Why D?

An opinionated talk on why you should pick D for your next project

---

Átila Neves
@atilaneves
https://github.com/atilaneves/
https://www.linkedin.com/in/atilaneves/

Who are you, again?

- PhD student at the NA60 experiment (SPS)
- Fellow in the IT dept in the security team
- Software Engineer at Cisco Systems
- Independent consultant
- Co-leader of the D programming language
- Responsible for D's standard library

From C++ to D

- 2013: Writing C++ for 15 years
- (Back to) Emacs, Arch Linux, zsh, and . . . D
- DConf, Andrei's book, first few lines
- 2014: First DConf
- Maintainer of multiple D packages
- 2016: Approached at DConf
- 2019: D co-lead

Why?

- “Enable the realisation of ideas, by shifting the way programmers think”
- Code should be beautiful
- Least amount of code possible, but not less

D example: Average line length

```d
import std;
void main() {
  auto sum = 0.0;
  auto count = stdin
      .byLine
      .tee!(line => sum += line.length)
      .walkLength;
  writeln("Average line length: ", count ? sum / count : 0);
}
```

D example: Call D from Python

```d
import autowrap;
mixin(wrapDlang!(LibraryName("mylib"), Module("mymodule")));
```

How?

- CTFE by default
- String mixins
- Compile-time reflection
- Anything can be a template parameter
- ImportC
- Built-in slices/arrays and associative arrays
- `immutable` / `shared`
- Ranges
- Memory safety / bug prevention
- Built-in documentation and testing

CTFE by default

```
uint factorial(uint i) {
    return i == 0 ? 1 : i * factorial(i - 1);
}
enum fac5 = factorial(5); // compile-time constant
enum val = sqrt(50); // from std.math
// but also:
writeln(sqrt(49)); // runtime
```

String mixins — generate code at compile-time

```
import std;
string code(int i) {
    return text(`auto x = `, i, `;`);
}
mixin(code(5));
assert(x == 5);
```

Compile-time reflection

```
struct Struct {
    int theInt;
    string theString;
}
alias members = __traits(allMembers, Struct);
static assert(members[0] == "theInt");
static assert(members[1] == "theString");
alias typeofTheString = typeof(__traits(getMember, Struct, "theString"));
static assert(is(typeofTheString == string));
```

Anything can be a template parameter

```
import std.meta: Filter;

enum isSize4(T) = T.sizeof == 4; // short template metafunction syntax
// !(...) is the equivalent of <...> in C++
alias types4 = Filter!(isSize4, int, long, float, double);
static assert(types4.length == 2);
```

ImportC — a C compiler in the D compiler

```c
// cfuncs.c
int square(int i) { return i * i; }
```

```d
// app.d
import cfuncs;
void main() { assert(square(3) == 9); }
```

Built-in slices/arrays and associative arrays

```
auto arr = [1, 2, 3];
arr ~= 4;
assert(arr == [1, 2, 3, 4]);
arr ~= [5, 6];
assert(arr == [1, 2, 3, 4, 5, 6]);
assert(arr[2..4] == [3, 4]);

auto aa = ["foo": 1, "bar": 2];
assert(aa["bar"] == 2);
aa["answer"] = 42;
```

How D is familiar

- AOT compiled statically typed language
- Procedural programming
- `if`/`while`/`do`/`for`/`foreach`, functions
- C-like syntax, C standard library available
- OOP (`class` / `interface`) like Java or C#
- Generic programming à la C++
- No compromise on runtime performance like C++

How D is not like C++

- The elephant in the room: GC
- Easier and better metaprogramming
- Modules
- Faster to compile
- Just. . . simpler and easier

Garbage Collection

Dispelling GC myths

- GCs are not magically slow
- RC is a form of GC
- Safety by default is preferable, bottlenecks are unpredictable
- `malloc`/`new` are slow anyway

```
// https://atilaoncode.blog/2015/02/11/the-craziest-code-i-ever-wrote/
template<size_t... Sizes>
constexpr auto myMakeStrings(const char (&...args)[Sizes]) ->
MyTuple<String<Sizes>...> {
    //myMakeTuple(myMakeString(arg1), myMakeString(arg2), ...)
    return myMakeTuple(myMakeString(args)...);
}
```

Metaprogramming in D

```
auto myMakeStrings(S...)(S strings) {
    return tuple(strings);
}
```

Concurrency in D — actors

D uses the Actor Model. Actors can:

- Send messages to other actors
- Create more actors
- Decide what to do for the next message it receives

Concurrency in D — example

```
void main() {
   auto tid = spawn(&writer);
   foreach (i; 0 .. 100) {
      writeln("Main thread: ", i);
      tid.send(thisTid, i);
      enforce(receiveOnly!Tid() == tid);
   }
}
void writer() {
   for (;;) {
      auto msg = receiveOnly!(Tid, int)();
      writeln("Secondary thread: ", msg[1]);
      msg[0].send(thisTid);
   }
```

Concurrency in D: `shared` and `immutable`

- `immutable` is implicitly `shared`
- `shared` can be sent to another thread
- Using `shared` is . . . tricky

Structured Concurrency

- Threads are to concurrency what `goto` is to regular programming.
- In short: don't spawn threads, don't join them
- Coming to D's standard library

dcompute — D on OpenCL/CUDA

Kernel:

```
@kernel void saxpy(GlobalPointer!(float) res,
                   float alpha,
                   GlobalPointer!(float) x,
                   GlobalPointer!(float) y,
                   size_t N)
{
    auto i = GlobalIndex.x;
    if (i >= N) return;
    res[i] = alpha * x[i] + y[i];
}
```

dcompute — CUDA invocation

```
q.enqueue!(saxpy)
    ([N, 1, 1], [1, 1, 1]) // Grid & block & optional shared memory
    (b_res, alpha, b_x, b_y, N); // kernel arguments
// equivalent to the following CUDA code:
// saxpy<<<1, N, 0, q>>>(b_res, alpha, b_x, b_y, N);
```

How D could benefit CERN

- A lot less code
- Fewer bugs
- Far fewer, if any, memory safety bugs
- Runtime performance at least the same as C++
- Compile-time reflection makes binary protocols easy

Future developements in D

- Stabilise the language
- “Finish” v1 of Phobos, the standard library
- Language editions
- Phobos v2, v3, ...
- World domination?

Conclusion

- D: beautiful and powerful
- D has nearly all of the features of C++
- D has features C++ doesn't
- D requires less code to do the same thing
- Metaprogramming in D is for mere mortals
- D produces performant code
- One language to rule them all

Questions?

Slide intentionally left blank