Level Up with WebAssembly

Chapter 1. Introduction

Chapter 2. Setup + Hello World

Chapter 3. Wait but why?

Chapter 4. WebAssembly in the browser

Chapter 5. The Module Object

Chapter 6. File Management

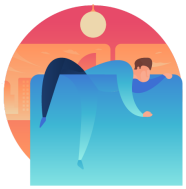
Chapter 7. Compiling Existing Tools

Chapter 8. Graphics

Chapter 9. Persisting the File System

Chapter 10. WebAssembly + WebWorkers

Chapter 11. Conclusion



Appendix. Troubleshooting

Chapter 1. Introduction

What is WebAssembly?

WebAssembly (a.k.a. WASM) is a programming language meant to be run in the browser, alongside JavaScript. WebAssembly is a fairly recent language: the MVP was released in 2017, with support for all major browsers including Firefox, Chrome, Safari, and Edge. According to caniuse.com, it's estimated that over 80% of users are running browsers that support WebAssembly.

But unlike most programming languages, this one is—for the most part—not meant to be a language you develop in directly! Instead, it acts as a compilation target for C/C++/Rust scripts (support for more languages in on the roadmap). What this means is that you can take a C++ codebase, compile it to WebAssembly, and run it in the browser at "near-native" speeds (i.e. almost as fast as compiling the C program to binary and running it on the command-line).

If this sounds similar to Java Applets or Flash applications, it's far from it. WebAssembly is not a plugin; it's a feature of the web, developed with the involvement of all major browser vendors. As a result, WebAssembly integrates with the web much more seamlessly than Flash or Java ever did. And because it's meant to be used as a compilation target from existing languages like C, C++ and Rust, it is in that sense more language-agnostic.

Who uses WebAssembly?

So far, WebAssembly has been used for three major applications:

1. Gaming on the web
2. Graphics-based applications
3. Data analysis tools

Here are a few examples:

- **Unity**'s game engine now supports WebAssembly as a build target; Here's a sample game.
- **AutoCAD** ported their 30-year-old code base into WebAssembly to enable its use on the web.
- **Figma,** a prototyping tool for designers, uses WebAssembly to improve their load time by 3X.
- **uBlock**, an ad-blocker browser plugin, is starting to use WebAssembly to improve performance.
- **Pyodide**, a port of commonly-used scientific packages from Python to WebAssembly!
- **Blazor,** a framework to port .NET web applications to WebAssembly.
- **fastq.bio**, a data analysis tool I developed that uses WebAssembly to speed up calculations.

See this curated list for more WebAssembly applications, demos and resources.

Hello World example

Let's dig right in, and look at a simple WebAssembly script. We'll start with a C program (`hello.c`) that outputs Hello World:

```c
#include <stdio.h>

int main() {
  printf("Hello World!\n");
  return 0;
}
```

To compile this C program to binary, we would normally do the following:

```
$ gcc hello.c -o hello
```

Instead, we'll use the Emscripten tool `emcc` to compile our C program into WebAssembly. In Chapter 2, we'll cover how to set up your environment for `emcc`, but for now, here's a preview of what the command would look like:

```
$ emcc hello.c -o hello.js
```

As you can see it's very similar to the `gcc` one, which is very convenient!

Running this command will create two files:

- `hello.wasm`, which contains the compiled, binary assembly code
- `hello.js`, which contains auto-generated JavaScript glue code

Now it's your turn! In the next chapter, we'll discuss how to set up your environment to compile and run your own WebAssembly scripts. But if you're curious about what WebAssembly looks like under the hood, check out the next section of this chapter before heading over to Chapter 2.

Under the hood (advanced)

You may be wondering what the WebAssembly code actually looks like when compiled. Let's dive in.

When you compile the Hello World program above to WebAssembly, you can represent it as WebAssembly Text format (or WAT 😬), which looks something like this:

```
(module
  (type $FUNCSIG$ii (func (param i32) (result i32)))
  (import "env" "puts" (func $puts (param i32) (result i32)))
  (table 0 anyfunc)
  (memory $0 1)
  (data (i32.const 16) "Hello World!\00")
  (export "memory" (memory $0))
  (export "main" (func $main))
  (func $main (; 1 ;) (result i32)
    (drop
      (call $puts
         (i32.const 16)
      )
    )
    (i32.const 0)
  )
)
```

The format above uses *S-expressions*, i.e. parentheses to represent the structure of a tree, where the leafs of the tree are the parentheses that contain no other parentheses. We won't be editing the WAT format in this book, but for a deep dive into the syntax, see this in-depth MDN article.

If we take the WAT code above and convert it to assembly (and binary in grey), we get the following:

```
wasm-function[1]:
  sub rsp, 0x18                          ; 0x000000 48 83 ec 18
  cmp qword ptr [r14 + 0x28], rsp        ; 0x000004 49 39 66 28
  jae 0x56                               ; 0x000008 0f 83 48 00 00 00
 0x00000e:
  mov edi, 0x10                          ; 0x00000e bf 10 00 00 00
  mov qword ptr [rsp], r14               ; 0x000013 4c 89 34 24
  mov rax, qword ptr [r14 + 0x30]        ; 0x000017 49 8b 46 30
  mov r14, qword ptr [r14 + 0x38]        ; 0x00001b 4d 8b 76 38
  mov r15, qword ptr [r14 + 0x18]        ; 0x00001f 4d 8b 7e 18
  call rax                               ; 0x000023 ff d0
  mov r14, qword ptr [rsp]               ; 0x000025 4c 8b 34 24
  mov r15, qword ptr [r14 + 0x18]        ; 0x000029 4d 8b 7e 18
  xor eax, eax                           ; 0x00002d 33 c0
  nop                                    ; 0x00002f 66 90
  add rsp, 0x18                          ; 0x000031 48 83 c4 18
  ret                                    ; 0x000035 c3

wasm-function[0]:
  sub rsp, 0x18                          ; 0x000000 48 83 ec 18
  mov qword ptr [rsp], r14               ; 0x000004 4c 89 34 24
  mov rax, qword ptr [r14 + 0x30]        ; 0x000008 49 8b 46 30
  mov r14, qword ptr [r14 + 0x38]        ; 0x00000c 4d 8b 76 38
  mov r15, qword ptr [r14 + 0x18]        ; 0x000010 4d 8b 7e 18
  call rax                               ; 0x000014 ff d0
  mov r14, qword ptr [rsp]               ; 0x000016 4c 8b 34 24
  mov r15, qword ptr [r14 + 0x18]        ; 0x00001a 4d 8b 7e 18
  nop                                    ; 0x00001e 66 90
  add rsp, 0x18                          ; 0x000020 48 83 c4 18
  ret                                    ; 0x000024 c3
```

If you've done low-level assembly coding before, this will look familiar. If not, that's ok, we won't be writing code directly in WebAssembly. But if you're curious about how your own C functions get compiled to WebAssembly, check out WasmExplorer and WebAssembly Studio.

It's nice to know what WebAssembly looks like when the browser is interpreting it. It's essentially low-level machine code that a virtual machine can execute (see this blog post and this one too for details). And that's why WebAssembly has the potential to be so fast: unlike JavaScript, the browser can

parse a compact binary format, all variables are typed so no guesswork is required, and we don't need to optimize the code on the fly.

And the fact that it's low level is clear because it takes ~30 lines of code just to output a string to the screen! Also notice that the commands are mostly basic math and control flow operations: e.g. `add` (adding two numbers), `sub` (subtracting two numbers), `mov` (move data from one location to another), `cmp` (calculates the difference of two numbers), and `jae` (jump to location if the result of cmp was ≥ 0).

If you're interested in going deeper into the assembly instructions of the WebAssembly language (which, by the way, makes for great bedtime reading), see the W3C WebAssembly Spec.

Chapter 2. Setup + Hello World

Throughout this book we'll use Emscripten, a tool that builds on the LLVM compiler, and allows you to compile C/C++ programs into WebAssembly. As we'll see in future chapters, it provides many useful tools and features that makes developing in WebAssembly more accessible.

Despite being an awesome tool, Emscripten also takes a while to install and has many dependencies, which can make it a challenging process. So let's avoid that rabbit hole and take a shortcut: Instead of building it from scratch, I've prepared a Docker image that comes pre-packaged with Emscripten.

Install Docker Desktop (skip if you already have Docker Desktop)

But first, you'll need Docker Desktop in order to fetch and run a pre-built Emscripten image. You can download Docker from here. Once downloaded, follow the installation instructions

Launch Docker Desktop (skip if Docker Desktop is running)

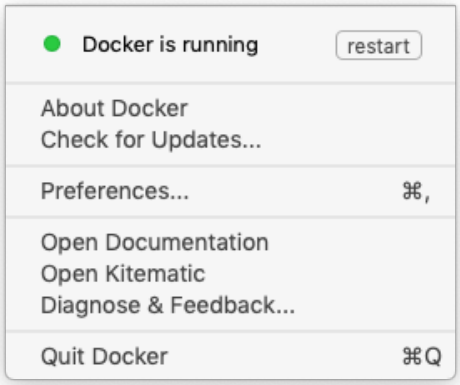
Once the installation is complete, launch Docker Desktop and wait a few minutes for it to initialize. To confirm it's ready, type `docker ps` on the command line and you should see something like this:

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED     ...
```

If it's not ready yet, you'll instead see this error:

```
$ docker ps
Cannot connect to the Docker daemon. Is the docker daemon running on this host?
```

If so, this means Docker Desktop is not running; you may need to wait a few more minutes. Alternatively, click the whale icon in your Mac OS menu bar / Windows menu, and you should see the message "Docker is running" once it's ready:



Setup Emscripten container

Now that Docker is ready, we'll fetch the Docker image that contains a pre-built version of Emscripten, along with all the dependencies we'll need throughout the book (the Dockerfile that defines this image is available on GitHub if you're curious):

```
$ docker pull robertaboukhalil/emsdk:1.38.26
```

Next, let's create a folder where we'll do all our WebAssembly work:

```
$ mkdir ~/wasm
$ cd ~/wasm
```

We'll now create a container from the image, and mount the `~/wasm` folder that's on your computer, to the folder `/src` inside the container:

```
$ docker run \                      # Create and run a container
    -d \                            # -d = Run container in the background (detach)
    -t \                            # -t = Stick around to accept input (tty)
    -p 12345:80 \                   # Expose port 80 within the container as 12345
    --name wasm \                   # Give container a name to reference it with
    --volume "$(pwd)":/src \        # Mount current dir ~/wasm to /src inside
    robertaboukhalil/emsdk          container
                                    # Create container from image we pulled above
```

This `docker run` command is a mouthful, but you should only have to do this once.

If creating the container worked successfully, Docker outputs a long alphanumeric string, which is the container ID (the `--name` parameter above allows us to refer to this container as "`wasm`" instead of that long string!).

Note

1) `--volume`: The volume mount allows us to: **(1)** Develop scripts locally on `~/wasm`; **(2)** Compile the scripts inside the container; and **(3)** Run the compiled WebAssembly in a browser outside the container, all without having to transfer files back and forth with the container.
2) `-p 12345:80`: The container contains a web server running to allow us to access the HTML files from outside the container. We're telling docker to allow us to access that server by accessing port 12345 from outside the container.

Warning

If you are getting permission errors with regards to mounting ~/wasm, open Docker Desktop → Preferences → File Sharing tab → Add ~/wasm using the + button

To start using Emscripten, let's go inside the container using `docker exec`:

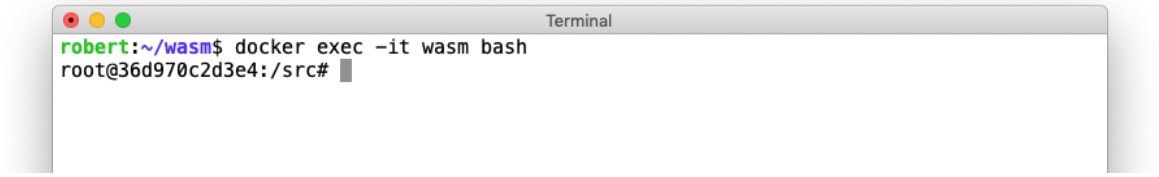
```
$ docker exec \                     # Run command inside container
    -i \                            # Enable interactive mode
    -t \                            # Same as docker run; enables terminal input
    wasm \                          # The container we want to run a command in
    bash                            # The command to run is just bash
```

Or for short:

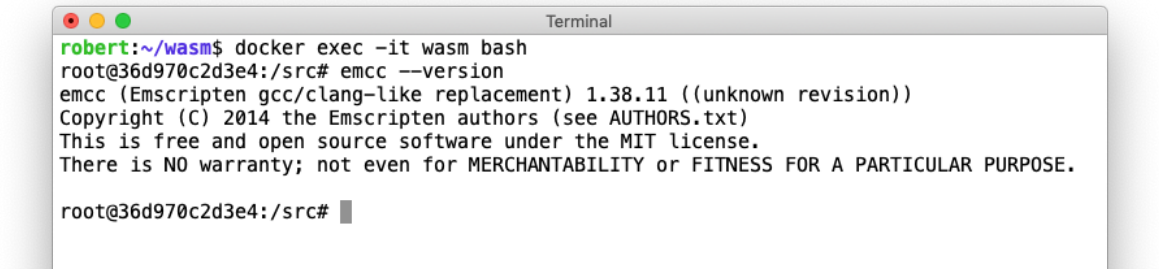
```
$ docker exec -it wasm bash
```

> **Note**
>
> Although you only need to create the container once with `docker run`, you'll need to run `docker exec` everytime you need to enter the container (e.g. when you close your terminal or restart your computer). It might be worth editing your `~/.bash_profile` or `~/.bashrc` to add a shortcut: `alias wasm="docker exec -it wasm bash"`, which allows you to enter the container just by typing "wasm" on the command line!

If that was successful, your command line will look something like this:



We're now inside the wasm container! Your container ID will be different, but you should still find yourself in the `/src` folder. Let's verify that everything worked by retrieving the Emscripten version:



Your version will be different, but you're good to go if you can run that command without errors.

Run the Hello World example

Create the file `~/wasm/hello.c`, and paste our Hello World example (do this in your favorite IDE outside the container):

```c
#include <stdio.h>

int main() {
  printf("Hello World!\n");
  return 0;
}
```

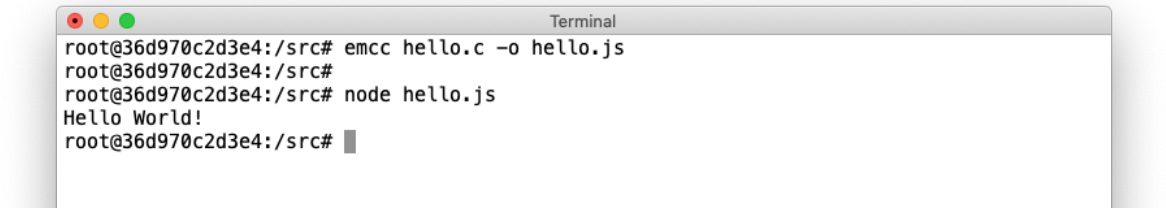
Next, let's compile the script (do this inside the container):

```
$ emcc hello.c -o hello.js
```

This should create `hello.js` and `hello.wasm`:

```
$ ls
hello.c     hello.js    hello.wasm
```

And let's execute that script with Node (we'll cover running WebAssembly from the browser next)—do that inside the container:



And voila!

Chapter 3. Wait but why?

Perhaps the most overlooked question when evaluating new software paradigms like WebAssembly isn't how it works or why it's awesome, but rather when we should not use it. So let's take a step back before we go any further, and discuss why WebAssembly is commonly used, when it's a good idea to use it, and when it should be avoided.

Why Use WebAssembly?

In general, WebAssembly helps in two ways:

1. **Speed**: Speed up execution in the browser
2. **Portability**: Re-use existing C/C++/Rust tools in the browser

When talking about WebAssembly, most of the focus is on the speedup aspect, which is an obvious advantage, in particular for image and data analysis. But **in some cases, the portability advantage is almost more important than speeding up the runtime**. An example of that is gaming and desktop applications, where the ability to port a large, existing codebase to the browser is highly appealing.

Another example of that is genomics, where some of the most popular packages for DNA sequencing analysis are written in C/C++, so WebAssembly allows us to avoid re-writing complex algorithms in

JavaScript, reducing the possibility for errors, and making the lives of genomics web tool builders easier. As we'll see in Chapter 10, these two advantages, coupled with using WebWorkers to ensure the browser UI remains responsive, has the potential for a highly improved user experience.

Also, keep in mind that although large desktop applications/games have been ported to the web, you don't need to revamp your entire application to make use of WebAssembly; you can also target a small portion of your app that would benefit from performance optimization.

When to use WebAssembly

As a good rule of thumb, WebAssembly is useful when building data- or graphics-heavy web applications that fit into one of the following categories:

1) Graphical software such as games or desktop applications
2) Data analysis in the browser
3) Educational tools and tutorials
4) File preprocessing

On the other hand, if you're building Uber for parrots 🦜, WebAssembly might not be the best tool for you—unless of course you intend to monetize by mining bitcoins on your user's phones (please don't).

Graphical Software

Possibly the most popular use case for WebAssembly is the ability to port existing desktop applications and games to run them in the browser. Being able to run games on the web without starting from scratch, and knowing that those games could be highly responsive, is extremely promising.

In Chapter 8, we'll port over a Pacman game to the browser to illustrate WebAssembly's potential, but also the challenges you may face along the way.

Data Analysis Tools

WebAssembly can also enable data-heavy applications on the web, including image processing (e.g. for photo and video editing), and data analysis/wrangling (e.g. algorithms, simulations).

In most such data-centric web applications, there are two key portions to the app: **data analysis**—where the compute-heavy portion of the app happens—and **data exploration**—where you can interactively explore your input and output data. Traditionally, we've mapped those two components by doing the data analysis portion on the server (back-end), and run the interactive visualizations in the browser (front-end).

In some cases, however, we find ourselves needing to do some data analysis in the browser to provide the user with an improved user experience, especially if doing an analysis in the browser will increase responsiveness and can reduce the number of round trips to the server. You can still use JavaScript for everything, but there are several issues with that:

- You usually need to rewrite an existing server tool into JavaScript **(not fun)**
- That JavaScript code is generally slower than the original tool **(not fast)**
- Running compute-intensive JavaScript in the browser disrupts the user experience **(not good)**

For example, if you need to run some data analysis algorithm on the front-end, compiling existing C/C++/Rust tools could significantly speed up not only page load time, but also development time! Or if your application allows your users to perform custom JSON querying, you could compile the tool `jq` to WebAssembly and run it in the browser (which we do in Chapter 7!).

File Preprocessing

This category is really an extension of Data Analysis Tools. Some application require users to provide large files as input, but where they might only use a fraction of the information on the backend. Instead of uploading the large file to a server and processing it there, it's worthwhile parsing the file locally, extract the information needed, and upload just that (you could even gzip the contents before uploading it!). This is even more useful if the file has a binary format for which you have C/C++ code for parsing.

Finally, there's the "fail fast, fail often" upload use case, where, if you ask the user to upload large data files that will be analyzed on the back-end, you can validate that the file is of the expected format in the browser, while the upload is in progress. This is especially useful if there are existing command-line tools for performing this check, which would allow you to more easily perform this on the front-end. Even better is if you run some quality control checks on the file as it's being uploaded, and notify the user of potential data quality issues so that they can decide whether to continue with the upload or not.

Education

One underexplored category of WebAssembly applications is education. Consider, for example, a UNIX command line tutorial where we compile the command line tools to WebAssembly, and run them in the browser alongside a virtual file system (see Chapter 6). This obviates the need for simulating a command line in JavaScript, with restrictions on inputs and outputs. And it's also more secure, removing the need to send requests to the backend, evaluating them, and returning the output.

An extension of this is using WebAssembly to provide trial version of desktop software, where users don't need to summon the activation energy to download and install a desktop application; they just do so in the browser.

Note

The common theme here is this: beyond porting an entire application to the web, **we can use WebAssembly to leverage the user's browser to perform computations, in order to improve the app's responsiveness and/or speed**.

When to avoid WebAssembly

Conversely, let's discuss when WebAssembly is not the right solution. Here are a few examples:

Need large reference files loaded in memory

Although it is possible to compile to WebAssembly and pre-load files in memory (Chapter 9), if those files are large, the user experience will suffer. Also, keep in mind that the current MVP version of WebAssembly only supports 32-bit addresses, so you can "only" load 4 GB of data into memory. The 64-bit version is coming soon, but even then, if a web app requires anywhere near 4 GB of RAM, I would venture a guess that maybe your app should not be running in a browser.

Analyzing large files in the front-end

For example, if you're running a clustering algorithm that requires you to first calculate a pairwise distance matrix of all rows, WebAssembly is not a good tool for that unless the file is small, since building a pairwise matrix can be very slow since you to see all the data in order to build it.

Require lot of communication between JS & WASM

If there's a lot of communication back and forth between the main thread and your WebAssembly module, this will most likely render your application slow, and risk not providing you any speedup (and perhaps slowing down your application!). Examples of communication includes having the WebAssembly module output large amounts of data to stdout, or if the logic is heavily split between the WebAssembly module and the JavaScript code.

Note

Although undeniably awesome, WebAssembly is just another tool in your toolbox. Use it wisely.

Chapter 4. WebAssembly in the browser

So far, we've talked about how to compile C into WebAssembly and we verified that it worked by running in on the command line with Node.js. In this chapter, we'll introduce how to run our compiled WebAssembly tools in the browser.

Hello World in the browser

Let's revisit our Hello World program from Chapter 2:

```
#include <stdio.h>

int main() {
  printf("Hello World!\n");
  return 0;
}
```

And we talked about how we can compile it to WebAssembly by doing:

```
$ emcc hello.c -o hello.js
```

Running it in the browser is surprisingly simple; we just create `hello.html` and include `hello.js`:

```
<script src="hello.js"></script>
```

If you open http://localhost:12345/hello.html in your browser, you'll see "Hello World!" printed in the developer console! (Note that the port will be different if you didn't use the same port number as we did in Chapter 2)

> **Note**
>
> You can open the developer console in Firefox or Chrome by typing `Cmd + Option + I` on Mac or `Ctrl + Shift + I` on Windows.

Call main() on demand

This was very easy because, by default, Emscripten executes the `main()` function once the WebAssembly module is ready. But what if we wanted to only run `main()` in response to a user event? In those cases, we want to disable running `main()` at page load. To do so, we disable the `INVOKE_RUN` flag:

```
$ emcc hello.c \
    -o hello.js \
    -s INVOKE_RUN=0
```

Now when you open `hello.html`, you won't see Hello World on the console. The easiest way to call `main()` is to use a convenience function provided by Emscripten—type the following in the developer console (don't type the `>`):

```
> Module.callMain()
Hello World!
```

And that's it!

Call main() with function arguments

Next, let's add support for function arguments. We can modify our `main()` function to take a name and greeting as arguments, and output a custom greeting. Let's call this script `hello_custom.c`:

```
#include <stdio.h>

int main(int argc, char **argv) {
  printf("%s %s!\n", argv[1], argv[2]);
  return 0;
}
```

Compile it again:

```
$ emcc hello_custom.c \
    -o hello_custom.js \
    -s INVOKE_RUN=0
```

Now our HTML file looks like:

```
<script src="hello_custom.js"></script>
```

And we can call `main()` with two arguments:

```
> Module.callMain(["Good morning", "Robert"])
Good morning Robert!
```

It's that simple! Of course, this only works for calling `main()`, but in the next section we'll discuss a more general approach that will allow us to export and call any C function of interest.

Calling custom C functions from JavaScript

By default, Emscripten will make the `main()` function available for calling from JavaScript by using `Module.callMain()` as we saw above. However, what if we wanted to expose a function that isn't `main()`?

Approach 1: Expose functions at compilation time

Let's consider the following simple C program, `increment.c`, where the function `increment()` outputs an incremented number:

```
#include <stdio.h>

int increment(int num) {
  num++;
  printf("%d\n", num);
  return 0;
}
```

To make `increment()` accessible from JavaScript, here's how we compile our script:

```
$ emcc increment.c \
    -o increment.js \
    -s INVOKE_RUN=0 \
    -s EXPORTED_FUNCTIONS="['_increment']" \
    -s EXTRA_EXPORTED_RUNTIME_METHODS="['cwrap']"
```

Let's explain what those two new flags do.

We use the `EXPORTED_FUNCTIONS` flag to export functions of interest (in this case, `increment()`). Note the underscore in front of the function name—this is because the LLVM linker prepends underscores to function names during compilation (details here).

We then use the `EXTRA_EXPORTED_RUNTIME_METHODS` flag to also include the `cwrap()` function, which will allow us to call the exported C function directly from JavaScript (more details about that in a second), with no underscores needed for that flag.

To now make `increment()` available to call from JavaScript we use the `cwrap()` function by defining our `increment.html` as:

```
<script type="text/javascript">
// The Module object helps us define options related to the execution of the
// WebAssembly code. We'll cover it in more depth in Chapter 5.
Module = {};
```

```
// Define the function to be called when the WebAssembly module is ready to use
Module.onRuntimeInitialized = function()
{
  // The cwrap() function allows us to wrap an exported C function so that it
  // can be called directly from JavaScript
  var increment = Module.cwrap(
    'increment',  // the C function we want to call from JavaScript
    'number',     // its return type (accepted: number, string or array)
    ['number']    // its list of arguments (a number in our case)
  );

  // Call the function
  console.log(increment(41));
};
</script>
<script src="increment.js"></script>
```

If you open `increment.html` in your browser, you should see an output of 42 in the developer console.

Note

If we hadn't wrapped our code inside the `Module.onRuntimeInitialized` function, we would have gotten an error in the console that looks something like `"you need to wait for the runtime to be ready"`. You can think of `Module.onRuntimeInitialized` as the WebAssembly equivalent of `document.addEventListener("DOMContentLoaded")` or jQuery's `$(document).ready()`.

Warning

Although you can call C functions directly from JavaScript by calling `_functionName()`, e.g. `_increment(41)`, this won't work for non-numeric arguments such as strings or array, so it's usually best to use `cwrap()`.

Note that in our example C code, we didn't have a `main()` function; But if we did, we could have also exported it by adding main() to the list of functions to export at compilation time:

```
emcc increment.c \
    -o increment.js \
    -s INVOKE_RUN=0 \
    -s EXPORTED_FUNCTIONS="['_main', '_increment']" \
    -s EXTRA_EXPORTED_RUNTIME_METHODS="['cwrap']"
```

Approach 2: Expose functions within the C code

The approach we discussed above is the best approach to take if you don't want to modify the original C code. Alternatively, Emscripten allows you to specify which functions to export directly in the C code by adding the flag `EMSCRIPTEN_KEEPALIVE` before a function declaration (save this as `increment2.c`):

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

EMSCRIPTEN_KEEPALIVE int increment(int num) {
  num++;
  printf("%d\n", num);
  return 0;
}
```

Also note that we included `emscripten.h`.

Now when you compile your code, you don't need to specify which functions to expose (but we still need `cwrap`!):

```
emcc increment2.c \
    -o increment2.js \
    -s INVOKE_RUN=0 \
    -s EXTRA_EXPORTED_RUNTIME_METHODS="['cwrap']"
```

And this is what our `increment2.html` file now looks like:

```
<script type="text/javascript">
// The Module object helps us define options related to the execution of the
// WebAssembly code. We'll cover it in more depth in Chapter 5.
Module = {};

// Define the function to be called when the WebAssembly module is ready to use
Module.onRuntimeInitialized = function()
{
  // As we've seen before, we use cwrap() to export the function
  var increment = Module.cwrap(
    'increment',  // the C function we want to call from JavaScript
```

```
    'number',     // its return type (accepted: number, string or array)
    ['number']    // its list of arguments (a number in our case)
  );

  // Call the function
  console.log(increment(41));
};
</script>
<script src="increment2.js"></script>
```

Which approach to use?

Which approach to use mostly depends on your needs. I tend to lean towards the first one if I can help it, because I prefer to keep the C code as free of JavaScript-specific constructs, and as close to its original form as possible. This is especially important if you're compiling a third-party library that you intend to re-compile to WebAssembly on a regular basis as updates are made available.

Calling JavaScript functions from C

Emscripten even supports executing JavaScript code from within your C code:

```
emscripten_run_script('console.log("test")');
```

The string we pass to that function will be executed in the browser with `eval()`. You can refer to the Emscripten documentation for alternative approaches. That said, in most cases, you should minimize calls to JavaScript within the C code if you can help it. In my view, the benefits of using WebAssembly are maximized if you treat the WebAssembly code you run as standalone, isolated functions, instead of a mash of JavaScript and C, which can easily become hard to reason about.

A note about C++

So far, we've focused on compiling C code to WebAssembly. One caveat of exposing functions from C++ code is something known as "name mangling". You can read all about it here, but the gist of it is that C++ function names get overwritten by the compiler to unique identifiers.

Let's look at a simple example and why that's an issue (`increment3.cpp`):

```
#include <iostream>

int increment(int num) {
```

```
  num++;
  std::cout << num << std::endl;
  return 0;
}
```

If we do our usual:

```
em++ increment3.cpp \
    -o increment3.js \
    -s INVOKE_RUN=0 \
    -s EXPORTED_FUNCTIONS="['_increment']" \
    -s EXTRA_EXPORTED_RUNTIME_METHODS="['cwrap']"
```

Note

Note that we've used emcc as proxy to gcc to compile C code, and em++ as proxy to g++, to compile C++ code.

We'll get a note from Emscripten warning us that it couldn't find the function _increment:

```
WARNING:root:function requested to be exported, but not implemented:
"_increment"
```

Note

C "mangles" function names by prepending them with an underscore, which is why we've been adding underscores in front of function names.

To resolve this issue, we need to add `extern "C"` in front of functions that we anticipate will need to be called from JavaScript:

```
#include <iostream>

extern "C"
int increment(int num) {
  num++;
  std::cout << num << std::endl;
  return 0;
}
```

Let's compile again, and this time, we shouldn't see any warnings:

```
em++ increment3.cpp \
    -o increment3.js \
    -s INVOKE_RUN=0 \
    -s EXPORTED_FUNCTIONS="['_increment']" \
    -s EXTRA_EXPORTED_RUNTIME_METHODS="['cwrap']"
```

We can now use the same HTML file as previously, but loading `increment3.js` (using the `Module._increment()` shortcut instead of `cwrap()` for brevity):

```
<script type="text/javascript">
// The Module object helps us define options related to the execution of the
// WebAssembly code. We'll cover it in more depth in Chapter 5.
Module = {};

// Define the function to be called when the WebAssembly module is ready to use
Module.onRuntimeInitialized = () => console.log(Module._increment(41));
</script>
<script src="increment3.js"></script>
```

Note

So far, we've initialized our WebAssembly code with the `.js` glue code that Emscripten generated for us. If you wanted to do the same without it, you *could* fetch the `.wasm` binary and feed it to `WebAssembly.instantiateStreaming()`, which will instantiate your WebAssembly module. If you look at the `.js` glue code from any of the examples in this chapter, you'll see that function being called.

So under the scenes, Emscripten does the same, but it also automates a lot more for us, including helping us export C functions, converting variables in JavaScript to a format understandable by our WebAssembly code, providing convenience functions like `callMain()`, handling memory management, etc.

In this book, we'll stick to using Emscripten since it's much more convenient, but if you see references to functions such as `WebAssembly.instantiateStreaming()` in other resources, that's what they are used for.

Calling main() with cwrap() (advanced)

Recall our custom greeting script's `main()` function:

```
#include <stdio.h>

int main(int argc, char **argv) {
  printf("%s %s!\n", argv[1], argv[2]);
  return 0;
}
```

You'll notice that exporting `main()` as:

```
> var main = Module.cwrap('main', 'number', ['number', 'array']);
> // argc = 3; argv = ['hello', 'Good morning', 'Robert']
> main(3, ['Good morning', 'Robert']);
(null) (null)!
```

doesn't quite work. This is because our array argument is actually an array of strings (i.e. an array of arrays). This can be quite the head scratcher, but if you're ever curious about how something in WebAssembly works, it's worth checking out the code generated by Emscripten.

Let's look at the `callMain()` function that Emscripten provides us by opening the `hello.js` file, and searching for the function definition of `Module['callMain']`.

I've simplified and annotated that function below for clarity:

```
Module['callMain'] = function callMain(args = [])
{
  // Since the first argument is the program name, increment by one
  var argc = args.length + 1;

  // Allocate 4 bytes of stack space per argument
  var argv = stackAlloc((argc + 1) * 4);

  // Store first argument, which is the script name (default = ./this.program)
  // A few notes:
  // (1) The arrays HEAP32 and HEAP8 are different views on the same heap data:
  //     HEAP32 = each element is a 32-bit number (i.e. 4 bytes)
  //     HEAP8  = each element is an 8-bit number (i.e. 1 byte)
```

```
  //     ==> HEAP8 is a larger array since each element takes up less space
  //     ==> i.e. HEAP8.length == HEAP32.length * 4
  //
  // (2) The function allocateUTF8OnStack() stores each character of the given
  //     string as a position within HEAP8
  //
  // (3) The binary arithmetic expression  x >> n  is  x/(2^n)  in decimal,
  //     ==> argv >> 2 means we're dividing argv by 4
  //     ==> to access a 32-bit value, divide the address by 4
  HEAP32[argv >> 2] = allocateUTF8OnStack(Module['thisProgram']);

  // Store remaining string arguments at contiguous locations in the heap
  for (var i = 1; i < argc; i++)
    HEAP32[(argv >> 2) + i] = allocateUTF8OnStack(args[i - 1]);
  HEAP32[(argv >> 2) + argc] = 0;

  // Call main(), and pass argc and the address where argv is found
  var ret = Module['_main'](argc, argv, 0);
}
```

... hence why there is a `callMain()` function to abstract out the complexity 😄.

For more details, here are some more resources:

- Emscripten Memory Representation
- Details on Typed Arrays Mode 2
- Original Emscripten Paper

Chapter 5. The Module Object

What is the Module object?

So far, we've come across the `Module` object, and used it a few times for its utility functions—such as `Module.callMain()` and `Module.cwrap()`—and to customize the behavior of our WebAssembly module via parameters such as `Module.onRuntimeInitialized`.

In this chapter, we'll discuss the Module object more broadly and why it's useful. In a nutshell, the Module object is simply a JavaScript object that contains various parameters and callback functions, which are used by the glue code generated by Emscripten (e.g. `hello.js`) to customize our WebAssembly module.

Configure initial conditions

Let's consider one example where Module can help us save time. We previously discussed how you can prevent the `main()` function from being automatically called by compiling your code by setting the flag `INVOKE_RUN=0`.

Doing this change required re-compiling our code from scratch which, while it was fast for the simple example we looked at, is also inconvenient. A simple alternative would have been to set `Module.noInitialRun` to `true`, which saves us having to re-compile our code.

As an example, let's consider our Hello World program from Chapter 2:

```
#include <stdio.h>

int main() {
  printf("Hello World!\n");
  return 0;
}
```

Again, we can compile it to `hello.js` using emcc:

```
$ emcc hello.c -o hello.js
```

Let's see how we can use the Module object to define the module's behavior on page load:

```
<script type="text/javascript">
var Module = {
  // Don't run main() at page load
  noInitialRun: true,
  // Run custom function on page load
  onRuntimeInitialized: () => {
    console.log("Launching main...");
    Module.callMain();
    console.log("Done");
  }
};
</script>
<script src="hello.js"></script>
```

where `onRuntimeInitialized` is what is executed once the WebAssembly module is ready (we can't call C functions from the module until it's been initialized successfully). Running this in the browser, you should see the following on the developer console:

```
Launching main...
Hello World!
Done
```

Custom stdout/stderr handling

By default, if your C code outputs text to `stdout`, it will show up in the developer console (`console.log`), whereas outputs to `stderr` will show up as warnings in the console (`console.warn`).

However, you may want to catch `stdout` and `stderr` yourself and store them in memory for later processing, or to display them to the user in your app's UI instead of the console. To do so, we can use `Module.print` and `Module.printErr`:

```
<script type="text/javascript">
var output = [];
var Module = {
  // Don't run main on page load
  noInitialRun: true,
  // Run custom function on page load
  onRuntimeInitialized: () => {
    console.log("Launching main...");
    Module.callMain();
    console.log("Done");
  },
  // Custom function to process stdout: keep in memory
  print: stdout => output.push(stdout),
  // Custom function to process stderr: show in console as errors (in red)
  printErr: stderr => console.error(stderr)
};
</script>
<script src="hello.js"></script>
```

If you run this in the browser, once the module is ready, output will be an array containing a single `Hello World!` element. Each time you call `Module.callMain()` in the console, you'll notice that the `output` variable will have one more such element.

Commonly-used Module parameters

Here is a table summarizing the Module parameters we've discussed so far, along with a few new ones that may come in handy:

Customize initialization		
noInitialRun	Parameter	Set to true if you don't want `main()` to be called at page load
arguments	Parameter	Array of arguments sent to `main()` function once module is initialized (only valid if `noInitialRun = false`).
onRuntimeInitialized	Callback	Called once WebAssembly module is ready to be used. e.g. `() => console.log("Initialized");`
locateFile	Callback	By default, the `.wasm` file will be loaded from the same folder as the `.js` file. If that is not correct, use this function to define a different path, e.g. `path => `wasm/${path}``, or even a URL to download the file from, e.g. `path => `https://domain.com/wasm/${path}``
preInit	Function(s)	This function is called before the WebAssembly module is downloaded and initialized. `preInit` accepts an array of functions; note that the last function in the array will be executed first. This is a useful place to mount other file systems as we'll cover in Chapter 6.
preRun	Function(s)	This function is called after preInit, after the WebAssembly module is loaded, but before `callMain()` is called. As with `preInit`, `preRun` either accepts a function or an array of functions; last function in the array is executed first.
onAbort	Callback	This function is called if the `.wasm` file you're trying to load is not found; useful for detecting such errors and notifying the user and/or logging an error in your systems. This function is also called if you make a call to `abort()` within your C code.
Customize stdout/stderr behavior		
print	Callback	Custom function to capture stdout, e.g. `out => alert(out)`
printErr	Callback	Custom function to capture stderr, e.g. `err => alert(err)`
logReadFiles	Parameter	If set to `true`, will output "`read file: <path>`" to `stderr` the first time you read from a file (see Chapter 6 for details about file management with WebAssembly).

Recall our `hello_custom.c` example from Chapter 4, which outputs a custom greeting and name:

```
#include <stdio.h>

int main(int argc, char **argv) {
  printf("%s %s!\n", argv[1], argv[2]);
  return 0;
}
```

Here's an example that makes use of several of these Module parameters so you can see the order in which these different callbacks are called:

```
<script type="text/javascript">
var Module = {
  /* Pre-initialization */
  preInit: [
    () => console.log('Pre initialization 1'),
    () => console.log('Pre initialization 2')
  ],
  preRun: [
    () => console.log('Pre run 1'),
    () => console.log('Pre run 2')
  ],

  /* Initialization */
  // Run main on page load
  noInitialRun: false,
  // Use these arguments for the first call to main
  arguments: ["Good evening", "Robert"],

  // When the module is ready
  onRuntimeInitialized: () => console.log("Ready"),
};
</script>
<script src="hello_custom.js"></script>
```

Other, less commonly-used, parameters and callbacks can be found on the Emscripten documentation page.

Chapter 6. File Management

Introduction

One feature that is needed by a lot of command line tools is the ability to read and write files on the user's file system. For example, your C program might use `fopen()` to load your input data, and `fprintf()` to output the result of a calculation or file manipulation to disk.

But wait a minute—we're operating in the browser, why would we need files? Well, here are a few examples of when having that feature would be useful:

- **Parsing user files in the browser**
  - Example 1: Your app requires the user to provide a large file, from which you only consider a small portion at the back-end; you can improve the user experience by parsing and uploading only the needed information.
  - Example 2: If you can do some computations in the front-end and save the round trip to your servers, you could, in some cases, avoid uploading files altogether.
  - Example 3: If you're porting your C/C++ desktop app to the web and you need to parse user files in the browser, having a virtual file system will help.

- **Configuration files**
  - Example 1: An existing C library that you're converting to WebAssembly requires a configuration file as input, and it might not be worth the trouble to modify the library to support command-line arguments instead of files. See Chapter 9 on how to pre-mount existing files.

Although we don't have direct access to the user's file system (thankfully!), Emscripten does provide a virtual, POSIX-like file system (the source code is available here if you're curious about how the sausage is made).

What this means is that we have access to something that looks like a file system, in that we can create files and folders, read/write files, assign permissions, and more. The main difference is that this virtual file system is not connected in any way to the file system on the user's computer, and hence you cannot accidentally overwrite your users' files. In fact, this virtual file system is only stored in memory, so it disappears when you refresh the page (although we discuss how to persist this file system in Chapter 9).

Enable the virtual file system

Assuming we have access to the file `myFile.txt` on our virtual file system (we'll address this in a minute), consider a simple C program (call this `file.c`) that outputs the contents of `myFile.txt`:

```
#include <stdio.h>

int main()
{
  // Open myFile.txt for reading
  FILE *fp = fopen("myFile.txt", "r");

  // Until we reach the end of the file (EOF), output the next character
  char c;
  while((c = fgetc(fp)) != EOF)
    printf("%c", c);

  // Close the file
  fclose(fp);

  // Flush stdout; otherwise, we get a warning in the developer console that
  // looks like this: "stdio streams had content in them that was not flushed"
  printf("\n");

  return 0;
}
```

You usually don't need to explicitly specify that you need a virtual file system in your WebAssembly module, as Emscripten infers it from your code, but to make sure it's included, you can enable the `FORCE_FILESYSTEM` flag:

```
$ emcc file.c \
    -o file.js \
    -s FORCE_FILESYSTEM=1
```

That's all we need to enable the file system; now let's see how we can use it!

How to mount files

Since our virtual file system doesn't have any files on it, let's create `myFile.txt` using some mock content and mount it onto our virtual file system:

```
<script type="text/javascript">
var Module = {
  // Mount myFile.txt to root folder /
  onRuntimeInitialized: () => FS.writeFile("/myFile.txt", "abc\ndef\nghi")
};
</script>
<script src="file.js"></script>
```

Here we're using the function `FS.writeFile()` which, as the name implies, writes a file on our virtual file system, given some data provided in a string. If you navigate to this HTML page in the browser, you'll see that the compiled C file was able to read the contents of the file we created. You should see this in the developer console:

```
abc
def
ghi
```

> **Note**
>
> Whereas the global `Module` object contains useful functions for managing the behavior of our WebAssembly code, the global `FS` object provides us with functions to perform common file system operations.

Next, we'd like to process the file ourselves instead of having `main()` run at page load. We can do that using the `Module.onRuntimeInitialized()` callback, as we discussed previously in Chapter 5:

```
<script type="text/javascript">
var Module = {
  // Don't auto-run main()
  noInitialRun: true,
  onRuntimeInitialized: () =>
  {
    // Mount myFile.txt to root folder /
    FS.writeFile("/myFile.txt", "abc\ndef\nghi");

    // Read the file back. Note that we specify a UTF-8 encoding; otherwise
    // FS.readFile() will return the Uint8Array [97, 98, 99, 10, 100, 101, ...],
    // where each array element represents a character!
    fileContents = FS.readFile("/myFile.txt", { encoding: "utf8" });
    console.log("File contents:");
    console.log(fileContents);
    console.log("Number of lines:", fileContents.split("\n").length);
  }
};
</script>
<script src="file.js"></script>
```

Note

We're mounting the file to `/` only within the virtual file system in the browser; we can't affect the user's local file system, and data on the file system is lost when you refresh the page (see Chapter 9 for how to use IDBFS to persist the state of your virtual file system).

As you can see above, we're able to read the file on demand, and parse it. In this example, all we did was count the number of lines, but if this was a CSV or TSV file, we could have used a library like PapaParse to auto-detect the delimiters and convert the `fileContents` string into an array of rows.

Commonly-used FS functions

Emscripten's virtual file system also supports what you'd expect (try typing those in the console):

```
// Create a folder
FS.mkdir("/data");

// Create a new file
FS.writeFile("myFile.txt", "some contents");
```

```
// Another approach to creating files, with more options
// Instead of: FS.writeFile("/data/myFile2.txt", "abcdef");
FS.createDataFile(
  "/data",          // folder within which we'll save the file
  "myFile2.txt",    // file name
  "abcdef",         // file contents (a string)
  true,             // is this file readable?
  true              // is this file writable?
);

// We can also rename a file
FS.rename("/data/myFile2.txt", "/data/myFile3.txt");

// Read the file back
FS.readFile("/data/myFile3.txt", { encoding: "utf8" });

// Delete the file
FS.unlink("/data/myFile3.txt");

// Do the equivalent of "ls" on the command line
FS.readdir("/");
FS.readdir("/data");

// Delete remaining file in /data
FS.unlink("/data/myFile.txt");

// Delete the folder /data
FS.rmdir("/data");

// Get current working directory
FS.cwd();
```

This should give you a good idea about what is possible with Emscripten's virtual file system. For more information, the full FS API documentation can be found here, and although we discussed a few more advanced FS APIs, you'll find the rest at this page.

Enable gzip support

Enabling gzip support is a commonly needed functionality on the web, so Emscripten makes it fairly easy to include in your application. Luckily, you don't need to manually compile the zlib library, and link it to your project. Instead, you simply enable the `USE_ZLIB` flag:

```
emcc hello.c \
    -o hello.js \
    -s USE_ZLIB=1
```

Note that the first time you use `USE_ZLIB` will take longer than usual, as it'll be fetch the zlib library, port it to WebAssembly and cache it for later use:

```
INFO:root:generating port: zlib.bc... (this will be cached in
"/root/.emscripten_cache/asmjs/zlib.bc" for subsequent builds)
INFO:root: - ok
```

Chapter 7. Compiling Existing Tools

So far, we've seen how to use WebAssembly with small snippets of code that we wrote ourselves. We now turn our attention to compiling libraries and other larger, existing codebases.

A simple example

Let's see how we can compile a small off-the-shelf utility. For this example, we'll consider a tool called seqtk, which is a command line tool used in the field of bioinformatics for parsing common file formats. The details of what seqtk does or how it works don't matter too much here, as we're treating it as a black box that we want to port to WebAssembly.

Start by cloning the repository (seqtk is available at https://github.com/lh3/seqtk):

```
$ git clone https://github.com/lh3/seqtk.git
Cloning into 'seqtk'...
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 335 (delta 0), reused 1 (delta 0), pack-reused 331
Receiving objects: 100% (335/335), 149.22 KiB | 653.00 KiB/s, done.
Resolving deltas: 100% (191/191), done.
```

Go into the folder and you'll notice it's a project with one `.c` file and two `.h` files:

```
$ cd seqtk/
$ ls
LICENSE       Makefile       README.md       khash.h       kseq.h       seqtk.c
```

First, let's see how we would normally compile this tool from C to binary (not WebAssembly) using the `gcc` compiler (the command below is from the Makefile; we'll see in a bit how to use the Makefile directly):

```
$ gcc seqtk.c \
    -o ./seqtk \
    -O2 \
    -lm \
    -lz
```

```
# Use gcc compiler
# Compile to binary
# Ask compiler to optimize the code (docs)
# Include the math library
# Include zlib library (support reading/writing .gz files)
```

Once compiled, we can launch `./seqtk` on the command line, and a help screen will show up, with a list of accepted commands (the output below was truncated for space):

```
$ ./seqtk
Usage:   seqtk <command> <arguments>
Version: 1.3-r106

Command: seq       common transformation of FASTA/Q
         comp      get the nucleotide composition of FASTA/Q
...
```

Now let's compile the same program to WebAssembly using `emcc` (differences are in red):

```
$ emcc seqtk.c \
    -o seqtk.js \
    -O2 \
    -lm \
    -s USE_ZLIB=1
```

```
# Use emcc compiler
# Compile to WebAssembly
# Ask compiler to optimize the code (docs)
# Include the math library
# Include the zlib library (see Chapter 6)
```

As you can see, this is very similar to what we used to compile to binary. Here they are side by side:

```
$ emcc seqtk.c \
    -o seqtk.js \
    -O2 \
    -lm \
    -s USE_ZLIB=1
```

```
$ gcc seqtk.c \
    -o seqtk \
    -O2 \
    -lm \
    -lz
```

Let's validate that we compiled `seqtk` to WebAssembly properly: create a `seqtk.html` file that loads `seqtk.js`:

```
<script src="seqtk.js"></script>
```

If you point your browser to `seqtk.html`, you'll see the same output we saw earlier on the command line, but this time it'll show up in the developer console in yellow as a `console.warn()`:

```
> Usage:   seqtk <command> <arguments>
> Version: 1.3-r106
> Command: seq       common transformation of FASTA/Q
>          comp      get the nucleotide composition of FASTA/Q
...
```

To verify that it works beyond just the main output, let's call the command `comp` from the console. In a nutshell, `comp` takes as input a file with `.fa` extension, and returns information about how many times in that file we saw the letters A, C, G, T, which are the "letters" we find in our DNA (i.e. here, `comp` is short for composition of DNA letters).

Let's give it a real file to run on. Write the following in the console:

```
> FS.writeFile("/test.fa", ">my_sequence\nACGTACGTA");
```

This creates the file `test.fa` with some sample data; namely, it defines `my_sequence` as the sequence of DNA letters `ACGTACGTA`, which contains 3 A's, 2 C's, 2 G's, and 2 T's.

Now let's call the `comp` utility, with `test.fa` as an additional argument:

```
> Module.callMain(["comp", "test.fa"])
my_sequence 9     3     2     2     2     0     0     0     4     0     0     0
```

i.e. there are 9 letters, with 3 A's, 2 C's, 2 G's, and 2 T's as expected.

As further validation, we get exactly the same output when running seqtk on the command line:

```
$ echo -e ">my_sequence\nACGTACGTA" > test.fa
$ ./seqtk comp test.fa
my_sequence 9     3     2     2     2     0     0     0     4     0     0     0
```

Congratulations, you just compiled your first C library into WebAssembly!

Using Makefiles

In the case of `seqtk`, the codebase is fairly simple, so we wrote our own `emcc` statement, but for more involved codebases, this may be too hectic. In the `seqtk` package, a Makefile is included, so let's see if we can use it directly to compile to WebAssembly, instead of writing our own `emcc` statement.

Emscripten provides useful tools called `emconfigure` and `emmake`, which are essentially wrappers around `configure` and `make` that help you port applications to WebAssembly.

Before compiling, let's make sure we don't have any lingering compiled `seqtk` binaries:

```
$ make clean
```

Here's what the Makefile looks like in the repository:

```
CC=gcc
CFLAGS=-g -Wall -O2 -Wno-unused-function
BINDIR=/usr/local/bin

all:seqtk

seqtk:seqtk.c khash.h kseq.h
    $(CC) $(CFLAGS) seqtk.c -o $@ -lz -lm

install:all
    install seqtk $(BINDIR)

clean:
    rm -fr gmon.out *.o ext/*.o a.out seqtk trimadap *~ *.a *.dSYM session*
```

We'll need to make a few changes (highlighted in red above) before compiling to WebAssembly, namely:

1. Replace `CC=gcc` with `emcc`
2. Add `-s USE_ZLIB=1` as a flag to emcc

To do that, let's use the `emmake` utility:

```
emmake make seqtk \
    CC=emcc \
    CFLAGS="-s USE_ZLIB=1"
```

This will create a `seqtk` output file. We can't execute that file directly because it's a binary file in a LLVM bitcode format. This is because the original Makefile specifies `-o seqtk` instead of `.js` as we've been doing. This is beyond the scope of this book but you can confirm that it's indeed LLVM bitcode using the LLVM bitcode analyzer:

```
llvm-bcanalyzer seqtk
```

Let's rename this file to `seqtk.o` (that's the extension that `emcc` recognizes; another option is `.bc`), and compile it to `.wasm/.js`:

```
$ mv seqtk seqtk.o
$ emcc seqtk.o -o seqtk.js -s USE_ZLIB
```

Now we can make sure it works using node:

```
$ node seqtk.js
```

As we've seen in this section, Emscripten provides useful tools for leveraging the existing Makefile of the library we're compiling, to make compilation easier. Now onto a much larger codebase!

Compiling the jq command line tool

So far, we've covered compiling relatively small codebases. Let's turn our attention to `jq`, a much larger project that consists of tens of thousands of lines of code. If you're not familiar with `jq`, it's a command line tool for parsing, creating and modifying JSON strings directly on the command line. It's a must-have tool for wrangling JSON on the command line, so if you've not used it before, I highly recommend you check it out.

First, let's clone the repo and enter the source code folder:

```
git clone https://github.com/stedolan/jq.git
cd jq
```

Before we compile `jq`, we need to fetch `oniguruma`, an open-source library that `jq` uses to support regular expressions; that library is a [submodule in the git repo]():

```
git submodule update --init
```

> **Note**
>
> The general approach to compiling codebases to WebAssembly starts by making sure you're able to compile them to binary (which is sometimes not straightforward!), then mapping those steps to WebAssembly using `emconfigure`, `emmake`, and `emcc`.

You'll find the instructions for compiling jq to binary in `jq`'s [README file]()—shown below in the left panel. And in the right panel, you'll find the corresponding instructions to compile jq to WebAssembly (with differences highlighted in red):

```
# Generate ./configure file
autoreconf -fi

# Run ./configure
./configure \
    --with-oniguruma=builtin \
    --disable-maintainer-mode

# Build jq executable
make LDFLAGS=-all-static
```

```
# Generate ./configure file
autoreconf -fi

# Run ./configure
emconfigure ./configure \
    --with-oniguruma=builtin \
    --disable-maintainer-mode

# Build jq executable
emmake make LDFLAGS=-all-static
```

So far so good! The only difference is that we're using Emscripten's `emconfigure` and `emmake` tools to wrap around the existing `configure` and `make` utilities, such that they use `emcc` and `em++` instead of `gcc` and `g++`.

However, we do have a few more steps to go. The `emmake` command above created the WebAssembly executable `jq`. Since we want the convenience of the .js glue code that Emscripten generates for us, let's create it:

```
# But first, rename the file to .o; otherwise,
# emcc complains that the "file has an unknown suffix"
mv jq jq.o
```

```
# Generate .js and .wasm files
emcc jq.o -o jq.js \
  -s ERROR_ON_UNDEFINED_SYMBOLS=0
```

Note that we use the flag `ERROR_ON_UNDEFINED_SYMBOLS` to ignore warnings such as `"undefined symbol: llvm_fma_f64"` (if you try compiling with just `emcc jq.o -o jq.js`, you'll see the error).

As you can see, the differences during the build process are surprisingly minimal given that we're compiling tens of thousands of lines of code from C to being able to run them in the browser! Can you imagine re-writing `jq` from scratch, in JavaScript? 😱

To make sure it works, let's try some examples from the jq tutorial directly on the command line:

```
# Output the description of the latest commit on the jq repo
$ curl -s "https://api.github.com/repos/stedolan/jq/commits?per_page=5" | \
    node jq.js '.[0].commit.message'
"Improve linking time by marking subtrees with unbound symbols"

# Output an array of commit messages and committer's name
$ curl -s "https://api.github.com/repos/stedolan/jq/commits?per_page=5" | \
    node jq.js '.[] | {message: .commit.message, name: .commit.committer.name}'
{
  "message": "Improve linking time by marking subtrees with unbound symbols",
  "name": "Nico Williams"
}
[...]
```

Building an interactive jq app

Now let's take it a step further and use the compiled WebAssembly in an app that lets us interactively make jq queries right in the browser. There are apps that already exist to do that—namely jqplay.org—though those tools tend to rely on sending the request to the backend, where it is evaluated. This means the potential for slower response time, and could be a security concern (we need to make sure users can't execute arbitrary commands on the server).

The app we're building here uses almost exactly the same logic that I used to build jqkungfu.com!

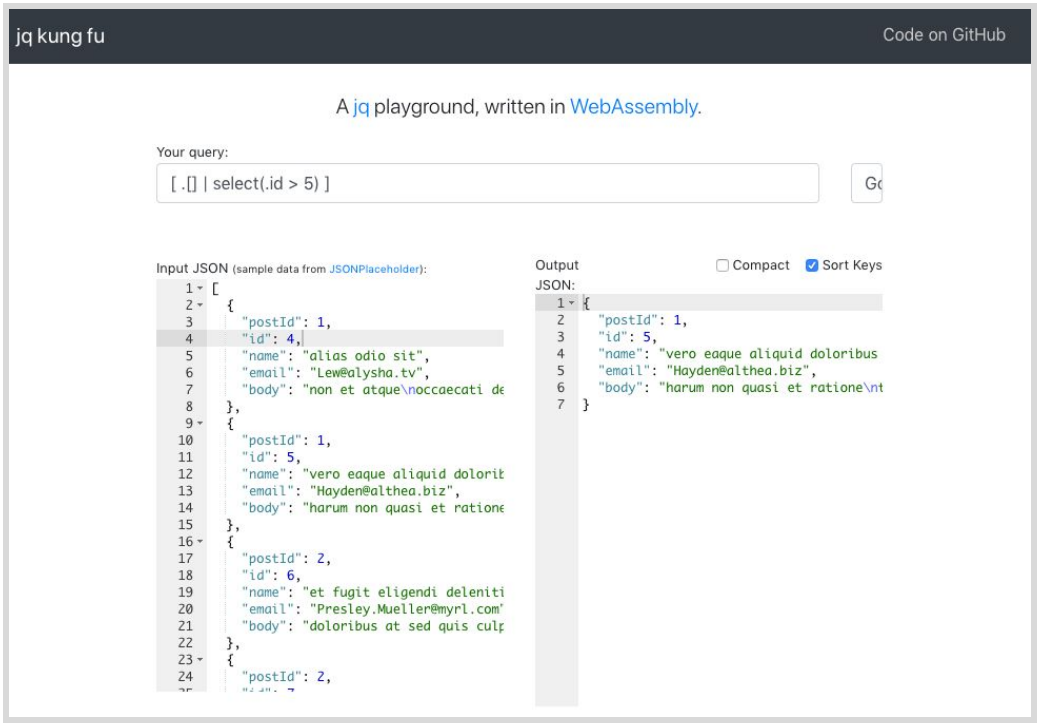
The HTML/JS code below is all you need to enable that (with a minimal UI that is!). The code is commented so you can follow along:

```
<!-- Just a bit of styling -->
<style type="text/css">
#input { width: 50%; height: 200px; }
#output { width: 50%; height: 200px; }
</style>

<!-- The main logic-->
<script type="text/javascript">
// WebAssembly module config
var output = [];
var Module = {
      // Don't automatically run main() function on page load
      noInitialRun: true,

      // Print functions (capture jq output)
      print: stdout => output = stdout,
      printErr: stderr => console.warn(stderr),

      // When the module is ready, enable the "Run" button
      onRuntimeInitialized: function() {
            document.getElementById("btnRun").disabled = false;
      }
};

// Utility function that runs jq, given a JSON string and a query
function jq(jsonStr, query)
{
      var fileName = "/tmp/data.json";

      // Create file from JSON string
      FS.writeFile(fileName, jsonStr);

      // Launch jq's main() function on that file
      //   -M to disable colors
      //   -r to output in raw format
      //   -c to output in compressed format
      output = [];
      Module.callMain([ "-M", "-r", "-c", query, fileName ]);
```

```
	// Re-open stdout/stderr since jq closes them when it exits
	// Otherwise we get an error about having a "Bad file descriptor"
	// if we run jq twice in a row
	FS.streams[1] = FS.open("/dev/stdout", "w");
	FS.streams[2] = FS.open("/dev/stderr", "w");

	return output;
}

// On page load
document.addEventListener("DOMContentLoaded", function()
{
	// On button click
	document.getElementById("btnRun").addEventListener("click", function()
	{
		// Run jq and update output textarea
		document.getElementById("output").value = jq(
			document.getElementById("input").value,
			document.getElementById("query").value
		);
	});
});
</script>

<!-- Load jq WebAssembly module -->
<script src="jq.js"></script>

<!-- Basic UI for jq -->
<input id="query" type="text" value=".b"><br />
<textarea id="input">
{
	"a": 123,
	"b": {
		"bb": 12,
		"cc": 45
	},
	"c": 789
}
</textarea><br />

<input id="btnRun" type="button" value="Run" disabled><br /><br />
<textarea id="output"></textarea>
```

Here's a screenshot of the code above in action:

Admittedly, this doesn't look like much, but with a bit more CSS, it can look like this:



See GitHub for jqkungfu's source code.

To me, this is where WebAssembly's strength shines through: the ability to take off-the-shelf command-line utilities, and turn them into libraries that are useful on the web, whether to speed up existing code or to enable a feature in the first place, is extremely powerful. `jq` in particular is a great example of that; it's a powerful command-line tool, yet can also be very useful on the web. Here we built a tool for playing around with `jq` outside the command line, but you can easily imagine how it could also be used in data-heavy web applications, to enable users to query JSON data using the familiar language that `jq` provides.

Note

If you want more practice, there's nothing better for learning WebAssembly than picking a repo at random, compiling it, and google-ing the error messages you encounter along the way!

Chapter 8. Graphics

Introduction

In this chapter, we'll focus on how we can combine WebAssembly with WebGL, an API to enable graphics in the browser, and which resembles that of OpenGL ES 2.0 (ES stands for Embedded Systems, which is a subset of OpenGL APIs commonly used in mobile devices). For the details on what Emscripten supports and what it doesn't for porting to OpenGL, see this link.

Using WebGL allows us to draw graphics in a performant manner to an HTML5 Canvas without needing any additional browser plugins. It's worth noting, however, that you don't need WebGL or WebAssembly to use the `<canvas>` tag; you can use the JavaScript Canvas API to draw directly on a canvas element (see the MDN documentation for details and sample Canvas API code).

A simple example

Let's start simple by compiling some C code written with OpenGL that draws simple shapes on the screen. Luckily, the Emscripten repo comes with some handy OpenGL examples from the book *OpenGL ES 2.0 Programming Guide*.

Let's look at the Hello Triangle example from that guide, where the goal is to draw a red triangle on the screen.

First, we'll download the code from GitHub (you can do that outside the container), go to the folder containing the code examples (we also make a copy of `Hello_Triangle.c` for convenience:

```
git clone https://github.com/kripken/emscripten.git
cd emscripten/tests/glbook
cp Chapter_2/Hello_Triangle/Hello_Triangle.c triangle.c
```

We can now compile `triangle.c` using emcc the same way we did in the past, but with two slight differences:

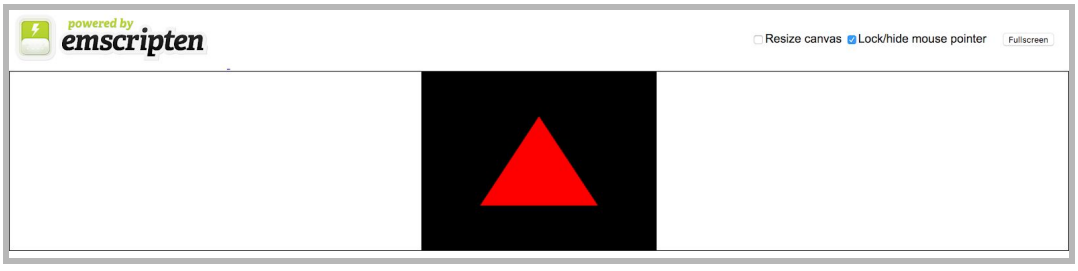
1. We don't just compile our `.c` code; we also include commonly-used `.c` utilities (stored in the Common/ folder) that are used in the `glbook/` examples; and
2. We use `gcc`'s `-I` flag to specify a folder that contains header files used by our program

Here's how we do it:

```
$ emcc ./Common/*.c triangle.c \
    -o triangle.html \
    -I ./Common/
```

Note that instead of sending the output to `triangle.js`, we output to `triangle.html`. This is a useful Emscripten feature that auto-generates a valid HTML template that initializes the `.js` and `.wasm` that is generated, and uses <canvas> to display our graphics. We previously didn't use it since our previous examples were all console-based, and we wanted to demonstrate how you generally initialize the Module object, but it's a good shortcut for quickly testing that your app is working.

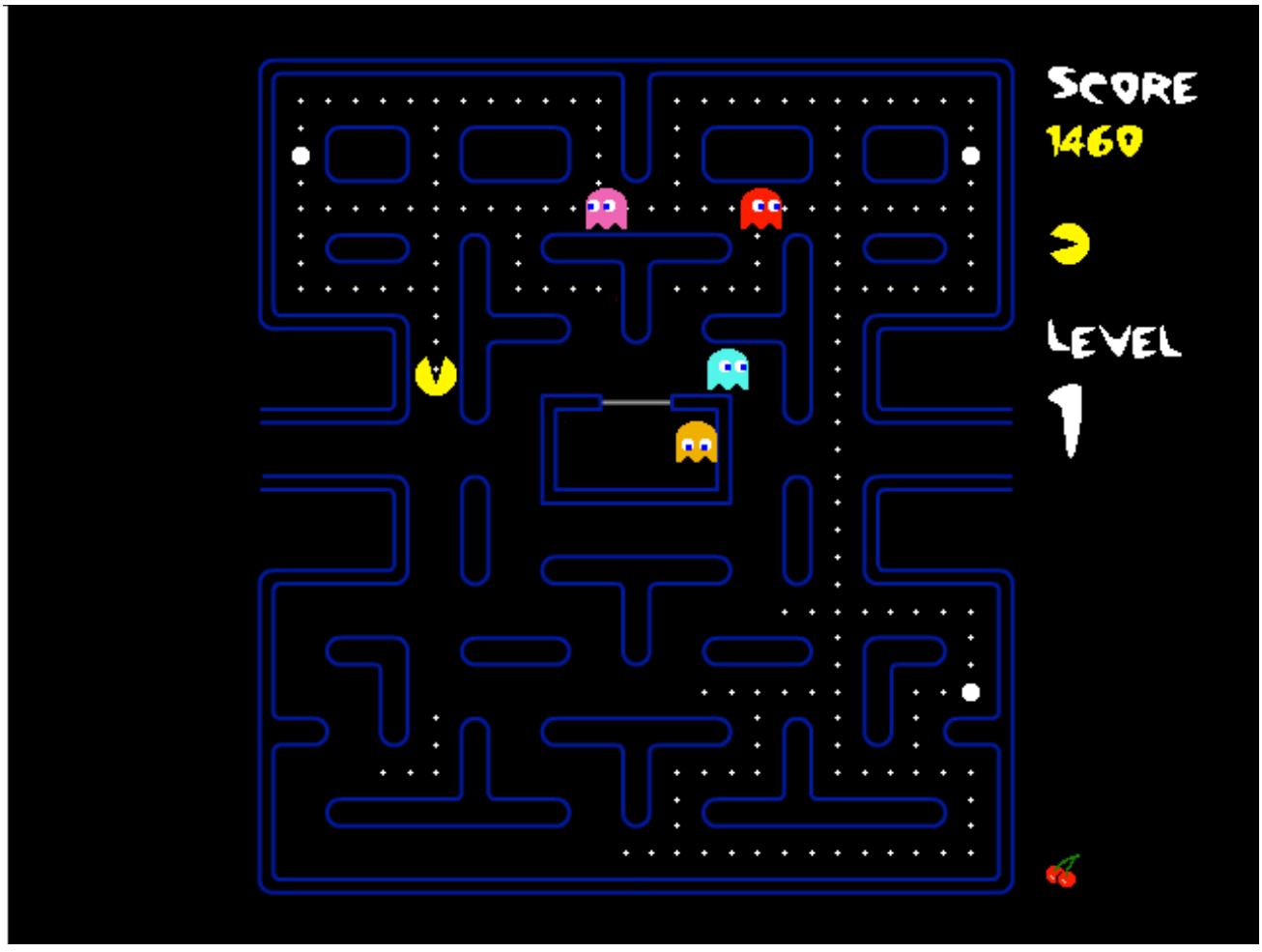
Now if we launch `triangle.html` in the browser, you should see a red triangle in your browser:



Congratulations, you've just compiled your first OpenGL project into WebAssembly! As an exercise, you can try compiling other examples shown in the tests/glbook folder. Or... head on over to the next section, where we port Pacman to WebAssembly.

Compile Pacman to WebAssembly

Let's now jump right into a much more interesting example: Pacman! You can find an open-source C++ clone of the game on GitHub: https://github.com/ebuc99/pacman.



A note about the SDL library

When looking for games to port to WebAssembly, you'll notice that many C/C++ games or graphical programs use the "SDL" library (Pacman included!). SDL is short for "Simple DirectMedia Layer" and, in a nutshell, is a library with very useful utility functions, including handling user input via mouse/keyboard/joystick/etc, playing audio files, and more. For details, see the SDL documentation page.

Because SDL2 is so popular, it has already been ported to WebAssembly, and has special flags that Emscripten supports out of the box (including `-s USE_SDL=2`) so that it can be linked to your project more easily.

> **Warning**
>
> This chapter relies on SDL2, which requires certain libraries to be installed in your container. You should have those already installed if you used the container provided with this book (back in Chapter 2), but if not, install the needed libraries by running: `apt-get install -y libsdl2-dev libsdl2-image-dev libsdl2-mixer-dev libsdl2-ttf-dev`

Our game plan

I chose this game as an example because it highlights a lot of the issues you'll commonly see when compiling games, or more generally, graphical applications that use SDL.

Let's start by cloning the repo:

```
git clone "https://github.com/ebuc99/pacman.git"
cd pacman
```

Looking at the README file, the build instructions are similar to what we've seen in Chapter 7 for compiling C/C++ codebases. Here is the overall pattern for compiling to binary on the left, and WebAssembly on the right (differences in red):

```
./configure
make
make install
```

```
emconfigure ./configure
emmake make
em++ [...] -o pacman.html
```

In reality, it's a more complicated than that; let's dig into each line to understand how it needs to be adapted for WebAssembly compilation with Emscripten.

As it's the case with many games, porting to WebAssembly can be an arduous and complex step, and Pacman is no different. So I won't immediately be showing you the final result of how you can compile Pacman to WebAssembly, because it'll just seem like lots of magic commands with no clear reasoning behind what we're doing. Instead, this section will take you one step--and one error--at a time, effectively simulating what you'll encounter as you set out to port games to WebAssembly. If you're impatient, you can skip the next few sections.

Step 1: emconfigure

In most cases, this step is straightforward; you just need to call `emconfigure ./configure` and you're all set. However, if we do that, we get this error (main part in red):

```
$ emconfigure ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
[...]
checking for SDL - version >= 2.0...
emscripten sdl2-config called with /emscripten/system/bin/sdl2-config --cflags
emscripten sdl2-config called with /emscripten/system/bin/sdl2-config --libs
emscripten sdl2-config called with /emscripten/system/bin/sdl2-config --version
emscripten sdl2-config called with /emscripten/system/bin/sdl2-config --version
emscripten sdl2-config called with /emscripten/system/bin/sdl2-config --version
2.0, bad version string
no
configure: error: *** SDL version 2.0 not found!
ERROR:root:Configure step failed with non-zero return code: 1.
Command line: ./configure at /src/pacman
```

It seems the configure file is expecting SDL version 2.0, which we should already have. Let's check by using the `sdl2-config --version` program mentioned in the error above.

Indeed, typing `/emscripten/system/bin/sdl2-config --version` in the command line, it seems we have version 2.0.0 instead of 2.0, which is causing the error 🤦‍♀️.

Let's modify `./configure` to avoid this error by replacing the expected SDL version from 2.0 to 2.0.0:

```
$ sed -i "s/^SDL_VERSION=2.0$/SDL_VERSION=2.0.0/" configure
```

Note

As we saw previously, the configure script is used to generate a custom `Makefile`, given the user's environment. Well it turns out there's a utility called `autoconf` that generates this `./configure` file, given a `configure.ac` file.

Here we modified the `configure` file directly because it was a simple change. In some cases, this might be too unwieldy to be feasible. In those cases, you can modify `configure.ac` (if it's available in the repo) and use `autoconf` to generate an updated `./configure` file, simply by typing `autoconf` on the command line.

Next, the README file suggests that we'll need the following SDL packages: `sdl-image`, `sdl-ttf` and `sdl-mixer`, which translates to the following Emscripten flags: `-s USE_SDL=2 -s USE_SDL_IMAGE=2 -s USE_SDL_TTF=2 -s USE_SDL_MIXER=2`.

So here's what the final `emconfigure` command will look like:

```
$ SDL_LIBS="-s USE_SDL=2 -s USE_SDL_IMAGE=2 -s USE_SDL_TTF=2 -s USE_SDL_MIXER=2"
$ emconfigure ./configure \
  SDL_LIBS="$SDL_LIBS" \
  CXX=em++ CC=em++
```

Note that we also set the CC and CXX flags to em++ to ensure that we don't use g++ during the compilation. On to the next step!

Step 2: emmake

The `emmake` command for this game should be straightforward, and with similar flags to what we passed on to `emconfigure`:

```
$ emmake make \
  SDL_LIBS="$SDL_LIBS" \
  CXX=em++ CC=em++
```

However, running this command gives us this error:

```
./constants.h:4:10: fatal error: 'SDL2/SDL.h' file not found
#include <SDL2/SDL.h>
         ^~~~~~~~~~~~
```

With Emscripten and SDL, you'll generally have to modify your .c/.cpp/.h files to replace `#include <SDL2/somefile.h>` with simply `#include <somefile.h>`.

Instead of doing this manually, we can write a Bash 3-liner to do this for us:

```
# Get a (unique) list of files we need to modify, and use sed to modify each one
$ grep -R -l "include <SDL2/" src/ | \
    sort | uniq | \
    xargs -n1 -I{} sed -i 's|include <SDL2/|include <|' {}
```

Now if we run our emmake command from above, it should complete with no errors (just warnings):

```
$ emmake make \
  SDL_LIBS="$SDL_LIBS" \
  CXX=em++ CC=em++
```

> **Note**
>
> If you see warnings about how `emcc: cannot find library "SDL2_mixer"`, that's ok because the Makefile contains references to `-l SDL2_mixer`, and we're using `-s USE_SDL_MIXER=2` as the equivalent.

Step 3: generate the .html

If you think this was too easy, you're right, but let's keep going by generating the .html file containing the Pacman game:

```
# As we did with the jq library in Chapter 7, we rename the LLVM bitcode output
# to .bc so that em++ recognizes it as a valid input file
cp src/pacman src/pacman.bc

# As with the simple triangle example above, output a .html file and attach
# files used by the game, including fonts, images, and sounds
em++ src/pacman.bc \
    -o pacman.html \
    $SDL_LIBS \
    --preload-file data/@/usr/local/share/pacman/ \
    --use-preload-plugins \
    -s EXIT_RUNTIME=1
```

A few notes about the new flags we're using:

- We use the `--preload-file` flag to load the files within `./data`. These files include fonts, images, and sounds needed by the game. We use the `@` to define where on the virtual file system these files will be available. If you look at the source code of the game, it expects to find these files at `/usr/local/share/pacman`. In Chapter 9, we discuss preloading files in more details.
- The `--use-preload-plugins` ensures that the file types will be inferred by their extension.
- We enable the `EXIT_RUNTIME` flag (see this FAQ entry).

If you open pacman.html in your browser, you will see this screen:

Warning

If you encounter this error in the console: `emscripten_get_num_gamepads() can only be called after having first called emscripten_sample_gamepad_data()`, this is a known issue from an older version of Emscripten (< v1.38.26); make sure you're using the latest version (which you should if you used the container we created in Chapter 2).

Although there are no errors, your browser tab will likely freeze after a few seconds. This is because it's stuck in an infinite loop! Back to square 1.

Step 4: Un-infinite-loopifying the code

Games are commonly built as infinite loops that wait around for user input, or mouse movement, or an animation to finish, etc. Although that works for a desktop application, this does not fly in the browser, where infinite loops can crash your browser, or at the very least crash the tab your app is running in.

Here's a snippet from one such infinite loop from `src/game.cpp`, line 201 (infinite loop in red):

```
void Game::start() {
  init();
  Sounds::getInstance()->playIntro();
```

```
  // game loop
  while (eventloop()) {
    handleAnimations();

    // handle time based counters
    handleStartOffset();
    handleHuntingMode();
    handleSleep();
    handleFruit();

    // move all figures, if they are allowed to move - and check, what happened
    checkedMove();
    Pacman::getInstance()->check_eat_pills();
    checkScoreForExtraLife();
    checkedRedraw();
    if (checkLastPillEaten())
       continue;
    checkGhostTouched();

    updateDelayTime();
  }
  stop(true);
}
```

What this means for us is that we have to restructure some of the C/C++ code to replace infinite loops with something the browser can handle, i.e. calling the `eventloop()` function periodically instead of running it infinitely. As you'll see in a second, this is one of the more difficult parts of porting graphical applications to WebAssembly.

Generally, we can use Emscripten's `emscripten_set_main_loop()` function in our C code to define a function to call periodically:

```
// Set the main loop
emscripten_set_main_loop(
  eventloop,  // name of function to call
  0,          // framerate (use zero to let the browser handle it)
  1           // whether to simulate an infinite loop
);
```

This is easy enough for cases when you have a separate function, but it's trickier with Pacman, where we have objects and their internal state. This is an issue because `emscripten_set_main_loop()`

requires you to provide a static function (or static method within a class), so we'll have to do some acrobatics to track the current instance of a class in memory so that the static method can operate on it.

Let's first replace the infinite loop with a call to `emscripten_set_main_loop()` (new code in red):

```
void Game::start() {
  init();
  Sounds::getInstance()->playIntro();
  // game loop
  emscripten_set_main_loop(Game::mainloop, 0, 1);
}
```

We can now define the `Game::mainloop` function, but since it has to be a static method, we fetch the current instance of game (`Game::getInstance()`) and modify the code to run all the methods on that current instance (modified code in red):

```
void Game::mainloop()
{
  // Fetch current instance of the Game (needed because this method has to be
  // static for emscripten to call it)
  Game *g = Game::getInstance();

  if(g->eventloop()) {
    g->handleAnimations();

    // handle time based counters
    g->handleStartOffset();
    g->handleHuntingMode();
    g->handleSleep();
    g->handleFruit();

    // move all figures, if they are allowed to move - and check, what happened
    g->checkedMove();
    Pacman::getInstance()->check_eat_pills();
    g->checkScoreForExtraLife();
    g->checkedRedraw();
    if (g->checkLastPillEaten())
       return;  // instead of continue;
    g->checkGhostTouched();

    g->updateDelayTime();
```

```
  } else {
    g->stop(true);
  }
}
```

The original code didn't need to use the `g->` pattern, since it was calling methods on the current instance of the `Game` class; we have to use this approach since `Game::mainloop` is now a static method.

We also need to modify `game.h` to declare the `Game::mainloop` function:

```
class Game {
    public:
       static Game *getInstance();
       static void cleanUpInstance();
       static void mainloop();

    [...]
};
```

Phew, that wasn't trivial! 😅 Unfortunately, this isn't the only infinite loop in the game, which means we also need to modify those, using the same principles shown above. We won't go through them in detail, but see the attached code for the final code. To see how each file was modified, you can also use `git diff`.

Step 5: Try again

Now that we've converted all the infinite loops, we can compile the code again:

```
$ emmake make \
  SDL_LIBS="$SDL_LIBS" \
  CXX=em++ CC=em++
```

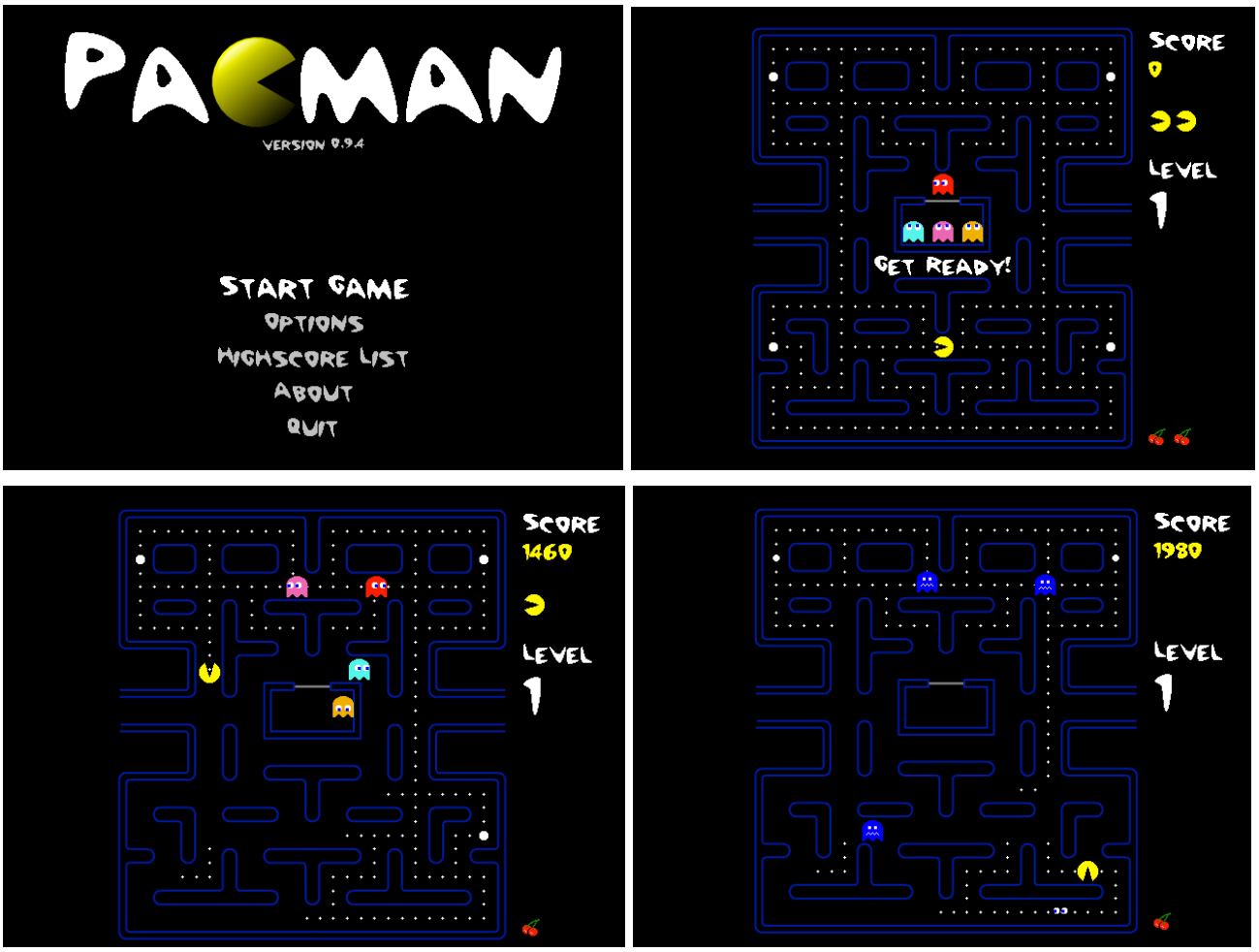
Followed by:

```
# As we did with the jq library in Chapter 7, we rename the LLVM bitcode output
# to .bc so that em++ recognizes it as a valid input file
cp src/pacman src/pacman.bc

# As with the simple triangle example above, output a .html file and attach
# files used by the game, including fonts, images, and sounds
```

```
em++ src/pacman.bc \
    -o ../pacman.html \
    $SDL_LIBS \
    --preload-file data/@/usr/local/share/pacman/ \
    --use-preload-plugins \
    -s EXIT_RUNTIME=1
```

Step 6: Play the game!

Finally, launch pacman.html in your browser, and you should be able to play Pacman:



What's Next?

As you might imagine, there's a lot of more to graphics with WebAssembly, and we've barely scratched the surface! In fact, you could probably fill an entire book about OpenGL and doing graphics with WebAssembly.

The goal here was to give you an example of how building graphics-based applications could be done in WebAssembly. For more inspiration, refer to this list. It includes great examples of applications that were ported over to the web with WebAssembly, including Unity games like AngryBots and Tanks, a port of Doom3 (!), and even a Mac OS 7 emulator—all running in your browser!

Chapter 9. Persisting the File System

Introduction

In all our examples above, refreshing the page wipes out the virtual file system. This is because the default file system used by Emscripten is only stored in memory, and is thus called `MEMFS`. On the other hand, the `IDBFS` file system (IndexedDB File System), allows us to persist the virtual file system in the user's browser using the IndexedDB API, so that refreshing the page maintains the contents of the file system!

IndexedDB is a WebAPI to allow developers to store data in the browser—similar to LocalStorage—but centered around a database with indexing support, as opposed to a key/value store. Luckily, Emscripten abstracts out IndexedDB so you don't need to understand how it works. Nonetheless, if you're curious about how you can use IndexedDB in your other projects, visit these MDN pages: Introduction to IndexedDB, and Using IndexedDB. And if you want to see how Emscripten uses IndexedDB, take a look at the IDBStore.js source code.

Create a persistent file system

To demonstrate how it can be used, let's use this HTML page as an example:

```
<script type="text/javascript">
var Module = {
  onRuntimeInitialized: () => FS.writeFile("/myFile.txt", "abc\ndef\nghi")
};
</script>
<script src="file.js"></script>
```

Load that page in the browser, open your developer console, and type the following to create a persistent virtual file system in the browser:

```
// By default, / mounts MEMFS, so let's create a folder /data,
// where we'll store our persistent files
FS.mkdir("/data");

// Now we'll mount a file system of type IDBFS to /data
FS.mount(IDBFS, {}, "/data");

// Create files inside and outside the IDBFS
FS.writeFile("/file1", "some contents");
FS.writeFile("/data/file2", "some contents");

// Save current state to IndexedDB
FS.syncfs(
  // populate = true: populate from IndexedDB
  // populate = false: save to IndexedDB
  false,
  // callback called on error
  err => console.warn(err)
);
```

Note that Emscripten will mount a `MEMFS` file system to the root folder /, but created the folder `/data` and mounted a `IDBFS` file system onto it.

Now refresh the page or type:

```
// Refresh page
location.reload();
```

Reload the file system after page refresh

When you refresh the page, the file system will be empty, so we need to recreate the folder `/data`, mount an IDBFS file system to it, and reload the file system contents from IndexedDB using the `FS.syncfs()` function:

```
// Recreate and mount IDBFS file system to /data
FS.mkdir("/data");
FS.mount(IDBFS, {}, "/data");

// Populate file system contents from IndexedDB
FS.syncfs(
  // populate = true: populate from IndexedDB
  // populate = false: save to IndexedDB
  true,
  // callback called on error
  err => console.warn(err)
);


// We can still load the file after refreshing the page!
FS.readFile("/data/file2", { encoding: "utf8" });

// This errors out since the file wasn't stored in the persistent /data folder
FS.readFile("/file1", { encoding: "utf8" });
```

Now let's make this slightly more automated.

Putting it all together

In reality, you don't want to be calling these functions manually to make sure you're syncing the file system properly. Here's what a more automated approach would look like when run outside the developer console:

```
<script type="text/javascript">
// Convenience functions to setup the file system
const fsSetup = path => {
  FS.mkdir(path);
  FS.mount(IDBFS, {}, path);
};
```

```
const fsLoad = () => FS.syncfs(true,  err => console.warn(err));
const fsSave = () => FS.syncfs(false, err => console.warn(err));

// Save changes to memory when leave the page
window.onbeforeunload = () => fsSave();

// Setup Module
var Module = {
 preInit: () => {
    // Create /myFile.txt (file.c looks for this file)
    FS.writeFile("/myFile.txt", "abc\ndef\nghi");

    // Load from memory
    fsSetup("/data");
    fsLoad();
  }
};
</script>
<script src="file.js"></script>
```

Note

You may have noticed that we used `preInit` here instead of `onRuntimeInitialized`. It's always advisable to perform file system operations in `preInit`, and we'll see a good example of why that is in the section below on Pre-loading Files at Compile Time.

Note

I usually write convenience functions to call `FS.syncfs()` because I often mix up whether the first parameter should be set to true or false.

Now load this page in the browser and we can read the file already without worrying about setting up our file system:

```
// Outputs "some contents"
FS.readFile("/data/file2", {encoding: "utf8"})
```

After we modify the file and refresh the page:

```
FS.writeFile("/data/file2", "new contents");
location.reload();
```

Reading the file again will give us its updated value!

```
// Outputs "new contents"
FS.readFile("/data/file2", {encoding: "utf8"})
```

Pre-loading files at compile-time

Another approach to persisting files on the file system is to pre-load them at compile time. For example, let's create a file called `somefile.txt` in the same folder as `file.c` containing:

```
this is
a test
```

Now we can tell Emscripten that we would like this file to be mounted at runtime:

```
$ emcc file.c \
    -o file.js \
    --preload-file somefile.txt
```

Next, open your browser to file.html and type the following in the console to see the contents of the file we pre-loaded:

```
// Read the pre-loaded file
FS.readFile("/somefile.txt", { encoding: "utf8" });
```

> **Note**
>
> In addition to having Emscripten generate `file.js` and `file.wasm`, we now have `file.data`, which contains the contents of the file we preloaded.

By default, pre-loaded files are simply mounted to `/`, but we can be more specific about where the file should be saved to, and under which name:

```
$ emcc file.c \
    -o file.js \
    --preload-file somefile.txt@/tmp/myfile.txt
```

Now we can read the file from `/tmp/myfile.txt`:

```
// Read the pre-loaded file
FS.readFile("/tmp/myfile.txt", { encoding: "utf8" });
```

Recall that in the previous section, we mounted an IDBFS file system onto the folder `/data`. What happens when we pre-load files to the same `/data` folder? Let's give it a try!

```
$ emcc file.c \
    -o file.js \
    --preload-file somefile.txt@/data/myfile.txt
```

If you open your browser to `file.html` and list the contents of /data, you'll notice that the IDBFS and the pre-loaded files play well together:

```
> FS.readdir("/data")
  (4) [".", "..", "file2", "test.txt"]
```

BUT that is only because in our HTML file, we mounted the IDBFS file system inside of `Module.preInit` instead of `Module.onRuntimeInitialized`.

If we modify our HTML file to mount the file system inside `Module.onRuntimeInitialized`, we get an error saying that the IDBFS file system can't be mounted because the "File exists" already since the pre-loaded file was loaded into the file system at `/data`. So using `Module.preInit` ensures that the mounting of the file system happens before pre-loaded files are mounted.

Warning

Use `Module.preInit` instead of `Module.onRuntimeInitialized` for initializing file systems.

Mounting File objects instead of strings

So far we've only created files by passing their content as strings to a function, which is inconvenient for larger files. In Chapter 10, we'll see how we can mount files directly to the virtual file system.

Chapter 10. WebAssembly + WebWorkers

Introduction

As you might have noticed, the biggest issue with using this virtual file system is that we have to store the contents of a file in a string before we can mount it. This is extremely inconvenient, especially when all we want to do is to sample some data from a very large file provided by the user.

Ideally, we would like to allow the user to specify a file from their computer and mount it directly to the file system without loading the whole shebang into a string, but instead by providing the filesystem with a `File` or `Blob` object.

WebWorkers Recap

In a nutshell, a WebWorker is a way to execute some JavaScript code in a background thread on your browser, which can help improve your application's responsiveness. Here, we make use of WebWorkers not only to maintain responsiveness, but also to mount a File or Blob object directly to the virtual file system, without having to read it all into a string first.

A simple example

Recall that our virtual file system will mount a `MEMFS` file system to the root folder `/`. As we saw in Chapter 9, we can persist the file system by mounting a `IDBFS` file system to /data. Here, we'll use a third type of virtual file system, `WORKERFS`, that only runs inside WebWorkers.

Also note that since we'll be using the file system inside the worker, we'll need to initialize the WebAssembly module within the worker.

Here's what our HTML file `worker_blobs.html` looks like:

```
<script type="text/javascript">
// Launch worker
var worker = new Worker("worker_blobs.js");

// Catch messages received from worker
worker.onmessage = msg => {
  console.log("[Main thread] Got message back:", msg.data);
}

// Send a Blob object to the WebWorker
var blob = new Blob(['blob data']);
worker.postMessage(blob);
</script>
```

Here, we simply launch the WebWorker, create a mock Blob object, and pass it to the worker. That's the easy part—the tricky part is defining the WebWorker's Javascript file, `worker_blobs.js`. That file will now initialize the WebAssembly module, mount a `WORKERFS` file system, and create a file based on the Blob object it received:

```
// worker_blobs.js
var myBlob = null;

var Module = {
  // Don't auto-run main()
  noInitialRun: true,

  // When done initializing
  onRuntimeInitialized: () =>
  {
```

```
    // Set up the virtual file system inside the worker
    FS.mkdir('/data');
    FS.mount(WORKERFS, {
      // This is where the list of File objects go
      files: [],
      // This is where the list of Blob objects go
      // Since Blobs don't have names (unlike Files),
      // we need to explicitly define one
      blobs: [{
        name: "blob.txt",
        data: myBlob
      }]
    }, '/data');

    // Output contents of file blob.txt, which points to the Blob object
    fileContents = FS.readFile("/data/blob.txt", { encoding: "utf8" });
    console.log("File contents:");
    console.log(fileContents);

    // Done
    self.postMessage("Got it");
  }
};

// Process messages sent to this worker
// (we represent the scope using self instead of window)
self.onmessage = msg => {
  myBlob = msg.data;
  // Import a WebAssembly module we used previously for file manipulation
  self.importScripts("file.js");
}
```

To summarize:

1. Once the Worker receives the Blob from the main thread, it saves it to the global variable `myBlob`, so it can be mounted to the file system later on.
2. We use the special syntax `self.importScripts()` to import JavaScript the WebAssembly module `file.js` within the WebWorker. Note that we wrote `file.js` back in Chapter 6 for file management (we can't just use `hello.js` since we need access to Emscripten's virtual file system functionality).

3. Next, the `Module.onRuntimeInitialized()` function is called, which initialized a `WORKERFS` file system with the Blob object saved as `/data/blob.txt`.
4. We use the `FS.readFile()` function we saw in Chapter 6 to read the file contents.

Mount a File object

So far, we've seen how to mount Blob objects that we defined ourselves. Now let's ask the user to select a file from their computer, which we will pass to the WebWorker, and mount to the file system.

Here's what the `worker_files.html` will now look like:

```
<input type="file" id="upload" multiple>

<script type="text/javascript">
window.onload = function()
{
  // Worker setup
  var worker = new Worker("worker_files.js");
  worker.onmessage = msg => {
    console.log("[Main thread] Got message back:", msg.data);
  }

  // Send file to WebWorker
  document
    .getElementById('upload')
    .addEventListener('change', function(e){
      worker.postMessage(this.files)
    });
}
</script>
```

This looks very similar to the Blob version, except that we added a file input. As soon as the user selects one or more files from their computer, we send the corresponding File objects to the WebWorker, where we mount them to the file system. Here's what that looks like (in the `worker_files.js` file):

```
var myFiles = null;

var Module = {
  // Don't auto-run main()
  noInitialRun: true,
  // When done initializing
```

```
  onRuntimeInitialized: () =>
  {
    // Set up the virtual file system inside the worker
    FS.mkdir('/data');
    FS.mount(WORKERFS, {
      // This is where the list of File objects go
      files: myFiles,
      // No Blobs this time
      blobs: []
    }, '/data');

    // List available files
    console.log(FS.readdir("/data"));

    // Done
    self.postMessage("Got it");
  }
};

// Process messages sent to this worker
// (in a WebWorker, we represent scope using self instead of window)
self.onmessage = msg => {
  myFiles = msg.data;
  myFiles.name = "file.txt";
  console.log(myFiles);

  // Import a WebAssembly module we used previously for file manipulation
  self.importScripts("file.js");
}
```

Launch `worker_files.html`, open the console, and verify that files you select are properly shown as mounted.

A useful application of this is to take a file as input from the user, take a chunk of it (e.g. first/last N megabytes or randomly sample N megabytes), mount just that portion to the file system, and finally execute a command line program where one of the arguments is that file.

Here's an example of how this can be done:

```
// Given a File object, extract first 10 MB of the file
// .slice() will convert the File object to a Blob object
```

```
var blob = file.slice(0, 10*1024*1024);
```

Once you have the Blob object, you can simply mount it to the file system as we've shown earlier. We now have a way to parse user files without first reading them into strings!

Chapter 11. Conclusion

This marks the end of the book, but certainly not the end of your (or my) learning journey with WebAssembly. As we've seen, it's no easy feat to develop with WebAssembly and the toolchain around it, but my hope is that this book serves you well as a stepping stone for building your own software with WebAssembly.

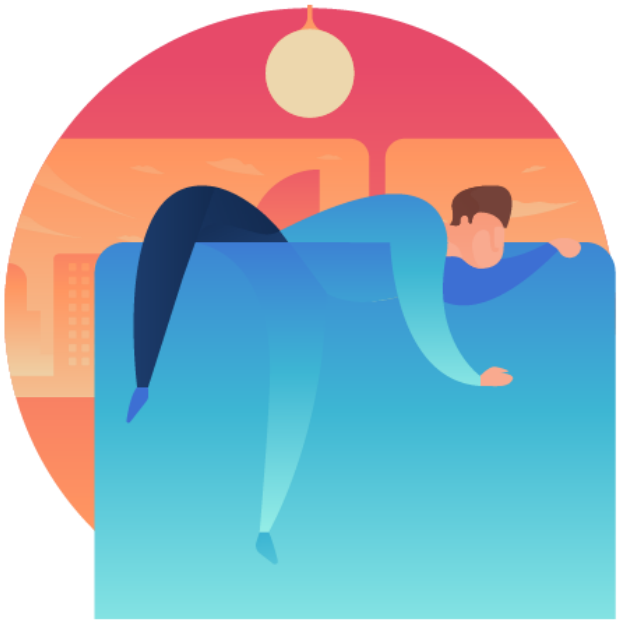
I hope I've been able to convince you that WebAssembly is an incredibly powerful tool that opens the door to doing things that were not imaginable a few years ago. The combination of portability, speed, and security are what makes WebAssembly an extraordinary tool. At the same time, despite my overwhelming excitement about WebAssembly and the future it holds, I hope that a healthy dose of moderation also shined through. Although a powerful tool, it should be used with care, so that WebAssembly does not become the proverbial hammer.

And speaking of the future, this book mostly covers WebAssembly features that have been released as part of the WebAssembly MVP. There's lots more to come in the coming months and years, including support for threads and SIMD instructions (i.e. single instruction, multiple data), the ability to access and manipulate the DOM (via host bindings), support for other languages than C/C++/Rust (via garbage collection support), the ability to load WebAssembly code the same way we load JavaScript modules (via the `import` keyword), and more.

Finally, I'll leave you with a few additional resources that I have found to be useful. I hope you will join me in exploring the wondrous world of WebAssembly:

- Articles / Talks
  - A cartoon intro to WebAssembly, by Lin Clark
  - WebAssembly Demystified, by Andre Weissflog
  - A comparison of how JavaScript and WebAssembly work, by Alexander Zlatkov
  - WebAssembly's post-MVP future, by Lin Clark, Till Schneidereit, and Luke Wagner
  - Bytecode Adventures with WebAssembly and V8, by Deepti Gandluri
- Repositories
  - Emscripten
- Tools
  - WasmExplorer
  - WebAssembly Studio
- Communities
  - Emscripten Google Group
  - Awesome WASM, a curated list of WebAssembly resources
  - WASM Weekly, a newsletter with weekly news about WebAssembly
- Documentation
  - Emscripten documentation, a vast but useful resource
  - W3C WebAssembly Specification, detailed specifications; interesting but most likely TMI

Appendix. Troubleshooting



[Docker: Cannot connect to the daemon]
[Docker: No such container]
[Docker: Mounts denied]
[Docker: Reuse after computer restart]
[Docker: dpkg was interrupted]
[Docker: make and modification times in the future]
[Browser: Incorrect response MIME type]
[Emscripten: fatal error: GLES2/gl2.h]
[Emscripten: Updating]

Docker: Cannot connect to the daemon

If you get the following error while running Docker:

```
Cannot connect to the Docker daemon. Is the docker daemon running on this host?
```

This means that your Docker Desktop app is not running. All you have to do is to launch the application, which may take a few minutes to initialize.

Docker: No such container

If you're using docker exec to launch emcc within a container to compile your code and you see this error:

```
Error response from daemon: No such container: wasm
```

This means your container is no longer running. If you followed the instructions in Chapter 2, you should be able to start up the container again by simply doing:

```
$ docker start wasm
```

Docker: Mounts denied

If you're trying to launch the wasm docker container and come across this error:

```
docker: Error response from daemon: Mounts denied:
The path <some path here>
is not shared from OS X and is not known to Docker.
```

This means that Docker doesn't have permission to mount the folder of interest. To enable that, click on the Docker Desktop icon, go to Preferences → File Sharing tab. Add the folder of interest to the list and restart Docker to apply the changes.

Docker: Reuse after computer restart

After restarting your laptop, you may find that docker ps gives you an empty output. If that's the case and you already followed the instructions from Chapter 2, try launching the container by doing:

```
$ docker start wasm
```

That should output "wasm" on the screen. If not, revisit Chapter 2 to set up the container.

Docker: dpkg was interrupted

When running the `apt-get install` command, if you see this error:

```
E: dpkg was interrupted, you must manually run 'dpkg --configure -a' to correct
the problem.
```

Run `dpkg --configure -a`, then re-run your original command.

Docker: make and modification times in the future

When running `emmake`, if you notice a warning that looks like this:

```
make: Warning: File 'makefile' has modification time 3987 s in the future
```

This is most likely because your Docker container is out of sync with the host (I've noticed this issue arises when I put my laptop to sleep). To resolve this, restart Docker Desktop, and start the container once Docker Desktop is ready (`docker start wasm`).

Browser: Incorrect response MIME type

When you run a simple HTTP server in Python using:

```
$ python -m SimpleHTTPServer 12345
```

You may see this error show up in your browser's developer console:

```
wasm streaming compile failed: TypeError: Failed to execute 'compile' on
'WebAssembly': Incorrect response MIME type. Expected 'application/wasm'.
```

This is because the browser is requesting a file with mime-type `application/wasm`, which isn't supported out of the box in Python's `SimpleHTTPServer`, hence the error.

To address that, let's create a file `launch.py` that we'll use to launch our web server, and define the mime-type `application/wasm`:

```
#!/usr/bin/python

import SimpleHTTPServer, SocketServer

# Choose a port number for the web server
PORT = 12345

# Define "application/wasm" as a mime-type when the server delivers a .wasm file
class Handler(SimpleHTTPServer.SimpleHTTPRequestHandler):
      pass

Handler.extensions_map[".wasm"] = "application/wasm"

# Launch server
print("Launching server on port {}...".format(PORT))
httpd = SocketServer.TCPServer(("", PORT), Handler)
httpd.serve_forever()
```

Next, run chmod +x server.py, and you can now launch the server by doing ./server.py. If you followed the instructions in Chapter 2, you will not need to do this.

Emscripten: fatal error: GLES2/gl2.h

If you get the error:

```
./Common/esUtil.h:23:23: fatal error: GLES2/gl2.h: No such file or directory
```

You're just missing a library. Login to your container and install the missing libraries:

```
apt-get install -y libgles2-mesa-dev
```

Emscripten: Updating

Follow the instructions from the Emscripten website:

```
# Fetch code (or git pull if not first time)
git clone https://github.com/emscripten-core/emsdk.git
cd emsdk

# Install latest emsdk
./emsdk install latest
./emsdk activate latest
source ./emsdk_env.sh
```