Frictionless D Adoption for the Masses

or: How I Learned to Stop Worrying and Love the C Preprocessor

Átila Neves, Ph.D.
DConf, May 2019
If you remember just one thing...

- Programming is about people
Story time: John Montagu, the 4th Earl of Sandwich
Why tell a story?

- You’re now unlikely to forget the story of the invention of the sandwich
- People are sensitive to storytelling
- More parts of the brain are activated
In contrast...

- Bullet points
- Can be pretty boring
- Nobody is going to remember this slide
Stories are important . . .

- . . . Because they’re important to people
- And programming is about people
Why is Átila?

- D user since 2013
- DConf 2014 speaker
- 2014: In a new team put in charge of a legacy C codebase
Tests not included
What language?

- The test language doesn’t have to be C
- My choice was between C++ or D
- I chose C++. I didn’t want to, but I did.
Convincing is hard, let’s go shopping!

- Colleagues I’d never worked with
- 8 different meetings on the merits of automated testing
- If you’re arguing you’re losing (Dan Saks at CppCon 2016)
Change: what is good for? Absolutely Nothing

(a) Naïve model of change

(b) Satir change model

- From “Peopleware: Productive Software and Teams”
- Change doesn’t happen until people feel safe
- Also from Peopleware: people dislike change
- Loss aversion: twice as powerful as similar gain
- Automated testing chaotic enough for them
But D can call C

- From dlang.org:
  ```c
  extern (C) int strcmp(const(char)* string1, const(char)* string2);
  ```
- Unnecessary: already in core.std.meta.string
- Simpler than “real” code
- In reality:
  ```c
  extern (C) int weird_api(Foo* foo, Bar* bar, int flags);
  ```
- Foo is in foo.h, Bar in bar.h, fields in other headers
extern (C) int weird_api(Foo* foo, Bar* bar, int flags);

- flags meant to be calculated from a macro:

```
#define FLAGS(x, y, z) (((x) * 1024) | ((y) * 64) | (z))
```

Foo foo;
Bar bar;
weird_api(&foo, &bar, FLAGS(1, 2, 3));

// checking error codes is for amateurs
struct Struct {
    struct Foo {
        struct Bar* bar;
    } foo;
};

// because typing is more important than reading
#define getvalue(x) (x).foo.bar->value
• Manual wrapping too laborious
• dstep didn’t work
• Warped didn’t work...
• Calypso was non-starter
The elephant in the room

```c
extern "C" {
    #include "my_api.h"
}
```
C++’s rise to power: a historic perspective

- In practice, a superset of C
- Incremental adoption at no cost
- No loss $\implies$ no aversion
- Can’t lose by arguing when there’s no arguing
- C++’s killer feature: #include
- Conclusion: shamelessly copy C++’s approach
The goal

Emulate the C++ experience of interfacing to C:

```c
#include "nanomsg/nn.h"
#include "nanomsg/pubsub.h"

void main() {
    const sock = nn_socket(AF_SP, NN_PUB);
    scope(exit) nn_close(sock);
}
```
dpp Approach

- Use `libclang` to parse the C headers
- Translate the C AST into D syntax
  - Deal with impedance mismatch such as multiple C declarations
- Expand the translations in place
  - Originally per header file
- Translations are not meant to be checked in
- Macros?
Enabling preprocessor macro usage

- libclang has an option to remember macros
- Redefine all macros in the #included headers

```c
// was: #include "header.h"
extern(C) int add(int, int);
#define MACRO 42
```

- Run the C preprocessor on the dpp file
  - If you can't beat them, join them
- Call a D compiler on the resulting valid D code
  - Replacing the compiler is scary, wrapping it is chocolate and bunnies
• Fortunately already had bindings from dstep
• Add @safe @nogc pure nothrow to every function
  • Exception made for callbacks
• Add in to all parameters
• Wrote OOP-like wrapper for the C functions
switch(cursor.kind) with(Cursor.Kind) {
    default: return [];
    case StructDecl:
        string[] ret;

        ret ~= `struct Foo {`;
        foreach(field; cursor) {
            ret ~= translateField(field);
        }
        ret ~= `};`

        return ret;

    case FunctionDecl:

with(Cursor.Kind) {
    return [
        StructDecl: &translateStruct,
        UnionDecl: &translateUnion,
        EnumDecl: &translateEnum,
        FunctionDecl: &translateFunction,
        FieldDecl: &translateField,
        TypedefDecl: &translateTypedef,
        MacroDefinition: &translateMacro,
        InclusionDirective: &ignore,
        EnumConstantDecl: &translateEnumConstant,
        VarDecl: &translateVariable,
    ];
}
shouldCompile(
    C(
        q{
            struct Foo { int ints[4]; };
        }
    ),
    D(
        q{
            auto f = Foo();
            static assert(f.sizeof == 16, "Wrong sizeof for Foo");
            static assert(is(typeof(Foo ints) == int[4]));
        }
    ),
);
Could not execute `dmd -o- -c app.d`:
app.d(65): Error: static assert: "Wrong sizeof for Foo"

app.d:
----------
      extern(C)
      {
      struct Foo
      
      int[4] ints;
      }
    }
void main() {
  auto f = Foo();
  static assert(f.sizeof == 15, "Wrong sizeof for Foo");
  static assert(is(typeof(Foo.ints) == int[4]));
}
To understand recursion, you must first understand recursion

- Child cursors get “sent back” to the main translation function
- Cursor types get translated in a similar recursive manner
- Bonus: not having to write production code (TDD FTW)
// Apparently valid C code (who knew?)
struct BadlyNamed {
    void (*why)(void);
    struct why* (*func)(void);
};

// when inlining was new I guess
#define redOnesGoFaster() (42)
int (redOnesGoFaster)(void);
Macros: not so fast

#define OOPS1(x) (x)->foo
#define OOPS2(x) sizeof(x)
#define OOPS3(x) ((void*)(x)) // C cast (easy mode)
#define OOPS4(x) ((MyStruct*)(x)) // C cast (normal mode)
#define OOPS5(T, x) ((T*)(x)) // C cast (hard mode)

// Not valid D code
#define STRUCT_INIT(type) { STRUCT_EXTRA_INIT 1, type },
Does it work?

- The nanomsg slide works
- curl example just worked
  - With std.string.toStringz, std.conv.text, and std.stdio.stderr
- C standard library: stdio.h (printf), stdlib.h (malloc, free)
- #include <Python.h> just worked
  - Would get around 3.6 → 3.7 pyd crash
- Modulo bugs, yes!
#include "Python.h"
#include "datetime.h"
#include "structmember.h"

enum isPython3 = is(PyModuleDef);
enum isPython2 = !isPython3;
The holy grail

```cpp
#include <vector>

vector<int> v;
v.push_back(42);
```

- Has to be as easy as that
- Never mind the standard library: Qt? Eigen?
Apparently C++ is complicated

- libclang is not all it’s made out to be
  - No way to query for constexpr
  - No way to get a struct’s template parameters
- Algorithm to output D struct or class
- std::is_reference_v can’t be translated
  - Almost definitely going to be used in SFINAE
- D is the only language with any hope of translating C++
  - Template specialisations
  - Template constraints can emulate SFINAE, std::void_t, concepts?
Hacking around the C++ standard library

- Tell dpp to ignore everything in namespace std
- Define ignored cursors ourselves:

```c++
void takesVector(ref const(vector<int>));

extern(C++, "std") {
  struct allocator(T);
  struct vector(T, A = allocator!T);
}
```
Conclusion

• Programming is about **people**
• If you’re arguing, you’re losing
• Out-C++ C++
• Go forth and `#include`
Questions?

Slide intentionally left blank