Tables of Contents

1. Basics

1.1 Sizes

Wasm, for now, uses 32-bit memory addressing, so a 32 bit integer (i32) is used for sizes of structs, strings, and arrays. However, this would need to change in the future since there's a proposal to add 64-bit memory indexes to WebAssembly: https://github.com/WebAssembly/memory64

1.2 Built-in functions

There are two types of built-in functions in MyWASM:

1.2.1 WASM Built-ins functions

Functions that are implemented in WASM itself:

1.2.1.1 itos

Signature: **string itos(int num)**

Converts an ***i32*** value to a string.

1.2.1.2 itod

Signature: **double itod(int num)**

Converts an ***i32*** value to ***f64***. This is implemented using ***f64.convert_i32_s***.

1.2.1.3 dtos

Signature: **string itos(double num)**

Converts an ***f64*** value to a string.

1.2.1.4 dtoi

Signature: **int dtoi(double num)**

Converts an ***f64*** value to ***i32.*** This is implemented using ***i32.trunc_f64_s***.

1.2.1.5 stod

Signature: **double stod(string str)**

Converts a string to an ***f64*** value.

1.2.1.6 stoi

Signature: **int stoi(string str)**

Converts a string to an ***i32*** value. First the string is converted to ***f64*** using ***stod,*** and then it is casted to ***i32***.

1.2.1.7 length

Signature:

1. **int length(string str)**
2. **int length(array <any_type> arr)**

The length of arrays are stored at ***arr[-1]***, where ***arr*** is the pointer to the array. So all ***length*** does is read ***arr[-1]*** and return the value. Since, as mentioned before, sizes are 32-bit integers, reading -1 index means 4 bits to the left, irrespective of the type of array. Strings are just arrays of ***i32*** values, so the same function can be used without having to implement anything special for it.

1.2.1.8 get

Signature: **string get(int index, string str)**

Returns the character at the i^{th} index of a string. Since strings are just arrays of integers, all we need to do is ***str[i]*** and return the value.

1.2.1.9 string_ini

Signature: **int string_ini(int size)**

Initializes a string and returns the pointer to it.

1.2.1.10 string_ini_assign

Signature: **int string_ini_assign(int str_ptr, int index, int value)**

Takes in the string pointer and the index, and assigns it the given value.

Note: there are other built-in functions that are used solely for internal usage, and hence have not been documented here.

1.2.2 Host Built-ins functions

Functions that need to be implemented by the host language due to technical limitations.

1.2.2.1 print

Signature: **void print(string str)**

WASM does not support basic I/O, so print needs to be implemented by the host language. When print is called, the pointer to the string is passed, which should then be used to print the string. When print is used with non-string values, then the value is casted to a string using the appropriate function mentioned in 1.2.1

1.2.2.2 input

Signature: **string input()**

The host language must take in the input from the user, then call ***string_ini*** to initiate the string, and then call ***string_ini_assign*** for each character of the string. Then it must return the pointer to the string that was returned from ***string_ini***.

1.2.2.3 allocate_memory

Signature: **int allocate_memory(int requestedSize)**

Refer to 2.2

1.2.2.4 deallocate_memory

Signature: **void deallocate_memory(int index, int size)**

Refer to 2.2

1.2.2.5 sleep

Signature: **void sleep(int ms)**

Sleeps for ***ms*** milliseconds

1.2.2.6 random

Signature: **void random()**

Returns a random ***f64*** between 0 (inclusive) and 1 (exclusive)

1.3 Base types

1.3.1 int

WASM has native support for ints: ***i32***.

1.3.2 double

WASM has native support for doubles: ***f64***.

1.3.3 bool

WASM does not support int8, and neither does it support int16 (https://github.com/WebAssembly/design/issues/85), so ***i32*** are used to store bool.

1.3.4 string

The way strings are implemented is rather complex to fit in this section, so refer to section 5.

Note: Since pointers are ints, ***i32*** is used for arrays, structs and strings.

1.4 Scope and local variables

WASM has no concept of scope other than local variables in functions. This is problematic since variables can be declared in other blocks other than

functions like ***if***, ***while***, and ***for***. On top of that, WASM's local variables must be declared right at the top of the functions. This means that we need to keep track of all local variables in all the local scopes. Another thing we need to consider is that local variables can have the same names, so we need to transform them in a way that they retain their original meaning. This is done in MyWASM in the following manner:

Number the current scope by its name. For example, if we have encountered 5 **if** statements in the current scope, then name the current scope as "if_6". Then all the variables encountered in that if's scope will have the prefix "if_6". Let's consider a concrete case:

Original code:

```
function void main() {
 int a = 10;
 if(true) {
   int a = 20;
   if(true) {
     int a = 30;
   }
 } else {
   int a = 40;
   if(true) {
     int a = 50;
   } else {
     int a = 60;
   }
 }
}
```

Transformed code:

```
function void main() {
 int a = 10;
 if (true) {
   int if1_a = 20;
   if (true) {
     int if1_if2_a = 30;
```

```
    }
  }
  else {
    int else1_a = 40;
    if (true) {
      int else1_if3_a = 50;
    }
    else {
      int else1_else2_a = 60;
    }
  }
}
```

2. Memory Management

2.1. WASM Garbage Collection Proposal

I could have used WASM's instructions for garbage collection, essentially giving me an option to not have to implement any kind of memory management. However, the GC proposal (https://github.com/WebAssembly/gc) is really new and it was hard to find a compiler that would allow compiling WebAssembly Text (WAT) to its binary format. And as far as I know, stable versions of most WASM runtimes don't even support the GC opcodes yet. So, I decided to implement my own memory management, which was way more fun than using native opcodes.

2.2 Built-in functions for Memory Management

Since WASM has access to linear memory, MyWASM manages memory in the following manner:

- **allocate_memory(int n)**: This function takes in a single ***i32*** value, and returns the offset of the memory in a way that ***n*** bits from that offset can be used.

- **deallocate_memory(int index, int size)**: This deallocates ***size*** bits starting from offset *index*.

- A variable called ***global_offset*** which keeps track of where memory allocation may begin; the key word being may because having the ability to deallocate memory means that an offset lesser than ***global_offset*** may be available for allocation.

  NOTE: These functions need to be implemented by the host language that's running the WASM runtime, rather than in WASM itself. This is because of one fundamental problem: keeping track of deallocated bits needs allocating memory, which inturn means deallocating that memory, which means we need to allocate memory to keep track of those bits, which means we need to deallocate memory... I think the problem is pretty obvious here.

2.3 How deallocation works

If ***size*** bits from the offset ***index*** have to be deallocated, the memory manager keeps track that ***size*** bits are available. This is done using an AVL tree, where the keys are the size available, and the value of a particular key is a queue-like data structure which keeps track of at what offsets are ***size*** bits available.

2.4 How allocation works

When ***size*** bits are requested, the smallest size greater than or equal to ***size*** is grabbed from the AVL tree, and the first value of the respective queue is dequeued. If the dequeued value is equal to ***size*** then it is returned. If it is not, then the excess bits are added back to the AVL tree. If all sizes in the AVL tree are lesser than the requested size, then ***global_offset*** is used to allocate the memory.

2.5 Null Values

Since WASM does not have an explicit null value, the 0th index of the linear memory is used as null. This means that ***global_offset*** begins with the default value of 4. The first 4 bytes of the linear memory must always be 0. It is important to note here that ***global_offset*** may not always begin at 4 due to string pooling. Refer to 5.2.2 and 6.1 to know more.

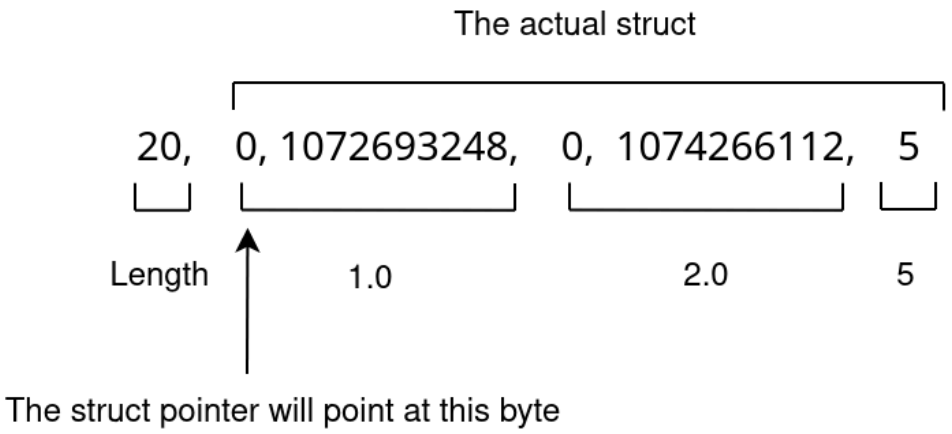
3. Struct

3.1 How structs are stored

When a struct is initiated, its total size is calculated and that information is used to allocate the struct. For example, consider the following struct:

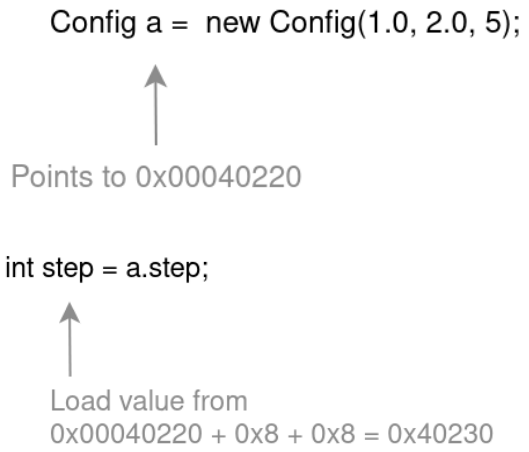
```
struct Config {
 double start;
 double end;
 int step;
}
```

Its total size is (8 + 8 + 4) = 20 bits. However, we need to track how big the struct is, so we allocate the first 4 bytes to store the struct's length. So, we allocate 20 + 4 bits in total. If the struct was initialized like **new Config(1.0, 2.0, 5)** then the linear memory will look like:



3.2 Assigning and reading a struct's field

When a particular needs to be read, all we need to do is add the appropriate byte offset to the struct's pointer. For example, consider the following code:



Values are assigned in a similar way; the cumulative is calculated and then the value is assigned.

3.3 Deletion of structs

The length that was stored right before the struct's pointer is used to deallocate it. For example, consider the struct defined in the previous section ***Config,*** when **delete a;** is called, the 4 bytes before ***a*** are read to find how many bits need to be deallocated, then ***deallocate_memory*** is called with the appropriate offset and size. The length of the struct is deallocated along with the struct.

3.4 Accessing field of null

Since null's value is 0, we can detect when we try reading anything from null. For example, consider the following code:

```
Struct a {
    int b;
}
```

```
function void main() {
    a instance = null;
    print(a.b);
}
```

Value of ***a*** would be 0, which means it is null. Thus when calculating the offset of the fieldname ***b*** we can check that the offset of the struct isn't 0. If it is, we can safely assume that ***a*** is null and throw an error using WASM's ***unreachable***.

4. Arrays

4.1 How Arrays are stored

Arrays are stored in a similar fashion as structs; the length is stored in the 4 bytes before the pointer of the array. However, there's one important distinction: the length is not the number of bits the arrays take, but rather the number of elements it has. For example, ***new int[10]*** would store the length as 10 and not as 40.

4.2 Types of Arrays

There are four base types, and they are either 4-bytes or 8-bytes. And since pointers are also 4-bytes, we only need arrays containing 4-bytes values or 8-bytes values.

4.3 Deletion of Arrays

The length of the array, and the size of its value (either ***i32*** or ***i64***) are used to delete the array.

4.4 Accessing elements of null

Similar to how we detect that structs are null, we can use the same technique for arrays. Before accessing the i^{th} element, we can check whether the offset of the array pointer is 0 or not. If it is, we can throw an error.

5. Strings

Strings, at their core, are just int arrays, so refer to the array section to learn more. However, the way characters are stored is different from other programming languages to make the implementation more intuitive and less complex.

5.1 How Strings are Stored

The way strings in MyWASM are stored is similar to char arrays in C, which means each element of the array represents a character. The way they are different is that MyWASM supports – utf-8 which is a variable-length character encoding standard. However, it always allocates 4-bytes for each character, thus wasting memory. For the sake of simplicity, I don't plan on changing that.

5.2 String pooling

String literals are stored using WASM's data section.

5.2.1. Why?

If MyWASM were to use ***string_ini*** to initiate the string, and then call ***string_ini_assign*** to assign each character for string literals, then that'd mean for a string that's considerably long, it'd take multiple opcodes to assign each character. Hence, it's more efficient to use WASM's data section. Also, optimizing the code would take considerably longer.

5.2.2. How?

The string is converted to binary during compilation. It is iterated, and each character is converted to a hex bytestring of length 8. It is highly important to note that WASM is little-endian (https://github.com/WebAssembly/design/issues/1212), and thus the bytestring must be in little-endian as well.

The bytestring in the data_section is copied to WASM's linear memory. Thus, it means that ***global_offset*** must be incremented to reflect this. The offset of a particular string literal is stored by the compiler and reused whenever the string literal is referenced. Since MyWASM supports imports, we also need to consider other files' ***global_offset***. Refer to 6.1 to know more.

5.3 When are strings copied?

Since strings are a base type, and at the same time pointers, passing them to functions, and assigning them to other variables can get pretty unintuitive really quickly. One option is to pass the pointer to the string, and the other is to copy the string each time it's referenced.

In my opinion, the latter option is more intuitive than the former. However, the major downside of copying the string is that it is more prone to memory leaks. This is because whenever a string is assigned or passed to a function, they must be deleted using the ***delete*** keyword. Consider the following code:

```
string a = "hello world";
string b = a;
some_function(b);
delete b;
delete a;

void some_function(string str) {
    .
    .
    delete str;
}
```

When ***a*** is assigned to ***b***, **a** is actually getting copied to a new string, which is then getting assigned to ***b***. Similarly, when ***b*** is passed to ***some_function***, it is getting copied, and then the pointer to the new string is getting passed to it. It is important to do this because strings in MyWASM are mutable, just like C char arrays.

5.4 String concatenation

5.4.1 How it works

When concatenating two strings, the length of string is added and then an appropriate amount of memory is allocated. Then both strings are copied to the new memory location (using WASM's ***memory.copy***), and then the new string pointer is returned.

5.4.2 Potential problems with string concatenation

Other than the problem mentioned in 5.3, string concatenation is probably what makes strings trickier than other base types. At first, I considered not supporting string concatenation at all, since it was really easy to leak memory. For example, consider the following code:

```
string a = “hello”;
a = a + “ world”;
```

Here, ***a*** is being assigned itself after concatenating a string literal to it. However, since concatenation returns a new string pointer, the reference to ***a***'s original pointer to is lost, thus resulting in a memory leak. The correct way of doing this would be:

```
string tmp = a;
delete a;
a = tmp + " world";
delete tmp;
```

This is not ideal, however, there are three alternatives I could think of: either detect that a string is being concatenated and then assigned to itself, remove string concatenation, or do nothing. The first option would add unnecessary complexity to the compiler, and add a lot of overhead. So, I decided to do nothing about this, and rely on the programmer to clean up strings.

5.5 Deletion of Strings

Since strings are just int arrays, we can just use the same method to delete it.

6. Imports and Namespaces

Namespaces are implemented using a similar technique mentioned in 1.4. Prefixes are used to differentiate between different namespaces' functions and structs.

To import a file and give it a namespace, we just need to write the following code:

```
Import "File.mypl" as Namespace;
```

Then the functions and structs can be accessed using the scope operator (::):

```
Namespace::some_function(10);
Namespace::some_struct a = new Namespace::some_struact(null);
```

6.1. String pooling and imports

Since string pooling changes the ***global_offset***, we need to add the calculated ***global_offset***s of all the imported files and then add them up, and then use that value as the new ***global_offset***.