We're not crazy, we promise!

Trials and tribulations of a programming language designer

Átila Neves, Ph.D.

DConf 2020

Pertinent quotes

- “Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.” — Antoine de Saint-Exupéry
- “Everything should be made as simple as possible, but not simpler.” — Albert Einstein
- “Lisp programmers know the value of everything and the cost of nothing” — Alan Perlis

Why? What?

- Why do we care about `@safe`?
- Less is more: language design principles
- What I'd like to be working on
- What I'm actually working on

Why do we care about `@safe`?

- Me as a user: "because I'm lazy"
- Because it's good marketing
- Because leaning on the compiler is a good idea
- Because defaults matter

Types

- D is a statically typed language
- Nobody seems to want to change that (good!)
- Type systems don't let friends write bugs
- Trade-off: ease-of-use vs defect prevention

Memory safety = preventing bugs

- Bugs due to memory safety violations are particularly costly
- If type systems prevent bugs. . . leverage ours?
- Goal: minimise or eliminate memory safety bugs without limiting the power of a systems programming language

I don't want to go back to this:

```cpp
// In a ~10kSLOC C++ codebase
struct SomeStruct { /* .. */ };
SomeStruct onlyGodKnowsWhyThisIsHere[9];
```

Memory safety errors (single-threaded)

- Out-of-bounds access
- Issues calling `free`
    - double free
    - freeing a pointer on the stack
    - freeing `0x42`
    - freeing an aliased variable
- Reading from or writing to an invalid pointer
    - Use after free
    - Pointer to popped part of the stack
    - Dereferencing `0x42`

Mitigations in D

- Out-of-bounds: Slice access checks, foreach
- Issues related to calling `free`: don't
    - Not needed with GC allocated memory anyway
    - Don't manually allocate memory
    - Leak?
- Invalid pointer usage: ...

GC: automatic memory safety?

- In Java and Go? Yes. In D? No.
- Java doesn't have &
- Go does, but it can mean GC allocation:

```go
func NewFile(fd int, name string) *File {
  if fd < 0 {
    return nil
  }
  return &File{fd, name, nil, 0}
}
```

- D: Taking the address of locals can be an issue
- D: Manually managed pointers can be an issue

Pointers in D

- Infinite lifetime
  - GC-allocated memory
  - Address of a TLS variable
- Finite lifetime
  - Address of a local variable / parameter: fixed by DIP1000
  - malloc / allocator: fixed by . . . ???

`@safe` code is easy to write

With `-preview=dip1000` and only GC heap allocations:

- Don't do pointer arithmetic, including slicing pointers
- Don't use slices' `.ptr` property (`&slice[0]` instead)
- Don't use casts

See `https://dlang.org/spec/function.htmlsafe-functions` for more

`@safe` and `@nogc`?

- D proudly advertises that the GC isn't mandatory
- We not so proudly omit the inevitable bugs
- How can we have our GC-averse cake and eat it too?

Goal: `@safe @nogc` code

It's not currently possible to write a @safe vector library type in D:

```
auto v = vector(1, 2, 3, 4);
auto s = v[];
v ~= 5;  // could reallocate here
s[3] = 42;  // oops
```

Goal: `@safe` `@nogc` code

- `T*` should always be usable from `@safe` code
  - Infinite lifetime
  - Finite scoped lifetime (DIP1000)
  - Even if obtained from malloc (freeing however. . .)

Ok, `@safe` is great, but why by default?

- Defaults matter
- dub packages with dependencies can be made `@safe`
- Fewer bugs all around

Less is more

Less is more

The more features. . .

- The harder it is to teach the language
- The more bugs the implementation has
- The more likely they interact in unexpected ways

Out of few, many

- Guiding principle: create few orthogonal powerful primitives
- Use those powerful primitives to write everything else in
- Guideline: prefer library solutions to language features
- The default answer to language additions should be no

Simplicity as a guiding principle

- Simple: as defined by Rich Hickey in “Simple made easy”
- Simple: unentangled, decoupled

Simple vs complex

- Pure functions vs impure ones
- const vs auto
- values vs references
- Explicit vs implicit
- algorithms vs for loops

Case study: library-based OOP

- OOP doesn't need to be a language feature
- It's a library in Common Lisp (CLOS)
- Can be used in C, but not good:
    - No subtyping = no type safety (hello `void*` my old friend)
    - Manual initialisation of function slots in the virtual table
    - Hard to find actual implementation being called — gdb to the rescue!
- Do we *need* OOP in the language?

Why OOP? Returning related types

(mostly stolen from Louis Dionne's talk "Runtime polymorphism: back to the basics")

```
struct Car   { void accelerate();  }
struct Truck { void accelerate();  }
struct Plane { void accelereate(); }

??? getVehicle(string vehicle) {
    switch(vehicle) {
        default: throw new Exception("Unknown vehicle " ~ vehicle);
        case "car":   return Car(...);
        case "truck": return Truck(...);
        case "plane": return Plane(...);
    }
}
```

Why OOP? Storing related types

(mostly stolen from Louis Dionne's talk "Runtime polymorphism: back to the basics")

```
// stores types that can accelerate
???[] vehicles;

vehicles ~= Car(...);
vehicles ~= Truck(...);
vehicles ~= Plane(...);

foreach(ref vehicle; vehicles)
    vehicle.accelerate;
```

Goal: manipulate an open set of related types with different representations

Obvious D solution: inheritance

```
interface Vehicle    { void accelerate(); }
class Car: Vehicle   { override void accelerate() { /* ... */ } }
class Truck: Vehicle { override void accelerate() { /* ... */ } }
class Plane: Vehicle { override void accelerate() { /* ... */ } }

Vehicle getVehicle(string vehicle);
Vehicle[] vehicles;
```

Under the hood

```
// See https://dlang.org/spec/abi.html#classes
struct VehicleVTable {
  object.Interface instance;
  void function() accelerate;
}

struct CarImpl {
  immutable(void*)* __vptr;  // ptr to CarVTable
  void* __monitor;
  immutable(void*)* __vptr_Vehicle; // ptr to VehicleVTable
}

alias Car = CarImpl*;
```

Problems with inheritance

- Reference semantics (aliasing issues)
- Heap allocations
    - GC allocations might be an issue for certain applications
    - Non-GC allocations introduce memory management issues
- Billion dollar error semantics (null)
- Intrusive (types must opt-in ahead of time)
- Fixed binary layout (did you want monitor? You get monitor)

What I actually want (pseudocode)

```
interface Vehicle { void accelerate(); }
import lib: Motorcycle; // struct Motorcyle { void accelerate(); }
struct Car   { void accelerate(); }
struct Truck { void accelerate(); }

void main() {
    Vehicle[] vehicles = [ Car(), Truck(), Motorcyle() ];
    foreach(ref vehicle; vehicles)
        vehicle.accelerate;
}
```

What D can do now

```
import tardy;  // https://github.com/atilaneves/tardy
interface IVehicle { void accelerate(); }
alias Vehicle = Polymorphic!IVehicle;
import lib: Motorcycle; // struct Motorcyle { void accelerate(); }
struct Car   { void accelerate(); }
struct Truck { void accelerate(); }

void main() {
  Vehicle[] vehicles = [ Vehicle(Car()), Vehicle(Truck()),
                         Vehicle(Motorcyle()) ];
    foreach(ref vehicle; vehicles)
        vehicle.accelerate;
}
```

The library solution is more flexible

- None of the problems mentioned earlier
- Possibility of user-controlled policies
    - Small buffer optimisation
    - Default value or reference semantics?
    - What allocator to use when heap allocation is needed?
    - User-specified binary layout

D is a powerful language

- Let's use it to its fullest potential
- Let's prefer library solutions to language changes
- No, I don't mean "let's remove classes"

What I'd like to work on

- Making automem `@safe`
- Finishing my reflection library, mirror
- Lightning-fast unittest feedback
- Next-gen Phobos
- Easy C++ interop
- Move semantics
- Implementing "Build systems à la carte" in D

Reflection in D

- `__traits`
- `std.traits`
- Sometimes cumbersome
- Akin to using OS threads directly

Why mirror? Looping through struct member functions

```
void func() { // need this for {{, can't be at module scope
    // {{ due to alias
    static foreach(memberName; __traits(allMembers, MyStruct)) {{
        alias member = __traits(getMember, MyStruct, memberName);
        static if(isPublic!member) {  // BYOT
            static if(isMemberFunction!member) {  // BYOT
                static foreach(overload; __traits(getOverloads,
                                                  MyStruct,
                                                  memberName)) {
                    pragma(msg, __traits(identifier, overload));
                }
            }
        }
    }}
```

Why mirror? Looping through struct member functions

```
import mirror;

static foreach(overload; MemberFunctionsByOverload!MyStruct) {
    pragma(msg, __traits(identifier, overload));
}
```

mirror: runtime reflection

- Compile-time ¿ run-time
- Compile-time reflection can *generate* run-time info
- Issue: I don't know of decent use cases

What I'm actually working on instead

- Reviewing PRs
- Making dmd usable with ninja by tracking dependencies like gcc
- Improving build speeds (don't forget the linker)
  - Removing the -unittest hack
  - Possibly look into emitting fewer symbols
- Fix linker errors due to templates
  - Implementing a version of `-allinst` that works
  - Understanding the current template emission algorithm
  - Implementing an algorithm that works

What is the -unittest hack?

Once upon a time. . .

```
module std.foo;
version(unittest) {
    void helperFunction(T)() { /* ... */ }
}
```

Why doesn't `-allinst` work?

- Nobody knows
- Speculative templates make it harder:

```
static if(__traits(compiles, foo!bar))
    foo!bar;
```

Questions?

Slide intentionally left blank