

# Emscripten

## Using Non-JS Libraries in JS

Alon Zakai  
Mozilla

# Non-JS Libraries in JS? Why?

- **Plenty of [existing](#) library code out there**
  - Mainly C and C++
  - Time-consuming to manually port it to JavaScript
  - If you develop a new native app, you can reuse those libraries
- **We should be able to do that on the web too!**

# Demo #1: Bullet

- The Bullet **physics engine** is used in many AAA games
- **150,000 lines of (well-written yet complex) C++**
  - Manually ported to Java: JBullet
  - JBullet manually ported to JavaScript: bullet.js
  - Manual ports are partial and lag behind Bullet
- **ammo.js** is a port of Bullet to JS using Emscripten. Example uses:
  - <http://syntensity.com/static/ammo.html>
  - <http://cjcliffe.github.com/CubicVR.js/cubicvr/samples/physics>
  - <http://jsfiddle.net/strelzoff/6zG3F/>

# Demo #2: SQLite

- SQLite is a popular open source **database**
- Could have become part of the web (WebSQL), but

*"This document was on the W3C Recommendation track but specification work has stopped. The specification reached an impasse: all interested implementors have used the same SQL backend (Sqlite), but we need multiple independent implementations to proceed along a standardisation path."*

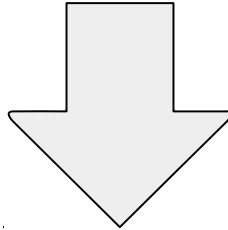
- [http://www.syntensity.com/static/sqlite\\_wip.html](http://www.syntensity.com/static/sqlite_wip.html)

# Demo #3: Python, Ruby, Lua

- Popular **dynamic languages** implemented in C
  - Existing implementations in JavaScript (e.g., pyjamas for Python), but those only target modified, limited subsets of the language
- Compiling the entire **original** implementation to JavaScript gives you the full language. Some example websites:
  - <http://repl.it> (python, ruby, lua)
  - <http://zodb.ws> (python)
  - <http://pythonfiddle.com> (python)

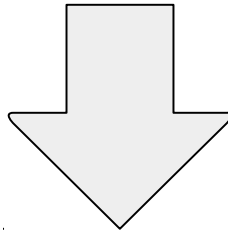
How does this work?

**C/C++ code**



**LLVM**

**LLVM bitcode**



**Emscripten**

**JavaScript**

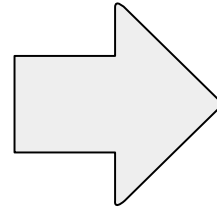
# Emscripten: Overview

- Translate most LLVM bitcode 1 to 1:

```
%0 = alloca i32
```

```
var $0;
```

```
%1 = call @func(i32 %0)
```



```
var $1 = _func($0);
```

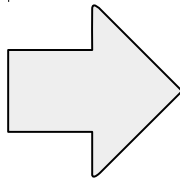
```
%2 = add nsw i32 %1, 22
```

```
var $2 = $1 + 22;
```

# Emscripten: Semantics / Behavior

- The memory space is implemented as a single large array, **HEAP**.
  - **HEAP[x]** is a read, **HEAP[x] = ...** is a write
  - Pointers are simply integer indexes
- Some C/C++ semantics require **additional work** to correct

```
int x = 5;  
x = x / 2;
```



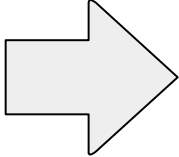
```
var x = 5;  
x = Math.floor(x/2);
```

- Similar issues with overflows, signs, etc.

# Emscripten: Optimization

- **Construct high-level JavaScript loops and ifs** from low-level branching data

```
block1:  
  %1 = call @f(i8 %0);  
  br label %block2  
  
block2:  
  ...  
  ...
```



```
while (...) {  
  var $1 = _f($0);  
  ..  
}
```

- **Implement variables as native JavaScript variables where possible** (`var x`), otherwise they are in the emulated stack (in `HEAP`)

# Emscripten: Runtime Environment

- **Implement C system APIs in JavaScript:** C standard library, POSIX, SDL, etc.
- A simple JavaScript API lets you set up a **virtual filesystem**,

```
FS.createLazyFile('/', 'file.dat',  
                  'http://somewhere.com/file');
```

- Compiled C/C++ code uses the filesystem **normally**,

```
#include<stdio.h>  
[...]  
FILE *f = fopen("file.dat", "rb");  
fread(...)
```

How can you use this?

# Compiling to JavaScript

- Emscripten's **compiler** compiles LLVM into JavaScript
  - The compiler is just concerned with getting the code into JS in a way that **works**
  - The "raw" compiled code isn't easy to use! Name mangling, lifecycle management, etc.
- Emscripten's **bindings generator** parses the header files and generates glue code to connect everything in a **convenient** manner

# Compiling to JavaScript

- **Build the library** (almost normally):

```
CC=emmake.py CXX=emmake.py ./configure  
make
```

- Run the **Emscripten compiler**:

```
python emscripten.py library.ll
```

- Run the **Emscripten bindings generator**:

```
python bindings_generator.py lib header.h
```

- **Combine the outputs** into `library.js`

# Existing Library Code

Say you have some library that looks like this:

```
class DataClass {  
public:  
    DataClass(int x, int y);  
};  
  
class ProcessorClass {  
public:  
    void addData(DataClass *data);  
};
```

# Use on the Web

```
<html>
..
<script src="library.js">
<script>
  var data1 =
    new DataClass(5, 12);
  var data2 =
    new DataClass(100, 99);

  var processor =
    new ProcessorClass();

  processor.addData(data1);
  processor.addData(data2);
</script>
</html>
```

# Use on the Web

```
<html>
..
<script src="library.js">
<script>
  var data1 =
    new DataClass(5, 12);
  var data2 =
    new DataClass(100, 99);

  var processor =
    new ProcessorClass();

  processor.addData(data1);
  processor.addData(data2);
</script>
</html>
```

```
// C++

DataClass *data1 =
  new DataClass(5, 12);
DataClass *data2 =
  new DataClass(100, 99);

ProcessorClass *processor =
  new ProcessorClass();

processor->addData(data1);
processor->addData(data2);
```

# Bindings API

- `var inst = new Class(...);` returns a **wrapper object**, which internally interfaces with the C/C++ data
- Wrapper objects have **wrapper functions**, for example: `inst.doSomething(5, otherInst);` will convert arguments etc.
- Limitations:
  - You must call `destroy(inst);` manually
  - Comparing wrappers of the same underlying object cast to different classes returns false

# Usage: Summary and Caveats

- **Works well on many large [real-world](#) codebases:** Doom, zlib, Poppler, OpenJPEG, FreeType, Bullet, SQLite, Python, Ruby, Lua, etc.
  - Bindings generator is newer and less mature
- To port a C/C++ library to JavaScript, **you will need to know C/C++**
- Generating working code is straightforward, usually without any modifications to the original library code, but **optimizing the code may take some effort**

But isn't this horribly slow?

(Don't we need  
native apps/NaCl/Dart/Java/Flash/Silverlight/Unity?)

# No, This Isn't Slow

Benchmark	SM	SM w/TA
dmalloc	3.39	2.09
fannkuch	5.36	4.95
fasta	2.81	1.98
memops	5.78	4.75
primes	3.52	3.18
raytrace	7.26	8.34

- Numbers are times slower than `gcc -O3`. So **1 = gcc**
- **SM**: SpiderMonkey (Firefox), **TA**: Typed Arrays

# No, This Isn't Slow

- <http://shootout.alioth.debian.org> data:
  - **C++:** 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
  - **Java:** 1, 1, 1, 1, 1, 1, 2, 2, 2, 3, 3
  - **Scala:** 1, 1, 1, 2, 2, 2, 2, 2, 2, 4, 9
  - **Haskell:** 1, 1, 2, 2, 2, 2, 3, 5, 7
  - **C#:** 1, 1, 2, 2, 2, 2, 3, 3, 4, 8, 12
  - ..
  - **JavaScript:** 2, 2, 3, 5, 5, 8 (our data, not shootout)
  - ..
  - **Python:** 1, 3, 7, 37, 42, 46, 61, 71, 84, 110
  - **Ruby:** 4, 7, 8, 21, 27, 50, 76, 81, 110, 216

# No, This Isn't Slow. *Well, Mostly...*

- Speed can **vary** greatly from benchmark to benchmark, and on real-world code
- **Bugs** in JS engines can lead to very bad performance
  - Memory space (HEAP) sometimes not well-optimized
  - Issues with large JavaScript files
- Various useful JS compiler technologies in the works (type inference, SSA optimizations, etc.)
- ...not native speed yet. But getting close!

# The Future

```
module LibC from "https://example.org/libc.js";  
import printf from LibC;  
printf("hello, world\n");
```

...for **any** library in **any** language

**[emscripten.org](https://emscripten.org)**

<https://github.com/kripken/emscripten>

- Feel free to ask for help on IRC or github
- Thank you!