Translating C to D

Translating C to D

- Why?
- How?
- The tool I made to automate it
- Aftermath

Why

- Lots of C code out there
- C code snippets in docs / stack overflow
- Want to use them in D project

Sometimes changes are needed

```
#include <windows.h>
void send(void) {
    INPUT inputs[4] = {};
    // ...
    UINT uSent = SendInput(ARRAYSIZE(inputs), inputs, sizeof(INPUT));
}
```

```
import core.sys.windows.windows;
void send() {
    INPUT[4] inputs;
    // ...
    UINT uSent = SendInput(inputs.length, inputs.ptr, INPUT.sizeof);
}
```

Sometimes not 🙂

```
// Identical in both C and D
float fabs(float x) {
    if (x < 0.0)
        return -x;
    return x;
}
```

Why

- Single file C libraries

https://github.com/nothings/stb

- Simply copy them into your project
- Some of them translated to D for the same convenience

https://github.com/adamdruppe/arsd (vorbis, ttf)

Why

Larger ports:

- https://github.com/schveiguy/draylib
- https://github.com/AuburnSounds/audio-formats
- https://github.com/d-gamedev-team/dimgui

Why

Libraries I translated:

- https://github.com/nothings/stb/blob/master/stb_perlin.h (400 LOC)
- https://github.com/RandyGaul/cute_headers/blob/master/cute_png.h (2 KLOC)
- https://github.com/matp/tiny-regex (1 KLOC)
- https://github.com/glfw/glfw (40 KLOC)
- https://github.com/andrewrk/libsoundio (10 KLOC)

Those aren't single file!

True, but still useful to be in D

- dub has no support for compiling C libraries
  - hard to configure `dub.sdl` / `dub.json`
- Dynamic linking is clumsy
  - need to make sure user has right .dll/.so
- Static linking is error-prone
  - fiddle with linker flags

Linker Errors!

```
LINK : warning LNK4098: defaultlib 'MSVCRT' conflicts with use of other libs
use /NODEFAULTLIB:library
```

```
glfw3.lib(win32_init.c.obj) : error LNK2019:
unresolved external symbol __imp__RegisterDeviceNotificationW@12
referenced in function _createHelperWindow
```

```
lld-link: error: undefined symbol: __GSHandlerCheck
lld-link: error: undefined symbol: __security_check_cookie
lld-link: error: undefined symbol: __security_cookie
```

Why

- Switch legacy code to a modern language
- DMD used to be written in 'C+'
  (C++ but sticking to C feature set + classes)
- Now in D (frontend 2015, backend 2018)
- Same for tools such as Digital Mars 'make':

https://dlang.org/blog/2018/06/11/dasbetterc-converting-make-c-to-d/

How

Relatively easy, because D has:

- Familiar syntax
- C features
  - types (`char[20]`, `int*`, `float`, `struct`)
  - operators (`<<` `&` `*`)
  - statements (`goto` `switch` `do` `while` `for`)
- "If it looks like C and compiles, it acts the same"

How

- Programming in D for C Programmers: https://dlang.org/articles/ctod.html
- Copy the C code
- Give it a `.d` extension
- Add `extern(C):` on top
- Edit until dmd stops giving errors

Approach

Walter Bright approach:

- Do it one function at a time
- Run the test suite after each translation
- Resist the urge to fix, refactor, clean up, etc.

Approach

My approach:

- Do it all at once
- Once it's finally done, debug the things that are broken
- Still, don't refactor early

Changes: syntax

- `obj->member` `obj.member`
- `(int) x` `cast(int) x`
- `sizeof x` `x.sizeof`
- `NULL` `null`
- `1.f` `1.0f`
- `typedef struct {} S;` `struct S {}`
- C identifiers that are D keywords, `in` `in_`

Changes: statements

- Add `default: break;` to `switch` in D
- Add `goto case;` for switch case fall through
- Empty statement `;` disallowed, `for (;;);` `for (;;){}`
- `if (errorCode = apiFunc())` disallowed
  - workaround: `if ((errorCode = apiFunc()) != 0)`
  - or: `if (auto errorCode = apiFunc())`

Changes: pointers

- D: use `&` to take address of function, `f(&callback)`
- D: use `.ptr` to take address of static array
- No implicit pointer casts like C: `char* x = malloc(1);`

Changes: basic types

```
unsigned short int x;
unsigned short x;
uint16_t x;
uint16 x;
__u16 x;
```

D:

```
ushort x;
```

Changes: basic types

- Be careful of variable sizes
- D's 8-byte `long` in C is `long long`

C: `sizeof long`	Windows	Linux
32-bit	4	4
64-bit	4	8

- `import core.stdc.config: c_long;`

Changes: complex types

- C types read like expressions

```
int x[3][4];
char *(*bar)(int);
```

- D types read from right to left

```
int[4][3] x;
char* function(int) bar;
```

Initializers

C99 has expressive struct/array initializers with designators

```
void f() {
    drawRect(&(Rectangle){.pos = {2, 4}, .size = {16, 32}});
    int arr[] = {[3] = 30, [2] = 20};
}
```

```
void f() {
    auto tmp = Rectangle(pos: vec2(2, 4), size: vec2(16, 32));
    drawRect(&tmp);
    int[4] arr = [3: 30, 2: 20];
}
```

Pitfall: initializers

In D, `float` and `char` initialize to `NaN` / `0xFF`, so make it explicit:

```
float f = 0;
char[512] buffer = 0;
```

In C, local variables are uninitialized by default. In D you need `= void`

```
void fun() {
    char[512] buffer = void;
}
```

Pitfall: static

- C has no name mangling
- public functions often have prefixes: `sqlite3_open`, `glfwInit`
- private functions use `static`

```
static void init() {}
```

- D's `static` is different! Needs mangling to avoid name conflict.

```
private extern(D) void init() {}
```

Pitfall: passing static arrays by value

- Static arrays are consistently value types in D
- In C, they are passed as a pointer

```
void modify(char c[16])
{
    c[0] = 0;
}
```

```
void modify(ref char[16] c)
{
    c[0] = 0;
}
```

Pitfall: passing static arrays by value

Looks less obvious in actual code:

```
typedef char RegexCharacterClass[(UCHAR_MAX + CHAR_BIT - 1) / CHAR_BIT];

static inline int regexCharacterClassContains(const RegexCharacterClass klass
  int ch) {
  return klass[ch / CHAR_BIT] & (1 << ch % CHAR_BIT);
}
static inline int regexCharacterClassAdd(RegexCharacterClass klass, int ch)
  klass[ch / CHAR_BIT] |= 1 << (ch % CHAR_BIT);
  return ch;
}
```

Pitfall: address of slice

I've done this with OpenGL's `glBufferData`:

```
float[] vertices = [-0.6f, -0.4f, 0.6f, -0.4f, 0.0f, 0.6f];
void bufferData(size_t size, void* buf);
void main() {
    bufferData(vertices.sizeof, &vertices); // WRONG
}
```

- `&vertices` points to the slice's `(length, ptr)` pair, not the data!
- Similarly, `vertices.sizeof` is simply `16`

Macros

- Before C files are compiled, the pre-processor expands macros
- D doesn't have it
- The C Preprocessor vs D: https://dlang.org/articles/pretod.html

Macros

Some are easy

```
#include "./lib/something.h"
#define PI 3.1415926535
#define SQR(X) ((X) * (X))
#ifdef _WIN32
#endif
```

```
import lib.something;
enum PI = 3.1415926535;
auto SQR(T)(T x) { return x * x; }
version (Windows) {}
```

Macros

When non-trivial, expand them, or use string mixins

```
#define INITIAL_CHECK if (!lib_initialized) return;
void libFunc(void) {
    INITIAL_CHECK
}
```

```
enum INITIAL_CHECK = "if (!lib_initialized) return;";
void libFunc() {
    mixin(INITIAL_CHECK);
}
```

Macros

- When the macros are part of a cross-platform API, give up

C:

```
#include <stdatomic.h>
```

D:

```
import core.atomic;
```

Pitfall: Macros

- Watch for arguments with side effects

```
#define SQR(X) ((X) * (X))

void f(void)
{
    int x = 3;
    int y = SQR(x++);
}
```

Reduce tedious typing

- Translating by hand is tedious
- VIM macros only help so much
- dstep can translate types, but only in headers, and gives errors:

```
/usr/include/alsa/input.h:65:50: error: unknown type name 'FILE'
/usr/include/alsa/input.h:66:69: error: unknown type name 'ssize_t'
/usr/include/alsa/input.h:73:53: error: unknown type name 'size_t'
```

- Regular expressions don't scale
- I need a tool using a C parser

ctod

Try it!

https://dkorpel.github.io/ctod/

```
dub fetch ctod
dub run ctod -- yourfile.c
```

ctod: concept

- Uses tree-sitter parser, which has excellent error recovery
- Performs string replacements on AST nodes
- Prints back as close to valid D as possible

ctod: development

- Run `ctod` on a C file
- Inspect output, look for invalid D
- Enter the C code in the [tree-sitter playground]()
- Add code to recognize and translate the pattern
- Repeat

ctod: development

- More and more advanced
- Parses types and function signatures
- Keeps symbol table so add `.ptr` to static arrays
- Dangerously close to C-compiler

ctod: limitations

- macro translation very primitive
- Parser trips up on weird macros

```
GLFWAPI void glfwFun() {}
```

- But: no errors!

Aftermath

- Translation done, time to make it more idiomatic

```
enum { REGEX_NODE_TYPE_EPSILON, REGEX_NODE_TYPE_CHARACTER }
memcpy(a, b, sizeof(a));
for (int i = 0; i < n; i++) //...
```

```
enum NodeType { epsilon, character }
a[] = b[];
foreach (i; 0 .. n) //...
```

Aftermath

- Add attributes
- Get `nothrow` `@nogc` for free
- `pure`: replace global error variable with returned error code
- Can we add `@safe`?

Aftermath: safe

- Replace C-strings / pointer-length-pairs with slices

```
void fun(int* ptr, size_t length, const char *str)
```

```
void fun(int[] data, const(char)[] str)
```

- Replace pointer math with indices
- Replace re-inventions of dynamic arrays

Aftermath: safe

- I made the png lib translation `@safe` and added fuzz tests
- Basically all array index operations were unsafe
- Checks that were there not robust to overflow:

```
int readLength = readBits(...);
int backwardsLength = readBits(...);
CHECK(s->out - backwardsLength >= s->begin);
CHECK(s->out + readLength <= s->end);
```

- Dangling pointer into array after it gets resized

Should you translate C to D?

- Still a lot of manual work
- Translation gets behind when update releases
  - I translated glfw 3.3.2, now at 3.3.8
- DMD can now compile .c files! (ImportC)
- Small / stable code: go for it!
- If you're maintaining it: go for it!