Emscripting a C library to Wasm

By Surma

Surma is a contributor to Web**Fundamentals**

Sometimes you want to use a library that is only available as C or C++ code. Traditionally, this is where you give up. Well, not anymore, because now we have Emscripten and WebAssembly (or Wasm)!

Note: In this article I will describe my journey of compiling libwebp to Wasm. To make use of this article as well as Wasm in general, you will need knowledge of C, especially pointers, memory management and compiler options.

The toolchain

I set myself the goal of working out how to compile some existing C code to Wasm. There's been some noise around LLVM's Wasm backend, so I started digging into that. While you can get simple programs to compile this way, the second you want to use C's standard library or even compile multiple files, you will probably run into problems. This led me to the major lesson I learned:

While Emscripten *used* to be a C-to-asm.js compiler, it has since matured to target Wasm and is in the process of switching to the official LLVM backend internally. Emscripten also provides a Wasm-compatible implementation of C's standard library. **Use Emscripten**. It carries a lot of hidden work, emulates a file system, provides memory management, wraps OpenGL with WebGL — a lot of things that you really don't need to experience developing for yourself.

While that might sound like you have to worry about bloat — I certainly worried — the Emscripten compiler removes everything that's not needed. In my experiments the resulting Wasm modules are appropriately sized for the logic that they contain and the Emscripten and WebAssembly teams are working on making them even smaller in the future.

You can get Emscripten by following the instructions on their website or using Homebrew. If you are a fan of dockerized commands like me and don't want to install things on your system just to have a play with WebAssembly, there is a well-maintained Docker image that you can use instead:

```
$ docker pull trzeci/emscripten
$ docker run --rm -v $(pwd):/src trzeci/emscripten emcc <emcc options here>
```

Compiling something simple

Let's take the almost canonical example of writing a function in C that calculates the nth fibonacci number:

```
#include <emscripten.h>

EMSCRIPTEN_KEEPALIVE
int fib(int n) {
  int i, t, a = 0, b = 1;
  for (i = 0; i < n; i++) {
    t = a + b;
    a = b;
    b = t;
  }
  return b;
}
```

If you know C, the function itself shouldn't be too surprising. Even if you don't know C but know JavaScript, you will hopefully be able to understand what's going on here.

`emscripten.h` is a header file provided by Emscripten. We only need it so we have access to the `EMSCRIPTEN_KEEPALIVE` macro, but it provides much more functionality. This macro tells the compiler to not remove a function even if it appears unused. If we omitted that macro, the compiler would optimize the function away — nobody is using it after all.

Let's save all that in a file called `fib.c`. To turn it into a `.wasm` file we need to turn to Emscripten's compiler command `emcc`:

```
$ emcc -O3 -s WASM=1 -s EXTRA_EXPORTED_RUNTIME_METHODS='["cwrap"]' fib.c
```

Let's dissect this command. `emcc` is Emscripten's compiler. `fib.c` is our C file. So far, so good. `-s WASM=1` tells Emscripten to give us a Wasm file instead of an asm.js file.. `-s EXTRA_EXPORTED_RUNTIME_METHODS='["cwrap"]'` tells the compiler to leave the `cwrap()` function available in the JavaScript file — more on this function later. `-O3` tells the compiler to optimize aggressively. You can choose lower numbers to decrease build time, but that will also make the resulting bundles bigger as the compiler might not remove unused code.

After running the command you should end up with a JavaScript file called `a.out.js` and a WebAssembly file called `a.out.wasm`. The Wasm file (or "module") contains our compiled C code and should be fairly small. The JavaScript file takes care of loading and initializing our Wasm module and providing a nicer API. If needed it will also take care of setting up the stack, the heap and other functionality usually expected to be provided by the operating system when writing C code. As such the JavaScript file is a bit bigger, weighing in at 19KB (~5KB gzip'd).

Running something simple

The easiest way to load and run your module is to use the generated JavaScript file. Once you load that file, you will have a `Module` global at your disposal. Use `cwrap` to create a JavaScript native function that takes care of converting parameters to something C-friendly and invoking the wrapped function. `cwrap` takes the function name, return type and argument types as arguments, in that order:

```
<script src="a.out.js"></script>
<script>
  Module.onRuntimeInitialized = _ => {
    const fib = Module.cwrap('fib', 'number', ['number']);
    console.log(fib(12));
  };
</script>
```

If you run this code, you should see the "233" in the console, which is the 12th Fibonacci number.

Note: Emscripten offers a couple of options to handle loading multiple modules. More about that in their documentation.

The holy grail: Compiling a C library

Up until now, the C code we have written was written with Wasm in mind. A core use-case for WebAssembly, however, is to take the existing ecosystem of C libraries and allow developers to use them on the web. These libraries often rely on C's standard library, an operating system, a file system and other things. Emscripten provides most of these features, although there are some limitations.

Let's go back to my original goal: compiling an encoder for WebP to Wasm. The source for the WebP codec is written in C and available on GitHub as well as some extensive API documentation. That's a pretty good starting point.

```
$ git clone https://github.com/webmproject/libwebp
```

To start off simple, let's try to expose **`WebPGetEncoderVersion()`** from **`encode.h`** to JavaScript by writing a C file called **`webp.c`**:

```
#include "emscripten.h"
#include "src/webp/encode.h"

EMSCRIPTEN_KEEPALIVE
int version() {
  return WebPGetEncoderVersion();
}
```

This is a good simple program to test if we can get the source code of libwebp to compile, as we don't require any parameters or complex data structures to invoke this function.

To compile this program, we need to tell the compiler where it can find libwebp's header files using the **`-I`** flag and also pass it all the C files of libwebp that it needs. I'm going to be honest: I just gave it **all** the C files I could find and relied on the compiler to strip out everything that was unnecessary. It seemed to work brilliantly!

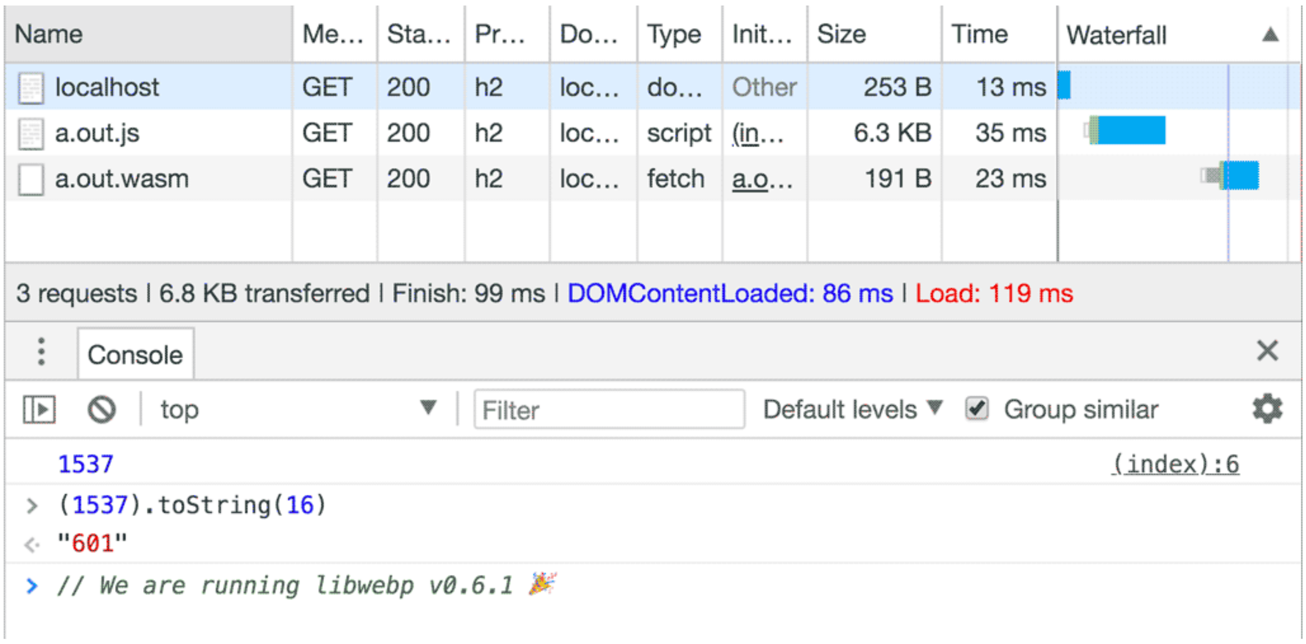
```
$ emcc -O3 -s WASM=1 -s EXTRA_EXPORTED_RUNTIME_METHODS='["cwrap"]' \
    -I libwebp \
    webp.c \
    libwebp/src/{dec,dsp,demux,enc,mux,utils}/*.c
```

Note: This strategy will not work with every C project out there. Many projects rely on autoconf/automake to generate system-specific code before compilation. Emscripten provides **`emconfigure`** and **`emmake`** to wrap these commands and inject the appropriate parameters. You can find more in the Emscripten documentation.

Now we only need some HTML and JavaScript to load our shiny new module:

```
<script src="/a.out.js"></script>
<script>
  Module.onRuntimeInitialized = async _ => {
    const api = {
      version: Module.cwrap('version', 'number', []),
    };
    console.log(api.version());
  };
</script>
```

And we will see the correction version number in the output:



Name	Me…	Sta…	Pr…	Do…	Type	Init…	Size	Time	Waterfall
localhost	GET	200	h2	loc…	do…	Other	253 B	13 ms	
a.out.js	GET	200	h2	loc…	script	(in…	6.3 KB	35 ms	
a.out.wasm	GET	200	h2	loc…	fetch	a.o…	191 B	23 ms	

3 requests | 6.8 KB transferred | Finish: 99 ms | DOMContentLoaded: 86 ms | Load: 119 ms

```
1537                                                    (index):6
> (1537).toString(16)
< "601"
> // We are running libwebp v0.6.1 🎉
```

Note: libwebp returns the current version a.b.c as a hexadecimal number 0xabc. So v0.6.1 is encoded as 0x000601 = 1537.

Get an image from JavaScript into Wasm

Getting the encoder's version number is great and all, but encoding an actual image would be more impressive, right? Let's do that, then.

The first question we have to answer is: How do we get the image into Wasm land? Looking at the encoding API of libwebp, it expects an array of bytes in RGB, RGBA, BGR or BGRA. Luckily, the Canvas API has `getImageData()`, that gives us an Uint8ClampedArray containing the image data in RGBA:

```
async function loadImage(src) {
  // Load image
  const imgBlob = await fetch(src).then(resp => resp.blob());
  const img = await createImageBitmap(imgBlob);
  // Make canvas same size as image
  const canvas = document.createElement('canvas');
  canvas.width = img.width;
  canvas.height = img.height;
  // Draw image onto canvas
  const ctx = canvas.getContext('2d');
  ctx.drawImage(img, 0, 0);
  return ctx.getImageData(0, 0, img.width, img.height);
}
```

Now it's "only" a matter of copying the data from JavaScript land into Wasm land. For that, we need to expose two additional functions. One that allocates memory for the image inside Wasm land and one that frees it up again:

```
EMSCRIPTEN_KEEPALIVE
uint8_t* create_buffer(int width, int height) {
  return malloc(width * height * 4 * sizeof(uint8_t));
}

EMSCRIPTEN_KEEPALIVE
void destroy_buffer(uint8_t* p) {
  free(p);
}
```

`create_buffer` allocates a buffer for the RGBA image — hence 4 bytes per pixel. The pointer returned by `malloc()` is the address of the first memory cell of that buffer. When the pointer is returned to JavaScript land, it is treated as just a number. After exposing the function to JavaScript using `cwrap`, we can use that number to find the start of our buffer and copy the image data.

```
const api = {
  version: Module.cwrap('version', 'number', []),
  create_buffer: Module.cwrap('create_buffer', 'number', ['number', 'number']),
  destroy_buffer: Module.cwrap('destroy_buffer', '', ['number']),
};
```

```
const image = await loadImage('/image.jpg');
const p = api.create_buffer(image.width, image.height);
Module.HEAP8.set(image.data, p);
// ... call encoder ...
api.destroy_buffer(p);
```

Grand Finale: Encode the image

The image is now available in Wasm land. It is time to call the WebP encoder to do its job! Looking at the WebP documentation, `WebPEncodeRGBA` seems like a perfect fit. The function takes a pointer to the input image and its dimensions, as well as a quality option between 0 and 100. It also allocates an output buffer for us, that we need to free using `WebPFree()` once we are done with the WebP image.

The result of the encoding operation is an output buffer and its length. Because functions in C can't have arrays as return types (unless we allocate memory dynamically), I resorted to a static global array. I know, not clean C (in fact, it relies on the fact that Wasm pointers are 32bit wide), but to keep things simple I think this is a fair shortcut.

```
int result[2];
EMSCRIPTEN_KEEPALIVE
void encode(uint8_t* img_in, int width, int height, float quality) {
  uint8_t* img_out;
  size_t size;

  size = WebPEncodeRGBA(img_in, width, height, width * 4, quality, &img_out);

  result[0] = (int)img_out;
  result[1] = size;
}

EMSCRIPTEN_KEEPALIVE
void free_result(uint8_t* result) {
  WebPFree(result);
}

EMSCRIPTEN_KEEPALIVE
int get_result_pointer() {
  return result[0];
}

EMSCRIPTEN_KEEPALIVE
int get_result_size() {
  return result[1];
}
```

Now with all of that in place, we can call the encoding function, grab the pointer and image size, put it in a JavaScript-land buffer of our own, and release all the Wasm-land buffers we have allocated in the process.

```
api.encode(p, image.width, image.height, 100);
const resultPointer = api.get_result_pointer();
const resultSize = api.get_result_size();
const resultView = new Uint8Array(Module.HEAP8.buffer, resultPointer, resultSize)
const result = new Uint8Array(resultView);
api.free_result(resultPointer);
```

Note: `new Uint8Array(someBuffer)` will create a new view onto the same memory chunk, while `new Uint8Array(someTypedArray)` will copy the data.

Depending on the size of your image, you might run into an error where Wasm can't grow the memory enough to accommodate both the input and the output image:

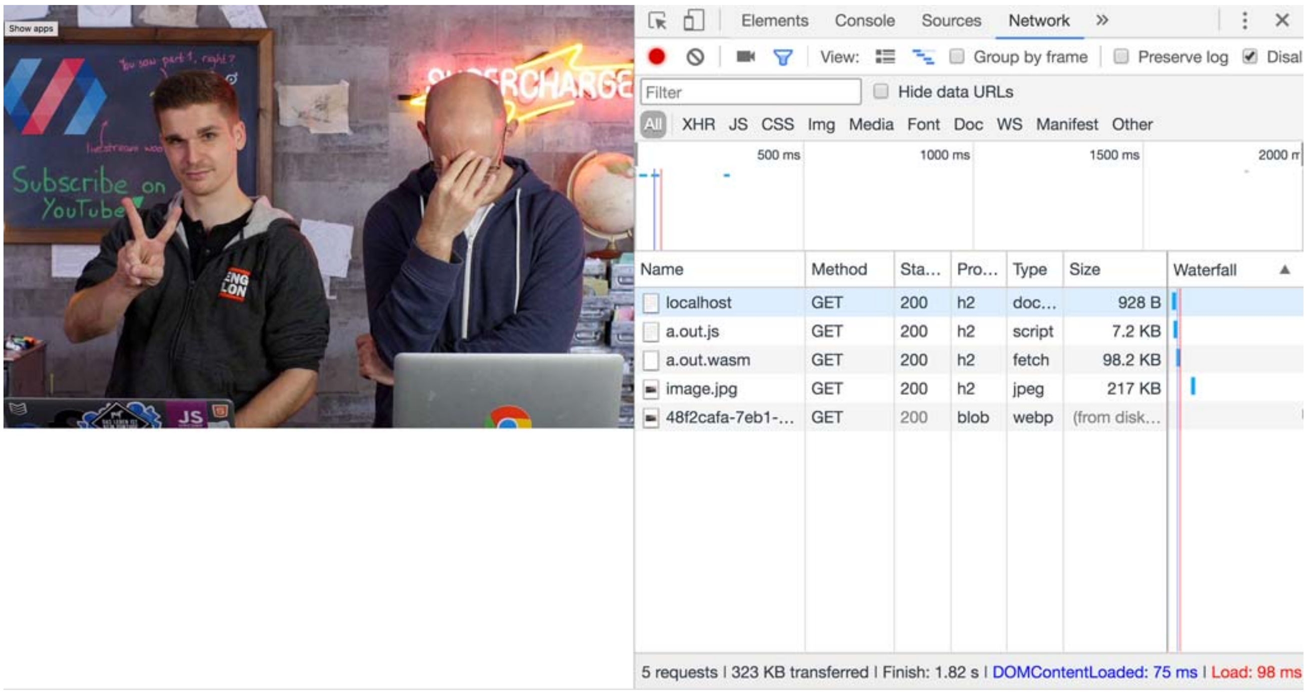
```
⚠ ▼Cannot enlarge memory arrays. Either (1) compile with  -s              a.out.js:1
  TOTAL_MEMORY=X  with X higher than the current value 16777216, (2) compile with  -s
  ALLOW_MEMORY_GROWTH=1  which allows increasing the size at runtime, or (3) if you
  want malloc to return NULL (0) instead of this abort, compile with  -s
  ABORTING_MALLOC=0
```

Luckily, the solution to this problem is in the error message! We just need to add `-s ALLOW_MEMORY_GROWTH=1` to our compilation command.

And there you have it! We compiled a WebP encoder and transcoded a JPEG image to WebP. To prove that it worked, we can turn our result buffer into a blob and use it on an `<img>` element:

```
const blob = new Blob([result], {type: 'image/webp'});
const blobURL = URL.createObjectURL(blob);
const img = document.createElement('img');
img.src = blobURL;
document.body.appendChild(img)
```

Behold, the glory of a new WebP image!


Subscribe on YouTube
SUPERCHARGE
Elements Console Sources Network
View: Group by frame Preserve log Disal
Filter Hide data URLs
All XHR JS CSS Img Media Font Doc WS Manifest Other
500 ms 1000 ms 1500 ms 2000 m

Name	Method	Sta...	Pro...	Type	Size	Waterfall
localhost	GET	200	h2	doc...	928 B	
a.out.js	GET	200	h2	script	7.2 KB	
a.out.wasm	GET	200	h2	fetch	98.2 KB	
image.jpg	GET	200	h2	jpeg	217 KB	
48f2cafa-7eb1-...	GET	200	blob	webp	(from disk...	

5 requests | 323 KB transferred | Finish: 1.82 s | DOMContentLoaded: 75 ms | Load: 98 ms

Conclusion

It's not a walk in the park to get a C library to work in the browser, but once you understand the overall process and how the data flow works, it becomes easier and the results can be mind-blowing.

WebAssembly opens many new possibilities on the web for processing, number crunching and gaming. Keep in mind that Wasm is not a silver bullet that should be applied to everything, but when you hit one of those bottlenecks, Wasm can be an incredibly helpful tool.

Bonus content: Running something simple the hard way

If you want to try and avoid the generated JavaScript file, you might be able to. Let's go back to the Fibonacci example. To load and run it ourselves, we can do the following:

```
<!doctype html>
<script>
  (async function() {
    const imports = {
      env: {
        memory: new WebAssembly.Memory({initial: 1}),
        STACKTOP: 0,
      }
    };
    const {instance} = await WebAssembly.instantiateStreaming(fetch('/a.out.wasm'
    console.log(instance.exports._fib(12));
  })();
</script>
```

Note: Make sure that your `.wasm` files have `Content-Type: application/wasm`. Otherwise they will be rejected by WebAssembly.

WebAssembly modules that have been created by Emscripten have no memory to work with unless you provide them with memory. The way you provide a Wasm module with *anything* is by using the `imports` object — the second parameter of the `instantiateStreaming` function. The Wasm module can access everything inside the imports object, but nothing else outside of it. By convention, modules compiled by Emscripting expect a couple of things from the loading JavaScript environment:

- Firstly, there is `env.memory`. The Wasm module is unaware of the outside world so to speak, so it needs to get some memory to work with. Enter `WebAssembly.Memory`. It represents a (optionally growable) piece of linear memory. The sizing parameters are in "in units of WebAssembly pages", meaning the code above allocates 1 page of memory, with each page having a size of 64 KiB. Without providing a `maximum` option, the memory is theoretically unbounded in growth (Chrome currently has a hard limit of 2GB). Most WebAssembly modules shouldn't need to set a maximum.
- `env.STACKTOP` defines where the stack is supposed to start growing. The stack is needed to make function calls and to allocate memory for local variables. Since we don't do any dynamic memory management shenanigans in our little Fibonacci program, we can just use the entire memory as a stack, hence `STACKTOP = 0`.

Last updated July 2, 2018.