Dive into D

Sebastian Wilzbach
seb@wilzba.ch

2017-01-27

Overview

Why I love D

- **Productivity** of Python
- **Performance** of C++
- **Fast** compilation

Hello World

```
import std.stdio;

void main()
{
    writeln("Hello␣TNG");
}
```

Hello World: Local imports

```d
void main()
{
    import std.stdio;
    writeln("Hello␣TNG");
}
```

Hello World: Selective imports

```
void main()
{
    import std.stdio : writeln;
    writeln("Hello␣TNG");
}
```

Hello World: Named selective imports

```d
void main()
{
    import std.stdio : hello = writeln;
    hello("Hello␣TNG");
}
```

Hello World: static imports

```d
void main()
{
    static import std.stdio;
    std.stdio.writeln("Hello␣TNG");
}
```

Hello World: C API

```d
void main()
{
    import core.stdc.stdio;
    printf("Hello␣TNG");
}
```

Hello World: UFCS (Unified Function Call Syntax)

```d
void main()
{
    import std.math, std.datetime, std.stdio;
    "Hello␣TNG".writeln; // writeln("Hello TNG")
    42.seconds.writeln; // writeln(seconds(42))
    3.pow(4).writeln; // writeln(pow(3, 4))
}
```

Hello World: Lambdas

```d
void main()
{
    import std.algorithm, std.stdio, std.uni;
    "Hello␣TNG".map!toLower.writeln;
    "Hello␣TNG".map!(x => ++x).writeln; // Ifmmp!UOH
    "Hello␣TNG".map!`++a`.writeln;
}
```

Boring history

1999	First LoC (Walter Bright)
2004	D 1.0
2007	First D 2.0 (Andrei Alexandrescu)
2010	"stable" D 2.0

- DMD (DigitalMars)
- LDC (LLVM)
- GDC (GNU)

Compile and execute

```
> dmd hello.d && ./hello
```

```
> rdmd hello.d
```

Basic types

- same size **regardless** of the architecture
- default-initialized (`.init`)
- `int.max`, `float.nan`, ...

- `bool`
- `byte`, `ubyte`, `char` (8 bit)
- `short`, `ushort`, `wchar` (16 bit)
- `int`, `uint`, `dchar` (32 bit)
- `long`, `ulong` (64 bit)
- `float`, `double`, `real` ...

Arrays

```d
int[10] staticArr;
int[] dynamicArr; // int* _ptr, size_t length

dynamicArr = [1, 2];
dynamicArr ~= 4; // [1, 2, 4]
dynamicArr[1]; // 2

auto slice = dynamicArr[0..2];
slice[0] = 42;
dynamicArr.writeln; // [42, 2, 4]
slice.writeln; // [42, 2]
```

Associatice Arrays

```d
alias string = immutable(char)[];

int[string] dict;
dict["abc"]++; // ["abc": 1]

if (auto val = "abc" in dict)
    val.writeln;

foreach (ref value; dict.byValue)
    value = 1;
```

Transitive const

```
const arr = [1, 2, 3];
arr[1] = 4; // ERROR
arr = [5]; // ERROR

const(int)[] arr = [1, 2, 3];
arr[1] = 4; // ERROR
arr = [5];

import std.experimental.typecons;
auto headConst = [1, 2, 3].makeFinal;
headConst[1] = 4
headConst = [5]; // ERROR
```

Associatice Arrays

```d
int[string] dict;
dict["abc"]++; // ["abc": 1]

if (auto val = "abc" in dict)
    val.writeln;

foreach (ref value; dict.byValue)
    value = 1;
```

Allocation

```d
struct StructColor { int r, g, b; }
class ClassColor { int r, g, b; }

StructColor c1; // stack
ClassColor c2;
c2 = new ClassColor; // GC-Heap

auto c3 = RefCounted!StructColor;
auto c4 = Unique!ClassColor;
auto c5 = scoped!ClassColor;
```

Allocation with scope guards

```
import core.stdc.stdlib : free, malloc;
auto color = malloc(Color.sizeof);
scope(exit) free(color);

import std.experimental.allocator;
auto color = theAllocator.make!StructColor;
scope(exit) theAllocator.dispose(color);
```

OOP

```d
class Dug {
    string name;
}
interface Shoutable {
    string eat()
    abstract serveDinner() {
        eat.writeln;
    }
}
```

```d
class RedDuck : Dug {
    string food;
    this(string name, string food = "bananas") {
        super(name);
        this.food = food;
    }
    override string eat() {
        import std.format : format;
        return "%s ate %s".format(name, reaction);
    }
}
auto duck = new RedDuck("Barnie");
duck.shout; // Barnie ate bananas
```

Ranges: InputRange

```
struct InputRange {
    @property empty() const;
    void popFront();
    T front();
}
```

Ranges: Input

```
struct InputRange(T) {
    private T arr;
    @property empty() const {
        return arr.length == 0;
    }
    void popFront() {
        arr = arr[1 .. $];
    }
    auto front() {
        return arr[0];
    }
}
```

Ranges: Map

```
struct MapRange(alias fun, T) {
    private T r;
    @property empty() const {
        return r.empty;
    }
    void popFront() {
        r.popFront;
    }
    T front() {
        return fun(r.front);
    }
}
```

Ranges: Example

```
import std.algorithm, std.conv, std.range,
    std.stdio;
stdin
    .byLineCopy
    .map!(line => line
            .splitter(",")
            .map!(to!int)
            .sum)
    .enumerate
    .filter(line => line.value > 10)
    .front.index;
```

Ranges

```d
struct ForwardRange {
    typeof(this) save();
}
static assert(isInputRange!ForwardRange);

struct BiDirectionalRange {
    void popBack();
    T back();
}
static assert(isInputRange!BiDirectionalRange);
```

Ranges

```d
struct RandomAccessRange {
    T opIndex(size_t index);
}
static assert(isFowardRange!RandomAccessRange);
static assert(isBidirectionalRange!RandomAccessRange);

struct SliceableRange {
    typeof(this) opSlice(size_t start, size_t end);
}
static assert(isFowardRange!SliceableRange);
```

Concurrent programming: parallel

```
import std.array, std.parallelism;
auto arr = 100_000_000.iota(0.5).array;
foreach (ref el; arr.parallel)
    el = el.sqrt;
```

Concurrent programming: Fibers

```
import std.concurrency;
auto r = new Generator!int({
    foreach (i; 0..10)
        i.yield;
});
```

Concurrent programming: Synchronization

```
shared int a;
synchronized {
    a += 10;
});

import core.atomic : atomicOp;
shared int a;
a.atomicOp!a+=(10);
```

Assembler

```
size_t syscall(size_t ident, size_t n) {
    size_t ret;
    asm {
        mov RAX, ident;
        mov RDI, n[RBP];
        syscall;
        mov ret, RAX;
    }
    return ret;
}
```

Subtyping with alias this

```
struct StupidInt {
    private theInt;
    alias theInt this;
    typeof(this) opUnary(string op)() pure nothrow
        if (op=="++" || op=="--")
    {
        static if (op == "++")
            return theInt + 2;
        else
            return theInt - 2;
    }
}
StupidInt a;
a++; // 2
```

SafeD

```
void safeFun @safe {
    int* a = new int; // safe
    int* b = &a + 5; // ERROR
}
```

Functional D

```
auto add(A, B)(A a, B b) pure {
    return a + b;
}

import std.functional : memoize;
alias fastFun = memoize!expensiveFunction;
```

String mixins

```d
auto calculate(string op, T)(T lhs, T rhs)
{
    return mixin("lhs" ~ op ~ "rhs");
}

calculate!"+"(2, 3); // 5
calculate!"-"(2, 3); // -1
```

String mixins : Bit manipulation

```d
import std.bitmanip;
struct A
{
    int a;
    mixin(bitfields!(
        uint, "x", 2,
        int, "y", 3,
        uint, "z", 2,
        bool, "flag", 1));
}
```

```d
A obj;
obj.x = 2;
obj.z = obj.x;
```

CTFE

```d
enum sum = 1 + 1;
static assert (sum == 2);

import std.math : pow;
enum p = 2.pow(5);
static assert (p == 32);

import std.regex : ctRegex;
auto ctr = ctRegex!`DL[a-z]ng`;
```

Design by Interspection (Dbl)

```
module std.algorithm.searching.commonPrefix;
/**
Returns the common prefix of two ranges
without the auto-decoding special case.

Params:
    pred = Predicate for commonality comparison
    r1 = A forward range of elements.
    r2 = An input range of elements.

Returns:
A slice of r1 which contains the characters
that both ranges start with.
 */
auto commonPrefix(alias pred = "a␣==␣b", R1, R2)
                 (R1 r1, R2 r2)
if (isForwardRange!R1 && isInputRange!R2 &&
    !isNarrowString!R1 &&
    is(typeof(binaryFun!pred(r1.front,
                             r2.front))))
```

commonPrefix: static if

```
import std.algorithm.comparison : min;
static if (isRandomAccessRange!R1 &&
           isRandomAccessRange!R2 &&
           hasLength!R1 && hasLength!R2 &&
           hasSlicing!R1) {
    immutable limit = min(r1.length, r2.length);
    foreach (i; 0 .. limit {
        if (!binaryFun!pred(r1[i], r2[i])) {
            return r1[0 .. i];
        }
    }
    return r1[0 .. limit];
}
```

commonPrefix: else

```
import std.range : takeExactly;
auto result = r1.save;
size_t i = 0;
for (;
     !r1.empty && !r2.empty &&
     binaryFun!pred(r1.front, r2.front);
     ++i, r1.popFront(), r2.popFront())
{}
return takeExactly(result, i);
```

Unittest & Assert

```d
int sum(int a, int b)
{
    return a + b;
}
unittest
{
    assert(2.add(3) == 5);
}
```

```
> rdmd -unittest -main hello.d
```

@property

```
struct Color {
    private string _name;
    @property void name(in string name) {
        _name = name;
    }
    @property string name() {
        return _name;
    }
}
Color c;
c.name = "red";
```

Contracts

```
int simple_sqrt(int n)
in {
    assert(n >= 0);
}
out (result) {
    assert(result * result == n);
}
body {
    import std.math : sqrt;
    return n.sqrt;
}
```

Invariant

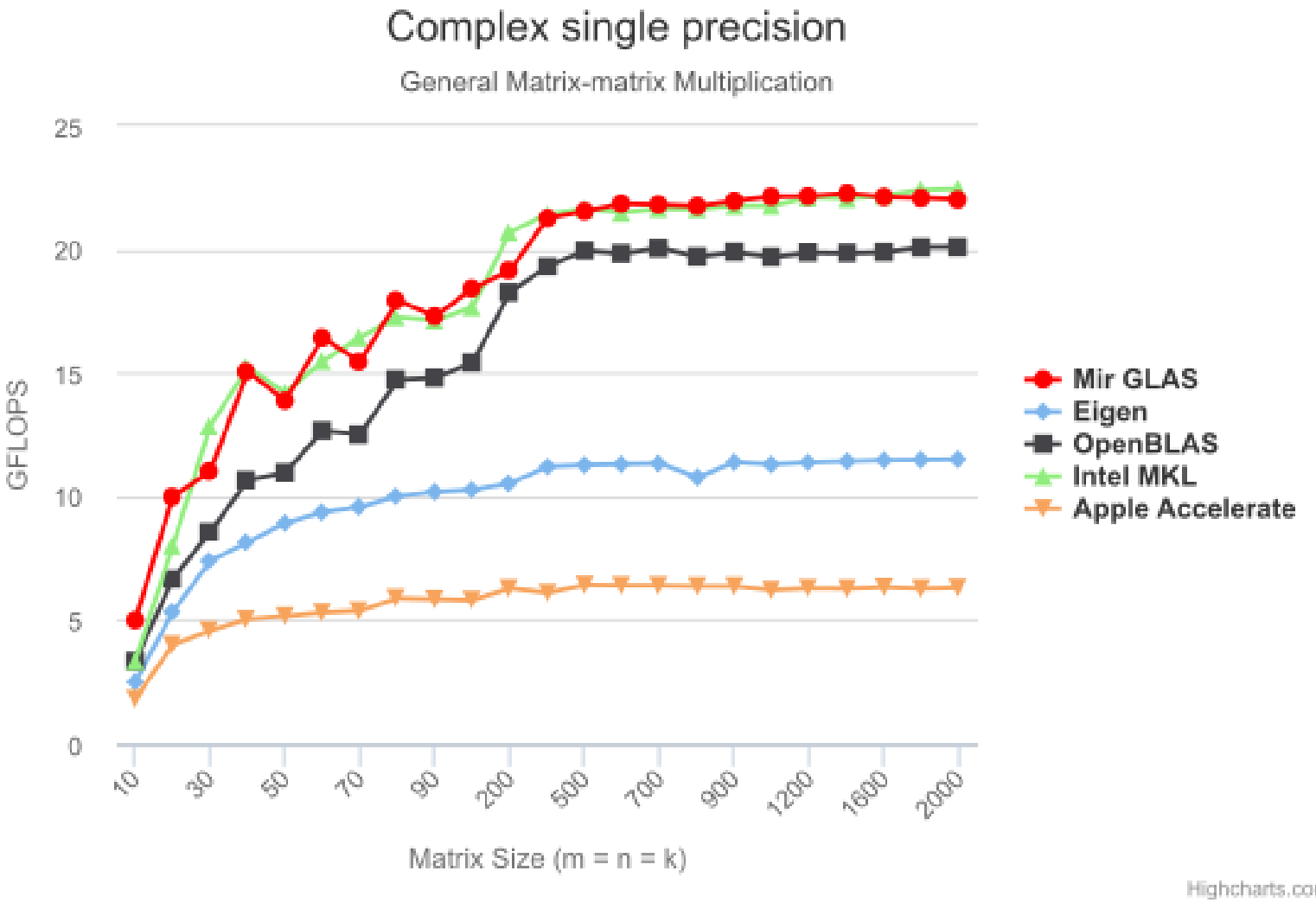
```
struct EvenVector {
    int x, y;
    invariant {
        assert(x % 2 == 0);
        assert(y % 2 == 0);
    }
}

EvenVector ev;
ev.x++; // AssertError
```

Use cases

- Vibe.d
- PowerNex
- QuantumBreak (Xbox)
- Passenger Information Systems (Munich, Berlin, ...)
- Data mining (eBay, Sociomantic, AdRoll, ...)

D For Science: Libmir



Downsides

- "Small" community (many 1 man projects)
- DUB requires adaption
- GC is still hard to opt out (betterC)
- Regressions happen from time to time
- CTFE engine can make compilation slow
- No named parameters for functions
- Phobos modules have severe quality differences (std.json, std.xml)
- IDE support differs
- `assert` should do a comparison
- Decorator hell: `@property @nogc pure @safe override`

Where to go from here?

- Get started
  - tour.dlang.org
  - `curl i.dlang.io | bash -s`
- Next dates
  - 14.2: D and the Cloud (next DLang Munich Meetup)
  - 4.5-6.5: DConf17 (Berlin)
- Stay in touch
  - blog.dlang.org
  - @D_Programming