# D Programming Language Specification

## 0.1 Table of Contents

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

- Unit Tests
- Garbage Collection
- Floating Point
- Inline Assembler
- Embedded Documentation
- Interfacing To C
- Interfacing To C++
- Portability Guide
- Named Character Entities
- Memory Safety
- Application Binary Interface

## 0.2 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 what those tokens are. The grammar is designed to be suitable for high speed scanning, it has a minimum of special case rules, there is only one phase of translation, and to make it easy to write a correct scanner for. The tokens are readily recognizable by those familiar with C and C++.

### 0.2.1 Source Text

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

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

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

| Format | BOM |
|---|---|
| UTF-8 | EF BB BF |
| UTF-16BE | FE FF |
| UTF-16LE | FF FE |
| UTF-32BE | 00 00 FE FF |
| UTF-32LE | FF FE 00 00 |
| ASCII | no BOM |

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

There are no digraphs or trigraphs in D.

The source text is decoded from its source representation into Unicode *Character*s. The *Character*s are further divided into: *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 tries to make the longest token it can. For example `>>` is a right shift token, not two greater than tokens. An exception to this rule is that 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.

### 0.2.2 Character Set

```
Character:
    any Unicode character
```

### 0.2.3 End of File

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

The source text is terminated by whichever comes first.

### 0.2.4 End of Line

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

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

### 0.2.5 White Space

```
WhiteSpace:
    Space
    Space WhiteSpace

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

### 0.2.6 Comments

```
Comment:
    BlockComment
    LineComment
    NestingBlockComment

BlockComment:
    /* Characters */

LineComment:
    // Characters EndOfLine

NestingBlockComment:
    /+ NestingBlockCommentCharacters +/

NestingBlockCommentCharacters:
    NestingBlockCommentCharacter
    NestingBlockCommentCharacter NestingBlockCommentCharacters

NestingBlockCommentCharacter:
    Character
    NestingBlockComment

Characters:
    Character
    Character Characters
```

D has three kinds of comments:

1. Block comments can span multiple lines, but do not nest.
2. Line comments terminate at the end of the line.
3. Nesting 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.

### 0.2.7 Tokens

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

### 0.2.8 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. (This is the C99 Standard.) Identifiers can be arbitrarily long, and are case sensitive. Identifiers starting with __ (two underscores) are reserved.

### 0.2.9 String Literals

```
StringLiteral:
   WysiwygString
   AlternateWysiwygString
   DoubleQuotedString

   HexString
   DelimitedString
   TokenString

WysiwygString:
   r" WysiwygCharacters " StringPostfix<sub>opt</sub>

AlternateWysiwygString:
   ‘ WysiwygCharacters ‘ StringPostfix<sub>opt</sub>

WysiwygCharacters:
   WysiwygCharacter
   WysiwygCharacter WysiwygCharacters

WysiwygCharacter:
   Character
   EndOfLine

DoubleQuotedString:
   " DoubleQuotedCharacters " StringPostfix<sub>opt</sub>

DoubleQuotedCharacters:
   DoubleQuotedCharacter
   DoubleQuotedCharacter DoubleQuotedCharacters

DoubleQuotedCharacter:
   Character
   EscapeSequence
   EndOfLine

EscapeSequence:
   \'
   \"
   \?
   \\
   \a
   \b
   \f
   \n
   \r
   \t
   \v
   \ EndOfFile
   \x HexDigit HexDigit
   \ OctalDigit
   \ OctalDigit OctalDigit
   \ OctalDigit OctalDigit OctalDigit
   \u HexDigit HexDigit HexDigit HexDigit
   \U HexDigit HexDigit HexDigit HexDigit HexDigit HexDigit HexDigit HexDigit
   \ NamedCharacterEntity

HexString:
   x" HexStringChars " StringPostfix<sub>opt</sub>

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, an escape sequence, a delimited string, a token string, or a hex string.

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

## 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 r" ":

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

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

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

## Double Quoted Strings

Double quoted strings are enclosed by "". Escape sequences can be embedded into them with the typical \ notation.

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

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

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

The following are all equivalent:

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

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

| Postfix | Type | 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.

String literals are read only. Writes to string literals cannot always be detected, but cause undefined behavior.

### 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:

| 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:

```
writefln(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{foo}              // "foo"
q{/*}*/ }           // "/*}*/ "
q{ foo(q{hello}); } // " foo(q{hello}); "
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
```

## 0.2.10 Character Literals

```
CharacterLiteral:
    ' SingleQuotedCharacter '

SingleQuotedCharacter:
    Character
    EscapeSequence
```

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

### 0.2.11 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 BinaryDigits

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
   _

BinaryDigitsUS:
   BinaryDigitUS
   BinaryDigitUS BinaryDigitsUS

BinaryDigit:
   0
   1

BinaryDigitUS:
   BinaryDigit
   _

OctalDigits:
   OctalDigit
   OctalDigit OctalDigits
```

```
OctalDigitsUS:
   OctalDigitUS
   OctalDigitUS OctalDigitsUS

OctalDigit:
   0
   1
   2
   3
   4
   5
   6
   7

OctalDigitUS:
   OctalDigit
   _

HexDigits:
   HexDigit
   HexDigit HexDigits

HexDigitsUS:
   HexDigitUS
   HexDigitUS HexDigitsUS

HexDigitsNoSingleUS:
   HexDigit
   HexDigit HexDigitsUS
   HexDigitsUS HexDigit

HexDigit:
   DecimalDigit
   HexLetter

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

```
D
E
F
_
```

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

Decimal integers are a sequence of decimal digits.

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

C-style octal integer notation was deemed too easy to mix up with decimal notation. The above 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’.

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

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

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

The type of the integer is resolved as follows:

| Literal | Type |
|---|---|
| *Usual decimal notation* | |
| `0 .. 2_147_483_647` | `int` |
| `2_147_483_648 .. 9_223_372_036_854_775_807` | `long` |
| *Explicit suffixes* | |
| `0L .. 9_223_372_036_854_775_807L` | `long` |
| `0U .. 4_294_967_296U` | `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` |

### 0.2.12 Floating Literals

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

Float:
   DecimalFloat
   HexFloat

DecimalFloat:
   LeadingDecimal .
   LeadingDecimal . DecimalDigits
   DecimalDigits . DecimalDigitsNoSingleUS DecimalExponent
   . DecimalInteger
   . DecimalInteger DecimalExponent
   LeadingDecimal DecimalExponent

DecimalExponent
   DecimalExponentStart DecimalDigitsNoSingleUS

DecimalExponentStart
   e
   E
   e+
   E+
   e-
   E-

HexFloat:
   HexPrefix HexDigitsNoSingleUS . HexDigitsNoSingleUS HexExponent
   HexPrefix . HexDigitsNoSingleUS 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 with a **0x** and the exponent is a **p** or **P** followed by a decimal number serving as the exponent of 2.

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

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

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

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

Examples:

```
0x1.FFFFFFFFFFFFFp1023 // double.max
0x1p-52                // double.epsilon
1.175494351e-38F       // float.min
```

```
6.3i                    // idouble 6.3
6.3fi                   // ifloat 6.3
6.3Li                   // ireal 6.3
```

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

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

```
4.5 + 6.2i  // complex number (phased out)
```

### 0.2.13 Keywords

Keywords are reserved identifiers.

```
Keyword:
    abstract
    alias
    align
    asm
    assert
    auto

    body
    bool
    break
    byte

    case
    cast
    catch
    cdouble
    cent
    cfloat
    char
    class
    const
    continue
    creal

    dchar
    debug
    default
    delegate
    delete
    deprecated
    do
```

```
double

else
enum
export
extern

false
final
finally
float
for
foreach
foreach_reverse
function

goto

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

lazy
long

macro
mixin
module

new
nothrow
null

out
override

package
```

```
pragma
private
protected
public
pure
real
ref
return

scope
shared
short
static
struct
super
switch
synchronized

template
this
throw
true
try
typedef
typeid
typeof

ubyte
ucent
uint
ulong
union
unittest
ushort

version
void
volatile

wchar
while
with

__FILE__
__LINE__
__gshared
```

```
    __traits
    __vector
    __parameters
```

### 0.2.14 Special Tokens

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

| 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, such as "Digital Mars D" |
| `__VERSION__` | Compiler version as an integer, such as 2001 |

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

For example:

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

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