MoDel ALL THE THINGS!

Using the tools the D compiler gives you

Steven Schveighoffer Dconf Online 2022 - December 17

Modeling power of D

- Interfaces and Classes
- Introspection
- User Defined Attributes (UDA)
- Generative models
- Hooking calls (`opDispatch`)

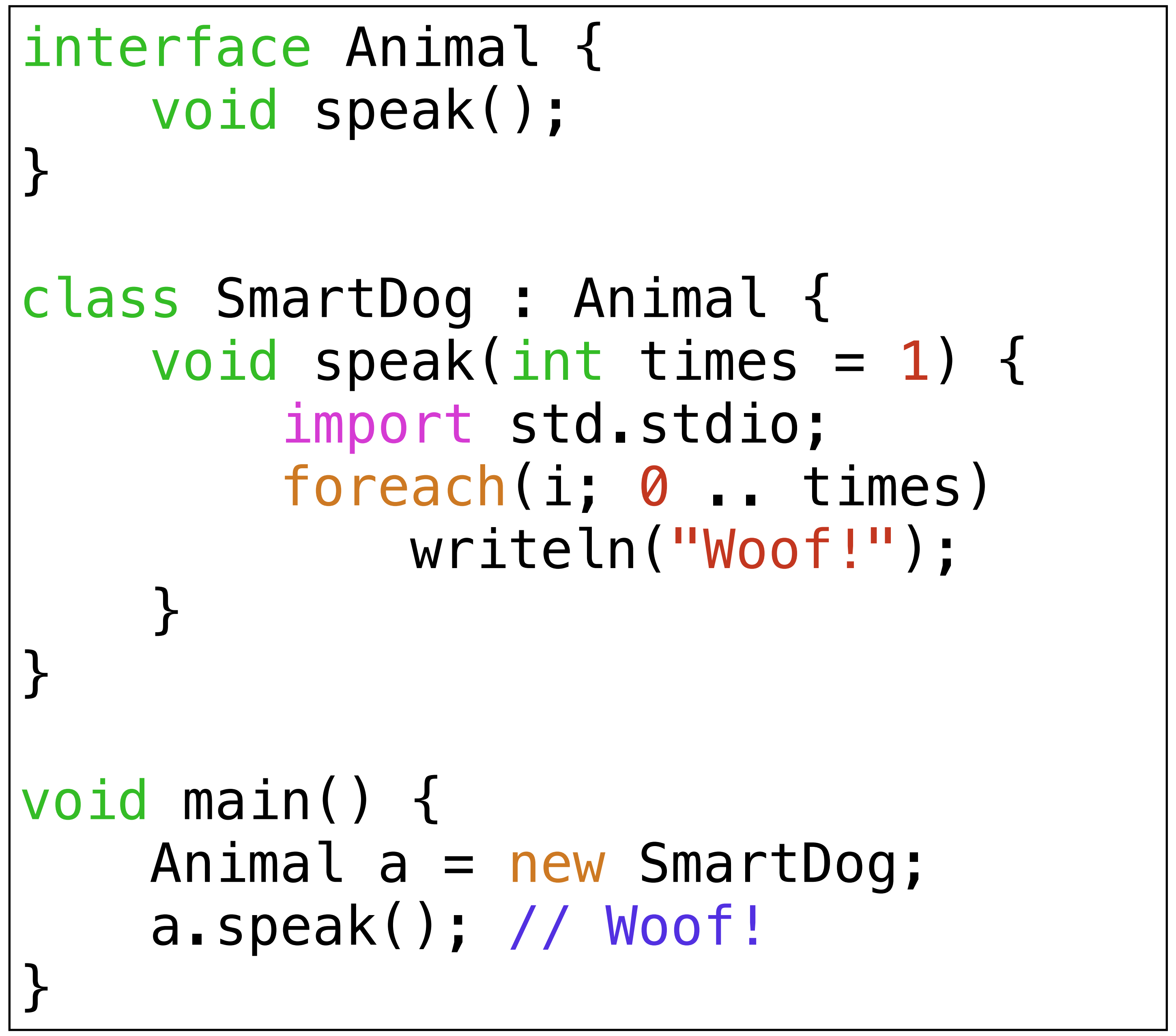
The original modeling system

- In D, the basic model system is the interface.
- Compiler provides a way to model how an API for a type should look.

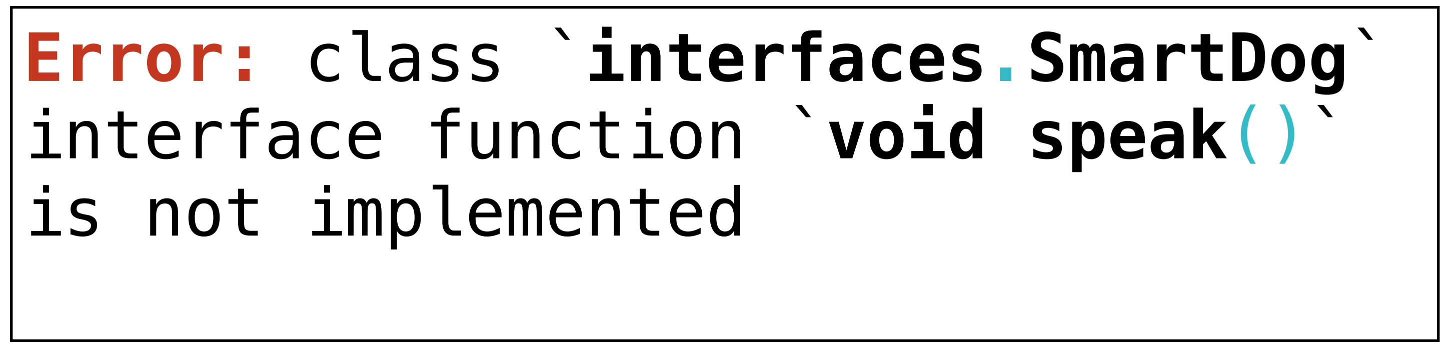
```d
interface Animal {
    void speak();
}

class Dog : Animal {
    void speak() {
        import std.stdio;
        writeln("Woof!");
    }
}

void main() {
    Animal a = new Dog;
    a.speak(); // Woof!
}
```

The original modeling system

- In D, the basic model system is the interface.
- Compiler provides a way to model how an API for a type should look.
- But it’s very rigid!
- Members cannot be optional
- No field members
- No compile-time api members (types, enums, etc)



```
interface Animal {
    void speak();
}

class SmartDog : Animal {
    void speak(int times = 1) {
        import std.stdio;
        foreach(i; 0 .. times)
            writeln("Woof!");
    }
}

void main() {
    Animal a = new SmartDog;
    a.speak(); // Woof!
}
```

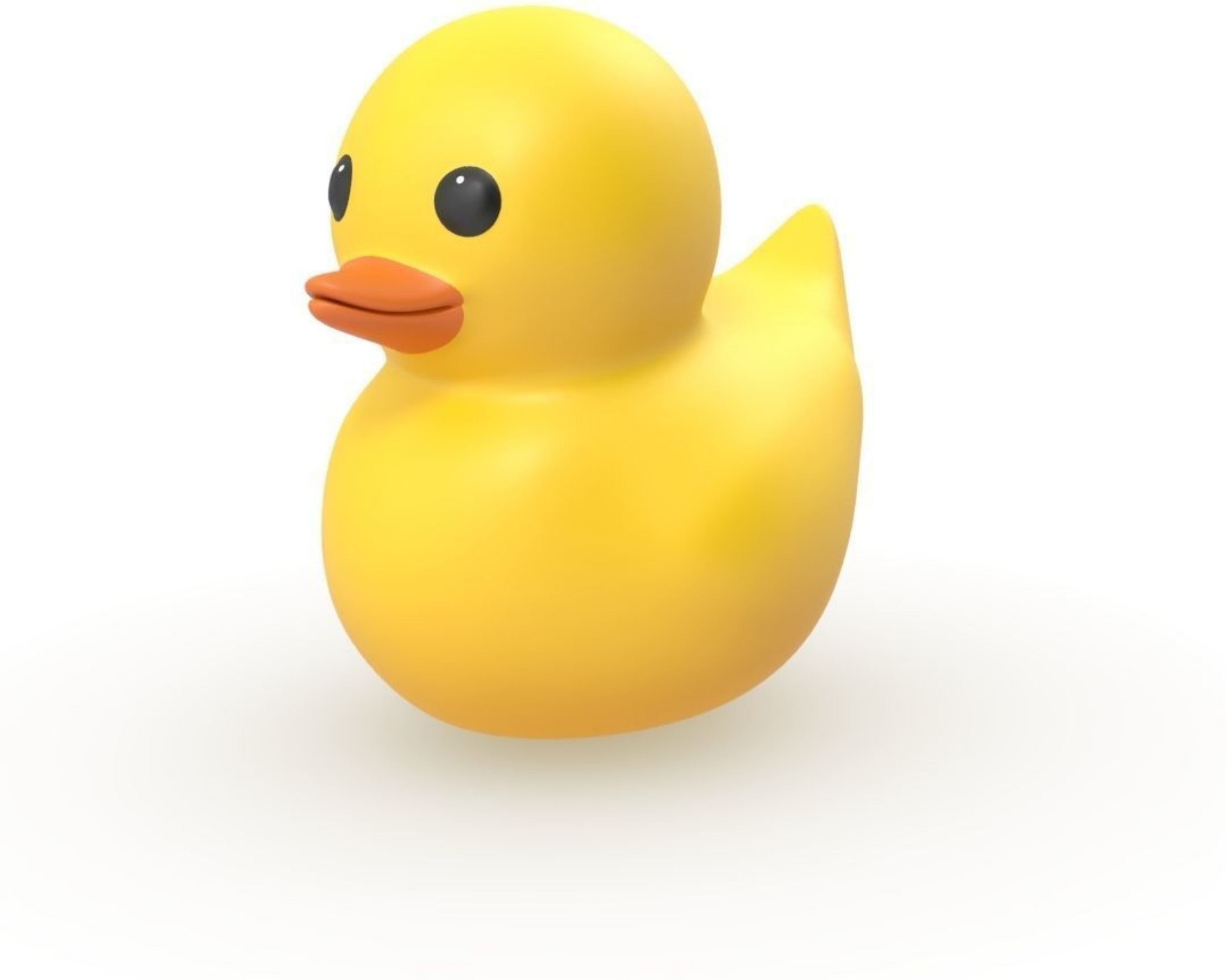


```
Error: class `interfaces.SmartDog`
interface function `void speak()`
is not implemented
```

“Duck Typing”

- “If it walks like a duck, and quacks like a duck,…”
- Less rigid about API
- But also not really “defined”

```
void trainAnimal(Animal)(Animal a) {
    a.speak();
}

class SmartDog {
    void speak(int times = 1) {
        import std.stdio;
        foreach(i; 0 .. times)
            writeln("Woof!");
    }
}

void main() {
    auto a = new SmartDog;
    trainAnimal(a);
}
```



“Duck Typing”

- “If it walks like a duck, and quacks like a duck,…”
- Less rigid about API
- But also not really “defined”
- template constraints can be used to define “interface”.
- But these are nebulous and repetitive

```
void trainAnimal(Animal)(Animal a)
  if(__traits(compiles, a.speak())) {
    a.speak();
}

class SmartDog {
    void speak(int times = 1) {
        import std.stdio;
        foreach(i; 0 .. times)
            writeln("Woof!");
    }
}

void main() {
    auto a = new SmartDog;
    trainAnimal(a);
}
```

Strawman Structs

Source: https://github.com/schveiguy/strawman

- Use a struct to model what another aggregate should contain.
- Replace complex constraint with a self-documenting model.
- Easy to point at the “strawman” and say what should be supported.

Strawman Structs

Source: https://github.com/schveiguy/strawman

- Can’t introspect uninstantiated templates.
- Create a “Type” tag that can be used to identify parameterized types
- Using “Self” type to refer to the actual type being tested.
- Support “inheritance” by defining what types it’s also like.

```
struct Any(string t, Types...) {
    enum tag = t;
    alias types = Types;
}

struct Self {}

mixin template isAlso(T...) {
    alias _alsoLike = T;
}
```

Strawman Structs

Source: https://github.com/schveiguy/strawman

```d
enum bool isInputRange(R) =
    is(typeof(R.init) == R)
    && is(ReturnType!((R r) => r.empty) == bool)
    && (is(typeof((return ref R r) => r.front)) || is(typeof(ref
(return ref R r) => r.front)))
    && !is(ReturnType!((R r) => r.front) == void)
    && is(typeof((R r) => r.popFront));

enum bool isForwardRange(R) = isInputRange!R
    && is(ReturnType!((R r) => r.save) == R);

enum bool isBidirectionalRange(R) = isForwardRange!R
    && is(typeof((R r) => r.popBack))
    && is(ReturnType!((R r) => r.back) == ElementType!R);

enum bool isRandomAccessRange(R) =
    is(typeof(lvalueOf!R[1]) == ElementType!R)
    && !(isAutodecodableString!R && !isAggregateType!R)
    && isForwardRange!R
    && (isBidirectionalRange!R || isInfinite!R)
    && (hasLength!R || isInfinite!R)
    && (isInfinite!R || !is(typeof(lvalueOf!R[$ - 1]))
        || is(typeof(lvalueOf!R[$ - 1]) == ElementType!R));
```

```d
struct InputRangeModel {
    Any!"Element" front();
    void popFront();
    bool empty();
}

struct ForwardRangeModel {
    Self save();
    mixin isAlso!InputRangeModel;
}

struct BidirectionalRangeModel {
    Any!"Element" back();
    void popBack();
    mixin isAlso!ForwardRangeModel;
}

// note, does not cover infinite ranges at the moment
struct RandomAccessRangeModel {
    Any!"Element" opIndex(size_t idx);
    size_t length();
    mixin isAlso!BidirectionalRangeModel;
}
```

std.getopt

```
import std.getopt, std.stdio;

void main(string[] args ) {
    bool verbose;
    int age;
    string name;
    auto gresult = getopt(args,
                          "verbose", &verbose,
                          "age", &age,
                          config.required, "name", &name);
    writefln("hello, %s, your age of %s is nice", name, age);
    if(verbose) {
        writeln("distributing personally identifying information on internet...");
        writeln("done!");
    }
}
```

- `getopt` uses modeling

std.getopt

```
import std.getopt, std.stdio;

void main(string[] args ) {
    bool verbose;
    int age;
    string name;
    auto gresult = getopt(args,
                          "verbose", &verbose,
                          "age", &age,
                          config.required, "name", &name);
    writefln("hello, %s, your age of %s is nice", name, age);
    if(verbose) {
        writeln("distributing personally identifying information on internet...");
        writeln("done!");
    }
}
```

- You must repeat the names 3x
- Structure split between getopt call and the declaration
- parameter order is library enforced, with not-so-good error messages

`std.getopt`

- `getopt` is essentially a DSL describing how to populate data from command line parameters

- Most options consist of name -> variable to store

🤔

std.getopt vs. schlib.getopt2

Source: https://github.com/schveiguy/getopt2

```
import schlib.getopt2, std.getopt, std.stdio;

void main(string[] args ) {
    struct Options {
        bool verbose;
        int age;
        @required string name;
    }
    Options opts;
    auto gresult = getopt2(args, opts);
    writefln("hello, %s, your age of %s is nice", opts.name, opts.age);
    if(opts.verbose) {
        writeln("distributing personally identifying information on internet...");
        writeln("done!");
    }
}
```

- No more repeating yourself!
- The compiler parses and validates the structure for you!
- UDAs better than interspersed config parameters

Using Introspection Blueprints

Source: https://github.com/schveiguy/getopt2

```
static struct memberTraits {
    string name;
    string shortname;
    string description;
    bool required;
    bool bundling;
    bool caseSensitive;
    bool incremental;
}
```

```
memberTraits getMemberTraits(string n)() {
    memberTraits result;
    result.name = n;
    static foreach(att; __traits(getAttributes, __traits(getMember, T, n))) {
        static if(is(typeof(att) == description))
            result.description = att.desc;
        static if(is(typeof(att) == optname))
            result.name = att.n;
        static if(is(typeof(att) == shortname))
            result.shortname = att.sn;
        static if(is(att == incremental))
            result.incremental = true;
        static if(is(typeof(att) == std.getopt.config)) {
            if(att == std.getopt.config.caseSensitive)
                result.caseSensitive = true;
            else if(att == std.getopt.config.required)
                result.required = true;
            else if(att == std.getopt.config.bundling)
                result.bundling = true;
        }
    }
    return result;
}
```

Comparison: rdmd

Original code

- Some options in module namespace
- Comments to “help” link options to variables
- Much repetition of names throughout.

```
private bool chatty, buildOnly, dryRun, force;
private string userTempDir;
private string[] exclusions = defaultExclusions; // packages that are to be excluded
private string[] extraFiles = [];
private string compiler = null;

int main(string[] args) {
    // Parse the -o option (-ofmyfile or -odmydir).
    void dashOh(string key, string value) {
        ... // stuff
    }

    // start the web browser on documentation page
    void man() {
        std.process.browse("http://dlang.org/rdmd.html");
    }

    auto programPos = indexOfProgram(args);
    assert(programPos > 0);
    auto argsBeforeProgram = args[0 .. programPos];

    bool bailout;    // bailout set by functions called in getopt if
                     // program should exit
    string[] loop;       // set by --loop
    bool addStubMain;// set by --main
    string[] eval;     // set by --eval
    bool makeDepend;
    string makeDepFile;
```

Comparison: rdmd

Original code

- Some options in module namespace
- Comments to “help” link options to variables
- Much repetition of names throughout.

```
        getopt(args,
            std.getopt.config.caseSensitive,
            std.getopt.config.passThrough,
            "build-only", &buildOnly,
            "chatty", &chatty,
            "compiler", &compiler,
            "dry-run", &dryRun,
            "eval", &eval,
            "loop", &loop,
            "exclude", &exclusions,
            "include", (string opt, string p) {
               exclusions = exclusions.filter!(ex => ex != p).array();
            },
            "extra-file", &extraFiles,
            "force", &force,
            "help", { writeln(helpString); bailout = true; },
            "main", &addStubMain,
            "makedepend", &makeDepend,
            "makedepfile", &makeDepFile,
            "man", { man(); bailout = true; },
            "tmpdir", &userTempDir,
            "o", &dashOh);
```

Comparison: rdmd

New code

- All options now in one place
- Must copy to globals (or make option struct global)
- Much less manual linking of options to names/variables.
- Updates to option processing much more contained.
- passThrough option now passed on function call

```
int main(string[] args)
{
    bool bailout;    // bailout set by functions called in getopt if
                     // program should exit
    struct Opts {
        @caseSensitive:
        void o(string key, string value) { ... /* stuff */ }
        void man() { std.process.browse("http://dlang.org/rdmd.html"); }

        string[] loop;
        @optname("main") bool addStubMain;
        string[] eval;
        @optname("makedepend") bool makeDepend;
        @optname("makedepfile") string makeDepFile;

        @optname("exclude") string[] exclusions = defaultExclusions;
        void include(string opt, string p) {
            exclusions = exclusions.filter!(ex => ex != p).array();
        }
        @optname("extra-file") string[] extraFiles;
        void help() {
            writeln(helpString);
            bailout = true;
        }
        // moved to struct:
        bool force, chatty;
        string compiler;
        @optname("build-only") bool buildOnly;
        @optname("dry-run") bool dryRun;
        @optname("tmpdir") string userTempDir;
    }
```

Comparison: rdmd

New code

- All options now in one place
- Must copy to globals (or make option struct global)
- Much less manual linking of options to names/variables.
- Updates to option processing much more contained.
- passThrough option now passed on function call

```
    Opts opts;
    getopt2(argsBeforeProgram, opts, std.getopt.config.passThrough);

    // todo: copy some options to module namespace
```

Generating with Models

- Models can be used to guide generated harnesses
- `std.typecons`: `WhiteHole` and `BlackHole` (added 2010 in v2.047), implemented with `AutoImplement`

```
import std.typecons : WhiteHole, BlackHole;

interface Animal {
    void speak();
}

alias SilentAnimal = BlackHole!Animal;
alias BrokenAnimal = WhiteHole!Animal;

void main() {
    Animal animal = new SilentAnimal;
    animal.speak();
    animal = new BrokenAnimal;
    animal.speak(); // asserts NotImplementedError
}
```

- Use an existing model to provide an implementation that matches or mimics that model
- For interfaces/classes, the entire model must match!

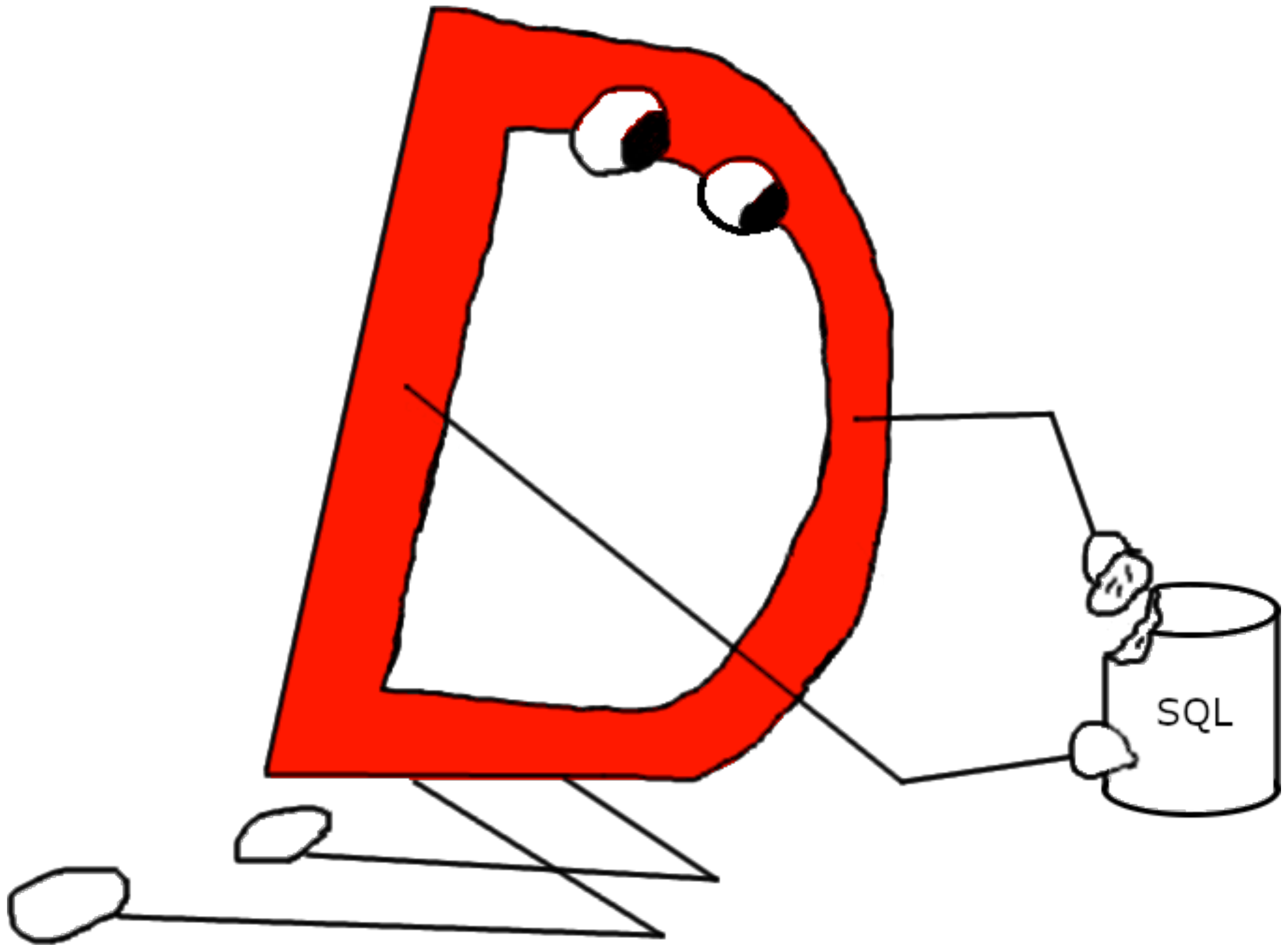
Generating with Models

- `opDispatch`: user-guided model generation (added 2009 in 2.037)
- Like `AutoImplement`, but uses actual calls to determine which methods need implementing.
- “Sparse” implementation that implements “Duck Type” usage patterns

```
struct Vector {
    float x, y, z, w;
}

struct Swizzler(T) {
    T val;
    auto opDispatch(string s)() {
        import std.algorithm, std.range, std.conv;
        return mixin("T(", s.map!(v => chain("val.", only(v)))
                     .join(','), ")");
    }
}

auto swizzler(T)(T val) {
    return Swizzler!T(val);
}

void main() {
    Vector v = {1, 2, 3, 4};
    assert(swizzler(v).wwxy == Vector(4, 4, 1, 2));
}
```

Generating with Models: `sqlbuilder`

Source: https://github.com/schveiguy/sqlbuilder



Generating with Models: `sqlbuilder`

Source: https://github.com/schveiguy/sqlbuilder

- Use models to represent table rows
- UDAs describe SQL-specific information
- Serialization can be done via introspection
- Use model to create tables in DB

```
struct Author {
    string firstName;
    string lastName;
    @primaryKey @autoIncrement int id = -1;
}

enum BookType {
    Reference,
    Fiction
}

static struct Book {
    @unique @colType("VARCHAR(100)") string title;
    int author_id;
    BookType book_type;
    @primaryKey @autoIncrement int id = -1;
}

static struct Review {
    int book_id;
    Nullable!string comment;
    int rating;
}

void main() {
    auto conn = createConnection(); // DB specific
    conn.exec(createTableSql!Author);
    conn.exec(createTableSql!Book);
    conn.exec(createTableSql!Review);
}
```

Generating with Models: sqlbuilder

Source: https://github.com/schveiguy/sqlbuilder

- Relationships can be defined with UDAs
- This allows adding constraints to the DB if desired
- Names for relationships that aren’t in the DB!

```
struct Author {
    string firstName;
    string lastName;
    @primaryKey @autoIncrement int id = -1;

    static @mapping("author_id") @refersTo!Book Relation books;
}

static struct Book {
    @unique @colType("VARCHAR(100)") string title;
    @refersTo!Author("author") int author_id;
    BookType book_type;
    @primaryKey @autoIncrement int id = -1;

    static @mapping("book_id") @refersTo!Review Relation reviews;
}

static struct Review {
    @mustReferTo!Book("book") int book_id;
    Nullable!string comment;
    int rating;
}
```

Generating with Models: sqlbuilder

Source: https://github.com/schveiguy/sqlbuilder

- DataSet type is a model of the actual relationships in the database
- Instead of fields or functions, uses opDispatch for all methods
- Model fields get mapped to Column definitions.
- Relations get mapped to properly configured DataSet with the new type
- Nullable columns automatically determined based on joins
- String-interpolation ready!

```
struct Author {
    string firstName;
    string lastName;
    @primaryKey @autoIncrement int id = -1;

    static @mapping("author_id") @refersTo!Book Relation books;
}

static struct Book {
    @unique @colType("VARCHAR(100)") string title;
    @refersTo!Author("author") int author_id;
    BookType book_type;
    @primaryKey @autoIncrement int id = -1;

    static @mapping("book_id") @refersTo!Review Relation reviews;
}

DataSet!Author ds;
auto query = select(ds, ds.books)
    .where(ds.books.book_type, " = ", BookType.reference.param);
foreach(Author a, Nullable!Book b; conn.fetch(query))
    if(!b.isNull)
        writefln("Reference book %s written by %s %s",
             b.get.name, a.firstName, a.lastName);
```

Generating with Models: `sqlbuilder`

Source: https://github.com/schveiguy/sqlbuilder

- `DataSet` implementation quite straightforward
- About 70 lines of code, but uses introspection defined elsewhere.
- Split into two `opDispatch` calls, one for fields, and one for relationships.
- Can nest as much as needed.

```
struct DataSet(T, alias core, bool AN) {
    alias RowType = T;
    enum tableDef = core;
    enum anyNull = AN;

    @property auto opDispatch(string item)() if (isField!(T, item)) {
        static if(anyNull && !isNullable!(__traits(getMember, T, item)))
            alias X = Nullable!(typeof(__traits(getMember, T, item)));
        else static if(is(getAllowNullType!(__traits(getMember, T, 
item))))
            alias X = getAllowNullType!(__traits(getMember, T, item));
        else
            alias X = typeof(__traits(getMember, T, item));

        static auto result() {
            return makeColumnDef!(X) (core, core.as,
                        getColumnName!(__traits(getMember, T, item)));
        }
        if(__ctfe) return result();
        static col = result();
        return col;
    }
```

Generating with Models: `sqlbuilder`

Source: https://github.com/schveiguy/sqlbuilder

- Actual example from my code base (Payroll Man lives!)

```
        DataSet!PlanPeriod ds;
        auto query = baseQuery.select(ds, ds.plan, ds.plan.person)
            .where(orSpec)
            .where(ds.amount_payable, " <> ", ds.amount_paid)
            .where(ds.amount_payable, " <> ", ds.amount_potential)
            .where(endGroupSpec)
            .orderBy(ds.plan.person.lastname, ds.plan.person_id, ds.period_start.descend);

        foreach(period, plan, person; conn.fetch(query))
        {
            ...
```

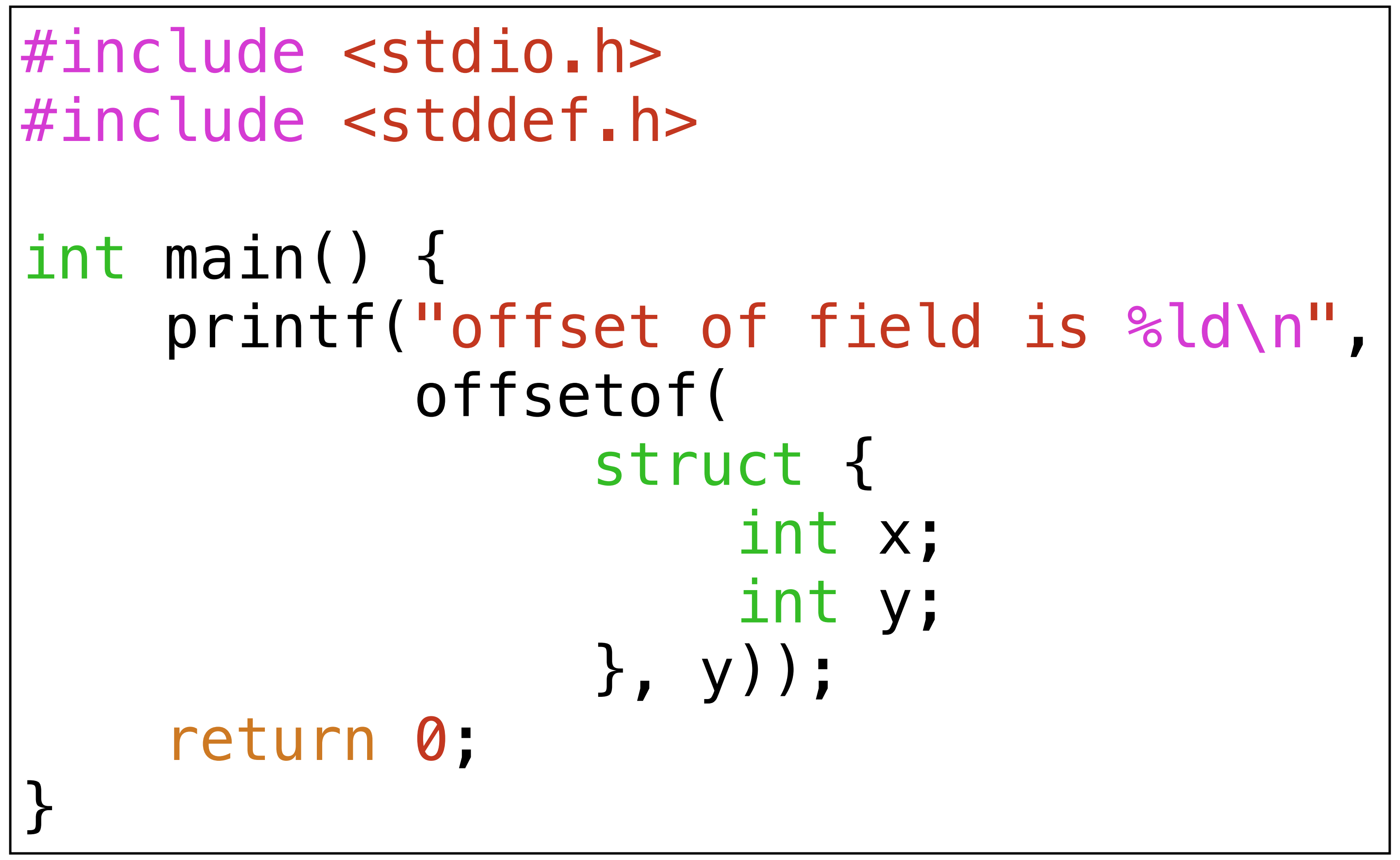
Struct lambdas

- D Structs need to be declared first

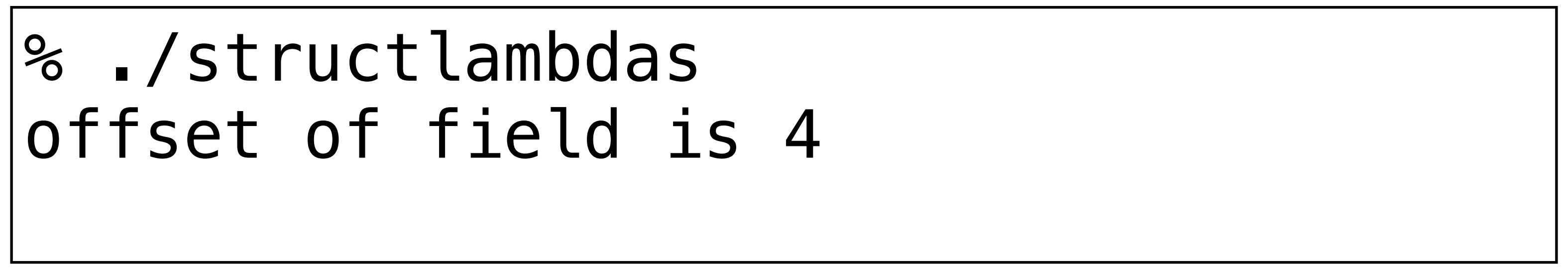
```
import schlib.getopt2, std.getopt, std.stdio;

void main(string[] args ) {
    struct Options {
        bool verbose;
        int age;
        @required string name;
    }
    Options opts;
    auto gresult = getopt2(args, opts);
    writefln("hello, %s, your age of %s is nice", opts.name, opts.age);
    if(opts.verbose) {
        writeln("distributing personally identifying information on internet...");
        writeln("done!");
    }
}
```

Struct lambdas

- C Structs can be defined in-line!
- But of course limited utility
- C should not be able to beat D here!



```c
#include <stdio.h>
#include <stddef.h>

int main() {
    printf("offset of field is %ld\n",
           offsetof(
                struct {
                    int x;
                    int y;
                }, y));
    return 0;
}
```



```
% ./structlambdas
offset of field is 4
```

Struct lambdas

- D cannot use the same syntax, because struct definitions end the declaration at the brace
- Even if the grammar allowed it, `.y.offsetof` would be a *new statement*

```
import std.stdio;

int main() {
    writefln("offset of field is %s",
                struct {
                    int x;
                    int y;
                }.y.offsetof);
    return 0;
}
```

```
structlambdas.d(5): Error: expression expected, not `struct`
structlambdas.d(5): Error: found `{` when expecting `)`
structlambdas.d(6): Error: found `int` when expecting `;` following statement
structlambdas.d(8): Error: no identifier for declarator `.y.offsetof`
structlambdas.d(8): Error: declaration expected, not `)`
structlambdas.d(9): Error: declaration expected, not `return`
structlambdas.d(10): Error: unmatched closing brace
```

Struct lambdas

- Function lambdas use a dedicated syntax, we could invent a syntax for inline struct definition
- But one possible change would be to allow struct lambdas when passing types to templates.

```
import schlib.getopt2, std.getopt, std.stdio;

void main(string[] args ) {
    auto opts = getopt2!(struct {
        bool verbose;
        int age;
        @required string name;
     })(args); // handles help automatically

    writefln("hello, %s, your age of %s is nice", opts.name, opts.age);
    if(opts.verbose) {
        writeln("distributing personally identifying information on internet...");
        writeln("done!");
    }
}
```

Conclusion

- Using models to tell the compiler the “thing” you are talking about also makes it easier to understand for users
- The real code can be messy, but the model can be minimal and pretty!
- If you find yourself implementing a modeling system manually, stop! Use the modeling system the compiler gives you.
- Use all the tools to make models easy to configure and understand:
  - introspection
  - UDAs
  - opDispatch

Thanks to @WebFreak for this awesome graphic!