A practical guide to WebAssembly memory

Radu Matei

January 10, 2021

Memory in WebAssembly is one of the topics that creates confusion for newcomers, particularly for those with experience in languages with memory management features like garbage collection, such as JavaScript, Go, or Java. In this article we explore using memory in WebAssembly in various scenarios - passing JavaScript arrays to Rust and AssemblyScript modules, checking for some basic memory leaks using Valgrind, or exchanging strings between runtimes and modules using Wasmtime.

When instantiating a WebAssembly module, by default it does not have access to anything outside its sandbox runtime. It can declare and use its own memory buffers, but otherwise, it cannot access anything outside its environment, unless explicitly allowed by the host through module imports. This includes browser APIs, system files, libraries, or devices, which leaves two main ways of communicating between an instantiated module and the underlying host runtime:

- arguments and return values from invoking imported and exported functions - while this is the simplest way of passing information between a module instance and a host runtime, it is limited to exchanging fundamental WebAssembly data types (`i32 | i64 | f32 | f64`; as the interface types and multi-value Wasm proposals get implemented, runtimes would also be able to exchange additional types, for example strings, and return more than one such value).
- using WebAssembly memory - until the interface types proposal gets implemented, raw WebAssembly memory can be used to pass non-fundamental data types between runtimes and instances - for example arrays, strings, network socket streams, deep learning models, or serialized data (such as JSON or protocol buffers).

In practice, a combination of the two is mostly used: passing pointers as arguments and return values, and using memory to copy the data to and from those pointers, which is what this article explores in the next sections.

WebAssembly memory

Memory in WebAssembly is represented as a contiguous, mutable array of uninterpreted bytes. It can dynamically grow (in units of the *page size*, which is equal to 64Ki), and the bytes can be mutated by the module, through memory instructions, or by the host runtime, if the particular memory instance has been exported by the module. The indexes of a linear memory array can be considered memory addresses, and instructions that need to access a memory location are given an offset relative to the start of the memory. This ensures a few important things related to memory safety in WebAssembly (assuming the runtime implementation is bug-free):

- a potentially malicious module cannot use an arbitrary memory address and access data outside of its linear memories.
- because the size of a memory is always known, the runtime can check whether a memory offset that a module is trying to access is still within the boundaries of its allocated memory.
- as a result of the previous points, a module cannot access the memory of another module, the memory of the runtime, or the memory of the underlying operating system of the runtime, unless explicitly given access to.

> While WebAssembly runtimes do an excellent job of isolating the memory instances of different instances, within their own linear memories, WebAssembly modules are not safe from memory vulnerabilities, such as buffer overflow, or use-after-free. The *Progressive Memory Safety for WebAssembly* paper proposes an extension to WebAssembly that would add a new safe memory segment, which could be accessed exclusively through *handles* (strongly-typed objects that *encapsulate bounds-checked, memory-safe pointers to the segment memory*). The security-performance trade-offs proposed by the paper could prevent entire classes of memory safety issues within WebAssembly programs, together with proposed hardware security implementations.

WebAssembly modules can be written in various programming languages, each with its own memory management model, and when attempting to write or read a module's memory from the host runtime, one should be aware of how the module handles allocations and deallocations. Conceptually, there is a choice between copying the data from the host to the module, or the module owning the data and being responsible for managing its lifetime, and depending on the use case, both approaches are valid. In either case, the module must export functionality to allocate memory, and the host must understand how to read and write to and from it. In most cases, when using code generation libraries such as `wasm-bindgen`, memory allocation and deallocation are handled by the library - however, it is worth writing a simple version of an allocator for arrays to better understand the mechanics.

The complete code from this article can be found on GitHub.

Passing arrays to Rust WebAssembly modules

Let's assume we are attempting to write a WebAssembly module with a function that takes an array as input, and returns the sum of all the elements of the array, then invoke this function from a host runtime (such as the Node.js WASI runtime, a browser runtime, or Wasmtime). Because arrays are not fundamental data types in WebAssembly, we have to allocate a number of bytes (depending on the data type of the elements) in the module's linear memory, pass the data from the runtime into the memory, then invoke the entrypoint with a pointer to the start of the array and its length. First, we need to write a very simple allocator. A Rust implementation for this allocates a new `Vec<u8>` given a capacity `len`, and returns a mutable pointer to the start of the allocated memory block. Before returning, it is extremely important to make sure the memory that was just allocated does not go out of scope at the end of the function - ths is ensured using `std::mem::forget(buf)`:

```rust
/// Allocate memory into the module's linear memory
/// and return the offset to the start of the block.
#[no_mangle]
pub fn alloc(len: usize) -> *mut u8 {
    // create a new mutable buffer with capacity `len`
    let mut buf = Vec::with_capacity(len);
    // take a mutable pointer to the buffer
    let ptr = buf.as_mut_ptr();
    // take ownership of the memory block and
    // ensure that its destructor is not
    // called when the object goes out of scope
    // at the end of the function
    std::mem::forget(buf);
    // return the pointer so the runtime
    // can write data at this offset
    return ptr;
}
```

Assuming the host runtime called `alloc` and filled the linear memory with the desired data, it can now invoke a function that performs the actual computation, by passing the pointer returned by `alloc`, together with the length of the array. These arguments are used with Rust's `Vec::from_raw_parts` to create a `Vec<u8>` with the respective length and capacity, compute the element-wise sum, then return it. Notice the function *is* unsafe, because `Vec::from_raw_parts` assumes memory has been correctly allocated previously (which, if the runtime called `alloc` properly, should be satisfied):

```
/// Given a pointer to the start of a byte array and
/// its length, return the sum of its elements.
#[no_mangle]
pub unsafe fn array_sum(ptr: *mut u8, len: usize) -> u8 {
    // create a Vec<u8> from the pointer to the
    // linear memory and the length
    let data = Vec::from_raw_parts(ptr, len, len);
    // actually compute the sum and return it
    data.iter().sum()
}
```

Compiling this Rust program to a WebAssembly target (`cargo build --target wasm32-unknown-unknown`, or `--target wasm32-wasi`), the output is a `.wasm` module that can be instantiated in a compatible JavaScript runtime - for example, the WASI runtime which was recently added to Node.js, or from a modern browser.

The WebAssembly module exports two functions: one for allocating a byte array, the other for performing a computation using a byte array present at the allocated memory. In order to copy an array into the module's linear memory from the JavaScript runtime, the `alloc` exported function must be called using the array's length as parameter. This function returns an offset into the module's linear memory, which can be used to fill a new JavaScript `ArrayBuffer` (using the `TypedArray.prototype.set()` function):

```
// Copy `data` into the `instance` exported memory buffer.
function copyMemory(data, instance) {
  // the `alloc` function returns an offset in
  // the module's memory to the start of the block
  var ptr = instance.exports.alloc(data.length);
  // create a typed `ArrayBuffer` at `ptr` of proper size
  var mem = new Uint8Array(instance.exports.memory.buffer, ptr, data.length);
  // copy the content of `data` into the memory buffer
  mem.set(new Uint8Array(data));
  // return the pointer
  return ptr;
}
```

Instead of `set`, any other function available on `TypedArray` can be used to fill the `ArrayBuffer` that points to the module's memory, and if the module will write anything to its memory and return an offset, that information will be available to the JavaScript runtime in the `mem` variable - it is not the case in this example, as the function directly returns a value.

Finally, we use the `copyMemory` function above to copy the array and get the offset, then invoke the function that performs the actual computation, `array_sum`:

```
// Invoke the `array_sum` exported method and
// log the result to the console
function arraySum(array, instance) {
  // copy the contents of `array` into the
  // module's memory and get the offset
  var ptr = copyMemory(array, instance);
  // invoke the module's `array_sum` exported function
  // and log the result
  var res = instance.exports.array_sum(ptr, array.length);
  console.log("Result: " + res);
}
```

Fetching the module bytes (depending on whether this is running in a browser or Node.js, this can be done either through the `fetch` API, or through the Node.js `fs.readFile/Sync` API) and instantiating the module, we can now invoke the `arraySum` function and pass a plain JavaScript array as argument - of course, this will print `Result: 15` to the console:

```
(async () => {
  const mod = new WebAssembly.Module(module_bytes);
  // instantiate the module
  const instance = await WebAssembly.instantiate(mod, {});
  // execute the `arraySum` function and pass a
  // plain JavaScript array as data
  arraySum([1, 2, 3, 4, 5], instance);
})();
```

`dealloc` or `free`, and a basic way of checking for memory leaks

Earlier in the article, it was explicitly mentioned that, besides *allocating* bytes in its own linear memory, the module should also allow functionality for *deallocating*, or *freeing* memory. So why didn't the example above include any deallocations? This is where Rust ownership becomes useful.

Recall that in the `alloc` function we used `std::mem::forget(buf)` to ensure the allocated memory block does not go out of scope at the end of the function, since it is actually needed later (i.e. in a different function). Then, in the `array_sum` function, a Rust `Vec<u8>` is recreated from the pointer and length using `Vec::from_raw_parts`. Reading the documentation for `Vec::from_raw_parts`, the new `Vec<u8>` takes ownership of the memory block represented by `ptr`:

> The ownership of `ptr` is effectively transferred to the `Vec<T>` which may then deallocate, reallocate or change the contents of memory pointed to by the pointer at will. Ensure that nothing else uses the pointer after calling this function.

This means that when the new `Vec<u8>` goes out of scope, at the end of the `array_sum` function, the memory pointed by `ptr` should be deallocated.

Is there a way to convince ourselves of that? Let's explore how the Rust API we built for WebAssembly would be directly used from Rust, without using a WebAssembly runtime:

```
fn main() {
    // create a `Vec<u8>` as input
    let input = vec![1 as u8, 2, 3, 4, 5];
    // call the `alloc` function
    let ptr = alloc(input.len());
    let res: u8;
    unsafe {
        // copy the contents of `input`into the buffer
        // returned by `alloc`
        std::ptr::copy(input.as_ptr(), ptr, input.len());
        // call the `array_sum` function with the pointer
        // and the length of the array
        res = array_sum(ptr, input.len());
    }
    // print the result
    println!("Result: {:#?}", res);
}
```

This is is *not* using a WebAssembly runtime yet, but using plain Rust to compile a program that can be profiled for memory leaks. We can directly use the Rust compiler, `rustc`, to create an executable:

```
$ rustc src/lib.rs -o mem
$ ./mem
Result: 15
```

The part we *can* check for memory leaks is the Rust implementation - particularly, we need to make sure the `array_sum` function, which takes ownership of a pointer where *something* wrote data, is properly deallocating before returning:

```
$  valgrind --tool=memcheck ./mem
 Memcheck, a memory error detector
 Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
...
 LEAK SUMMARY:
    definitely lost: 0 bytes in 0 blocks
    indirectly lost: 0 bytes in 0 blocks
      possibly lost: 0 bytes in 0 blocks
...
 ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 4 from 4)
```

According to Valgrind, there are no memory leaks in this part of the implementation. This is obviously *not* a comprehensive way of testing for memory leaks in a WebAssembly environment, but it should be enough to convince us that there are no memory leaks in these two functions.

But what if we didn't read the `Vec::from_raw_parts` documentation and called a deallocation function anyway? (This definitely *didn't* happen to me!). We can write a simple deallocator function for a memory block by calling `std::mem::drop` using the desired value, which effectively takes ownership of `data` and goes out of scope:

```rust
#[no_mangle]
pub unsafe fn dealloc(ptr: *mut u8, size: usize) {
    let data = Vec::from_raw_parts(ptr, size, size);

    std::mem::drop(data);
}
```

We can update the Rust program above to call `dealloc` after getting the computation result:

```rust
fn main() {
    let input = vec![1 as u8, 2, 3, 4, 5];
    let ptr = alloc(input.len());
    let res: u8;
    unsafe {
        std::ptr::copy(input.as_ptr(), ptr, input.len());
        res = array_sum(ptr, input.len());
        // this is the only difference compared to
        // the previous example - `dealloc` is called
        // after calling `array_sum`
        dealloc(ptr, input.len());
    }
    println!("Result: {:#?}", res);
}
```

Trying to execute the program now, it fails because it is trying to deallocate an already freed memory block:

```
$ rustc src/lib.rs -o mem
$ ./mem
mem(5207,0x10b997dc0) malloc: *** error for object 0x7fb2a8c01c00:
        pointer being freed was not allocated
mem(5207,0x10b997dc0) malloc: *** set a breakpoint in
        malloc_error_break to debug
```

Running Valgrind again highlights the error as taking place in the `dealloc` function:

```
Invalid free() / delete / delete[] / realloc()
at 0x100167A0D: free (in /vgpreload_memcheck-amd64-darwin.so)
by 0x1000027C1: alloc::alloc::dealloc (in ./mem)
by 0x100002921: <alloc::alloc::Global as
                    core::alloc::AllocRef>::dealloc (in ./mem)
by 0x100001C6D: <alloc::raw_vec::RawVec<T,A>
                    as core::ops::drop::Drop>::drop (in ./mem)
by 0x100001E7C: core::ptr::drop_in_place (in ./mem)
by 0x100001ECD: core::ptr::drop_in_place (in ./mem)
by 0x100001D58: core::mem::drop (in ./mem)
ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 4 from 4)
```

Whenever passing ownership for some data from JavaScript (or another host runtime) to Rust, the Rust implementation is responsible for deallocating its own memory (and this comment does a great job of explaining how and when `free` or `dealloc` calls should be made depending on who owns the data). Similarly, whenever passing ownership from Rust to JavaScript (or another host runtime), the runtime is responsible for ensuring the deallocation of data it received.

A final note around allocating and deallocating - `wasm-bindgen`'s implementation for `malloc` and `free` is a more general-purpose solution compared to always using vectors - the following implementations are adapted from `wasm-bindgen`, with updated names to avoid clashing with the global `malloc` and `free` symbols. Essentially, they rely on a globally configured allocator to perform the actual operations - this can be the standard library allocator, or another one, such as `wee_alloc`:

```rust
use std::alloc::{alloc, dealloc, Layout};

#[no_mangle]
pub unsafe fn my_alloc(len: usize) -> *mut u8 {
    let align = std::mem::align_of::<usize>();
    let layout = Layout::from_size_align_unchecked(size, align);
    alloc(layout)
}

#[no_mangle]
pub unsafe fn my_dealloc(ptr: *mut u8, size: usize) {
    let align = std::mem::align_of::<usize>();
    let layout = Layout::from_size_align_unchecked(size, align);
    dealloc(ptr, layout);
}
```

Passing arrays to AssemblyScript modules

So far we have seen how to write a Rust module that doesn't use a code generator or special loader in order to pass arrays to allocate memory for an array. AssemblyScript is a relatively new programming language, with a syntax similar to TypeScript, which compiles natively to WebAssembly. For people new to the language, the arrays example is definitely the best way to understand how to write arrays into memory from JavaScript - however, it is worth exploring how to achieve the same functionality as the Rust module from above, without using any specialized loaders: write a function that allocates memory for a byte array, then another one that performs some computation on arrays passed from a JavaScript runtime.

Allocating is done similarly to Rust:

```
// Allocate memory for a new byte array of
// size `len` and return the offset into
// the module's linear memory to the start
// of the block.
export function alloc(len: i32): usize {
  // create a new AssemblyScript byte array
  let buf = new Array<u8>(len);
  let buf_ptr = memory.data(8);
  // create a pointer to the byte array and
  // return it
  store<Array<u8>>(buf_ptr, buf);
  return buf_ptr;
}
```

Loading the array from memory, however, is done slightly differently here (`changetype` didn't seem to work here, for some reason) - read the values found at `buf_ptr`, until reaching `buf_ptr + len` (the size of a `u8` is 1 byte), and compute the sum of all elements:

```
export function array_sum(buf_ptr: usize, len: i32): u8 {
    let result: u8 = 0;
    for(let i = 0; i < len; i++) {
      // for each new item in the array,
      // load a byte from the memory and add
      // it to the result
      result += load<u8>(buf_ptr + i) as u8;
    }
    return result as u8;
}
```

At this point, declare an `abort` function so we don't have to define any module imports, build the module (`asc index.ts --use abort=index/abort`), and we can use this module as a drop-in replacement for the Rust module we built above:

```
export function abort(
  message: string | null,
  fileName: string | null,
  lineNumber: u32,
  columnNumber: u32
): void {}
```

A call from JavaScript to the module's `__release` function should be included once a memory block is no longer needed, particularly for long-running modules - this instructs AssemblyScript's reference counter to collect objects that are no longer referenced: `instance.exports.__release(ptr);`

It should be pointed out again that AssemblyScript's own loader is better suited for allocating memory for arrays or strings, as it actually contains checks related to alignment and carry.

Passing arrays to modules using Wasmtime

While Wasmtime is implemented in Rust, which is a fundamentally different language compared to JavaScript, we have to implement surprisingly similar steps in order to execute the WebAssembly modules we built: copy a byte array into the module's linear memory,then invoke the `array_sum` function and get the result.

Let's start with the function that copies the memory. Using Wasmtime's API, get the module's exported memory, call the module's `alloc` function with the array size, then copy the contents of the input at the offset returned by `alloc`, relative to the start of the memory. Conceptually, it is the same thing that the `copyMemory` JavaScript function implements:

```rust
/// Copy a byte array into an instance's linear memory
/// and return the offset relative to the module's memory.
fn copy_memory(
    bytes: &Vec<u8>,
    instance: &Instance
) -> Result<isize, anyhow::Error> {
    // Get the "memory" export of the module.
    // If the module does not export it, just panic,
    // since we are not going to be able to copy the data.
    let memory = instance
        .get_memory(MEMORY)
        .expect("expected memory not found");

    // The module is not using any bindgen libraries,
    // so it should export its own alloc function.
    //
    // Get the guest's exported alloc function, and call it with the
    // length of the byte array we are trying to copy.
    // The result is an offset relative to the module's linear memory,
    // which is used to copy the bytes into the module's memory.
    // Then, return the offset.
    let alloc = instance
        .get_func(ALLOC_FN)
        .expect("expected alloc function not found");
    let alloc_result = alloc.call(&vec![Val::from(bytes.len() as i32)])?;

    let guest_ptr_offset = match alloc_result
        .get(0)
        .expect("expected the result of the allocation to have one value")
    {
        Val::I32(val) => *val as isize,
        _ => return Err(Error::msg("guest pointer must be Val::I32")),
    };
    unsafe {
        let raw = memory.data_ptr().offset(guest_ptr_offset);
        raw.copy_from(bytes.as_ptr(), bytes.len());
    }
    return Ok(guest_ptr_offset);
}
```

Next, we implement a function that creates a new WebAssembly instance (this is not explicitly shown, as Wasmtime's API makes this extremely easy), copies the input into its memory, then invokes the module's exported `array_sum` function and prints the result on the console:

```
/// Invoke the module's `array_sum` exported method
/// and print the result to the console.
fn array_sum(input: Vec<u8>) -> Result<i32, anyhow::Error> {
    // create a new Wasmtime instance
    let instance = create_instance(WASM.to_string())?;
    // write the input array to the module's linear memory
    let ptr = copy_memory(&input, &instance)?;
    // get the module's exported `array_sum` function
    let array_sum = instance
        .get_func(ARRAY_SUM_FN)
        .expect("expected array_sum function not found");

    // call the `array_sum` function with the pointer to the
    // array and length
    let results = array_sum.call(&vec![
            Val::from(ptr as i32),
            Val::from(input.len() as i32)])?;
    // return the result
    match results
        .get(0)
        .expect("expected the result of array_sum to have one value")
    {
        Val::I32(val) => Ok(*val),

        _ => return Err(Error::msg("cannot get result")),
    }
}
```

Finally, we can create an entrypoint to this program. Running it, we replicated the functionality in a completely different WebAssembly runtime:

```
pub fn main() {
    let input = vec![1 as u8, 2, 3, 4, 5];
    let res = array_sum(input).unwrap();
    println!("Result from running {}: {:#?}", WASM, res);
}
```

So far, we have exclusively passed byte arrays *from* the runtime *to* the module. While this is enough for passing any sort of binary data to modules, it is not the only scenario - modules can also return complex data back to the runtime by writing in their linear memory and returning the pointer to the start of the data, and we will explore this scenario in the next section by exchanging strings.

Exchanging strings between modules and runtimes

Theoretically, passing strings from a runtime to a module can be done using almost exactly the same implementation we saw so far - encoding strings as UTF byte arrays, we can just reuse the `alloc` function from the previous examples. However, unlike previously, where the computation function returned a value (the element-wise sum of the array), we now want the runtime to get back a string (or a byte array). Similarly, we can return a pointer into the module's linear memory where the string is located, and have the runtime get a *view* into the module's memory at that offset and read it.

The important thing to note here is the need to use `std::mem::forget` for the string we are returning to the runtime across the WebAssembly boundary. Because its content will have to be read externally to this function, we need to make sure it does not go out of scope when the function returns, so the WebAssembly runtime can read its content. This also means the runtime is now effectively owning this data, and should ensure it is freed when no longer needed.

```
/// Given a pointer to the start of a byte array and
/// its length, read a string, create its uppercase
/// representation, then return the pointer to it
#[no_mangle]
pub unsafe fn upper(ptr: *mut u8, len: usize) -> *mut u8 {
    // create a `Vec<u8>` from the pointer and length
    // here we could also use Rust's excellent FFI
    // libraries to read a string, but for simplicity,
    // we are using the same method as for plain byte arrays
    let data = Vec::from_raw_parts(ptr, len, len);
    // read a Rust `String` from the byte array,
    let input_str = String::from_utf8(data).unwrap();
    // transform the string to uppercase, then turn it into owned bytes
    let mut upper = input_str.to_ascii_uppercase().as_bytes().to_owned();
    let ptr = upper.as_mut_ptr();
    // take ownership of the memory block where the result string
    // is written and ensure its destructor is not
    // called whe the object goes out of scope
    // at the end of the function
    std::mem::forget(upper);
    // return the pointer to the uppercase string
    // so the runtime can read data from this offset
    ptr
}
```

From JavaScript, we can reuse the copyMemory function implemented earlier, provided we first transfer the JavaScript input string to UTF-8, then invoke the module's upper function, which returns the offset into the linear memory where the result string was written.

```
// Invoke the `upper` function from the module
// and log the result to the console.
function upper(input, instance) {
  // transform the input string into its UTF-8
  // representation
  var bytes = new TextEncoder("utf-8").encode(input);
  // copy the contents of the string into
  // the module's memory
  var ptr = copyMemory(bytes, instance);
  // call the module's `upper` function and
  // get the offset into the memory where the
  // module wrote the result string
  var res_ptr = instance.exports.upper(ptr, bytes.length);
  // read the string from the module's memory,
  // store it, and log it to the console
  var result = readString(res_ptr, bytes.length, instance);
  console.log(result);
  // the JavaScript runtime took ownership of the
  // data returned by the module, which did not
  // deallocate it - so we need to clean it up
  deallocGuestMemory(res_ptr, bytes.length, instance);
}
```

Reading from an instance's memory can be done using a slice of size len (which in this case is equal to the input's size) into the typed array used to represent the memory:

```
// Read a string from the instance's memory.
function readString(ptr, len, instance) {
  var m = new Uint8Array(instance.exports.memory.buffer, ptr, len);
  var decoder = new TextDecoder("utf-8");
  // return a slice of size `len` from the module's
  // memory, starting at offset `ptr`
  return decoder.decode(m.slice(0, len));
}
```

Finally, as hinted to earlier, because the WebAssembly module prevented the memory block that contains the string result to go out of scope, we need to manually deallocate that memory when it is no longer needed. This can be done by calling the module's exported `dealloc` function using the result string's pointer (calling `dealloc` using the input string would try to deallocate already freed memory, and would result in a `pointer being freed was not allocated` error):

```
function deallocGuestMemory(ptr, len, instance) {
  // call the module's `dealloc` function
  instance.exports.dealloc(ptr, len);
}
```

We can now use the `upper` function with a native JavaScript string, execute the WebAssembly function we implemented to get an uppercase string, and log it to the console:

```
(async () => {
  const mod = new WebAssembly.Module(module_bytes);
  const instance = await WebAssembly.instantiate(mod, {});

  upper("this should be uppercase", instance);
})();
```

Running the script with Node.js would print the uppercase representation of the input to the console.

For brevity, most of the Wasmtime implementation for passing strings will be omitted here, since it is similar to the `array_sum` function - however, of interest is reading a chunk of data from the instance's memory, using the result pointer from the module. We can use the `data_unchecked` function to get a Rust slice into the module's linear memory, read `len` bytes, then try to read a Rust `String` from those bytes and return it:

```
/// Read a Rust `String` from a module's memory
/// given an offset and length.
pub unsafe fn read_string(
    memory: &Memory,
    data_ptr: u32,
    len: u32,
) -> Result<String, anyhow::Error> {
    // get a raw byte array from the module's linear memory
    // at offset `data_ptr` and length `len`.
    let data = memory
```

```
            .data_unchecked()
            .get(data_ptr as u32 as usize..)
            .and_then(|arr| arr.get(..len as u32 as usize));
        // attempt to read a UTF-8 string from the memory
        let str = match data {
            Some(data) => match std::str::from_utf8(data) {
                Ok(s) => s,
                Err(_) => return Err(Error::msg("invalid utf-8")),
            },
            None => return Err(Error::msg("pointer/length out of bounds")),
        };

        Ok(String::from(str))
    }
```

It is worth mentioning that in most cases when a module is writing an array in memory and returning the pointer, the length of the array should also be returned somehow - usually done by passing a pointer as argument where the module can write the length of the array (this is a side effect of not having multiple return value yet). We didn't have to do this because the length of the result in our example was always going to be the same as the input.

We can apply the exact same principles and exchange any array between modules and runtimes. As long as both the module and runtime agree on the data they are exchanging, they can share number arrays, byte arrays, strings, JSON, protocol buffers, or any other serialization format.

Memory in WebAssembly can be a confusing topic, particularly for those getting started, and while using `wasm-bindgen` and the AssemblyScript loader can significantly simplify exchanging complex data types, it is worth understanding the internals of memory management in WebAssembly, and hopefully this article helps someone build their awesome WebAssembly application.