WEBASSEMBLY

- A Web Standard available across all 4 major browsers
- A binary (and corresponding text) format for executable code
- Designed to execute near the speed of native machine code
- A compilation target for a wide range of languages
- Executed in a sandbox for safety

HISTORY AND PRESENT

- Preceeded by many vendor specific attempts to speed up Web Applications
  - NaCl
  - asm.js
- Announced in 2015
- Close to 75% support in deployed web browsers
- Can be compiled to asm.js via a polyfill

WEB ASSEMBLY LANGUAGES

- WAT - Web Assembly Text Format
- C/C++
- Rust
- TypeScript/AssemblyScript
- Upcoming: *.NET

WEB ASSEMBLY TEXT FORMAT

```
(module
  (func $add (param $lhs i32) (param $rhs i32) (result i32)
    get_local $lhs
    get_local $rhs
    i32.add)
  (export "add" (func $add))
)
```

Compile using the wat2wasm tool:

```
wat2wasm simple.wat -o simple.wasm
```

DEVELOPMENT ENVIRONMENTS

- WebAssembly Studio
  - Missing C++ Support
- Visual Studio Code
- CLion

TOOLCHAIN FOR C++

BASE FEATURES

- C standard library
- OpenGL (converted to WebGL calls)
- pthreads (experimental)
- 32-bit at this point, 64-bit support planned

MEMORY

- Limited, linear memory space
  - Initial and optional maximum size specifyable
- Fully isolated from the memory of other modules
- Memory returned by WebAssembly calls are converted to indices into linear memory in JavaScript

EMSCRIPTEN

- Use emcc to compile C++ code
- Can target asm.js(default) and WebAssembly
- Generates JavaScript glue code in addition to wasm binary

BUILDING WASM

```
emcc -s WASM=1 -o hello.html hello.c
```

- Generates JavaScript glue code as well as an example HTML file
- Executes the "main" function on load if it exists

RUNNING WASM

```
emrun --port 8081 hello.html
```

- Starts a server to avoid security issues with the Web Browser's security features

SIMPLE WORKED EXAMPLE

```c
#include <stdio.h>
int main(int argc, char ** argv) {
  printf("Hello, world!\n");
}
```

"main" function called automatically by the generated .js file on load

CALLING C/C++ FUNCTIONS FROM JAVASCRIPT

```
#include <emscripten.h>
EMSCRIPTEN_KEEPALIVE
int fib(int n) {
    ...
    return b;
}
```

Use "EMSCRIPTEN_KEEPALIVE" macro to force the compiler to include the function even if it appears not to be used

CALLING C/C++ FUNCTIONS FROM JAVASCRIPT (CONTINUED...)

```
const fib = Module.cwrap('fib', 'number', ['number']);
```

- Wrap the C function in JavaScript using the "cwrap" function using the following arguments
  - Function name
  - Return type
  - Array of function argument types

CALLING C/C++ FUNCTIONS FROM JAVASCRIPT (CONTINUED...)

Compile using:

```
emcc -s WASM=1 -o fib.js fib.c -s EXTRA_EXPORTED_RUNTIME_METHO
```

The "-s EXTRA_EXPORTED_RUNTIME_METHODS='["cwrap"]'" option is required to wrap C functions in JavaScript

CALLING JAVASCRIPT FROM C++

```
EM_JS(void, drawCircle, (float y, float x, float r), {
    const canvas = document.querySelector('canvas');
    ...
    context.fill();
});
```

- Use the "EM_JS" macro to wrap JavaScript code into a function callable in C/C++ using the following arguments:
  - Return type
  - Function name
  - Function arguments

CALLING JAVASCRIPT FROM C++ (CONTINUED...)

- Then just call the function in C++ code:

```
int main() {
    drawCircle(400.0, 300.0, 50.0);
    return 0;
}
```

REAL-WORLD EXAMPLES

- AutoCAD
- Construct3

Architectural Stack In both cases:

- Frontend: React.js + TypeScript
- Backend: Webassembly with C++

OUTLOOK TO THE FUTURE

- Direct access to Web APIs
- Multi-threaded execution with shared memory
- Zero-cost exceptions
- 128-bit SIMD
- 64-bit support
- Stable ABI
- Debugging
  - http://webassemblycode.com/using-browsers-debug-webassembly/