

The D Programming Language

An Introduction

```
import std.stdio;
void main()
{
    writeln("Hello world!");
}
```

```
import std.stdio;
void main()
{
    foreach(c; "Hello world!\n") {
        write(c);
    }
}
```

```
import std.stdio;
void main()
{
    mixin("writeln(\"Hello world!\");");
}
```

Ali Çehreli
acehrel@yahoo.com

September 2009

Resources

- `digitalmars.com`
articles, specs, newsgroups, everything
- Andrei Alexandrescu's article **“The Case for D”** published on ACCU's CVu, and Dr.Dobb's Journal
- `dsources.org` for D projects and hosting
- `www.prowiki.org/wiki4d/wiki.cgi`

A Layman's Bullets

- High-level system language
- Evolved from C++
- Features from Java, C#, Eiffel, and others

- Garbage collection (RAII too)
- Safe programming
- Functional programming
- Safe memory model

Paradigms

- Imperative
- Object oriented
- Metaprogramming
- Functional

History

- D1
 - Created by Walter Bright in 2001
 - Stable
- D2
 - At alpha stage since June 2007
 - Andrei Alexandrescu, Bartosz Milewski and others
 - Vibrant

Philosophy

- No committee
- Open source
- Newsgroup discussions
- Practicality
- High performance
- Separate lexing, parsing, and analysis

Tools

Compilers

- dmd: Digital Mars
- gdc: GNU
- LDC: dmd front-end, LLVM back-end
- D for .NET

Debugging

- gdb (limited support)
- WinDbg and Ddbg for Windows
- ZeroBUGS (experimental)

IDE

- Various, but not widespread

Unicode Source Code

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

```
void main()  
{  
    bool ASCII_geçmişte_kaldı = true;  
    // "ASCII is in the past"  
}
```

Crippled code in some other languages:

dön: return

don: underwear

öldü: died

oldu: happened

etc.

(obscene cases do exist)

Fundamental Types - 1

void
bool false/true

Character types:

char 8
wchar 16
dchar 32

Integer types:

byte 8
short 16
int 32
long 64
cent 128 (reserved for future use)

Unsigned versions:

ubyte, ushort, uint, ulong, ucent

Floating point types:

float 32
double 64
real largest on the platform, at least 64 (80 on x86)

Fundamental Types - 2

Complex number types:
(pairs of floating point values)

float
double
real

Imaginary number types:

ifloat
idouble
ireal

“The language of pairs is incorrect for Complex Arithmetic; it needs the Imaginary type.” – Prof. Kahan

“How Java's Floating-Point Hurts Everyone Everywhere”

<http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf>

$(1 - \text{infinity} * i) * i$ should be $(\text{infinity} + i)$

if complex number are *pairs*:

$(1 - \text{infinity} * i) * (0 + i)$ yields $(\text{infinity} + \text{NaN} * i)$

Program Correctness

- SafeD – A safe subset of D that eliminates memory errors
- Unit tests
- **Contract programming** (invented by Eiffel's creator Bertrand Meyer; trademarked as “Design by Contract”)
 - Function pre-conditions
 - Function post-conditions
 - Class invariants

SafeD

No memory corruptions...

NO:

- Inline assembler
- Casting non-pointer to pointer
- Casting away const or immutable
- Casting away shared
- Casting T1* to T2*
 - T*-to-void* **ok**
 - int*-to-short*, etc. **ok**

Contract Programming and Unit Test Examples

```
long square_root(long x)
in
{
    assert(x >= 0);
}
out (result)
{
    assert((result * result) <= x && (result+1) * (result+1) >= x);
}
body
{
    return cast(long)std.math.sqrt(cast(real)x);
}
unittest
{
    assert(square_root(36) == 6);
}
```

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

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

immutable (a.k.a. invariant) and const

- immutable: can never change
- const: can not change through this reference
- Both are transitive:

```
const Record * p = new Record;  
p = new Record; // ERROR; similar to C  
p.id = 42;      // also ERROR
```

- Incompatible:

```
void foo(immutable int * value)  
{  
    // ...  
}  
// ...  
const int i = 42;  
foo(&i);    // ERROR
```

Arrays

First class types

```
// Static array  
int[5] a = [ 0, 1, 2, 3, 4 ];  
// a.length = 7; ERROR
```

```
// Dynamic array  
int[] b = [ 0, 1, 2, 3, 4 ];  
b.length = 7;
```

```
// Slice; a reference into b  
int[] c = b[2..5];
```

```
// Concatenation and appending  
int[] d = b ~ c;  
d ~= c;
```

```
// d:  0 1 2 3 4 0 0  2 3 4  2 3 4
```

```
// $: length  
d[0..$];
```

Associative arrays

```
double[string] dict;    // array of double, indexed by string

dict["monday"] = 1.1;
dict["tuesday"] = 2.2;

// [monday:1.1, tuesday:2.2]
writeln(dict);

// dict[monday]=1.1
// dict[tuesday]=2.2
foreach (string key, double value; dict) {
    writeln("dict[" + key + "]=", value);
}
```

Unicode strings

```
char[] sc = "aŞ".dup; // UTF-8 (same as "aŞ".dup)
wchar[] sw = "aŞ".dup; // UTF-16
dchar[] sd = "aŞ".dup; // UTF-32

writeln(sc.length); // 3
writeln(sw.length); // 2
writeln(sd.length); // 2
```

- String literals are immutable
- `string` is an alias of `immutable(char)[]`

```
char[] s0 = "hello"; // ERROR
char[] s1 = "hello".dup; // ok

function_taking_string(s1.idup); // .idup necessary
```

Built-in strings are fast

- 2 to 30 times faster compilation
- 2 to 3 times faster run time

<http://www.digitalmars.com/d/2.0/cppstrings.html>

Libraries

Two competing “standard” libraries; soon to be joined over a druntime:

- **Phobos** by Digital Mars (currently being improved mostly by Andrei Alexandrescu)
- **Tango** by others

C libraries can be linked directly as long as there are D bindings written; C++ libraries are partially supported

Many “D bindings” around:

- GUI: QtD, GtkD, wxD, D-FLTK, D/Tk, DWT, DFL, etc.
- IBM's ICU, etc.

Sample D Binding

```
// my_ncurses.d

module my_ncurses;

extern (C):

enum TRUE = 1;
enum FALSE = 0;

alias void WINDOW;

WINDOW * initscr();
int cbreak();
int noecho();
int nonl();

int intrflush(WINDOW * win, bool bf);

// Hack: extern NCURSES_EXPORT_VAR(WINDOW *) stdscr;
WINDOW * stdscr;

int keypad(WINDOW *win, bool bf);

int getch();
int printw(const char *fmt, ...);
int endwin();
```

From iterators to ranges

Andrei Alexandrescu's

- recent newsgroup discussions
- upcoming article “Iterators Must Go (Ahead)”

Ranges:

- Are safer to use
- Have simpler definitions and uses
- Offer interesting new opportunities
- Can be lazy (STL's `set_union` must be eager)

```
// C++'s STL: Puts the nth element at the nth position
//           The two ranges around nth are not sorted
void nth_element(It first, It nth, It last);
```

```
// D's Phobos:
void nth_element(R1 left, R2 right);

sort(Chain(r0, r1, r2));
```

struct vs class

Closer to the roots:

- **struct**

value semantics with limited OO support

yes: constructors, destructors, hidden members, overloading, etc.

no: inheritance

- **class**

reference semantics with “full” OO support

class

- Single inheritance, multiple interfaces
- Constructors can call other constructors
 - `super` is the parent constructor
 - `this(int)` can call `this()`
- Single operator that returns the common -1,0,1 results for all comparisons: `opCmp`
- Garbage collected, but can be scope-guarded
- Properties

```
scope f = new File();
```

More classes

- scope classes
- new and delete can be overridden
- alias this: for type conversions

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

    static this()
    {
        // ...
    }

    unittest
    {
        // ...
    }
}
```

Interfaces

```
interface D  
{  
    int foo();  
}
```

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

Modules

- ~~#include~~ `import`
 - No include guards
 - No forward declarations
- Uses file system directory structure
(`std.stdio` is the file `std/stdio.d`)

Function Templates

```
auto half(T)(T i)
{
    return i / 2;
}
// ...
half!(int)(10);
half!int(10);
half(10);
```

```
auto foo(T0, T1 = T0)(T0 x, T1 y = 1)
{
    // ...
}
```

// Copied from digitalmars.com

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

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

Class Templates

```
class Foo(T)
{
    T my_member;
}
```

```
Foo!(int) x;
Foo!int y;
```

static if, static assert, and is

```
auto half(T)(T i)
{
    static assert( !is(T : int));

    static if (is (T : double) || is (T : float)) {
        writeln("called for ", T.stringof);
    }

    return i / 2;
}
```

typedef	the type that Type is a typedef of
struct	Type
union	Type
class	Type
interface	Type
super	TypeTuple of base classes and interfaces
enum	the base type of the enum
function	TypeTuple of the function parameter types
delegate	the function type of the delegate
return	the return type of the function, delegate, or function pointer
const	Type
invariant	Type

Template mixins

```
template Foo(T)
{
    T x = 5;
}
```

```
mixin Foo!(int);           // create x of type int
```

string mixins

```
template GenStruct(string Name, string M1)
{
    const char[] GenStruct = "struct " ~ Name ~ "{ int " ~ M1 ~ "; }";
}

mixin(GenStruct!("Foo", "bar"));
```

Code generation

```
// from std/algorithm.d
void sort(alias less = "a < b", SwapStrategy ss = SwapStrategy.unstable, Range)
    (Range r)
{
    // ...
}

// ...
sort!("a > b")(array);
```

Function Parameters

```
int foo(in int x, out int y, ref int z, int q);
```

in: equivalent to **const** scope

out: default initialized

ref: referenced

none: mutable copy

Pure Functions

- Parameters are all immutable or are implicitly convertible to immutable
- Does not read or write any global mutable state

```
int x;  
immutable int y;  
const int* pz;  
  
pure int foo(int i,      // ok, implicitly convertible to immutable  
             char* p,   // error, not immutable  
             const char* q, // error, not immutable  
             invariant int* s // ok, immutable  
            )  
{  
    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;  
}
```

More Functions

- Nested functions
- Function literals

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

Delegates (closures)

```
int delegate() test()
{
    int b = 3;

    return delegate { b += 6; return b; };
}

void main()
{
    int delegate() dg = test();
    dg();
}
```

`function` would be a compiler error:

Lazy Evaluation - 1

// The problem:

```
void log(char[] message)
{
    if (logging)
        fprintfn(logfile, message);
}

void foo(int i)
{
    log("Entering foo() with i set to " ~ toString(i));
}
```

// A solution (no macros in D):

```
void foo(int i)
{
    if (logging) log("Entering foo() with i set to " ~ toString(i));
}
```

// Usually through a macro in C:

```
#define LOG(string) (logging && log(string))
```

Lazy Evaluation - 2

```
// The "delegate" solution
void log(char[] delegate() dg)
{
    if (logging)
        fprintln(logfile, dg());
}

void foo(int i)
{
    log( { return "Entering foo() with i set to " ~ toString(i); } );
}
```

Andrei Alexandrescu's simplification: Any expression can be implicitly converted to a delegate that returns either void or the type of the expression

Tomasz Stachowiak's suggestion: the `lazy` keyword

```
void log(lazy char[] dg)
{
    if (logging)
        fprintln(logfile, dg());
}

void foo(int i)
{
    log("Entering foo() with i set to " ~ toString(i));
}
```

Multithreading

- Theoretical support for safe multithreading
- Global storage is thread-local by default
- Bartosz Milewski blogs about models applicable to D

```
shared int flags;
```

```
// ...
```

```
int* p = &flags;           // error, flags is shared  
shared(int)* q = &flags;  // ok
```

Embedded Documentation

ddoc: the documentation tool

```
///  
/// This is a one line documentation comment.
```

```
/** So is this. */
```

```
/** And this. */
```

```
/**  
    This is a brief documentation comment.  
 */
```

etc.