Using the compiler

Similar to TypeScript's `tsc` transpiling to JavaScript, AssemblyScript's `asc` compiles to WebAssembly.

Compiler options

The compiler supports various options available on the command line, in a configuration file and programmatically. On the command line, it looks like this:

Entry file(s)

Non-option arguments are treated as the names of entry files. A single program can have multiple entries, with the exports of each entry becoming the exports of the WebAssembly module. Exports of imported files that are not entry files do not become WebAssembly module exports.

```sh
asc entryFile.ts
```

General

```
--version, -v         Prints just the compiler's version and exits.
--help, -h            Prints this message and exits.
--config              Configuration file to apply. CLI arguments take precedenc
--target              Configuration file target to use. Defaults to 'release'.
```

Optimization

The compiler can optimize for both speed (`-Ospeed`) and size (`-Osize`), as well as produce a debug build (`--debug`).

```
                        Make a release build    -O --noAssert
                        Make a debug build      --debug
                        Optimize for speed      -Ospeed
                        Optimize for size       -Osize

--optimizeLevel         How much to focus on optimizing code. [0-3]
--shrinkLevel           How much to focus on shrinking code size. [0-2, s=1, z=2]
--converge              Re-optimizes until no further improvements can be made.
--noAssert              Replaces assertions with just their value without trappin
```

Optimization levels can also be tweaked manually: `--optimizeLevel` (0-3) indicates how much the compiler focuses on optimizing the code with `--shrinkLevel` (0-2, 1=s, 2=z) indicating how much it focuses on keeping the size low during code generation and while optimizing. A shorthand for both is `-O[optimizeLevel][shrinkLevel]`, with shrink level indicated by optionally appending the letter `s` (1) or `z` (2).

Output

Typical output formats are WebAssembly binary (.wasm, `--outFile`) and/or text format (.wat, `--textFile`). Often, both are used in tandem to run and also inspect generated code.

```
--outFile, -o           Specifies the WebAssembly output file (.wasm).
--textFile, -t          Specifies the WebAssembly text output file (.wat).
--bindings, -b          Specifies the bindings to generate (.js + .d.ts).

                          esm  JavaScript bindings & typings for ESM integration.
                          raw  Like esm, but exports just the instantiate functio
                               Useful where modules are meant to be instantiated
                               multiple times or non-ESM imports must be provided
```

Debugging

For easier debugging during development, a **source map** can be emitted alongside the WebAssembly binary, and debug symbols can be embedded:

Features

There are several flags that enable or disable specific WebAssembly or compiler features. By default, only the bare minimum is exposed, and fully standardized WebAssembly features will be used.

```
--importMemory          Imports the memory from 'env.memory'.
--noExportMemory        Does not export the memory as 'memory'.
--initialMemory         Sets the initial memory size in pages.
--maximumMemory         Sets the maximum memory size in pages.
--sharedMemory          Declare memory as shared. Requires maximumMemory.
--zeroFilledMemory      Assume imported memory is zeroed. Requires importMemory.
--importTable           Imports the function table from 'env.table'.
--exportTable           Exports the function table as 'table'.
--exportStart           Exports the start function using the specified name inste
                        of calling it implicitly. Useful for WASI or to obtain th
                        exported memory before executing any code accessing it.
--runtime               Specifies the runtime variant to include in the program.

                          incremental  TLSF + incremental GC (default)
                          minimal      TLSF + lightweight GC invoked externally
                          stub         Minimal runtime stub (never frees)
                          ...          Path to a custom runtime implementation

--exportRuntime         Exports the runtime helpers (__new, __collect etc.).
--stackSize             Overrides the stack size. Only relevant for incremental G
                        or when using a custom runtime that requires stack space.
                        Defaults to 0 without and to 16384 with incremental GC.
--enable                Enables WebAssembly features being disabled by default.

                          threads             Threading and atomic operations.
                          simd                SIMD types and operations.
                          reference-types     Reference types and operations.
                          gc                  Garbage collection (WIP).

--disable               Disables WebAssembly features being enabled by default.

                          mutable-globals     Mutable global imports and exports.
                          sign-extension      Sign-extension operations
                          nontrapping-f2i     Non-trapping float to integer ops.
                          bulk-memory         Bulk memory operations.
```

```
--lowMemoryLimit      Enforces very low (<64k) memory constraints.
```

Linking

Specifying the base offsets of compiler-generated memory respectively the table leaves some space for other data in front. In its current form this is mostly useful to link additional data into an AssemblyScript binary after compilation, be it by populating the binary itself or initializing the data externally upon initialization. One good example is leaving some scratch space for a frame buffer.

```
--memoryBase          Sets the start offset of emitted memory segments.
--tableBase           Sets the start offset of emitted table elements.
```

API

To integrate with the compiler, for example to post-process the AST, one or multiple custom transforms can be specified.

```
--transform           Specifies the path to a custom transform to load.
```

Other

Other options include those forwarded to Binaryen and various flags useful in certain situations.

Binaryen

```
--trapMode            Sets the trap mode to use.

                       allow  Allow trapping operations. This is the default.
                       clamp  Replace trapping operations with clamping semanti
                       js     Replace trapping operations with JS semantics.

--runPasses           Specifies additional Binaryen passes to run after other
```

And the kitchen sink

```
--baseDir             Specifies the base directory of input and output files.
--noColors            Disables terminal colors.
--noUnsafe            Disallows the use of unsafe features in user code.
                      Does not affect library files and external modules.
--noEmit              Performs compilation as usual but does not emit code.
--stats               Prints statistics on I/O and compile times.
--pedantic            Make yourself sad for no good reason.
--lib                 Adds one or multiple paths to custom library components a
                      uses exports of all top-level files at this path as globa
--path                Adds one or multiple paths to package resolution, similar
                      to node_modules. Prefers an 'ascMain' entry in a package'
                      package.json and falls back to an inner 'assembly/' folde
--wasm                Uses the specified Wasm binary of the compiler.
-- ...                Specifies node.js options (CLI only). See: node --help
```

Configuration file

Instead of providing the options outlined above on the command line, a configuration file typically named **asconfig.json** can be used. It may look like in the following example, excluding comments:

```json
{
  "extends": "./path/to/other/asconfig.json", // (optional) base config
  "entries": [
    // (optional) entry files, e.g.
    "./assembly/index.ts"
  ],
  "options": {
    // common options, e.g.
    "importTable": true
  },
  "targets": {
    // (optional) targets
    "release": {
      // additional options for the "release" target, e.g.
      "optimize": true,
      "outFile": "myModule.release.wasm"
```

```
      "outFile": "myModule.debug.wasm"
    }
  }
}
```

Per-target options, e.g. `targets.release`, add to and override top-level `options`. Options provided on the command line override any options in the configuration file. Usage is, for example:

```sh
asc --config asconfig.json --target release
```

Programmatic usage

The compiler API can also be used programmatically:

```js
import asc from "assemblyscript/asc";

const { error, stdout, stderr, stats } = await asc.main([
  // Command line options
  "myModule.ts",
  "--outFile", "myModule.wasm",
  "--optimize",
  "--sourceMap",
  "--stats"
], {
  // Additional API options
  stdout?: ...,
  stderr?: ...,
  readFile?: ...,
  writeFile?: ...,
  listFiles?: ...,
  reportDiagnostic?: ...,
  transforms?: ...
});
if (error) {
  console.log("Compilation failed: " + error.message);
  console.log(stderr.toString());
} else {
  console.log(stdout.toString());
}
```

```
<script async src="https://cdn.jsdelivr.net/npm/es-module-shims@1/dist/es-modul
<script type="importmap">
{
  "imports": {
    "binaryen": "https://cdn.jsdelivr.net/npm/binaryen@x.x.x/index.js",
    "long": "https://cdn.jsdelivr.net/npm/long@x.x.x/index.js",
    "assemblyscript": "https://cdn.jsdelivr.net/npm/assemblyscript@x.x.x/dist/a
    "assemblyscript/asc": "https://cdn.jsdelivr.net/npm/assemblyscript@x.x.x/di
  }
}
</script>
<script type="module">
import asc from "assemblyscript/asc";
...
</script>
```

Note that the matching versions of the respective dependencies need to be filled in instead of `x.x.x`. Current versions can be obtained from the generated **web.html** file. The **es-module-shims** dependency polyfills support for import maps where not yet available.

Host bindings

WebAssembly alone cannot yet transfer higher level data types like strings, arrays and objects over module boundaries, so for now some amount of glue code is required to exchange these data structures with the host / JavaScript.

The compiler can generate the necessary bindings using the `--bindings` command line option (either as an ES module or a raw instantiate function), enabling exchange of:

Type	Strategy	Description
Number	By value	Basic numeric types except 64-bit integers.
BigInt	By value	64-bit integers via js-bigint-integration.
Boolean	By value	Coerced to `true` or `false`.
Externref	By reference	Using reference-types.
String	Copy	

TypedArray	Copy	Any `Int8Array`, `Float64Array` etc.
Array	Copy	Any `Array<T>`
StaticArray	Copy	Any `StaticArray<T>`
Object	Copy	If a plain object. That is: Has no constructor or non-public fields.
Object	By reference	If not a plain object. Passed as an opaque reference counted pointer.

Note the two different strategies used for **Object**: In some situations, say when calling a Web API, it may be preferable to copy the object as a whole, field by field, which is the strategy chosen for plain objects with no constructor or non-public fields:

```ts
// Copied to a JS object
class PlainObject {
  field: string;
}

export function getObject(): PlainObject {
  return {
    field: "hello world"
  };
}
```

However, copying may not be desirable in every situation, say when individual object properties are meant to be modified externally where serializing/deserializing the object as a whole would result in unnecessary overhead. To support this use case, the compiler can pass just an opaque reference to the object, which can be enforced by providing an empty `constructor` (not a plain object anymore):

```ts
// Not copied to a JS object
class ComplexObject {
  constructor() {} // !
  field: string | null;
}

export function newObject(): ComplexObject {
  return new ComplexObject();
```

```
}

export function getObjectField(target: ComplexObject): string | null {
  return target.field;
}
```

Also note that exporting an entire `class` has no effect at the module boundary (yet), and it is instead recommended to expose only the needed functionality as shown in the example above. Supported elements at the boundary are globals, functions and enums.

Using ESM bindings

Bindings generated with `--bindings esm` perform all the steps from compilation over instantiation to exporting the final interface. To do so, a few assumptions had to be made:

- The WebAssembly binary is located next to the JavaScript bindings file using the same name but with a `.wasm` extension.

  ```
  build/mymodule.js
  build/mymodule.wasm
  ```

  ```
  import * as myModule from "./build/mymodule.js"
  ```
  js

- JavaScript globals in `globalThis` can be accessed directly via the `env` module namespace. For example, `console.log` can be manually imported through:

  ```
  @external("env", "console.log")
  declare function consoleLog(s: string): void
  ```
  ts

  Note that this is just an example and `console.log` is already provided by the standard library when called from an AssemblyScript file. Other global functions not already provided by the standard library may require an import as of this example, though.

- Imports from other namespaces than `env`, i.e. `(import "module" "name")`, become an `import { name } from "module"` within the binding. Importing a custom function from a

```
@external("./otherfile.js", "myFunction")
declare function myFunction(...): ...
```

Similarly, importing a custom function from, say, a Node.js dependency can be achieved through:

```ts
@external("othermodule", "myFunction")
declare function myFunction(...): ...
```

These assumptions cannot be intercepted or customized since, to provide static ESM exports from the bindings file directly, instantiation must start immediately when the bindings file is imported. If customization is required, `--bindings raw` can be used instead.

Using raw bindings

The signature of the single `instantiate` function exported by `--bindings raw` is:

```ts
export async function instantiate(module: WebAssembly.Module, imports?: WebAsse
```

Note that the function does not make any assumptions on how the module is to be compiled, but instead expects a readily compiled `WebAssembly.Module` as in this example:

```js
import { instantiate } from "./module.js"
const exports = await instantiate(await WebAssembly.compileStreaming(fetch("./m
```

Unlike `--bindings esm`, raw bindings also do not make any assumptions on how imports are resolved, so these must be provided manually as part of the imports object. For example, to achieve a similar result as with ESM bindings, but now customizable:

```js
import { instantiate } from "./module.js"
import * as other from "./otherfile.js"
export const {
  export1,
  export2,
```

Debugging

The debugging workflow is similar to debugging JavaScript since both Wasm and JS execute in the same engine, and the compiler provides various options to set up additional WebAssembly-specific debug information. Note that any sort of optimization should be disabled in debug builds.

Debug symbols

When compiling with the `--debug` option, the compiler appends a name section to the binary, containing names of functions, globals, locals and so on. These names will show up in stack traces.

Source maps

The compiler can generate a source map alongside a binary using the `--sourceMap` option. By default, a relative source map path will be embedded in the binary which browsers can pick up when instantiating a module from a `fetch` response. In environments that do not provide `fetch` or an equivalent mechanism, like in Node.js, it is alternatively possible to embed an absolute source map path through `--sourceMap path/to/source/map`.

Breakpoints

Some JavaScript engines also support adding break points directly in WebAssembly code. Please consult your engine's documentation: Chrome, Firefox, Node.js, Safari.

Transforms

AssemblyScript is compiled statically, so code transformation cannot be done at runtime but must instead be performed at compile-time. To enable this, the compiler frontend (asc)

Specifying `--transform ./myTransform.js` on the command line will load the node module pointed to by `./myTransform.js`.

```js
import * as assemblyscript from "assemblyscript"
import { Transform } from "assemblyscript/transform"
class MyTransform extends Transform {
  ...
}
export default MyTransform
```

Properties

A transform is an ES6 class/node module with the following inherited properties:

- `readonly program: Program`

  Reference to the `Program` instance.

- `readonly baseDir: string`

  Base directory used by the compiler.

- `readonly stdout: OutputStream`

  Output stream used by the compiler.

- `readonly stderr: OutputStream`

  Error stream uses by the compiler.

- `readonly log: typeof console.log`

  Logs a message to console.

- `function writeFile(filename: string, contents: string | Uint8Array`

  Writes a file to disk.

- `function readFile(filename: string, baseDir: string): string | null`

  Reads a file from disk.

Lists all files in a directory.

Hooks

The frontend will call several hooks, if present on the transform, during the compilation process:

- `function afterParse(parser: Parser): void`

  Called when parsing is complete, before a program is initialized from the AST. Note that types are not yet known at this stage and there is no easy way to obtain them.

- `function afterInitialize(program: Program): void`

  Called once the program is initialized, before it is being compiled. Types are known at this stage, respectively can be resolved where necessary.

- `function afterCompile(module: Module): void`

  Called with the resulting module before it is being emitted. Useful to modify the IR before writing any output, for example to replace imports with actual functionality or to add custom sections.

Transforms are a very powerful feature, but may require profound knowledge of the compiler to utilize them to their full extent, so reading through the compiler sources is a plus.

Portability

With AssemblyScript being very similar to TypeScript, there comes the opportunity to compile the same code to JavaScript with `tsc` and WebAssembly with `asc`. The AssemblyScript compiler itself is portable code. Writing portable code is largely a matter of double-checking that the intent translates to the same outcome in both the strictly typed AssemblyScript and the types-stripped-away TypeScript worlds.

Portable standard library

like the portable conversions mentioned below.

Also note that some parts of JavaScript's standard library function a little more loosely than how they would when compiling to WebAssembly. While the portable definitions try to take care of this, one example where this can happen is `Map#get` returning `undefined` when a key cannot be found in JavaScript, while resulting in an abort in WebAssembly, where it is necessary to first check that the key exists using `Map#has`.

To use the portable library, extend `assemblyscript/std/portable.json` instead of `assemblyscript/std/assembly.json` within `tsconfig.json` and add the following somewhere along your build step so the portable features are present in the environment:

```js
import "assemblyscript/std/portable.js"
```

Note that the portable standard library is still a work in progress and so far focuses on functionality useful to make the compiler itself portable, so if you need something specific, feel free to improve its definitions and feature set.

Portable conversions

While `asc` understands the meaning of

```ts
// non-portable
let someFloat: f32 = 1.5
let someInt: i32 = <i32>someFloat
```

and then inserts the correct conversion steps, `tsc` does not because all numeric types are just aliases of `number`. Hence, when targeting JavaScript with `tsc`, the above will result in

```js
var someFloat = 1.5
var someInt = someFloat
```

which is obviously wrong. To account for this, portable conversions can be used, resulting in actually portable code. For example

will essentially result in

```js
var someFloat = 1.5
var someInt = someFloat | 0
```

which is correct. The best way of dealing with this is asking yourself the question: What would this code do when compiled to JavaScript?

Portable overflows

Likewise, again because `asc` knows the meaning but `tsc` does not, overflows must be handled explicitly:

```ts
// non-portable
let someU8: u8 = 255
let someOtherU8: u8 = someU8 + 1
```

```ts
// portable
let someU8: u8 = 255
let someOtherU8: u8 = u8(someU8 + 1)
```

essentially resulting in

```js
let someU8 = 255
let someOtherU8 = (someU8 + 1) & 0xff
```

Non-portable code

In JavaScript, all numeric values are IEEE754 doubles that cannot represent the full range of values fitting in a 64-bit integer (**max. safe integer** is `2^53 - 1`). Hence `i64` and `u64` are not portable and not present in `std/portable`. There are several ways to deal with this. One is to use an i64 polyfill like **in this example**.

expected to be relatively specific (for instance: the portable compiler accesses Binaryen's memory).

Edit this page on GitHub

← Getting started

Concepts →