WebAssembly Introduction

Overview

WebAssembly (Wasm) is register-based virtual instruction set supported by modern web browsers and other systems. Many programming languages – including C, C++, C#, and Rust – can be compiled with Wasm as the target instruction set. This allows programs written in these languages to execute on any platform supporting Wasm (including web browsers). The Wasm VM is a guest that resides on a host (the "real" computer) within host process(es) (e.g., the web browser). Setup instructions target the process to set up these tools in Ubuntu 20.04. We recommend students either "native" Ubuntu 20.04 or WSL2 Ubuntu 20.04 (on Windows 10/11).

Emscripten Tools

We'll use the Emscripten toolchain for WebAssembly. Download and install it as follows:

```
sudo apt install git curl python3
git clone
https://github.com/emscripten-core/emsdk.git
```

Before using the toolchain in a session, it should also be activated:

```
emsdk/emsdk activate
latest
```

Building Wasm Binaries

The Emscripten compiler can build source in a similar fashion to other compilers. Like standard executables, the entry point (`main`) will automatically run when loaded in a typical fashion. While Wasm binaries can be used on their own, the VM is typically isolated from the rest of the host. To simplify preparation of WASM builds, the toolchain provides a mechanism to generate "glue" code. The compiler can target three different types of output – HTML, JS, and pure WASM:

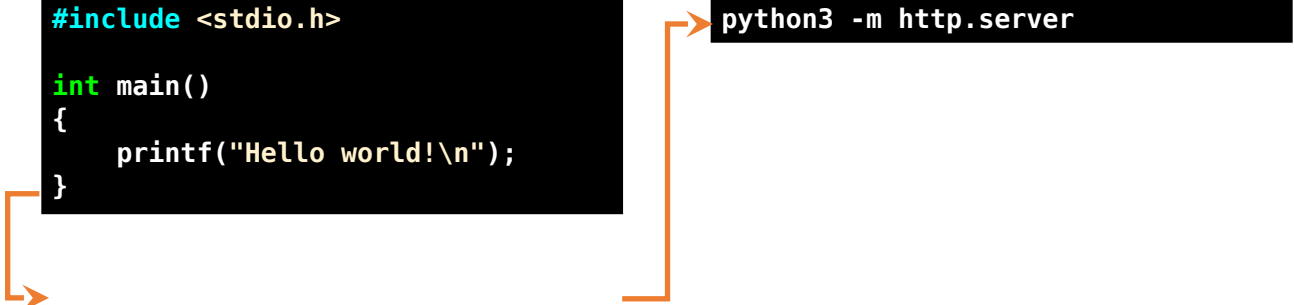| Output Generation Mode	Directive	Result
HTML: Standalone web page	`-o hello.html`	`hello.html`, `hello.js`, `hello.wasm`
JavaScript: JS "glue", Wasm binary	`-o hello.js`	`hello.js`, `hello.wasm`
Wasm: Standalone binary	`-o hello.wasm`	`hello.wasm`

For example, consider source file `hello.c`.

```c
#include <stdio.h>

int main()
{
    printf("Hello world!\n");
}
```
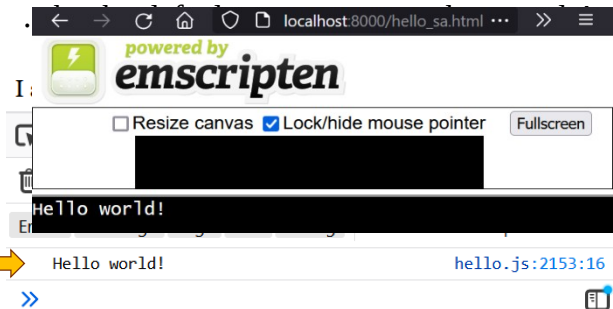
We can then run a lightweight server…

```
python3 -m http.server
```

…and see the results in a pseudo-terminal:
If built as a standalone web page (HTML)….
We can also write our own HTML file…

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

```
emcc hello.c -o hello_sa.html
```

```
emcc hello.c -o hello.js
python3 -m http.server
```

Fetch API

Wasm VM does not have access to local files by default for security reasons, so we just *fetch* data files via HTTP. By default, Wasm executes on the main thread, so fetches are non-blocking. They use initialization and callback functions to start and process downloads, respectively:

```cpp
#include <iostream>
#include <string>
#include <emscripten/fetch.h>

void allDone(emscripten_fetch_t *result)
{
  if (result->status == 200) // HTTP response 200 is "OK" (success)
  {
    std::cout << "URL: " << result->url << "; Size: " << result->numBytes << "\n";
    std::cout << "User Message: " << (char *) (result->userData) << "\n";
    std::cout << "File: " << std::string(result->data, result->numBytes) << "\n";
  }
  else
    std::cout << "Stuff's broke: [" << result->statusText << "]\n";

  emscripten_fetch_close(result); // Cleanup
}

int main() // Main initializes the fetch in this case.
{
  emscripten_fetch_attr_t request; // Fetch request attribute structure
  emscripten_fetch_attr_init(&request); // Initialize structure values
  strcpy(request.requestMethod, "GET"); // Set request method
  request.attributes = EMSCRIPTEN_FETCH_LOAD_TO_MEMORY; // Load into memory
  request.onsuccess = request.onerror = allDone; // Callback(s); can be different
  request.userData = (void*) "howdy"; // Set this to anything & retrieve
```

To include the Fetch API, add the build option:

```
em++ fetch.cpp -o fetch.js -s
FETCH
```

Note that while there are blocking (synchronous) variants of the Fetch API, they are incomplete and have many limitations. While you are free to investigate them, they are difficult to use.
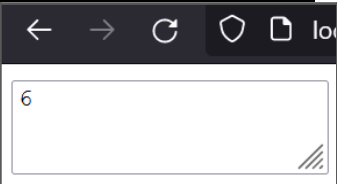
C-Style Bindings

While running Wasm directly can be useful, its power lies in being able to bind native functions to JavaScript, allowing execution to go back and forth as needed between the languages / runtimes. Exporting C-compatible API functions for binding can be done via command line arguments and/or preprocessor directives (which is a common approach for many platforms).

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

EMSCRIPTEN_KEEPALIVE
const char* greeting()
{
    return "Howdy howdy!";
}

EMSCRIPTEN_KEEPALIVE int
get_len()
{
    return strlen("Howdy
howdy!");
}

EMSCRIPTEN_KEEPALIVE
int add_me_brah(int a, int b)
{
```

OR...

```
#include <string.h>

const char* greeting()
{
    return "Howdy howdy!";
}

int get_len()
{
    return strlen("Howdy howdy!");
}

int add_me_brah(int a, int b)
{
    return a + b;
}
```

```
emcc hello.c -o hello_c.js -s \
EXPORTED_FUNCTIONS='["_greeting",
```

If exported, functions can be invoked from JavaScript.

```
<!doctype html><html><body>
  <textarea id="out"></textarea>
  <script src="hello.js"></script>
  <script>
    Module.onRuntimeInitialized = () => // Once loaded...
    {
      out = document.getElementById('out'); // get HTML e
      out.value = Module._add_me_brah(1, 5); // call func
    }
  </script>
</body></html>
```

This works well for simple / numeric functionality – but for longer chunks of memory (e.g., strings, images, and other binary data), it is necessary to bind / share memory between environments.

C++ Bindings: Embind

The C-compatible bindings have some limitations, especially when it c[...] Emscripten also includes **Embind**, a C++-specific library that allows dir[...] Document Object Model (DOM) without additional JavaScript snippets. [...] used to export C++ entities for use in JavaScript or to prepare JavaScript types that can be used when invoking JavaScript from C++. These bindings are complex, as they handle the conversion of data and functions between runtimes, but they provide powerful interoperability mechanisms.
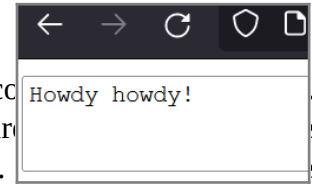
```
element = document.getElementById('output'); // find output
element
buffer = Module.asm.memory.buffer;            // ASM memory
buffer
howdy = Module._greeting();                   // Get string
ptr
                                              // Get string
```

Exporting From C++

Embind exports C++ functions, classes, and memory types while dispensing with preprocessor symbols and compile parameters for function exports:

```cpp
#include <string>
#include <iostream>
#include <emscripten/bind.h>
using namespace emscripten;

void ohai(std::string wut)
{
    std::cout << wut;
}

EMSCRIPTEN_BINDINGS(my_module
)
{
    function("ohai", &ohai);
```

```
em++ ohai.cpp -o ohai.js -lembind
```

```html
<!doctype html><html><body>
  <script src="hello.js"></script>
  <script>
    Module.onRuntimeInitialized = ()
=>
    {
      Module.ohai("Hi again");
    }
  </script>
```

Example of Embind C++ Wasm bindings, function export

```cpp
#include <emscripten/bind.h>
using namespace emscripten;

class Stufs
{
    int foo;
public:
    Stufs(int bar) : foo(bar) {}
    int getFoo() { return foo; }
};

EMSCRIPTEN_BINDINGS(my_module)
```

```html
<!doctype html><html><body>
  <script src="stuffs.js"></script>
  <script>
    Module.onRuntimeInitialized = ()
=>
    {
      var thingy = new
Module.Stufs(42);
      console.log(thingy.getFoo());
      thingy.delete(); // Necessary!
    }
  </script>
```

Importing from JavaScript

Perhaps the most powerful feature of Embind is its mechanisms to directly access and manipulate JavaScript objects – including the DOM – from C++. Embind provides value conversion between JavaScript and C++ types on the fly within C++ code, allowing JavaScript objects to be referenced and JavaScript functions and methods to be executed. Here's an example (assume "textbox" is a **TextArea** element within the body of the HTML document):

```cpp
#include <emscripten/val.h>

using emscripten::val; // "val" is the embind JS reference type.


int main() // Finds a textarea element and sets its text.
{
  val document = val::global("document"); // Get global "document" variable
  val text = document.call<val>("getElementById", val("textbox")); //
Method call
  text.set("height", val(200)); // Set attribute

  char nums[5] = { 0, 1, 2, 3, 4 };
  val view = val(typed_memory_view(5, nums)); // Create a memory view
  text.set("value", view); // Set textarea text
}
```

While a full API is available, the following functions are most common in Emscripten C++:

R val::**call**<R>(const char *method, val param0, …, val paramN)
Invokes instance **method**, using parameters, returning type R: *instance.method(param0, …);*

void val::**set**(const char *attribute, val value)
Sets instance **attribute** to **value**: *instance.attribute = value;*

typed_memory_view **typed_memory_view**(size_t size, unsigned char *pointer)†
Constructor; creates a new **typed_memory_view** of designated **size** using the **pointer**.
†*There are several variants for different pointer types.*