They're Coming To Take Me Away

- or -

Adding Modules to C in 10 Lines of Code

by Walter Bright
April 2022
https://twitter.com/WalterBright
https://dconf.org/2022

#include is as primitive as a VHS copy of 2001: A Space Odyssey

- C'mon, why haven't we fixed that yet?
- It took C++ 20 years to do modules
- Modules must be hard
- My method can work with your C compiler, too!

I'm Sticking My Neck Out

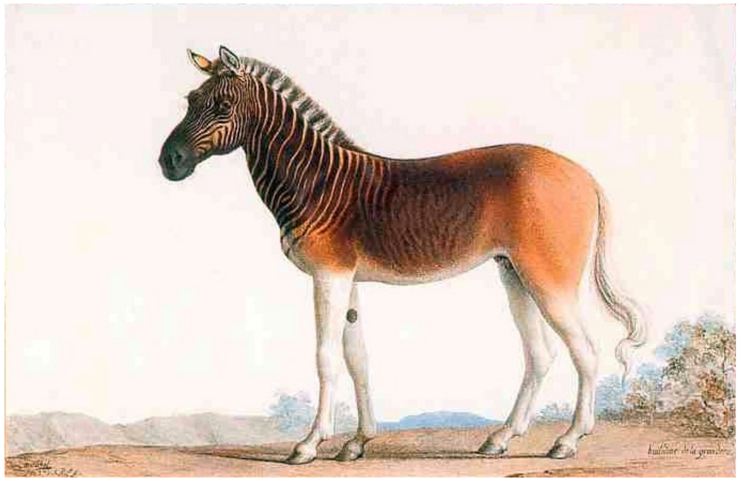
Be honest - how many of you are here to watch me crash and burn?

But I'm Going To Make You Suffer First

And present some boring (but necessary) background

D is designed to be easy to interface to C

- Zero cost
- Compatible types
- Familiar syntax
- Compatible semantics
- Leverage existing C code



To Make C Code Accessible From D

- Simply translate the C .h file to D
- It's easy
- Doesn't take long
- Shouldn't be a problem

~~To Make C Code Accessible From D~~

- ~~Simply translate the C .h file to D~~
- ~~It's easy~~
- ~~Doesn't take long~~
- ~~Shouldn't be a problem~~

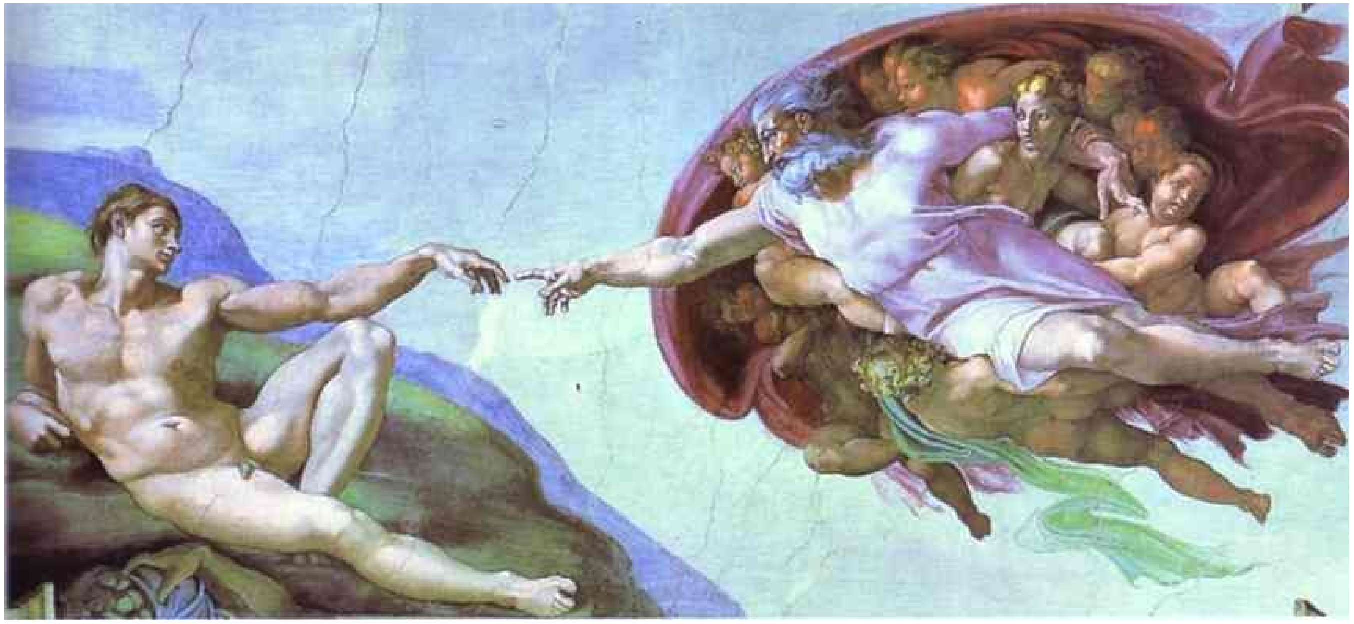
What would be the simplest, most obvious, most perfect way for D code to simply get all the declarations from, say, stdio.h?

```
import stdio;
```

Behold!

ImportC

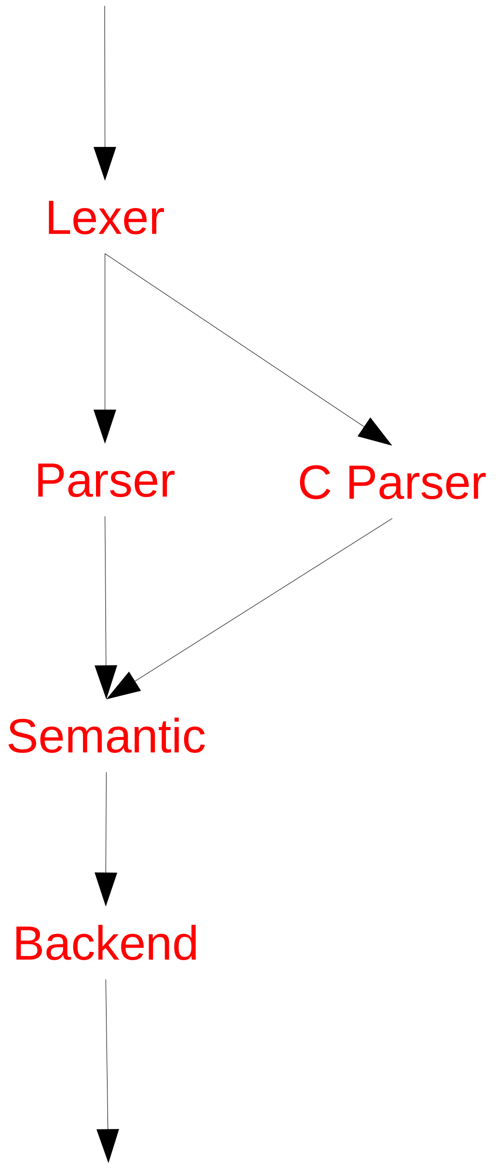
was conceived!



In order for that to work, the D compiler added a builtin C compiler. Yes, a complete C compiler. Was this like going around the horn?

Not totally, as the D and C compiler share a lot of code.

ImportC is just a new parser



And so, the D programmer could mix and match D and C files directly, no need for translation.

Enough of that story

On to Walter's Folly

The Foundational Idea

Q: What is the root data structure of a compiled piece of C code?

It's Just A Symbol Table

- Functions
- Variables
- Enums
- Structs
- Unions
- Typedefs

Which we can represent as a simple
array of global declarations.

hello.c

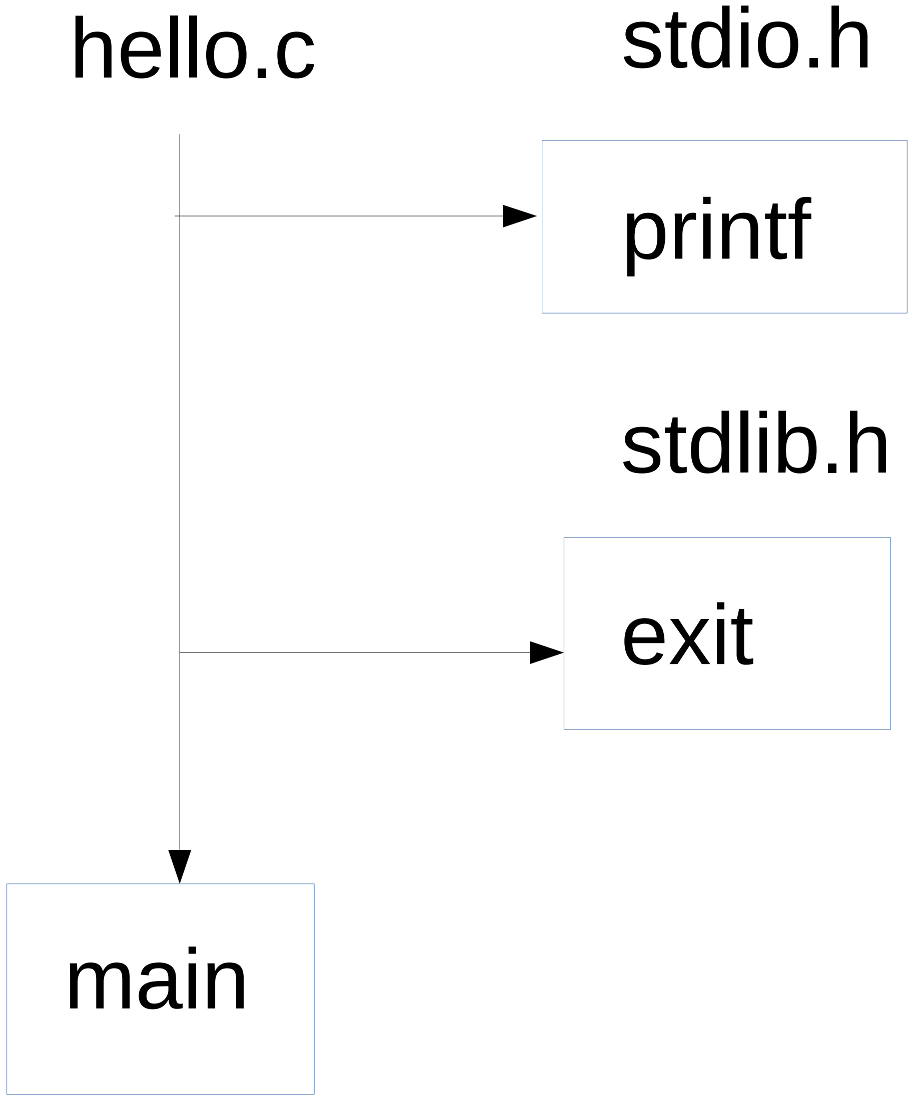
```
#include <stdio.h>
#include <stdlib.h>

void main() {
    printf(“hello world\n”);
    exit(0);
}
```

Symbol Table with #include

printf
exit
main

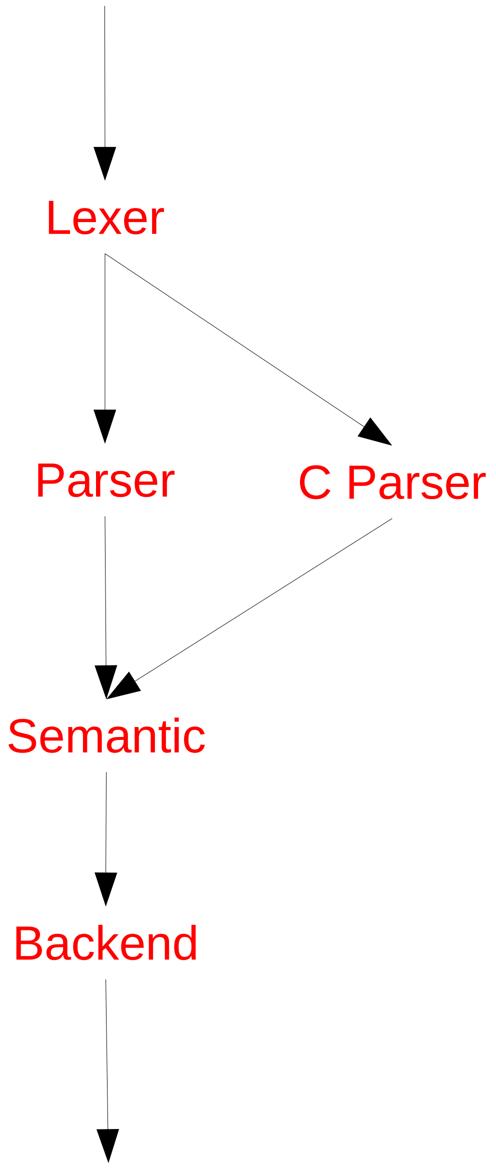
Symbol Table with Imports



Problems To Solve

- Instruct the C compiler to import another C file
- Run a separate instance of the C compiler on the imported C file to generate a symbol table for it
- Adjust the symbol table lookup so if it isn't found in the global symbol table, to look in the imported C file symbol tables

Recall How Compiler is Organized



Instruct C Compiler to Import Another C File

- Add __import keyword
- Hijack D parser and parse it like D's import declaration

https://dlang.org/spec/module.html#import=declaration

Which looks like:

__import name;

(with some extra syntax to import only selected symbols, or to create aliases to

Run A Separate Instance of the C Compiler

Once the ImportDeclaration is added to the C symbol table, when the semantic
Analyzer sees that, it creates another instance of the C compiler, runs it
To compile the imported code, and adds that module's symbol table to the
Importer's symbol table.

Semantics To Look Up Names In Imports

The D and C semantics are shared code, so the D semantics that
Look up unresolved names in imports is already there.

The 10 Lines of Code

https://github.com/dlang/dmd/pull/13539

```
if (token.value == TOK._import) // import declaration extension
{
    auto a = parseImport();
    if (a && a.length)
        symbols.append(a);
    return;
}
```

Plus a couple declarations. That's it!

Goodies

1. No need for preprocessor barriers or #pragma once. Modules are only compiled once, no matter how many times they are imported, and how many other modules import them. Even circular imports work. This makes for quick compiles.

2. The semantics of a module do not affect any modules it imports

3. The preprocessor macros from an imported module are hidden from the importer

4. Static symbols in the imported modules are not visible to the importer

5. No extra disk files to store module symbol tables and track dependencies

6. Don't have to write C .h files anymore - just import the .c file

Not So Good

1. The preprocessor macros from an imported module are hidden from the importer

2. Macro metaprogramming is right out

3. Many (most) C header files declare macros for use by the #including file

But do you really need those macros?

Name Collisions

```
a.h: int f;

b.h: void f();

c.c: __import a;
     __import b;
     void test() { f(); }  // error: a.f or b.f?
```

(Of course, this wouldn't work if a.h and b.h were #include'd.)

module c:

```
__import a;
__import b : f;

void test() { f(); } // ok, b.f overrides a.f
```

Retrofitting Into Existing C Compiler

- As long as the compiler doesn't use global variables to store the compiler's state
  - i.e. it can run another instance of itselve
- This is the problem with adding it to the Digital Mars C compiler – it's all global variables
  - Still doable, just would require a fair chunk of work

Why Has Nobody Done This In 40 Years?

- Including me
- I can't explain it
- I don't know what went on with C++ for 20 years

How Does It Compare With C++ Modules?

- I know very little about C++ modules
- But it looks pretty similar
- One big difference is C modules derive their name from the file name, i.e.:

```
__import stdio;
```

Looks for stdio.h, then stdio.c

Would This Idea Work For C++?

- I don't see any impediment
- I'd avoid the C++ BMI (Binary Module Interface)

C++ can #include C files

- C can #include C files
- C++ can #include C files
- C++ can #include C++ files
- hmmmm

Let's Compare With Import!

- D can import D files
- D can import C files
- C can import C files
- hmmmmm

Yes, A D Source File Can Be Imported, With No Extra Coding!

Which means....

Function Overloading

math.d:

```
int square(int i) { return i * i; }
double square(double d) { return d * d; }
```

compute.c:

```
__import math;

double sumOfSquares(int i, double d)
{
    return square(i) +  // calls square(int)
           square(d);   // calls square(double)
}
```

No more need for _Generic!

Templates for C, OMG!

math.d:

```
T square(T)(T i) { return i * i; }
```

compute.c:

```
__import math;

double sumOfSquares(int i, double d)
{
    return square(i) + // calls square!int()
           square(d);  // calls square!double()
}
```

Ruff Edges

- No scheme for public/private members
- Typedefs and parsing difficulties
- Macros for manifest constants
- Like putting a V8 in a Volkswagen Bug

So You Decide

- Did I deliver the goods?
- Are you going to rush out to add modules to your favorite compiler?
- Is #include really most sincerely dead?

References

- https://dlang.org/spec/importc.html
- https://dconf.org/2022