributes `@safe`, `@nogc`, `pure`, `const`, and `scope` as much as possible on the `toHash` and `opEquals` overrides.

13.4 Using Structs or Unions as the KeyType

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

```
size_t toHash() const @safe pure nothrow;
bool opEquals(ref const typeof(this) s) const @safe pure nothrow;
```

For example:

```
import std.string;

struct MyString
{
    string str;

    size_t toHash() const @safe pure nothrow
    {
        size_t hash;
        foreach (char c; str)
            hash = (hash * 9) + c;
        return hash;
    }

    bool opEquals(ref const MyString s) const @safe pure nothrow
```

```
    {
        return std.string.cmp(this.str, s.str) == 0;
    }
}
```

The functions can use `@trusted` instead of `@safe`.

Implementation Defined: `opCmp` is not used to check for equality by the associative array. For this reason, and for legacy reasons, an associative array key is not allowed to define a specialized `opCmp`, but omit a specialized `opEquals`. This restriction may be removed in future versions of D.

Undefined Behavior:

1. If `toHash` must consistently be the same value when `opEquals` returns true. In other words, two structs that are considered equal should always have the same hash value. Otherwise, undefined behavior will result.

Best Practices:

1. Use the attributes `@nogc` as much as possible on the `toHash` and `opEquals` overrides.

13.5 Construction or Assignment on Setting AA Entries

When an AA indexing access appears on the left side of an assignment operator, it is specially handled for setting an AA entry associated with the key.

```
string[int] aa;
string s;
s = aa[1];          // throws RangeError in runtime
aa[1] = "hello";    // handled for setting AA entry
s = aa[1];          // succeeds to lookup
assert(s == "hello");
```

If the assigned value type is equivalent with the AA element type:

1. If the indexing key does not yet exist in AA, a new AA entry will be allocated, and it will be initialized with the assigned value.
2. If the indexing key already exists in the AA, the setting runs normal assignment.

```
struct S
{
    int val;
    void opAssign(S rhs) { this.val = rhs.val * 2; }
}
```

```
S[int] aa;
aa[1] = S(10);  // first setting initializes the entry aa[1]
assert(aa[1].val == 10);
aa[1] = S(10);  // second setting invokes normal assignment, and
                // operator-overloading rewrites it to member opAssign function.
assert(aa[1].val == 20);
```

If the assigned value type is **not** equivalent with the AA element type, the expression could invoke operator overloading with normal indexing access:

```
struct S
{
    int val;
    void opAssign(int v) { this.val = v * 2; }
}
S[int] aa;
aa[1] = 10;     // is rewritten to: aa[1].opAssign(10), and
                // throws RangeError before opAssign is called
```

However, if the AA element type is a struct which supports an implicit constructor call from the assigned value, implicit construction is used for setting the AA entry:

```
struct S
{
    int val;
    this(int v) { this.val = v; }
    void opAssign(int v) { this.val = v * 2; }
}
S s = 1;    // OK, rewritten to: S s = S(1);
  s = 1;    // OK, rewritten to: s.opAssign(1);

S[int] aa;
aa[1] = 10; // first setting is rewritten to: aa[1] = S(10);
assert(aa[1].val == 10);
aa[1] = 10; // second setting is rewritten to: aa[1].opAssign(10);
assert(aa[1].val == 20);
```

This is designed for efficient memory reuse with some value-semantics structs, eg. `std.bigint.BigInt`.

```
import std.bigint;
BigInt[string] aa;
aa["a"] = 10;   // construct BigInt(10) and move it in AA
aa["a"] = 20;   // call aa["a"].opAssign(20)
```

13.6 Inserting if not present

When AA access requires that there must be a value corresponding to the key, a value must be constructed and inserted if not present. The `require` function provides a means to construct a new value via a lazy argument. The lazy argument is evaluated when the key is not present. The `require` operation avoids the need to perform multiple key lookups.

```
class C{}
C[string] aa;

auto a = aa.require("a", new C);   // lookup "a", construct if not present
```

Sometimes it is necessary to know whether the value was constructed or already exists. The `require` function doesn't provide a boolean parameter to indicate whether the value was constructed but instead allows the construction via a function or delegate. This allows the use of any mechanism as demonstrated below.

```
class C{}
C[string] aa;

bool constructed;
auto a = aa.require("a", { constructed=true; return new C;}());
assert(constructed == true);

C newc;
auto b = aa.require("b", { newc = new C; return newc;}());
assert(b is newc);
```

13.7 Advanced updating

Typically updating a value in an associative array is simply done with an assign statement.

```
int[string] aa;

aa["a"] = 3;  // set value associated with key "a" to 3
```

Sometimes it is necessary to perform different operations depending on whether a value already exists or needs to be constructed. The `update` function provides a means to construct a new value via the `create` delegate or update an existing value via the `update` delegate. The `update` operation avoids the need to perform multiple key lookups.

```
class C{}
C[string] aa;
```

```
C older;
C newer;
aa.update("a",
{
    newer = new C;
    return newer;
},
(ref C c)
{
    older = c;
    newer = new C;
    return newer;
});
```

13.8 Static Initialization of AAs

NOTE: Not yet implemented.

```
immutable long[string] aa = [
  "foo": 5,
  "bar": 10,
  "baz": 2000
];

unittest
{
    assert(aa["foo"] == 5);
    assert(aa["bar"] == 10);
    assert(aa["baz"] == 2000);
}
```

13.9 Runtime Initialization of Immutable AAs

Immutable associative arrays are often desirable, but sometimes initialization must be done at runtime. This can be achieved with a constructor (static constructor depending on scope), a buffer associative array and `assumeUnique`:

```
immutable long[string] aa;
```

```
shared static this()
{
    import std.exception : assumeUnique;
    import std.conv : to;

    long[string] temp; // mutable buffer
    foreach(i; 0 .. 10)
    {
        temp[to!string(i)] = i;
    }
    temp.rehash; // for faster lookups

    aa = assumeUnique(temp);
}

unittest
{
    assert(aa["1"] == 1);
    assert(aa["5"] == 5);
    assert(aa["9"] == 9);
}
```

13.10 Construction and Reference Semantics

An Associative Array defaults to `null`, and is constructed upon assigning the first key/value pair. However, once constructed, an associative array has *reference semantics*, meaning that assigning one array to another does not copy the data. This is especially important when attempting to create multiple references to the same array.

```
int[int] aa;             // defaults to null
int[int] aa2 = aa;       // copies the null reference
aa[1] = 1;
assert(aa2.length == 0); // aa2 still is null
aa2 = aa;
aa2[2] = 2;
assert(aa[2] == 2);      // now both refer to the same instance
```

13.11 Properties

Properties for associative arrays are:

Associative Array Properties

Property	Description
`.sizeof`	Returns the size of the reference to the associative array; it is 4 in 32-bit builds and 8 on 64-bit builds.
`.length`	Returns number of values in the associative array. Unlike for dynamic arrays, it is

<hr>

Associative Array Example: word count

Let's consider the file is ASCII encoded with LF EOL. In general case we should use *dchar c* for iteration over code points and functions from std.uni.

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

void main (string[] args)
{
    ulong totalWords, totalLines, totalChars;
    ulong[string] dictionary;

    writeln("   lines   words   bytes file");
    foreach (arg; args[1 .. $]) // for each argument except the first one
    {
        ulong wordCount, lineCount, charCount;

        foreach(line; File(arg).byLine())
        {
            bool inWord;
            size_t wordStart;

            void tryFinishWord(size_t wordEnd)
            {
                if (inWord)
                {
                    auto word = line[wordStart .. wordEnd];
                    ++dictionary[word.idup];   // increment count for word
                    inWord = false;
                }
            }

            foreach (i, char c; line)
            {
                if (std.ascii.isDigit(c))
                {
                    // c is a digit (0..9)
```

```
            }
            else if (std.ascii.isAlpha(c))
            {
                // c is an ASCII letter (A..Z, a..z)
                if (!inWord)
                {
                    wordStart = i;
                    inWord = true;
                    ++wordCount;
                }
            }
            else
                tryFinishWord(i);
            ++charCount;
        }
        tryFinishWord(line.length);
        ++lineCount;
    }

    writefln("%8s%8s%8s␣%s", lineCount, wordCount, charCount, arg);
    totalWords += wordCount;
    totalLines += lineCount;
    totalChars += charCount;
}

if (args.length > 2)
{
    writefln("-------------------------------------\n%8s%8s%8s␣total",
             totalLines, totalWords, totalChars);
}

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

Associative Array Example: counting Tuples

An Associative Array can be iterated in key/value fashion using a foreach statement. As an example, the number of occurrences of all possible substrings of length 2 (aka 2-mers) in a string will be counted:

```
import std.range : dropOne, only, save, zip;
import std.stdio : writefln;
import std.typecons : Tuple;
import std.utf : byCodeUnit; // avoids UTF-8 auto-decoding

int[Tuple!(immutable char, immutable char)] aa;

// The string `arr` has a limited alphabet: {A, C, G, T}
// Thus, for better performance, iteration can be done _without_ decoding
auto arr = "AGATAGA".byCodeUnit;

// iterate over all pairs in the string and observe each pair
// ('A', 'G'), ('G', 'A'), ('A', 'T'), ...
foreach (window; arr.zip(arr.save.dropOne))
    aa[window]++;

// iterate over all key/value pairs of the Associative Array
foreach (key, value; aa)
{
    // the second parameter uses tuple-expansion
    writefln("key: %s [%s], value: %d", key, key.expand.only, value);
}
```

```
% rdmd count.d
key: Tuple!(immutable(char), immutable(char))('A', 'G') [AG], value: 2
key: Tuple!(immutable(char), immutable(char))('G', 'A') [GA], value: 2
key: Tuple!(immutable(char), immutable(char))('A', 'T') [AT], value: 1
key: Tuple!(immutable(char), immutable(char))('T', 'A') [TA], value: 1
```

Chapter 14

Structs and Unions

Whereas classes are reference types, structs are value types. Structs and unions are simple aggregations of data and their associated operations on that data.

```
AggregateDeclaration:
   ClassDeclaration
   InterfaceDeclaration
   StructDeclaration
   UnionDeclaration
```

```
StructDeclaration:
   struct Identifier ;
   struct Identifier AggregateBody
   StructTemplateDeclaration
   AnonStructDeclaration
```

```
AnonStructDeclaration:
   struct AggregateBody
```

```
UnionDeclaration:
   union Identifier ;
   union Identifier AggregateBody
   UnionTemplateDeclaration
   AnonUnionDeclaration
```

AnonUnionDeclaration:
  `union` *AggregateBody*

AggregateBody:
  `{` *DeclDefs*$_{\text{opt}}$ `}`

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

Structs and unions may not contain an instance of themselves, however, they may contain a pointer to the same type.

Best Practices:

1. Bit fields are supported with the bitfields template.

14.1 Struct Layout

The non-static data members of a struct are called *fields*. Fields are laid out in lexical order. Fields are aligned according to the Align Attribute in effect. Unnamed padding is inserted between fields to align fields. There is no padding between the first field and the start of the object.

Structs with no fields of non-zero size (aka *Empty Structs*) have a size of one byte.

Non-static function-nested D structs, which access the context of their enclosing scope, have an extra field.

Implementation Defined:

1. The default layout of the fields of a struct is an exact match with the *associated C compiler*.
2. g++ and clang++ differ in how empty structs are handled. Both return `1` from `sizeof`, however, clang++ does not push them onto the parameter stack while g++ does. This is a binary incompatibility between g++ and clang++. dmd follows clang++ behavior for OSX and FreeBSD, and g++ behavior for Linux and other Posix platforms.
3. clang and gcc both return `0` from `sizeof` for empty structs. Using `extern "C++"` in clang++ and g++ does not cause them to conform to the behavior of their respective C compilers.

Undefined Behavior:

1. The padding data can be accessed, but its contents are undefined.
2. Do not pass or return structs with no fields of non-zero size to `extern (C)` functions. According to C11 6.7.2.1p8 this is undefined behavior.

Best Practices:

1. When laying out a struct to match an externally defined layout, use align attributes to describe an exact match. Using a Static Assert to ensure the result is as expected.
2. Although the contents of the padding are often zero, do not rely on that.
3. Avoid using empty structs when interfacing with C and C++ code.
4. Avoid using empty structs as parameters or arguments to variadic functions.

14.2 Plain Old Data

A struct or union is *Plain Old Data* (POD) if it meets the following criteria:

1. it is not nested
2. it has no postblits, destructors, or assignment operators
3. it has no `ref` fields or fields that are themselves non-POD

Best Practices: Structs or unions that interface with C code should be POD.

14.3 Opaque Structs and Unions

Opaque struct and union declarations do not have a *AggregateBody*:

```
struct S;
union U;
struct V(T);
union W(T);
```

The members are completely hidden to the user, and so the only operations on those types are ones that do not require any knowledge of the contents of those types. For example:

```
struct S;
S.sizeof; // error, size is not known
S s;      // error, cannot initialize unknown contents
S* p;     // ok, knowledge of members is not necessary
```

Best Practices: They can be used to implement the PIMPL idiom.

14.4 Default Initialization of Structs

Struct fields are by default initialized to whatever the *Initializer* for the field is, and if none is supplied, to the default initializer for the field's type.

```
struct S { int a = 4; int b; }
S x; // x.a is set to 4, x.b to 0
```

The default initializers are evaluated at compile time.

The default initializers may not contain references to mutable data.

14.5 Static Initialization of Structs

If a *StructInitializer* is supplied, the fields are initialized by the *StructMemberInitializer* syntax. *StructMemberInitializers* with the *Identifier : NonVoidInitializer* syntax may be appear in any order, where *Identifier* is the field identifer. *StructMemberInitializer*s with the *NonVoidInitializer* syntax appear in the lexical order of the fields in the *StructDeclaration*.

Fields not specified in the *StructInitializer* are default initialized.

```
struct S { int a, b, c, d = 7; }
S r;                          // r.a = 0, r.b = 0, r.c = 0, r.d = 7
S s = { a:1, b:2 };           // s.a = 1, s.b = 2, s.c = 0, s.d = 7
S t = { c:4, b:5, a:2, d:5 }; // t.a = 2, t.b = 5, t.c = 4, t.d = 5
S u = { 1, 2 };               // u.a = 1, u.b = 2, u.c = 0, u.d = 7
S v = { 1, d:3 };             // v.a = 1, v.b = 0, v.c = 0, v.d = 3
S w = { b:1, 3 };             // w.a = 0, w.b = 1, w.c = 3, w.d = 7
```

Initializing a field more than once is an error:

```
S x = { 1, a:2 };  // error: duplicate initializer for field ‘a‘
```

14.6 Default Initialization of Unions

Unions are by default initialized to whatever the *Initializer* for the first field is, and if none is supplied, to the default initializer for the first field's type.

If the union is larger than the first field, the remaining bits are set to 0.

```
union U { int a = 4; long b; }
U x; // x.a is set to 4, x.b to an implementation-defined value

union V { int a; long b = 4; }
V y; // y.a is set to 0, y.b to an implementation-defined value

union W { int a = 4; long b = 5; } // error: overlapping default initialization for ‘a‘ and ‘b‘
```

The default initializer is evaluated at compile time.

Implementation Defined: The values the fields other than the default initialized field are set to.

14.7 Static Initialization of Unions

Unions are initialized similarly to structs, except that only one initializer is allowed.

```
union U { int a; double b; }
U u = { 2 };       // u.a = 2
U v = { b : 5.0 }; // v.b = 5.0

U w = { 2, 3 };    // error: overlapping initialization for field ‘a‘ and ‘b‘
```

If the union is larger than the initialized field, the remaining bits are set to 0.
Implementation Defined: The values the fields other than the initialized field are set to.

14.8 Dynamic Initialization of Structs

The static initializer syntax can also be used to initialize non-static variables. The initializer need not be evaluable at compile time.

```
struct S { int a, b, c, d = 7; }

void test(int i)
{
    S q = { 1, b:i }; // q.a = 1, q.b = i, q.c = 0, q.d = 7
}
```

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)
    {
        assert(0);
    }
}

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

14.9 Struct Literals

Struct literals consist of the name of the struct followed by a parenthesized argument list:

```
struct S { int x; float y; }

int foo(S s) { return s.x; }

foo( S(1, 2) ); // set field x to 1, field y to 2
```

Struct literals are syntactically like function calls. If a struct has a member function named `opCall`, then struct literals for that struct are not possible. See also opCall operator overloading for the issue workaround. It is an error if there are more arguments than fields of the struct. If there are fewer arguments than fields, the remaining fields are initialized with their respective default initializers. If there are anonymous unions in the struct, only the first member of the anonymous union can be initialized with a struct literal, and all subsequent non-overlapping fields are default initialized.

14.10 Struct Properties

Struct Properties

Name	Description
`.sizeof`	Size in bytes of struct
`.alignof`	Size boundary struct needs to be aligned on

14.11 Struct Instance Properties

Struct Instance Properties

Name	Description
`.tupleof`	An expression sequence of all struct fields - see Class Properties for a class-based example.

14.12 Struct Field Properties

Struct Field Properties

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

14.13 Const, Immutable and Shared Structs

A struct declaration can have a storage class of `const`, `immutable` or `shared`. It has an equivalent effect as declaring each member of the struct as `const`, `immutable` or `shared`.

```
const struct S { int a; int b = 2; }
void main()
{
    S s = S(3); // initializes s.a to 3
    S t;        // initializes t.a to 0
    t = s;      // error, t.a and t.b are const, so cannot modify them.
    t.a = 4;    // error, t.a is const
}
```

14.14 Struct Constructors

Struct constructors are used to initialize an instance of a struct when a more complex construction is needed than is allowed by static initialization or a struct literal.

Constructors are defined with a function name of `this` and having no return value. The grammar is the same as for the class *Constructor*.

A struct constructor is called by the name of the struct followed by *Parameters*. If the *ParameterList* is empty, the struct instance is default initialized.

```
struct S
{
    int x, y = 4, z = 6;
```

```
    this(int a, int b)
    {
        x = a;
        y = b;
    }
}

void main()
{
    S a = S(4, 5); // calls S.this(4, 5):  a.x = 4, a.y = 5, a.z = 6
    S b = S();  // default initialized:    a.x = 0, b.y = 4, b.y = 6
    S c = S(1); // error, matching this(int) not found
}
```

A *default constructor* (i.e. one with an empty *ParameterList*) is not allowed.

```
struct S
{
    int x;
    this() { } // error, struct default constructor not allowed
}
```

Constructors can call other constructors for the same struct in order to share common initializations (this is called a delegating constructor):

```
struct S
{
    int j = 1;
    long k = 2;
    this(long k)
    {
        this.k = k;
    }
    this(int i)
    {
        // At this point: j=1, k=2
        this(6); // delegating constructor call
        // At this point: j=1, k=6
        j = i;
        // At this point: j=i, k=6
    }
}
```

The following restrictions apply to struct construction:

1. If a constructor's code contains a delegate constructor call, all possible execution paths through the constructor must make exactly one delegate constructor call:

```
struct S
{
    int a;
    this(int i) { }

    this(char c)
    {
        c || this(1); // error, not on all paths
    }

    this(wchar w)
    {
        (w) ? this(1) : this('c'); } // ok

    this(byte b)
    {
        foreach (i; 0 .. b)
        {
            this(1);  // error, inside loop
        }
    }
}
```

2. It is illegal to refer to `this` implicitly or explicitly prior to making a delegate constructor call.
3. Once the delegate constructor returns, all fields are considered constructed.
4. Delegate constructor calls cannot appear after labels.

When an instance of a struct is created, the following steps happen:

1. The raw data is statically initialized using the values provided in the struct definition. This operation is equivalent to doing a memory copy of a static version of the object onto the newly allocated one.
2. If there is a constructor defined for the struct, the constructor matching the argument list is called.
3. If struct invariant checking is turned on, the struct invariant is called at the end of the constructor.

A constructor qualifier (`const`, `immutable` or `shared`) constructs the object instance with that specific qualifier.

```
struct S1
{
    int[] a;
    this(int n) { a = new int[](n); }
}
struct S2
{
    int[] a;
    this(int n) immutable { a = new int[](n); }
}
void main()
{
    // Mutable constructor creates mutable object.
    S1 m1 = S1(1);

    // Constructed mutable object is implicitly convertible to const.
    const S1 c1 = S1(1);

    // Constructed mutable object is not implicitly convertible to immutable.
    immutable i1 = S1(1); // error

    // Mutable constructor cannot construct immutable object.
    auto x1 = immutable S1(1); // error


    // Immutable constructor creates immutable object.
    immutable i2 = immutable S2(1);

    // Immutable constructor cannot construct mutable object.
    auto x2 = S2(1); // error

    // Constructed immutable object is not implicitly convertible to mutable.
    S2 m2 = immutable S2(1); // error

    // Constructed immutable object is implicitly convertible to const.
    const S2 c2 = immutable S2(1);
}
```

Constructors can be overloaded with different attributes.

```
struct S
{
    this(int);            // non-shared mutable constructor
    this(int) shared;     // shared mutable constructor
    this(int) immutable; // immutable constructor
}

S m = S(1);
shared s = shared S(2);
immutable i = immutable S(3);
```

If the constructor can create a unique object (i.e. if it is `pure`), the object is implicitly convertible to any qualifiers.

```
struct S
{
    this(int) pure;
    // Based on the definition, this creates a mutable object. But the
    // created object cannot contain any mutable global data.
    // Therefore the created object is unique.

    this(int[] arr) immutable pure;
    // Based on the definition, this creates an immutable object. But
    // the argument int[] never appears in the created object so it
    // isn't implicitly convertible to immutable. Also, it cannot store
    // any immutable global data.
    // Therefore the created object is unique.
}

immutable i = immutable S(1); // this(int) pure is called
shared s = shared S(1);       // this(int) pure is called
S m = S([1,2,3]);             // this(int[]) immutable pure is called
```

14.15 Disabling Default Struct Construction

If struct constructor is annotated with `@disable` and has an empty *ParameterList*, the struct has disabled default construction. The only way it can be constructed is via a call to another constructor with a non-empty *ParameterList*.

A struct with a disabled default constructor, and no other constructors, cannot be instantiated other than via a *VoidInitializer*.

A disabled default constructor may not have a *FunctionBody*.

If any fields have disabled default construction, the struct default construction is also disabled.

```
struct S
{
    int x;

    // Disables default construction
    @disable this();

    this(int v) { x = v; }
}
struct T
{
    int y;
    S s;
}
void main()
{
    S s;          // error: default construction is disabled
    S t = S();    // error: also disabled
    S u = S(1);   // constructed by calling `S.this(1)`
    S v = void;   // not initialized, but allowed
    S w = { 1 };  // error: cannot use { } since constructor exists
    S[3] a;       // error: default construction is disabled
    S[3] b = [S(1), S(20), S(-2)]; // ok
    T t;          // error: default construction is disabled
}
```

Best Practices: Disabling default construction is useful when the default value, such as `null`, is not acceptable.

14.16 Union Constructors

Unions are constructed in the same way as structs.

14.17 Field initialization inside constructor

In a constructor body, if a delegate constructor is called, all field assignments are considered assignments. Otherwise, the first instance of field assignment is its initialization, and assignments of the form `field = expression` are treated as equivalent to `typeof(field)(expression)`. The values of fields may be read before initialization or construction with a delegate constructor.

```
struct S
{
    int num;
    int ber;
    this(int i)
    {
        num = i + 1;   // initialization
        num = i + 2;   // assignment
        ber = ber + 1; // ok to read before initialization
    }
    this(int i, int j)
    {
        this(i);
        num = i + 1;  // assignment
    }
}
```

If the field type has an `opAssign` method, it will not be used for initialization.

```
struct A
{
    this(int n) {}
    void opAssign(A rhs) {}
}
struct S
{
    A val;
    this(int i)
    {
        val = A(i);  // val is initialized to the value of A(i)
        val = A(2);  // rewritten to val.opAssign(A(2))
    }
}
```

If the field type is not mutable, multiple initialization will be rejected.

```
struct S
{
    immutable int num;
    this(int)
    {
        num = 1;  // OK
        num = 2;  // Error: assignment to immutable
    }
}
```

If the field is initialized on one path, it must be initialized on all paths.

```
struct S
{
    immutable int num;
    immutable int ber;
    this(int i)
    {
        if (i)
            num = 3;   // initialization
        else
            num = 4;   // initialization
    }
    this(long j)
    {
        j ? (num = 3) : (num = 4); // ok
        j || (ber = 3);  // Error: intialized on only one path
        j && (ber = 3);  // Error: intialized on only one path
    }
}
```

A field initialization may not appear in a loop or after a label.

```
struct S
{
    immutable int num;
    immutable string str;
    this(int j)
    {
        foreach (i; 0..j)
        {
            num = 1;    // Error: field initialization not allowed in loops
```

```
        }
        size_t i = 0;
    Label:
        str = "hello";  // Error: field initialization not allowed after labels
        if (i++ < 2)
            goto Label;
    }
    this(int j, int k)
    {
        switch (j)
        {
            case 1: ++j; break;
            default: break;
        }
        num = j;        // Error: `case` and `default` are also labels
    }
}
```

If a field's type has disabled default construction, then it must be initialized in the constructor.

```
struct S { int y; @disable this(); }

struct T
{
    S s;
    this(S t) { s = t; }       // ok
    this(int i) { this('c'); } // ok
    this(char) { }             // Error: s not initialized
}
```

14.18 Struct Copy Constructors

Copy constructors are used to initialize a `struct` instance from another `struct` of the same type.

A constructor declaration is a copy constructor declaration if and only if it is a constructor declaration that takes only one non-default parameter by reference that is of the same type as `typeof(this)`, followed by any number of default parameters:

```
struct A
{
    this(ref return scope A rhs) {}                        // copy constructor
    this(ref return scope const A rhs, int b = 7) {}       // copy constructor with default pa
```

```
}
```

The copy constructor is type checked as a normal constructor.

If a copy constructor is defined, implicit calls to it will be inserted in the following situations:

1. When a variable is explicitly initialized:

   ```
   struct A
   {
       this(ref return scope A rhs) {}
   }

   void main()
   {
       A a;
       A b = a; // copy constructor gets called
   }
   ```

2. When a parameter is passed by value to a function:

   ```
   struct A
   {
       this(ref return scope A another) {}
   }

   void fun(A a) {}

   void main()
   {
       A a;
       fun(a);    // copy constructor gets called
   }
   ```

3. When a parameter is returned by value from a function and Named Returned Value Optiomization (NRVO) cannot be performed:

   ```
   struct A
   {
       this(ref return scope A another) {}
   }

   A fun()
   {
   ```

```
    A a;
    return a;       // NRVO, no copy constructor call
}

A a;
A gun()
{
    return a;       // cannot perform NRVO, rewrite to: return (A __tmp; __tmp.copyCtor(a
}

void main()
{
    A a = fun();
    A b = gun();
}
```

When a copy constructor is defined for a `struct`, all implicit blitting is disabled for that `struct`:

```
struct A
{
    int[] a;
    this(ref return scope A rhs) {}
}

void fun(immutable A) {}

void main()
{
    immutable A a;
    fun(a);          // error: copy constructor cannot be called with types (immutable) immuta
}
```

The copy constructor can be overloaded with different qualifiers applied to the parameter (copying from a qualified source) or to the copy constructor itself (copying to a qualified destination):

```
struct A
{
    this(ref return scope A another) {}                          // 1 - mutable source, mutable
    this(ref return scope immutable A another) {}                // 2 - immutable source, mutabl
    this(ref return scope A another) immutable {}                // 3 - mutable source, immutabl
    this(ref return scope immutable A another) immutable {}      // 4 - immutable source, immuta
```

```
}

void main()
{
    A a;
    immutable A ia;

    A a2 = a;      // calls 1
    A a3 = ia;     // calls 2
    immutable A a4 = a;     // calls 3
    immutable A a5 = ia;    // calls 4
}
```

The `inout` qualifier may be applied to the copy constructor parameter in order to specify that mutable, `const`, or `immutable` types are treated the same:

```
struct A
{
    this(ref return scope inout A rhs) immutable {}
}

void main()
{
    A r1;
    const(A) r2;
    immutable(A) r3;

    // All call the same copy constructor because `inout` acts like a wildcard
    immutable(A) a = r1;
    immutable(A) b = r2;
    immutable(A) c = r3;
}
```

A copy constructor is generated implicitly by the compiler for a `struct S` if all of the following conditions are met:

1. `S` does not explicitly declare any copy constructors;
2. `S` defines at least one direct member that has a copy constructor, and that member is not overlapped (by means of `union`) with any other member.

If the restrictions above are met, the following copy constructor is generated:

```
this(ref return scope inout(S) src) inout
{
    foreach (i, ref inout field; src.tupleof)
        this.tupleof[i] = field;
}
```

If the generated copy constructor fails to type check, it will receive the `@disable` attribute.

f an `union S` has fields that define a copy constructor, whenever an object of type `S` is initialized by copy, an error will be issued. The same rule applies to overlapped fields (anonymous unions).

A `struct` that defines a copy constructor is not a POD.

14.19 Struct Postblits

```
Postblit:
    this ( this ) MemberFunctionAttributes_opt ;
    this ( this ) MemberFunctionAttributes_opt FunctionBody
```

WARNING: The postblit is considered legacy and is not recommended for new code. Code should use copy constructors defined in the previous section. For backward compatibility reasons, a `struct` that defines both a copy constructor and a postblit will only use the postblit for implicit copying.

Copy construction is defined as initializing a struct instance from another struct of the same type. Copy construction is divided into two parts:

1. blitting the fields, i.e. copying the bits
2. running *postblit* on the result

The first part is done automatically by the language, the second part is done if a postblit function is defined for the struct. The postblit has access only to the destination struct object, not the source. Its job is to 'fix up' the destination as necessary, such as making copies of referenced data, incrementing reference counts, etc. For example:

```
struct S
{
    int[] a;    // array is privately owned by this instance
    this(this)
    {
        a = a.dup;
    }
}
```

Disabling struct postblit makes the object not copyable.

```
struct T
{
    @disable this(this);  // disabling makes T not copyable
}
struct S
{
    T t;   // uncopyable member makes S also not copyable
}

void main()
{
    S s;
    S t = s; // error, S is not copyable
}
```

Depending on the struct layout, the compiler may generate the following internal postblit functions:

1. `void __postblit()`. The compiler assigns this name to the explicitly defined postblit `this(this)` so that it can be treated exactly as a normal function. Note that if a struct defines a postblit, it cannot define a function named `__postblit` - no matter the signature - as this would result in a compilation error due to the name conflict.
2. `void __fieldPostblit()`. If a struct `X` has at least one `struct` member that in turn defines (explicitly or implicitly) a postblit, then a field postblit is generated for `X` that calls all the underlying postblits of the struct fields in declaration order.
3. `void __aggrPostblit()`. If a struct has an explicitly defined postblit and at least 1 struct member that has a postblit (explicit or implicit) an aggregated postblit is generated which calls `__fieldPostblit` first and then `__postblit`.
4. `void __xpostblit()`. The field and aggregated postblits, although generated for a struct, are not actual struct members. In order to be able to call them, the compiler internally creates an alias, called `__xpostblit` which is a member of the struct and which points to the generated postblit that is the most inclusive.

```
// struct with alias __xpostblit = __postblit
struct X
{
    this(this) {}
}
```

```
// struct with alias __xpostblit = __fieldPostblit
// which contains a call to X.__xpostblit
struct Y
{
    X a;
}

// struct with alias __xpostblit = __aggrPostblit which contains
// a call to Y.__xpostblit and a call to Z.__postblit
struct Z
{
    Y a;
    this(this) {}
}

void main()
{
    // X has __postblit and __xpostblit (pointing to __postblit)
    static assert(__traits(hasMember, X, "__postblit"));
    static assert(__traits(hasMember, X, "__xpostblit"));

    // Y does not have __postblit, but has __xpostblit (pointing to __fieldPostblit)
    static assert(!__traits(hasMember, Y, "__postblit"));
    static assert(__traits(hasMember, Y, "__xpostblit"));
    // __fieldPostblit is not a member of the struct
    static assert(!__traits(hasMember, Y, "__fieldPostblit"));

    // Z has  __postblit and __xpostblit (pointing to __aggrPostblit)
    static assert(__traits(hasMember, Z, "__postblit"));
    static assert(__traits(hasMember, Z, "__xpostblit"));
    // __aggrPostblit is not a member of the struct
    static assert(!__traits(hasMember, Z, "__aggrPostblit"));
}
```

Neither of the above postblits is defined for structs that don't define `this(this)` and don't have fields that transitively define it. If a struct does not define a postblit (implicit or explicit) but defines functions that use the same name/signature as the internally generated postblits, the compiler is able to identify that the functions are not actual postblits and does not insert calls to them when the struct is copied. Example:

```
struct X
```

```
{}

int a;

struct Y
{
    int a;
    X b;
    void __fieldPostPostblit()
    {
        a = 42;
    }
}

void main()
{
    static assert(!__traits(hasMember, X, "__postblit"));
    static assert(!__traits(hasMember, X, "__xpostblit"));

    static assert(!__traits(hasMember, Y, "__postblit"));
    static assert(!__traits(hasMember, Y, "__xpostblit"));

    Y y;
    auto y2 = y;
    assert(a == 0); // __fieldPostBlit does not get called
}
```

Postblits cannot be overloaded. If two or more postblits are defined, even if the signatures differ, the compiler assigns the `__postblit` name to both and later issues a conflicting function name error:

```
struct X
{
    this(this) {}
    this(this) const {} // error: function X.__postblit conflicts with function X.__postblit
}
```

The following describes the behavior of the qualified postblit definitions:

1. `const`. When a postblit is qualified with `const` as in `this(this) const;` or `const this(this);` then the postblit is succesfully called on mutable (unqualified), `const`,

and `immutable` objects, but the postblit cannot modify the object because it regards it as `const`; hence `const` postblits are of limited usefulness. Example:

```
struct S
{
    int n;
    this(this) const
    {
        import std.stdio : writeln;
        writeln("postblit␣called");
        //++n; // error: cannot modify this.n in ‘const‘ function
    }
}

void main()
{
    S s1;
    auto s2 = s1;
    const S s3;
    auto s4 = s3;
    immutable S s5;
    auto s6 = s5;
}
```

2. `immutable`. When a postblit is qualified with `immutable` as in `this(this) immutable` or `immutable this(this)` the code is ill-formed. The `immutable` postblit passes the compilation phase but cannot be invoked. Example:

```
struct Y
{
    // not invoked anywhere, no error is issued
    this(this) immutable
    { }
}

struct S
{
    this(this) immutable
    { }
}

void main()
```

```
{
    S s1;
    auto s2 = s1;    // error: immutable method `__postblit` is not callable using a mutable
    const S s3;
    auto s4 = s3;    // error: immutable method `__postblit` is not callable using a mutable
    immutable S s5;
    auto s6 = s5;    // error: immutable method `__postblit` is not callable using a mutable
}
```

3. `shared`. When a postblit is qualified with `shared` as in `this(this) shared` or `shared this(this)` solely `shared` objects may invoke the postblit; attempts of postbliting unshared objects will result in compile time errors:

```
struct S
{
    this(this) shared
    { }
}

void main()
{
    S s1;
    auto s2 = s1;    // error: shared method `__postblit` is not callable using a non-shared
    const S s3;
    auto s4 = s3;    // error: shared method `__postblit` is not callable using a non-shared
    immutable S s5;
    auto s6 = s5;    // error: shared method `__postblit` is not callable using a non-shared

    // calling the shared postblit on a shared object is accepted
    shared S s7;
    auto s8 = s7;
}
```

An unqualified postblit will get called even if the struct is instantiated as `immutable` or `const`, but the compiler issues an error if the struct is instantiated as `shared`:

```
struct S
{
    int n;
    this(this) { ++n; }
}
```

```
void main()
{
    immutable S a;      // shared S a; => error : non-shared method is not callable using a sh
    auto a2 = a;
    import std.stdio: writeln;
    writeln(a2.n);     // prints 1
}
```

From a postblit perspective, qualifiying the struct definition yields the same result as explicitly qualifying the postblit.

The following table lists all the possibilities of grouping qualifiers for a postblit associated with the type of object that needs to be used in order to succesfully invoke the postblit:

Qualifier Groups

object type to be invoked on	`const`	`immutable`	`shared`
any object type	✓		
uncallable		✓	
shared object			✓
uncallable	✓	✓	
shared object	✓		✓
uncallable		✓	✓
uncallable	✓	✓	✓

Note that when `const` and `immutable` are used to explicitly qualify a postblit as in `this(this) const immutable`; or `const immutable this(this)`; - the order in which the qualifiers are declared does not matter - the compiler generates a conflicting attribute error, however declaring the struct as `const`/`immutable` and the postblit as `immutable`/`const` achieves the effect of applying both qualifiers to the postblit. In both cases the postblit is qualified with the more restrictive qualifier, which is `immutable`.

The postblits `__fieldPostblit` and `__aggrPostblit` are generated without any implicit qualifiers and are not considered struct members. This leads to the situation where qualifying an entire struct declaration with `const` or `immutable` does not have any impact on the above-mentioned postblits. However, since `__xpostblit` is a member of the struct and an alias of one of the other postblits, the qualifiers applied to the struct will affect the aliased postblit.

```
struct S
```

```
{
    this(this)
    { }
}

// `__xpostblit` aliases the aggregated postblit so the `const` applies to it.
// However, the aggregated postblit calls the field postblit which does not have
// any qualifier applied, resulting in a qualifier mismatch error
const struct B
{
    S a;        // error : mutable method B.__fieldPostblit is not callable using a const object
    this(this)
    { }
}

// `__xpostblit` aliases the field postblit; no error
const struct B2
{
    S a;
}

// Similar to B
immutable struct C
{
    S a;        // error : mutable method C.__fieldPostblit is not callable using a immutable obje
    this(this)
    { }
}

// Similar to B2, compiles
immutable struct C2
{
    S a;
}
```

In the above situations the errors do not contain line numbers because the errors are regarding generated code.

Qualifying an entire struct as `shared` correctly propagates the attribute to the generated postblits:

```
shared struct A
```

```
{
    this(this)
    {
        import std.stdio : writeln;
        writeln("the shared postblit was called");
    }
}

struct B
{
    A a;
}

void main()
{
    shared B b1;
    auto b2 = b1;
}
```

Unions may have fields that have postblits. However, a union itself never has a postblit. Copying a union does not result in postblit calls for any fields. If those calls are desired, they must be inserted explicitly by the programmer:

```
struct S
{
    int count;
    this(this)
    {
        ++count;
    }
}

union U
{
    S s;
}

void main()
{
    U a = U.init;
    U b = a;
```

```
    assert(b.s.count == 0);
    b.s.__postblit;
    assert(b.s.count == 1);
}
```

14.20 Struct Destructors

Destructors are called when an object goes out of scope. Their purpose is to free up resources owned by the struct object.

Unions may have fields that have destructors. However, a union itself never has a destructor. When a union goes out of scope, destructors for it's fields are not called. If those calls are desired, they must be inserted explicitly by the programmer:

```
struct S
{
    ~this()
    {
        import std.stdio;
        writeln("S is being destructed");
    }
}

union U
{
    S s;
}

void main()
{
    import std.stdio;
    {
        writeln("entering first scope");
        U u = U.init;
        scope (exit) writeln("exiting first scope");
    }
    {
        writeln("entering second scope");
        U u = U.init;
        scope (exit)
        {
```

```
            writeln("exiting second scope");
            destroy(u.s);
        }
    }
}
```

14.21 Struct Invariants

```
StructInvariant:
    invariant ( ) BlockStatement
    invariant BlockStatement
    invariant ( AssertArguments ) ;
```

*StructInvariant*s specify the relationships among the members of a struct instance. Those relationships must hold for any interactions with the instance from its public interface.

The invariant is in the form of a `const` member function. The invariant is defined to *hold* if all the *AssertExpression*s within the invariant that are executed succeed.

If the invariant does not hold, then the program enters an invalid state.

Any invariants for fields are applied before the struct invariant.

There may be multiple invariants in a struct. They are applied in lexical order.

*StructInvariant*s must hold at the exit of the struct constructor (if any), and at the entry of the struct destructor (if any).

*StructInvariant*s must hold at the entry and exit of all public or exported non-static member functions. The order of application of invariants is:

1. preconditions
2. invariant
3. function body
4. invariant
5. postconditions

The invariant need not hold if the struct instance is implicitly constructed using the default `.init` value.

```
struct Date
{
    this(int d, int h)
    {
        day = d;    // days are 1..31
```

```
        hour = h;   // hours are 0..23
    }

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

  private:
    int day;
    int hour;
}
```

Public or exported non-static member functions cannot be called from within an invariant.

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

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

Undefined Behavior: happens if the invariant does not hold and execution continues.

Implementation Defined:

1. Whether the *StructInvariant* is executed at runtime or not. This is typically controlled with a compiler switch.
2. The behavior when the invariant does not hold is typically the same as for when *AssertExpressions* fail.

Best Practices:

1. Do not indirectly call exported or public member functions within a struct invariant, as this can result in infinite recursion.
2. Avoid reliance on side effects in the invariant. as the invariant may or may not be executed.
3. Avoid having mutable public fields of structs with invariants, as then the invariant cannot verify the public interface.

14.22 Identity Assignment Overload

While copy construction takes care of initializing an object from another object of the same type, assignment is defined as copying the contents of a source object over those of a destination object, calling the destination object's destructor if it has one in the process:

```
struct S { ... }  // S has postblit or destructor
S s;      // default construction of s
S t = s;  // t is copy-constructed from s
t = s;    // t is assigned from s
```

Struct assignment `t=s` is defined to be semantically equivalent to:

```
t.opAssign(s);
```

where `opAssign` is a member function of S:

```
ref S opAssign(ref S s)
{
    S tmp = this;   // bitcopy this into tmp
    this = s;       // bitcopy s into this
    tmp.__dtor();   // call destructor on tmp
    return this;
}
```

An identity assignment overload is required for a struct if one or more of these conditions hold:

- it has a destructor
- it has a postblit
- it has a field with an identity assignment overload

If an identity assignment overload is required and does not exist, an identity assignment overload function of the type `ref S opAssign(ref S)` will be automatically generated.

A user-defined one can implement the equivalent semantics, but can be more efficient.

One reason a custom `opAssign` might be more efficient is if the struct has a reference to a local buffer:

```
struct S
{
    int[] buf;
    int a;

    ref S opAssign(ref const S s) return
    {
```

```
        a = s.a;
        return this;
    }

    this(this)
    {
        buf = buf.dup;
    }
}
```

Here, `S` has a temporary workspace `buf[]`. The normal postblit will pointlessly free and reallocate it. The custom `opAssign` will reuse the existing storage.

14.23 Nested Structs

A *nested struct* is a struct that is declared inside the scope of a function or a templated struct that has aliases to local functions as a template argument. Nested structs have member functions. It has access to the context of its enclosing scope (via an added hidden field).

```
void foo()
{
    int i = 7;
    struct SS
    {
        int x,y;
        int bar() { return x + i + 1; }
    }
    SS s;
    s.x = 3;
    s.bar(); // returns 11
}
```

A struct can be prevented from being nested by using the static attribute, but then of course it will not be able to access variables from its enclosing scope.

```
void foo()
{
    int i = 7;
    static struct SS
    {
        int x, y;
        int bar()
```

```
        {
            return i; // error, SS is not a nested struct
        }
    }
}
```

14.24 Unions and Special Member Functions

Unions may not have postblits, destructors, or invariants.

Chapter 15

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 ;
    class Identifier BaseClassList opt AggregateBody
    ClassTemplateDeclaration
```

```
BaseClassList:
    : SuperClass
    : SuperClass , Interfaces
    : Interfaces
```

```
SuperClass:
    BasicType
```

Interfaces:
    Interface
    Interface , *Interfaces*

Interface:
    BasicType

Classes consist of:

- a super class
- interfaces
- a nested anonymous metaclass for classes declared with `extern (Objective-C)`
- dynamic fields
- static fields
- types
- an optional synchronized attribute
- member functions
  - static member functions
  - Virtual Functions
  - Constructors
  - Destructors
  - Static Constructors
  - Static Destructors
  - *SharedStaticConstructor*s
  - *SharedStaticDestructor*s
  - Class Invariants
  - Unit Tests
  - Class Allocators
  - Class Deallocators
  - Alias This

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;
```

15.1 Access Control

Access to class members is controlled using visibility attributes. The default visibility attribute is `public`.

15.2 Fields

Class members are always accessed with the . operator.

Members of a base class can be accessed by prepending the name of the base class followed by a dot:

```
class A { int a; int a2;}
class B : A { int a; }

void foo(B b)
{
    b.a = 3;   // accesses field B.a
    b.a2 = 4;  // accesses field A.a2
    b.A.a = 5; // accesses field A.a
}
```

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.

Fields of `extern(Objective-C)` classes have a dynamic offset. That means that the base class can change (add or remove instance variables) without the subclasses needing to recompile or relink.

15.3 Field Properties

The `.offsetof` property gives the offset in bytes of the field from the beginning of the class instantiation. `.offsetof` is not available for fields of `extern(Objective-C)` classes due to their fields having a dynamic offset. `.offsetof` can only be applied to expressions which produce the type of the field itself, not the class type:

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

    o = Foo.x.offsetof; // error, Foo.x needs a 'this' reference
    o = foo.x.offsetof; // ok
}
```

15.4 Class Properties

The `.tupleof` property is an expression sequence of all the fields in the class, excluding the hidden fields and the fields in the base class. `.tupleof` is not available for `extern(Objective-C)` classes due to their fields having a dynamic offset.

```
class Foo { int x; long y; }

void test(Foo foo)
{
    import std.stdio;
    static assert(typeof(foo.tupleof).stringof == `(int, long)`);

    foo.tupleof[0] = 1; // set foo.x to 1
    foo.tupleof[1] = 2; // set foo.y to 2
    foreach (x; foo.tupleof)
        write(x);       // prints 12
}
```

The properties `.__vptr` and `.__monitor` give access to the class object's vtbl[] and monitor, respectively, but should not be used in user code.

15.5 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.

15.6 Member Functions

Non-static member functions or static member functions with `Objective-C` linkage have an extra hidden parameter called *this* through which the class object's other members can be accessed.

Member functions with the Objective-C linkage has an additional hidden, anonymous, parameter which is the selector the function was called with.

Non-static member functions can have, in addition to the usual *FunctionAttributes*, the attributes `const`, `immutable`, `shared`, or `inout`. These attributes apply to the hidden *this* parameter.

Static member functions with the Objective-C linkage are placed in the hidden nested metaclass as non-static member functions.

```
class C
{
    int a;
    const void foo()
    {
        a = 3; // error, 'this' is const
    }
    void foo() immutable
    {
        a = 3; // error, 'this' is immutable
    }
}
```

15.7 Synchronized Classes

All member functions of synchronized classes are synchronized. A static member function is synchronized on the *classinfo* object for the class, which means that one monitor is used for all static member functions for that synchronized class. For non-static functions of a synchronized class, the monitor used is part of the class object. For example:

```
synchronized class Foo
{
    void bar() { ...statements... }
}
```

is equivalent to (as far as the monitors go):

```
synchronized class Foo
{
    void bar()
    {
        synchronized (this) { ...statements... }
    }
}
```

Member functions of non-synchronized classes can be individually marked as synchronized.

```
class Foo
{
    synchronized void foo() { }  // foo is synchronized
}
```

Member fields of a synchronized class cannot be public:

```
synchronized class Foo
{
    int foo;  // disallowed: public field
}

synchronized class Bar
{
    private int bar;  // ok
}
```

The `synchronized` attribute can only be applied to classes, structs cannot be marked to be synchronized.

15.8 Constructors

Constructor:
  `this` *Parameters* *MemberFunctionAttributes*$_{\text{opt}}$ `;`
  `this` *Parameters* *MemberFunctionAttributes*$_{\text{opt}}$ *FunctionBody*
  ConstructorTemplate

Fields are by default initialized to the default initializer for their type (usually 0 for integer types and NAN for floating point types). If the field declaration has an optional *Initializer* that will 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
}
```

The *Initializer* is evaluated at compile time.

This 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 call other constructors for the same class in order to share common initializations (this is called a delegating constructor):

```
class C
{
    int j;
    this()
```

```
    {
        ...
    }
    this(int i)
    {
        this(); // delegating constructor call
        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 is implicitly generated with the form:

```
this() { }
```

The following restrictions apply to class construction:

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

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

   Implementation Defined: The compiler is not required to detect cyclic constructor calls.
   Undefined Behavior: If the program executes with cyclic constructor calls.
2. If a constructor's code contains a delegate constructor call, all possible execution paths through the constructor must make exactly one delegate 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 delegate constructor call.
4. Delegate constructor calls cannot appear after labels.

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

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

The following steps happen:

1. Storage is allocated for the object. If this fails, rather than return `null`, an OutOfMemoryError 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. 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.
3. If there is a constructor defined for the class, the constructor matching the argument list is called.
4. If a delegating constructor is not called, a call to the base class's default constructor is issued.
5. The body of the constructor is executed.
6. If class invariant checking is turned on, the class invariant is called at the end of the constructor.

Constructors can have one of these member function attributes: `const`, `immutable`, and `shared`. Construction of qualified objects will then be restricted to the implemented qualified constructors.

```
class C
{
    this();   // non-shared mutable constructor
}

// create mutable object
C m = new C();

// create const object using by mutable constructor
const C c2 = new const C();

// a mutable constructor cannot create an immutable object
// immutable C i = new immutable C();

// a mutable constructor cannot create a shared object
// shared C s = new shared C();
```

Constructors can be overloaded with different attributes.

```
class C
{
    this();               // non-shared mutable constructor
    this() shared;        // shared mutable constructor
    this() immutable;     // immutable constructor
}

C m = new C();
shared s = new shared C();
immutable i = new immutable C();
```

If the constructor can create a unique object (e.g. if it is `pure`), the object can be implicitly convertible to any qualifiers.

```
class C
{
    this() pure;
    // Based on the definition, this creates a mutable object. But the
    // created object cannot contain any mutable global data.
    // Therefore the created object is unique.

    this(int[] arr) immutable pure;
    // Based on the definition, this creates an immutable object. But
    // the argument int[] never appears in the created object so it
    // isn't implicitly convertible to immutable. Also, it cannot store
    // any immutable global data.
    // Therefore the created object is unique.
}

immutable i = new immutable C();           // this() pure is called
shared s = new shared C();                 // this() pure is called
C m = new C([1,2,3]);       // this(int[]) immutable pure is called
```

15.9 Field initialization inside constructor

In a constructor body, the first instance of field assignment is its initialization.

```
class C
{
    int num;
    this()
```

```
    {
        num = 1;  // initialization
        num = 2;  // assignment
    }
}
```

If the field type has an `opAssign` method, it will not be used for initialization.

```
struct A
{
    this(int n) {}
    void opAssign(A rhs) {}
}
class C
{
    A val;
    this()
    {
        val = A(1);  // val is initialized to the value of A(1)
        val = A(2);  // rewritten to val.opAssign(A(2))
    }
}
```

If the field type is not mutable, multiple initialization will be rejected.

```
class C
{
    immutable int num;
    this()
    {
        num = 1;  // OK
        num = 2;  // Error: multiple field initialization
    }
}
```

If the field is initialized on one path, it must be initialized on all paths.

```
class C
{
    immutable int num;
    immutable int ber;
    this(int i)
    {
```

```
        if (i)
            num = 3;   // initialization
        else
            num = 4;   // initialization
    }
    this(long j)
    {
        j ? (num = 3) : (num = 4); // ok
        j || (ber = 3);  // error, intialized on only one path
        j && (ber = 3);  // error, intialized on only one path
    }
}
```

A field initialization may not appear in a loop or after a label.

```
class C
{
    immutable int num;
    immutable string str;
    this()
    {
        foreach (i; 0..2)
        {
            num = 1;    // Error: field initialization not allowed in loops
        }
        size_t i = 0;
    Label:
        str = "hello";  // Error: field initialization not allowed after labels
        if (i++ < 2)
            goto Label;
    }
}
```

If a field's type has disabled default construction, then it must be initialized in the constructor.

```
struct S { int y; @disable this(); }

class C
{
    S s;
    this(S t) { s = t; }    // ok
    this(int i) { this(); } // ok
```

```
    this() { }              // error, s not initialized
}
```

15.10 Destructors

Destructor:
    ~ this ( ) *MemberFunctionAttributes*$_{\text{opt}}$;
    ~ this ( ) *MemberFunctionAttributes*$_{\text{opt}}$ *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 `destroy`, and then the garbage collector calls the destructor immediately. 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.

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 unreferenced objects is not specified. This means that when the garbage collector calls a destructor for an object of a class that has members which are references to garbage collected objects, those references may no longer be valid. This means that destructors cannot reference sub objects. This rule does not apply to auto objects or objects destructed with `destroy`, as the destructor is not being run by the garbage collector, meaning all references are valid.

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

15.11 Static Constructors

StaticConstructor:
    static this ( ) *MemberFunctionAttributes*$_{\text{opt}}$;

`static this ( )` *MemberFunctionAttributes*$_{\text{opt}}$ *FunctionBody*

A static constructor is a function that performs initializations of thread local data before the `main()` function gets control for the main thread, and upon thread startup.

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
    }
}
```

If `main()` or the thread returns normally, (does not throw an exception), the static destructor is added to the list of functions to be called on thread 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
}
```

15.12 Static Destructors

StaticDestructor:
    `static ~ this ( )` *MemberFunctionAttributes*$_{\text{opt}}$ `;`
    `static ~ this ( )` *MemberFunctionAttributes*$_{\text{opt}}$ *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 thread 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
}
```

15.13 Shared Static Constructors

```
SharedStaticConstructor:
    shared static this ( ) MemberFunctionAttributes(opt) ;
    shared static this ( ) MemberFunctionAttributes(opt) FunctionBody
```

Shared static constructors are executed before any *StaticConstructor*s, and are intended for initializing any shared global data.

15.14 Shared Static Destructors

```
SharedStaticDestructor:
    shared static ~ this ( ) MemberFunctionAttributes(opt) ;
    shared static ~ this ( ) MemberFunctionAttributes(opt) FunctionBody
```

Shared static destructors are executed at program termination in the reverse order that *SharedStaticConstructor*s were executed.

15.15 Class Invariants

```
ClassInvariant:
    invariant ( ) BlockStatement
    invariant BlockStatement
    invariant ( AssertArguments ) ;
```

ClassInvariant specify the relationships among the members of a class instance. Those relationships must hold for any interactions with the instance from its public interface.

The invariant is in the form of a `const` member function. The invariant is defined to *hold* if all the *AssertExpressions* within the invariant that are executed succeed.

If the invariant does not hold, then the program enters an invalid state.

Any class invariants for base classes are applied before the class invariant for the derived class.

There may be multiple invariants in a class. They are applied in lexical order.

ClassInvariants must hold at the exit of the class constructor (if any), at the entry of the class destructor (if any).

ClassInvariants must hold at the entry and exit of all public or exported non-static member functions. The order of application of invariants is:

1. preconditions
2. invariant
3. function body
4. invariant
5. postconditions

The invariant need not hold if the class instance is implicitly constructed using the default `.init` value.

```
class Date
{
    this(int d, int h)
    {
        day = d;    // days are 1..31
        hour = h;   // hours are 0..23
    }

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

  private:
    int day;
    int hour;
}
```

Public or exported non-static member functions cannot be called from within an invariant.

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

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

Undefined Behavior: happens if the invariant does not hold and execution continues.
Implementation Defined:

1. Whether the *Class Invariant* is executed at runtime or not. This is typically controlled with a compiler switch.
2. The behavior when the invariant does not hold is typically the same as for when *AssertExpressions* fail.

Best Practices:

1. Do not indirectly call exported or public member functions within a class invariant, as this can result in infinite recursion.
2. Avoid reliance on side effects in the invariant. as the invariant may or may not be executed.
3. Avoid having mutable public fields of classes with invariants, as then the invariant cannot verify the public interface.

15.16 Class Allocators

Note: Class allocators are deprecated in D2.

```
Allocator:
    new Parameters ;
    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(string 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.

15.17 Class Deallocators

Note: Class deallocators and the delete operator are deprecated in D2. Use the `destroy` function to finalize an object by calling its destructor. The memory of the object is **not** immediately deallocated, instead the GC will collect the memory of the object at an undetermined point after finalization:

```
class Foo { int x; this() { x = 1; } }
Foo foo = new Foo;
destroy(foo);
assert(foo.x == int.init);  // object is still accessible
```

```
Deallocator:
    delete Parameters ;
    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.

15.18 Alias This

```
AliasThis:
    alias Identifier this ;
```

An *AliasThis* declaration names a member to subtype. The *Identifier* names that member.

A class or struct can be implicitly converted to the *AliasThis* member.

```
struct S
{
    int x;
    alias x this;
}

int foo(int i) { return i * 2; }

void test()
{
```

```
    S s;
    s.x = 7;
    int i = -s;  // i == -7
    i = s + 8;   // i == 15
    i = s + s;   // i == 14
    i = 9 + s;   // i == 16
    i = foo(s);  // implicit conversion to int
}
```

If the member is a class or struct, undefined lookups will be forwarded to the *AliasThis* member.

```
struct Foo
{
    int baz = 4;
    int get() { return 7; }
}

class Bar
{
    Foo foo;
    alias foo this;
}

void test()
{
    auto bar = new Bar;
    int i = bar.baz; // i == 4
    i = bar.get(); // i == 7
}
```

If the *Identifier* refers to a property member function with no parameters, conversions and undefined lookups are forwarded to the return value of the function.

```
struct S
{
    int x;
    @property int get()
    {
        return x * 2;
    }
    alias get this;
}
```

```
void test()
{
    S s;
    s.x = 2;
    int i = s; // i == 4
}
```

If an aggregate declaration defines an `opCmp` or `opEquals` method, it will take precedence to that of the aliased this member. Note that, unlike an `opCmp` method, an `opEquals` method is implicitly defined for a `struct` declaration if a user defined one isn't provided; this means that if the aliased this member `opEquals` is preferred it should be explicitly defined:

```
struct S
{
    int a;
    bool opEquals(S rhs) const
    {
        return this.a == rhs.a;
    }
}

struct T
{
    int b;
    S s;
    alias s this;
}

struct U
{
    int a;
    bool opCmp(U rhs) const
    {
        return this.a < rhs.a;
    }
}

struct V
{
    int b;
```

```
    U u;
    alias u this;
}

void main()
{
    S s1, s2;
    T t1, t2;
    U u1, u2;
    V v1, v2;

    assert(s1 == s2);       // calls S.opEquals
    assert(t1 == t2);       // calls compiler generated T.opEquals that implements member-wise
    assert(!(u1 < u2));     // calls U.opCmp
    assert(!(v1 < v2));     // calls U.opCmp because V does not define an opCmp method
                            // so the alias this of v1 is employed; U.opCmp expects a
                            // paramter of type U, so alias this of v2 is used

    assert(s1 == t1);       // calls s1.opEquals(t1.s);
    assert(t1 == s1);       // calls t1.s.opEquals(s1);
    assert(!(u1 < v1));     // calls u1.opCmp(v1.u);
    assert(!(v1 < u1));     // calls v1.u.opCmp(v1);
}
```

*Attribute*s are ignored for `AliasThis`.

Multiple *AliasThis* are allowed. For implicit conversions and forwarded lookups, all *AliasThis* declarations are attempted; if more than one *AliasThis* is eligible, the ambiguity is disallowed by raising an error. Note: Multiple *AliasThis* is currently unimplemented.

15.19 Scope Classes

Note: Scope classes have been recommended for deprecation.

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

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

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

A 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 a 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.

15.20 Final Classes

Final classes cannot be subclassed:

```
final class A { }
class B : A { }  // error, class A is final
```

Methods of a final class are `final` by default.

15.21 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
}
```

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 there is no enclosing class context, `.outer` would return a pointer to enclosing function frame with `void*`.

```
class Outer
{
    class Inner1
    {
        Outer getOuter()
        {
            return this.outer;
        }
    }

    void foo()
    {
        Inner1 i = new Inner1;
        assert(i.getOuter() is this);
    }

    void bar()
    {
        // x is referenced from nested scope, so
        // bar makes a closure envronment.
        int x = 1;

        class Inner2
        {
            Outer getOuter()
            {
                x = 2;
                // The Inner2 instance owns function frame of bar
                // as static frame pointer, but .outer yet returns
                // the enclosing Outer class instance property.
```

```
                return this.outer;
            }
        }

        Inner2 i = new Inner2;
        assert(i.getOuter() is this);
    }

    static void baz()
    {
        // make a closure envronment
        int x = 1;

        class Inner3
        {
            void* getOuter()
            {
                x = 2;
                // There's no accessible enclosing class instance, so
                // .outer property returns the function frame of bar.
                return this.outer;
            }
        }

        Inner3 i = new Inner3;
        assert(i.getOuter() !is null);
    }
}
```

15.22 Anonymous Nested Classes

An anonymous nested class is both defined and instantiated with a *NewAnonClassExpression*:

NewAnonClassExpression:
  `new` *AllocatorArguments*$_{\text{opt}}$ `class` *ClassArguments*$_{\text{opt}}$ *SuperClass*$_{\text{opt}}$ *Interfaces*$_{\text{opt}}$ *AggregateBody*

ClassArguments:
  `(` *ArgumentList*$_{\text{opt}}$ `)`

which is equivalent to:

```
class Identifier : SuperClass Interfaces AggregateBody
new (ArgumentList) Identifier (ArgumentList);
```

where *Identifier* is the name generated for the anonymous nested class.

15.23 Const, Immutable and Shared Classes

If a *ClassDeclaration* has a `const`, `immutable` or `shared` storage class, then it is as if each member of the class was declared with that storage class. If a base class is const, immutable or shared, then all classes derived from it are also const, immutable or shared.

Chapter 16

Interfaces

```
InterfaceDeclaration:
    interface Identifier ;
    interface Identifier BaseInterfaceList opt AggregateBody
    InterfaceTemplateDeclaration
```

```
BaseInterfaceList:
    : Interfaces
```

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.

Some operating system objects, like COM/OLE/ActiveX for Win32, have specialized interfaces. D interfaces that are compatible with COM/OLE/ActiveX are called *COM Interfaces*.

C++ Interfaces are another form of interfaces, meant to be binary compatible with C++.

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
```

Virtual interface member functions do not have implementations. Interfaces are expected to implement static or final functions.

```
interface D
{
    void bar() { }  // error, implementation not allowed
    static void foo() { } // ok
    final void abc() { } // ok
}
```

Interfaces can have function templates in the members. All instantiated functions are implicitly `final`.

```
interface D
{
    void foo(T)() { }  // ok, it's implicitly final
}
```

Classes that inherit from an interface may not override final or static interface member functions.

```
interface D
{
    void bar();
    static void foo() { }
    final void abc() { }
}

class C : D
{
    void bar() { } // ok
    void foo() { } // error, cannot override static D.foo()
    void abc() { } // error, cannot override final D.abc()
}
```

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
```

16.1 Interfaces with Contracts

Interface member functions can have contracts even though there is no body for the function. The contracts are inherited by any class member function that implements that interface member function.

```
interface I
{
    int foo(int i)
    in { assert(i > 7); }
    out (result) { assert(result & 1); }

    void bar();
}
```

16.2 Const and Immutable Interfaces

If an interface has `const` or `immutable` storage class, then all members of the interface are `const` or `immutable`. This storage class is not inherited.

16.3 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 `core.stdc.windows.com.IUnknown`. A COM interface differs from a regular D interface in that:

- It derives from the interface `core.stdc.windows.com.IUnknown`.
- It cannot be the argument to `destroy`.
- 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.
- Classes derived from COM interfaces are COM classes.
- The default linkage for member functions of COM classes is `extern(System)`.
  Note that if you want to implement or override any base-class methods of D interfaces or classes (ones which do not inherit from `IUnknown`), you have to explicitly mark them as having the `extern(D)` linkage:

```
import core.sys.windows.windows;
import core.stdc.windows.com;

interface IText
{
    void write();
}

abstract class Printer : IText
{
    void print() { }
}

class C : Printer, IUnknown
{
    // Implements the IText `write` class method.
    extern(D) void write() { }

    // Overrides the Printer `print` class method.
    extern(D) override void print() { }

    // Overrides the Object base class `toString` method.
    extern(D) override string toString() { return "Class C"; }

    // Methods of class implementing the IUnknown interface have
    // the extern(System) calling convention by default.
    HRESULT QueryInterface(const(IID)*, void**);
    uint AddRef();
    uint Release();
}
```

The same applies to other `Object` methods such as `opCmp`, `toHash`, etc.

- The first member of the `vtbl[]` is not the pointer to the InterfaceInfo, but the first virtual function pointer.

For more information, see Modern COM Programming in D

16.4 C++ Interfaces

C++ interfaces are interfaces declared with C++ linkage:

```
extern (C++) interface Ifoo
{
    void foo();
    void bar();
}
```

which is meant to correspond with the following C++ declaration:

```
class Ifoo
{
    virtual void foo();
    virtual void bar();
};
```

Any interface that derives from a C++ interface is also a C++ interface. A C++ interface differs from a D interface in that:

- It cannot be the argument to `destroy`.
- References cannot be upcast to the enclosing class object, nor can they be downcast to a derived interface.
- The C++ calling convention is the default convention for its member functions, rather than the D calling convention.
- The first member of the `vtbl[]` is not the pointer to the `Interface`, but the first virtual function pointer.

Chapter 17

Enums

```
EnumDeclaration:
   enum Identifier EnumBody
   enum Identifier : EnumBaseType EnumBody
   AnonymousEnumDeclaration
```

```
EnumBaseType:
   Type
```

```
EnumBody:
   { EnumMembers }
   ;
```

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

```
EnumMemberAttributes:
   EnumMemberAttribute
   EnumMemberAttribute EnumMemberAttributes
```

EnumMemberAttribute:
  DeprecatedAttribute
  UserDefinedAttribute
  `@`*disable*

EnumMember:
  EnumMemberAttributes$_{\text{opt}}$ *Identifier*
  EnumMemberAttributes$_{\text{opt}}$ *Identifier* `=` *AssignExpression*

AnonymousEnumDeclaration:
  `enum` `:` *EnumBaseType* `{` *EnumMembers* `}`
  `enum` `{` *EnumMembers* `}`
  `enum` `{` *AnonymousEnumMembers* `}`

AnonymousEnumMembers:
  AnonymousEnumMember
  AnonymousEnumMember `,`
  AnonymousEnumMember `,` *AnonymousEnumMembers*

AnonymousEnumMember:
  EnumMember
  Type *Identifier* `=` *AssignExpression*

Enum declarations are used to define a group of constants.

17.1 Named Enums

Named enums are used to declare related constants and group them by giving them a unique type. The *EnumMembers* are declared in the scope of the named enum. The named enum declares a new type, and all the *EnumMembers* have that type.

This defines a new type `X` which has values `X.A=0`, `X.B=1`, `X.C=2`:

```
enum X { A, B, C }  // named enum
```

If the *EnumBaseType* is not explicitly set, and the first *EnumMember* has an *AssignExpression*, it is set to the type of that *AssignExpression*. Otherwise, it defaults to type `int`.

Named enum members may not have individual *Type*s.

A named enum member can be implicitly cast to its *EnumBaseType*, but *EnumBaseType* types cannot be implicitly cast to an enum type.

The value of an *EnumMember* is given by its *AssignExpression*. If there is no *AssignExpression* and it is the first *EnumMember*, its value is *EnumBaseType*`.init`.

If there is no *AssignExpression* and it is not the first *EnumMember*, it is given the value of the previous *EnumMember*`+1`. If the value of the previous *EnumMember* is *EnumBaseType*`.max`, it is an error. If the value of the previous *EnumMember*`+1` is the same as the value of the previous *EnumMember*, it is an error. (This can happen with floating point types.)

All *EnumMember*s are in scope for the *AssignExpression*s.

```
enum A = 3;
enum B
{
    A = A // error, circular reference
}
enum C
{
    A = B,  // A = 4
    B = D,  // B = 4
    C = 3,  // C = 3
    D       // D = 4
}
enum E : C
{
    E1 = C.D,
    E2      // error, C.D is C.max
}
```

An empty enum body (For example `enum E;`) signifies an opaque enum - the enum members are unknown.

Enum Default Initializer

The `.init` property of an enum type is the value of the first member of that enum. This is also the default initializer for the enum type.

```
enum X { A=3, B, C }
X x;      // x is initialized to 3
```

Enum Properties

Enum properties only exist for named enums.

<caption>Named Enum Properties</caption>

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

For example:

```
enum X { A=3, B=1, C=4, D, E=2 }
X.init   // is X.A
X.min    // is X.B
X.max    // is X.D
X.sizeof // is same as int.sizeof
```

The *EnumBaseType* of named enums must support comparison in order to compute the `.max` and `.min` properties.

17.2 Anonymous Enums

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* can have different types. Those types are given by the first of:

1. The *Type*, if present. Types are not permitted when an *EnumBaseType* is present.
2. The *EnumBaseType*, if present.
3. The type of the *AssignExpression*, if present.
4. The type of the previous *EnumMember*, if present.
5. `int`

```
enum { A, B, C }  // anonymous enum
```

Defines the constants A=0, B=1, C=2, all of type `int`.

Enums must have at least one member.

The value of an *EnumMember* is given by its *AssignExpression*. If there is no *AssignExpression* and it is the first *EnumMember*, its value is the `.init` property of the *EnumMember*'s type.

If there is no *AssignExpression* and it is not the first *EnumMember*, it is given the value of the previous *EnumMember*`+1`. If the value of the previous *EnumMember* is the `.max` property if the previous *EnumMember*'s type, it is an error. If the value of the previous *EnumMember*`+1` is the same as the value of the previous *EnumMember*, it is an error. (This can happen with floating point types.)

All *EnumMember*s are in scope for the *AssignExpression*s.

```
enum { A, B = 5+7, C, D = 8+C, E }
```

Sets A=0, B=12, C=13, D=21, and E=22, all of type `int`.

```
enum : long { A = 3, B }
```

Sets A=3, B=4 all of type `long`.

```
enum : string
{
    A = "hello",
    B = "betty",
    C     // error, cannot add 1 to "betty"
}

enum
{
    A = 1.2f,  // A is 1.2f of type float
    B,         // B is 2.2f of type float
    int C = 3, // C is 3 of type int
    D          // D is 4 of type int
}
```

17.3 Manifest Constants

If there is only one member of an anonymous enum, the `{ }` can be omitted. Gramatically speaking, this is an *AutoDeclaration*.

```
enum i = 4;      // i is 4 of type int
enum long l = 3; // l is 3 of type long
```

Manifest constants are not lvalues, meaning their address cannot be taken. They exist only in the memory of the compiler.

```
enum size = __traits(classInstanceSize, Foo);  // evaluated at compile-time
```

The initializer for a manifest constant is evaluated using compile time function evaluation.

```
template Foo(T)
{
    // Not bad, but the 'size' variable will be located in the executable.
    const size_t size = T.sizeof;       // evaluated at compile-time

    // ... use of 'size' at compile time ...
```

```
}

template Bar(T)
{
    // Better, the manifest constant has no runtime location in the executable.
    enum size_t size = T.sizeof;        // evaluated at compile-time

    // ... use of 'size' at compile time ...

    // If you take the address of Foo!T.size, it would also go into exe file.
    auto p = &Foo!T.size;
}
```

Chapter 18

Type Qualifiers

Type qualifiers modify a type by applying a *TypeCtor*. *TypeCtor*s are: `const`, `immutable`, `shared`, and `inout`. Each applies transitively to all subtypes.

18.1 Const and Immutable

When examining a data structure or interface, it is very helpful to be able to easily tell which data can be expected to not change, which data might change, and who may change that data. This is done with the aid of the language typing system. Data can be marked as const or immutable, with the default being changeable (or *mutable*).

`immutable` applies to data that cannot change. Immutable data values, once constructed, remain the same for the duration of the program's execution. Immutable data can be placed in ROM (Read Only Memory) or in memory pages marked by the hardware as read only. Since immutable data does not change, it enables many opportunities for program optimization, and has applications in functional style programming.

`const` applies to data that cannot be changed by the const reference to that data. It may, however, be changed by another reference to that same data. Const finds applications in passing data through interfaces that promise not to modify them.

Both immutable and const are *transitive*, which means that any data reachable through an immutable reference is also immutable, and likewise for const.

18.2 Immutable Storage Class

The simplest immutable declarations use it as a storage class. It can be used to declare manifest constants.

```
immutable int x = 3;  // x is set to 3
```

```
x = 4;        // error, x is immutable
char[x] s;    // s is an array of 3 chars
```

The type can be inferred from the initializer:

```
immutable y = 4; // y is of type int
y = 5;           // error, y is immutable
```

If the initializer is not present, the immutable can be initialized from the corresponding constructor:

```
immutable int z;
void test()
{
    z = 3; // error, z is immutable
}
static this()
{
    z = 3; // ok, can set immutable that doesn't
           // have static initializer
}
```

The initializer for a non-local immutable declaration must be evaluatable at compile time:

```
int foo(int f) { return f * 3; }
int i = 5;
immutable x = 3 * 4;      // ok, 12
immutable y = i + 1;      // error, cannot evaluate at compile time
immutable z = foo(2) + 1; // ok, foo(2) can be evaluated at compile time, 7
```

The initializer for a non-static local immutable declaration is evaluated at run time:

```
int foo(int f)
{
    immutable x = f + 1;  // evaluated at run time
    x = 3;                // error, x is immutable
}
```

Because immutable is transitive, data referred to by an immutable is also immutable:

```
immutable char[] s = "foo";
s[0] = 'a';  // error, s refers to immutable data
s = "bar";   // error, s is immutable
```

Immutable declarations can appear as lvalues, i.e. they can have their address taken, and occupy storage.

18.3 Const Storage Class

A const declaration is exactly like an immutable declaration, with the following differences:

- Any data referenced by the const declaration cannot be changed from the const declaration, but it might be changed by other references to the same data.
- The type of a const declaration is itself const.

18.4 Immutable Type

Data that will never change its value can be typed as immutable. The immutable keyword can be used as a *type qualifier*:

```
immutable(char)[] s = "hello";
```

The immutable applies to the type within the following parentheses. So, while `s` can be assigned new values, the contents of `s[]` cannot be:

```
s[0] = 'b';  // error, s[] is immutable
s = null;    // ok, s itself is not immutable
```

Immutability is transitive, meaning it applies to anything that can be referenced from the immutable type:

```
immutable(char*)** p = ...;
p = ...;        // ok, p is not immutable
*p = ...;       // ok, *p is not immutable
**p = ...;      // error, **p is immutable
***p = ...;     // error, ***p is immutable
```

Immutable used as a storage class is equivalent to using immutable as a type qualifier for the entire type of a declaration:

```
immutable int x = 3;   // x is typed as immutable(int)
immutable(int) y = 3;  // y is immutable
```

18.5 Creating Immutable Data

The first way is to use a literal that is already immutable, such as string literals. String literals are always immutable.

```
auto s = "hello";   // s is immutable(char)[5]
char[] p = "world"; // error, cannot implicitly convert immutable
                    // to mutable
```

The second way is to cast data to immutable. When doing so, it is up to the programmer to ensure that any mutable references to the same data are not used to modify the data after the cast.

```
char[] s = ['a'];
s[0] = 'b'; // ok
immutable(char)[] p = cast(immutable)s; // ok, if data is not mutated
                                        // through s anymore
s[0] = 'c'; // undefined behavior
immutable(char)[] q = cast(immutable)s.dup; // always ok, unique reference

char[][] s2 = [['a', 'b'], ['c', 'd']];
immutable(char[][]) p2 = cast(immutable)s2.dup; // dangerous, only the first
                                                // level of elements is unique
s2[0] = ['x', 'y']; // ok, doesn't affect p2
s2[1][0] = 'z'; // undefined behavior
immutable(char[][]) q2 = [s2[0].dup, s2[1].dup]; // always ok, unique references
```

The `.idup` property is a convenient way to create an immutable copy of an array:

```
auto p = s.idup;
p[0] = ...;       // error, p[] is immutable
```

18.6 Removing Immutable or Const with a Cast

An immutable or const type qualifier can be removed with a cast:

```
immutable int* p = ...;
int* q = cast(int*)p;
```

This does not mean, however, that one can change the data:

```
*q = 3; // allowed by compiler, but result is undefined behavior
```

The ability to cast away immutable-correctness is necessary in some cases where the static typing is incorrect and not fixable, such as when referencing code in a library one cannot change. Casting is, as always, a blunt and effective instrument, and when using it to cast away immutable-correctness, one must assume the responsibility to ensure the immutability of the data, as the compiler will no longer be able to statically do so.

Note that casting away a const qualifier and then mutating is undefined behavior, too, even when the referenced data is mutable. This is so that compilers and programmers can make assumptions based on const alone. For example, here it may be assumed that `f` does not alter `x`:

```
void f(const int* a);
void main()
```

```
{
    int x = 1;
    f(&x);
    assert(x == 1); // guaranteed to hold
}
```

18.7 Immutable Member Functions

Immutable member functions are guaranteed that the object and anything referred to by the `this` reference is immutable. They are declared as:

```
struct S
{
    int x;

    void foo() immutable
    {
        x = 4;      // error, x is immutable
        this.x = 4; // error, x is immutable
    }
}
```

Note that using immutable on the left hand side of a method does not apply to the return type:

```
struct S
{
    immutable int[] bar()  // bar is still immutable, return type is not!
    {
    }
}
```

To make the return type immutable, you need to surround the return type with parentheses:

```
struct S
{
    immutable(int[]) bar()  // bar is now mutable, return type is immutable.
    {
    }
}
```

To make both the return type and the method immutable, you can write:

```
struct S
```

```
{
    immutable(int[]) bar() immutable
    {
    }
}
```

18.8 Const Type

Const types are like immutable types, except that const forms a read-only *view* of data. Other aliases to that same data may change it at any time.

18.9 Const Member Functions

Const member functions are functions that are not allowed to change any part of the object through the member function's this reference.

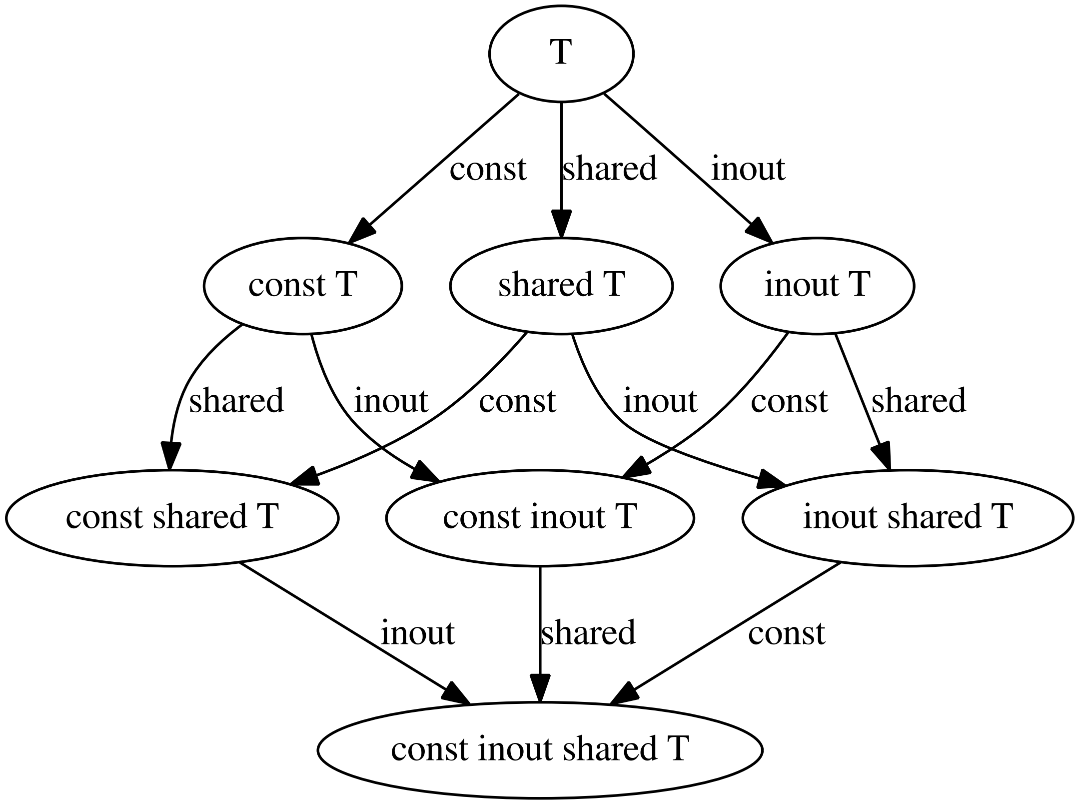
18.10 Combining Qualifiers

More than one qualifier may apply to a type. The order of application is irrelevant, for example given an unqualified type `T`, `const shared T` and `shared const T` are the same type. For that reason, this document depicts qualifier combinations without parentheses unless necessary and in alphabetic order.

Applying a qualifier to a type that already has that qualifier is legal but has no effect, e.g. given an unqualified type `T`, `shared(const shared T)` yields the type `const shared T`.

Applying the `immutable` qualifier to any type (qualified or not) results in `immutable T`. Applying any qualifier to `immutable T` results in `immutable T`. This makes `immutable` a fixed point of qualifier combinations and makes types such as `const(immutable(shared T))` impossible to create.

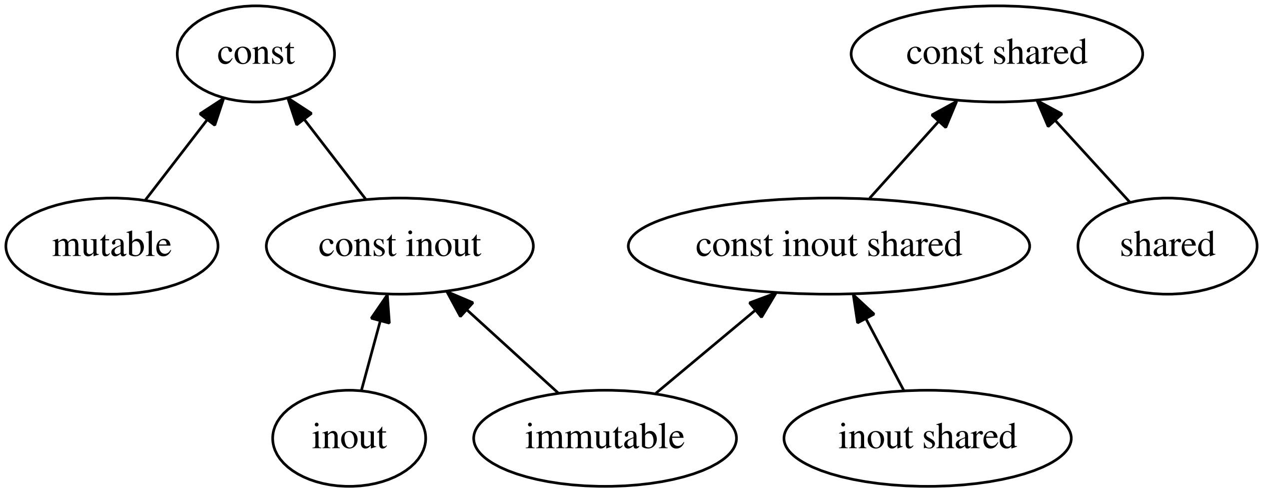
Assuming `T` is an unqualified type, the graph below illustrates how qualifiers combine (combinations with `immutable` are omitted). For each node, applying the qualifier labeling the edge leads to the resulting type.



18.11 Implicit Qualifier Conversions

Values that have no mutable indirections (including structs that don't contain any field with mutable indirections) can be implicitly converted across *mutable*, `const`, `immutable`, `const shared`, `inout` and `inout shared`.

References to qualified objects can be implicitly converted according to the following rules:



In the graph above, any directed path is a legal implicit conversion. No other qualifier combinations than the ones shown is valid. If a directed path exists between two sets of qualifiers, the types thus qualified are called qualifier-convertible. The same information is shown below in tabular format:

Implicit Conversion of Reference Types

from/to	*mutable*	`const`	`shared`	`const shared`	`inout`	`const inout`	`inout shared`	`const inout shared`	`immutable`
mutable	✓	✓							
`const`		✓							
`const inout`		✓				✓			
`const shared`				✓					
`const inout shared`				✓				✓	
`immutable`		✓		✓		✓		✓	✓
`inout`		✓			✓	✓			
`shared`			✓	✓					
`inout shared`				✓			✓	✓	

If an implicit conversion is disallowed by the table, an *Expression* may be converted if:

An expression may be converted from mutable or shared to immutable if the expression is unique and all expressions it transitively refers to are either unique or immutable.

An expression may be converted from mutable to shared if the expression is unique and all expressions it transitively refers to are either unique, immutable, or shared.

An expression may be converted from immutable to mutable if the expression is unique.

An expression may be converted from shared to mutable if the expression is unique.

A *Unique Expression* is one for which there are no other references to the value of the expression and all expressions it transitively refers to are either also unique or are immutable. For example:

```
void main()
{
    immutable int** p = new int*(null); // ok, unique

    int x;
    immutable int** q = new int*(&x); // error, there may be other references to x

    immutable int y;
    immutable int** r = new immutable(int)*(&y); // ok, y is immutable
}
```

Otherwise, a *CastExpression* can be used to force a conversion when an implicit version is disallowed, but this cannot be done in `@safe` code, and the correctness of it must be verified by the user.

Chapter 19

Functions

19.1 Grammar

Function declaration

FuncDeclaration:
  StorageClasses$_{\text{opt}}$ *BasicType* *FuncDeclarator* *FunctionBody*
  AutoFuncDeclaration

AutoFuncDeclaration:
  StorageClasses *Identifier* *FuncDeclaratorSuffix* *FunctionBody*

FuncDeclarator:
  BasicType2$_{\text{opt}}$ *Identifier* *FuncDeclaratorSuffix*

FuncDeclaratorSuffix:
  Parameters *MemberFunctionAttributes*$_{\text{opt}}$
  TemplateParameters *Parameters* *MemberFunctionAttributes*$_{\text{opt}}$ *Constraint*$_{\text{opt}}$

Parameters

Parameters:
  (*ParameterList*$_{\text{opt}}$)

ParameterList:
    Parameter
    Parameter `,` *ParameterList*
    `...`

Parameter:
    ParameterAttributes$_{\text{opt}}$ *BasicType* *Declarator*
    ParameterAttributes$_{\text{opt}}$ *BasicType* *Declarator* `...`
    ParameterAttributes$_{\text{opt}}$ *BasicType* *Declarator* `=` *AssignExpression*
    ParameterAttributes$_{\text{opt}}$ *Type*
    ParameterAttributes$_{\text{opt}}$ *Type* `...`

ParameterAttributes:
    InOut
    UserDefinedAttribute
    ParameterAttributes *InOut*
    ParameterAttributes *UserDefinedAttribute*
    ParameterAttributes

InOut:
    `auto`
    TypeCtor
    `final`
    `in`
    `lazy`
    `out`
    `ref`
    `return ref`
    `scope`

Function attributes

FunctionAttributes:
    FunctionAttribute
    FunctionAttribute *FunctionAttributes*

FunctionAttribute:
  `nothrow`
  `pure`
  Property

MemberFunctionAttributes:
  MemberFunctionAttribute
  MemberFunctionAttribute MemberFunctionAttributes

MemberFunctionAttribute:
  `const`
  `immutable`
  `inout`
  `return`
  `shared`
  FunctionAttribute

Function body

FunctionBody:
  SpecifiedFunctionBody
  MissingFunctionBody

FunctionLiteralBody:
  SpecifiedFunctionBody

SpecifiedFunctionBody:
  `do`$_{\text{opt}}$ *BlockStatement*
  FunctionContracts$_{\text{opt}}$ *InOutContractExpression* `do`$_{\text{opt}}$ *BlockStatement*
  FunctionContracts$_{\text{opt}}$ *InOutStatement* `do` *BlockStatement*

MissingFunctionBody:

```
    ;
    FunctionContracts_opt InOutContractExpression ;
    FunctionContracts_opt InOutStatement
```

Function contracts

```
FunctionContracts:
    FunctionContract
    FunctionContract FunctionContracts
```

```
FunctionContract:
    InOutContractExpression
    InOutStatement
```

```
InOutContractExpression:
    InContractExpression
    OutContractExpression
```

```
InOutStatement:
    InStatement
    OutStatement
```

```
InContractExpression:
    in ( AssertArguments )
```

```
OutContractExpression:
    out ( ; AssertArguments )
    out ( Identifier ; AssertArguments )
```

```
InStatement:
    in BlockStatement
```

```
OutStatement:
    out BlockStatement
    out ( Identifier ) BlockStatement
```

19.2 Contracts

The `in` and `out` blocks or expressions of a function declaration specify the pre- and post-conditions of the function. They are used in Contract Programming. The code inside these blocks should not have any side-effects, including modifying function parameters and/or return values.

19.3 Function Return Values

Function return values are considered to be rvalues. This means they cannot be passed by reference to other functions.

19.4 Functions Without Bodies

Functions without bodies:

```
int foo();
```

that are not declared as `abstract` are expected to have their implementations elsewhere, and that implementation will be provided at the link step. This enables an implementation of a function to be completely hidden from the user of it, and the implementation may be in another language such as C, assembler, etc.

19.5 Pure Functions

Pure functions are functions that cannot directly access global or static mutable state. `pure` guarantees that a pure function call won't access or modify any implicit state in the program.

Unlike other functional programming languages, D's `pure` functions allow modification of the caller state through their mutable parameters.

```
pure int foo(int[] arr) { arr[] += 1; return arr.length; }
int[] a = [1, 2, 3];
foo(a);
assert(a == [2, 3, 4]);
```

A `pure` function accepting parameters with mutable indirections offers what's called "weak purity" because it can change program state transitively through its arguments. A `pure` function that has no parameter with mutable indirections is called "strongly pure" and fulfills the purity definition in traditional functional languages. Weakly `pure` functions are useful as reusable building blocks for strongly pure functions.

To prevent mutation, D offers the `immutable` type qualifier. If all of a `pure` function's parameters are `immutable` or copied values without any indirections (e.g. `int`), the type system guarantees no side effects.

```
struct S { double x; }
pure int foo(immutable(int)[] arr, int num, S val)
{
    //arr[num] = 1; // compile error
    num = 2;        // has no side effect to the caller side
    val.x = 3.14;   // ditto
    return arr.length;
}
```

The maximum guarantee of `pure` is called "strong purity". It can enable optimizations based on the fact that a function is guaranteed to not mutate anything which isn't passed to it. For cases where the compiler can guarantee that a pure function cannot alter its arguments, it can enable full, functional purity (i.e. the guarantee that the function will always return the same result for the same arguments). To that end, a pure function:

- does not read or write any global or static mutable state
- cannot call functions that are not pure
- can override an impure function, but cannot be overridden by an impure function
- is covariant with an impure function
- cannot perform I/O

This definition of mutable functions is more general than the one traditionally employed by pure functional languages because it allows a D pure function to use state mutation, as long as all state is created internally or reachable through its arguments. In particular, a pure function may allocate memory by means of e.g. `new` or `malloc` without these being special cases. A pure function is allowed to loop indefinitely or terminate the program.

As a concession to practicality, a pure function can also:

- read and write the floating point exception flags
- read and write the floating point mode flags, as long as those flags are restored to their initial state upon function entry
- perform impure operations in statements that are in a *ConditionalStatement* controlled by a *DebugCondition*.

A pure function can throw exceptions.

```
import std.stdio;
int x;
immutable int y;
const int* pz;

pure int foo(int i,
             char* p,
             const char* q,
             immutable int* s)
{
    debug writeln("in␣foo()"); // ok, impure code allowed in debug statement
    x = i;   // error, modifying global state
    i = x;   // error, reading mutable global state
    i = y;   // ok, reading immutable global state
    i = *pz; // error, reading const global state
    return i;
}
```

An implementation may assume that a `pure` function that (a) accepts only parameters without mutable indirections, and (b) returns a result without mutable indirections, will have the same effect for all invocation with equivalent arguments, and is allowed to memoize the result of the function under the assumption that equivalent parameters always produce equivalent results. Such functions are termed *strongly `pure`* functions in this document. Note that a strongly pure function may still have behavior inconsistent with memoization by e.g. using `cast`s or by changing behavior depending on the address of its parameters. An implementation is currently not required to enforce validity of memoization in all cases.

A pure function that accepts only parameters without mutable indirections and returns a result that has mutable indirections is called a *pure factory function*. An implementation may assume that all mutable memory returned by the call is not referenced by any other part of the program, i.e. it is newly allocated by the function. Conversely, the mutable references of the result may be assumed to not refer to any object that existed before the function call. For example:

```
struct List { int payload; List* next; }
pure List* make(int a, int b)
{
    auto result = new List(a, null);
    result.next = new List(b, result);
    return result;
}
```

Here, an implementation may assume (without having knowledge of the body of `make`) that all references in `make`'s result refer to other `List` objects created by `make`, and that no other part of the program refers to any of these objects.

Any `pure` function that is not strongly pure cannot be assumed to be memoizable, and calls to it may not be elided even if it returns `void` (save for compiler optimizations that prove the function has no effect). Function calls may still be elided, or results be memoized, by means of traditional inlining and optimization techniques available for all functions.

If a strongly pure function throws an exception or an error, the assumptions related to memoization and references do not carry to the thrown exception.

Pure destructors do not benefit of special elision.

19.6 Nothrow Functions

Nothrow functions can only throw exceptions derived from class *Error*.

Nothrow functions are covariant with throwing ones.

19.7 Ref Functions

Ref functions allow functions to return by reference. This is analogous to ref function parameters.

```
ref int foo()
{
    auto p = new int;
    return *p;
}
...
foo() = 3;  // reference returns can be lvalues
```

19.8 Auto Functions

Auto functions have their return type inferred from any *ReturnStatement*s in the function body.

An auto function is declared without a return type. If it does not already have a storage class, use the auto storage class.

If there are multiple *ReturnStatement*s, the types of them must be implicitly convertible to a common type. If there are no *ReturnStatement*s, the return type is inferred to be void.

```
auto foo(int x) { return x + 3; }          // inferred to be int
auto bar(int x) { return x; return 2.5; }  // inferred to be double
```

19.9 Auto Ref Functions

Auto ref functions infer their return type just as auto functions do. In addition, they become ref functions if all return expressions are lvalues, and it would not be a reference to a local or a parameter.

```
auto ref f1(int x)     { return x; }  // value return
auto ref f2()          { return 3; }  // value return
auto ref f3(ref int x) { return x; }  // ref return
auto ref f4(out int x) { return x; }  // ref return
auto ref f5() { static int x; return x; }  // ref return
```

The ref-ness of a function is determined from all *ReturnStatement*s in the function body:

```
auto ref f1(ref int x) { return 3; return x; }  // ok, value return
auto ref f2(ref int x) { return x; return 3; }  // ok, value return
auto ref f3(ref int x, ref double y)
{
    return x; return y;
    // The return type is deduced to double, but cast(double)x is not an lvalue,
    // then become a value return.
}
```

Auto ref function can have explicit return type.

```
auto ref int (ref int x) { return x; }  // ok, ref return
auto ref int foo(double x) { return x; }   // error, cannot convert double to int
```

19.10 Inout Functions

Functions that deal with mutable, const, or immutable types with equanimity often need to transmit their type to the return value:

```
int[] f1(int[] a, int x, int y) { return a[x .. y]; }

const(int)[] f2(const(int)[] a, int x, int y) { return a[x .. y]; }

immutable(int)[] f3(immutable(int)[] a, int x, int y) { return a[x .. y]; }
```

The code generated by these three functions is identical. To indicate that these can be one function, the inout type constructor is employed:

```
inout(int)[] foo(inout(int)[] a, int x, int y) { return a[x .. y]; }
```

The inout forms a wildcard that stands in for any of mutable, const, immutable, inout, or inout const. When the function is called, the inout of the return type is changed to whatever the mutable, const, immutable, inout, or inout const status of the argument type to the parameter inout was.

Inout types can be implicitly converted to const or inout const, but to nothing else. Other types cannot be implicitly converted to inout. Casting to or from inout is not allowed in @safe functions.

A set of arguments to a function with inout parameters is considered a match if any inout argument types match exactly, or:

1. No argument types are composed of inout types.
2. A mutable, const or immutable argument type can be matched against each corresponding parameter inout type.

If such a match occurs, the inout is considered the common qualifier of the matched qualifiers. If more than two parameters exist, the common qualifier calculation is recursively applied.

Common qualifier of the two type qualifiers

	mutable	`const`	`immutable`	`inout`	`inout const`
mutable (= m)	m	c	c	c	c
`const` (= c)	c	c	c	c	c
`immutable` (= i)	c	c	i	wc	wc
`inout` (= w)	c	c	wc	w	wc
`inout const` (= wc)	c	c	wc	wc	wc

The inout in the return type is then rewritten to be the inout matched qualifiers:

```
int[] ma;
const(int)[] ca;
immutable(int)[] ia;

inout(int)[] foo(inout(int)[] a) { return a; }
void test1()
{
    // inout matches to mutable, so inout(int)[] is
    // rewritten to int[]
    int[] x = foo(ma);

    // inout matches to const, so inout(int)[] is
    // rewritten to const(int)[]
    const(int)[] y = foo(ca);

    // inout matches to immutable, so inout(int)[] is
```

```
    // rewritten to immutable(int)[]
    immutable(int)[] z = foo(ia);
}

inout(const(int))[] bar(inout(int)[] a) { return a; }
void test2()
{
    // inout matches to mutable, so inout(const(int))[] is
    // rewritten to const(int)[]
    const(int)[] x = bar(ma);

    // inout matches to const, so inout(const(int))[] is
    // rewritten to const(int)[]
    const(int)[] y = bar(ca);

    // inout matches to immutable, so inout(int)[] is
    // rewritten to immutable(int)[]
    immutable(int)[] z = bar(ia);
}
```

Note: Shared types are not overlooked. Shared types cannot be matched with inout.
Nested functions inside pure function are implicitly marked as pure.

```
pure int foo(int x, immutable int y)
{
    int bar()
    // implicitly marked as pure, to be "weak purity"
    // hidden context pointer is mutable
    {
        x = 10;     // can access states in enclosing scope
                    // through the mutable context pointer
        return x;
    }
    pragma(msg, typeof(&bar));  // int delegate() pure

    int baz() immutable
    // qualify hidden context pointer with immutable,
    // and has no other parameters, make "strong purity"
    {
        //return x; // error, cannot access mutable data
                    // through the immutable context pointer
```

```
        return y;   // ok
    }

    // can call pure nested functions
    return bar() + baz();
}
```

19.11 Optional Parentheses

If a function call passes no explicit argument, i.e. it would syntactically use `()`, then these parentheses may be omitted, similar to a getter invocation of a property function.

```
void foo() {}   // no arguments
void fun(int x = 10) { }
void bar(int[] arr) {}  // for UFCS

void main()
{
    foo();      // OK
    foo;        // also OK
    fun;        // OK

    int[] arr;
    arr.bar();  // OK
    arr.bar;    // also OK
}
```

Optional parentheses are not applied to delegates or function pointers.

```
void main()
{
    int function() fp;

    assert(fp == 6);    // Error, incompatible types int function() and int
    assert(*fp == 6);   // Error, incompatible types int() and int

    int delegate() dg;
    assert(dg == 6);    // Error, incompatible types int delegate() and int
}
```

If a function returns a delegate or function pointer, the parantheses are required if the returned value is to be called.

```
struct S {
    int function() callfp() { return &numfp; }
    int delegate() calldg() return { return &numdg; }
    int numdg() { return 6; }
}

int numfp() { return 6; }

void main()
{
    S s;
    int function() fp;

    fp = s.callfp;
    assert(fp() == 6);

    fp = s.callfp();
    assert(fp() == 6);

    int x = s.callfp()();
    assert(x == 6);

    int delegate() dg;

    dg = s.calldg;
    assert(dg() == 6);

    dg = s.calldg();
    assert(dg() == 6);

    int y = s.calldg()();
    assert(y == 6);
}
```

19.12 Property Functions

WARNING: The definition and usefulness of property functions is being reviewed, and the implementation is currently incomplete. Using property functions is not recommended until the definition is more certain and implementation more mature.

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

Simple getter and setter properties can be written using UFCS. These can be enhanced with the additon of the `@property` attribute to the function, which adds the following behaviors:

- `@property` functions cannot be overloaded with non-`@property` functions with the same name.
- `@property` functions can only have zero, one or two parameters.
- `@property` functions cannot have variadic parameters.
- For the expression `typeof(exp)` where `exp` is an `@property` function, the type is the return type of the function, rather than the type of the function.
- For the expression `__traits(compiles, exp)` where `exp` is an `@property` function, a further check is made to see if the function can be called.
- `@property` are mangled differently, meaning that `@property` must be consistently used across different compilation units.
- The ObjectiveC interface recognizes `@property` setter functions as special and modifies them accordingly.

A simple property would be:

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

    @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, have their address taken, etc.

The built in properties `.sizeof`, `.alignof`, and `.mangleof` may not be declared as fields or methods in structs, unions, classes or enums.

If a property function has no parameters, it works as a getter. If has exactly one parameter, it works as a setter.

19.13 Virtual Functions

Virtual functions are functions that are called indirectly through a function pointer table, called a vtbl[], rather than directly. All `public` and `protected` member functions which are non-static and are not templatized are virtual unless the compiler can determine that they will never be overridden (e.g. they are marked with `final` and do not override any functions in a base class), in which case, it will make them non-virtual. Static or `final` functions with `Objective-C` linkage are virtual as well. This results in fewer bugs caused by not declaring a function virtual and then overriding it anyway.

Member functions which are `private` or `package` are never virtual, and hence cannot be 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
{
    override int def() { ... }  // ok, overrides A.def
    override 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
{
    // overrides and is covariant with Foo.test()
    override B test() { return null; }
}
```

Virtual functions all have a hidden parameter called the *this* reference, which refers to the class object for which the function is called.

Functions with `Objective-C` linkage has an additional hidden, unnamed, parameter which is the selector it was called with.

To avoid dynamic binding on member function call, insert base class name before the member function name. For example:

```
class B
{
```

```
    int foo() { return 1; }
}
class C : B
{
    override int foo() { return 2; }

    void test()
    {
        assert(B.foo() == 1);  // translated to this.B.foo(), and
                               // calls B.foo statically.
        assert(C.foo() == 2);  // calls C.foo statically, even if
                               // the actual instance of 'this' is D.
    }
}
class D : C
{
    override int foo() { return 3; }
}
void main()
{
    auto d = new D();
    assert(d.foo() == 3);    // calls D.foo
    assert(d.B.foo() == 1);  // calls B.foo
    assert(d.C.foo() == 2);  // calls C.foo
    d.test();
}
```

Function Inheritance and Overriding

A function 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);  // issues runtime error (instead of calling 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 foo = A.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)
}
```

If such an *AliasDeclaration* is not used, the derived class's functions completely override all the functions of the same name in the base class, even if the types of the parameters in the base class functions are different. If, through implicit conversions to the base class, those other functions do get called, a compile-time error will be given:

```
class A
{
    void set(long i) { }
    void set(int i)  { }
}
class B : A
{
    void set(long i) { }
}

void foo(A a)
{
    int i;
    a.set(3);   // error, use of A.set(int) is hidden by B
                // use 'alias set = A.set;' to introduce base class overload set.
    assert(i == 1);
```

```
}

void main()
{
    foo(new B);
}
```

If an error occurs during the compilation of your program, the use of overloads and overrides needs to be reexamined in the relevant classes.

The compiler will not give an error if the hidden function is disjoint, as far as overloading is concerned, from all the other virtual functions is the inheritance hierarchy.

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
}
```

If a derived class overrides a base class member function with diferrent *FunctionAttributes*, the missing attributes will be automatically compensated by the compiler.

```
class B
{
    void foo() pure nothrow @safe {}
}
class D : B
{
    override void foo() {}
}
void main()
{
    auto d = new D();
    pragma(msg, typeof(&d.foo));
    // prints "void delegate() pure nothrow @safe" in compile time
}
```

It's not allowed to mark an overridden method with the attributes `@disable` or `deprecated`. To stop the compilation or to output the deprecation message, the compiler must be able to determine the target of the call, which can't be guaranteed when it is virtual.

```
class B
{
    void foo() {}
}

class D : B
{
    @disable override void foo() {}
}

void main()
{
    B b = new D;
    b.foo(); // compiles and calls the most derived even if disabled.
}
```

Static functions with `Objective-C` linkage are overridable.

19.14 Inline Functions

The compiler makes the decision whether to inline a function or not. This decision may be controlled by `pragma(inline)`, assuming that the compiler implements it, which is not mandatory.

Note that any *FunctionLiteral* should be inlined when used in its declaration scope.

19.15 Function Overloading

Functions are overloaded based on how well the arguments to a function can match up with the parameters. The function with the *best* match is selected. The levels of matching are:

1. no match
2. match with implicit conversions
3. match with qualifier conversion (if the argument type is qualifier-convertible to the parameter type)
4. exact match

Each argument (including any `this` pointer) is compared against the function's corresponding parameter, to determine the match level for that argument. The match level for a function is the *worst* match level of each of its arguments.

Literals do not match `ref` or `out` parameters.

If two or more functions have the same match level, then *partial ordering* is used to try to find the best match. Partial ordering finds the most specialized function. If neither function is more specialized than the other, then it is an ambiguity error. Partial ordering is determined for functions `f()` and `g()` by taking the parameter types of `f()`, constructing a list of arguments by taking the default values of those types, and attempting to match them against `g()`. If it succeeds, then `g()` is at least as specialized as `f()`. For example:

```
class A { }
class B : A { }
class C : B { }
void foo(A);
void foo(B);

void test()
{
    C c;
    /* Both foo(A) and foo(B) match with implicit conversion rules.
     * Applying partial ordering rules,
     * foo(B) cannot be called with an A, and foo(A) can be called
     * with a B. Therefore, foo(B) is more specialized, and is selected.
     */
    foo(c); // calls foo(B)
}
```

A function with a variadic argument is considered less specialized than a function without.

Functions defined with non-D linkage cannot be overloaded. This is because the name mangling might not take the parameter types into account.

Overload Sets

Functions declared at the same scope overload against each other, and are called an *Overload Set*. A typical example of an overload set are functions defined at module level:

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

`A.foo()` and `A.foo(long)` form an overload set. A different module can also define functions with the same name:

```
module B;
class C { }
void foo(C) { }
void foo(int i) { }
```

and A and B can be imported by a third module, C. Both overload sets, the `A.foo` overload set and the `B.foo` overload set, are found. An instance of `foo` is selected based on it matching in exactly one overload set:

```
import A;
import B;

void bar(C c)
{
    foo();    // calls A.foo()
    foo(1L);  // calls A.foo(long)
    foo(c);   // calls B.foo(C)
    foo(1,2); // error, does not match any foo
    foo(1);   // error, matches A.foo(long) and B.foo(int)
    A.foo(1); // calls A.foo(long)
}
```

Even though `B.foo(int)` is a better match than `A.foo(long)` for `foo(1)`, it is an error because the two matches are in different overload sets.

Overload sets can be merged with an alias declaration:

```
import A;
import B;
```

```
alias foo = A.foo;
alias foo = B.foo;

void bar(C c)
{
    foo();    // calls A.foo()
    foo(1L);  // calls A.foo(long)
    foo(c);   // calls B.foo(C)
    foo(1,2); // error, does not match any foo
    foo(1);   // calls B.foo(int)
    A.foo(1); // calls A.foo(long)
}
```

19.16 Function Parameters

Parameter Storage Classes

Parameter storage classes are `in`, `out`, `ref`, `lazy`, `const`, `immutable`, `shared`, `inout` or `scope`. For example:

```
int foo(in int x, out int y, ref int z, int q);
```

x is `in`, y is `out`, z is `ref`, and q is none.

- The function declaration makes it clear what the inputs and outputs to the function are.
- It eliminates the need for IDL (interface description language) as a separate language.
- It provides more information to the compiler, enabling more error checking and possibly better code generation.

Parameter Storage Classes

Storage Class	Description
none	parameter becomes a mutable copy of its argument
`in`	defined as `scope const`. However `in` has not yet been properly implemented so it's current implementation is equivalent to `const`. It is recommended to avoid using `in` until it is properly defined and implemented. Use `scope const` or `const` explicitly instead.
`out`	parameter is initialized upon function entry with the default value for its type
`ref`	parameter is passed by reference
`scope`	references in the parameter cannot be escaped (e.g. assigned to a global variable). Ignored for parameters with no references
`return`	Parameter may be returned or copied to the first parameter, but otherwise does not escape from the function. Such copies are required not to outlive the argument(s) they were derived from. Ignored for parameters with no references. See Scope Parameters.
`lazy`	argument is evaluated by the called function and not by the caller
`const`	argument is implicitly converted to a const type
`immutable`	argument is implicitly converted to an immutable type
`shared`	argument is implicitly converted to a shared type
`inout`	argument is implicitly converted to an inout type

```
void foo(out int x)
{
    // x is set to int.init,
    // which is 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(ref 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/ref apply only to the reference and not the contents.

Lazy Parameters

An argument to a `lazy` parameter is not evaluated before the function is called. The argument is only evaluated if/when the parameter is evaluated within the function. Hence, a `lazy` argument can be executed 0 or more times.

```
import std.stdio : writeln;

void main()
{
    int x;
    3.times(writeln(x++));
    writeln("-");
    writeln(x);
}

void times(int n, lazy void exp)
{
    while (n--)
        exp();
}
```

prints to the console:

```
0
1
2
-
3
```

A `lazy` parameter cannot be an lvalue.

The underlying delegate of the lazy parameter may be extracted by using the & operator:

```
void test(lazy int dg)
{
    int delegate() dg_ = &dg;
    assert(dg_() == 7);
    assert(dg == dg_());
}

void main()
{
    int a = 7;
    test(a);
}
```

A lazy parameter of type void can accept an argument of any type.

See Also: Lazy Variadic Functions

Function Default Arguments

Function parameter declarations can have default values:

```
void foo(int x, int y = 3)
{
    ...
}
...
foo(4);   // same as foo(4, 3);
```

Default parameters are resolved and semantically checked in the context of the function declaration.

```
module m;
private immutable int b;
pure void g(int a=b){}

import m;
int b;
pure void f()
{
  g();  // ok, uses m.b
}
```

The attributes of the *AssignExpression* are applied where the default expression is used.

```
module m;
int b;
pure void g(int a=b){}
```

```
import m;
enum int b = 3;
pure void f()
{
  g();  // error, cannot access mutable global 'm.b' in pure function
}
```

If the default value for a parameter is given, all following parameters must also have default values.

Return Ref Parameters

Note: The `return` attribute is currently only enforced by `dmd` when the `-dip25` switch is passed.

Return ref parameters are used with ref functions to ensure that the returned reference will not outlive the matching argument's lifetime.

```
ref int identity(return ref int x) {
  return x; // pass-through function that does nothing
}

ref int fun() {
  int x;
  return identity(x); // Error: escaping reference to local variable x
}

ref int gun(return ref int x) {
  return identity(x); // OK
}
```

Struct non-static methods marked with the `return` attribute ensure the returned reference will not outlive the struct instance.

```
struct S
{
    private int x;
    ref int get() return { return x; }
}

ref int escape()
```

```
{
    S s;
    return s.get(); // Error: escaping reference to local variable s
}
```

Returning the address of a `ref` variable is also checked.

```
int* pluto(ref int i)
{
    return &i; // error: returning &i escapes a reference to parameter i
}

int* mars(return ref int i)
{
    return &i;  // ok
}
```

If the function returns `void`, and the first parameter is `ref` or `out`, then all subsequent `return ref` parameters are considered as being assigned to the first parameter for lifetime checking. The `this` reference parameter to a struct non-static member function is considered the first parameter.

If there are multiple `return ref` parameters, the lifetime of the return value is the smallest lifetime of the corresponding arguments.

Neither the type of the `return ref` parameter(s) nor the type of the return value is considered when determining the lifetime of the return value.

It is not an error if the return type does not contain any indirections.

```
int mercury(return ref int i)
{
    return i; // ok
}
```

Template functions, auto functions, nested functions and lambdas can deduce the `return` attribute.

```
ref int templateFunction()(ref int i)
{
    return i; // ok
}

ref auto autoFunction(ref int i)
{
    return i; // ok
```

```
}

void uranus()
{
    ref int nestedFunction(ref int i)
    {
        return i; // ok
    }
}

void venus()
{
    auto lambdaFunction =
        (ref int i)
        {
            return &i; // ok
        };
}
```

`inout ref` parameters imply the `return` attribute.

```
inout(int)* neptune(inout ref int i)
{
    return &i;  // ok
}
```

Return Scope Parameters

Parameters marked as `return scope` that contain indirections can only escape those indirections via the function's return value.

```
@safe:

int* gp;
void thorin(scope int*);
void gloin(int*);
int* balin(return scope int* p, scope int* q, int* r)
{
     gp = p; // error, p escapes to global gp
     gp = q; // error, q escapes to global gp
     gp = r; // ok
```

```
    thorin(p); // ok, p does not escape thorin()
    thorin(q); // ok
    thorin(r); // ok

    gloin(p); // error, gloin() escapes p
    gloin(q); // error, gloin() escapes q
    gloin(r); // ok that gloin() escapes r

    return p; // ok
    return q; // error, cannot return 'scope' q
    return r; // ok
}
```

Class references are considered pointers that are subject to `scope`.

```
@safe:

class C { }
C gp;
void thorin(scope C);
void gloin(C);
C balin(return scope C p, scope C q, C r)
{
    gp = p; // error, p escapes to global gp
    gp = q; // error, q escapes to global gp
    gp = r; // ok

    thorin(p); // ok, p does not escape thorin()
    thorin(q); // ok
    thorin(r); // ok

    gloin(p); // error, gloin() escapes p
    gloin(q); // error, gloin() escapes q
    gloin(r); // ok that gloin() escapes r

    return p; // ok
    return q; // error, cannot return 'scope' q
    return r; // ok
}
```

`return scope` can be applied to the `this` of class and interface member functions.

```
class C
{
    C bofur() return scope { return this; }
}
```

Template functions, auto functions, nested functions and lambdas can deduce the `return scope` attribute.

Note: Checks for `scope` parameters are currently enabled only for `@safe` functions when compiled with the `-dip1000` flag.

Ref Return Scope Parameters

Parameters marked as `ref return scope` come in two forms:

```
U xerxes(ref return scope V v);      // (1) ref and return scope
ref U xerxes(ref return scope V v);  // (2) return ref and scope
```

The first form attaches the `return` to the `scope`, and has return scope parameter semantics for the value of the `ref` parameter.

The second form attaches the `return` to the `ref`, and has return ref parameter semantics with additional scope parameter semantics.

Although a struct constructor returns a reference to the instance being constructed, it is treated as form (1).

The lexical order of the attributes `ref`, `return`, and `scope` is not significant.

It is not possible to have both `return ref` and `return scope` semantics for the same parameter.

```
@safe:

struct S
{
    this(return scope   ref int* p) { ptr = p; }

    int  val;
    int* ptr;
}

int* foo1(return scope   ref S s);
int  foo2(return scope   ref S s);

ref int* foo3(return ref   scope S s);
ref int  foo4(return ref   scope S s);
```

```
int* test1(scope S s)
{
    return foo1(s);  // Error: scope variable `s` may not be returned
    return foo3(s);  // Error: scope variable `s` may not be returned
}

int test2(S s)
{
    return foo2(s);
    return foo4(s);
}

ref int* test3(S s)
{
    return foo3(s);  // Error: returning `foo3(s)` escapes a reference to parameter `s`
}

ref int test4(S s)
{
    return foo4(s);  // Error: returning `foo4(s)` escapes a reference to parameter `s`
}

S test5(ref scope int* p)
{
    return S(p); // Error: scope variable `p` may not be returned
}

S test6(ref return scope int* p)
{
    return S(p);
}
```

User-Defined Attributes for Parameters

See also: *User-Defined Attributes*

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) void 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`.

C-style variadic functions cannot be marked as `@safe`.

Access to variadic arguments is done using the standard library module `core.stdc.stdarg`.

```
import core.stdc.stdarg;

void test()
{
    foo(3, 4, 5);   // first variadic argument is 5
}

void foo(int x, int y, ...)
{
    va_list args;

    va_start(args, y);  // y is the last named parameter

    int z;
    va_arg(args, z);  // z is set to 5
}
```

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
```

To access them, the following import is required:

```
import core.vararg;
```

These variadic functions have a special local variable declared for them, `_argptr`, which is a `core.vararg` reference to the first of the variadic arguments. To access the arguments, `_argptr` must be used in conjuction with `va_arg`:

```
import core.vararg;

void test()
{
    foo(3, 4, 5);   // first variadic argument is 5
}

void foo(int x, int y, ...)
{
    int z;

    z = va_arg!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 type safety to be checked at run time.

```
import std.stdio;
import core.vararg;

class Foo { int x = 3; }
class Bar { long y = 4; }

void printargs(int x, ...)
{
    writefln("%d arguments", _arguments.length);
```

```
    for (int i = 0; i < _arguments.length; i++)
    {
        writeln(_arguments[i]);

        if (_arguments[i] == typeid(int))
        {
            int j = va_arg!(int)(_argptr);
            writefln("\t%d", j);
        }
        else if (_arguments[i] == typeid(long))
        {
            long j = va_arg!(long)(_argptr);
            writefln("\t%d", j);
        }
        else if (_arguments[i] == typeid(double))
        {
            double d = va_arg!(double)(_argptr);
            writefln("\t%g", d);
        }
        else if (_arguments[i] == typeid(Foo))
        {
            Foo f = va_arg!(Foo)(_argptr);
            writefln("\t%s", f);
        }
        else if (_arguments[i] == typeid(Bar))
        {
            Bar b = va_arg!(Bar)(_argptr);
            writefln("\t%s", b);
        }
        else
            assert(0);
    }
}

void main()
{
    Foo f = new Foo();
    Bar b = new Bar();

    writefln("%s", f);
```

```
    printargs(1, 2, 3L, 4.5, f, b);
}
```

which prints:

```
0x00870FE0
5 arguments
int
        2
long
        3
double
        4.5
Foo
        0x00870FE0
Bar
        0x00870FD0
```

D-style variadic functions cannot be marked as `@safe`.

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 main()
{
    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;
    string s;

    this(int x, string 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 );
```

The lazy variadic delegate solution is preferable to using a lazy variadic array, because each array index would evaluate every element:

```
import std.stdio;

void foo(lazy int[] arr...)
{
    writeln(arr[0]); // 1
    writeln(arr[1]); // 4, not 2
}

void main()
{
    int x;
    foo(++x, ++x);
}
```

19.17 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 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.

Local Static Variables

Local variables in functions can be declared as static or `__gshared` in which case they are statically allocated rather than being allocated on the stack. As such, their value persists beyond the exit of the function.

```
void foo()
{
    static int n;
    if (++n == 100)
        writeln("called␣100␣times");
}
```

The initializer for a static variable must be evaluatable at compile time, and they are initialized upon the start of the thread (or the start of the program for `__gshared`). There are no static constructors or static destructors for static local variables.

Although static variable name visibility follows the usual scoping rules, the names of them must be unique within a particular function.

```
void main()
{
    { static int x; }
    { static int x; } // error
    { int i; }
    { int i; } // ok
}
```

19.18 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;
    }
}
```

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
}
```

There are several workarounds for this limitation:

- Declare the functions to be static members of a nested struct:

  ```
  void test()
  {
      static struct S
      {
          static void foo() { bar(); } // ok
          static void bar() { foo(); } // ok
      }

      S.foo();  // compiles (but note the infinite runtime loop)
  }
  ```

- Declare one or more of the functions to be function templates even if they take no specific template arguments:

  ```
  void test()
  {
      void foo()() { bar(); } // ok (foo is a function template)
      void bar()   { foo(); } // ok
  }
  ```

- Declare the functions inside of a mixin template:

  ```
  mixin template T()
  {
      void foo() { bar(); } // ok
      void bar() { foo(); } // ok
  ```

```
}

void main()
{
    mixin T!();
}
```

- Use a delegate:

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

Nested functions cannot be overloaded.

19.19 Delegates, Function Pointers, and 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
}
```

Note: Two functions with identical bodies, or two functions that compile to identical assembly code, are not guaranteed to have distinct function pointer values. The compiler is free to merge functions bodies into one if they compile to identical code.

```
int abc(int x) { return x + 1; }
int def(int y) { return y + 1; }

int delegate(int) fp1 = &abc;
int delegate(int) fp2 = &def;
// Do not rely on fp1 and fp2 being different values; the compiler may merge
// them.
```

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 referenced by a nested function are still valid even after the function exits (this is different from D 1.0). This is called a *closure*. Returning addresses of stack variables, however, is not a closure and is an error.

```
int* bar()
{
    int b;
    test();
    int i = dg(); // ok, test.a is in a closure and still 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.

Functions and delegates declared at module scope are zero-initialized by default. However both can be initialized to any function pointer (including a function literal). For delegates, the context pointer `.ptr` is initialized to null.

```
int function() foo = { return 42; };
int delegate() bar = { return 43; };
int delegate() baz;
void main()
{
    assert(foo() == 42);
    assert(bar() == 43);
    assert(baz is null);
}
```

Function pointers can be passed to functions taking a delegate argument by passing them through the `std.functional.toDelegate` template, which converts any callable to a delegate without context.

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

Anonymous Functions and Anonymous Delegates

See *FunctionLiterals*.

19.20 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(string[] args) { ... }
int main() { ... }
int main(string[] args) { ... }
```

19.21 Function Templates

Template functions are useful for avoiding code duplication - instead of writing several copies of a function, each with a different parameter type, a single function template can be sufficient. For example:

```
// Only one copy of func needs to be written
void func(T)(T x)
{
    writeln(x);
}
void main()
{
    func!(int)(1); // pass an int
    func(1);     // pass an int, inferring T = int
    func("x");   // pass a string
    func(1.0);   // pass a float
    struct S {}
    S s;
    func(s);     // pass a struct
}
```

`func` takes a template parameter `T` and a runtime parameter, `x`. `T` is a placeholder identifier that can accept any type. In this case `T` can be inferred from the runtime argument type.

Note: Using the name `T` is just a convention. The name `TypeOfX` could have been used instead.

For more information, see function templates.

19.22 Compile Time Function Execution (CTFE)

Functions which are both portable and free of global side-effects can be executed at compile time. In certain contexts, such compile time execution is guaranteed. It is called Compile Time Function Execution (CTFE) then. The contexts that trigger CTFE are:

- initialization of a static variable or a manifest constant
- static initializers of struct/class members
- dimension of a static array
- argument for a template value parameter
- `static if`
- `static foreach`
- `static assert`
- `mixin` statement
- `pragma` argument

```
enum eval(Args...) = Args[0];

int square(int i)
{
    return i * i;
}

void foo()
{
    static j = square(3);      // CTFE
    writeln(j);
    assert(square(4));         // run time
    writeln(eval!(square(5))); // CTFE
}
```

CTFE is subject to the following restrictions:

1. The function source code must be available to the compiler. Functions which exist in the source code only as extern declarations cannot be executed in CTFE.
2. Executed expressions may not reference any global or local static variables.
3. asm statements are not permitted
4. Non-portable casts (eg, from `int[]` to `float[]`), including casts which depend on endianness, are not permitted. Casts between signed and unsigned types are permitted
5. Reinterpretation of overlapped fields in a Union is not permitted.

Pointers are permitted in CTFE, provided they are used safely:

- C-style semantics on pointer arithmetic are strictly enforced. Pointer arithmetic is permitted only on pointers which point to static or dynamic array elements. Such pointers must point to an element of the array, or to the first element past the array. Pointer arithmetic is completely forbidden on pointers which are null, or which point to a non-array.
- The memory location of different memory blocks is not defined. Ordered comparison (`<`, `<=`, `>`, `>=`) between two pointers is permitted when both pointers point to the same array, or when at least one pointer is `null`.
- Pointer comparisons between independent memory blocks will generate a compile-time error, unless two such comparisons are combined using `&&` or `||` to yield a result which is independent of the ordering of memory blocks. Each comparison must consist of two pointer expressions compared with `<`, `<=`, `>`, or `>=`, and may optionally be negated with `!`.
  For example, the expression `(p1 > q1 && p2 <= q2)` is permitted when `p1`, `p2` are expressions yielding pointers to memory block P, and `q1`, `q2` are expressions yielding pointers to memory block Q, even when P and Q are unrelated memory blocks. It returns true if `[p1..p2]` lies inside `[q1..q2]`, and false otherwise. Similarly, the expression `(p1 < q1` | p2 > q2)| is true if `[p1..p2]` lies outside `[q1..q2]`, and false otherwise.
- Equality comparisons (==, !=, is, !is) are permitted between all pointers, without restriction.
- Any pointer may be cast to `void*` and from `void*` back to its original type. Casting between pointer and non-pointer types is prohibited.

Note that the above restrictions apply only to expressions which are actually executed. For example:

```
static int y = 0;

int countTen(int x)
{
    if (x > 10)
        ++y;
    return x;
}

static assert(countTen(6) == 6);    // OK
static assert(countTen(12) == 12);  // invalid, modifies y.
```

The `__ctfe` boolean pseudo-variable, which evaluates to true in CTFE, but false otherwise, can be used to provide an alternative execution path to avoid operations which are forbidden in CTFE. Every usage of `__ctfe` is evaluated before code generation and therefore has no run-time cost, even if no optimizer is used.

Executing functions via CTFE can take considerably longer than executing it at run time. If the function goes into an infinite loop, it will hang at compile time (rather than hanging at run time).

Non-recoverable errors (such as assert failures) do not throw exceptions; instead, they end interpretation immediately.

Functions executed via CTFE can give different results from run time in the following scenarios:

- floating point computations may be done at a higher precision than run time
- dependency on implementation defined order of evaluation
- use of uninitialized variables

These are the same kinds of scenarios where different optimization settings affect the results.

String Mixins and Compile Time Function Execution

Any functions that execute in CTFE must also be executable at run time. The compile time evaluation of a function does the equivalent of running the function at run time. This means that the semantics of a function cannot depend on compile time values of the function. For example:

```
int foo(string s)
{
    return mixin(s);
}

const int x = foo("1");
```

is illegal, because the runtime code for `foo` cannot be generated. A function template would be the appropriate method to implement this sort of thing.

19.23 No-GC Functions

No-GC functions are functions marked with the `@nogc` attribute. Those functions do not allocate memory on the GC heap, through the following language features:

- constructing an array on the heap
- resizing an array by writing to its `.length` property
- array concatenation and appending
- constructing an associative array on the heap
- indexing an associative array (because it may throw `RangeError` if the specified key is not present)
- allocating an object on the heap

```
@nogc void foo()
{
    auto a = ['a'];     // error, allocates
    a.length = 1;       // error, array resizing allocates
    a = a ~ a;          // error, arrays concatenation allocates
    a ~= 'c';           // error, appending to arrays allocates

    auto aa = ["x":1]; // error, allocates
    aa["abc"];          // error, indexing may allocate and throws

    auto p = new int;  // error, operator new allocates
}
```

No-GC functions cannot call functions that are not `@nogc`.

```
@nogc void foo()
{
    bar();              // error, bar() may allocate
}

void bar() { }
```

No-GC functions cannot be closures.

```
@nogc int delegate() foo()
{
    int n;              // error, variable n cannot be allocated on heap
    return (){ return n; }
}
```

`@nogc` affects the type of the function. A `@nogc` function is covariant with a non-`@nogc` function.

```
void function() fp;
void function() @nogc gp;  // pointer to @nogc function

void foo();
@nogc void bar();

void test()
{
    fp = &foo; // ok
    fp = &bar; // ok, it's covariant
    gp = &foo; // error, not contravariant
```

```
    gp = &bar; // ok
}
```

To ease debugging, in a *ConditionalStatement* controlled by a *DebugCondition* `@nogc` functions can call functions that are not `@nogc`.

19.24 Function Safety

Safe functions are functions that are statically checked to exhibit no possibility of *undefined behavior*. Undefined behavior is often used as a vector for malicious attacks.

Safe Functions

Safe functions are marked with the `@safe` attribute.

The following operations are not allowed in safe functions:

- No casting from a pointer type to any type with pointers other than `void*`.
- No casting from any non-pointer type to a pointer type.
- No pointer arithmetic (including pointer indexing).
- Cannot access unions that have pointers or references overlapping with other types.
- Calling any system functions.
- No catching of exceptions that are not derived from `class Exception`.
- Disallow `@system` asm statements.
- No explicit casting of mutable objects to immutable.
- No explicit casting of immutable objects to mutable.
- No explicit casting of thread local objects to shared.
- No explicit casting of shared objects to thread local.
- No taking the address of a local variable or function parameter.
- Cannot access `__gshared` variables.
- Cannot use `void` initializers for pointers.
- Cannot use `void` initializers for class or interface references.

When indexing and slicing an array, an out of bounds access will cause a runtime error, in order to prevent undefined behavior.

Functions nested inside safe functions default to being safe functions.

Safe functions are covariant with trusted or system functions.

Note: The verifiable safety of functions may be compromised by bugs in the compiler and specification. Please report all such errors so they can be corrected.

Trusted Functions

Trusted functions are marked with the `@trusted` attribute.

Trusted functions are guaranteed to not exhibit any undefined behavior if called by a safe function. Furthermore, calls to trusted functions cannot lead to undefined behavior in `@safe` code that is executed afterwards. It is the responsibility of the programmer to ensure that these guarantees are upheld.

Example:

```
immutable(int)* f(int* p) @trusted
{
    version (none) p[2] = 13;
    // Invalid. p[2] is out of bounds. This line would exhibit undefined
    // behavior.

    version (none) p[1] = 13;
    // Invalid. In this program, p[1] happens to be in-bounds, so the
    // line would not exhibit undefined behavior, but a trusted function
    // is not allowed to rely on this.

    version (none) return cast(immutable) p;
    // Invalid. @safe code still has mutable access and could trigger
    // undefined behavior by overwriting the value later on.

    int* p2 = new int;
    *p2 = 42;
    return cast(immutable) p2;
    // Valid. After f returns, no mutable aliases of p2 can exist.
}

void main() @safe
{
    int[2] a = [10, 20];
    int* mp = &a[0];
    immutable(int)* ip = f(mp);
    assert(a[1] == 20); // Guaranteed. f cannot access a[1].
    assert(ip !is mp); // Guaranteed. f cannot introduce unsafe aliasing.
}
```

Trusted functions may call safe, trusted, or system functions.

Trusted functions are covariant with safe or system functions.

Best Practices: Trusted functions should be kept small so that they are easier to manually verify.

System Functions

System functions are functions not marked with `@safe` or `@trusted` and are not nested inside `@safe` functions. System functions may be marked with the `@system` attribute. A function being system does not mean it actually is unsafe, it just means that the compiler is unable to verify that it cannot exhibit undefined behavior.

System functions are **not** covariant with trusted or safe functions.

19.25 Function Attribute Inference

*FunctionLiteral*s and function templates, since their function bodies are always present, infer the `pure`, `nothrow`, `@safe`, and `@nogc` attributes unless specifically overridden.

Attribute inference is not done for other functions, even if the function body is present.

The inference is done by determining if the function body follows the rules of the particular attribute.

Cyclic functions (i.e. functions that wind up directly or indirectly calling themselves) are inferred as being impure, throwing, and @system.

If a function attempts to test itself for those attributes, then the function is inferred as not having those attributes.

19.26 Uniform Function Call Syntax (UFCS)

A free function can be called with a syntax that looks as if the function were a member function of its first parameter type.

```
void func(X thisObj);

X obj;
obj.func();
// If 'obj' does not have regular member 'func',
// it's automatically rewritten to 'func(obj)'
```

This provides a way to add functions to a class externally as if they were public final member functions, which enables function chaining and component programming.

```
stdin.byLine(KeepTerminator.yes)
    .map!(a => a.idup)
    .array
```

```
    .sort
    .copy(stdout.lockingTextWriter());
```

It also works with `@property` functions:

```
@property prop(X thisObj);
@property prop(X thisObj, int value);

X obj;
obj.prop;      // Rewrites to: prop(obj);
obj.prop = 1;  // Rewrites to: prop(obj, 1);
```

Syntactically parenthesis-less check for `@property` functions is done at the same time as UFCS rewrite.

When UFCS rewrite is necessary, compiler searches the name on accessible module level scope, in order from the innermost scope.

```
module a;
void foo(X);
alias boo = foo;
void main()
{
    void bar(X);
    import b : baz;  // void baz(X);

    X obj;
    obj.foo();    // OK, calls a.foo;
    //obj.bar();  // NG, UFCS does not see nested functions
    obj.baz();    // OK, calls b.baz, because it is declared at the
                  // top level scope of module b

    import b : boo = baz;
    obj.boo();    // OK, calls aliased b.baz instead of a.boo (== a.foo),
                  // because the declared alias name 'boo' in local scope
                  // overrides module scope name
}
class C
{
    void mfoo(X);
    static void sbar(X);
    import b : ibaz = baz;  // void baz(X);
    void test()
```

```
    {
        X obj;
        //obj.mfoo();  // NG, UFCS does not see member functions
        //obj.sbar();  // NG, UFCS does not see static member functions
        obj.ibaz();    // OK, ibaz is an alias of baz which declared at
                       //     the top level scope of module b
    }
}
```

The reason why local symbols are not considered by UFCS, is to avoid unexpected name conflicts. See below problematic examples.

```
int front(int[] arr) { return arr[0]; }

void main()
{
    int[] a = [1,2,3];
    auto x = a.front();   // call .front by UFCS

    auto front = x;       // front is now a variable
    auto y = a.front();   // Error, front is not a function
}

class C
{
    int[] arr;
    int front()
    {
        return arr.front(); // Error, C.front is not callable
                            // using argument types (int[])
    }
}
```

Chapter 20

Operator Overloading

Operator overloading is accomplished by rewriting operators whose operands are class or struct objects into calls to specially named members. No additional syntax is used.

- Unary Operator Overloading
- Cast Operator Overloading
- Binary Operator Overloading
- Overloading the Comparison Operators
  - Overloading `==` and `!=`
  - Overloading `<`, `<=`, `>`, and `>=`
- Function Call Operator Overloading
- Assignment Operator Overloading
- Op Assignment Operator Overloading
- Array Indexing and Slicing Operators Overloading
  - Index Operator Overloading
  - Slice Operator Overloading
  - Dollar Operator Overloading
- Forwarding
- D1 style operator overloading

20.1 Unary Operator Overloading

Overloadable Unary Operators

op	*rewrite*
`-`*e*	*e*`.opUnary!("-")()`
`+`*e*	*e*`.opUnary!("+")()`
`~`*e*	*e*`.opUnary!("~")()`
`*`*e*	*e*`.opUnary!("*")()`
`++`*e*	*e*`.opUnary!("++")()`
`--`*e*	*e*`.opUnary!("--")()`

For example, in order to overload the `-` (negation) operator for struct S, and no other operator:

```
struct S
{
    int m;

    int opUnary(string s)() if (s == "-")
    {
        return -m;
    }
}

int foo(S s)
{
    return -s;
}
```

Postincrement *e*`++` and Postdecrement *e*`--` Operators

These are not directly overloadable, but instead are rewritten in terms of the $++e$ and $-e$ prefix operators:

Postfix Operator Rewrites

op	*rewrite*
e`--`	`(auto t = `*e*`, --`*e*`, t)`
e`++`	`(auto t = `*e*`, ++`*e*`, t)`

Overloading Index Unary Operators

Overloadable Index Unary Operators

op	*rewrite*
`-`a`[`b_1, b_2, ... b_n`]`	a`.opIndexUnary!("-")(`b_1, b_2, ... b_n`)`
`+`a`[`b_1, b_2, ... b_n`]`	a`.opIndexUnary!("+")(`b_1, b_2, ... b_n`)`
`~`a`[`b_1, b_2, ... b_n`]`	a`.opIndexUnary!("~")(`b_1, b_2, ... b_n`)`
`*`a`[`b_1, b_2, ... b_n`]`	a`.opIndexUnary!("*")(`b_1, b_2, ... b_n`)`
`++`a`[`b_1, b_2, ... b_n`]`	a`.opIndexUnary!("++")(`b_1, b_2, ... b_n`)`
`--`a`[`b_1, b_2, ... b_n`]`	a`.opIndexUnary!("--")(`b_1, b_2, ... b_n`)`

Overloading Slice Unary Operators

Overloadable Slice Unary Operators

op	*rewrite*
`-`a`[`$i..j$`]`	a`.opIndexUnary!("-")(`a`.opSlice(`i, j`))`
`+`a`[`$i..j$`]`	a`.opIndexUnary!("+")(`a`.opSlice(`i, j`))`
`~`a`[`$i..j$`]`	a`.opIndexUnary!("~")(`a`.opSlice(`i, j`))`
`*`a`[`$i..j$`]`	a`.opIndexUnary!("*")(`a`.opSlice(`i, j`))`
`++`a`[`$i..j$`]`	a`.opIndexUnary!("++")(`a`.opSlice(`i, j`))`
`--`a`[`$i..j$`]`	a`.opIndexUnary!("--")(`a`.opSlice(`i, j`))`
`-`a`[ ]`	a`.opIndexUnary!("-")()`
`+`a`[ ]`	a`.opIndexUnary!("+")()`
`~`a`[ ]`	a`.opIndexUnary!("~")()`
`*`a`[ ]`	a`.opIndexUnary!("*")()`
`++`a`[ ]`	a`.opIndexUnary!("++")()`
`--`a`[ ]`	a`.opIndexUnary!("--")()`

For backward compatibility, if the above rewrites fail to compile and `opSliceUnary` is defined, then the rewrites `a.opSliceUnary!(op)(i, j)` and `a.opSliceUnary!(op)` are tried instead, respectively.

20.2 Cast Operator Overloading

To define how one type can be cast to another, define the `opCast` template method, which is used as follows:

Cast Operators

op	*rewrite*
`cast(`*type*`)` *e*	*e*`.opCast!(`*type*`)()`

Note that `opCast` is only ever used with an explicit `cast` expression, except in the case of boolean operations (see next section)

Boolean Operations

Notably absent from the list of overloaded unary operators is the ! logical negation operator. More obscurely absent is a unary operator to convert to a bool result. Instead, these are covered by a rewrite to:

```
opCast!(bool)(e)
```

So,

```
if (e)   =>  if (e.opCast!(bool))
if (!e)  =>  if (!e.opCast!(bool))
```

etc., whenever a bool result is expected. This only happens, however, for instances of structs. Class references are converted to bool by checking to see if the class reference is null or not.

20.3 Binary Operator Overloading

The following binary operators are overloadable:

Overloadable Binary Operators

`+`	`-`	`*`	`/`	`%`	`^^`	`&`
`|`	`^`	`<<`	`>>`	`>>>`	`~`	`in`

The expression:

a *op* b

is rewritten as both:

a.opBinary!("*op*")(b)
b.opBinaryRight!("*op*")(a)

and the one with the 'better' match is selected. It is an error for both to equally match.

Operator overloading for a number of operators can be done at the same time. For example, if only the + or - operators are supported:

```
T opBinary(string op)(T rhs)
{
    static if (op == "+") return data + rhs.data;
    else static if (op == "-") return data - rhs.data;
    else static assert(0, "Operator "~op~" not implemented");
}
```

To do them all en masse:

```
T opBinary(string op)(T rhs)
{
    return mixin("data "~op~" rhs.data");
}
```

Note that `opIn` and `opIn_r` have been deprecated in favor of `opBinary!"in"` and `opBinaryRight!"in"` respectively.

20.4 Overloading the Comparison Operators

D allows overloading of the comparison operators `==`, `!=`, `<`, `<=`, `>=`, `>` via two functions, `opEquals` and `opCmp`.

The equality and inequality operators are treated separately because while practically all user-defined types can be compared for equality, only a subset of types have a meaningful ordering. For example, while it makes sense to determine if two RGB color vectors are equal, it is not meaningful to say that one color is greater than another, because colors do not have an ordering. Thus, one would define `opEquals` for a `Color` type, but not `opCmp`.

Furthermore, even with orderable types, the order relation may not be linear. For example, one may define an ordering on sets via the subset relation, such that `x < y` is true if `x` is a (strict) subset of `y`. If `x` and `y` are disjoint sets, then neither `x < y` nor `y < x` holds, but that does not imply that `x == y`. Thus, it is insufficient to determine equality purely based on `opCmp` alone. For this reason, `opCmp` is only used for the inequality operators `<`, `<=`, `>=`, and `>`. The equality operators `==` and `!=` always employ `opEquals` instead.

Therefore, it is the programmer's responsibility to ensure that `opCmp` and `opEquals` are consistent with each other. If `opEquals` is not specified, the compiler provides a default version that does member-wise comparison. If this suffices, one may define only `opCmp` to customize the behaviour of the inequality operators. But if not, then a custom version of `opEquals` should be defined as well, in order to preserve consistent semantics between the two kinds of comparison operators.

Finally, if the user-defined type is to be used as a key in the built-in associative arrays, then the programmer must ensure that the semantics of `opEquals` and `toHash` are consistent. If not, the associative array may not work in the expected manner.

Overloading `==` and `!=`

Expressions of the form `a != b` are rewritten as `!(a == b)`.

Given `a == b` :

1. If a and b are both class objects, then the expression is rewritten as:

   ```
   .object.opEquals(a, b)
   ```

   and that function is implemented as:

   ```
   bool opEquals(Object a, Object b)
   {
       if (a is b) return true;
       if (a is null || b is null) return false;
       if (typeid(a) == typeid(b)) return a.opEquals(b);
       return a.opEquals(b) && b.opEquals(a);
   }
   ```

2. Otherwise the expressions `a.opEquals(b)` and `b.opEquals(a)` are tried. If both resolve to the same `opEquals` function, then the expression is rewritten to be `a.opEquals(b)`.
3. If one is a better match than the other, or one compiles and the other does not, the first is selected.
4. Otherwise, an error results.

If overridding `Object.opEquals()` for classes, the class member function signature should look like:

```
class C
{
    override bool opEquals(Object o) { ... }
}
```

If structs declare an `opEquals` member function for the identity comparison, it could have several forms, such as:

```
struct S
{
    // lhs should be mutable object
    bool opEquals(const S s) { ... }        // for r-values (e.g. temporaries)
    bool opEquals(ref const S s) { ... }    // for l-values (e.g. variables)

    // both hand side can be const object
    bool opEquals(const S s) const { ... }  // for r-values (e.g. temporaries)
}
```

Alternatively, you can declare a single templated `opEquals` function with an auto ref parameter:

```
struct S
{
    // for l-values and r-values,
    // with converting both hand side implicitly to const
    bool opEquals()(auto ref const S s) const { ... }
}
```

Overloading <, <=, >, and >=

Comparison operations are rewritten as follows:

Rewriting of comparison operations

comparison	rewrite 1	rewrite 2
`a < b`	`a.opCmp(b) < 0`	`b.opCmp(a) > 0`
`a <= b`	`a.opCmp(b) <= 0`	`b.opCmp(a) >= 0`
`a >b`	`a.opCmp(b) > 0`	`b.opCmp(a) < 0`
`a >= b`	`a.opCmp(b) >= 0`	`b.opCmp(a) <= 0`

Both rewrites are tried. If only one compiles, that one is taken. If they both resolve to the same function, the first rewrite is done. If they resolve to different functions, the best matching one is used. If they both match the same, but are different functions, an ambiguity error results.

If overriding `Object.opCmp()` for classes, the class member function signature should look like:

```
class C
{
    override int opCmp(Object o) { ... }
}
```

If structs declare an `opCmp` member function, it should have the following form:

```
struct S
{
    int opCmp(ref const S s) const { ... }
}
```

Note that `opCmp` is only used for the inequality operators; expressions like `a == b` always uses `opEquals`. If `opCmp` is defined but `opEquals` isn't, the compiler will supply a default version of `opEquals` that performs member-wise comparison. If this member-wise comparison is not consistent with the user-defined `opCmp`, then it is up to the programmer to supply an appropriate version of `opEquals`. Otherwise, inequalities like `a <= b` will behave inconsistently with equalities like `a == b`.

20.5 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.

Note that merely declaring `opCall` automatically disables struct literal syntax. To avoid the limitation, you need to also declare a constructor so that it takes priority over `opCall` in `Type(...)` syntax.

```
struct Multiplier
{
    int factor;
    this(int num) { factor = num; }
    int opCall(int value) { return value * factor; }
}
void test()
{
    Multiplier m = Multiplier(10);  // invoke constructor
    int result = m(5);              // invoke opCall
    assert(result == 50);
}
```

Static opCall

`static opCall` also works as expected for a function call operator with type names.

```
struct Double
```

```
{
    static int opCall(int x) { return x * 2; }
}
void test()
{
    int i = Double(2);
    assert(i == 4);
}
```

Mixing struct constructors and `static opCall` is not allowed.

```
struct S
{
    this(int i) {}
    static S opCall()  // disallowed due to constructor
    {
        return S.init;
    }
}
```

Note: `static opCall` can be used to simulate struct constructors with no arguments, but this is not recommended practice. Instead, the preferred solution is to use a factory function to create struct instances.

20.6 Assignment Operator Overloading

The assignment operator `=` can be overloaded if the left hand side is a struct aggregate, and `opAssign` is a member function of that aggregate.

For struct types, operator overloading for the identity assignment is allowed.

```
struct S
{
    // identity assignment, allowed.
    void opAssign(S rhs);

    // not identity assignment, also allowed.
    void opAssign(int);
}
S s;
s = S();      // Rewritten to s.opAssign(S());
s = 1;        // Rewritten to s.opAssign(1);
```

However for class types, identity assignment is not allowed. All class types have reference semantics, so identity assignment by default rebinds the left-hand-side to the argument at the right, and this is not overridable.

```
class C
{
    // If X is the same type as C or the type which is
    // implicitly convertible to C, then opAssign would
    // accept identity assignment, which is disallowed.
    // C opAssign(...);
    // C opAssign(X);
    // C opAssign(X, ...);
    // C opAssign(X ...);
    // C opAssign(X, U = defaultValue, etc.);

    // not an identity assignment - allowed
    void opAssign(int);
}
C c = new C();
c = new C();  // Rebinding referencee
c = 1;        // Rewritten to c.opAssign(1);
```

Index Assignment Operator Overloading

If the left hand side of an assignment is an index operation on a struct or class instance, it can be overloaded by providing an `opIndexAssign` member function. Expressions of the form `a[`b_1, b_2, ... b_n`] = c` are rewritten as `a.opIndexAssign(c,` b_1, b_2, ... b_n`)`.

```
struct A
{
    int opIndexAssign(int value, size_t i1, size_t i2);
}

void test()
{
    A a;
    a[i,3] = 7;  // same as a.opIndexAssign(7,i,3);
}
```

Slice Assignment Operator Overloading

If the left hand side of an assignment is a slice operation on a struct or class instance, it can be overloaded by implementing an `opIndexAssign` member function that takes the return value of the `opSlice` function as parameter(s). Expressions of the form `a[`*i..j*`] = c` are rewritten as `a.opIndexAssign(c, a.opSlice(`*i*, *j*`))`, and `a[] = c` as `a.opIndexAssign(c)`.

See Array Indexing and Slicing Operators Overloading for more details.

```
struct A
{
    int opIndexAssign(int v);  // overloads a[] = v
    int opIndexAssign(int v, size_t[2] x);  // overloads a[i .. j] = v
    int[2] opSlice(size_t x, size_t y);     // overloads i .. j
}

void test()
{
    A a;
    int v;

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

For backward compatibility, if rewriting `a[`*i..j*`]` as `a.opIndexAssign(a.opSlice(`*i*, *j*`))` fails to compile, the legacy rewrite `opSliceAssign(c, `*i*, *j*`)` is used instead.

20.7 Op Assignment Operator Overloading

The following op assignment operators are overloadable:

Overloadable Op Assignment Operators

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

The expression:

`a` *`op`*`= b`

is rewritten as:

`a.opOpAssign!("`*`op`*`")(b)`

Index Op Assignment Operator Overloading

If the left hand side of an *op*= is an index expression on a struct or class instance and `opIndexOpAssign` is a member:

`a[`b_1`, `b_2`, ... `b_n`] `*op*`= c`

it is rewritten as:

`a.opIndexOpAssign!("`*op*`")(c, `b_1`, `b_2`, ... `b_n`)`

Slice Op Assignment Operator Overloading

If the left hand side of an *op*= is a slice expression on a struct or class instance and `opIndexOpAssign` is a member:

`a[`*i*`..`*j*`] `*op*`= c`

it is rewritten as:

`a.opIndexOpAssign!("`*op*`")(c, a.opSlice(`*i*`, `*j*`))`

and

`a[] `*op*`= c`

it is rewritten as:

`a.opIndexOpAssign!("`*op*`")(c)`

For backward compatibility, if the above rewrites fail and `opSliceOpAssign` is defined, then the rewrites `a.opSliceOpAssign(c, i, j)` and `a.opSliceOpAssign(c)` are tried, respectively.

20.8 Array Indexing and Slicing Operators Overloading

The array indexing and slicing operators are overloaded by implementing the `opIndex`, `opSlice`, and `opDollar` methods. These may be combined to implement multidimensional arrays.

The code example below shows a simple implementation of a 2-dimensional array with overloaded indexing and slicing operators. The explanations of the various constructs employed are given in the sections following.

```
struct Array2D(E)
{
    E[] impl;
    int stride;
    int width, height;
```

```
this(int width, int height, E[] initialData = [])
{
    impl = initialData;
    this.stride = this.width = width;
    this.height = height;
    impl.length = width * height;
}

// Index a single element, e.g., arr[0, 1]
ref E opIndex(int i, int j) { return impl[i + stride*j]; }

// Array slicing, e.g., arr[1..2, 1..2], arr[2, 0..$], arr[0..$, 1].
Array2D opIndex(int[2] r1, int[2] r2)
{
    Array2D result;

    auto startOffset = r1[0] + r2[0]*stride;
    auto endOffset = r1[1] + (r2[1] - 1)*stride;
    result.impl = this.impl[startOffset .. endOffset];

    result.stride = this.stride;
    result.width = r1[1] - r1[0];
    result.height = r2[1] - r2[0];

    return result;
}
auto opIndex(int[2] r1, int j) { return opIndex(r1, [j, j+1]); }
auto opIndex(int i, int[2] r2) { return opIndex([i, i+1], r2); }

// Support for `x..y` notation in slicing operator for the given dimension.
int[2] opSlice(size_t dim)(int start, int end)
    if (dim >= 0 && dim < 2)
in { assert(start >= 0 && end <= this.opDollar!dim); }
body
{
    return [start, end];
}

// Support `$` in slicing notation, e.g., arr[1..$, 0..$-1].
@property int opDollar(size_t dim : 0)() { return width; }
```

```
    @property int opDollar(size_t dim : 1)() { return height; }
}

unittest
{
    auto arr = Array2D!int(4, 3, [
        0, 1, 2,  3,
        4, 5, 6,  7,
        8, 9, 10, 11
    ]);

    // Basic indexing
    assert(arr[0, 0] == 0);
    assert(arr[1, 0] == 1);
    assert(arr[0, 1] == 4);

    // Use of opDollar
    assert(arr[$-1, 0] == 3);
    assert(arr[0, $-1] == 8);   // Note the value of $ differs by dimension
    assert(arr[$-1, $-1] == 11);

    // Slicing
    auto slice1 = arr[1..$, 0..$];
    assert(slice1[0, 0] == 1 && slice1[1, 0] == 2  && slice1[2, 0] == 3 &&
           slice1[0, 1] == 5 && slice1[1, 1] == 6  && slice1[2, 1] == 7 &&
           slice1[0, 2] == 9 && slice1[1, 2] == 10 && slice1[2, 2] == 11);

    auto slice2 = slice1[0..2, 1..$];
    assert(slice2[0, 0] == 5 && slice2[1, 0] == 6 &&
           slice2[0, 1] == 9 && slice2[1, 1] == 10);

    // Thin slices
    auto slice3 = arr[2, 0..$];
    assert(slice3[0, 0] == 2 &&
           slice3[0, 1] == 6 &&
           slice3[0, 2] == 10);

    auto slice4 = arr[0..3, 2];
    assert(slice4[0, 0] == 8 && slice4[1, 0] == 9 && slice4[2, 0] == 10);
}
```

Index Operator Overloading

Expressions of the form `arr[`b_1`,` b_2`, ...` b_n`]` are translated into `arr.opIndex(`b_1`,` b_2`, ...` b_n`)`. For example:

```
struct A
{
    int opIndex(size_t i1, size_t i2, size_t i3);
}

void test()
{
    A a;
    int i;
    i = a[5,6,7];  // same as i = a.opIndex(5,6,7);
}
```

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

If an index expression can be rewritten using `opIndexAssign` or `opIndexOpAssign`, those are preferred over `opIndex`.

Slice Operator Overloading

Overloading the slicing operator means overloading expressions like `a[]` or `a[`$i..j$`]`, where the expressions inside the square brackets contain slice expressions of the form $i..j$.

To overload `a[]`, simply define `opIndex` with no parameters:

```
struct S
{
    int[] impl;
    int[] opIndex()
    {
        return impl[];
    }
}
void test()
{
    auto s = S([1,2,3]);
    auto t = s[]; // calls s.opIndex()
    assert(t == [1,2,3]);
}
```

To overload array indexing of the form `a[`$i..j$`, ...]`, two steps are needed. First, the expressions of the form $i..j$ are translated via `opSlice` into user-defined objects that encapsulate the endpoints i and j. Then these user-defined objects are passed to `opIndex` to perform the actual slicing. This design was chosen in order to support mixed indexing and slicing in multidimensional arrays; for example, in translating expressions like `arr[1, 2..3, 4]`.

More precisely, an expression of the form `arr[`b_1, b_2, ... b_n`]` is translated into `arr.opIndex(`c_1, c_2, ... c_n`)`. Each argument b_i can be either a single expression, in which case it is passed directly as the corresponding argument c_i to `opIndex`; or it can be a slice expression of the form x_i`..`y_i, in which case the corresponding argument c_i to `opIndex` is `arr.opSlice!i(`x_i`, `y_i`)`. Namely:

op	*rewrite*
`arr[1, 2, 3]`	`arr.opIndex(1, 2, 3)`
`arr[1..2, 3..4, 5..6]`	`arr.opIndex(arr.opSlice!0(1,2), arr.opSlice!1(3,4), arr.opSlice!2(5,6))`
`arr[1, 2..3, 4]`	`arr.opIndex(1, arr.opSlice!1(2,3), 4)`

Similar translations are done for assignment operators involving slicing, for example:

op	*rewrite*
`arr[1, 2..3, 4] = c`	`arr.opIndexAssign(c, 1, arr.opSlice!1(2, 3), 4)`
`arr[2, 3..4] += c`	`arr.opIndexOpAssign!"+"(c, 2, arr.opSlice!1(2, 3))`

The intention is that `opSlice!i` should return a user-defined object that represents an interval of indices along the `i`'th dimension of the array. This object is then passed to `opIndex` to perform the actual slicing operation. If only one-dimensional slicing is desired, `opSlice` may be declared without the compile-time parameter `i`.

Note that in all cases, `arr` is only evaluated once. Thus, an expression like `getArray()[1, 2..3, $-1]=c` has the effect of:

```
auto __tmp = getArray();
__tmp.opIndexAssign(c, 1, __tmp.opSlice!1(2,3), __tmp.opDollar!2 - 1);
```

where the initial function call to `getArray` is only executed once.

For backward compatibility, `a[]` and `a[`$i..j$`]` can also be overloaded by implementing `opSlice()` with no arguments and `opSlice(`i, j`)` with two arguments, respectively. This only applies for one-dimensional slicing, and dates from when D did not have full support for multidimensional arrays. This usage of `opSlice` is discouraged.

Dollar Operator Overloading

Within the arguments to array index and slicing operators, `$` gets translated to `opDollar!i`, where `i` is the position of the expression `$` appears in. For example:

op	*rewrite*
`arr[$-1, $-2, 3]`	`arr.opIndex(arr.opDollar!0 - 1, arr.opDollar!1 - 2, 3)`
`arr[1, 2, 3..$]`	`arr.opIndex(1, 2, arr.opSlice!2(3, arr.opDollar!2))`

The intention is that `opDollar!i` should return the length of the array along its `i`'th dimension, or a user-defined object representing the end of the array along that dimension, that is understood by `opSlice` and `opIndex`.

```
struct Rectangle
{
    int width, height;
    int[][] impl;
    this(int w, int h)
    {
        width = w;
        height = h;
        impl = new int[w][h];
    }
    int opIndex(size_t i1, size_t i2)
    {
        return impl[i1][i2];
    }
    int opDollar(size_t pos)()
    {
        static if (pos==0)
            return width;
        else
            return height;
    }
}

void test()
{
    auto r = Rectangle(10,20);
    int i = r[$-1, 0];    // same as: r.opIndex(r.opDollar!0, 0),
```

```
                          // which is r.opIndex(r.width-1, 0)
    int j = r[0, $-1];    // same as: r.opIndex(0, r.opDollar!1)
                          // which is r.opIndex(0, r.height-1)
}
```

As the above example shows, a different compile-time argument is passed to `opDollar` depending on which argument it appears in. A `$` appearing in the first argument gets translated to `opDollar!0`, a `$` appearing in the second argument gets translated to `opDollar!1`, and so on. Thus, the appropriate value for `$` can be returned to implement multidimensional arrays.

Note that `opDollar!i` is only evaluated once for each `i` where `$` occurs in the corresponding position in the indexing operation. Thus, an expression like `arr[$-sqrt($), 0, $-1]` has the effect of:

```
auto __tmp1 = arr.opDollar!0;
auto __tmp2 = arr.opDollar!2;
arr.opIndex(__tmp1 - sqrt(__tmp1), 0, __tmp2 - 1);
```

If `opIndex` is declared with only one argument, the compile-time argument to `opDollar` may be omitted. In this case, it is illegal to use `$` inside an array indexing expression with more than one argument.

20.9 Forwarding

Member names not found in a class or struct can be forwarded to a template function named `opDispatch` for resolution.

```
import std.stdio;

struct S
{
    void opDispatch(string s, T)(T i)
    {
        writefln("S.opDispatch('%s', %s)", s, i);
    }
}

class C
{
    void opDispatch(string s)(int i)
    {
        writefln("C.opDispatch('%s', %s)", s, i);
    }
```

```
}

struct D
{
    template opDispatch(string s)
    {
        enum int opDispatch = 8;
    }
}

void main()
{
    S s;
    s.opDispatch!("hello")(7);
    s.foo(7);

    auto c = new C();
    c.foo(8);

    D d;
    writefln("d.foo = %s", d.foo);
    assert(d.foo == 8);
}
```

20.10 D1 style operator overloading

The D1 operator overload mechanisms are deprecated.

Chapter 21

Templates

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

```
TemplateDeclaration:
    template Identifier TemplateParameters Constraint opt { DeclDefs opt }


TemplateParameters:
    ( TemplateParameterList opt )


TemplateParameterList:
    TemplateParameter
    TemplateParameter ,
    TemplateParameter , TemplateParameterList


TemplateParameter:
    TemplateTypeParameter
    TemplateValueParameter
    TemplateAliasParameter
    TemplateSequenceParameter
    TemplateThisParameter
```

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 sequences. 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. Sequences can contain zero 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.

21.1 Explicit Template Instantiation

Templates are explicitly instantiated with:

```
TemplateInstance:
    Identifier TemplateArguments


TemplateArguments:
    ! ( TemplateArgumentList opt )
    ! TemplateSingleArgument


TemplateArgumentList:
    TemplateArgument
    TemplateArgument ,
    TemplateArgument , TemplateArgumentList


TemplateArgument:
    Type
    AssignExpression
    Symbol


Symbol:
    SymbolTail
    . SymbolTail
```

SymbolTail:
  Identifier
  Identifier . *SymbolTail*
  TemplateInstance
  TemplateInstance . *SymbolTail*

TemplateSingleArgument:
  Identifier
  BasicTypeX
  CharacterLiteral
  StringLiteral
  IntegerLiteral
  FloatLiteral
  `true`
  `false`
  `null`
  `this`
  SpecialKeyword

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

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

If the *TemplateArgument* is one token long, the parentheses can be omitted:

```
TFoo!int.Ptr x;   // same as TFoo!(int).Ptr x;
```

A template instantiation can be aliased:

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

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

```
template TFoo(T) { T f; }
alias a = TFoo!(int);
alias b = TFoo!(int);
```

```
...
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 the same instance. This example uses a `struct` template:

```
struct TFoo(int x) { }

// Different template parameters create different struct types
static assert(!is(TFoo!(3) == TFoo!(2)));
// 3 and 2+1 are both 3 of type int - same TFoo instance
static assert(is(TFoo!(3) == TFoo!(2 + 1)));

// 3u is implicitly converted to 3 to match int parameter,
// and refers to exactly the same instance as TFoo!(3)
static assert(is(TFoo!(3) == TFoo!(3u)));
```

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

Practical Example

A simple generic copy template would be:

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

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

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

See also function templates.

21.2 Instantiation Scope

*TemplateInstance*s are always instantiated 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 f = TFoo!(int); // 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 f = TFoo!(int);
...
f.bar();  // will call a.func(double)
```

TemplateParameter specializations and default values are evaluated in the scope of the *TemplateDeclaration*.

21.3 Argument Deduction

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

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

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

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

For example:

```
template TFoo(T) { }
alias foo1 = TFoo!(int);     // (1) T is deduced to be int
alias foo2 = TFoo!(char*);   // (1) T is deduced to be char*

template TBar(T : T*) { }
alias bar = TBar!(char*);    // (2) T is deduced to be char

template TAbc(D, U : D[]) { }
alias abc1 = TAbc!(int, int[]);  // (2) D is deduced to be int, U is int[]
alias abc2 = TAbc!(char, int[]); // (4) error, D is both char and int

template TDef(D : E*, E) { }
alias def = TDef!(int*, int);  // (1) E is int
                               // (3) D is int*
```

Deduction from a specialization can provide values for more than one parameter:

```
template Foo(T: T[U], U)
{
    ...
}

Foo!(int[long])  // instantiates Foo with T set to int, U set to long
```

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 foo = TFoo!(B);      // (3) T is B

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

21.4 Template Type Parameters

```
TemplateTypeParameter:
    Identifier
    Identifier TemplateTypeParameterSpecialization
    Identifier TemplateTypeParameterDefault
    Identifier TemplateTypeParameterSpecialization TemplateTypeParameterDefault
```

```
TemplateTypeParameterSpecialization:
    : Type
```

```
TemplateTypeParameterDefault:
    = Type
```

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

The template picked to instantiate is the one that is most specialized that fits the types of the *TemplateArgumentList*. If the result is ambiguous, it is an error.

21.5 Template This Parameters

```
TemplateThisParameter:
    this TemplateTypeParameter
```

*TemplateThisParameter*s are used in member function templates to pick up the type of the *this* reference.

```
import std.stdio;

struct S
{
    const void foo(this T)(int i)
    {
        writeln(typeid(T));
    }
}

void main()
{
    const(S) s;
    (&s).foo(1);
    S s2;
    s2.foo(2);
    immutable(S) s3;
    s3.foo(3);
}
```

Prints:

```
const(S)
S
immutable(S)
```

This is especially useful when used with inheritance. For example, you might want to implement a final base method which returns a derived class type. Typically you would return a base type, but this won't allow you to call or access derived properties of the type:

```
interface Addable(T)
{
    final auto add(T t)
    {
        return this;
    }
}

class List(T) : Addable!T
{
```

```
    List remove(T t)
    {
        return this;
    }
}

void main()
{
    auto list = new List!int;
    list.add(1).remove(1);  // error: no 'remove' method for Addable!int
}
```

Here the method `add` returns the base type, which doesn't implement the `remove` method. The `template this` parameter can be used for this purpose:

```
interface Addable(T)
{
    final R add(this R)(T t)
    {
        return cast(R)this;  // cast is necessary, but safe
    }
}

class List(T) : Addable!T
{
    List remove(T t)
    {
        return this;
    }
}

void main()
{
    auto list = new List!int;
    list.add(1).remove(1);  // ok
}
```

21.6 Template Value Parameters

TemplateValueParameter:

```
    BasicType Declarator
    BasicType Declarator TemplateValueParameterSpecialization
    BasicType Declarator TemplateValueParameterDefault
    BasicType Declarator TemplateValueParameterSpecialization TemplateValueParameterDefault
```

```
TemplateValueParameterSpecialization:
    : ConditionalExpression
```

```
TemplateValueParameterDefault:
    = AssignExpression
    = SpecialKeyword
```

Template value parameter types can be any type which can be statically initialized at compile time. Template value arguments can be integer values, floating point values, nulls, string values, array literals of template value arguments, associative array literals of template value arguments, or struct literals of template value arguments.

```
template foo(string s)
{
    string bar() { return s ~ " betty"; }
}

void main()
{
    writefln("%s", foo!("hello").bar()); // prints: hello betty
}
```

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

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

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

21.7 Template Alias Parameters

TemplateAliasParameter:
  alias *Identifier* *TemplateAliasParameterSpecialization*$_{opt}$ *TemplateAliasParameterDefault*$_{opt}$
  alias *BasicType* *Declarator* *TemplateAliasParameterSpecialization*$_{opt}$ *TemplateAliasParameterD*

TemplateAliasParameterSpecialization:
  : *Type*
  : *ConditionalExpression*

TemplateAliasParameterDefault:
  = *Type*
  = *ConditionalExpression*

Alias parameters enable templates to be parameterized with symbol names or values computed at compile-time. Almost any kind of D symbol can be used, including user-defined type names, global names, local names, module names, template names, and template instance names.

Symbol examples:

- User-defined type names

```
class Foo
{
    static int x;
}

template Bar(alias a)
{
    alias sym = a.x;
}

void main()
{
    alias bar = Bar!(Foo);
    bar.sym = 3;  // sets Foo.x to 3
}
```

- Global names

```
shared int x;

template Foo(alias var)
{
    auto ptr = &var;
}

void main()
{
    alias bar = Foo!(x);
    *bar.ptr = 3;       // set x to 3
    static shared int y;
    alias abc = Foo!(y);
    *abc.ptr = 3;       // set y to 3
}
```

- Local names

```
template Foo(alias var)
{
    void inc() { var++; }
}

void main()
{
    int v = 4;
    alias foo = Foo!v;
    foo.inc();
    assert(v == 5);
}
```

See also Implicit Template Nesting.

- Module names

```
import std.conv;

template Foo(alias a)
{
    alias sym = a.text;
}

void main()
```

```
{
    alias bar = Foo!(std.conv);
    bar.sym(3);   // calls std.conv.text(3)
}
```

- Template names

```
shared int x;

template Foo(alias var)
{
    auto ptr = &var;
}

template Bar(alias Tem)
{
    alias instance = Tem!(x);
}

void main()
{
    alias bar = Bar!(Foo);
    *bar.instance.ptr = 3;  // sets x to 3
}
```

- Template instance names

```
shared int x;

template Foo(alias var)
{
    auto ptr = &var;
}

template Bar(alias sym)
{
    alias p = sym.ptr;
}

void main()
{
    alias foo = Foo!(x);
```

```
    alias bar = Bar!(foo);
    *bar.p = 3;  // sets x to 3
}
```

Value examples:

- Literals

```
template Foo(alias x, alias y)
{
    static int i = x;
    static string s = y;
}

void main()
{
    alias foo = Foo!(3, "bar");
    writeln(foo.i, foo.s);  // prints 3bar
}
```

- Compile-time values

```
template Foo(alias x)
{
    static int i = x;
}

void main()
{
    // compile-time argument evaluation
    enum two = 1 + 1;
    alias foo = Foo!(5 * two);
    assert(foo.i == 10);
    static assert(foo.stringof == "Foo!10");

    // compile-time function evaluation
    int get10() { return 10; }
    alias bar = Foo!(get10());
    // bar is the same template instance as foo
    assert(&bar.i is &foo.i);
}
```

- Lambdas

```
template Foo(alias fun)
{
    enum val = fun(2);
}

alias foo = Foo!((int x) => x * x);
static assert(foo.val == 4);
```

Typed alias parameters

Alias parameters can also be typed. These parameters will accept symbols of that type:

```
template Foo(alias int x) { }
int x;
float f;

Foo!x;  // ok
Foo!f;  // fails to instantiate
```

Specialization

Alias parameters can accept both literals and user-defined type symbols, but they are less specialized than the matches to type parameters and value parameters:

```
template Foo(T)         { ... }  // #1
template Foo(int n)     { ... }  // #2
template Foo(alias sym) { ... }  // #3

struct S {}
int var;

alias foo1  = Foo!(S);      // instantiates #1
alias foo2  = Foo!(1);      // instantiates #2
alias foo3a = Foo!([1,2]);  // instantiates #3
alias foo3b = Foo!(var);    // instantiates #3

template Bar(alias A) { ... }                  // #4
template Bar(T : U!V, alias U, V...) { ... }  // #5
```

```
class C(T) {}
alias bar = Bar!(C!int);    // instantiates #5
```

21.8 Template Sequence Parameters

```
TemplateSequenceParameter:
    Identifier ...
```

If the last template parameter in the *TemplateParameterList* is declared as a *TemplateSequenceParameter*, it is a match with any trailing template arguments. Such a sequence of arguments can be defined using the template `std.meta.AliasSeq` and will thus henceforth be referred to by that name for clarity. An *AliasSeq* is not itself a type, value, or symbol. It is a compile-time sequence of any mix of types, values or symbols.

An *AliasSeq* whose elements consist entirely of types is called a type sequence or *TypeSeq*. An *AliasSeq* whose elements consist entirely of values is called a value sequence or *ValueSeq*.

An *AliasSeq* can be used as an argument list to instantiate another template, or as the list of parameters for a function.

```
template print(args...)
{
    void print()
    {
        writeln("args are ", args); // args is a ValueSeq
    }
}

template write(Args...)
{
    void write(Args args) // Args is a TypeSeq
                          // args is a ValueSeq
    {
        writeln("args are ", args);
    }
}

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
```

```
}
```

The number of elements in an *AliasSeq* can be retrieved with the `.length` property. The nth element can be retrieved by indexing the *AliasSeq* with $[n]$, and sub-sequences are denoted by the slicing syntax.

AliasSeq-s are static compile-time entities, there is no way to dynamically change, add, or remove elements either at compile-time or run-time.

Type sequences can be deduced from the trailing parameters of an implicitly instantiated function template:

```
template print(T, Args...)
{
    void print(T first, Args args)
    {
        writeln(first);
        static if (args.length) // if more arguments
            print(args);        // recurse for remaining arguments
    }
}

void main()
{
    print(1, 'a', 6.8);
}
```

prints:

```
1
a
6.8
```

Type sequences can also be deduced from the type of a delegate or function parameter list passed as a function argument:

```
import std.stdio;

/* Partially applies a delegate by tying its first argument to a particular value.
 * R = return type
 * T = first argument type
 * Args = TypeSeq of remaining argument types
 */
R delegate(Args) partial(R, T, Args...)(R delegate(T, Args) dg, T first)
{
```

```
    // return a closure
    return (Args args) => dg(first, args);
}

void main()
{
    int plus(int x, int y, int z)
    {
        return x + y + z;
    }

    auto plus_two = partial(&plus, 2);
    writefln("%d", plus_two(6, 8)); // prints 16
}
```

See also: `std.functional.partial`

Specialization

If both a template with a sequence parameter and a template without a sequence parameter exactly match a template instantiation, the template without a *TemplateSequenceParameter* is selected.

```
template Foo(T)         { pragma(msg, "1"); }   // #1
template Foo(int n)     { pragma(msg, "2"); }   // #2
template Foo(alias sym) { pragma(msg, "3"); }   // #3
template Foo(Args...)   { pragma(msg, "4"); }   // #4

import std.stdio;

// Any sole template argument will never match to #4
alias foo1 = Foo!(int);          // instantiates #1
alias foo2 = Foo!(3);            // instantiates #2
alias foo3 = Foo!(std);          // instantiates #3

alias foo4 = Foo!(int, 3, std);  // instantiates #4
```

21.9 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*
```

21.10 Eponymous Templates

If a template contains members whose name is the same as the template identifier then these members are assumed to be referred to in a template instantiation:

```
template foo(T)
{
    T foo; // declare variable foo of type T
}

void main()
{
    foo!(int) = 6; // instead of foo!(int).foo
}
```

Using functions and more types than the template:

```
template foo(S, T)
{
    // each member contains all the template parameters
    void foo(S s, T t) {}
    void foo(S s, T t, string) {}
}

void main()
{
    foo(1, 2, "test"); // foo!(int, int).foo(1, 2, "test")
    foo(1, 2); // foo!(int, int).foo(1, 2)
}
```

When the template parameters must be deduced, the eponymous members can't rely on a `static if` condition since the deduction relies on how the in members are used:

```
template foo(T)
{
    static if (is(T)) // T is not yet known...
```

```
        void foo(T t) {} // T is deduced from the member usage
}

void main()
{
    foo(0); // Error: cannot deduce function from argument types
    foo!int(0); // Ok since no deduction necessary
}
```

21.11 Template Constructors

ConstructorTemplate:
    `this` *TemplateParameters* *Parameters* *MemberFunctionAttributes*$_{\text{opt}}$ *Constraint*$_{\text{opt}}$ `:`
    `this` *TemplateParameters* *Parameters* *MemberFunctionAttributes*$_{\text{opt}}$ *Constraint*$_{\text{opt}}$ *FunctionBody*

Templates can be used to form constructors for classes and structs.

21.12 Aggregate Templates

ClassTemplateDeclaration:
    `class` *Identifier* *TemplateParameters* `;`
    `class` *Identifier* *TemplateParameters* *Constraint*$_{\text{opt}}$ *BaseClassList*$_{\text{opt}}$ *AggregateBody*
    `class` *Identifier* *TemplateParameters* *BaseClassList*$_{\text{opt}}$ *Constraint*$_{\text{opt}}$ *AggregateBody*

InterfaceTemplateDeclaration:
    `interface` *Identifier* *TemplateParameters* `;`
    `interface` *Identifier* *TemplateParameters* *Constraint*$_{\text{opt}}$ *BaseInterfaceList*$_{\text{opt}}$ *AggregateBody*
    `interface` *Identifier* *TemplateParameters* *BaseInterfaceList* *Constraint* *AggregateBody*

StructTemplateDeclaration:
    `struct` *Identifier* *TemplateParameters* `;`
    `struct` *Identifier* *TemplateParameters* *Constraint*$_{\text{opt}}$ *AggregateBody*

UnionTemplateDeclaration:
    `union` *Identifier* *TemplateParameters* `;`

```
    union Identifier TemplateParameters Constraintopt AggregateBody
```

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;
}
```

Analogously to class templates, struct, union and interfaces can be transformed into templates by supplying a template parameter list.

21.13 Function Templates

If a template declares exactly one member, and that member is a function with the same name as the template, it is a function template declaration. Alternatively, a function template declaration is a function declaration with a *TemplateParameterList* immediately preceding the *Parameters*.

A function template to compute the square of type T is:

```
T Square(T)(T t)
{
    return t * t;
}
```

It is lowered to:

```
template Square(T)
{
    T Square(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
```

If there are fewer arguments supplied in the *TemplateArgumentList* than parameters in the *TemplateParameterList*, the arguments fulfill parameters from left to right, and the rest of the parameters are then deduced from the function arguments.

The process of deducing template type parameters from function arguments is called Implicit Function Template Instantiation (IFTI).

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*
```

If template type parameters match the literal expressions on function arguments, the deduced types may consider narrowing conversions of them.

```
void foo(T)(T v)        { pragma(msg, "in␣foo,␣T␣=␣", T); }
void bar(T)(T v, T[] a) { pragma(msg, "in␣bar,␣T␣=␣", T); }

foo(1);
// an integer literal type is analyzed as int by default
// then T is deduced to int

short[] arr;
bar(1, arr);
// arr is short[], and the interger literal 1 is
// implicitly convertible to short.
```

```
// then T will be deduced to short.

bar(1, [2.0, 3.0]);
// the array literal is analyzed as double[],
// and the interger literal 1 is implicitly convertible to double.
// then T will be deduced to double.
```

The deduced type parameter for dynamic array and pointer arguments has an unqualified head:

```
void foo(T)(T arg) { pragma(msg, T); }

int[] marr;
const(int[]) carr;
immutable(int[]) iarr;
foo(marr);  // T == int[]
foo(carr);  // T == const(int)[]
foo(iarr);  // T == immutable(int)[]

int* mptr;
const(int*) cptr;
immutable(int*) iptr;
foo(mptr);  // T == int*
foo(cptr);  // T == const(int)*
foo(iptr);  // T == immutable(int)*
```

Type parameter deduction is not influenced by the order of function arguments.

Function templates can have their return types deduced based on the *ReturnStatements* in the function, just as with normal functions. See Auto Functions.

```
auto Square(T)(T t)
{
    return t * t;
}
```

Variadic Function Templates can have parameters with default values. These parameters are always set to their default value in case of IFTI.

```
size_t fun(T...)(T t, string file = __FILE__)
{
    import std.stdio;
    writeln(file, "␣", t);
    return T.length;
}
```

```
assert(fun(1, "foo") == 2);  // uses IFTI
assert(fun!int(1, "foo") == 1);  // no IFTI
```

21.14 Variable Templates

Same as aggregates and functions, variable declarations with *Initializer* can have optional template parameters:

```
enum string constant(TL...) = TL.stringof;
ubyte[T.sizeof] storage(T) = 0;
auto array(alias a) = a;
```

These declarations are transformed into templates:

```
template constant(TL...)
{
    enum string constant = TL.stringof;
}
template storage(T)
{
    ubyte[T.sizeof] storage = 0;
}
template array(alias a)
{
    auto array = a;
}
```

21.15 Alias Templates

AliasDeclaration can also have optional template parameters:

```
alias Sequence(TL...) = TL;
```

It is lowered to:

```
template Sequence(TL...)
{
    alias Sequence = TL;
}
```

Function Templates with Auto Ref Parameters

An auto ref function template parameter becomes a ref parameter if its corresponding argument is an lvalue, otherwise it becomes a value parameter:

```
int foo(Args...)(auto ref Args args)
{
    int result;

    foreach (i, v; args)
    {
        if (v == 10)
            assert(__traits(isRef, args[i]));
        else
            assert(!__traits(isRef, args[i]));
        result += v;
    }
    return result;
}

void main()
{
    int y = 10;
    int r;
    r = foo(8);       // returns 8
    r = foo(y);       // returns 10
    r = foo(3, 4, y); // returns 17
    r = foo(4, 5, y); // returns 19
    r = foo(y, 6, y); // returns 26
}
```

Auto ref parameters can be combined with auto ref return attributes:

```
auto ref min(T, U)(auto ref T lhs, auto ref U rhs)
{
    return lhs > rhs ? rhs : lhs;
}

void main()
{
    int x = 7, y = 8;
    int i;
```

```
    i = min(4, 3);      // returns 3
    i = min(x, y);      // returns 7
    min(x, y) = 10;     // sets x to 10
    static assert(!__traits(compiles, min(3, y) = 10));
    static assert(!__traits(compiles, min(y, 3) = 10));
}
```

21.16 Nested Templates

If a template is declared in aggregate or function local scope, the instantiated functions will implicitly capture the context of the enclosing scope.

```
class C
{
    int num;

    this(int n) { num = n; }

    template Foo()
    {
        // 'foo' can access 'this' reference of class C object.
        void foo(int n) { this.num = n; }
    }
}

void main()
{
    auto c = new C(1);
    assert(c.num == 1);

    c.Foo!().foo(5);
    assert(c.num == 5);

    template Bar()
    {
        // 'bar' can access local variable of 'main' function.
        void bar(int n) { c.num = n; }
    }
    Bar!().bar(10);
```

```
    assert(c.num == 10);
}
```

Above, `Foo!().foo` will work just the same as a member function of class `C`, and `Bar!().bar` will work just the same as a nested function within function `main()`.

Implicit Nesting

If a template has a template alias parameter, and is instantiated with a local symbol, the instantiated function will implicitly become nested in order to access runtime data of the given local symbol.

```
template Foo(alias sym)
{
    void foo() { sym = 10; }
}

class C
{
    int num;

    this(int n) { num = n; }

    void main()
    {
        assert(this.num == 1);

        alias fooX = Foo!(C.num).foo;

        // fooX will become member function implicitly, so &fooX
        //     returns a delegate object.
        static assert(is(typeof(&fooX) == delegate));

        fooX(); // called by using valid 'this' reference.
        assert(this.num == 10);  // OK
    }
}

void main()
{
    new C(1).main();
```

```
    int num;
    alias fooX = Foo!num.foo;

    // fooX will become nested function implicitly, so &fooX
    //     returns a delegate object.
    static assert(is(typeof(&fooX) == delegate));

    fooX();
    assert(num == 10);  // OK
}
```

Not only functions, but also instantiated class and struct types can become nested via implicitly captured context.

```
class C
{
    int num;
    this(int n) { num = n; }

    class N(T)
    {
        // instantiated class N!T can become nested in C
        T foo() { return num * 2; }
    }
}

void main()
{
    auto c = new C(10);
    auto n = c.new N!int();
    assert(n.foo() == 20);
}

void main()
{
    int num = 10;
    struct S(T)
    {
        // instantiated struct S!T can become nested in main()
        T foo() { return num * 2; }
    }
```

```
    S!int s;
    assert(s.foo() == 20);
}
```

A templated `struct` can become a nested `struct` if it is instantiated with a local symbol passed as an aliased argument:

```
struct A(alias F)
{
    int fun(int i) { return F(i); }
}

A!F makeA(alias F)() { return A!F(); }

void main()
{
    int x = 40;
    int fun(int i) { return x + i; }
    A!fun a = makeA!fun();
    assert(a.fun(2) == 42);
}
```

Context Limitation

Currently nested templates can capture at most one context. As a typical example, non-static template member functions cannot take local symbol by using template alias parameter.

```
class C
{
    int num;
    void foo(alias sym)() { num = sym * 2; }
}

void main()
{
    auto c = new C();
    int var = 10;
    c.foo!var();    // NG, foo!var requires two contexts, 'this' and 'main()'
}
```

But, if one context is indirectly accessible from other context, it is allowed.

```
int sum(alias x, alias y)() { return x + y; }

void main()
{
    int a = 10;
    void nested()
    {
        int b = 20;
        assert(sum!(a, b)() == 30);
    }
    nested();
}
```

Two local variables `a` and `b` are in different contexts, but outer context is indirectly accessible from innter context, so nested template instance `sum!(a, b)` will capture only inner context.

21.17 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
}
```

21.18 Template Constraints

```
Constraint:
    if ( Expression )
```

*Constraint*s are used to impose additional constraints on matching arguments to a template beyond what is possible in the *TemplateParameterList*. The *Expression* is computed at compile time and returns a result that is converted to a boolean value. If that value is true, then the template is matched, otherwise the template is not matched.

For example, the following function template only matches with odd values of `N`:

```
void foo(int N)()
    if (N & 1)
{
    ...
}
...
foo!(3)();  // OK, matches
foo!(4)();  // Error, no match
```

Template constraints can be used with aggregate types (structs, classes, unions). Constraints are effectively used with library module "std.traits":

```
import std.traits;

struct Bar(T)
    if (isIntegral!T)
{
    ...
}
...
auto x = Bar!int;       // OK, int is an integral type
auto y = Bar!double;    // Error, double does not satisfy constraint
```

21.19 Limitations

Templates cannot be used to add non-static fields or virtual functions to classes or interfaces. For example:

```
class Foo
{
    template TBar(T)
    {
        T xx;               // becomes a static field of Foo
```

```
        int func(T) { ... } // non-virtual

        static T yy;                      // Ok
        static int func(T t, int y) { ... } // Ok
    }
}
```

Chapter 22

Template Mixins

A *TemplateMixin* takes an arbitrary set of declarations from the body of a *TemplateDeclaration* and inserts them into the current context.

```
TemplateMixinDeclaration:
    mixin template Identifier TemplateParameters Constraint_opt { DeclDefs_opt }
```

```
TemplateMixin:
    mixin MixinTemplateName TemplateArguments_opt Identifier_opt ;
```

```
MixinTemplateName:
    . QualifiedIdentifierList
    QualifiedIdentifierList
    Typeof . QualifiedIdentifierList
```

```
QualifiedIdentifierList:
    Identifier
    Identifier . QualifiedIdentifierList
    TemplateInstance . QualifiedIdentifierList
```

A *TemplateMixin* can occur in declaration lists of modules, classes, structs, unions, and as a statement. The *MixinTemplateName* refers to a *TemplateDeclaration* or *TemplateMixinDeclaration*. 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 into a nested scope. It is useful for injecting parameterized 'boilerplate' code, as well as for creating templated nested functions, which is not possible with template instantiations.

```
mixin template Foo()
{
    int x = 5;
}

mixin Foo;

struct Bar
{
    mixin Foo;
}

void test()
{
    writefln("x = %d", x);          // prints 5
    {
        Bar b;
        int x = 3;

        writefln("b.x = %d", b.x); // prints 5
        writefln("x = %d", x);     // prints 3
        {
            mixin Foo;
            writefln("x = %d", x); // prints 5
            x = 4;
            writefln("x = %d", x); // prints 4
        }
        writefln("x = %d", x);     // prints 3
    }
    writefln("x = %d", x);          // prints 5
}
```

Mixins can be parameterized:

```
mixin template Foo(T)
{
```

```
    T x = 5;
}

mixin Foo!(int);            // create x of type int
```

Mixins can add virtual functions to a class:

```
mixin template Foo()
{
    void func() { writeln("Foo.func()"); }
}

class Bar
{
    mixin Foo;
}

class Code : Bar
{
    override void func() { writeln("Code.func()"); }
}

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;

mixin 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:

```
mixin 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:

```
mixin 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(); goto case;
                case 7:      s(); goto case;
                case 6:      s(); goto case;
                case 5:      s(); goto case;
                case 4:      s(); goto case;
                case 3:      s(); goto case;
                case 2:      s(); goto case;
                case 1:      s(); continue;
                default:     assert(0, "Impossible");
                        } while (--n > 0);
```

```
            }
        }
    }
}

void foo() { writeln("foo"); }

void test()
{
    int i = 1;
    int j = 11;

    mixin duffs_device!(i, j, delegate { foo(); });
    duff_loop();  // executes foo() 10 times
}
```

22.1 Mixin Scope

The declarations in a mixin are placed in a nested scope and then '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;

mixin template Foo()
{
    int x = 5;
    int y = 5;
}

mixin Foo;
int y = 3;

void test()
{
    writefln("x = %d", x);  // prints 3
    writefln("y = %d", 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:

```
mixin template Foo()
{
    int x = 5;
    void func(int x) { }
}

mixin template Bar()
{
    int x = 4;
    void func(long x) { }
}

mixin Foo;
mixin Bar;

void test()
{
    import std.stdio : writefln;
    writefln("x = %d", x); // error, x is ambiguous
    func(1);               // error, func is ambiguous
}
```

The call to `func()` is ambiguous because Foo.func and Bar.func are in different scopes.

If a mixin has an *Identifier*, it can be used to disambiguate between conflicting symbols:

```
int x = 6;

mixin template Foo()
{
    int x = 5;
    int y = 7;
    void func() { }
}

mixin template Bar()
{
    int x = 4;
    void func() { }
}
```

```
mixin Foo F;
mixin Bar B;

void test()
{
    writefln("y = %d", y);     // prints 7
    writefln("x = %d", x);     // prints 6
    writefln("F.x = %d", F.x); // prints 5
    writefln("B.x = %d", 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:

```
mixin template Foo()
{
    void func(int x) {  }
}

mixin template Bar()
{
    void func(long x) {  }
}

mixin Foo!() F;
mixin Bar!() B;

alias func = F.func;
alias func = B.func;

void main()
{
    func(1);  // calls B.func
    func(1L); // calls F.func
}
```

A mixin has its own scope, even if a declaration is overridden by the enclosing one:

```
int x = 4;
```

```
mixin template Foo()
{
    int x = 5;
    int bar() { return x; }
}

mixin Foo;

void test()
{
    writefln("x␣=␣%d", x);         // prints 4
    writefln("bar()␣=␣%d", bar()); // prints 5
}
```

Chapter 23

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.

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

<img src="images/d4.gif" style="max-width:100

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.

23.1 Assert Contract

The most basic contract is the *AssertExpression*. An **assert** declares an expression that must evaluate to true, with an optional failure string as a second argument:

```
assert(expression);
assert(expression, "message to display on failure");
```

As a contract, an `assert` represents a guarantee that the code *must* uphold. Any failure of this expression represents a logic error in the code that must be fixed in the source code. A program for which the assert contract is false is, by definition, invalid.

As a debugging aid, the compiler may insert a runtime check to verify that the expression is indeed true. If it is false, an `AssertError` is thrown. When compiling for release, this check is not generated. The special `assert(0)` expression, however, is generated even in release mode. See the *AssertExpression* documentation for more information.

The compiler is free to assume the assert expression is true and optimize subsequent code accordingly.

Undefined Behavior: The subsequent execution of the program after an assert contract is false.

23.2 Pre and Post Contracts

The pre contracts specify the preconditions before a statement is executed. The most typical use of this would be in validating the parameters to a function. The post contracts validate the result of the statement. The most typical use of this would be in validating the return value of a function and of any side effects it has. In D, pre contracts begin with `in`, and post contracts begin with `out`. They come at the end of the function signature and before the opening brace of the function body.

Pre and post contracts can be written either in expression form (feature introduced in DMD 2.081.0), with a syntax similar to **assert**, or as block statements containing arbitrary code.

The expression form is:

```
in (expression)
in (expression, "failure␣string")
out (identifier; expression)
out (identifier; expression, "failure␣string")
out (; expression)
out (; expression, "failure␣string")
{
    ...function body...
}
```

The block statement form is:

```
in
{
    ...contract preconditions...
}
out
{
```

```
    ...contract postconditions...
}
out (identifier)
{
    ...contract postconditions...
}
do
{
    ...function body...
}
```

The optional identifier in either type of `out` contract is set to the return value of the function.

By definition, if a pre contract fails, then the function received bad parameters. If a post contract fails, then there is a bug in the function. In either case, an `assert` statement within the corresponding `in` or `out` block will throw an `AssertError`.

The keyword `do` can be used to announce the function body. Although any number of pre or post contracts of any form may follow each other, `do` is required only when the last contract before the body is a block statement. (Before the acceptance of DIP1003, the keyword `body` was required instead of `do`, and may still be encountered in old code bases. In the long term, `body` may be deprecated, but for now it's allowed both as a keyword in this context and as an identifier elsewhere, although `do` is preferred.)

Though any arbitrary D code is allowed in the `in` and `out` contract blocks, their only function should be to verify incoming and outgoing data. 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, `in` and `out` contracts are not inserted.

Here is an example function in both forms:

```
int fun(ref int a, int b)
in (a > 0)
in (b >= 0, "b cannot be negative!")
out (r; r > 0, "return must be positive")
out (; a != 0)
{
    // function body
}

int fun(ref int a, int b)
in
{
    assert(a > 0);
    assert(b >= 0, "b cannot be negative!");
```

```
}
out (r)
{
    assert(r > 0, "return␣must␣be␣positive");
    assert(a != 0);
}
do
{
    // function body
}
```

The two functions are almost identical semantically. The expressions in the first are lowered to contract blocks that look almost exactly like the second, except that a separate block is created for each expression in the first, thus avoiding shadowing variable names.

23.3 In, Out and Inheritance

If a function in a derived class overrides a function from 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 any `in` contracts on functions overriding it have no useful effect.

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

23.4 Invariants

Invariants are used to specify characteristics of a class or struct that must always 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;

    this(int d, int h)
    {
        day = d;
```

```
        hour = h;
    }

    invariant
    {
        assert(1 <= day && day <= 31);
        assert(0 <= hour && hour < 24, "hour␣out␣of␣bounds");
    }
}
```

Invariant blocks should contain `assert` expressions, and should throw `AssertErrors` when they fail. Since DMD version 2.081.0, invariants can also be written as expression statements, with `assert` implied:

```
    invariant (1 <= day && day <= 31);
    invariant (0 <= hour && hour < 24, "hour␣out␣of␣bounds");
```

The invariant is a contract saying that the `asserts` must hold true. The invariant is checked when a class or struct constructor completes, and at the start of the class or struct destructor. For public or exported functions, the order of execution is:

1. preconditions
2. invariant
3. function body
4. invariant
5. postconditions

The invariant is not checked if the class or struct is implicitly constructed using the default `.init` value.

The code in the invariant may not call any public non-static members of the class or struct, 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.

Invariants are implicitly `const`.

Since the invariant is called at the start of public or exported members, such members should not be called from constructors.

```
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 with an `assert()` expression:

1. classes need to pass a class object
2. structs need to pass the address of an instance

```
auto mydate = new Date(); //class
auto s = S();             //struct
...
assert(mydate);           // check that class Date invariant holds
assert(&s);               // check that struct S invariant holds
```

Class invariants are inherited, that is, any class invariant is implicitly in addition to the invariants of its base classes.

There can be more than one invariant declared per class or struct.

When compiling for release, the invariant code is not generated, and the compiled program runs at maximum speed. The compiler is free to assume the invariant holds true, regardless of whether code is generated for it or not, and may optimize code accordingly.

23.5 References

- Contracts Reading List
- Adding Contracts to Java

Chapter 24

Conditional Compilation

Conditional compilation is the process of selecting which code to compile and which code to not compile.

```
ConditionalDeclaration:
    Condition DeclarationBlock
    Condition DeclarationBlock else DeclarationBlock
    Condition : DeclDefs opt
    Condition DeclarationBlock else : DeclDefs opt
```

```
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

24.1 Version Condition

VersionCondition:
    `version ( IntegerLiteral )`
    `version ( Identifier )`
    `version ( unittest )`
    `version ( assert )`

Versions enable multiple versions of a module to be implemented with a single source file.

The *VersionCondition* is satisfied if the *IntegerLiteral* 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
}
```

The `version(unittest)` is satisfied if and only if the code is compiled with unit tests enabled (the `-unittest` option on `dmd`).

24.2 Version Specification

```
VersionSpecification:
    version = Identifier ;
    version = IntegerLiteral ;
```

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
```

VersionSpecifications may only appear at module scope.

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.

Table 24.1: Predefined Version Identifiers

Version Identifier	Description
`DigitalMars`	DMD (Digital Mars D) is the compiler
`GNU`	GDC (GNU D Compiler) is the compiler
`LDC`	LDC (LLVM D Compiler) is the compiler
`SDC`	SDC (Stupid D Compiler) is the compiler
`Windows`	Microsoft Windows systems
`Win32`	Microsoft 32-bit Windows systems
`Win64`	Microsoft 64-bit Windows systems
`linux`	All Linux systems
`OSX`	macOS
`iOS`	iOS
`TVOS`	tvOS
`WatchOS`	watchOS
`FreeBSD`	FreeBSD
`OpenBSD`	OpenBSD
`NetBSD`	NetBSD
`DragonFlyBSD`	DragonFlyBSD
`BSD`	All other BSDs
`Solaris`	Solaris
`Posix`	All POSIX systems (includes Linux, FreeBSD, OS X, Solaris, etc.)
`AIX`	IBM Advanced Interactive eXecutive OS
`Haiku`	The Haiku operating system
`SkyOS`	The SkyOS operating system
`SysV3`	System V Release 3
`SysV4`	System V Release 4
`Hurd`	GNU Hurd
`Android`	The Android platform
`Emscripten`	The Emscripten platform
`PlayStation`	The PlayStation platform
`PlayStation4`	The PlayStation 4 platform
`Cygwin`	The Cygwin environment
`MinGW`	The MinGW environment
`FreeStanding`	An environment without an operating system (such as Bare-metal targets)
`CRuntime_Bionic`	Bionic C runtime
`CRuntime_DigitalMars`	DigitalMars C runtime

↩

(continued)

Version Identifier	Description
`CRuntime_Glibc`	Glibc C runtime
`CRuntime_Microsoft`	Microsoft C runtime
`CRuntime_Musl`	musl C runtime
`CRuntime_UClibc`	uClibc C runtime
`CRuntime_WASI`	WASI C runtime
`X86`	Intel and AMD 32-bit processors
`X86_64`	Intel and AMD 64-bit processors
`ARM`	The ARM architecture (32-bit) (AArch32 et al)
`ARM_Thumb`	ARM in any Thumb mode
`ARM_SoftFloat`	The ARM `soft` floating point ABI
`ARM_SoftFP`	The ARM `softfp` floating point ABI
`ARM_HardFloat`	The ARM `hardfp` floating point ABI
`AArch64`	The Advanced RISC Machine architecture (64-bit)
`AsmJS`	The asm.js intermediate programming language
`Epiphany`	The Epiphany architecture
`PPC`	The PowerPC architecture, 32-bit
`PPC_SoftFloat`	The PowerPC soft float ABI
`PPC_HardFloat`	The PowerPC hard float ABI
`PPC64`	The PowerPC architecture, 64-bit
`IA64`	The Itanium architecture (64-bit)
`MIPS32`	The MIPS architecture, 32-bit
`MIPS64`	The MIPS architecture, 64-bit
`MIPS_O32`	The MIPS O32 ABI
`MIPS_N32`	The MIPS N32 ABI
`MIPS_O64`	The MIPS O64 ABI
`MIPS_N64`	The MIPS N64 ABI
`MIPS_EABI`	The MIPS EABI
`MIPS_SoftFloat`	The MIPS `soft-float` ABI
`MIPS_HardFloat`	The MIPS `hard-float` ABI
`NVPTX`	The Nvidia Parallel Thread Execution (PTX) architecture, 32-bit
`NVPTX64`	The Nvidia Parallel Thread Execution (PTX) architecture, 64-bit
`RISCV32`	The RISC-V architecture, 32-bit
`RISCV64`	The RISC-V architecture, 64-bit
`SPARC`	The SPARC architecture, 32-bit

↩

(continued)

Version Identifier	Description
`SPARC_V8Plus`	The SPARC v8+ ABI
`SPARC_SoftFloat`	The SPARC soft float ABI
`SPARC_HardFloat`	The SPARC hard float ABI
`SPARC64`	The SPARC architecture, 64-bit
`S390`	The System/390 architecture, 32-bit
`SystemZ`	The System Z architecture, 64-bit
`HPPA`	The HP PA-RISC architecture, 32-bit
`HPPA64`	The HP PA-RISC architecture, 64-bit
`SH`	The SuperH architecture, 32-bit
`WebAssembly`	The WebAssembly virtual ISA (instruction set architecture), 32-bit
`WASI`	The WebAssembly System Interface
`Alpha`	The Alpha architecture
`Alpha_SoftFloat`	The Alpha soft float ABI
`Alpha_HardFloat`	The Alpha hard float ABI
`LittleEndian`	Byte order, least significant first
`BigEndian`	Byte order, most significant first
`ELFv1`	The Executable and Linkable Format v1
`ELFv2`	The Executable and Linkable Format v2
`D_BetterC`	D as Better C code (command line switch `-betterC`) is being generated
`D_Exceptions`	Exception handling is supported. Evaluates to `false` when compiling with command line switch `-betterC`
`D_ModuleInfo`	`ModuleInfo` is supported. Evaluates to `false` when compiling with command line switch `-betterC`
`D_TypeInfo`	Runtime type information (a.k.a `TypeInfo`) is supported. Evaluates to `false` when compiling with command line switch `-betterC`
`D_Coverage`	Code coverage analysis instrumentation (command line switch `-cov`) is being generated
`D_Ddoc`	Ddoc documentation (command line switch `-D`) is being generated
`D_InlineAsm_X86`	Inline assembler for X86 is implemented
`D_InlineAsm_X86_64`	Inline assembler for X86-64 is implemented
`D_LP64`	**Pointers** are 64 bits (command line switch `-m64`). (Do not confuse this with C's LP64 model)

↩

(continued)

Version Identifier	Description
D_X32	Pointers are 32 bits, but words are still 64 bits (x32 ABI) (This can be defined in parallel to X86_64)
D_HardFloat	The target hardware has a floating point unit
D_SoftFloat	The target hardware does not have a floating point unit
D_PIC	Position Independent Code (command line switch -fPIC) is being generated
D_SIMD	Vector extensions (via __simd) are supported
D_AVX	AVX Vector instructions are supported
D_AVX2	AVX2 Vector instructions are supported
D_Version2	This is a D version 2 compiler
D_NoBoundsChecks	Array bounds checks are disabled (command line switch -boundscheck=off)
D_ObjectiveC	The target supports interfacing with Objective-C
Core	Defined when building the standard runtime
Std	Define when building the standard library
unittest	Unit tests are enabled (command line switch -unittest)
assert	Checks are being emitted for *AssertExpressions*
none	Never defined; used to just disable a section of code
all	Always defined; used as the opposite of none

The following identifiers are defined, but are deprecated:

Predefined Version Identifiers (deprecated)

Version Identifier	Description	
darwin	The Darwin operating system; use OSX instead	
Thumb	ARM in Thumb mode; use ARM_Thumb instead	
S390X	The System/390X architecture	64-bit; use SystemZ instead

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. Further, all identifiers derived from the ones listed above by appending any character(s) are reserved. This means that e.g. ARM_foo and Windows_bar are reserved while foo_ARM and bar_Windows are not.

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.

24.3 Debug Condition

```
DebugCondition:
    debug
    debug ( IntegerLiteral )
    debug ( Identifier )
```

Two versions of programs are commonly built, a release build and a debug build. The debug build includes extra error checking code, test harnesses, pretty-printing code, etc. The debug statement conditionally compiles in its statement body. It is D's way of what in C is done with `#ifdef DEBUG` / `#endif` pairs.

The `debug` condition is satisfied when the `-debug` switch is passed to the compiler or when the debug level is $>= 1$.

The `debug ( `*IntegerLiteral*` )` condition is satisfied when the debug level is `>=` *IntegerLiteral*.

The `debug ( `*Identifier*` )` condition is satisfied when the debug identifier matches *Identifier*.

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

Debug Statement

A *ConditionalStatement* that has a *DebugCondition* is called a *DebugStatement*. *DebugStatements* have relaxed semantic checks in that `pure`, `@nogc`, `nothrow` and `@safe` checks are not done. Neither do *DebugStatements* influence the inference of `pure`, `@nogc`, `nothrow` and `@safe` attributes.

Undefined Behavior: Since these checks are bypassed, it is up to the programmer to ensure the code is correct. For example, throwing an exception in a `nothrow` function is undefined behavior.

Best Practices: This enables the easy insertion of code to provide debugging help, by bypassing the otherwise stringent attribute checks. Never ship release code that has *DebugStatements* enabled.

24.4 Debug Specification

```
DebugSpecification:
    debug = Identifier ;
    debug = IntegerLiteral ;
```

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) writeln("Foo");
debug = foo;    // error, foo used before set
```

DebugSpecifications may only appear at module scope.

Various different debug builds can be built with a parameter to debug:

```
debug(IntegerLiteral) { } // add in debug code if debug level is >= IntegerLiteral
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*.

24.5 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.

StaticIfConditions 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 INT = short;
    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.

24.6 Static Foreach

```
StaticForeach:
    static AggregateForeach
    static RangeForeach


StaticForeachDeclaration:
    StaticForeach DeclarationBlock
    StaticForeach : DeclDefs opt


StaticForeachStatement:
    StaticForeach NoScopeNonEmptyStatement
```

The aggregate/range bounds are evaluated at compile time and turned into a sequence of compile-time entities by evaluating corresponding code with a *ForeachStatement*/*ForeachRangeStatement* at compile time. The body of the `static foreach` is then copied a number of times that corresponds to the number of elements of the sequence. Within the i-th copy, the name of the `static foreach` variable is bound to the i-th entry of the sequence, either as an `enum` variable declaration (for constants) or an `alias` declaration (for symbols). (In particular, `static foreach` variables are never runtime variables.)

```
static foreach(i; [0, 1, 2, 3])
{
    pragma(msg, i);
}
```

`static foreach` supports multiple variables in cases where the corresponding `foreach` statement supports them. (In this case, `static foreach` generates a compile-time sequence of tuples, and the tuples are subsequently unpacked during iteration.)

```
static foreach(i, v; ['a', 'b', 'c', 'd'])
{
    static assert(i + 'a' == v);
}
```

Like bodies of *ConditionalDeclarations*, a `static foreach` body does not introduce a new scope. Therefore, it can be used to generate declarations:

```
import std.range : iota;
import std.algorithm : map;
import std.conv : text;
static foreach(i; iota(0, 3).map!text)
{
    mixin(`enum x` ~ i ~ ` = i;`);
}

pragma(msg, x0, "␣", x1,"␣", x2); // 0 1 2
```

As `static foreach` is a code generation construct and not a loop, `break` and `continue` cannot be used to change control flow within it. Instead of breaking or continuing a suitable enclosing statement, such an usage yields an error (this is to prevent misunderstandings).

```
int test(int x)
{
    int r = -1;
    switch(x)
    {
        static foreach(i; 0 .. 100)
        {
            case i:
                r = i;
                break; // error
        }
        default: break;
    }
    return r;
}

static foreach(i; 0 .. 200)
{
    static assert(test(i) == (i<100 ? i : -1));
}
```

An explicit `break`/`continue` label can be used to avoid this limitation. (Note that `static foreach` itself cannot be broken nor continued even if it is explicitly labeled.)

```
int test(int x)
{
```

```
    int r = -1;
    Lswitch: switch(x)
    {
        static foreach(i; 0 .. 100)
        {
            case i:
                r = i;
                break Lswitch;
        }
        default: break;
    }
    return r;
}

static foreach(i; 0 .. 200)
{
    static assert(test(i) == (i<100 ? i : -1));
}
```

24.7 Static Assert

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

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 *AssertExpressions*, *StaticAsserts* 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 *AssignExpression* can be used to supply additional information, such as a text string, that will be printed out along with the error diagnostic.

Chapter 25

Traits

Traits are extensions to the language to enable programs, at compile time, to get at information internal to the compiler. This is also known as compile time reflection. It is done as a special, easily extended syntax (similar to Pragmas) so that new capabilities can be added as required.

```
TraitsExpression:
    __traits ( TraitsKeyword , TraitsArguments )


TraitsKeyword:
    isAbstractClass
    isArithmetic
    isAssociativeArray
    isFinalClass
    isPOD
    isNested
    isFuture
    isDeprecated
    isFloating
    isIntegral
    isScalar
    isStaticArray
    isUnsigned
    isDisabled
    isVirtualFunction
    isVirtualMethod
    isAbstractFunction
    isFinalFunction
```

isStaticFunction
isOverrideFunction
isTemplate
isRef
isOut
isLazy
isReturnOnStack
isZeroInit
isModule
isPackage
hasMember
hasCopyConstructor
hasPostblit
identifier
getAliasThis
getAttributes
getFunctionAttributes
getFunctionVariadicStyle
getLinkage
getLocation
getMember
getOverloads
getParameterStorageClasses
getPointerBitmap
getProtection
getTargetInfo
getVirtualFunctions
getVirtualMethods
getUnitTests
parent
classInstanceSize
getVirtualIndex
allMembers
derivedMembers
isSame
compiles

TraitsArguments:
TraitsArgument

TraitsArgument , *TraitsArguments*

TraitsArgument:
AssignExpression
Type

25.1 *isArithmetic*

If the arguments are all either types that are arithmetic types, or expressions that are typed as arithmetic types, then `true` is returned. Otherwise, `false` is returned. If there are no arguments, `false` is returned.

```
import std.stdio;

void main()
{
    int i;
    writeln(__traits(isArithmetic, int));
    writeln(__traits(isArithmetic, i, i+1, int));
    writeln(__traits(isArithmetic));
    writeln(__traits(isArithmetic, int*));
}
```

Prints:

```
true
true
false
false
```

25.2 *isFloating*

Works like `isArithmetic`, except it's for floating point types (including imaginary and complex types).

```
import core.simd : float4;

enum E : float { a, b }
```

```
static assert(__traits(isFloating, float));
static assert(__traits(isFloating, idouble));
static assert(__traits(isFloating, creal));
static assert(__traits(isFloating, E));
static assert(__traits(isFloating, float4));

static assert(!__traits(isFloating, float[4]));
```

25.3 *isIntegral*

Works like `isArithmetic`, except it's for integral types (including character types).

```
import core.simd : int4;

enum E { a, b }

static assert(__traits(isIntegral, bool));
static assert(__traits(isIntegral, char));
static assert(__traits(isIntegral, int));
static assert(__traits(isIntegral, E));
static assert(__traits(isIntegral, int4));

static assert(!__traits(isIntegral, float));
static assert(!__traits(isIntegral, int[4]));
static assert(!__traits(isIntegral, void*));
```

25.4 *isScalar*

Works like `isArithmetic`, except it's for scalar types.

```
import core.simd : int4, void16;

enum E { a, b }

static assert(__traits(isScalar, bool));
static assert(__traits(isScalar, char));
static assert(__traits(isScalar, int));
static assert(__traits(isScalar, float));
static assert(__traits(isScalar, E));
```

```
static assert(__traits(isScalar, int4));
static assert(__traits(isScalar, void*)); // Includes pointers!

static assert(!__traits(isScalar, int[4]));
static assert(!__traits(isScalar, void16));
static assert(!__traits(isScalar, void));
static assert(!__traits(isScalar, typeof(null)));
static assert(!__traits(isScalar, Object));
```

25.5 *isUnsigned*

Works like `isArithmetic`, except it's for unsigned types.

```
import core.simd : uint4;

enum SignedEnum { a, b }
enum UnsignedEnum : uint { a, b }

static assert(__traits(isUnsigned, bool));
static assert(__traits(isUnsigned, char));
static assert(__traits(isUnsigned, uint));
static assert(__traits(isUnsigned, UnsignedEnum));
static assert(__traits(isUnsigned, uint4));

static assert(!__traits(isUnsigned, int));
static assert(!__traits(isUnsigned, float));
static assert(!__traits(isUnsigned, SignedEnum));
static assert(!__traits(isUnsigned, uint[4]));
static assert(!__traits(isUnsigned, void*));
```

25.6 *isStaticArray*

Works like `isArithmetic`, except it's for static array types.

```
import core.simd : int4;

enum E : int[4] { a = [1, 2, 3, 4] }

static array = [1, 2, 3]; // Not a static array: the type is inferred as int[] not int[3].
```

```
static assert(__traits(isStaticArray, void[0]));
static assert(__traits(isStaticArray, E));
static assert(!__traits(isStaticArray, int4));
static assert(!__traits(isStaticArray, array));
```

25.7 *isAssociativeArray*

Works like `isArithmetic`, except it's for associative array types.

25.8 *isAbstractClass*

If the arguments are all either types that are abstract classes, or expressions that are typed as abstract classes, then `true` is returned. Otherwise, `false` is returned. If there are no arguments, `false` is returned.

```
import std.stdio;

abstract class C { int foo(); }

void main()
{
    C c;
    writeln(__traits(isAbstractClass, C));
    writeln(__traits(isAbstractClass, c, C));
    writeln(__traits(isAbstractClass));
    writeln(__traits(isAbstractClass, int*));
}
```

Prints:

```
true
true
false
false
```

25.9 *isFinalClass*

Works like `isAbstractClass`, except it's for final classes.

25.10 *isPOD*

Takes one argument, which must be a type. It returns `true` if the type is a POD type, otherwise `false`.

25.11 *isNested*

Takes one argument. It returns `true` if the argument is a nested type which internally stores a context pointer, otherwise it returns `false`. Nested types can be classes, structs, and functions.

25.12 *isFuture*

Takes one argument. It returns `true` if the argument is a symbol marked with the `@future` keyword, otherwise `false`. Currently, only functions and variable declarations have support for the `@future` keyword.

25.13 *isDeprecated*

Takes one argument. It returns `true` if the argument is a symbol marked with the `deprecated` keyword, otherwise `false`.

25.14 *isDisabled*

Takes one argument and returns `true` if it's a function declaration marked with `@disable`.

```
struct Foo
{
    @disable void foo();
    void bar(){}
}

static assert(__traits(isDisabled, Foo.foo));
static assert(!__traits(isDisabled, Foo.bar));
```

For any other declaration even if `@disable` is a syntactically valid attribute `false` is returned because the annotation has no effect.

```
@disable struct Bar{}

static assert(!__traits(isDisabled, Bar));
```

25.15 *isVirtualFunction*

The same as *isVirtualMethod*, except that final functions that don't override anything return true.

25.16 *isVirtualMethod*

Takes one argument. If that argument is a virtual function, `true` is returned, otherwise `false`. Final functions that don't override anything return false.

```
import std.stdio;

struct S
{
    void bar() { }
}

class C
{
    void bar() { }
}

void main()
{
    writeln(__traits(isVirtualMethod, C.bar));  // true
    writeln(__traits(isVirtualMethod, S.bar));  // false
}
```

25.17 *isAbstractFunction*

Takes one argument. If that argument is an abstract function, `true` is returned, otherwise `false`.

```
import std.stdio;

struct S
{
    void bar() { }
}

class C
{
```

```
    void bar() { }
}

class AC
{
    abstract void foo();
}

void main()
{
    writeln(__traits(isAbstractFunction, C.bar));   // false
    writeln(__traits(isAbstractFunction, S.bar));   // false
    writeln(__traits(isAbstractFunction, AC.foo));  // true
}
```

25.18 *isFinalFunction*

Takes one argument. If that argument is a final function, `true` is returned, otherwise `false`.

```
import std.stdio;

struct S
{
    void bar() { }
}

class C
{
    void bar() { }
    final void foo();
}

final class FC
{
    void foo();
}

void main()
{
    writeln(__traits(isFinalFunction, C.bar));  // false
```

```
    writeln(__traits(isFinalFunction, S.bar));  // false
    writeln(__traits(isFinalFunction, C.foo));  // true
    writeln(__traits(isFinalFunction, FC.foo)); // true
}
```

25.19 *isOverrideFunction*

Takes one argument. If that argument is a function marked with override, `true` is returned, otherwise `false`.

```
import std.stdio;

class Base
{
    void foo() { }
}

class Foo : Base
{
    override void foo() { }
    void bar() { }
}

void main()
{
    writeln(__traits(isOverrideFunction, Base.foo)); // false
    writeln(__traits(isOverrideFunction, Foo.foo));  // true
    writeln(__traits(isOverrideFunction, Foo.bar));  // false
}
```

25.20 *isStaticFunction*

Takes one argument. If that argument is a static function, meaning it has no context pointer, `true` is returned, otherwise `false`.

```
struct A
{
    int foo() { return 3; }
    static int boo(int a) { return a; }
```

```
}

void main()
{
    assert(__traits(isStaticFunction, A.boo));
    assert(!__traits(isStaticFunction, A.foo));
    assert(__traits(isStaticFunction, main));
}
```

25.21 *isRef*, *isOut*, *isLazy*

Takes one argument. If that argument is a declaration, `true` is returned if it is ref, out, or lazy, otherwise `false`.

```
void fooref(ref int x)
{
    static assert(__traits(isRef, x));
    static assert(!__traits(isOut, x));
    static assert(!__traits(isLazy, x));
}

void fooout(out int x)
{
    static assert(!__traits(isRef, x));
    static assert(__traits(isOut, x));
    static assert(!__traits(isLazy, x));
}

void foolazy(lazy int x)
{
    static assert(!__traits(isRef, x));
    static assert(!__traits(isOut, x));
    static assert(__traits(isLazy, x));
}
```

25.22 *isTemplate*

Takes one argument. If that argument is a template then `true` is returned, otherwise `false`.

```
void foo(T)(){}
static assert(__traits(isTemplate,foo));
static assert(!__traits(isTemplate,foo!int()));
static assert(!__traits(isTemplate,"string"));
```

25.23 *isZeroInit*

Takes one argument which must be a type. If the type's default initializer is all zero bits then `true` is returned, otherwise `false`.

```
struct S1 { int x; }
struct S2 { int x = -1; }

static assert(__traits(isZeroInit, S1));
static assert(!__traits(isZeroInit, S2));

void test()
{
    int x = 3;
    static assert(__traits(isZeroInit, typeof(x)));
}

// `isZeroInit` will always return true for a class C
// because `C.init` is null reference.

class C { int x = -1; }

static assert(__traits(isZeroInit, C));

// For initializing arrays of element type `void`.
static assert(__traits(isZeroInit, void));
```

25.24 *isReturnOnStack*

Takes one argument which must either be a function symbol, function literal, a delegate, or a function pointer. It returns a `bool` which is `true` if the return value of the function is returned on the stack via a pointer to it passed as a hidden extra parameter to the function.

```
struct S { int[20] a; }
```

```
int test1();
S test2();

static assert(__traits(isReturnOnStack, test1) == false);
static assert(__traits(isReturnOnStack, test2) == true);
```

Implementation Defined: This is determined by the function ABI calling convention in use, which is often complex.

Best Practices: This has applications in:

1. Returning values in registers is often faster, so this can be used as a check on a hot function to ensure it is using the fastest method.
2. When using inline assembly to correctly call a function.
3. Testing that the compiler does this correctly is normally hackish and awkward, this enables efficient, direct, and simple testing.

25.25 *isModule*

Takes one argument. If that argument is a symbol that refers to a spec/module, module then `true` is returned, otherwise `false`. Package modules are considered to be modules even if they have not been directly imported as modules.

```
import std.algorithm.sorting;

// A regular package (no package.d)
static assert(!__traits(isModule, std));
// A package module (has a package.d file)
// Note that we haven't imported std.algorithm directly.
// (In other words, we don't have an "import std.algorithm;" directive.)
static assert(__traits(isModule, std.algorithm));
// A regular module
static assert(__traits(isModule, std.algorithm.sorting));
```

25.26 *isPackage*

Takes one argument. If that argument is a symbol that refers to a spec/module, package then `true` is returned, otherwise `false`.

```
import std.algorithm.sorting;
static assert(__traits(isPackage, std));
static assert(__traits(isPackage, std.algorithm));
```

```
static assert(!__traits(isPackage, std.algorithm.sorting));
```

25.27 *hasMember*

The first argument is a type that has members, or is an expression of a type that has members. The second argument is a string. If the string is a valid property of the type, `true` is returned, otherwise `false`.

```
import std.stdio;

struct S
{
    int m;

    import std.stdio; // imports write
}

void main()
{
    S s;

    writeln(__traits(hasMember, S, "m")); // true
    writeln(__traits(hasMember, s, "m")); // true
    writeln(__traits(hasMember, S, "y")); // false
    writeln(__traits(hasMember, S, "write")); // true
    writeln(__traits(hasMember, int, "sizeof")); // true
}
```

25.28 *hasCopyConstructor*

The argument is a type. If it is a struct with a copy constructor, returns `true`. Otherwise, return `false`. Note that a copy constructor is distinct from a postblit.

```
import std.stdio;

struct S
{
}
```

```
class C
{
}

struct P
{
    this(ref P rhs) {}
}

struct B
{
    this(this) {}
}

void main()
{
    writeln(__traits(hasCopyConstructor, S)); // false
    writeln(__traits(hasCopyConstructor, C)); // false
    writeln(__traits(hasCopyConstructor, P)); // true
    writeln(__traits(hasCopyConstructor, B)); // false, this is a postblit
}
```

25.29 *hasPostblit*

The argument is a type. If it is a struct with a postblit, returns `true`. Otherwise, return `false`. Note a postblit is distinct from a copy constructor.

```
import std.stdio;

struct S
{
}

class C
{
}

struct P
```

```
{
    this(ref P rhs) {}
}

struct B
{
    this(this) {}
}


void main()
{
    writeln(__traits(hasPostblit, S)); // false
    writeln(__traits(hasPostblit, C)); // false
    writeln(__traits(hasPostblit, P)); // false, this is a copy ctor
    writeln(__traits(hasPostblit, B)); // true
}
```

25.30 *identifier*

Takes one argument, a symbol. Returns the identifier for that symbol as a string literal.

```
import std.stdio;

int var = 123;
pragma(msg, typeof(var));                       // int
pragma(msg, typeof(__traits(identifier, var))); // string
writeln(var);                                   // 123
writeln(__traits(identifier, var));             // "var"
```

25.31 *getAliasThis*

Takes one argument, a type. If the type has `alias this` declarations, returns a sequence of the names (as `string`s) of the members used in those declarations. Otherwise returns an empty sequence.

```
alias AliasSeq(T...) = T;

struct S1
```

```
{
    string var;
    alias var this;
}
static assert(__traits(getAliasThis, S1) == AliasSeq!("var"));
static assert(__traits(getAliasThis, int).length == 0);

pragma(msg, __traits(getAliasThis, S1));
pragma(msg, __traits(getAliasThis, int));
```

Prints:

```
tuple("var")
tuple()
```

25.32 *getAttributes*

Takes one argument, a symbol. Returns a tuple of all attached user-defined attributes. If no UDAs exist it will return an empty tuple.

For more information, see: User-Defined Attributes

```
@(3) int a;
@("string", 7) int b;

enum Foo;
@Foo int c;

pragma(msg, __traits(getAttributes, a));
pragma(msg, __traits(getAttributes, b));
pragma(msg, __traits(getAttributes, c));
```

Prints:

```
tuple(3)
tuple("string", 7)
tuple((Foo))
```

25.33 *getFunctionVariadicStyle*

Takes one argument which must either be a function symbol, or a type that is a function, delegate or a function pointer. It returns a string identifying the kind of variadic arguments that are supported.

getFunctionVariadicStyle

result	kind	access	example
`"none"`	not a variadic function		`void foo();`
`"argptr"`	D style variadic function	`_argptr` and `_arguments`	`void bar(...)`
`"stdarg"`	C style variadic function	`core.stdc.stdarg`	`extern (C) void abc(int, ...)`
`"typesafe"`	typesafe variadic function	array on stack	`void def(int[] ...)`

```
import core.stdc.stdarg;

void novar() {}
extern(C) void cstyle(int, ...) {}
extern(C++) void cppstyle(int, ...) {}
void dstyle(...) {}
void typesafe(int[]...) {}

static assert(__traits(getFunctionVariadicStyle, novar) == "none");
static assert(__traits(getFunctionVariadicStyle, cstyle) == "stdarg");
static assert(__traits(getFunctionVariadicStyle, cppstyle) == "stdarg");
static assert(__traits(getFunctionVariadicStyle, dstyle) == "argptr");
static assert(__traits(getFunctionVariadicStyle, typesafe) == "typesafe");

static assert(__traits(getFunctionVariadicStyle, (int[] a...) {}) == "typesafe");
static assert(__traits(getFunctionVariadicStyle, typeof(cstyle)) == "stdarg");
```

25.34 *getFunctionAttributes*

Takes one argument which must either be a function symbol, function literal, or a function pointer. It returns a string tuple of all the attributes of that function **excluding** any user-defined attributes (UDAs can be retrieved with the getAttributes trait). If no attributes exist it will return an empty tuple.

Note: The order of the attributes in the returned tuple is implementation-defined and should not be relied upon.

A list of currently supported attributes are:

- `pure`, `nothrow`, `@nogc`, `@property`, `@system`, `@trusted`, `@safe`, `ref` and `@live`

Note: `ref` is a function attribute even though it applies to the return type.

Additionally the following attributes are only valid for non-static member functions:

- `const`, `immutable`, `inout`, `shared`

For example:

```
int sum(int x, int y) pure nothrow { return x + y; }

pragma(msg, __traits(getFunctionAttributes, sum));

struct S
{
    void test() const @system { }
}

pragma(msg, __traits(getFunctionAttributes, S.test));

void main(){}
```

Prints:

```
tuple("pure", "nothrow", "@system")
tuple("const", "@system")
```

Note that some attributes can be inferred. For example:

```
pragma(msg, __traits(getFunctionAttributes, (int x) @trusted { return x * 2; }));

void main(){}
```

Prints:

```
tuple("pure", "nothrow", "@nogc", "@trusted")
```

25.35 *getLinkage*

Takes one argument, which is a declaration symbol, or the type of a function, delegate, pointer to function, struct, class, or interface. Returns a string representing the LinkageAttribute of the declaration. The string is one of:

- `"D"`
- `"C"`
- `"C++"`
- `"Windows"`
- `"Objective-C"`

- "System"

```
extern (C) int fooc();
alias aliasc = fooc;

static assert(__traits(getLinkage, fooc) == "C");
static assert(__traits(getLinkage, aliasc) == "C");

extern (C++) struct FooCPPStruct {}
extern (C++) class FooCPPClass {}
extern (C++) interface FooCPPInterface {}

static assert(__traits(getLinkage, FooCPPStruct) == "C++");
static assert(__traits(getLinkage, FooCPPClass) == "C++");
static assert(__traits(getLinkage, FooCPPInterface) == "C++");
```

25.36 *getLocation*

Takes one argument which is a symbol. To disambiguate between overloads, pass the result of *getOverloads* with the desired index, to `getLocation`. Returns a `tuple(string, int, int)` whose entries correspond to the filename, line number and column number where the argument was declared.

25.37 *getMember*

Takes two arguments, the second must be a string. The result is an expression formed from the first argument, followed by a '.', followed by the second argument as an identifier.

```
import std.stdio;

struct S
{
    int mx;
    static int my;
}

void main()
{
    S s;
```

```
    __traits(getMember, s, "mx") = 1;  // same as s.mx=1;
    writeln(__traits(getMember, s, "m" ~ "x")); // 1

    // __traits(getMember, S, "mx") = 1;  // error, no this for S.mx
    __traits(getMember, S, "my") = 2;  // ok
}
```

25.38 *getOverloads*

The first argument is an aggregate (e.g. struct/class/module). The second argument is a `string` that matches the name of the member(s) to return. The third argument is a `bool`, and is optional. If `true`, the result will also include template overloads. The result is a tuple of all the overloads of the supplied name.

```
import std.stdio;

class D
{
    this() { }
    ~this() { }
    void foo() { }
    int foo(int) { return 2; }
    void bar(T)() { return T.init; }
    class bar(int n) {}
}

void main()
{
    D d = new D();

    foreach (t; __traits(getOverloads, D, "foo"))
        writeln(typeid(typeof(t)));

    alias b = typeof(__traits(getOverloads, D, "foo"));
    foreach (t; b)
        writeln(typeid(t));

    auto i = __traits(getOverloads, d, "foo")[1](1);
    writeln(i);
```

```
    foreach (t; __traits(getOverloads, D, "bar", true))
        writeln(t.stringof);
}
```

Prints:

```
void()
int()
void()
int()
2
bar(T)()
bar(int n)
```

25.39 *getParameterStorageClasses*

Takes two arguments. The first must either be a function symbol, or a type that is a function, delegate or a function pointer. The second is an integer identifying which parameter, where the first parameter is 0. It returns a tuple of strings representing the storage classes of that parameter.

```
ref int foo(return ref const int* p, scope int* a, out int b, lazy int c);

static assert(__traits(getParameterStorageClasses, foo, 0)[0] == "return");
static assert(__traits(getParameterStorageClasses, foo, 0)[1] == "ref");

static assert(__traits(getParameterStorageClasses, foo, 1)[0] == "scope");
static assert(__traits(getParameterStorageClasses, foo, 2)[0] == "out");
static assert(__traits(getParameterStorageClasses, typeof(&foo), 3)[0] == "lazy");
```

25.40 *getPointerBitmap*

The argument is a type. The result is an array of `size_t` describing the memory used by an instance of the given type.

The first element of the array is the size of the type (for classes it is the *classInstanceSize*).

The following elements describe the locations of GC managed pointers within the memory occupied by an instance of the type. For type T, there are `T.sizeof / size_t.sizeof` possible pointers represented by the bits of the array values.

This array can be used by a precise GC to avoid false pointers.

```
void main()
{
    static class C
    {
        // implicit virtual function table pointer not marked
        // implicit monitor field not marked, usually managed manually
        C next;
        size_t sz;
        void* p;
        void function () fn; // not a GC managed pointer
    }

    static struct S
    {
        size_t val1;
        void* p;
        C c;
        byte[] arr;          // { length, ptr }
        void delegate () dg; // { context, func }
    }

    static assert (__traits(getPointerBitmap, C) == [6*size_t.sizeof, 0b010100]);
    static assert (__traits(getPointerBitmap, S) == [7*size_t.sizeof, 0b0110110]);
}
```

25.41 *getProtection*

The argument is a symbol. The result is a string giving its protection level: "public", "private", "protected", "export", or "package".

```
import std.stdio;

class D
{
    export void foo() { }
    public int bar;
}

void main()
{
```

```
    D d = new D();

    auto i = __traits(getProtection, d.foo);
    writeln(i);

    auto j = __traits(getProtection, d.bar);
    writeln(j);
}
```

Prints:

```
export
public
```

25.42 *getTargetInfo*

Receives a string key as argument. The result is an expression describing the requested target information.

```
version (CppRuntime_Microsoft)
    static assert(__traits(getTargetInfo, "cppRuntimeLibrary") == "libcmt");
```

Keys are implementation defined, allowing relevant data for exotic targets. A reliable subset exists which are always available:

- `"cppRuntimeLibrary"` - The C++ runtime library affinity for this toolchain
- `"floatAbi"` - Floating point ABI; may be `"hard"`, `"soft"`, or `"softfp"`
- `"objectFormat"` - Target object format

25.43 *getVirtualFunctions*

The same as *getVirtualMethods*, except that final functions that do not override anything are included.

25.44 *getVirtualMethods*

The first argument is a class type or an expression of class type. The second argument is a string that matches the name of one of the functions of that class. The result is a tuple of the virtual overloads of that function. It does not include final functions that do not override anything.

```
import std.stdio;

class D
{
    this() { }
    ~this() { }
    void foo() { }
    int foo(int) { return 2; }
}

void main()
{
    D d = new D();

    foreach (t; __traits(getVirtualMethods, D, "foo"))
        writeln(typeid(typeof(t)));

    alias b = typeof(__traits(getVirtualMethods, D, "foo"));
    foreach (t; b)
        writeln(typeid(t));

    auto i = __traits(getVirtualMethods, d, "foo")[1](1);
    writeln(i);
}
```

Prints:

```
void()
int()
void()
int()
2
```

25.45 *getUnitTests*

Takes one argument, a symbol of an aggregate (e.g. struct/class/module). The result is a tuple of all the unit test functions of that aggregate. The functions returned are like normal nested static functions, CTFE will work and UDAs will be accessible.

Note:

The -unittest flag needs to be passed to the compiler. If the flag is not passed `__traits(getUnitTests)` will always return an empty tuple.

```
module foo;

import core.runtime;
import std.stdio;

struct name { string name; }

class Foo
{
    unittest
    {
        writeln("foo.Foo.unittest");
    }
}

@name("foo") unittest
{
    writeln("foo.unittest");
}

template Tuple (T...)
{
    alias Tuple = T;
}

shared static this()
{
  // Override the default unit test runner to do nothing. After that, "main" will
  // be called.
  Runtime.moduleUnitTester = { return true; };
}

void main()
{
    writeln("start main");
```

```
    alias tests = Tuple!(__traits(getUnitTests, foo));
    static assert(tests.length == 1);

    alias attributes = Tuple!(__traits(getAttributes, tests[0]));
    static assert(attributes.length == 1);

    foreach (test; tests)
        test();

    foreach (test; __traits(getUnitTests, Foo))
        test();
}
```

By default, the above will print:

```
start main
foo.unittest
foo.Foo.unittest
```

25.46 *parent*

Takes a single argument which must evaluate to a symbol. The result is the symbol that is the parent of it.

25.47 *classInstanceSize*

Takes a single argument, which must evaluate to either a class type or an expression of class type. The result is of type `size_t`, and the value is the number of bytes in the runtime instance of the class type. It is based on the static type of a class, not the polymorphic type.

25.48 *getVirtualIndex*

Takes a single argument which must evaluate to a function. The result is a `ptrdiff_t` containing the index of that function within the vtable of the parent type. If the function passed in is final and does not override a virtual function, `-1` is returned instead.

25.49 *allMembers*

Takes a single argument, which must evaluate to either a type or an expression of type. A tuple of string literals is returned, each of which is the name of a member of that type combined with all of the members of the base classes (if the type is a class). No name is repeated. Builtin properties are not included.

```
import std.stdio;

class D
{
    this() { }
    ~this() { }
    void foo() { }
    int foo(int) { return 0; }
}

void main()
{
    auto b = [ __traits(allMembers, D) ];
    writeln(b);
    // ["__ctor", "__dtor", "foo", "toString", "toHash", "opCmp", "opEquals",
    // "Monitor", "factory"]
}
```

The order in which the strings appear in the result is not defined.

25.50 *derivedMembers*

Takes a single argument, which must evaluate to either a type or an expression of type. A tuple of string literals is returned, each of which is the name of a member of that type. No name is repeated. Base class member names are not included. Builtin properties are not included.

```
import std.stdio;

class D
{
    this() { }
    ~this() { }
    void foo() { }
    int foo(int) { return 0; }
}

void main()
{
```

```
    auto a = [__traits(derivedMembers, D)];
    writeln(a);    // ["__ctor", "__dtor", "foo"]
}
```

The order in which the strings appear in the result is not defined.

25.51 *isSame*

Takes two arguments and returns bool `true` if they are the same symbol, `false` if not.

```
import std.stdio;

struct S { }

int foo();
int bar();

void main()
{
    writeln(__traits(isSame, foo, foo)); // true
    writeln(__traits(isSame, foo, bar)); // false
    writeln(__traits(isSame, foo, S));   // false
    writeln(__traits(isSame, S, S));     // true
    writeln(__traits(isSame, std, S));   // false
    writeln(__traits(isSame, std, std)); // true
}
```

If the two arguments are expressions made up of literals or enums that evaluate to the same value, true is returned.

If the two arguments are both lambda functions (or aliases to lambda functions), then they are compared for equality. For the comparison to be computed correctly, the following conditions must be met for both lambda functions:

1. The lambda function arguments must not have a template instantiation as an explicit argument type. Any other argument types (basic, user-defined, template) are supported.
2. The lambda function body must contain a single expression (no return statement) which contains only numeric values, manifest constants, enum values, function arguments and function calls. If the expression contains local variables or return statements, the function is considered incomparable.

If these constraints aren't fulfilled, the function is considered incomparable and `isSame` returns `false`.

```
int f() { return 2; }
void test(alias pred)()
{
    // f() from main is a different function from top-level f()
    static assert(!__traits(isSame, (int a) => a + f(), pred));
}

void main()
{
    static assert(__traits(isSame, (a, b) => a + b, (c, d) => c + d));
    static assert(__traits(isSame, a => ++a, b => ++b));
    static assert(!__traits(isSame, (int a, int b) => a + b, (a, b) => a + b));
    static assert(__traits(isSame, (a, b) => a + b + 10, (c, d) => c + d + 10));

    // lambdas accessing local variables are considered incomparable
    int b;
    static assert(!__traits(isSame, a => a + b, a => a + b));

    // lambdas calling other functions are comparable
    int f() { return 3;}
    static assert(__traits(isSame, a => a + f(), a => a + f()));
    test!((int a) => a + f())();

    class A
    {
        int a;
        this(int a)
        {
            this.a = a;
        }
    }

    class B
    {
        int a;
        this(int a)
        {
            this.a = a;
        }
    }
```

```
    static assert(__traits(isSame, (A a) => ++a.a, (A b) => ++b.a));
    // lambdas with different data types are considered incomparable,
    // even if the memory layout is the same
    static assert(!__traits(isSame, (A a) => ++a.a, (B a) => ++a.a));
}
```

25.52 *compiles*

Returns a bool `true` if all of the arguments compile (are semantically correct). The arguments can be symbols, types, or expressions that are syntactically correct. The arguments cannot be statements or declarations.

If there are no arguments, the result is `false`.

```
import std.stdio;

struct S
{
    static int s1;
    int s2;
}

int foo();
int bar();

void main()
{
    writeln(__traits(compiles));                      // false
    writeln(__traits(compiles, foo));                 // true
    writeln(__traits(compiles, foo + 1));             // true
    writeln(__traits(compiles, &foo + 1));            // false
    writeln(__traits(compiles, typeof(1)));           // true
    writeln(__traits(compiles, S.s1));                // true
    writeln(__traits(compiles, S.s3));                // false
    writeln(__traits(compiles, 1,2,3,int,long,std));  // true
    writeln(__traits(compiles, 3[1]));                // false
    writeln(__traits(compiles, 1,2,3,int,long,3[1])); // false
}
```

This is useful for:

- Giving better error messages inside generic code than the sometimes hard to follow compiler ones.
- Doing a finer grained specialization than template partial specialization allows for.

Chapter 26

Error Handling

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.

26.1 The Error Handling Problem

The traditional C way of detecting and reporting errors is not traditional, it is ad-hoc and varies from function to function, including:

- Returning a NULL pointer.
- Returning a 0 value.
- Returning a non-zero error code.
- Requiring errno to be checked.
- Requiring that a function be called to check if the previous function failed.

To deal with these possible errors, tedious error handling code must be added to each function call. If an error happened, code must be written to recover from the error, and the error must be reported to the user in some user friendly fashion. If an error cannot be handled locally, it must be explicitly propagated back to its caller. The long list of errno values needs to be converted into appropriate text to be displayed. Adding all the code to do this can consume a large part of the time spent coding a project - and still, if a new errno value is added to the runtime system, the old code can not properly display a meaningful error message.

Good error handling code tends to clutter up what otherwise would be a neat and clean looking implementation.

Even worse, good error handling code is itself error prone, tends to be the least tested (and therefore buggy) part of the project, and is frequently simply omitted. The end result is likely a "blue screen of death" as the program failed to deal with some unanticipated error.

Quick and dirty programs are not worth writing tedious error handling code for, and so such utilities tend to be like using a table saw with no blade guards.

What's needed is an error handling philosophy and methodology 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.

26.2 The D Error Handling Solution

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

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

The solution is to use exception handling to report errors. All errors are objects derived from abstract class `Error`. `Error` has a pure virtual function called toString() which produces a `string` 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.

Chapter 27

Unit Tests

```
UnitTest:
    unittest BlockStatement
```

Unit tests are a builtin framework of test cases applied to a module to determine if it is working properly. A D program can be run with unit tests enabled or disabled.

Unit tests are a special function defined like:

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

Individual tests are specified in the unit test using AssertExpressions. Unlike *AssertExpressions* used elsewhere, the assert is not assumed to hold, and upon assert failure the program is still in a defined state.

There can be any number of unit test functions in a module, including within struct, union and class declarations. They are executed in lexical order.

Unit tests, when enabled, are run after all static initialization is complete and before the `main()` function is called.

For example, given a class `Sum` that is used to add two values, a unit test can be given:

```
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);
    }
}
```

When unit tests are enabled, the version identifier `unittest` is predefined.

27.1 Attributed Unittests

A unittest may be attributed with any of the global function attributes. Such unittests are useful in verifying the given attribute(s) on a template function:

```
void myFunc(T)(T[] data)
{
    if (data.length > 2)
        data[0] = data[1];
}

@safe nothrow unittest
{
    auto arr = [1,2,3];
    myFunc(arr);
    assert(arr == [2,2,3]);
}
```

This unittest verifies that `myFunc` contains only `@safe`, `nothrow` code. Although this can also be accomplished by attaching these attributes to `myFunc` itself, that would prevent `myFunc` from being instantiated with types `T` that have `@system` or throwing code in their `opAssign` method, or other methods that `myFunc` may call. The above idiom allows `myFunc` to be instantiated with such types, yet at the same time verify that the `@system` and throwing behavior is not introduced by the code within `myFunc` itself.

Implementation Defined:

1. If unit tests are not enabled, the implementation is not required to check the *UnitTest* for syntactic or semantic correctness. This is to reduce the compile time impact of larger unit test sections. The tokens must still be valid, and the implementation can merely count `{` and `}` tokens to find the end of the *UnitTest*'s *BlockStatement*.
2. The presentation of unit test results to the user.
3. The method used to enable or disable the unit tests. Use of a compiler switch such as **-unittest** to enable them is suggested.
4. The order in which modules are called to run their unit tests.

5. Whether the program stops on the first unit test failure, or continues running the unit tests.

Best Practices:

1. Using unit tests in conjuction with coverage testing (such as **-cov**) is effective.
2. A unit test for a function should appear immediately following it.

27.2 Documented Unittests

Documented unittests allow the developer to deliver code examples to the user, while at the same time automatically verifying that the examples are valid. This avoids the frequent problem of having outdated documentation for some piece of code.

If a declaration is followed by a documented unittest, the code in the unittest will be inserted in the **example** section of the declaration:

```
/// Math class
class Math
{
    /// add function
    static int add(int x, int y) { return x + y; }

    ///
    unittest
    {
        assert(add(2, 2) == 4);
    }
}

///
unittest
{
    auto math = new Math();
    auto result = math.add(2, 2);
}
```

The above will generate the following documentation:

```
</big></dt>
<dd><u>Math</u> class<br><br>
<b>Example:</b><pre class="d_code"><font color="blue">auto</font> math = <font
    color="blue">new</font> <u>Math</u>;
<font color="blue">auto</font> result = math.add(2, 2);
```

```
</pre><br>
<dl><dt><big><a name="Math.add"></a>int <u>add</u>(int <i>x</i>, int <i>y</i>);
</big></dt>
<dd><u>add</u> function<br><br>
<b>Example:</b><pre class="d_code"><font color="blue">assert</font>(<u>add</u>(2,
   2) == 4);
</pre>
</dd>
</dl>
</dd>
</dl>
```

A unittest which is not documented, or is marked as private will not be used to generate code samples.

There can be multiple documented unittests and they can appear in any order. They will be attached to the last non-unittest declaration:

```
/// add function
int add(int x, int y) { return x + y; }

/// code sample generated
unittest
{
    assert(add(1, 1) == 2);
}

/// code sample not generated because the unittest is private
private unittest
{
    assert(add(2, 2) == 4);
}

unittest
{
    /// code sample not generated because the unittest isn't documented
    assert(add(3, 3) == 6);
}

/// code sample generated, even if it only includes comments (or is empty)
unittest
{
```

```
    /** assert(add(4, 4) == 8); */
}
```

The above will generate the following documentation:

```
</big></dt>
<dd><u>add</u> function<br><br>
<b>Examples:</b><br>
code sample generated
<pre class="d_code">
<font color="blue">assert</font>(<u>add</u>(1, 1) == 2);
</pre>
<br><br><b>Examples:</b><br>
code sample generated, even if it is empty or only includes comments
<pre class="d_code">
<font color="green">/** assert(add(4, 4) == 8); */</font>
</pre>
<br><br>
</dd>
</dl>
```

Chapter 28

Garbage Collection

D is a systems programming language with support for garbage collection. Usually it is not necessary to free memory explicitly. Just allocate as needed, and the garbage collector will periodically return all unused memory to the pool of available memory.

D also provides the mechanisms to write code where the garbage collector is **not involved**. More information is provided below.

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 often 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.
  - 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 tracing and freeing memory.

  - 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 collection is not a panacea. There are some downsides:

- It is not always obvious when the GC allocates memory, which in turn can trigger a collection, so the program can pause unexpectedly.
- The time it takes for a collection to complete is not bounded. While in practice it is very quick, this cannot normally be guaranteed.
- Normally, 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.
- Garbage collection should be implemented as a basic operating system kernel service. But since it is not, garbage collecting programs must carry around with them the garbage collection implementation. While this can be a shared library, it is still there.

These constraints are addressed by techniques outlined in Memory Management, including the mechanisms provided by D to control allocations outside the GC heap.

There is currently work in progress to make the runtime library free of GC heap allocations, to allow its use in scenarios where the use of GC infrastructure is not possible.

28.1 How Garbage Collection Works

The GC works by:

1. Stopping all other threads than the thread currently trying to allocate GC memory.
2. 'Hijacking' the current thread for GC work.
3. Scanning all 'root' memory ranges for pointers into GC allocated memory.
4. Recursively scanning all allocated memory pointed to by roots looking for more pointers into GC allocated memory.

5. Freeing all GC allocated memory that has no active pointers to it and do not need destructors to run.
6. Queueing all unreachable memory that needs destructors to run.
7. Resuming all other threads.
8. Running destructors for all queued memory.
9. Freeing any remaining unreachable memory.
10. Returning the current thread to whatever work it was doing.

28.2 Interfacing Garbage Collected Objects With Foreign Code

The garbage collector looks for roots in:

1. the static data segment
2. the stacks and register contents of each thread
3. the TLS (thread-local storage) areas of each thread
4. any roots added by core.memory.GC.addRoot() or core.memory.GC.addRange()

If the only pointer to an object is held outside of these areas, then the collector will miss it and free the memory.

To avoid this from happening, either

- maintain a pointer to the object in an area the collector does scan for pointers;
- add a root where a pointer to the object is stored using core.memory.GC.addRoot() or core.memory.GC.addRange().
- reallocate and copy the object using the foreign code's storage allocator or using the C runtime library's malloc/free.

28.3 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 fields for GC pointers.
- 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:

  ```
  struct Foo
  {
    align (1):
      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 the pointer size, but since this kind of implementation is not guaranteed by the C standard, use `memcpy()` only with extreme caution.
- Do not have pointers in a struct instance that point back to the same instance. The trouble with this is if the instance gets moved in memory, the pointer will point back to where it came from, with likely disastrous results.

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.

28.4 Working with the Garbage Collector

Garbage collection doesn't solve every memory deallocation problem. For example, if a pointer 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 because new stack frames are initialized anyway.

28.5 Object Pinning and a Moving Garbage Collector

Although D does not currently use a moving garbage collector, by following the rules listed above one can be implemented. No special action is required to pin objects. A moving collector will only move objects for which there are no ambiguous references, and for which it can update those references. All other objects will be automatically pinned.

28.6 D Operations That Involve the Garbage Collector

Some sections of code may need to avoid using the garbage collector. The following constructs may allocate memory using the garbage collector:

- *NewExpression*
- Array appending
- Array concatenation
- Array literals (except when used to initialize static data)
- Associative array literals
- Any insertion, removal, or lookups in an associative array
- Extracting keys or values from an associative array
- Taking the address of (i.e. making a delegate to) a nested function that accesses variables in an outer scope
- A function literal that accesses variables in an outer scope
- An *AssertExpression* that fails its condition

28.7 Configuring the Garbage Collector

Since version 2.067, The garbage collector can now be configured through the command line, the environment or by options embedded into the executable.

By default, GC options can only be passed on the command line of the program to run, e.g.

```
app "--DRT-gcopt=profile:1 minPoolSize:16" arguments to app
```

Available GC options are:

- disable:0|1 - start disabled
- profile:0|1 - enable profiling with summary when terminating program
- gc:conservative|precise|manual - select gc implementation (default = conservative)
- initReserve:N - initial memory to reserve in MB
- minPoolSize:N - initial and minimum pool size in MB
- maxPoolSize:N - maximum pool size in MB
- incPoolSize:N - pool size increment MB
- parallel:N - number of additional threads for marking
- heapSizeFactor:N - targeted heap size to used memory ratio

- cleanup:none|collect|finalize - how to treat live objects when terminating
  - collect: run a collection (the default for backward compatibility)
  - none: do nothing
  - finalize: all live objects are finalized unconditionally

In addition, –DRT-gcopt=help will show the list of options and their current settings.

Command line options starting with "–DRT-" are filtered out before calling main, so the program will not see them. They are still available via `rt_args`.

Configuration via the command line can be disabled by declaring a variable for the linker to pick up before using its default from the runtime:

```
extern(C) __gshared bool rt_cmdline_enabled = false;
```

Likewise, declare a boolean `rt_envvars_enabled` to enable configuration via the environment variable `DRT_GCOPT`:

```
extern(C) __gshared bool rt_envvars_enabled = true;
```

Setting default configuration properties in the executable can be done by specifying an array of options named `rt_options`:

```
extern(C) __gshared string[] rt_options = [ "gcopt=initReserve:100 profile:1" ];
```

Evaluation order of options is `rt_options`, then environment variables, then command line arguments, i.e. if command line arguments are not disabled, they can override options specified through the environment or embedded in the executable.

28.8 Precise Heap Scanning

If you select `precise` as the garbage collector via the options above, type information will be used to identify actual or possible pointers or references within heap allocated data objects. Non-pointer data will not be interpreted as a reference to other memory as a "false pointer". The collector has to make pessimistic assumptions if a memory slot can contain both a pointer or an integer value, it will still be scanned (e.g. in a `union`).

If you use the GC memory functions from `core.memory`, and plan to use it for data with a mixture of pointers and non-pointer data you should pass the TypeInfo of your allocated struct, class or type as the optional parameter. The default `null` is interpreted as memory that might contain pointers everywhere.

```
struct S { size_t hash; Data* data; }
S* s = cast(S*)GC.malloc(S.sizeof, 0, typeid(S));
```

Attention: Enabling precise scanning needs slightly more caution with type declarations. For example, if you reserve a buffer as part of a struct and later emplace an object instance with references to other allocations into this memory, you must not use basic integer types to reserve the space. Doing so will cause the garbage collector not to detect the references. Instead, use an array type that will scan this area conservatively. Using `void*` is usually the best option as it also ensures proper alignment for pointers being scanned by the GC.

28.9 Precise Scanning of the DATA and TLS segment

Windows only: As of version 2.075, the DATA (global shared data) and TLS segment (thread local data) of an executable or DLL can be configured to be scanned precisely by the garbage collector instead of conservatively. This takes advantage of information emitted by the compiler to identify possible mutable pointers inside these segments. Immutable pointers with initializers are excluded from scanning, too, as they can only point to preallocated memory.

Precise scanning can be enabled with the D runtime option "scanDataSeg". Possible option values are "conservative" (default) and "precise". As with the GC options, it can be specified on the command line, in the environment or embedded into the executable, e.g.

```
extern(C) __gshared string[] rt_options = [ "scanDataSeg=precise" ];
```

Attention: Enabling precise scanning needs slightly more caution typing global memory. For example, if you pre-allocate memory in the DATA/TLS segment and later emplace an object instance with references to other allocations into this memory, you must not use basic integer types to reserve the space. Doing so will cause the garbage collector not to detect the references. Instead, use an array type that will scan this area conservatively. Using `void*` is usually the best option as it also ensures proper alignment for pointers being scanned by the GC.

```
class Singleton { void[] mem; }
void*[(__traits(classInstanceSize, Singleton) - 1) / (void*).sizeof + 1]
    singleton_store;
static this()
{
    emplace!Singleton(singleton_store).mem = allocateMem();
}
Singleton singleton() { return cast(Singleton)singleton_store.ptr; }
```

For precise typing of that area, you can also let the compiler generate the class instance into the DATA segment for you:

```
class Singleton { void[] mem; }
shared(Singleton) singleton = new Singleton;
shared static this() { singleton.mem = allocateSharedMem(); }
```

This doesn't work for TLS memory, though.

28.10 Parallel marking

By default the garbage collector uses all available CPU cores to mark the heap.

This might affect your application if it has threads that are not suspended during the mark phase of the collection. You can configure the number of additional threads used for marking by GC option `parallel`, e.g. by passing `--DRT-gcopt=parallel:2` on the command line or embedding the option into the binary via `rt_options`. The number of threads actually created is limited to `core.cpuid.threadsPerCPU-1`. A value of `0` disables parallel marking completely.

28.11 Adding your own Garbage Collector

GC implementations are added to a registry that allows to supply more implementations by just linking them into the binary. To do so add a function that is executed before the D runtime initialization using `pragma(crt_constructor)`:

```
import core.gc.gcinterface, core.gc.registry;
extern (C) pragma(crt_constructor) void registerMyGC()
{
    registerGCFactory("mygc", &createMyGC);
}

GC createMyGC()
{
    __gshared instance = new MyGC;
    instance.initialize();
    return instance;
}

class MyGC : GC { /*...*/ }
```

[The GC modules defining the interface (gc.interface) and registration (gc.registry) are currently not public and are subject to change from version to version. Add an import search path to the druntime/src path to compile the example.]

The new GC is added to the list of available garbage collectors that can be selected via the usual configuration options, e.g. by embedding `rt_options` into the binary:

```
extern (C) __gshared string[] rt_options = ["gcopt=gc:mygc"];
```

You can also remove the standard GC implementation from a statically linked binary by redefining the function `extern(C) void* register_default_gcs()`. If no custom garbage collector has been registered all attempts to allocate GC managed memory will terminate the application with an appropriate message.

28.12 References

- Wikipedia
- GC FAQ
- Uniprocessor Garbage Collector Techniques
- Garbage Collection: Algorithms for Automatic Dynamic Memory Management

Chapter 29

Floating Point

29.1 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

29.2 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;
writeln(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;
writeln(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;
writeln(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.

29.3 Rounding Control

IEEE 754 floating point arithmetic includes the ability to set 4 different rounding modes. These are accessible via the functions in `core.stdc.fenv`.

If the floating-point rounding mode is changed within a function, it must be restored before the function exits. If this rule is violated (for example, by the use of inline asm), the rounding mode used for subsequent calculations is undefined.

29.4 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 `core.stdc.fenv`.

29.5 Floating Point Transformations

An implementation may perform transformations on floating point computations in order to reduce their strength, i.e. their runtime computation time. Because floating point math does not precisely follow mathematical rules, some transformations are not valid, even though some other programming languages still allow them.

The following transformations of floating point expressions are not allowed because under IEEE rules they could produce different results.

Disallowed Floating Point Transformations

transformation	comments
$x + 0 \rightarrow x$	not valid if x is -0
x - 0 $\rightarrow x$	not valid if x is ± 0 and rounding is towards -$\propto$
-$x \leftrightarrow 0$ - x	not valid if x is +0
x - $x \rightarrow 0$	not valid if x is NaN or $\pm\propto$
x - $y \leftrightarrow$ -(y - x)	not valid because (1-1=+0) whereas -(1-1)=-0
x * 0 $\rightarrow$ 0	not valid if x is NaN or $\pm\propto$
x / $c \leftrightarrow x$ * (1/c)	valid if (1/c) yields an exact result
x != $x \rightarrow$ false	not valid if x is a NaN
x == $x \rightarrow$ true	not valid if x is a NaN
x !op $y \leftrightarrow$!(x op y)	not valid if x or y is a NaN

Of course, transformations that would alter side effects are also invalid.

Chapter 30

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.

Implementations of D on different architectures, however, are free to innovate upon the memory model, function call/return conventions, argument passing conventions, etc.

This document describes the `x86` and `x86_64` implementations of the inline assembler. The inline assembler platform support that a compiler provides is indicated by the `D_InlineAsm_X86` and `D_InlineAsm_X86_64` version identifiers, respectively.

30.1 Asm statement

AsmStatement:
    `asm` *FunctionAttributes*$_{\text{opt}}$ `{` *AsmInstructionList*$_{\text{opt}}$ `}`

AsmInstructionList:
    AsmInstruction `;`
    AsmInstruction `;` *AsmInstructionList*

Assembler instructions must be located inside an `asm` block. Like functions, `asm` statements must be anotated with adequate function attributes to be compatible with the caller. Asm statements attributes must be explicitly defined, they are not infered.

```
void func1() pure nothrow @safe @nogc
{
```

```
    asm pure nothrow @trusted @nogc
    {}
}

void func2() @safe @nogc
{
    asm @nogc // Error: asm statement is assumed to be @system - mark it with '@trusted' if it is
    {}
}
```

30.2 Asm instruction

```
AsmInstruction:
    Identifier : AsmInstruction
    align IntegerExpression
    even
    naked
    db Operands
    ds Operands
    di Operands
    dl Operands
    df Operands
    dd Operands
    de Operands
    db StringLiteral
    ds StringLiteral
    di StringLiteral
    dl StringLiteral
    dw StringLiteral
    dq StringLiteral
    Opcode
    Opcode Operands
```

```
Operands:
    Operand
    Operand , Operands
```

30.3 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
}
```

30.4 align *IntegerExpression*

```
IntegerExpression:
    IntegerLiteral
    Identifier
```

Causes the assembler to emit NOP instructions to align the next assembler instruction on an *IntegerExpression* boundary. *IntegerExpression* must evaluate at compile time 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.

30.5 even

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

30.6 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.

30.7 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
}
```

30.8 Opcodes

A list of supported opcodes is at the end.

The following registers are supported. Register names are always in upper case.

```
Register:
   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
```

`x86_64` adds these additional registers.

Register64:

```
RAX  RBX  RCX  RDX
BPL  RBP
SPL  RSP
DIL  RDI
SIL  RSI
R8B  R8W  R8D  R8
R9B  R9W  R9D  R9
R10B R10W R10D R10
R11B R11W R11D R11
R12B R12W R12D R12
R13B R13W R13D R13
R14B R14W R14D R14
R15B R15W R15D R15
XMM8 XMM9 XMM10 XMM11 XMM12 XMM13 XMM14 XMM15
YMM0 YMM1 YMM2  YMM3  YMM4  YMM5  YMM6  YMM7
YMM8 YMM9 YMM10 YMM11 YMM12 YMM13 YMM14 YMM15
```

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

```
asm
{
```

```
        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
```

30.9 Operands

```
Operand:
    AsmExp


AsmExp:
    AsmLogOrExp
    AsmLogOrExp ? AsmExp : AsmExp


AsmLogOrExp:
    AsmLogAndExp
    AsmLogOrExp || AsmLogAndExp


AsmLogAndExp:
    AsmOrExp
    AsmLogAndExp && AsmOrExp


AsmOrExp:
    AsmXorExp
    AsmOrExp | AsmXorExp


AsmXorExp:
```

```
   AsmAndExp
   AsmXorExp ^ AsmAndExp

AsmAndExp:
   AsmEqualExp
   AsmAndExp & AsmEqualExp

AsmEqualExp:
   AsmRelExp
   AsmEqualExp == AsmRelExp
   AsmEqualExp != AsmRelExp

AsmRelExp:
   AsmShiftExp
   AsmRelExp < AsmShiftExp
   AsmRelExp <= AsmShiftExp
   AsmRelExp > AsmShiftExp
   AsmRelExp >= AsmShiftExp

AsmShiftExp:
   AsmAddExp
   AsmShiftExp << AsmAddExp
   AsmShiftExp >> AsmAddExp
   AsmShiftExp >>> AsmAddExp

AsmAddExp:
   AsmMulExp
   AsmAddExp + AsmMulExp
   AsmAddExp - AsmMulExp

AsmMulExp:
   AsmBrExp
   AsmMulExp * AsmBrExp
   AsmMulExp / AsmBrExp
```

```
    AsmMulExp % AsmBrExp

AsmBrExp:
    AsmUnaExp
    AsmBrExp [ AsmExp ]

AsmUnaExp:
    AsmTypePrefix AsmExp
    offsetof AsmExp
    seg AsmExp
    + AsmUnaExp
    - AsmUnaExp
    ! AsmUnaExp
    ~ AsmUnaExp
    AsmPrimaryExp

AsmPrimaryExp:
    IntegerLiteral
    FloatLiteral
    __LOCAL_SIZE
    $
    Register
    Register : AsmExp
    Register64
    Register64 : AsmExp
    DotIdentifier
    this

DotIdentifier:
    Identifier
    Identifier . DotIdentifier
    BasicTypeX . Identifier
```

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.

The `seg` means load the segment number that the symbol is in. This is not relevant for flat model code. Instead, do a move from the relevant segment register.

A dotted expression is evaluated during the compilation and then must either give a constant or indicate a higher level variable that fits in the target register or variable.

Operand Types

AsmTypePrefix:
```
    near ptr
    far ptr
    word ptr
    dword ptr
    qword ptr
    BasicTypeX 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  ;
```

`far ptr` is not relevant for flat model code.

Struct/Union/Class Member Offsets

To access members of an aggregate, given a pointer to the aggregate is in a register, use the `.offsetof` property of the qualified name of the member:

```
struct Foo { int a,b,c; }
int bar(Foo *f)
{
    asm
    {
        mov EBX,f                   ;
        mov EAX,Foo.b.offsetof[EBX] ;
    }
}
```

```
void main()
{
    Foo f = Foo(0, 2, 0);
    assert(bar(&f) == 2);
}
```

Alternatively, inside the scope of an aggregate, only the member name is needed:

```
struct Foo   // or class
{
    int a,b,c;
    int bar()
    {
        asm
        {
            mov EBX, this   ;
            mov EAX, b[EBX] ;
        }
    }
}
void main()
{
    Foo f = Foo(0, 2, 0);
    assert(f.bar() == 2);
}
```

Stack Variables

Stack variables (variables local to a function and allocated on the stack) are accessed via the name of the variable indexed by EBP:

```
int foo(int x)
{
    asm
    {
        mov EAX,x[EBP] ; // loads value of parameter x into EAX
        mov EAX,x      ; // does the same thing
    }
}
```

If the [EBP] is omitted, it is assumed for local variables. If `naked` is used, this no longer holds.

Special Symbols

$ Represents the program counter of the start of the next instruction. So,

```
jmp $ ;
```

branches to the instruction following the jmp instruction. The $ can only appear as the target of a jmp or call 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.

30.10 Opcodes Supported

Table 30.1: Opcodes

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	cmpxchg	cmpxchg8b	cmpxchg16b	
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

↩

(continued)

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
iretq	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

↩

(continued)

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

↩

(continued)

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
syscall	sysenter	sysexit	sysret	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

Table 30.2: Pentium 4 Opcodes

addsubpd	addsubps	fisttp	haddpd	haddps
hsubpd	hsubps	lddqu	monitor	movddup
movshdup	movsldup	mwait		

AMD Opcodes Supported

Table 30.3: AMD Opcodes

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

SIMD

SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2 and AVX are supported.

Chapter 31

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.

31.1 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 and Escape 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.

Section names starting with 'http://' or 'https://' are not recognized as section names.

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:
 * --------------------
 * writeln("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(const(char)[] filename) { ... }
```

See_Also: List of other symbols and URLs 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(string 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 =   <font color="magenta">$0</font>
 */
```

Escapes= The escapes section is a series of substitutions which replace special characters with a string. It's useful when the output format requires escaping of certain characters, for example in HTML **&** should be escaped with **&**. The syntax is /**c**/**string**/, where **c** is either a single character, or multiple characters separated by whitespace or commas, and **string** is the replacement text.

```
/**
 * ESCAPES = /&/AddressOf!/
 *           /!/Exclamation/
 *           /?/QuestionMark/
 *           /,/Comma/
 *           /{ }/Parens/
 *           /<,>/Arrows/
 */
```

31.2 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 beginning with at least three hyphens `-`, backticks ` or tildes `~` (ignoring whitespace) to delineate the code section:

```
/++
 + Our function.
 +
 + Example:
 + ---
 + import std.stdio;
 +
 + void foo()
 + {
 +     writeln("foo!");  /* print the string */
 + }
 + ---
 +/
```

Note that in the above example the documentation comment uses the /++ ... +/ form so that /* ... */ can be used inside the code section.

D code gets automatic syntax highlighting. To include code in another language without syntax highlighting, add a language string at the end of the top delimiter line:

```
/++
 + Some C++
 + ``` cpp
 + #include <iostream>
 +
 + void foo()
 + {
 +     std::cout << "foo!";
 + }
 + ```
 +/
```

Enable support for backticks, tildes and other code languages with the `-preview=markdown` compiler flag.

Inline Code

Inline code can be written between backtick characters (‘), similarly to the syntax used on GitHub, Reddit, Stack Overflow, and other websites. Both the opening and closing ‘ character must appear on the same line to trigger this behavior.

Text inside these sections will be escaped according to the rules described above, then wrapped in a `$(DDOC_BACKQUOTED)` macro. By default, this macro expands to be displayed as an inline text span, formatted as code.

A literal backtick character can be output either as a non-paired ‘ on a single line or by using the `$(BACKTICK)` macro.

```
/// Returns `true` if `a == b`.
void foo() {}

/// Backquoted `<html>` will be displayed to the user instead
/// of passed through as embedded HTML (see below).
void bar() {}
```

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:
 *
 * <ol>
 *   <li><a href="http://www.digitalmars.com">Digital Mars</a></li>
 *   <li><a href="http://www.classicempire.com">Empire</a></li>
 * </ol>
 */
```

Headings

Enable this feature with the `-preview=markdown` compiler flag.

A long documentation section can be subdivided by adding headings. A heading is a line of text that starts with one to six `#` characters followed by whitespace and then the heading text. The number of `#` characters determines the heading level. Headings may optionally end with any number of trailing `#` characters.

```
/**
 * # H1
 * ## H2
 * ### H3
 * #### H4 ###
 * ##### H5 ##
 * ###### H6 #
 */
```

Links

Enable this feature with the `-preview=markdown` compiler flag.

Documentation may link to other documentation or to a URL. There are four styles of links:

```
/**
 * Some links:
 *
 * 1. A [reference link][ref] and bare reference links: [ref] or [Object]
 * 2. An [inline link](https://dlang.org)
 * 3. A bare URL: https://dlang.org
 * 4. An ![image](https://dlang.org/images/d3.png)
 *
 * [ref]: https://dlang.org "The D Language Website"
 */
```

Reference Links

Reference-style links enclose a reference label in square brackets. They may optionally be preceeded by some link text, also enclosed in square brackets.

The reference label must match a reference defined elsewhere. This may be a D symbol in scope of the source code being documented, like `[Object]` in the example above, or it may be an explicit reference that is defined in the same documentation comment, like `[ref]` in the example above. In the example both instances of `[ref]` in item `1.` will be replaced with the URL and title text from the matching definition at the bottom of the example. The first link will read `reference link` and the second will read `ref`.

Reference definitions start with a label in square brackets, followed by a colon, a URL and an optional title wrapped in single or double quotes, or in parentheses. If a reference label would match both a D symbol and a reference definition then the reference definition is used.

The generated links to D symbols are relative if they have the same root package as the module being documented. If not, their URLs are preceeded by a `$(DDOC_ROOT_pkg)` macro, where `pkg` is the root package of the symbol being linked to. Links to D symbols are generated with a `$(DOC_EXTENSION)` macro after the module name. So the generated URL for `[Object]` in the above example is as if you had written:

```
$(DOC_ROOT_object)object$(DOC_EXTENSION)#.Object
```

You may define your own `DOC_ROOT_` macros for any external packages you wish to link to using a Macros section.

Inline Links

Inline-style links enclose link text in square brackets and the link URL in parentheses. Like reference links, the URL may optionally be followed by title text wrapped in single or double quotes, or in parentheses:

```
/// [a link with title text](https://dlang.org 'Some title text')
```

Bare URLs

Bare URLs are sequences of characters that start with `http://` or `https://`, continue with one or more characters from the set of letters, digits and `-_?=%&/+#~.`, and contain at least one period. URL recognition happens before all macro text substitution. The URL is wrapped in a `$(DDOC_LINK_AUTODETECT)` macro and is otherwise left untouched.

Images

Images have the same form as reference or inline links, but add an exclamation point `!` before the initial square bracket. What would be the link text in a normal link is used as the image's alt text.

Lists

Enable this feature with the `-preview=markdown` compiler flag.

Documentation may contain lists. Start an ordered list with a number followed by a period:

```
/**
 * 1. First this
 * 2. Then this
 *    1. A sub-item
```

```
 */
```

Start an unordered list with a hypen (-), an asterisk (*) or a plus (+). Subsequent items in the same list must also start with the same symbol:

```
/**
 * - A list
 * - With a second item
 *
 * + A different list
 *   - With a sub-item
 *
 * * A third list (note the double asterisks)
 */
```

Note the double asterisks in the example above. This is because the list is inside a documentation comment that is delimited with asterisks, so the initial asterisk is considered part of the documentation comment, not a list item. This is even true when other lines don't start with an asterisk:

```
/**
 - A list
 * Not a list because the asterisk is part of the documentation comment
 */

/++
 + + The caveat also applies to plus-delimited documentation comments
 +/
```

List items can include content like new paragraphs, headings, embedded code, or child list items. Simply indent the content to match the indent of the text after the list symbol:

```
/**
 * - A parent list item
 *
 *   With a second paragraph
 *
 *   - A sub-item
 *     ---
 *     // A code example inside the sub-item
 *     ---
 */
```

Tables

Enable this feature with the `-preview=markdown` compiler flag.

Data may be placed into a table. Tables consist of a single header row, a delimiter row, and zero or more data rows. Cells in each row are separated by pipe (|) characters. Initial and trailing |'s are optional. The number of cells in the delimiter row must match the number of cells in the header row:

```
/**
 *  | Item | Price |
 *  | ---- | ----: |
 *  | Wigs | $10 |
 *    Wheels | $13
 *  | Widgets | $200 |
 */
```

Cells in the delimiter row contain hyphens (`-`) and optional colons (`:`). A `:` to the left of the hyphens creates a left-aligned column, a `:` to the right of the hyphens creates a right-aligned column (like the example above), and `:`'s on both sides of the hyphens create a center-aligned column.

Quotes

Enable this feature with the `-preview=markdown` compiler flag.

Documentation may include a section of quoted material by prefixing each line of the section with a `>`. Quotes may include headings, lists, embedded code, etc.

```
/**
 * > To D, or not to D. -- Willeam NerdSpeare
 */
```

Lines of text that directly follow a quoted line are considered part of the quote:

```
/**
 * > This line
 * and this line are both part of the quote
 *
 * This line is not part of the quote.
 */
```

Horizontal Rules

Enable this feature with the `-preview=markdown` compiler flag.

Create a horizontal rule by adding a line containing three or more asterisks, underscores or hyphens:

```
/**
 * ***
 * ___
 */
```

As with lists, note that the initial `*` in the example above will be stripped because it is part of a documentation comment that is delimited with asterisks, so you need at least three subsequent asterisks.

To create a horizontal rule with hyphens, add spaces between the hyphens. Without the spaces they would be treated as the start or end of an embedded code block. Note that any horizontal rule may contain spaces:

```
/**
 * - - -
 * _ _ _
 * * * *
 */
```

Text Emphasis

Enable this feature with the `-preview=markdown` compiler flag.

A span of text wrapped in asterisks (`*`) is emphasized, and text wrapped in two asterisks (`**`) is strongly emphasized:

`*single asterisks*` is rendered as single asterisks.

`**double asterisks**` is rendered as double asterisks.

You can insert a literal asterisk by backslash-escaping it: `\*` is rendered as *.

Unlike Markdown, underscores (`\_`) are not supported for emphasizing text because it would break snake_case names and underscore prefix processing in identifier emphasis.

Identifier 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:

Characters and Entities

Character	Entity
`<`	<
`>`	>
`&`	&

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.

Punctuation Escapes

Enable this feature with the `-preview=markdown` compiler flag.

You can escape any ASCII punctuation symbol with a backslash `\`. Doing so outputs the original character without the backslash, except for the following characters which output predefined macros instead:

Characters and Escape Macros

Character	Macro
`(`	$(LPAREN)
`)`	$(RPAREN)
`,`	$(COMMA)
`$`	$(DOLLAR)

To output a backslash, simply use two backslashes in a row: `\\`. Note that backslashes inside embedded or inline code do not escape punctuation and are included in the output as-is. Backslashes before non-punctation are also included in the output as-is. For example, `C:\dmd2\bin\dmd.exe` does not require escaping its embedded backslashes.

No Documentation

No documentation is generated for the following constructs, even if they have a documentation comment:

- Invariants
- Postblits
- Destructors
- Static constructors and static destructors
- Class info, type info, and module info

31.3 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.

For example:

```
/**
Macros:
 PARAM = <u>$1</u>
 MATH_DOCS = <a href="https://dlang.org/phobos/std_math.html">Math Docs</a>
*/
module math;

/**
 * This function returns the sum of $(PARAM a) and $(PARAM b).
 * See also the $(MATH_DOCS).
 */
int sum(int a, int b) { return a + b; }
```

The above would generate the following output:

```
<h1>test</h1>
<dl><dt><big><a name="sum"></a>int <u>sum</u>(int <i>a</i>, int <i>b</i>);
</big></dt>
<dd>This function returns the <u>sum</u> of <u><i>a</i></u> and <u><i>b</i></u>.
 See also the <a href="https://dlang.org/phobos/std_math.html">Math Docs</a>.
</dd>
</dl>
```

The replacement text is 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 backslash-escaped: `$`.

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 backslash-escaped: `\(` or `\)`.

Macro definitions come from the following sources, in the specified order:

1. Predefined macros.
2. Definitions from file specified by sc.ini's or dmd.conf 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

A number of macros are predefined Ddoc, and represent the minimal definitions needed by Ddoc to format and highlight the presentation. The definitions are for simple HTML.

The implementations of all predefined macros are implementation-defined. The reference implementation's macro definitions can be found here.

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.

Table 31.1: Predefined Formatting Macros

Name	Description
`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

↩

(continued)

Name	Description
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
D_INLINECODE	argument is inline D code
LF	Insert a line feed (newline)
LPAREN	Insert a left parenthesis
RPAREN	Insert a right parenthesis
BACKTICK	Insert a backtick
DOLLAR	Insert a dollar sign
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>
```

Highlighting of D code is performed by the following macros:

Table 31.2: D Code Formatting Macros

Name	Description
`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.

Table 31.3: Ddoc Section Formatting Macros

Name	Description
`DDOC_CONSTRAINT`	Highlighting of a template constraint.
`DDOC_COMMENT`	Inserts a comment in the output.
`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`	Highlighting of the authors section.
`DDOC_BUGS`	Highlighting of the bugs section.
`DDOC_COPYRIGHT`	Highlighting of the copyright section.
`DDOC_DATE`	Highlighting of the date section.
`DDOC_DEPRECATED`	Highlighting of the deprecated section.
`DEPRECATED`	Wrapper for deprecated declarations.
`DDOC_EXAMPLES`	Highlighting of the examples section.
`DDOC_HISTORY`	Highlighting of the history section.
`DDOC_LICENSE`	Highlighting of the license section.
`DDOC_OVERLOAD_SEPARATOR`	Inserts a separator between overloads of a given name.
`DDOC_RETURNS`	Highlighting of the returns section.
`DDOC_SEE_ALSO`	Highlighting of the see-also section.
`DDOC_STANDARDS`	Highlighting of the standards section.
`DDOC_THROWS`	Highlighting of the throws section.
`DDOC_VERSION`	Highlighting of the version section.

↩

(continued)

Name	Description
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_PARAM	Highlighting of a template's individual parameters.
DDOC_TEMPLATE_PARAM_LIST	Highlighting of a template's parameter list.
DDOC_TEMPLATE_MEMBERS	Highlighting of all the members of a template.
DDOC_ENUM_BASETYPE	Highlighting of the type an enum is based upon
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_BLANKLINE	Inserts a blank line.
DDOC_ANCHOR	Expands to a named anchor used for hyperlinking to a particular declaration section. Argument $1 expands to the qualified declaration name.
DDOC_PSYMBOL	Highlighting of declaration name to which a particular section is referring.
DDOC_PSUPER_SYMBOL	Highlighting of the base type of a class.
DDOC_KEYWORD	Highlighting of D keywords.
DDOC_PARAM	Highlighting of function parameters.
DDOC_BACKQUOTED	Inserts inline code.
DDOC_AUTO_PSYMBOL_SUPPRESS	Highlighting of auto-detected symbol that starts with underscore
DDOC_AUTO_PSYMBOL	Highlighting of auto-detected symbol
DDOC_AUTO_KEYWORD	Highlighting of auto-detected keywords
DDOC_AUTO_PARAM	Highlighting of auto-detected parameters

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

Table 31.4: Generated Macro Definitions

Macro Name	Content
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.
DOCFILENAME	Set to the name of the generated output file.
SRCFILENAME	Set to the name of the source file the documentation is being generated from.

31.4 Using Ddoc to generate examples from unit tests

Ddoc can automatically generate usage examples for declarations using unit tests. If a declaration is followed by a documented unit test, the code from the test will be inserted into the example section of the declaration. This avoids the frequent problem of having outdated documentation for pieces of code.

To create a documented unit test just add three forward slashes before the unittest block, like this:

```
///
unittest
{
```

```
    ...
}
```

For more information please see the full section on documented unit tests.

31.5 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.

31.6 Security considerations

Note that DDoc comments may embed raw HTML, including <script> tags. Be careful when publishing or distributing rendered DDoc HTML generated from untrusted sources, as this may allow cross-site scripting.

31.7 Links to D documentation generators

A list of current D documentation generators which use Ddoc can be found on our wiki page.

Chapter 32

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.

32.1 Calling C Functions

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

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

```
extern (C) int strcmp(const char* string1, const 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 extended type modifiers in D. These are handled by linkage attributes, such as `extern (C)`.
- There is no volatile type modifier in D. To declare a C function that uses volatile, just drop the keyword 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)
    {
        ...
    }
}
```

32.2 Storage Allocation

C code explicitly manages memory with calls to malloc() and free(). D allocates memory using the D garbage collector, so no explicit frees are necessary.

D can still explicitly allocate memory using core.stdc.stdlib.malloc() and core.stdc.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 core.stdc.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
  std.gc.addRoot() or
  std.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 DLLs, etc.

32.3 Data Type Compatibility

D And C Type Equivalence

D	C 32 bit	C 64 bit
void	void	
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	
ulong	unsigned long long	unsigned long
core.stdc.config.c_long	long	long
core.stdc.config.c_ulong	unsigned long	unsigned long
long	long long	long (or long long)
ulong	unsigned long long	unsigned long (or unsigned long long)
float	float	
double	double	
real	long double	
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	
type1[type2]	no equivalent	
type function(params)	type(*)(params)	
type delegate(params)	no equivalent	
size_t	size_t	
ptrdiff_t	ptrdiff_t	

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

32.4 Passing D Array Arguments to C Functions

In C, arrays are passed to functions as pointers even if the function prototype says its an array. In D, static arrays are passed by value, not by reference. Thus, the function prototype must be adjusted to match what C expects.

D And C Function Prototype Equivalence

D type	C type
*T**	*T*[]
ref *T*[*dim*]	*T*[*dim*]

For example:

```
void foo(int a[3]) { ... } // C code

extern (C)
{
    void foo(ref int[3] a); // D prototype
}
```

32.5 Calling `printf()`

`printf` can be directly called from D code:

```
import core.stdc.stdio;

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

Printing values works as it does in C:

```
int apples;
printf("there␣are␣%d␣apples\n", apples);
```

Correctly matching the format specifier to the D type is necessary. The D compiler recognizes the printf formats and diagnoses mismatches with the supplied arguments. The specification for the formats used by D is the C99 specification 7.19.6.1.

A generous interpretation of what is a match between the argument and format specifier is taken, for example, an unsigned type can be printed with a signed format specifier. Diagnosed incompatibilites are:

- incompatible sizes which may cause argument misalignment
- dereferencing arguments that are not pointers
- insufficient number of arguments
- struct, array and slice arguments are not allowed
- non-pointer arguments to `s` specifier
- non-Standard formats
- undefined behavior per C99

Strings

A string cannot be printed directly. But `%.*s` can be used:

```
string s = "betty";
printf("hello␣%.*s\n", cast(int) s.length, s.ptr);
```

The cast to `int` is required.

`size_t` and `ptrdiff_t`

These use the `zd` and `dt` format specifiers respectively:

```
int* p, q;
printf("size␣of␣an␣int␣is␣%zt,␣pointer␣difference␣is␣%td\n", int.sizeof, p - q);
```

Non-Standard Format Specifiers

Non-Standard format specifiers will be rejected by the compiler. Since the checking is only done for formats as string literals, non-Standard ones can be used:

```
const char* format = "value:␣%K\n";
printf(format, value);
```

Modern Formatted Writing

An improved D function for formatted output is `std.stdio.writef()`.

32.6 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 #pragmas. 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, or use the std.bitmanip.bitfields library type. htod will convert bit fields to inline functions that do the right shift and masks.

D does not support declaring variables of anonymous struct types. In such a case you can define a named struct in D and make it private:

```
union Info  // C code
{
    struct
    {
        char *name;
    } file;
};

union Info  // D code
{
    private struct File
    {
        char* name;
    }
    File file;
}
```

32.7 Callbacks

D can easily call C callbacks (function pointers), and C can call callbacks provided by D code if the callback is an `extern(C)` function, or some other linkage that both sides have agreed to (e.g. `extern(Windows)`).

Here's an example of C code providing a callback to D code:

```
void someFunc(void *arg) { printf("Called␣someFunc!\n"); }  // C code
typedef void (*Callback)(void *);
extern "C" Callback getCallback(void)
{
```

```
    return someFunc;
}

extern(C) alias Callback = int function(int, int);  // D code
extern(C) Callback getCallback();
void main()
{
    Callback cb = getCallback();
    cb();  // invokes the callback
}
```

And an example of D code providing a callback to C code:

```
extern "C" void printer(int (*callback)(int, int))  // C code
{
    printf("calling␣callback␣with␣2␣and␣4␣returns:␣%d\n", callback(2, 4));
}

extern(C) alias Callback = int function(int, int);  // D code
extern(C) void printer(Callback callback);
extern(C) int sum(int x, int y) { return x + y; }
void main()
{
    printer(&sum);
}
```

For more info about callbacks read the closures section.

32.8 Using Existing C Libraries

Since D can call C code directly, it can also call any C library functions, giving D access to the smorgasbord of existing C libraries. To do so, however, one needs to write a D interface (.di) file, which is a translation of the C .h header file for the C library into D.

For popular C libraries, the first place to look for the corresponding D interface file is the Deimos Project. If it isn't there already, and you write one, please contribute it to the Deimos Project.

32.9 Accessing C Globals

C globals can be accessed directly from D. C globals have the C naming convention, and so must be in an `extern (C)` block. Use the `extern` storage class to indicate that the global is allocated in the C code, not the D code. C globals default to being in global, not thread local, storage. To reference global storage from D, use the `__gshared` storage class.

```
extern (C) extern __gshared int x;
```

Chapter 33

Interfacing to C++

This document specifies how to interface with C++ directly.

It is also possible to indirectly interface with C++ code, either through a C interface or a COM interface.

33.1 The General Idea

Being 100a fully functional C++ compiler front end to D. Anecdotal evidence suggests that writing such is a minimum of a 10 man-year project, essentially making a D compiler with such capability unimplementable. Other languages looking to hook up to C++ face the same problem, and the solutions have been:

1. Support the COM interface (but that only works for Windows).
2. Laboriously construct a C wrapper around the C++ code.
3. Use an automated tool such as SWIG to construct a C wrapper.
4. Reimplement the C++ code in the other language.
5. Give up.

D takes a pragmatic approach that assumes a couple modest accommodations can solve a significant chunk of the problem:

- matching C++ name mangling conventions
- matching C++ function calling conventions
- matching C++ virtual function table layout for single inheritance

33.2 Global Functions

C++ global functions, including those in namespaces, can be declared and called in D, or defined in D and called in C++.

Calling C++ Global Functions from D

Given a C++ function in a C++ source file:

```
#include <iostream>

using namespace std;

int foo(int i, int j, int k)
{
    cout << "i␣=␣" << i << endl;
    cout << "j␣=␣" << j << endl;
    cout << "k␣=␣" << k << endl;

    return 7;
}
```

In the corresponding D code, foo is declared as having C++ linkage and function calling conventions:

```
extern (C++) int foo(int i, int j, int k);
```

and then it can be called within the D code:

```
extern (C++) int foo(int i, int j, int k);

void main()
{
    foo(1, 2, 3);
}
```

Compiling the two files, the first with a C++ compiler, the second with a D compiler, linking them together, and then running it yields:

```
> g++ -c foo.cpp
> dmd bar.d foo.o -L-lstdc++ && ./bar
i = 1
j = 2
k = 3
```

There are several things going on here:

- D understands how C++ function names are "mangled" and the correct C++ function call/return sequence.
- Because modules are not part of C++, each function with C++ linkage in the global namespace must be globally unique within the program.
- There are no `__cdecl`, `__far`, `__stdcall`, `__declspec`, or other such nonstandard C++ extensions in D.
- There are no volatile type modifiers in D.
- Strings are not 0 terminated in D. See "Data Type Compatibility" for more information about this. However, string literals in D are 0 terminated.

Calling Global D Functions From C++

To make a D function accessible from C++, give it C++ linkage:

```
import std.stdio;

extern (C++) int foo(int i, int j, int k)
{
    writefln("i = %s", i);
    writefln("j = %s", j);
    writefln("k = %s", k);
    return 1;
}

extern (C++) void bar();

void main()
{
    bar();
}
```

The C++ end looks like:

```
int foo(int i, int j, int k);

void bar()
{
    foo(6, 7, 8);
}
```

Compiling, linking, and running produces the output:

```
> dmd -c foo.d
> g++ bar.cpp foo.o -lphobos2 -pthread -o bar && ./bar
i = 6
j = 7
k = 8
```

33.3 C++ Namespaces

C++ symbols that reside in namespaces can be accessed from D. A namespace can be added to the `extern (C++)` LinkageAttribute:

```
extern (C++, N) int foo(int i, int j, int k);

void main()
{
    N.foo(1, 2, 3);   // foo is in C++ namespace 'N'
}
```

33.4 Classes

C++ classes can be declared in D by using the `extern (C++)` attribute on `class`, `struct` and `interface` declarations. `extern (C++)` interfaces have the same restrictions as D interfaces, which means that Multiple Inheritance is supported to the extent that only one base class can have member fields.

`extern (C++)` structs do not support virtual functions but can be used to map C++ value types.

Unlike classes and interfaces with D linkage, `extern (C++)` classes and interfaces are not rooted in `Object` and cannot be used with `typeid`.

D structs and classes have different semantics whereas C++ structs and classes are basically the same. The use of a D struct or class depends on the C++ implementation and not on the used C++ keyword. When mapping a D `class` onto a C++ `struct`, use `extern(C++, struct)` to avoid linking problems with C++ compilers (notably MSVC) that distinguish between C++'s `class` and `struct` when mangling. Conversely, use `extern(C++, class)` to map a D `struct` onto a C++ `class`.

`extern(C++, class)` and `extern(C++, struct)` can be combined with C++ namespaces:

```
extern (C++, struct) extern (C++, foo)
class Bar
```

```
{
}
```

Using C++ Classes From D

The following example shows binding of a pure virtual function, its implementation in a derived class, a non-virtual member function, and a member field:

```
#include <iostream>

using namespace std;

class Base
{
    public:
        virtual void print3i(int a, int b, int c) = 0;
};

class Derived : public Base
{
    public:
        int field;
        Derived(int field) : field(field) {}

        void print3i(int a, int b, int c)
        {
            cout << "a = " << a << endl;
            cout << "b = " << b << endl;
            cout << "c = " << c << endl;
        }

        int mul(int factor);
};

int Derived::mul(int factor)
{
    return field * factor;
}

Derived *createInstance(int i)
```

```
{
    return new Derived(i);
}

void deleteInstance(Derived *&d)
{
    delete d;
    d = 0;
}
```

We can use it in D code like:

```
extern(C++)
{
    abstract class Base
    {
        void print3i(int a, int b, int c);
    }

    class Derived : Base
    {
        int field;
        @disable this();
        override void print3i(int a, int b, int c);
        final int mul(int factor);
    }

    Derived createInstance(int i);
    void deleteInstance(ref Derived d);
}

void main()
{
    import std.stdio;

    auto d1 = createInstance(5);
    writeln(d1.field);
    writeln(d1.mul(4));

    Base b1 = d1;
    b1.print3i(1, 2, 3);
```

```
    deleteInstance(d1);
    assert(d1 is null);

    auto d2 = createInstance(42);
    writeln(d2.field);

    deleteInstance(d2);
    assert(d2 is null);
}
```

Compiling, linking, and running produces the output:

```
> g++ base.cpp
> dmd main.d base.o -L-lstdc++ && ./main
5
20
a = 1
b = 2
c = 3
42
```

Note how in the above example, the constructor is not bindable and is instead disabled on the D side; an alternative would be to reimplement the constructor in D. See the section below on lifetime management for more information.

Using D Classes From C++

Given D code like:

```
extern (C++) int callE(E);

extern (C++) interface E
{
    int bar(int i, int j, int k);
}

class F : E
{
    extern (C++) int bar(int i, int j, int k)
    {
        import std.stdio : writefln;
```

```
        writefln("i = %s", i);
        writefln("j = %s", j);
        writefln("k = %s", k);
        return 8;
    }
}

void main()
{
    F f = new F();
    callE(f);
}
```

The C++ code to access it looks like:

```
class E
{
  public:
    virtual int bar(int i, int j, int k);
};


int callE(E *e)
{
    return e->bar(11, 12, 13);
}
```

```
> dmd -c base.d
> g++ klass.cpp base.o -lphobos2 -pthread -o klass && ./klass
i = 11
j = 12
k = 13
```

33.5 Structs

C++ allows a struct to inherit from a base struct. This is done in D using `alias this`:

```
struct Base { ... members ... };

struct Derived
{
```

```
    Base base;        // make it the first field
    alias base this;

    ... members ...
}
```

In both C++ and D, if a struct has zero fields, the struct still has a size of 1 byte. But, in C++ if the struct with zero fields is used as a base struct, its size is zero (called the Empty Base Optimization). There are two methods for emulating this behavior in D. The first forwards references to a function returning a faked reference to the base:

```
struct Base { ... members ... };

struct DerivedStruct
{
    static if (Base.tupleof.length > 0)
        Base base;
    else
        ref inout(Base) base() inout
        {
            return *cast(inout(Base)*)&this;
        }
    alias base this;

    ... members ...
}
```

The second makes use of template mixins:

```
mixin template BaseMembers()
{
    void memberFunction() { ... }
}

struct Base
{
    mixin BaseMembers!();
}

struct Derived
{
    mixin BaseMembers!();
```

```
    ... members ...
}
```

Note that the template mixin is evaluated in the context of its instantiation, not declaration. If this is a problem, the template mixin can use local imports, or have the member functions forward to the actual functions.

33.6 C++ Templates

C++ function and type templates can be bound by using the `extern (C++)` attribute on a function or type template declaration.

Note that all instantiations used in D code must be provided by linking to C++ object code or shared libraries containing the instantiations.

For example:

```
#include <iostream>

template<class T>
struct Foo
{
    private:
    T field;

    public:
    Foo(T t) : field(t) {}
    T get();
    void set(T t);
};

template<class T>
T Foo<T>::get()
{
    return field;
}

template<class T>
void Foo<T>::set(T t)
{
    field = t;
}
```

```
Foo<int> makeIntFoo(int i)
{
    return Foo<int>(i);
}

Foo<char> makeCharFoo(char c)
{
    return Foo<char>(c);
}

template<class T>
void increment(Foo<T> &foo)
{
    foo.set(foo.get() + 1);
}

template<class T>
void printThreeNext(Foo<T> foo)
{
    for(size_t i = 0; i < 3; ++i)
    {
        std::cout << foo.get() << std::endl;
        increment(foo);
    }
}

// The following two functions ensure that the required instantiations of
// printThreeNext are provided by this code module
void printThreeNexti(Foo<int> foo)
{
    printThreeNext(foo);
}

void printThreeNextc(Foo<char> foo)
{
    printThreeNext(foo);
}

extern(C++):
```

```
struct Foo(T)
{
    private:
    T field;

    public:
    @disable this();
    T get();
    void set(T t);
}

Foo!int makeIntFoo(int i);
Foo!char makeCharFoo(char c);
void increment(T)(ref Foo!T foo);
void printThreeNext(T)(Foo!T foo);

extern(D) void main()
{
    auto i = makeIntFoo(42);
    assert(i.get() == 42);
    i.set(1);
    increment(i);
    assert(i.get() == 2);

    auto c = makeCharFoo('a');
    increment(c);
    assert(c.get() == 'b');

    c.set('A');
    printThreeNext(c);
}
```

Compiling, linking, and running produces the output:

```
> g++ -c template.cpp
> dmd main.d template.o -L-lstdc++ && ./main
A
B
C
```

33.7 Function Overloading

C++ and D follow different rules for function overloading. D source code, even when calling `extern (C++)` functions, will still follow D overloading rules.

33.8 Memory Allocation

C++ code explicitly manages memory with calls to `::operator new()` and `::operator delete()`. D's `new` operator allocates memory using the D garbage collector, so no explicit delete is necessary. D's `new` operator is not compatible with C++'s `::operator new` and `::operator delete`. Attempting to allocate memory with D's `new` and deallocate with C++ `::operator delete` will result in miserable failure.

D can explicitly manage memory using a variety of library tools, such as with `std.experimental.allocator`. Additionally, `core.stdc.stdlib.malloc` and `core.stdc.stdlib.free` can be used directly for connecting to C++ functions that expect `malloc`'d buffers.

If pointers to memory allocated on the D garbage collector heap are passed to C++ functions, it's critical to ensure that the referenced memory will not be collected by the D garbage collector before the C++ function is done with it. This is accomplished by:

- Making a copy of the data using `std.experimental.allocator` or `core.stdc.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 using the `core.memory.GC.addRoot` or `core.memory.GC.addRange` functions.

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 registered with the D runtime, nor does it scan the data segments of shared libraries that aren't registered with the D runtime.

33.9 Data Type Compatibility

D And C++ Type Equivalence

D type	C++ type
void	**void**
byte	**signed char**
ubyte	**unsigned char**
char	**char** (chars are unsigned in D)
`core.stdc.stddef.wchar_t`	`wchar_t`
short	**short**
ushort	**unsigned short**
int	**int**
uint	**unsigned**
long	**long long**
ulong	**unsigned long long**
`core.stdc.config.cpp_long`	**long**
`core.stdc.config.cpp_ulong`	**unsigned long**
float	**float**
double	**double**
real	**long double**
`extern (C++)` **struct**	**struct** or **class**
`extern (C++)` **class**	**struct** or **class**
`extern (C++)` **interface**	**struct** or **class** with no member fields
union	**union**
enum	**enum**
type`*`	*type* `*`
ref *type* (in parameter lists only)	*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 when the D and C++ compilers used are companions on the host platform.

33.10 Packing and Alignment

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 #pragmas. 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 supports bitfields in the standard library: see `std.bitmanip.bitfields`.

33.11 Lifetime Management

C++ constructors, copy constructors, move constructors and destructors cannot be called directly in D code, and D constructors, postblit operators and destructors cannot be directly exported to C++ code. Interoperation of types with these special operators is possible by either 1) disabling the operator in the client language and only using it in the host language, or 2) faithfully reimplementing the operator in the client language. With the latter approach, care needs to be taken to ensure observable semantics remain the same with both implementations, which can be difficult, or in some edge cases impossible, due to differences in how the operators work in the two languages. For example, in D all objects are movable and there is no move constructor.

33.12 Special Member Functions

D cannot directly call C++ special member functions, and vice versa. These include constructors, destructors, conversion operators, operator overloading, and allocators.

33.13 Runtime Type Identification

D runtime type identification uses completely different techniques than C++. The two are incompatible.

33.14 Exception Handling

Exception interoperability is a work in progress.

At present, C++ exceptions cannot be caught in or thrown from D, and D exceptions cannot be caught in or thrown from C++. Additionally, objects in C++ stack frames are not guaranteed to be destroyed when unwinding the stack due to a D exception, and vice versa.

The plan is to support all of the above except throwing D exceptions directly in C++ code (but they will be throwable indirectly by calling into a D function with C++ linkage).

33.15 Comparing D Immutable and Const with C++ Const

Const, Immutable Comparison

Feature	D	C++98
`const` keyword	Yes	Yes
`immutable` keyword	Yes	No

const notation	`// Functional:`
	`//ptr to const ptr to const int`
	`const(int*)* p;`
		`// Postfix:`
		`//ptr to const ptr to const int`
		`const int *const *p;`

transitive const	`// Yes:`
	`//const ptr to const ptr to const int`
	`const int** p;`
	`**p = 3; // error`
		`// No:`
		`// const ptr to ptr to int`
		`int** const p;`
		`**p = 3;     // ok`

cast away const	`// Yes:`
	`// ptr to const int`
	`const(int)* p;`
	`int* q = cast(int*)p; // ok`
		`// Yes:`
		`// ptr to const int`
		`const int* p;`
		`int* q = const_cast<int*>p; //ok`

cast+mutate	`// No:`
	`// ptr to const int`
	`const(int)* p;`
	`int* q = cast(int*)p;`
	`*q = 3;   // undefined behavior`
		`// Yes:`
		`// ptr to const int`
		`const int* p;`
		`int* q = const_cast<int*>p;`
		`*q = 3;   // ok`

overloading	`// Yes:`
	`void foo(int x);`
	`void foo(const int x);  //ok`
		`// No:`
		`void foo(int x);`
		`void foo(const int x);  //error`

const/mutable aliasing	`// Yes:`
	`void foo(const int* x, int* y)`
	`{`
	`    bar(*x); // bar(3)`
	`    *y = 4;`
	`    bar(*x); // bar(4)`
	`}`
	`...`
	`int i = 3;`
	`foo(&i, &i);`
		`// Yes:`
		`void foo(const int* x, int* y)`
		`{`
		`    bar(*x); // bar(3)`
		`    *y = 4;`
		`    bar(*x); // bar(4)`
		`}`
		`...`
		`int i = 3;`
		`foo(&i, &i);`

immutable/mutable aliasing	`// No:`
	`void foo(immutable int* x, int* y)`
	`{`
	`    bar(*x); // bar(3)`
	`    *y = 4;  // undefined behavior`
	`    bar(*x); // bar(??)`
	`}`
	`...`
	`int i = 3;`
	`foo(cast(immutable)&i, &i);`
		No immutables

type of string literal	`immutable(char)[]`	`const char*`
string literal to non-const	not allowed	allowed, but deprecated

Chapter 34

Interfacing to Objective-C

D has some support for interfacing with Objective-C. It supports classes, subclasses, instance variables, instance and class methods. It is only available on macOS, compiling for 64bit.

Fully working example is available at the bottom.

34.1 Classes

Declaring an External Class

```
extern (Objective-C)
extern class NSString
{
    const(char)* UTF8String() @selector("UTF8String");
}
```

All Objective-C classes that should be accessible from within D need to be declared with the Objective-C linkage. If the class is declared as `extern` (in addition to `extern (Objective-C)`) it is expected to be defined externally.

The `@selector` attribute indicates which Objective-C selector should be used when calling this method. This attribute needs to be attached to all methods with the `Objective-C` linkage.

Defining a Class

```
// externally defined
extern (Objective-C)
extern class NSObject
{
    static NSObject alloc() @selector("alloc");
    NSObject init() @selector("init");
```

```
}

extern (Objective-C)
class Foo : NSObject
{
    override static Foo alloc() @selector("alloc");
    override Foo init() @selector("init");

    final int bar(int a) @selector("bar:")
    {
        return a;
    }
}

void main()
{
    assert(Foo.alloc.init.bar(3) == 3);
}
```

Defining an Objective-C class is exactly the same as declaring an external class but it should not be declared as `extern`.

To match the Objective-C semantics, `static` and `final` methods are virtual. `static` methods are overridable as well.

34.2 Instance Variables

```
// externally defined
extern (Objective-C)
extern class NSObject
{
    static NSObject alloc() @selector("alloc");
    NSObject init() @selector("init");
}

extern (Objective-C)
class Foo : NSObject
{
    int bar_;

    override static Foo alloc() @selector("alloc");
    override Foo init() @selector("init");
```

```
    int bar() @selector("bar")
    {
        return bar_;
    }
}

void main()
{
    auto foo = Foo.alloc.init;
    foo.bar_ = 3;
    assert(foo.bar == 3);
}
```

Declaring an instance variable looks exactly the same as for a regular D class.

To solve the fragile base class problem, instance variables in Objective-C has a dynamic offset. That means that the base class can change (add or remove instance variables) without the subclasses needing to recompile or relink. Thanks to this feature it's not necessary to declare instance variables when creating bindings to Objective-C classes.

34.3 Calling an Instance Method

Calling an Objective-C instance method uses the same syntax as calling regular D methods:

```
const(char)* result = object.UTF8String();
```

When the compiler sees a call to a method with Objective-C linkage it will generate a call similar to how an Objective-C compiler would call the method.

34.4 The `@selector` Attribute

The `@selector` attribute is a compiler recognized UDA. It is used to tell the compiler which selector to use when calling an Objective-C method.

Selectors in Objective-C can contain the colon character, which is not valid in D identifiers. D supports method overloading while Objective-C achieves something similar by using different selectors. For these two reasons it is better to be able to specify the selectors manually in D, instead of trying to infer it. This allows to have a more natural names for the methods in D. Example:

```
extern (Objective-C)
extern class NSString
{
    NSString initWith(in char*) @selector("initWithUTF8String:");
```

```
    NSString initWith(NSString) @selector("initWithString:");
}
```

Here the method `initWith` is overloaded with two versions, one accepting `in char*`, the other one `NSString`. These two methods are mapped to two different Objective-C selectors, `initWithUTF8String:` and `initWithString:`.

The attribute is defined in druntime in `core.attribute` and aliased in `object`, meaning it will be implicitly imported. The attribute is only defined when the version identifier `D_ObjectiveC` is enabled.

Compiler Checks

The compiler performs the following checks to enforce the correct usage of the `@selector` attribute:

- The attribute can only be attached to methods with Objective-C linkage
- The attribute can only be attached once to a method
- The attribute cannot be attached to a template method
- The number of colons in the selector needs to match the number of parameters the method is declared with

If any of the checks fail, a compile error will occur.

34.5 The `D_ObjectiveC` Version Identifier

The `D_ObjectiveC` version identifier is a predefined version identifier. It is enabled if Objective-C support is available for the target.

34.6 Objective-C Linkage

Objective-C linkage is achieved by attaching the `extern (Objective-C)` attribute to a class. Example:

```
extern (Objective-C)
extern class NSObject
{
    NSObject init() @selector("init");
}
```

All methods inside a class declared as `extern (Objective-C)` will get implicit Objective-C linkage.

The linkage is recognized on all platforms but will issue a compile error if it is used on a platform where Objective-C support is not available. This allows to easily hide Objective-C declarations from platforms where it is not available using the `version` statement, without resorting to string mixins or other workarounds.

34.7 Memory Management

The preferred way to do memory management in Objective-C is to use Automatic Reference Counting, ARC. This is not supported in D, therefore manual memory management is required to be used instead. This is achieved by calling `release` on an Objective-C instance, like in the old days of Objective-C.

34.8 Frameworks

Most Objective-C code is bundled in something called a "Framework". This is basically a regular directory, with the `.framework` extension and a specific directory layout. A framework contains a dynamic library, all public header files and any resources (images, sounds and so on) required by the framework.

These directories are recognized by some tools, like the Objective-C compiler and linker, to be frameworks. To link with a framework from DMD, use the following flags:

```
-L-framework -L<Framework>
```

where `<Framework>` is the name of the framework to link with, without the `.framework` extension. The two `-L` flags are required because the linker expects a space between the `-framework` flag and the name of the framework. DMD cannot handle this and will instead interpret the name of the framework as a separate flag.

Framework Paths

Using the above flag, the linker will search in the standard framework paths. The standard search paths for frameworks are:

- `/System/Library/Frameworks`
- `/Library/Frameworks`

The following flag from DMD can be used to add a new path in which to search for frameworks:

```
-L-F<framework_path>
```

For more information see the reference documentation and the `ld` man page.

34.9 Full Usage Example

This example will create an Objective-C string, `NSString`, and log the message using `NSLog` to stderr.

```
extern (Objective-C)
extern class NSString
{
    static NSString alloc() @selector("alloc");
    NSString initWithUTF8String(in char* str) @selector("initWithUTF8String:");
    void release() @selector("release");
}
```

This is a simplified declaration of the `NSString` class. The `alloc` method allocates an instance of the class. The `initWithUTF8String:` method will be used to convert a C string in UTF-8 to an Objective-C string, `NSString`. The `release` method is used to release an deallocate the string. Since D doesn't support ARC it's needed to manually release Objective-C instances.

```
extern (C) void NSLog(NSString, ...);
```

This `NSLog` function prints a message to the System Log facility, i.e. to stderr and Console.

```
auto str = NSString.alloc();
```

Allocate an instance of the class, `NSString`.

```
str = str.initWithUTF8String("Hello World!")
```

Initialize the Objective-C string using a C string.

```
NSLog(str);
```

Log the string to stderr, this will print something like this in the terminal:

```
2015-07-18 13:14:27.978 main[11045:2934950] Hello World!
```

```
str.release();
```

Release and deallocate the string.

All steps combined look like this:

```
module main;

extern (Objective-C)
extern class NSString
{
    static NSString alloc() @selector("alloc");
    NSString initWithUTF8String(in char* str) @selector("initWithUTF8String:");
```

```
    void release() @selector("release");
}

extern (C) void NSLog(NSString, ...);

void main()
{
    auto str = NSString.alloc().initWithUTF8String("Hello␣World!");
    NSLog(str);
    str.release();
}
```

When compiling the application remember to link with the required libraries, in this case the Foundation framework. Example:

```
dmd -L-framework -LFoundation main.d
```

Chapter 35

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.
  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 a `static assert` in the code to verify it:

  ```
  static assert(int.sizeof == (int*).sizeof);
  ```

35.1 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`, `.offsetof` and `.alignof` properties will be of type `size_t`.

35.2 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.

35.3 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*.

Chapter 36

Named Character Entities

NamedCharacterEntity:
   `&` *Identifier* `;`

The full list of named character entities from the HTML 5 Spec is supported except for the named entities which contain multiple code points. Below is a *partial* list of the named character entities.

Note: Not all glyphs will display properly in the **Glyph** column in all browsers.

Table 36.1: Named Character Entities

Name	Value	Glyph
`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	

↩

(continued)

Name	Value	Glyph
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	€

Table 36.2: Latin-1 (ISO-8859-1) Entities

Name	Value	Glyph
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	

↩

(continued)

Name	Value	Glyph
`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	Ð
`Ntilde`	209	Ñ

↩

(continued)

Name	Value	Glyph
`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

↩

(continued)

Name	Value	Glyph
`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

Table 36.3: Symbols and Greek letter entities

Name	Value	Glyph
`fnof`	402	ƒ
`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

↩

(continued)

Name	Value	Glyph
`Chi`	935	X
`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	o
`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	$\bullet$
`hellip`	8230	$\ldots$
`prime`	8242	$\prime$
`Prime`	8243	$\prime\prime$
`oline`	8254	

↩

(continued)

Name	Value	Glyph
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	∩

↩

(continued)

Name	Value	Glyph
`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	⌋
`loz`	9674	◊
`spades`	9824	♠
`clubs`	9827	♣
`hearts`	9829	♡
`diams`	9830	♢
`lang`	10216	⟨
`rang`	10217	⟩

Chapter 37

Memory Safety

Memory Safety for a program is defined as it being impossible for the program to corrupt memory. Therefore, the safe subset of D consists only of programming language features that are guaranteed to never result in memory corruption. See this article for a rationale.

Memory-safe code cannot use certain language features, such as:

- Casts that break the type system.
- Modification of pointer values.
- Taking the address of a local variable or function parameter.

37.1 Usage

There are three categories of functions from the perspective of memory safety:

- `@safe` functions
- `@trusted` functions
- `@system` functions

`@system` functions may perform any operation legal from the perspective of the language including inherently memory unsafe operations like returning pointers to expired stackframes. These functions may not be called directly from `@safe` functions.

`@trusted` functions have all the capabilities of `@system` functions but may be called from `@safe` functions. For this reason they should be very limited in the scope of their use. Typical uses of `@trusted` functions include wrapping system calls that take buffer pointer and length arguments separately so that @safe‘ functions may call them with arrays.

`@safe` functions have a number of restrictions on what they may do and are intended to disallow operations that may cause memory corruption. See `@safe` functions.

These attributes may be inferred when the compiler has the function body available, such as with templates.

Array bounds checks are necessary to enforce memory safety, so these are enabled (by default) for `@safe` code even in **-release** mode.

Scope and Return Parameters

The function parameter attributes `return` and `scope` are used to track what happens to low-level pointers passed to functions. Such pointers include: raw pointers, arrays, `this`, classes, `ref` parameters, delegate/lazy parameters, and aggregates containing a pointer.

`scope` ensures that no references to the pointed-to object are retained, in global variables or pointers passed to the function (and recursively to other functions called in the function), as a result of calling the function. Variables in the function body and parameter list that are `scope` may have their allocations elided as a result.

`return` indicates that either the return value of the function or the first parameter is a pointer derived from the `return` parameter or any other parameters also marked `return`. For constructors, `return` applies to the (implicitly returned) `this` reference. For void functions, `return` applies to the first parameter *iff* it is `ref`; this is to support UFCS, property setters and non-member functions (e.g. `put` used like `put(dest, source)`).

These attributes may appear after the formal parameter list, in which case they apply either to a method's `this` parameter, or to a free function's first parameter *iff* it is `ref`. `return` or `scope` is ignored when applied to a type that is not a low-level pointer.

Note: Checks for `scope` parameters are currently enabled only for `@safe` code compiled with the `-dip1000` command-line flag.

37.2 Limitations

Memory safety does not imply that code is portable, uses only sound programming practices, is free of byte order dependencies, or other bugs. It is focussed only on eliminating memory corruption possibilities.

Chapter 38

Application Binary Interface

A D implementation that conforms to the D ABI (Application Binary Interface) will be able to generate libraries, DLLs, etc., that can interoperate with D binaries built by other implementations.

38.1 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.

38.2 Endianness

The endianness (byte order) of the layout of the data will conform to the endianness of the target machine. The Intel x86 CPUs are *little endian* meaning that the value 0x0A0B0C0D is stored in memory as: `0D 0C 0B 0A`.

38.3 Basic Types

bool	8 bit byte with the values 0 for false and 1 for true
byte	8 bit signed value
ubyte	8 bit unsigned value
short	16 bit signed value
ushort	16 bit unsigned value
int	32 bit signed value
uint	32 bit unsigned value
long	64 bit signed value
ulong	64 bit unsigned value
cent	128 bit signed value
ucent	128 bit unsigned value
float	32 bit IEEE 754 floating point value
double	64 bit IEEE 754 floating point value
real	implementation defined floating point value, for x86 it is 80 bit IEEE 754 extended real

38.4 Delegates

Delegates are *fat pointers* with two parts:

Delegate Layout

offset	property	contents
`0`	`.ptr`	context pointer
ptrsize	`.funcptr`	pointer to function

The *context pointer* can be a class *this* reference, a struct *this* pointer, a pointer to a closure (nested functions) or a pointer to an enclosing function's stack frame (nested functions).

38.5 Structs

Conforms to the target's C ABI struct layout.

38.6 Classes

An object consists of:

Class Object Layout

size	property	contents
ptrsize	`.__vptr`	pointer to vtable
ptrsize	`.__monitor`	monitor
ptrsize...		vptrs for any interfaces implemented by this class in left to right, most to least derived, order
...	...	super's non-static fields and super's interface vptrs, from least to most derived
...	named fields	non-static fields

The vtable consists of:

Virtual Function Pointer Table Layout

size	contents
ptrsize	pointer to instance of TypeInfo
ptrsize...	pointers to virtual member functions

Casting a class object to an interface consists of adding the offset of the interface's corresponding vptr to the address of the base of the object. Casting an interface ptr back to the class type it came from involves getting the correct offset to subtract from it from the object.Interface entry at vtbl[0]. Adjustor thunks are created and pointers to them stored in the method entries in the vtbl[] in order to set the this pointer to the start of the object instance corresponding to the implementing method.

An adjustor thunk looks like:

```
ADD EAX,offset
JMP method
```

The leftmost side of the inheritance graph of the interfaces all share their vptrs, this is the single inheritance model. Every time the inheritance graph forks (for multiple inheritance) a new vptr is created and stored in the class' instance. Every time a virtual method is overridden, a new vtbl[] must be created with the updated method pointers in it.

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.

38.7 Interfaces

An interface is a pointer to a pointer to a vtbl[]. The vtbl[0] entry is a pointer to the corresponding instance of the object.Interface class. The rest of the `vtbl[1..$]` entries are pointers to the virtual functions implemented by that interface, in the order that they were declared.

A COM interface differs from a regular interface in that there is no object.Interface entry in `vtbl[0]`; the entries `vtbl[0..$]` are all the virtual function pointers, in the order that they were declared. This matches the COM object layout used by Windows.

A C++ interface differs from a regular interface in that it matches the layout of a C++ class using single inheritance on the target machine.

38.8 Arrays

A dynamic array consists of:

Dynamic Array Layout

offset	property	contents
`0`	`.length`	array dimension
`size_t`	`.ptr`	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 arrays and Dynamic arrays can be easily converted back and forth to each other.

38.9 Associative Arrays

Associative arrays consist of a pointer to an opaque, implementation defined type.

The current implementation is contained in and defined by rt/aaA.d.

38.10 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.

38.11 Name Mangling

D accomplishes typesafe linking by *mangling* a D identifier to include scope and type information.

```
MangledName:
    _D QualifiedName Type
    _D QualifiedName Z        // Internal
```

The *Type* above is the type of a variable or the return type of a function. This is never a *TypeFunction*, as the latter can only be bound to a value via a pointer to a function or a delegate.

```
QualifiedName:
    SymbolFunctionName
    SymbolFunctionName QualifiedName
```

```
SymbolFunctionName:
    SymbolName
    SymbolName TypeFunctionNoReturn
    SymbolName M TypeModifiers_opt TypeFunctionNoReturn
```

The **M** means that the symbol is a function that requires a `this` pointer. Class or struct fields are mangled without **M**. To disambiguate **M** from being a *Parameter* with modifier `scope`, the following type needs to be checked for being a *TypeFunction*.

```
SymbolName:
    LName
    TemplateInstanceName
    IdentifierBackRef
    0                         // anonymous symbols
```

Template Instance Names have the types and values of its parameters encoded into it:

```
TemplateInstanceName:
    TemplateID LName TemplateArgs Z
```

```
TemplateID:
    __T
    __U        // for symbols declared inside template constraint
```

```
TemplateArgs:
    TemplateArg
    TemplateArg TemplateArgs
```

```
TemplateArg:
    TemplateArgX
    H TemplateArgX
```

If a template argument matches a specialized template parameter, the argument is mangled with prefix **H**.

```
TemplateArgX:
    T Type
    V Type Value
    S QualifiedName
    X Number ExternallyMangledName
```

ExternallyMangledName can be any series of characters allowed on the current platform, e.g. generated by functions with C++ linkage or annotated with `pragma(mangle,...)`.

```
Value:
    n
    i Number
    N Number
    e HexFloat
    c HexFloat c HexFloat
    CharWidth Number _ HexDigits
    A Number Value...
```

S *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

CharWidth:
a
w
d

n is for **null** arguments.

i *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.

CharWidth Number _ HexDigits *CharWidth* 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 or asssociative array literal. *Number* is the length of the array. *Value* is repeated *Number* times for a normal array, and 2 * *Number* times for an associative array.

S *Number Value...* A struct 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*.

Back references

Any *LName* or non-basic *Type* (i.e. any type that does not encode as a fixed one or two character sequence) that has been emitted to the mangled symbol before will not be emitted again, but is referenced by a special sequence encoding the relative position of the original occurrence in the mangled symbol name.

Numbers in back references are encoded with base 26 by upper case letters **A** - **Z** for higher digits but lower case letters **a** - **z** for the last digit.

```
TypeBackRef:
    Q NumberBackRef
```

```
IdentifierBackRef:
    Q NumberBackRef
```

```
NumberBackRef:
    lower-case-letter
    upper-case-letter NumberBackRef
```

To distinguish between the type of the back reference a look-up of the back referenced character is necessary: An identifier back reference always points to a digit **0** to **9**, while a type back reference always points to a letter.

Type Mangling

Types are mangled using a simple linear scheme:

Type:
  TypeModifiers$_{\text{opt}}$ *TypeX*
  TypeBackRef

TypeX:
  TypeArray
  TypeStaticArray
  TypeAssocArray
  TypePointer
  TypeFunction
  TypeIdent
  TypeClass
  TypeStruct
  TypeEnum
  TypeTypedef
  TypeDelegate
  TypeVoid
  TypeByte
  TypeUbyte
  TypeShort
  TypeUshort
  TypeInt
  TypeUint
  TypeLong
  TypeUlong
  TypeCent
  TypeUcent
  TypeFloat
  TypeDouble
  TypeReal
  TypeIfloat

TypeIdouble
TypeIreal
TypeCfloat
TypeCdouble
TypeCreal
TypeBool
TypeChar
TypeWchar
TypeDchar
TypeNull
TypeTuple
TypeVector

TypeModifiers:
Const
Wild
Wild Const
Shared
Shared Const
Shared Wild
Shared Wild Const
Immutable

Shared:
`O`

Const:
`x`

Immutable:
`y`

Wild:
`Ng`

TypeArray:
   A *Type*

TypeStaticArray:
   G *Number* *Type*

TypeAssocArray:
   H *Type* *Type*

TypePointer:
   P *Type*

TypeVector:
   Nh *Type*

TypeFunction:
   TypeFunctionNoReturn *Type*

TypeFunctionNoReturn:
   CallConvention *FuncAttrs*$_{\text{opt}}$ *Parameters*$_{\text{opt}}$ *ParamClose*

CallConvention:

```
F       // D
U       // C
W       // Windows
R       // C++
Y       // Objective-C
```

FuncAttrs:
   FuncAttr
   FuncAttr *FuncAttrs*

FuncAttr:
  FuncAttrPure
  FuncAttrNothrow
  FuncAttrRef
  FuncAttrProperty
  FuncAttrNogc
  FuncAttrReturn
  FuncAttrScope
  FuncAttrTrusted
  FuncAttrSafe
  FuncAttrLive

Function attributes are emitted in the order as listed above.

FuncAttrPure:
  `Na`

FuncAttrNogc:
  `Ni`

FuncAttrNothrow:
  `Nb`

FuncAttrProperty:
  `Nd`

FuncAttrRef:
  `Nc`

FuncAttrReturn:
  `Nj`

```
FuncAttrScope:
    Nl

FuncAttrTrusted:
    Ne

FuncAttrSafe:
    Nf

FuncAttrLive:
    Nm

Parameters:
    Parameter
    Parameter Parameters

Parameter:
    Parameter2
    M Parameter2      // scope
    Nk Parameter2     // return

Parameter2:
    Type
    J Type      // out
    K Type      // ref
    L Type      // lazy

ParamClose
    X     // variadic T t...) style
    Y     // variadic T t,...) style
    Z     // not variadic
```

TypeIdent:
`I` *QualifiedName*

TypeClass:
`C` *QualifiedName*

TypeStruct:
`S` *QualifiedName*

TypeEnum:
`E` *QualifiedName*

TypeTypedef:
`T` *QualifiedName*

TypeDelegate:
`D` *TypeModifiers*$_{\text{opt}}$ *TypeFunction*

TypeVoid:
`v`

TypeByte:
`g`

TypeUbyte:
`h`

TypeShort:
`s`

TypeUshort:
`t`

TypeInt:
`i`

TypeUint:
`k`

TypeLong:
`l`

TypeUlong:
`m`

TypeCent:
`zi`

TypeUcent:
`zk`

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`

TypeNull:
`n`

TypeTuple:
`B` *Parameters* `Z`

38.12 Function Calling Conventions

The `extern (C)` and `extern (D)` calling convention matches the C calling convention used by the supported C compiler on the host system. Except that the extern (D) calling convention for Windows 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.
- References are returned as pointers in EAX.
- Delegates are returned with the pointer to the function in EDX and the context pointer in EAX.

- 1, 2 and 4 byte structs and static arrays are returned in EAX.
- 8 byte structs and static arrays are returned in EDX,EAX, where EDX gets the most significant half.
- For other sized structs and static arrays, 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.
- It is not a floating point type.

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 **ref** 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.

38.13 Exception Handling

Windows

Conforms to the Microsoft Windows Structured Exception Handling conventions.

Linux, FreeBSD and OS X

Uses static address range/handler tables. It is not compatible with the ELF/Mach-O exception handling tables. The stack is walked assuming it uses the EBP/RBP stack frame convention. The EBP/RBP convention must be used for every function that has an associated EH (Exception Handler) table.

For each function that has exception handlers, an EH table entry is generated.

EH Table Entry

field	description
`void*`	pointer to start of function
`DHandlerTable*`	pointer to corresponding EH data
`uint`	size in bytes of the function

The EH table entries are placed into the following special segments, which are concatenated by the linker.

EH Table Segment

Operating System	Segment Name
Windows	`FI`
Linux	`.deh_eh`
FreeBSD	`.deh_eh`
OS X	`__deh_eh`, `__DATA`

The rest of the EH data can be placed anywhere, it is immutable.

DHandlerTable

field	description
`void*`	pointer to start of function
`uint`	offset of ESP/RSP from EBP/RBP
`uint`	offset from start of function to return code
`uint`	number of entries in `DHandlerInfo[]`
`DHandlerInfo[]`	array of handler information

DHandlerInfo

field	description
`uint`	offset from function address to start of guarded section
`uint`	offset of end of guarded section
`int`	previous table index
`uint`	if != 0 offset to DCatchInfo data from start of table
`void*`	if not null, pointer to finally code to execute

DCatchInfo

field	description
`uint`	number of entries in `DCatchBlock[]`
`DCatchBlock[]`	array of catch information

DCatchBlock

field	description
`ClassInfo`	catch type
`uint`	offset from EBP/RBP to catch variable
`void*`, catch handler code	

38.14 Garbage Collection

The interface to this is found in Druntime's gc/gcinterface.d.

38.15 Runtime Helper Functions

These are found in Druntime's rt/.

38.16 Module Initialization and Termination

All the static constructors for a module are aggregated into a single function, and a pointer to that function is inserted into the ctor member of the ModuleInfo instance for that module.

All the static denstructors for a module are aggregated into a single function, and a pointer to that function is inserted into the dtor member of the ModuleInfo instance for that module.

38.17 Unit Testing

All the unit tests for a module are aggregated into a single function, and a pointer to that function is inserted into the unitTest member of the ModuleInfo instance for that module.

38.18 Symbolic Debugging

D has types that are not represented in existing C or C++ debuggers. These are dynamic arrays, associative arrays, and delegates. Representing these types as structs causes problems because function calling conventions for structs are often different than that for these types, which causes C/C++ debuggers to misrepresent things. For these debuggers, they are represented as a C type which does match the calling conventions for the type. The **dmd** compiler will generate only C symbolic type info with the **-gc** compiler switch.

Types for C Debuggers

D type	C representation
dynamic array	`unsigned long long`
associative array	`void*`
delegate	`long long`
`dchar`	`unsigned long`

For debuggers that can be modified to accept new types, the following extensions help them fully support the types.

Codeview Debugger Extensions

The D **dchar** type is represented by the special primitive type 0x78.

D makes use of the Codeview OEM generic type record indicated by `LF_OEM` (0x0015). The format is:

Codeview OEM Extensions for D

field size	2	2	2	2	2	2
D Type	Leaf Index	OEM Identifier	recOEM	num indices	type index	type index
dynamic array	`LF_OEM`	*OEM*	1	2	@*index*	@*element*
associative array	`LF_OEM`	*OEM*	2	2	@*key*	@*element*
delegate	`LF_OEM`	*OEM*	3	2	@*this*	@*function*

where:

OEM	0x42
index	type index of array index
key	type index of key
element	type index of array element
this	type index of context pointer
function	type index of function

These extensions can be pretty-printed by obj2asm. The Ddbg debugger supports them.

Chapter 39

Vector Extensions

CPUs often support specialized vector types and vector operations (a.k.a. *media instructions*). Vector types are a fixed array of floating or integer types, and vector operations operate simultaneously on them.

Specialized *Vector* types provide access to them.

The *VectorBaseType* must be a Static Array. The *VectorElementType* is the unqualified element type of the static array. The dimension of the static array is the number of elements in the vector.

Implementation Defined: Which vector types are supported depends on the target. The implementation is expected to only support the vector types that are implemented in the target's hardware.

Best Practices: Use the declarations in `core.simd` instead of the language *Vector* grammar.

39.1 `core.simd`

Vector types and operations are introduced by importing `core.simd`:

```
import core.simd;
```

Implementation Defined:

These types and operations will be the ones defined for the architecture the compiler is targeting. If a particular CPU family has varying support for vector types, an additional runtime check may be necessary. The compiler does not emit runtime checks; those must be done by the programmer.

Depending on the architecture, compiler flags may be required to activate support for SIMD types.

The types defined will all follow the naming convention:

`typeNN`

where *type* is the vector element type and *NN* is the number of those elements in the vector type. The type names will not be keywords.

Properties

Vector types have the property:

Vector Type Properties

Property	Description
.array	Returns static array representation

All the properties of the *VectorBaseType* work.

Conversions

Vector types of the same size can be implicitly converted among each other. Vector types can be cast to their *VectorBaseType*.

Integers and floating point values can be implicitly converted to their vector equivalents:

```
int4 v = 7;
v = 3 + v;   // add 3 to each element in v
```

Accessing Individual Vector Elements

They cannot be accessed directly, but can be when converted to an array type:

```
int4 v;
(cast(int*)&v)[3] = 2;   // set 3rd element of the 4 int vector
(cast(int[4])v)[3] = 2;  // set 3rd element of the 4 int vector
v.array[3] = 2;          // set 3rd element of the 4 int vector
v.ptr[3] = 2;            // set 3rd element of the 4 int vector
```

Conditional Compilation

If vector extensions are implemented, the version identifier `D_SIMD` is set.

Whether a type exists or not can be tested at compile time with an *IsExpression*:

```
static if (is(typeNN))
    ... yes, it is supported ...
else
    ... nope, use workaround ...
```

Whether a particular operation on a type is supported can be tested at compile time with:

```
float4 a,b;
static if (__traits(compiles, a+b))
    ... yes, it is supported ...
else
    ... nope, use workaround ...
```

For runtime testing to see if certain vector instructions are available, see the functions in core.cpuid.

A typical workaround would be to use array vector operations instead:

```
float4 a,b;
static if (__traits(compiles, a/b))
    c = a / b;
else
    c[] = a[] / b[];
```

39.2 X86 And X86_64 Vector Extension Implementation

Implementation Defined:

The following describes the specific implementation of the vector types for the X86 and X86_64 architectures.

The vector extensions are currently implemented for the OS X 32 bit target, and all 64 bit targets.

`core.simd` defines the following types:

Vector Types

Type Name	Description	gcc Equivalent
void16	16 bytes of untyped data	*no equivalent*
byte16	16 `bytes`	`signed char __attribute__((vector_size(16)))`
ubyte16	16 `ubytes`	`unsigned char __attribute__((vector_size(16)))`
short8	8 `shorts`	`short __attribute__((vector_size(16)))`
ushort8	8 `ushorts`	`ushort __attribute__((vector_size(16)))`
int4	4 `ints`	`int __attribute__((vector_size(16)))`
uint4	4 `uints`	`unsigned __attribute__((vector_size(16)))`
long2	2 `longs`	`long __attribute__((vector_size(16)))`
ulong2	2 `ulongs`	`unsigned long __attribute__((vector_size(16)))`
float4	4 `floats`	`float __attribute__((vector_size(16)))`
double2	2 `doubles`	`double __attribute__((vector_size(16)))`
void32	32 bytes of untyped data	*no equivalent*
byte32	32 `bytes`	`signed char __attribute__((vector_size(32)))`
ubyte32	32 `ubytes`	`unsigned char __attribute__((vector_size(32)))`
short16	16 `shorts`	`short __attribute__((vector_size(32)))`
ushort16	16 `ushorts`	`ushort __attribute__((vector_size(32)))`
int8	8 `ints`	`int __attribute__((vector_size(32)))`
uint8	8 `uints`	`unsigned __attribute__((vector_size(32)))`
long4	4 `longs`	`long __attribute__((vector_size(32)))`
ulong4	4 `ulongs`	`unsigned long __attribute__((vector_size(32)))`
float8	8 `floats`	`float __attribute__((vector_size(32)))`
double4	4 `doubles`	`double __attribute__((vector_size(32)))`

Note: for 32 bit gcc, it's `long long` instead of `long`.

Supported 128-bit Vector Operators

Operator	void16	byte16	ubyte16	short8	ushort8	int4	uint4	long2	ulong2	float4	double2
=	×	×	×	×	×	×	×	×	×	×	×
+	–	×	×	×	×	×	×	×	×	×	×
-	–	×	×	×	×	×	×	×	×	×	×
*	–	–	–	×	×	–	–	–	–	×	×
/	–	–	–	–	–	–	–	–	–	×	×
&	–	×	×	×	×	×	×	×	×	–	–
|	–	×	×	×	×	×	×	×	×	–	–
^	–	×	×	×	×	×	×	×	×	–	–
+=	–	×	×	×	×	×	×	×	×	×	×
-=	–	×	×	×	×	×	×	×	×	×	×
*=	–	–	–	×	×	–	–	–	–	×	×
/=	–	–	–	–	–	–	–	–	–	×	×
&=	–	×	×	×	×	×	×	×	×	–	–
|=	–	×	×	×	×	×	×	×	×	–	–
^=	–	×	×	×	×	×	×	×	×	–	–
unary~	–	×	×	×	×	×	×	×	×	–	–
unary+	–	×	×	×	×	×	×	×	×	×	×
unary-	–	×	×	×	×	×	×	×	×	×	×

Supported 256-bit Vector Operators

Operator	void32	byte32	ubyte32	short16	ushort16	int8	uint8	long4	ulong4	float8	double4
=	×	×	×	×	×	×	×	×	×	×	×
+	–	×	×	×	×	×	×	×	×	×	×
-	–	×	×	×	×	×	×	×	×	×	×
*	–	–	–	–	–	–	–	–	–	×	×
/	–	–	–	–	–	–	–	–	–	×	×
&	–	×	×	×	×	×	×	×	×	–	–
|	–	×	×	×	×	×	×	×	×	–	–
^	–	×	×	×	×	×	×	×	×	–	–
+=	–	×	×	×	×	×	×	×	×	×	×
-=	–	×	×	×	×	×	×	×	×	×	×
*=	–	–	–	–	–	–	–	–	–	×	×
/=	–	–	–	–	–	–	–	–	–	×	×
&=	–	×	×	×	×	×	×	×	×	–	–
|=	–	×	×	×	×	×	×	×	×	–	–
^=	–	×	×	×	×	×	×	×	×	–	–
unary~	–	×	×	×	×	×	×	×	×	–	–
unary+	–	×	×	×	×	×	×	×	×	×	×
unary-	–	×	×	×	×	×	×	×	×	×	×

Operators not listed are not supported at all.

Vector Operation Intrinsics

See `core.simd` for the supported intrinsics.

Chapter 40

Better C

It is straightforward to link C functions and libraries into D programs. But linking D functions and libraries into C programs is not straightforward.

D programs generally require:

1. The D runtime library to be linked in, because many features of the core language require runtime library support.
2. The `main()` function to be written in D, to ensure that the required runtime library support is properly initialized.

To link D functions and libraries into C programs, it's necessary to only require the C runtime library to be linked in. This is accomplished by defining a subset of D that fits this requirement, called **BetterC**.

Implementation Defined: **BetterC** is typically enabled by setting the `-betterC` command line flag for the implementation.

When **BetterC** is enabled, the predefined version `D_BetterC` can be used for conditional compilation.

An entire program can be written in **BetterC** by supplying a C `main()` function:

```
extern(C) void main()
{
    import core.stdc.stdio : printf;
    printf("Hello␣betterC\n");
}
```

```
> dmd -betterC hello.d && ./hello
Hello betterC
```

Limiting a program to this subset of runtime features is useful when targeting constrained environments where the use of such features is not practical or possible.

BetterC makes embedding D libraries in existing larger projects easier by:

1. Simplifying the process of integration at the build-system level
2. Removing the need to ensure that Druntime is properly initialized on calls to the library, for situations when an initialization step is not performed or would be difficult to insert before the library is used.
3. Mixing memory management strategies (GC + manual memory management) can be tricky, hence removing D's GC from the equation may be worthwhile sometimes.

40.1 Retained Features

Nearly the full language remains available. Highlights include:

1. Unrestricted use of compile-time features
2. Full metaprogramming facilities
3. Nested functions, nested structs, delegates and lambdas
4. Member functions, constructors, destructors, operating overloading, etc.
5. The full module system
6. Array slicing, and array bounds checking
7. RAII (yes, it can work without exceptions)
8. `scope(exit)`
9. Memory safety protections
10. Interfacing with C++
11. COM classes and C++ classes
12. `assert` failures are directed to the C runtime library
13. `switch` with strings
14. `final switch`
15. `unittest`
16. `printf` format validation

Running unittests in `-betterC`

While, testing can be done without the `-betterC` flag, it is sometimes desirable to run the testsuite in `-betterC` too. `unittest` blocks can be listed with the `getUnitTests` trait:

```
unittest
{
```

```
    assert(0);
}

extern(C) void main()
{
    static foreach(u; __traits(getUnitTests, __traits(parent, main)))
        u();
}
```

```
> dmd -betterC -unittest -run test.d
dmd_runpezoXK: foo.d:3: Assertion '0' failed.
```

However, in `-betterC` `assert` expressions don't use Druntime's assert and are directed to `assert` of the C runtime library instead.

40.2 Unavailable Features

D features not available with **BetterC**:

1. Garbage Collection
2. TypeInfo and ModuleInfo
3. Classes
4. Built-in threading (e.g. `core.thread`)
5. Dynamic arrays (though slices of static arrays work) and associative arrays
6. Exceptions
7. `synchronized` and `core.sync`
8. Static module constructors or destructors