Tutorial : introduction to WebAssembly with C++

B. Cottenceau LARIS/University of Angers France

Why this tutorial?

Using Emscripten/Javascript tools, you can expose on the Web some code written in C++.

As part of our research, we have developed calculation libraries in C++ (using boost): MinMaxGD for formal series for (max,+) systems, and ETVO (an extension of MinMaxGD). Thanks to the Emscripten tool chain, we can provide operating programs that can be run in a browser, and give easy access to our tools for basic use.

But this presentation is not totally self-contained. For instance, the installation phases of the tools (Emscripten/Python) are not explained here. The main focus is on what enables you to make Javascript/WASM(C++) wrappers. We present a few ways of exposing C++ functionalities for use in a Web page (HTML/JS/WASM).

WebAssembly: what is it ?

WebAssembly (wasm) is a bytecode (intermediate code) that can be used by a virtual machine integrated into web browsers. Like Java or .NET, the wasm bytecode is compiled in just-in-time by the browser. Wasm code can be exploited within web pages by interacting with Javascript (JS) code.

The wasm bytecode can be produced from advanced language sources, including C, C++ and RUST, but other languages will certainly be added to the list in the future. In the case of C/C++, the **Emscripten** tool chain handles C/C++ compilation to wasm. Emscripten also generates JS "glue-code" to simplify wasm loading and operation within an html page.

Other helpful links to discover WebAssembly:

- https://marcoselvatici.github.io/WASM_tutorial/ (a really good tutorial)
- https://www.tutorialspoint.com/webassembly/index.htm
- https://www.webassembly.fr/

A simple example to check the tool chain

Create and edit a text file (hello.cpp) for a C++ console program.

Then, use the Emscripten compiler to transform the C++ source into 2 output files

```
em++ hello.cpp -o hello.js          [command for the compilation with emscripten]
```

This command generates **`hello.js`** (javascript) and **`hello.wasm`** (WebAssembly)

```
// hello.cpp
#include <iostream>
using namespace std;

// em++ hello.cpp -o hello.js       [cmd to generate wasm+js]
// node hello.js                    [to execute the wasm version]

// em++ hello.cpp -o hello.html     [cmd to generate html+js+wasm]
// python -m http.server 8080       [to start a local web server ]

int main() {
  cout << "hello!\n";
  return 0;
}
```

Emscripten can also generate an html version (more exactly html/javascript/wasm) with the next command :

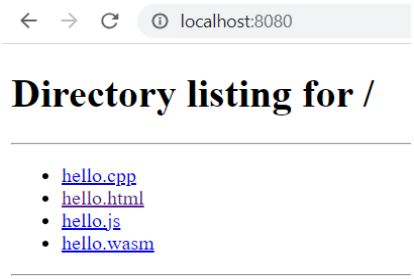
```
em++ hello.cpp -o hello.html
```

This command generates 3 files, **`hello.html, hello.js`** (javascript) and **`hello.wasm`** (WebAssembly)

To view the web page generated in this way, you can start a Web server on the machine (to supply the web page to a client). Since a python installation is required, you can easily start a web server with the following command (the web server starts listening on port 8080)

```
python -m http.server 8080
```

All that's left is to use a web browser to see the result (http://localhost:8080/)



The purpose of this first example is to check how the compilation chain works. Next, we propose a few ways of exposing C++ functions for use in HTML/JS

A C++ function invoked from JS

The emscripten compiler has a large number of options that can be activated, which sometimes results in very long commands. Tip: keep emscripten compilation commands in the C+ source files.

`-lembind`: to use binding options (`std::string` (C++) bound to JS string)

```
// source exwasm1.cpp
// em++ -lembind exwasm1.cpp -o exwasm1.js -s WASM=1    [compilation]
// output: exwasm1.wasm (bytecode) and exwasm1.js (JS glue-code)
#include<string>
#include<emscripten.h>
#include<emscripten/bind.h>
using namespace std;
using namespace emscripten;

// C++ function exposed to JS
string modifyStr(const string & s){ // returns a copy of s whose the first/last characters are
  string str(s);                    // replaced by 'X'
  if(str.size()>=3){
         str[0]='X';
         str[str.size()-1]='X';
  }
  return str;
}

EMSCRIPTEN_BINDINGS(module)
{
  emscripten::function("modify", &modifyStr);    //called from JS as Module.modify(...)
}
```

Compiling the C++ source produces a wasm file containing bytecode, as well as a JS "glue-code" file to simplify loading and calling wasm code from JS.

With this command, Emscripten generates a wasm file (`exwasm1.wasm`) and a glue-code JS file (`exwasm1.js`)

```
em++ -lembind exwasm1.cpp -o exwasm1.js -s WASM=1
```

This pair of files (JS glue-code/WASM) can be used within an HTML/JS page. The final web page is therefore in three parts

```
ex1.html / exwasm1.js / exwasm1.wasm
```

Below is an example (`ex1.html`) of how to exploit the WASM file via the glue-code file.

```
<!-- file ex1.html  -->
<style type="text/css">
 textarea {
  width: 100%;
  resize: vertical;
}
</style>

<div >
 <button onclick="bclick()">Click</button>
 <textarea id="textin" rows="5"></textarea>
 <textarea id="textout" rows="5"></textarea>
</div>

<!-- glue-code generated by emscripten  -->
<script src="exwasm1.js"></script>

<script>
<!-- called just after the WASM file is loaded -->
 Module['onRuntimeInitialized'] = function(){ textin.value = "input string"; }

<!-- click handler  -->
 function bclick(){
   textout.value= Module.modify(textin.value);  //call the exposed function
 }
</script>
```

How to bind C++ objects to JS variables

There are many ways in which the wasm bytecode (generated from C++) can interact with JS(javascript). Since this is a very short introduction, we'll concentrate here on C++ functions that exchange simple arguments (int, double, string), or arrays of numbers.

Numbers: in C++, we can distinguish between different types of integers and decimal points, but not in JS.
Strings: there's a binding between C++ `std::string` objects and JS strings.
Arrays: a specific emscripten C++ class named `emscripten::val` is used.

Note: `emscripten::val` does binding with all kinds of JS objects!

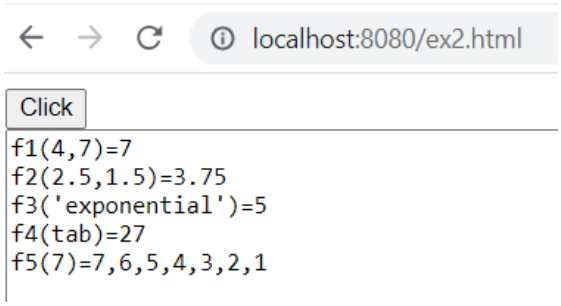
The following example illustrates different ways of exploiting arguments within C++ functions.

In C++ code, the easiest way to exploit JS arrays is to use the `emscripten::val` class (see func4 and func5)

```
// exwasm2.cpp

// em++ -lembind exwasm2.cpp -o exwasm2.js -s WASM=1  [emscripten command to compile]
// output: exwasm2.wasm (bytecode) and exwasm2.js (JS glue-code)

#include<string>
#include<vector>
#include<emscripten.h>
#include <emscripten/bind.h>
using namespace std;
using namespace emscripten;

int func1(int a, int b){ return std::max(a,b);}  // parameters of type int

double func2(double a, double b){ return a*b; }  // parameters of type double

int func3(const string & s){ return s.size()/2;}  // parameter of type string (C++)

int func4(emscripten::val arg){ // arg is assumed to be an array in javascript
       vector<int> v=vecFromJSArray<int>(arg);    // arg=a JS array
                  // v is a C++ object (vector<int>) filled with the values of arg
       int mx=0;
       if(v.size()>0){
         mx=v[0];
         for(int x:v) mx=std::max(mx,x);
       }
       return mx;  // returns the max value of arg (an array in JS)
}

emscripten::val func5(int s){  // returns an array in JS
       int v=s;
       vector<int> vres(s);
       for(int & x:vres){ x=v; v--;}   //v=[s,s-1,...,1]
       val array=val::array(vres.begin(),vres.end()); // creates an object from a C++ vector
       return array; // returns an object that will be used as a JS array
}

EMSCRIPTEN_BINDINGS(module){        // all these functions can be called from JS
  emscripten::function("f1", &func1);   // Module.f1(...) (JS) will call func1(...) (C++)
  emscripten::function("f2", &func2);
  emscripten::function("f3", &func3);
  emscripten::function("f4", &func4);
  emscripten::function("f5", &func5);
}
```

```
<!-- file ex2.html  -->
<style type="text/css">
 textarea {
  width: 100%;
  resize: vertical;
}
</style>

<div >
<button onclick="bclick()">Click</button>
<textarea id="textout" rows="5"></textarea>
</div>

<!-- JS glue-code generated by emscripten  -->
<script src="exwasm2.js"></script>

<script>
<!-- called once the WASM file is loaded -->
Module['onRuntimeInitialized'] = function(){      textout.value = 'wasm OK'; }
```

```
<!-- click handler  -->
function bclick()
{
 textout.value="f1(4,7)=";
 textout.value+= Module.f1(4,7) + "\n";
 textout.value+="f2(2.5,1.5)=";
 textout.value+= Module.f2(2.5,1.5) + "\n";
 textout.value+="f3('exponential')=";
 textout.value+= Module.f3('exponential') + "\n";
 var tab=[12,5,27,4,2]; // an array in JS
 textout.value+="f4(tab)=";
 textout.value+= Module.f4(tab) + "\n";
 textout.value+="f5(7)=";
 var tab2=Module.f5(7);  // returns an array
 textout.value+= tab2 + "\n";
}
```

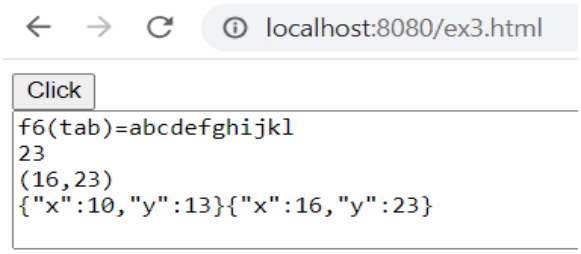


How to bind JS objects (with attributes) to `emscripten::val` objects in C++

Another example where objects are used in both parts; WASM (C++) and JS/HTML.

```
// exwasm3.cpp

//      em++ -lembind exwasm3.cpp -o exwasm3.js -s WASM=1
//      output: exwasm3.wasm (bytecode) and exwasm3.js (JS glue-code)

#include<string>
#include<vector>
#include<emscripten.h>
#include <emscripten/bind.h>
using namespace std;
using namespace emscripten;

string func6(emscripten::val arg){  // arg is assumed to be a JS array with strings
       vector<string> v=vecFromJSArray<string>(arg); // a vector<string> (C++) is filled
       string sres="";
       for(string s:v) sres+=s;
       return sres;  // all strings are concatenated and returned in one string
}

// arg = JS object (attributes x/y)  [see JS/HTML]
int func7(emscripten::val arg){
       int x=arg["x"].as<int>(); //reads the attribute x of the object arg (JS object)
       int y=arg["y"].as<int>(); //reads the attribute y of the object
       return x+y;
}

emscripten::val func8(int x,int y){
       // creates a JS object of type point (see JS/HTML)
       val point=val::global("point").new_(x,y);
       return point;
}

EMSCRIPTEN_BINDINGS(module){
  emscripten::function("f6", &func6);
  emscripten::function("f7", &func7);
   emscripten::function("f8", &func8);
}
```

```
<!-- file ex3.html  -->
<style type="text/css">
 textarea {
  width: 100%;
  resize: vertical;
}
</style>

<div >
 <button onclick="bclick()">Click</button>
 <textarea id="textout" rows="5"></textarea>
</div>

<!-- glue-code from emscripten  -->
<script src="exwasm3.js"></script>

<script>
<!-- called once WASM is loaded -->
Module['onRuntimeInitialized'] = function(){      textout.value = 'wasm OK'; }

function point(x,y)  // type point with 2 attributes x/y
{
       this.x=x;
       this.y=y;
       this.toString=function(){    //used for string outputs
              return '('+this.x+','+this.y+')';
       }
}

<!-- click handler  -->
function bclick()
{
 var tab=['ab','cde','fgh','ijkl'];
 textout.value="f6(tab)=";
 textout.value+= Module.f6(tab) + "\n";
 var p=new point(10,13); // JS object
 textout.value+= Module.f7(p) + "\n";

 var otherP=Module.f8(16,23);  // creates an object point from func8 C++/WASM
 textout.value+= otherP + "\n";
 textout.value+= JSON.stringify(p);        // JSON string description for p
 textout.value+= JSON.stringify(otherP);   // p and otherP are 2 JS objects point
}
</script>
```

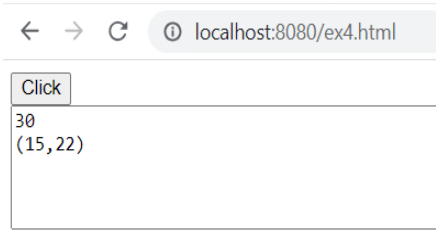


For JS classes created with the class keyword, it is slightly different

Since JavaScript allows objects to be created and completed, nothing is really set in stone. Oddly enough, the JS Object/Wasm link doesn't really work the same way if the objects are derived from JS classes created with the `class` keyword. In this case, however, you can do the following.

```
// exwasm4.cpp
// em++ -lembind exwasm4.cpp -o exwasm4.js -s WASM=1
// output: exwasm4.wasm (bytecode) and exwasm4.js (JS glue-code)
#include<string>
#include<vector>
#include<emscripten.h>
#include <emscripten/bind.h>
using namespace std;
using namespace emscripten;

// arg = is assumed to be a JS object of class point (attributes x/y)
int func9(emscripten::val arg){
       int x=arg["x"].as<int>();
       int y=arg["y"].as<int>();
       return x+y;
}

emscripten::val func10(int x,int y){
       // creates a JS object
       val point=val::object();
       point.set("x",x);
       point.set("y",y);      // creates onthefly two attributes
       return point;
}

EMSCRIPTEN_BINDINGS(module){
  emscripten::function("f9", &func9);
   emscripten::function("f10", &func10);
}
```

```
<!-- file ex4.html  -->
<style type="text/css">
 textarea {
  width: 100%;
  resize: vertical;
}
</style>
<div >
<button onclick="bclick()">Click</button>
<textarea id="textout" rows="5"></textarea>
</div>

<script src="exwasm4.js"></script>
<script>

Module['onRuntimeInitialized'] = function(){      textout.value = 'wasm OK'; }

class point{   // class JS
       constructor(x,y){ this.x=x;  this.y=y; }
       toString()
       { return '('+this.x+','+this.y+')'; }
}

<!-- click handler  -->
function bclick(){
 var p=new point(13,17);
 textout.value=Module.f9(p)+'\n';
 var pE=Module.f10(15,22);  // 'general' JS object without specific type
 var pN = Object.assign(new point,pE); // creates a point object with a copy of the values of pE
 textout.value+=pN;  // calls pN.toString()
}
</script>
```



An example where JS objects are linked to C++ variables

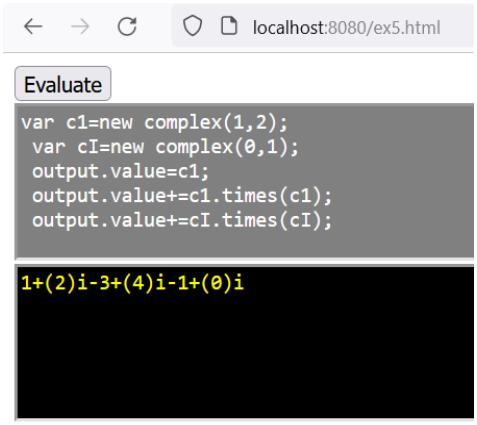
One way of exploiting C++ code is to create a javascript "mirror" type with attributes representing those of the C++ objects. In the example below, we want to exploit the `std::complex` C++ class. We'll create a JS `complex` type (mirror of `std::complex`). Two functions need to be added to create a C++ object from a JS object, and vice versa.

```
// exwasm5.cpp  (the C++ part to wrap JS to C++/WASM)

// em++ -lembind exwasm5.cpp -o exwasm5.js -s WASM=1 -s EXIT_RUNTIME=0 [compilation command]
// output: exwasm5.wasm (bytecode) and exwasm5.js (JS glue-code)

#include<string>
#include<sstream>
#include<complex>
#include<emscripten.h>
#include<emscripten/bind.h>

using namespace std;
using namespace emscripten;

/* In the JS part of the Web page, a complex type has to be defined
  with two attributes. For example, this one: (see the html file)

       function complex(r,i)
       {
              this.r=r;
              this.i=i;
              this.plus=function(c){return Module.cplus(this,c);}
              this.times=function(c){ return Module.ctimes(this,c);}
              this.toString=function(){return Module.cstring(this);}
       }
*/

// **** Two functions to wrap the mirrored types : complex JS <-> complex C++  **** //

// JS to C++(WASM)
// JS object complex (emscripten::val) -> std::complex (C++ object)
std::complex<double> valToComplex(emscripten::val vC) // vC represents a complex in JS
{
  double r=vC["r"].as<double>();     // attribute r from vC -> double ;
  double i=vC["i"].as<double>();     // attribute i from vC -> double;
  std::complex<double> objectCPP(r,i); // initialize a C++ object
  return objectCPP;
}

// C++(WASM) to JS
// std::complex -> emscripten::val (JS object complex)
emscripten::val complexToVal(const std::complex<double> & c)
{
  val C=val::global("complex").new_(c.real(),c.imag()); // instantiates a JS variable
  return C;
}


// ****  EXPOSED functions to  compute plus and times operations **** //

// vC1,vC2 assumed to be complex JS objects

 emscripten::val cplus(emscripten::val vC1,emscripten::val vC2) {
  std::complex<double> c1=valToComplex(vC1);   // c1 is a std::complex C++ object
  std::complex<double>c2=valToComplex(vC2);    // c2 is a std::complex C++ object
  std::complex<double> cRes=c1+c2;    // c1+c2 is a std::complex -> cRes is std::complex
  return complexToVal(cRes); // from std::complex to a variable in JS
  }
```

```
 // vC1,vC2 assumed to be complex JS objects

 emscripten::val ctimes(emscripten::val vC1,emscripten::val vC2) {
  std::complex<double> c1=valToComplex(vC1);   // c1 is a std::complex C++ object
  std::complex<double> c2=valToComplex(vC2);   // c2 is a std::complex C++ object
  std::complex<double> cRes=c1*c2;    // c1*c2 is a std::complex -> cRes is std::complex
  return complexToVal(cRes);
 }

 // vC1 assumed to be complex JS objects returns a string

 std::string cplxToString(emscripten::val vC1)  // to give a string output of a complex (JS)
 {
  std::complex<double> c1=valToComplex(vC1);  // JS to C++
  stringstream ss;
  ss << c1.real() << "+(" << c1.imag() << ")i";
  return ss.str();  // returns a string
 }


EMSCRIPTEN_BINDINGS(module){
   emscripten::function("cplus", &cplus);     // can be called from JS Module.cplus(__,__)
   emscripten::function("ctimes", &ctimes);
   emscripten::function("cstring", &cplxToString);
}
```

The C++ code compiled with emscripten produces bytecode (wasm) and a glue-code file. The following html/js page can exploit this bytecode. Here in particular, javascript language can be entered in the text area and evaluated as javascript code. The user can therefore test the use of the complex type (JS) mirroring the `std::complex` type (C++).

```
<!-- file ex5.html  -->
<style type="text/css">
 textarea {
  width: 100%;
  resize: vertical;
}
</style>
<div >
<button onclick="bevaluate()">Evaluate</button>
</div>

<div >
<textarea id="texta" name="texta" rows="5" columns="100"
style="color:white;background-color:grey;"
spellcheck="false"
></textarea>
</div>
<div>
<textarea id="output" name="output" rows="5" columns="100"
style="color:yellow;background-color:black;" readonly></textarea>
</div>

<!-- glue-code -->
<script src="exwasm5.js"></script>

<script>
Module['onRuntimeInitialized'] = function(){
texta.value = 'var c1=new complex(1,2);';
texta.value+='\n var cI=new complex(0,1);';
texta.value+= '\n output.value=c1;';
texta.value+= '\n output.value+=c1.times(c1);';
texta.value+= '\n output.value+=cI.times(cI);';
}
```



```
// kind of JS class to be wrapped to std::complex
function complex(r,i)
{
       this.r=r;
       this.i=i;
       this.plus=function(c){       return Module.cplus(this,c);  }
       this.times=function(c){      return Module.ctimes(this,c); }
       this.toString=function(){    return Module.cstring(this);  }
}

<!-- click handler  -->
function bevaluate(){
 // evaluate the first textarea as javascript code
 try{ eval(texta.value); }
 catch(error){ output.value=error;}
 }
</script>
```