WebAssembly Live! Mike Hand, DogFoodCon, October 4 2019
For those who did not attend my talk live, I do not have slides -- the majority of the time was a live demo, preceded by some mild preamble. I've created this sheet of info and references, but it will likely largely be of use to jog the memory of those who attended. Either way, I hope it is helpful! Please note that WebAssembly is undergoing a lot of change, and this outline of how to get up and running could go out of date at any time.

WebAssembly is:

- "Writing code in a language and compiling it to an assembly format native to the javascript virtual machine"
  - Could really be written in any language, including javascript itself (or typescript)
  - Not "browser-native", but "javascript-native" (node.js, etc)
- Never JITted, always "optimized"
- A good fit for intensive client side processing
- **Unable to access the DOM**

The demo leveraged Emscripten (https://emscripten.org/docs/getting_started/downloads.html) for several reasons:

- Easy to acquire
- Easy to get started with
- I was able to find resources to help *me* get started with using it after a couple false starts elsewhere

I won't reproduce the steps to get up and running with Emscripten here because they are laid out well on their website, and in case they change over time.

I'll deviate from the order I demoed things in here to eliminate some suspense: when running these projects locally and trying to include .wasm files locally, you will run into CORS issues. The easy way to get around this is using

```
emrun --port 8080 .
```

from the directory your project is contained in. (please note the "." is important in this command)

I took the following simple cpp file:
demo.cpp
```
#include<iostream>

int main(){
  std::cout << "hello dogfoodcon" << std::endl;
}
```

And ran emscripten against it:
```
em++ demo.cpp -o demo.html
```

By default this produces a ton of HTML/CSS/JS and a wasm file. We can see the output we expected, but it's all wrapped up in this virtual console that is really heavy and we'd like the code we write to eventually be able to be called a lot more interactively. So we edited the HTML file down to just the bare essentials:

```
<!doctype html>
<html lang="en-us">
  <body>
    <script async type="text/javascript" src="demo.js"></script>
  </body>
</html>
```

Refreshing should now result in a blank page, but the output should appear when viewing the js console.

At this point we modified our first cpp file and introduced a second cpp file:

demo.cpp

```
#include<iostream>
#include "adding.cpp"
int main(){
  std::cout << "adding 2+3: " << addTwoNumbers(2,3) << std::endl;
}
```

adding.cpp

```
int addTwoNumbers(int x, int y){
  return x + y;
}
```

Now when we compile, we don't want the HTML to be our result (since we've modified it the way we like it), but rather the javascript that gets produced which will load our wasm file:

```
em++ demo.cpp -o demo.js
```

We can once again refresh the page and checking the js console, we should see our output.

This is still a pretty poor way to use the code we wrote, however. It's "single use" and hard to engage with, requiring a page reload to run the result again (and a re-compile to change the inputs). Emscripten provides a simple command line flag to expose methods to the "outside world" which is helpful. There are two examples below, one to export all methods, globally, and the second to target specific methods you want to expose from wasm to javascript:

```
em++ demo.cpp -s EXPORT_ALL=1 -o demo.js
em++ demo.cpp -s EXPORTED_FUNCTIONS="[_addTwoNumbers]" -o demo.js
```

This *will* expose your cpp method to be accessible by javascript, but it’s a bit circuitous. You can attempt to find your method by starting to type in the javascript console:

```
Module.__Z
```

And then searching through the autocomplete options for something with `addTwoNumbers` in it’s name from there. For me, in the demo, it was:

```
Module.__Z13addTwoNumbersii
```

There are two things going on here, `Module` and “name mangling”. The `Module` object is created by emscripten as a place to store all the related code that it complies and makes available to us.

“Name mangling” is a feature of C++ to avoid naming collisions on method names. You can read up on that separately, and we’ll work around it below. But for now, you can call that method on demand, pass in two numbers of your choosing and you should see sensible results. You can also write “more extensive” javascript that references this method, for instance something like:

```
[1, 2, 3].forEach( function(e){
    console.log(Module.__Z13addTwoNumbersii(e, e))
})
```

As promised, a simple way to work around name mangling -- simply modify your cpp file as follows:

adding.cpp

```
extern "C" {
  int addTwoNumbers(int x, int y){
        return x+y;
  }
}
```

Once again recompiling and refreshing the page, we can now access this method from the js console as:

```
Module._addTwoNumbers
```

Please note that the first underscore is also automatically added by emscripten and will always be present.

Now we have a much friendlier, much more interactive way of calling the cpp method that we wrote, and we can now interact with it freely from the confines of javascript.

Please note that WebAssembly is undergoing a lot of change, and this outline of how to get up and running could go out of date at any time.

Mike Hand, October 2019.