

+ 22

The Ride with WebAssembly

Taking Your C++ and Going Places

NIPUN JINDAL & PRANAY KUMAR



20
22

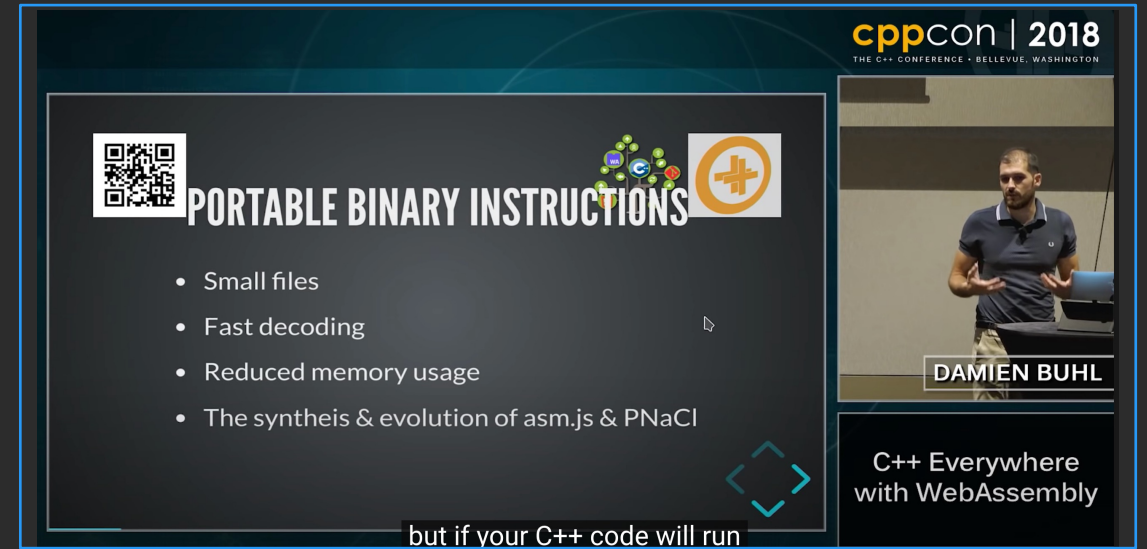
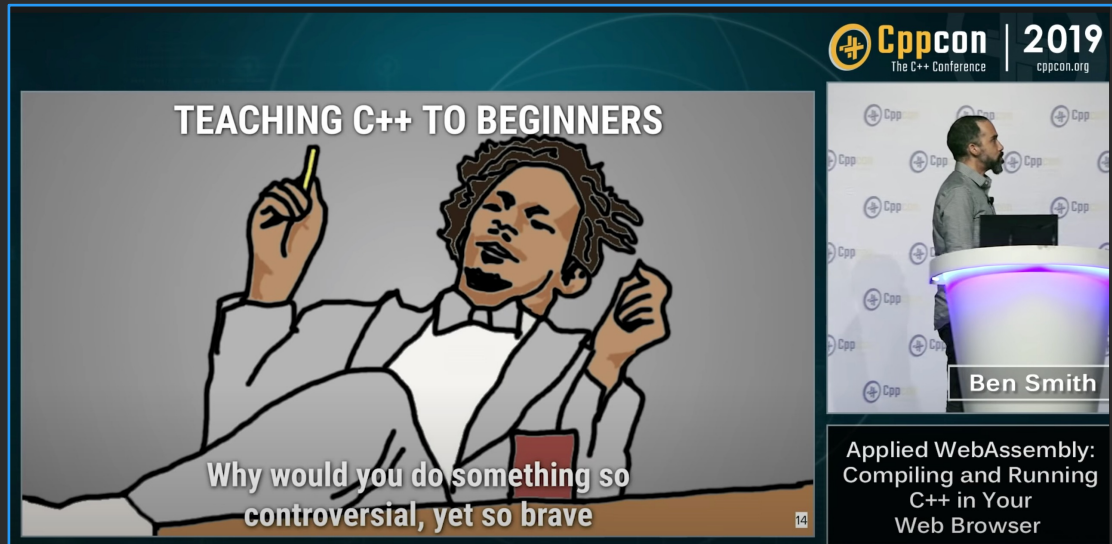


Why this talk?



WebAssembly: CppCon presence

- Compiling and Running C++ in Your Web Browser - Ben Smith - CppCon 2019
- C++ Everywhere with WebAssembly - Damien Buhl - CppCon 2018



WebAssembly: Recent advancements

- Debugging using DWARF symbols
- Loads of WebAssembly features supported in other browsers (such as Safari)
- SharedArrayBuffer usage to share memory between WebAssembly threads.
- Fixed-Width SIMD (Single Instruction, Multiple Data is a type of parallel processing)
- WASM Exception handling

2020

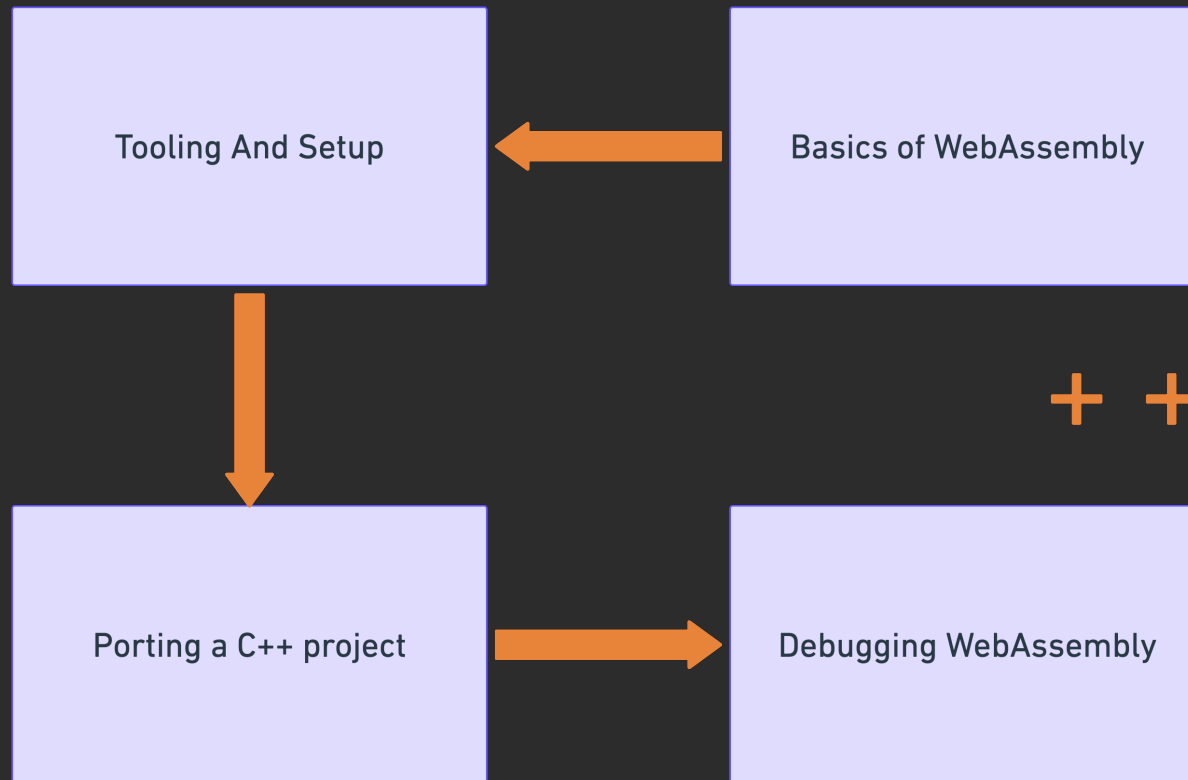


2022

Agenda



Agenda



...and more



Hands on session repo

<https://github.com/nipunjindal/cpp-con-wasm>

Includes source code and examples, docker readme and usage



Basics of WebAssembly



Basics of WebAssembly: What it is and why it's needed for you?

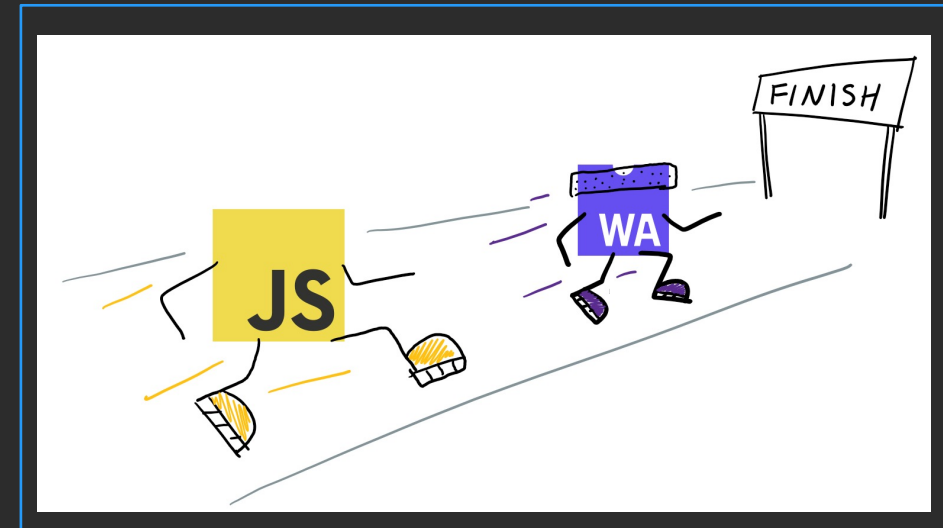
- WebAssembly (abbreviated *Wasm*) is a portable binary format accepted by all major browsers.
- It is designed as a portable compilation target for programming languages, enabling deployment on the web for client and server applications.



- To ship your library/code across the entire web stack, WebAssembly is your ticket.
- WASM was created to support Javascript, not to replace it.

Basics of WebAssembly: What makes it fast compared to JS?

- Compact: Faster to fetch as it's binary
- No parsing needed
- Closer to machine code
- No two-stage compilation
- No GC needed



Basics of WebAssembly: Advantages



Efficient and fast

- WASM stack machine is encoded in a size and load-time-efficient binary format.
- Near native speed using the cross-platform hardware capabilities.



Open and debuggable

- Assembly pretty-printing for better debug/test.
- Some browsers provide source debugging.



Pluggable

- Modules are pluggable, portable, can call into and out of the JavaScript context.



Safe and innovative

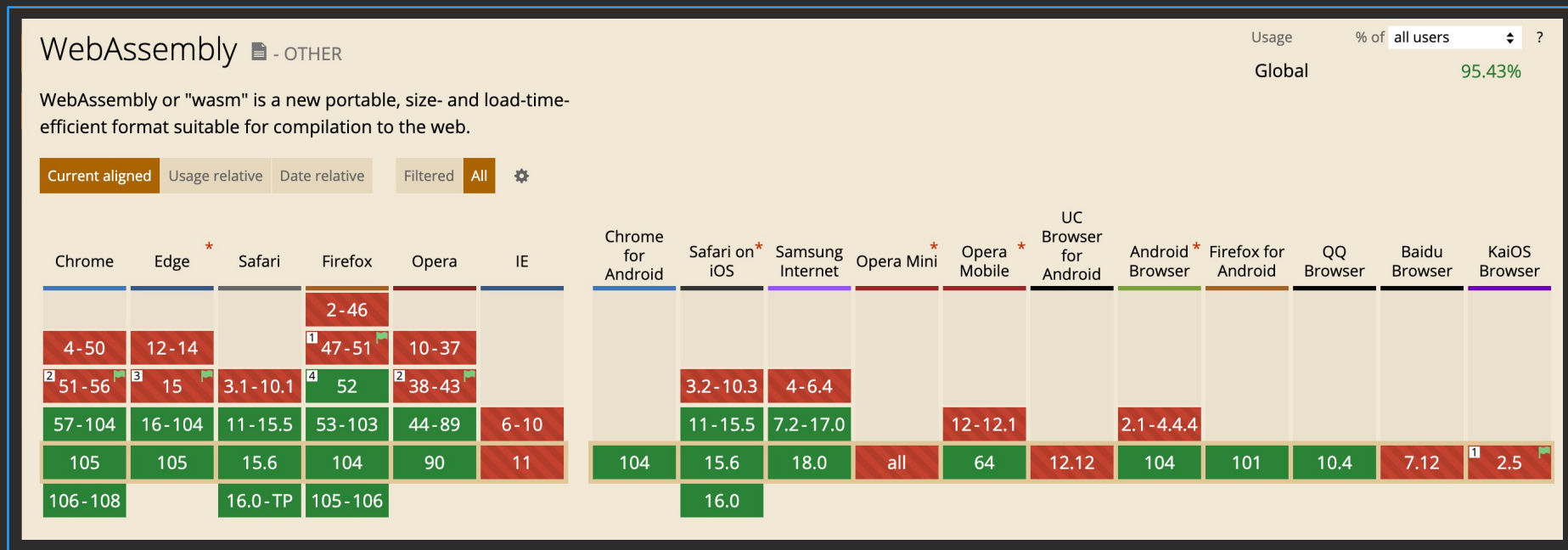
- Enforces the same-origin, permissions policies.
- Memory-safe, sandboxed execution env.

Basics of WebAssembly: What would I need in this ride?

- Just like a typical dev environment, WebAssembly has the two main pillars -
 - An execution Target (Browsers / node.js environment/ etc.)
 - As Chrome supports most of the WebAssembly features and is the most widely used browser, we have used chrome as the execution target.
 - A compilation toolchain (Just like gcc/clang)
 - Emscripten is most popular and actively developed toolchain for WebAssembly (can be assumed as a thin wrapper based on LLVM, just like gcc/clang) hence emscripten has been used as toolchain for the purpose of this talk.
- Which features are supported and would work well on runtime, depends on the above two.

Basics of WebAssembly: Check the support?

- Browser support can be checked at caniuse.com/wasm
- Emscripten has fantastic documentation <https://emscripten.org/docs>



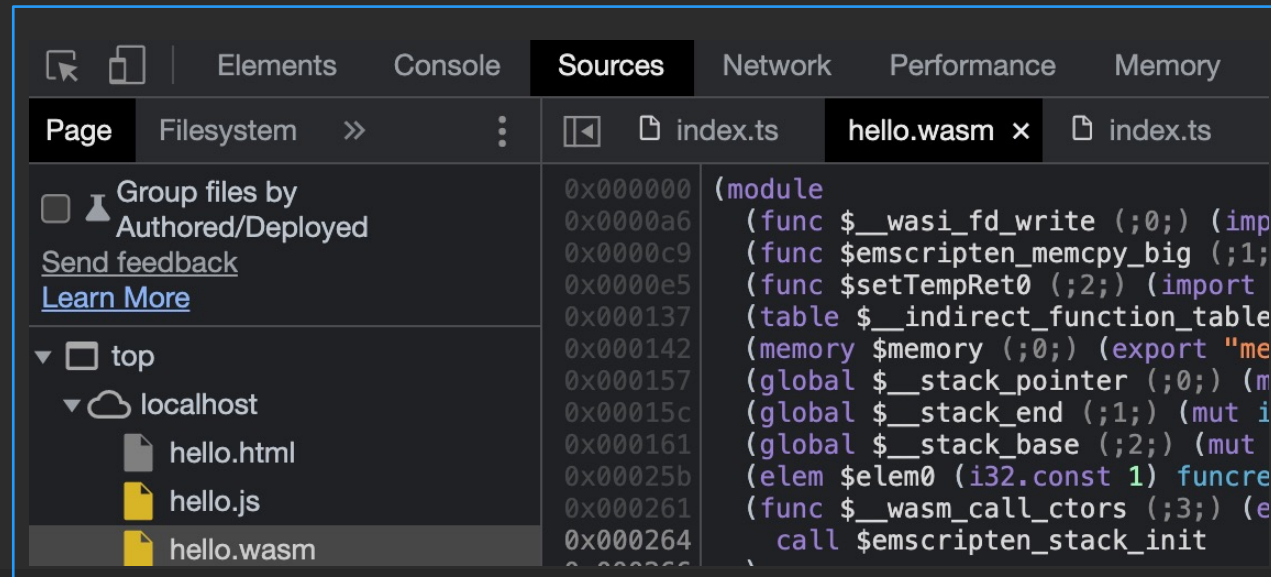
Basics of WebAssembly: Notable project to web via Emscripten

- SQLite (SQLite compiled to JavaScript with an easy-to-use API (through [sql.js](#))
- FreeType (TrueType font rendering in JavaScript, using FreeType)
- Ffmpeg (Audio/video encoder, the famous one!)
- Unreal Engine 4
- Unity engine
- Doom 3 ([Link](#))
- VIM ([Link](#))



Basics of WebAssembly: WAT and WAST

- WebAssembly is designed with the web's openness in mind, hence a text format equivalent of the binary format always exists, a.k.a WAT which is dumped by the compiler as an IR.
- When you try to debug any wasm in browser, it shows WAT by default for readability.
- WAST is a superset of the WebAssembly text format and not officially in the spec but it is used only for testing purposes.



Tooling and setup



Tooling Introduction and setup: Emscripten toolchain and compilation

- A complete open-source compiler toolchain to WebAssembly, compiles C and C++ code, or any other language that uses clang/LLVM.
- Supports various environments such as web, node.js, shell, service workers.

```

# Get the emsdk repo
git clone https://github.com/emscripten-core/emsdk.git

# Enter that directory
cd emsdk

# Fetch the latest version of the emsdk (not needed the first time you clone)
git pull

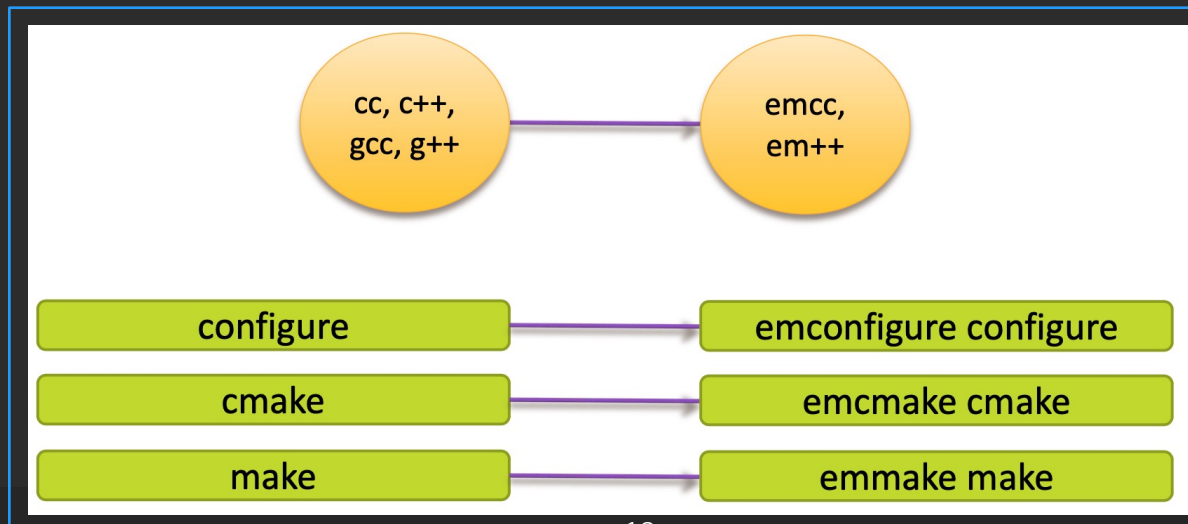
# Download and install the latest SDK tools.
./emsdk install latest

# Make the "latest" SDK "active" for the current user. (writes .emscripten file)
./emsdk activate latest

# Activate PATH and other environment variables in the current terminal
source ./emsdk_env.sh # use .bat for windows
```

Tooling Introduction and setup: Emscripten toolchain and compilation

- emcc/em++ is toolchain used to compile C++ code to wasm.
- Thin wrapper over LLVM.
- It has very similar options as gcc/clang and offers option for optimization, debugging etc.
- You can control output options as well.
- Separate flags may be required for features, such as exceptions, filesystem, network etc.



Tooling Introduction and setup: Sample configurations

Separate flags required for features, such as exceptions, filesystem, network etc. unlike gcc/clang.


- `-s TOTAL_MEMORY=1024MB`
 - Sets to 1GB memory (includes both stack and heap space) (Default is 16MB)
- `-s ALLOW_MEMORY_GROWTH=1`
 - Allows the total amount of memory used to change depending on the application demand.
- `-s DISABLE_EXCEPTION_CATCHING=0 or 1`
 - Catching C++ exceptions (specifically, emitting catch blocks) is turned off by default in `>= -O1`
 - To re-enable exceptions in optimized code, run with `-sDISABLE_EXCEPTION_CATCHING=0`
- `-sFETCH`
 - To include the network layer (Fetch API)

Tooling Introduction and setup: Sample configurations (cont..)

- `-pthread, -sUSE_PTHREADS=1`
 - Includes the threading support.
 - Pthreads + memory growth (`ALLOW_MEMORY_GROWTH`) is especially tricky and has few open issues.

Tooling and setup: Verifying toolchain and running emscripten

- Verifying the setup



```
./emcc -v
```

- Write a sample “*Hello world*” program.



```
#include <stdio.h>

int main() {
    printf("hello, world!\n");
    return 0;
}
```

Tooling and setup: Verifying toolchain and running emscripten

Running via JS glue layer (say in Node.js)

```
./emcc test/hello_world.c -o hello.js
```

Output files

hello.js

hello.wasm

You can directly consume wasm as well through your own custom JS glue layer

hello.js is the JS wrapper layer that loads *hello.wasm*

Running in browser directly

```
./emcc test/hello_world.c -o hello.html
```

Output files

hello.js

hello.wasm

hello.html
(Created by emsdk just for trying out hello.js in browser)

Tooling and setup: Verifying toolchain and running emscripten (cont..)

Running in Node.js

```
● ● ●  
$ node hello.js  
  
hello, world!
```

hello.js

hello.wasm

```
● ● ●  
<html>  
  <body>  
    <script src="hello.js"></script>  
  </body>  
</html>
```

You can create a custom html for loading *hello.js* in browser yourself

Running in browser directly



Loaded html in browser using a local webserver

hello.js

hello.wasm

Tooling and setup: Optimizing code

- Emscripten, like gcc and clang, generates unoptimized code by default.
- Generally, you should first compile and run your code without optimizations (the default).
- Once sure that the code runs correctly, more aggressive optimization techniques can be applied to make it load and run even faster.
 1. Code is optimized by specifying optimization flags when running emcc. The levels include: -O0 (O-zero i.e. no optimization), -O1, -O2, -O3, -Os, and -Oz.
 2. -Os, and -Oz focus on the code bundle size reduction
 3. -O3 is a generally a good setting for a release build as it optimizes for speed.
 4. First time setup and compilation may take time as wasm system libraries are generated and cached

Tooling and setup: System libraries and pre-ported deps

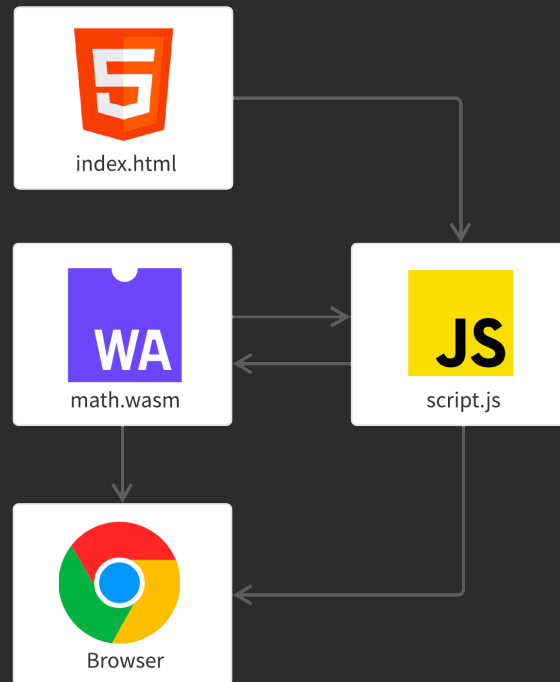
- Most native executables link against a system library (libc, libcxx, and so on)
- Emscripten provide its own implementation of system libraries (stdio, filesystem) to be linked.
- This implementation is a hybrid model based on WebAssembly System interface (wasi) and JS.
 - WASI is a system API interface (ABI and API) designed by Mozilla intended to be portable to any platform.
- Apart from system libs, standard pre-ported deps like boost, SDL are ready to use as well.

Calls interaction across layers



Calls interaction across layers: Interact with DOM and javascript

- WebAssembly compiled code interaction to \Leftrightarrow from JS is possible through various options available.
- Remember, WASM cannot interact with DOM directly. So, we need to use both JavaScript and WASM for practical applications.



Calls interaction: Three methods for Interacting with JS from C++

1. Using `emscripten_run_script()` (simply write inline javascript)

```
emscripten_run_script("alert('hi')");
```

2. Using `EM_JS()` (faster and preferred)

```
EM_JS(returnType, functionName, (), {  
    /*js code here*/  
});
```

Calls interaction: Three methods for Interacting with JS from C++(cont).

3. Using EM_ASM() (faster and preferred)

- You need to specify if the return value is an int, double or pointer type using the appropriate macro EM_ASM_INT, EM_ASM_DOUBLE or EM_ASM_PTR

```
int x = EM_ASM_INT({
    console.log('I received: ' + $0);
    return $0 + 1;
}, 100);
printf("%d\n", x);

// This would print 101
```

```
const std::string fileName = "test.txt";
EM_ASM(
{
    var js_file_name = UTF8ToString($0);
    var id = $1;
    // Perform file operations
},
name.c_str());
```

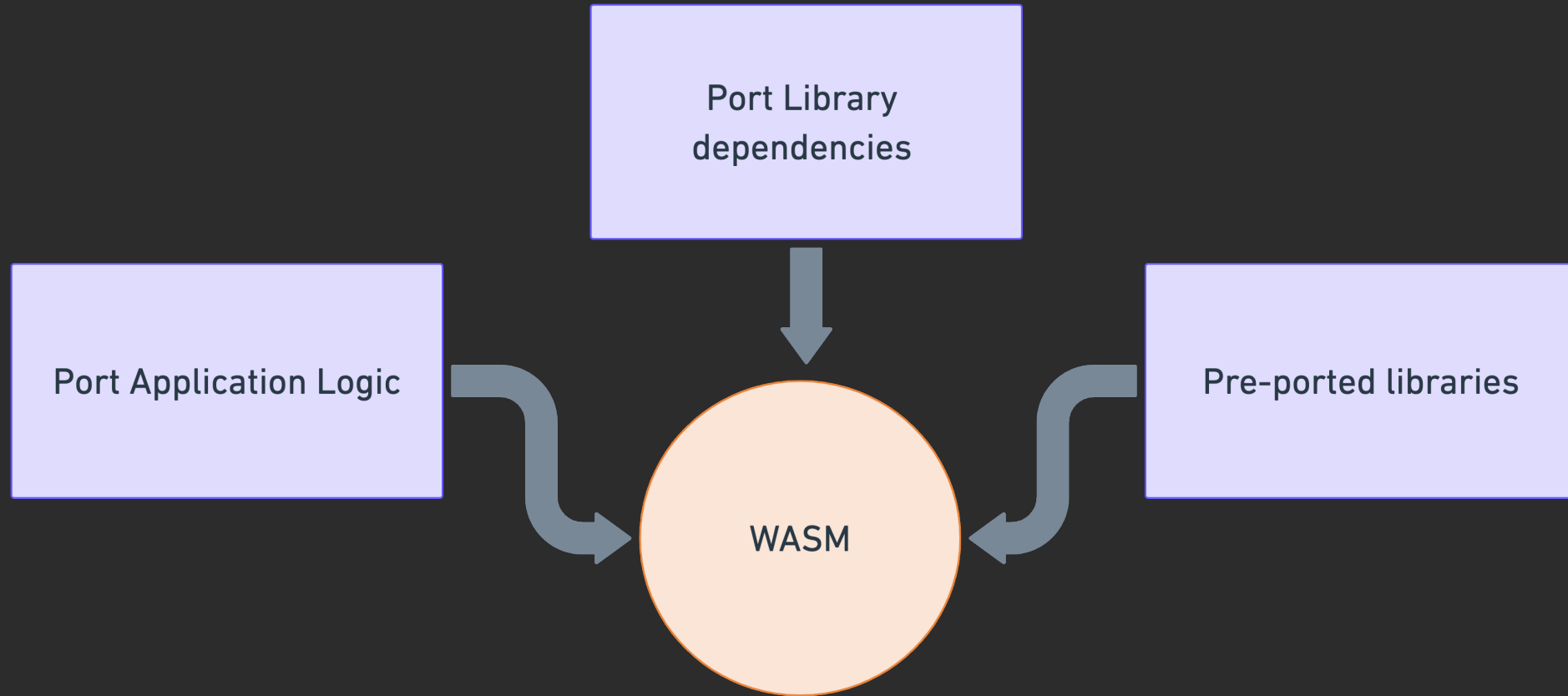
Calls interaction across layers: Interacting with the C++ from JS

- `embind` (covered with the hands on session project)
 - `struct`
 - `class`
 - `primitives`
 - `smart pointer`
 - `templates`

Porting a C++ project



Porting a C++ project



Porting a C++ project: Network layer

- Emscripten supports libc networking functions and you must use given asynchronous operations.
- Emscripten compiled applications have a number of ways to connect with online servers.
 - Websockets and POSIX Sockets supported.
 - XmlHttpRequests and Fetch API support is there.
- The Emscripten Fetch API allows make requests (HTTP GET, PUT, POST) from remote/local servers, compile with *-sFETCH* option.
- Also allows to persist the downloaded files locally in browser's IndexedDB storage, so that they can be reaccessed locally on subsequent page visits.

Porting a C++ project: Network layer (cont..)

- A sample to retrieve data file via fetch API GET request and load in memory.

```
#include <stdio.h>
#include <string.h>
#include <emscripten/fetch.h>

void downloadSucceeded(emscripten_fetch_t *fetch) {
    printf("Finished downloading %llu bytes from URL %s.\n", fetch->numBytes, fetch->url);
    // The data is now available at fetch->data[0] through fetch->data[fetch->numBytes-1];
    emscripten_fetch_close(fetch); // Free data associated with the fetch.
}

void downloadFailed(emscripten_fetch_t *fetch) {
    printf("Downloading %s failed, HTTP failure status code: %d.\n", fetch->url, fetch->status);
    emscripten_fetch_close(fetch); // Also free data on failure.
}

int main() {
    emscripten_fetch_attr_t attr;
    emscripten_fetch_attr_init(&attr);
    strcpy(attr.requestMethod, "GET"); // custom server for Cross origin request can be specified
    attr.attributes = EMSCRIPTEN_FETCH_LOAD_TO_MEMORY;
    attr.onsuccess = downloadSucceeded;
    attr.onerror = downloadFailed;
    emscripten_fetch(&attr, "myfile.dat");
}
```

Success callback

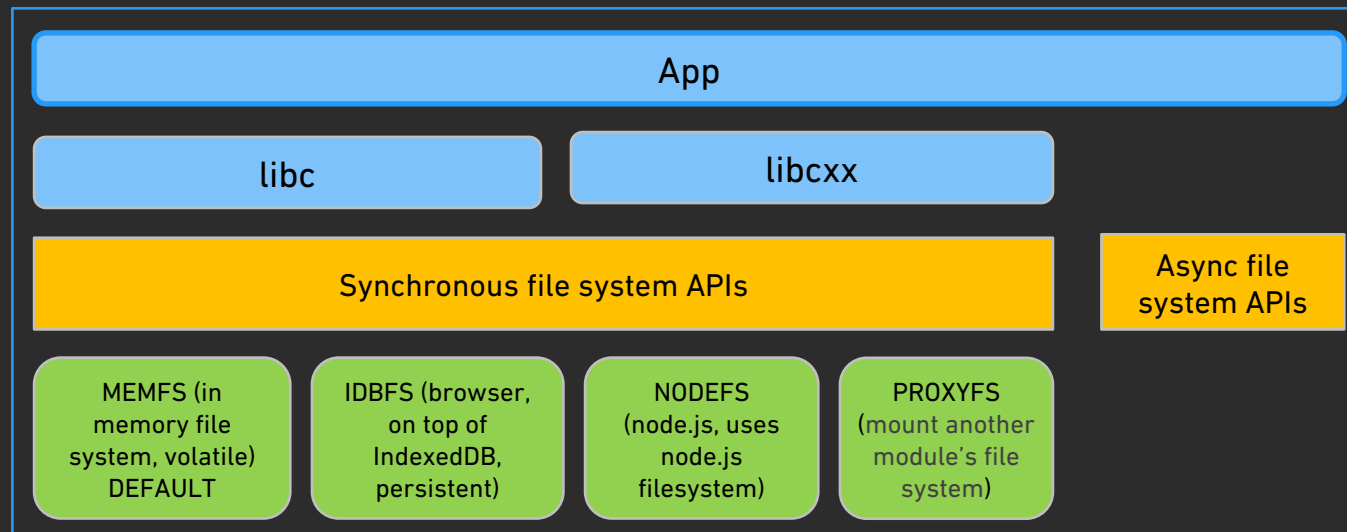
Error callback

Fetch API attributes that
need to be customize

Async Fetch API call

Porting a C++ project: Storage layer

- Code run in a browser environment is sandboxed, doesn't have direct access to the local filesystem.
- Emscripten provides a virtual file system that simulates the local file system, so that native code using synchronous file APIs can be compiled and run with little or no change.
 - May be preloaded with your specified data or linked to URLs for lazy loading.
 - Could cache your files for current session or across sessions depending upon your use case.



Porting a C++ project: Storage layer (cont..)


- Only the MEMFS filesystem is included by default. All others must be enabled explicitly, using “-lnodefs.js”(NODEFS), “-lidbfs.js”(IDBFS), or “-lproxyfs.js”(PROXYFS).
- FS object that exposes underlying filesystem interfaces, can be used for from JS as well,
- If your C/C++ code doesn't use files, but you want to use them from JS, then you can build with “-sFORCE_FILESYSTEM”, which will make the compiler include file system support even though it doesn't see it being used.

```
FS.mkdir('/persistent');
FS.mount(IDBFS, {}, '/persistent');

// sync from persisted state into memory and then
FS.syncfs(true, function (err) {
  // ...
  // Notify C++ code to read/write file under /persistent directory
});
```

Porting a C++ project: Exceptions

- By default, exception catching is disabled in emscripten. Executing a throw would abort the program and you would see similar message as below -



```
throw...  
exception thrown: 5246024 - Exception catching is disabled, this exception cannot be caught. Compile with -  
sNO_DISABLE_EXCEPTION_CATCHING or -sEXCEPTION_CATCHING_ALLOWED=[..] to catch
```

- If you want to opt-in, you have two following options.
 1. JavaScript-based exception support (Supported on all browsers/JS engines)
 - To enable it, pass `-fexceptions` at both compile time and link time.
 - Due to how WebAssembly currently implemented JS exceptions, this option can punish with high performance and size penalty.
 - You can reduce some overhead by specifying a list of allowed functions with `EXCEPTION_CATCHING_ALLOWED`

Porting a C++ project: Exceptions (cont..)

2. WebAssembly exception handling proposal (Supported on few major browsers)

- To enable it, pass `-fwasm-exceptions` at both compile time and link time.
 - This feature has built-in instructions for throwing /catching exceptions to WebAssembly.
- Practically, one should build two versions of wasm, one with exceptions enabled and the other without. By default, use the exceptions disabled (lighter build) and switch to the larger less performant one at runtime if an actual exception occurs.

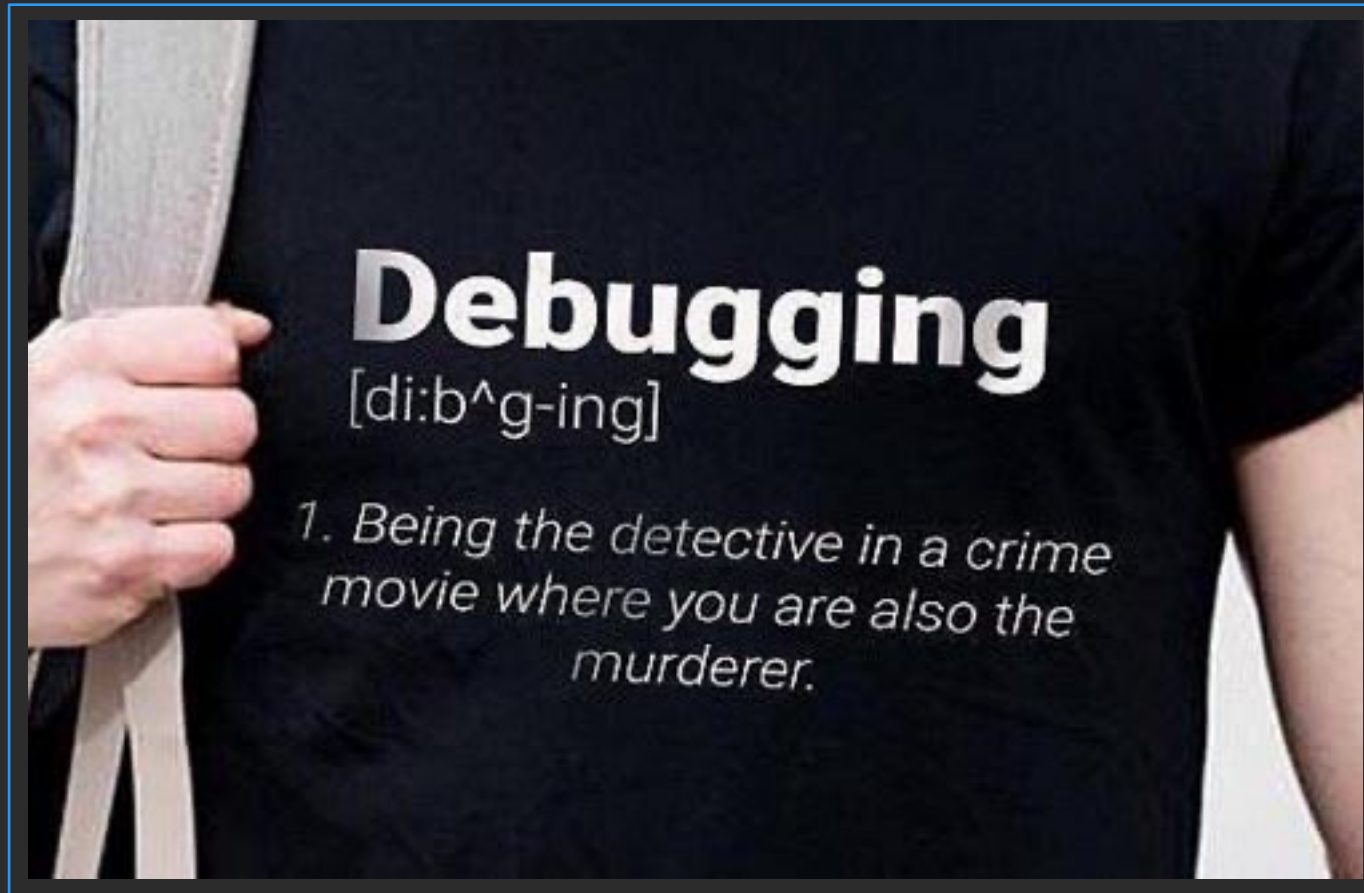
Porting a C++ project: Application Main Loop

- The browser event model uses co-operative multitasking.
- Each event has a “turn” to run, and must then return control to the browser event loop so that other events can be processed.
- A common cause of HTML pages hanging is JS that does not complete and return control.
- Hence, this can affect how an application using an infinite main loop should be written for web.
- You can specify custom event loops using *emscripten_set_main_loop_arg(..)* which can be cancelled later easily. More info at https://emscripten.org/docs/api_reference/emscripten.h.html#c.emscripten_set_main_loop

Debugging WebAssembly



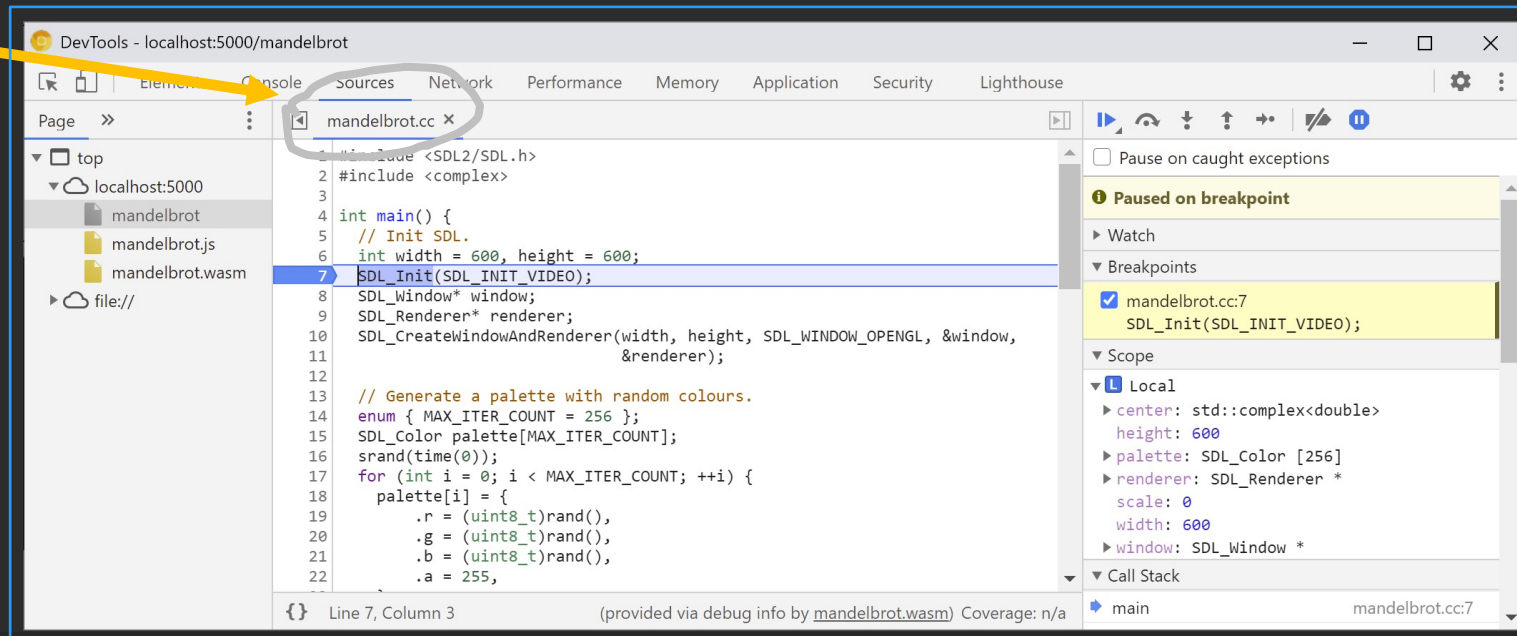
Debugging WebAssembly



Debugging WebAssembly: What and where?

- DWARF is a debugging file format used by many compilers and debuggers to support source level debugging. We can debug C++ code directly in Chrome browser (Using DWARF information).
- The `emcc -g` flag can be used to preserve debug information in the compiled output. `-g` flag be specified with an integer level: `-g0` (max optimization), `-g1` (preserve whitespace), `-g2` (preserve function names), and `-g3` (preserve everything, default level when setting `-g`)

How to achieve this magic?



Debugging WebAssembly in the Chrome browser.

Debugging WebAssembly: In Chrome using DWARF information

Step 1: Install chrome C++ DevTools extension for DWARF

Please install it by going to this link: goo.gle/wasm-debugging-extension

it helps with all the debugging information encoded in the WebAssembly file



Step 2: Enable WebAssembly debugging in the DevTools Experiments

Open Chrome DevTools > settings, go to the **Experiments** panel

Check “WebAssembly Debugging: Enable DWARF support”, then DevTools would ask for reload, go for it.



Step 3: Provide -g flag while compiling your library and source

When compiling, this the same as in *Clang* and *gcc*, it adds DWARF debug information to the object files

When linking, this is equivalent to -g3.

Debugging WebAssembly: In Chrome using DWARF information

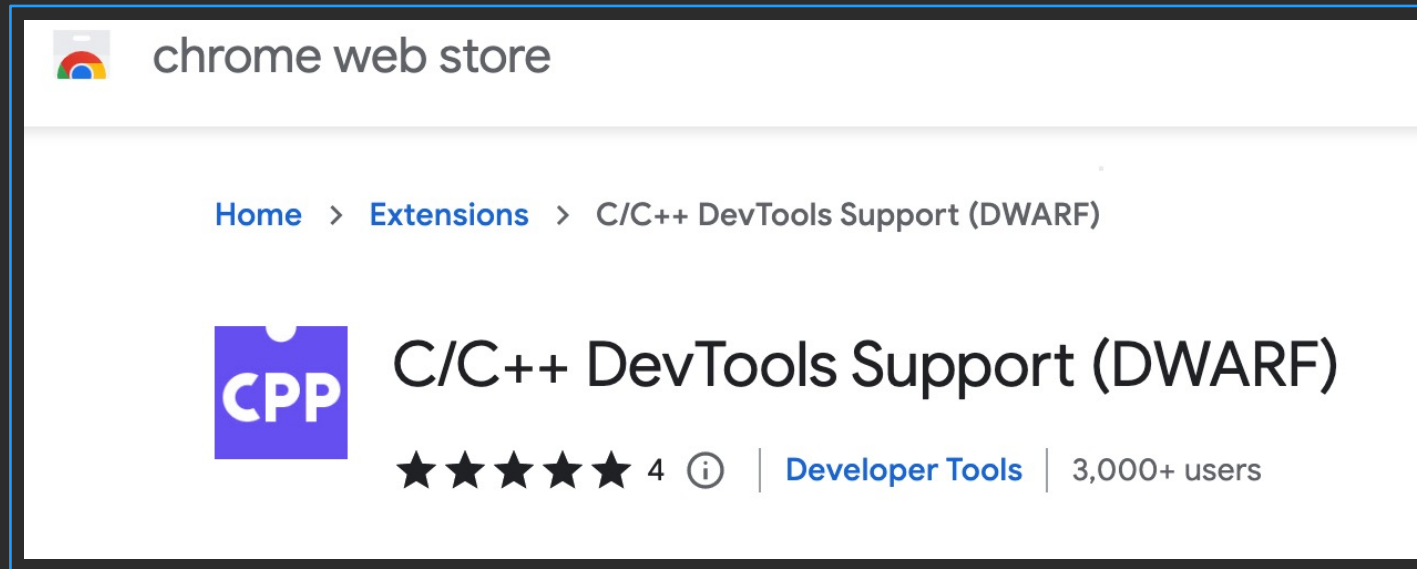


Debugging WebAssembly: In Chrome using DWARF information

Step 1: Install chrome C++ DevTools extension for DWARF

Please install it by going to this link: goo.gle/wasm-debugging-extension

it helps with all the debugging information encoded in the WebAssembly file

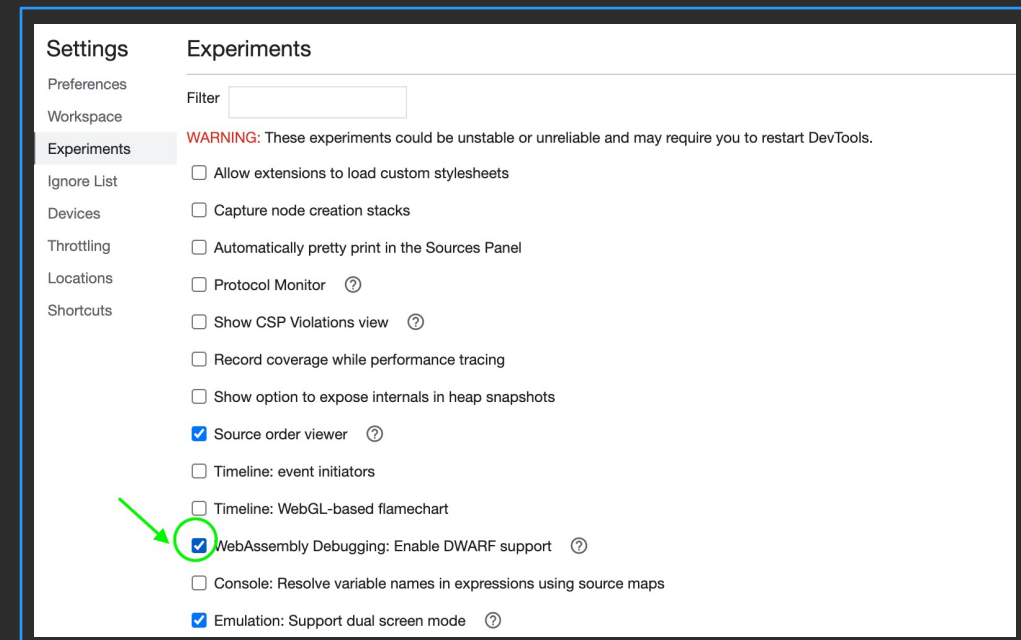
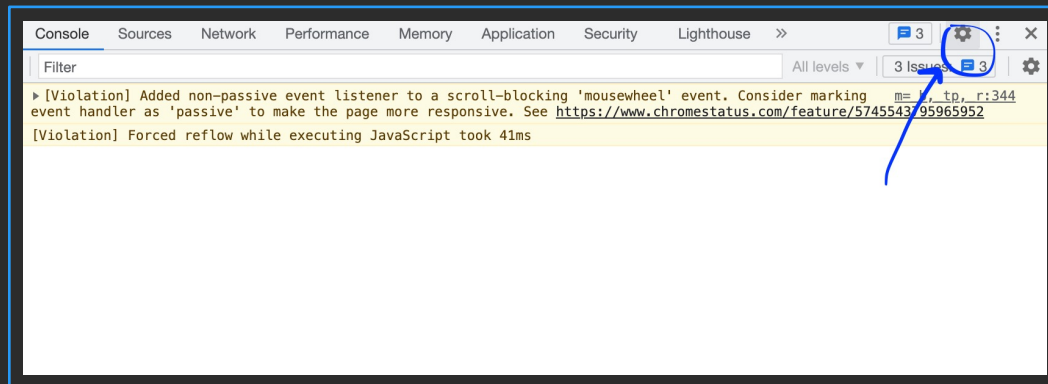


Debugging WebAssembly: In Chrome using DWARF information

Step 2: Enable WebAssembly debugging in the DevTools Experiments

Open Chrome DevTools > settings, go to the **Experiments** panel

Check “WebAssembly Debugging: Enable DWARF support”, then DevTools would ask for reload, go for it.



Debugging WebAssembly: In Chrome using DWARF information

Step 3: Provide -g flag while compiling your library and source

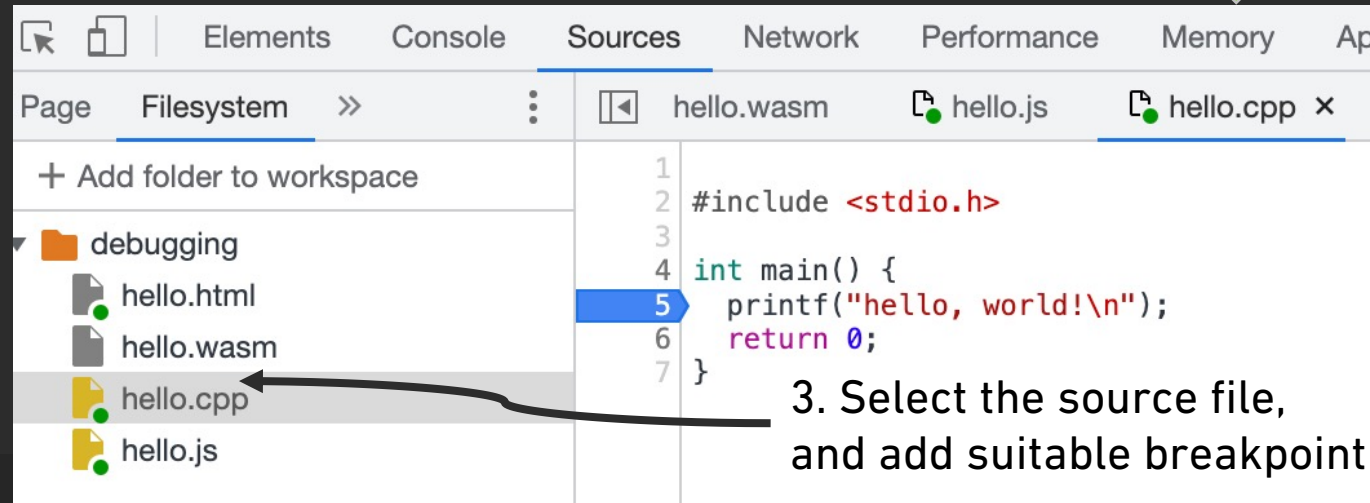
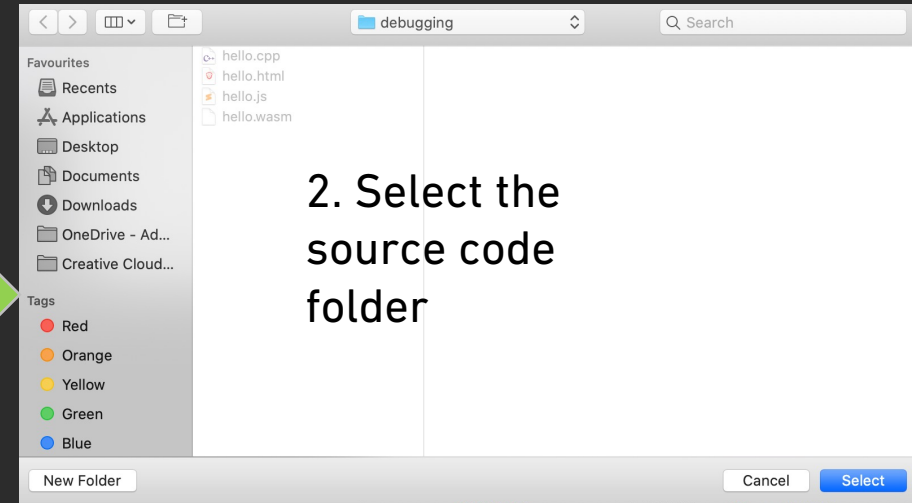
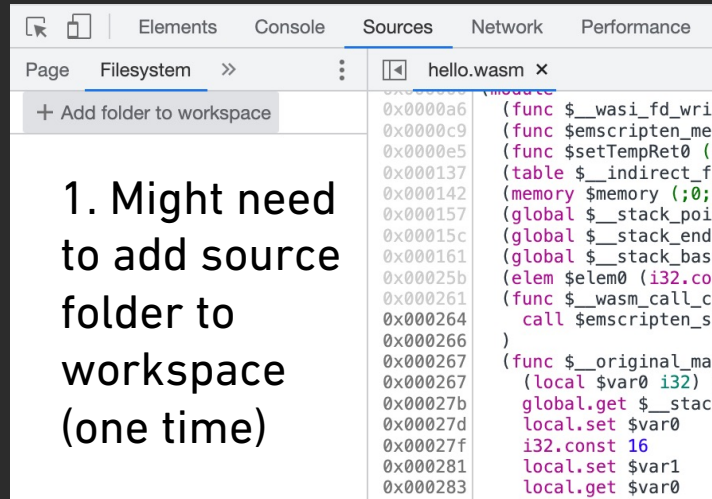
When compiling, this the same as in *Clang* and *gcc*, it adds DWARF debug information to the object files

Linking will take longer when generating dwarf symbols..



```
emcc -g hello.cpp -o mandelbrot.html
```


Debugging WebAssembly: In Chrome using DWARF information



4. Reload the web page

Debugging WebAssembly: In Chrome using DWARF information

powered by **emscripten**

Paused in debugger Running...

Resize canvas Lock/hide mouse pointer Fullscreen

Source tree

```
1 #include <stdio.h>
2
3
4 int main() {
5     printf("hello, world!\n");
6     return 0;
7 }
```

Paused on breakpoint

Watch

Breakpoints

- hello.cpp:5 printf("hello, world!\n");

Scope

Call Stack

Some call frames have warnings

main	hello.cpp:5
\$main	hello.wasm:0x2c4
(anonymous)	hello.js:1572
callMain	hello.js:2225
doRun	hello.js:2282
(anonymous)	hello.js:2293
setTimeout (async)	
run	hello.js:2289
runCaller	hello.js:2203
removeRunDependency	hello.js:1481
receiveInstance	hello.js:1661
receiveInstantiationResult	hello.js:1678
Promise.then (async)	
(anonymous)	hello.js:1707
Promise.then (async)	
instantiateAsync	hello.js:1704
createWasm	hello.js:1735
(anonymous)	hello.js:1910

XHR/fetch Breakpoints

DOM Breakpoints

Global Listeners

Voila!! Breakpoint is hit

Line 5, Column 3 (provided via debug info by hello.wasm) Coverage: n/a

Debugging WebAssembly: In Chrome using DWARF information

[illegible]

Debugging WebAssembly: Debugging optimized builds

- Like with any other languages, debugging works best if optimizations are disabled. Optimizations might inline functions one into another, reorder code, or remove parts of the code.
- Debugging with WebAssembly keep on evolving further.
- Use *-fno-inline* to disable function inlining, when compiling with any -O level optimizations.



```
emcc -g temp.c -o temp.html -O3 -fno-inline
```

WebAssembly: Traps and caution



WebAssembly: Traps and caution

- Filesystem access
 - Default filesystem is virtual and sandboxed hence each tab in a browser has its own cache.
- Memory limit
 - C++ access limited to 4GB in Chrome, 2GB in Firefox & Safari
- Zero-cost C++ Exception handling
 - Exception handling currently uses JavaScript exception handling and is very slow.
- Main UI thread blocking
 - (If not running in a web worker thread) Large loops and computation would block the main thread and you might get a browser notification to kill the page. You can specify custom event loops using *emscripten_set_main_loop_arg(..)* which can be cancelled later.
More info at https://emscripten.org/docs/api_reference/emscripten.h.html#c.emscripten_set_main_loop

WebAssembly: Traps and caution (cont..)

- Running Computation intensive heavy WebAssembly in a Web Worker
 - This keeps the main browser thread free to continue rendering and handling user interactions.
 - Message passing and handling using callback and events would be done in those such cases.
 - May pay an overhead cost for transferring any data here if it is large, but depends on your data types.

WebAssembly: Q & A



Thank you

Nipun Jindal | Sr. Computer Scientist

Pranay Kumar | Computer Scientist

Adobe Systems