Emscripten Tutorial

Using Emscripten is, at a base level, fairly simple. This tutorial takes you through the steps needed to compile your first Emscripten examples from the command line. It also shows how to work with files and set the main compiler optimization flags.

First things first

Make sure you have downloaded and installed Emscripten (the exact approach for doing this will depend your operating system: Linux, Windows, or Mac).

Emscripten is accessed using the Emscripten Compiler Frontend (emcc). This script invokes all the other tools needed to build your code, and can act as a drop-in replacement for a standard compiler like gcc or clang. It is called on the command line using `./emcc` or `./em++`.

> **Note**
>
> On Windows the tool is called using the slightly different syntax: `emcc` or `em++` . The remainder of this tutorial uses the Linux approach (`./emcc`).

For the next section you will need to open a command prompt:

On Linux or macOS, open a *Terminal*.

On Windows open the Emscripten Command Prompt, a command prompt that has been pre-configured with the correct system paths and settings to point to the active Emscripten tools. To access this prompt, type **Emscripten** in the Windows 8 start screen, and then select the **Emscripten Command Prompt** option.

Navigate with the command prompt to the emscripten directory under the SDK. This is a folder below the emsdk root directory, typically **<emsdk root directory>/upstream/emscripten/**. The examples below will depend on finding files relative to that location.

> **Note**
>
> In older emscripten versions the directory structure was different: the version number appeared, and the backend (fastcomp/upstream) did not, so you would use something like **<emsdk root directory>/emscripten/1.20.0/**.

Verifying Emscripten

If you haven't run Emscripten before, run it now with:

```
./emcc -v
```

If the output contains warnings about missing tools, see Verifying the Emscripten Development Environment for debugging help. Otherwise continue to the next sections where we'll build some code.

Running Emscripten

You can now compile your first C/C++ file to JavaScript.

First, lets have a look at the file to be compiled: **hello_world.c**. This is the simplest test code in the SDK, and as you can see, all it does is print "hello, world!" to the console and then exit.

```
/*
 * Copyright 2011 The Emscripten Authors.  All rights reserved.
 * Emscripten is available under two separate licenses, the MIT license and the
 * University of Illinois/NCSA Open Source License.  Both these licenses can be
 * found in the LICENSE file.
 */

#include <stdio.h>

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

To build the JavaScript version of this code, simply specify the C/C++ file after emcc (use em++ to force compilation as C++):

```
./emcc tests/hello_world.c
```

You should see two files generated by that command: **a.out.js** and **a.out.wasm**. The second is a WebAssembly file containing the compiled code, and the first is a JavaScript file containing the runtime support to load and execute it. You can run them using node.js:

```
node a.out.js
```

This prints "hello, world!" to the console, as expected.

> **Note**
>
> Older node.js versions do not have WebAssembly support yet. In that case you will see an error message suggesting that you build with `-s WASM=0` to disable WebAssembly, and then emscripten will emit the compiled code as JavaScript. In general, WebAssembly is recommended as it has widespread browser support and is more efficient both to execute and to download (and therefore emscripten emits it by default), but sometimes you may need your code to run in an environment where it is not yet present and so should disable it.

> **Tip**
>
> If an error occurs when calling emcc, run it with the `-v` option to print out a lot of useful debug information.

> **Note**
>
> In this section, and later on, we run some files from the `tests/` folder. That folder contains files for the Emscripten test suite. Some can be run standalone, but others must be run through the test harness itself, see Emscripten Test Suite for more information.

Generating HTML

Emscripten can also generate HTML for testing embedded JavaScript. To generate HTML, use the `-o <output>` command and specify an html file as the target file:

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

You can now open `hello.html` in a web browser.

> **Note**
>
> Unfortunately several browsers (including *Chrome*, *Safari*, and *Internet Explorer*) do not support `file://` XHR requests, and can't load extra files needed by the HTML (like a `.data` file, or packaged file data as mentioned lower down). For these browsers you'll need to serve the files using a local webserver and then open `http://localhost:8080/hello.html`).

Once you have the HTML loaded in your browser, you'll see a text area for displaying the output of the `printf()` calls in the native code.

The HTML output isn't limited just to just displaying text. You can also use the SDL API to show a colored cube in a `<canvas>` element (on browsers that support it). For an example, build the hello_world_sdl.c test code and then refresh the browser:

```
./emcc tests/hello_world_sdl.c -o hello.html
```

The source code for the second example is given below:

```
// Copyright 2011 The Emscripten Authors.  All rights reserved.
// Emscripten is available under two separate licenses, the MIT license and the
// University of Illinois/NCSA Open Source License.  Both these licenses can be
// found in the LICENSE file.

#include <stdio.h>
#include <SDL/SDL.h>

#ifdef __EMSCRIPTEN__
#include <emscripten.h>
#endif

int main(int argc, char* argv[]) {
  printf("hello, world!\n");

  SDL_Init(SDL_INIT_VIDEO);
  SDL_Surface *screen = SDL_SetVideoMode(256, 256, 32, SDL_SWSURFACE);

#ifdef TEST_SDL_LOCK_OPTS
  EM_ASM("SDL.defaults.copyOnLock = false; SDL.defaults.discardOnLock = true; SDL.defaults.opaqueFrontBuffer = fa
#endif

  if (SDL_MUSTLOCK(screen)) SDL_LockSurface(screen);
  for (int i = 0; i < 256; i++) {
    for (int j = 0; j < 256; j++) {
#ifdef TEST_SDL_LOCK_OPTS
      // Alpha behaves like in the browser, so write proper opaque pixels.
      int alpha = 255;
#else
      // To emulate native behavior with blitting to screen, alpha component is ignored. Test that it is so by out
      // data (and testing that it does get discarded)
      int alpha = (i+j) % 255;
#endif
      ((Uint32*)screen->pixels)[i * 256 + j] = SDL_MapRGBA(screen->format, i, j, 255-i, alpha);
    }
  }
  if (SDL_MUSTLOCK(screen)) SDL_UnlockSurface(screen);
  SDL_Flip(screen);

  printf("You should see a smoothly-colored square - no sharp lines but the square borders!\n");
  printf("And here is some text that should be HTML-friendly: amp: |&| double-quote: |"| quote: |'| less-than, g

  SDL_Quit();

  return 0;
}
```

Using files

> **Note**
>
> Your C/C++ code can access files using the normal libc stdio API (`fopen` , `fclose` , etc.)

JavaScript is usually run in the sandboxed environment of a web browser, without direct access to the local file system. Emscripten emulates a file system that you can access from your compiled C/C++ code using the normal libc stdio API.

Files that you want to access should be preloaded or embedded into the virtual file system. Preloading (or embedding) generates a virtual file system that corresponds to the file system structure at compile time, relative to *the current directory*.

The hello_world_file.cpp example shows how to load a file (both the test code and the file to be loaded shown below):

```
// Copyright 2013 The Emscripten Authors.  All rights reserved.
// Emscripten is available under two separate licenses, the MIT license and the
// University of Illinois/NCSA Open Source License.  Both these licenses can be
// found in the LICENSE file.

#include <stdio.h>

int main() {
  FILE *file = fopen("test/hello_world_file.txt", "rb");
  if (!file) {
    printf("cannot open file\n");
    return 1;
  }
  while (!feof(file)) {
    char c = fgetc(file);
    if (c != EOF) {
      putchar(c);
    }
  }
  fclose(file);
  return 0;
}
```

```
==
This data has been read from a file.
The file is readable as if it were at the same location in the filesystem, including directories, as in the local
==
```

> **Note**
>
> The example expects to be able to load a file located at **test/hello_world_file.txt**:
>
> ```
> FILE *file = fopen("test/hello_world_file.txt", "rb");
> ```
>
> We compile the example from the directory "above" **test** to ensure the virtual filesystem is created with the correct structure relative to the compile-time directory.

The following command is used to specify a data file to preload into Emscripten's virtual file system — before running any compiled code. This approach is useful because Browsers can only load data from the network asynchronously (except in Web Workers) while a lot of native code uses synchronous file system access. Preloading ensures that the asynchronous download of data files is complete (and the file is available) before compiled code has the opportunity to access the Emscripten file system.

```
./emcc tests/hello_world_file.cpp -o hello.html --preload-file test/hello_world_file.txt
```

Run the above command, then open **hello.html** in a web browser to see the data from **hello_world_file.txt** being displayed.

For more information about working with the file system see the File System Overview, File System API and Synchronous Virtual XHR Backed File System Usage.

Optimizing code

Emscripten, like *gcc* and *clang*, generates unoptimized code by default. You can generate slightly-optimized code with the `-O1` command line argument:

```
./emcc -O1 tests/hello_world.cpp
```

The "hello world" code created in **a.out.js** doesn't really need to be optimized, so you won't see a difference in speed when compared to the unoptimized version.

However, you can compare the generated code to see the differences. `-O1` applies several minor optimizations and removes some runtime assertions. For example, `printf` will have been replaced by `puts` in the generated code.

The optimizations provided by `-O2` (see here) are much more aggressive. If you run the following command and inspect the generated code (**a.out.js**) you will see that it looks very different:

```
./emcc -O2 tests/hello_world.cpp
```

For more information about compiler optimization options see Optimizing Code and the emcc tool reference.

Emscripten Test Suite and Benchmarks

Emscripten has a comprehensive test suite, which covers virtually all Emscripten functionality. These tests are an excellent resource for developers as they provide practical examples of most features, and are known to build successfully on the `main` branch.

See Emscripten Test Suite for more information.

General tips and next steps

This tutorial walked you through your first steps in calling Emscripten from the command line. There is, of course, far more you can do with the tool. Below are other general tips for using Emscripten:

This site has lots more information about compiling and building projects, integrating your native code with the web environment, packaging your code and publishing.

The Emscripten test suite is a great place to look for examples of how to use Emscripten. For example, if you want to better understand how the emcc `--pre-js` option works, search for `--pre-js` in the test suite: the test suite is extensive and there are likely to be at least some examples.

To learn how to use Emscripten in advanced ways, read src/settings.js and emcc) which describe the compiler options, and emscripten.h for details on JavaScript-specific C APIs that your C/C++ programs can use when compiled with Emscripten.

Read the FAQ.

When in doubt, get in touch!