

*Wenn du lieber gleich mit der Übung starten möchtest, kannst du den Theorieteil auch später lesen. Stell nur sicher, dass du ihn später auch liest. 😊*

## Wieso WebAssembly?

Die Idee hinter WebAssembly ist: Software in Programmiersprachen wie Go, Rust, C++ oder sogar C# zu schreiben und sie im Browser laufen zu lassen. Ohne vorherige Installation und mit fast der gleichen Geschwindigkeit wie Desktopanwendungen.

Netter Gedanke, nur warum sollte das jemand interessieren, es gibt doch schon JavaScript?

Nun, Anwendungen, die im Browser laufen, statt auf der Festplatte installiert zu sein, werden immer mehr und sie werden immer umfangreicher. Denk mal an: VS Code, Google Docs/Maps/Gmail/Classroom, MS Office 365, Bildbearbeitung im Browser, Browser Games, Simulationen, etc. Gleichzeitig gibt es kaum mehr reine Desktopanwendungen und sie werden immer weniger.

Es sind vor allem die Vorteile von Webanwendungen, die diese Entwicklung begünstigen:

- kein (bzw. geringer) Installationsaufwand
- immer aktuell durch Updates
- Erscheinungsbild und Funktionsweise sind unabhängig von Betriebssystem und Endgeräten
- portabler Code
- inhärente Client/Server Struktur, welche neue Möglichkeiten der Zusammenarbeit mit anderen Usern bietet
- Speicher am Server ist de facto unlimitiert, automatisches Backup

Für Unternehmen gibt es aber auch andere Vorteile:

- Durch eine Umstellung auf ein Abomodell ergibt:
  - o eine geringe Einstiegshürde für die Entscheidung zur Benutzung der Anwendung
  - o ein gleichmäßiger Geldfluss für das Unternehmen
- eine Möglichkeit der Auswertung und Verknüpfung von Kundendaten
- Feedback und damit die Auswertung der Art der Benutzung. Das ermöglicht die fortlaufende Adaptierung der Anwendung.

Allerdings gibt es auch Nachteile:

- Die Anwendung ist abhängig von der Internetverbindung. Das versucht man mit einem Offline Modus zu minimieren.
- **Die Performance ist deutlich schlechter als bei Desktop Anwendungen.**

Also **Performance** ist der Grund. Bei den meisten Webanwendungen spielt Performance nicht so eine große Rolle, aber es gibt doch Fälle, wo es praktisch wäre, im Browser hohe Performance zu erzielen:

- Bild- und Videobearbeitung
- Spiele
- Musikanwendungen
- Bilderkennung
- VR und Augmented Reality
- Remote Desktop
- Verschlüsselung/Entschlüsselung
- ...

Aber wieso ist JavaScript langsamer als eine kompilierte Desktopanwendung?

JavaScript hat eine lange Entwicklung hinter sich. JavaScript wurde 1995 innerhalb von zwei Wochen von **Brendan Eich** erfunden, während er bei Netscape (später Mozilla) arbeitete. Damals waren Webseiten bis auf die `<blink>` und `<marquee>` Tags ohne Animation und ziemlich statisch<sup>1</sup>.

Marc Andressen, der CEO von Netscape hatte die Idee, dass das anders werden sollte. Animationen und Interaktionen sollten Teil des Webs der Zukunft sein. Die Zielgruppe für diese neue Sprache waren nicht Programmierer, sondern Designer und Laien<sup>2</sup>. Das war die Zielvorgabe für die Entwicklung von JavaScript!

Seitdem hat sich das Aufgabengebiet von JavaScript stark erweitert. Keine hätte sich damals die heutigen komplexen Frontendanwendungen mit zigtausend Zeilen Code vorstellen können!

Und auch JavaScript hat sich deutlich verändert. Doch die Art und Weise, wie der resultierende Code an den Browser des Clients gesendet und ausgeführt wird, seit den Anfängen ziemlich gleich geblieben.

Betrachten wir dazu eine typische moderne JavaScript-Anwendung: Der Code, den wir schreiben, durchläuft einige ziemlich ausgeklügelte Transformationen - er wird transpiliert (danke Babel), nicht benutzte Teile werden entfernt (danke tree-shaking), er wird gebündelt (danke Webpack bzw. Parcel und Co.) und minimiert.

Unabhängig von den verwendeten Werkzeugen sieht der Production Code, der schließlich an den Browser ausgeliefert wird, typischerweise so aus:



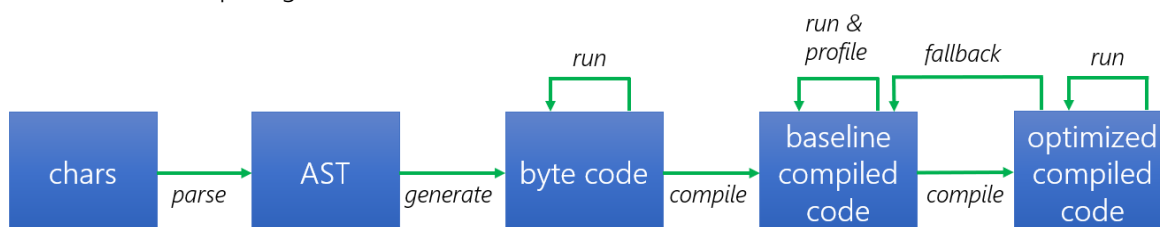
```

1 (function(t){function e(e){for(var r,i,c=e[0],l=e[1],u=e[2],d=e[3],f=0,p=[];f<c.length;f++)i=c[f],Object.prototype.hasOwnProperty.call(a,i)&&a[i]&&p.push(a[i][0]),a[i]=0;for(r in l)Object.prototype.hasOwnProperty.call(l,r)&&(t[r]=l[r]);m&&m(e),s.push.apply(s,d);while(p.length)p.shift();return o.push.apply(o,u|[]),n()}function n(){for(var t,e=0;e<o.length;e++){for(var n=o[e],r=!0,l=1;l<n.length;l++){var u=n[l];0!==a[u]&&(r=!1)}r&&(o.splice(e--,1),t=c[c.s=n[0]]);return 0===o.length&&(s.forEach(function(t){if(void 0===a[t]){a[t]=null;var e=document.createElement("link");c.nc&&e.setAttribute("nonce",c.nc),e.rel="prefetch",e.as="script",e.href=i(t),document.head.appendChild(e)}},s.length=0),t)}var r={},a={app:0},o=[],s=[];function i(t){return c.p+"js/"+({[t]:t}).+"71abdeb2"}function c(e){if(r[e])return r[e].exports;var n=r[e]={i:e,l:!1,exports:{}};return t[e].call(n,exports,n,n.exports,c),n.l=!0,n.exports;c.e=function(t){var e=[],n=a[t];if(0!==n)if(n)e.push(n[2]);else{var r=new Promise(function(e,r){n=a[t]=e,r});e.push(n[2]);r}}

```

Wenn ein Nutzer deine Webseite besucht, wird dieser JavaScript-Code über das HTTP-Protokoll angefordert und an seinen Browser gestreamt. Ist der Download abgeschlossen, beginnt der Prozess der Ausführung.

Der JavaScript-Code wird zunächst als eine Sammlung von Zeichen (**chars**) in den Speicher gelesen. Im ersten Schritt werden diese Zeichen, basierend auf der Grammatik der Sprache, in einen Abstract Syntax Tree (**AST**) geparkt, ein Format, das für die JavaScript-Engine des Browsers einfacher zu verarbeiten ist.

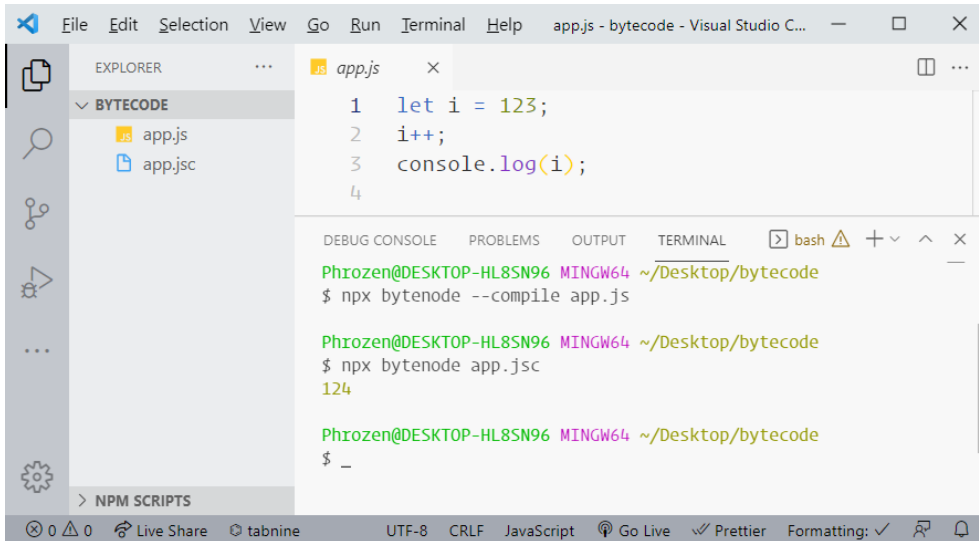


Der einfachste Weg, diesen Code auszuführen, ist die Umwandlung des **AST** in **byte code**, der interpretiert wird; dies führt jedoch auch zu einer schlechten Laufzeitleistung, was einer der Gründe ist, warum frühe Browser (die auf Interpreter angewiesen waren) langsam waren.

<sup>1</sup> <https://thehistoryoftheweb.com/blink-marquis-tag/>

<sup>2</sup> <https://auth0.com/blog/a-brief-history-of-javascript/>

Mit der npm Library **bytenode** kannst du selbst V8 Engine Bytecode erzeugen und auch ausführen.



The screenshot shows the Visual Studio Code interface with the Explorer sidebar on the left displaying a file named `app.js` under a folder named `BYTECODE`. The main editor window shows the content of `app.js`:

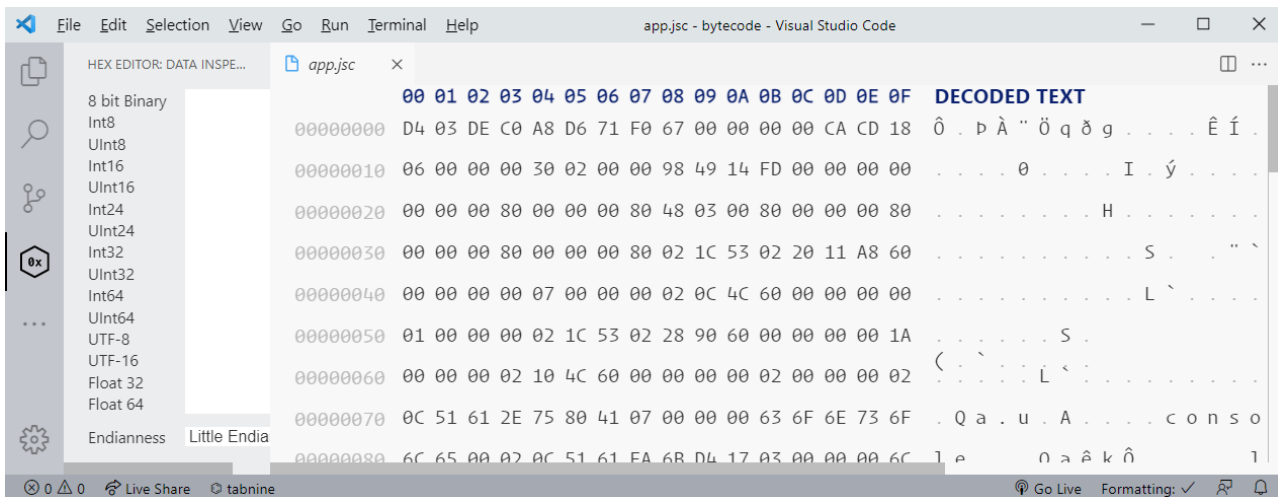
```
1 let i = 123;
2 i++;
3 console.log(i);
4
```

Below the editor, the TERMINAL panel is open, showing the execution of the `bytenode` command:

```
Phrozen@DESKTOP-HL8SN96 MINGW64 ~/Desktop/bytecode
$ npx bytenode --compile app.js

Phrozen@DESKTOP-HL8SN96 MINGW64 ~/Desktop/bytecode
$ npx bytenode app.js
124

Phrozen@DESKTOP-HL8SN96 MINGW64 ~/Desktop/bytecode
$ _
```



The screenshot shows the Visual Studio Code interface with the Explorer sidebar on the left displaying a file named `app.jsc` under a folder named `BYTECODE`. The main editor window is in the HEX EDITOR view, showing the binary data of `app.jsc` in hexadecimal and its decoded text representation:

Offset	Hex	Decoded Text
00000000	D4 03 DE C0 A8 D6 71 F0 67 00 00 00 00 CA CD 18	Ô . Þ Å " Ö q ð g . . . Ê Í .
00000010	06 00 00 00 30 02 00 00 98 49 14 FD 00 00 00 00	. . . . . 0 . . . . I . ý . . .
00000020	00 00 00 80 00 00 00 80 48 03 00 80 00 00 00 80	. . . . . H . . . . .
00000030	00 00 00 80 00 00 00 80 02 1C 53 02 20 11 A8 60	. . . . . S . . . . .
00000040	00 00 00 00 07 00 00 00 02 0C 4C 60 00 00 00 00	. . . . . L . . . . .
00000050	01 00 00 00 02 1C 53 02 28 90 60 00 00 00 00 1A	. . . . . S .
00000060	00 00 00 02 10 4C 60 00 00 00 00 02 00 00 00 02	( . . . . L < . . . . .
00000070	0C 51 61 2E 75 80 41 07 00 00 00 63 6F 6E 73 6F	. Q a . u . A . . . . c o n s o
00000080	6C 65 00 02 0C 51 61 FA 6B D4 17 03 00 00 00 6C	l e . n a ê k ô 1

Um die Laufzeitleistung zu verbessern, überwachen moderne Browser die Ausführung des Codes und indem sie bestimmte Annahmen treffen (z.B. könnte eine einfache Annahme sein, dass eine Variable immer von einem bestimmten Typ ist). Sie sind in der Lage, eine kompilierte Version des Codes zu erzeugen, die viel schneller ausgeführt wird (**optimized compiled code**). Dieser Prozess der Annahmen erlaubt es, eine optimierte Kompilierung zu erstellen, wobei der Code schrittweise zu höheren Optimierungstufen befördert wird. Wenn diese Annahmen jedoch fehlschlagen, fällt der Code auf die langsameren, weniger optimierten Stufen zurück (**fall back**).


Wichtig: Bytecode ist kein Maschinencode, sondern muss interpretiert werden (auch Java und C# erzeugen Bytecode, der von einer virtuellen Maschine (VM) interpretiert wird. Vergleiche dazu einen Compiler wie **gcc**, der aus C/C++ Maschinencode für den Zielprozessor erzeugt.

Nimm mal so ein einfaches C-Programm wie:

```
#include <stdio.h>

int main() {
    int i =123;
    i++;
    printf("%d",i);
    return 0;
}
```

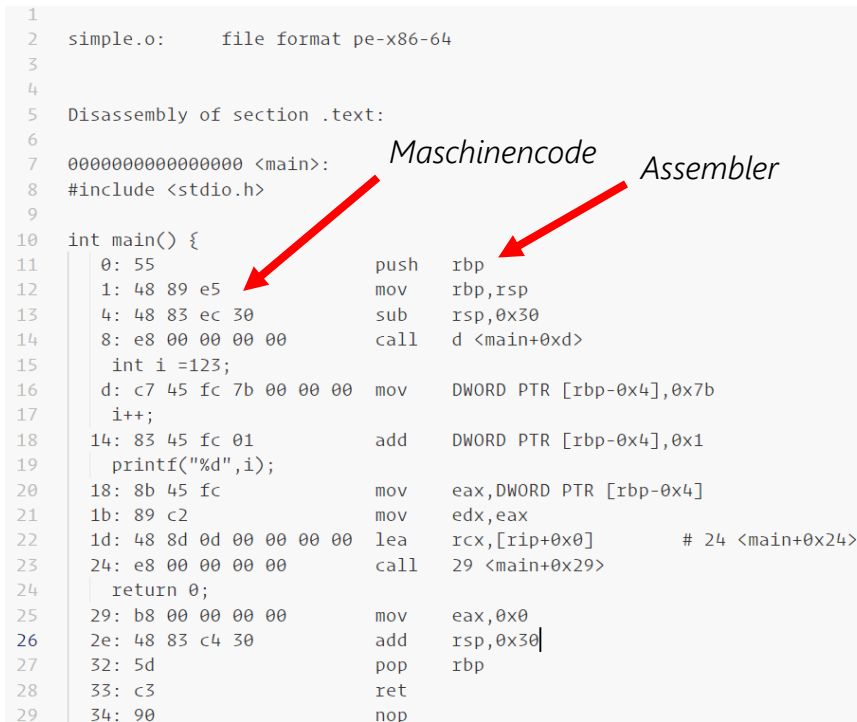
Übersetze es mit dem **gcc** Compiler: `gcc simple.c -o simple.exe`  
Das Programm **simple.exe** kann direkt ausgeführt werden!



```
C:\Windows\System32\cmd.exe

C:\TDM-GCC-64>simple
124
C:\TDM-GCC-64>
```

Du kannst dir aber auch das Zwischenergebnis als Maschinencode/Assemblercode ansehen:



```
1
2 simple.o:      file format pe-x86-64
3
4
5 Disassembly of section .text:
6
7 0000000000000000 <main>:
8 #include <stdio.h>
9
10 int main() {
11     0: 55                push    rbp
12     1: 48 89 e5          mov     rbp, rsp
13     4: 48 83 ec 30       sub     rsp, 0x30
14     8: e8 00 00 00 00    call    d <main+0xd>
15     int i = 123;
16     d: c7 45 fc 7b 00 00 00 mov     DWORD PTR [rbp-0x4], 0x7b
17     i++;
18     14: 83 45 fc 01       add     DWORD PTR [rbp-0x4], 0x1
19     printf("%d", i);
20     18: 8b 45 fc          mov     eax, DWORD PTR [rbp-0x4]
21     1b: 89 c2            mov     edx, eax
22     1d: 48 8d 0d 00 00 00 00 lea     rcx, [rip+0x0]      # 24 <main+0x24>
23     24: e8 00 00 00 00    call    29 <main+0x29>
24     return 0;
25     29: b8 00 00 00 00    mov     eax, 0x0
26     2e: 48 83 c4 30       add     rsp, 0x30
27     32: 5d              pop     rbp
28     33: c3              ret
29     34: 90              nop
```

Was ist der Unterschied zwischen Maschinencode und Bytecode? Maschinencode kann direkt vom Prozessor des Computers abgearbeitet werden. Bytecode ist gleichermaßen Maschinencode für einen Prozessor, der nicht existiert. Daher muss ein Programm (eine virtuelle Maschine) geschrieben werden, das diesen Bytecode als Maschinencode interpretieren kann.

Das kostet natürlich Performance. Daher ist Maschinencode immer schneller als Bytecode! Aber Bytecode ist unabhängig vom Prozessor und portabel!

## WebAssembly

**WebAssembly** wurde als ein neue Bytecode entwickelt. Es ist jedoch nicht als Bytecode-Format für JavaScript konzipiert, sondern als low-level portabler "Maschinencode" und Kompilierziel für Sprachen wie C, C++ und Rust. Siehe: <https://webassembly.org/docs/high-level-goals/>

Das wirft natürlich einige Fragen auf.

*Was ist der Unterschied zu dem JavaScript Bytecode?*

Nun, es gibt kein standardisiertes Bytecode-Format für JavaScript! Tatsächlich verwenden einige Implementierungen nicht einmal Bytecode! Zum Beispiel kompilierte V8 JavaScript in den ersten Jahren direkt in nativen Maschinencode, ohne einen Bytecode-Zwischenschritt. Chakra, SquirrelFish Extreme und SpiderMonkey verwenden alle Bytecode, aber sie verwenden unterschiedliche Bytecodes. dyn.js, TruffleJS, Nashorn und Rhine verwenden keinen JavaScript-spezifischen

Bytecode, sie kompilieren zu JVM-Bytecode (das ist der von Java). Ebenso kompiliert IronJS zu CLI CIL Bytecode (das ist der von C#).

*Warum wird der JavaScript Bytecode nicht standardisiert?*

Es ist nicht nur schwer, alle JavaScript-Engine-Anbieter dazu zu bringen, sich auf ein gemeinsames, standardisiertes Bytecode-Format zu einigen, sondern bedenke: Es macht keinen Sinn, dem Browser ein Bytecode-Format nur für JavaScript hinzuzufügen. Wenn du ein gemeinsames Bytecode-Format machst, wäre es schön, wenn es auch VBScript, Python, Ruby, Perl, Lua, PHP, etc. unterstützt. Also noch komplexer!

*Wird WebAssembly Javascript ersetzen?*

Aus <https://webassembly.org/docs/faq/>:

*No! WebAssembly is designed to be a complement to, not replacement of, JavaScript. While WebAssembly will, over time, allow many languages to be compiled to the Web, JavaScript has an incredible amount of momentum and will remain the single, privileged (as described above) dynamic language of the Web.*

*Wann kann man WebAssembly verwenden?*

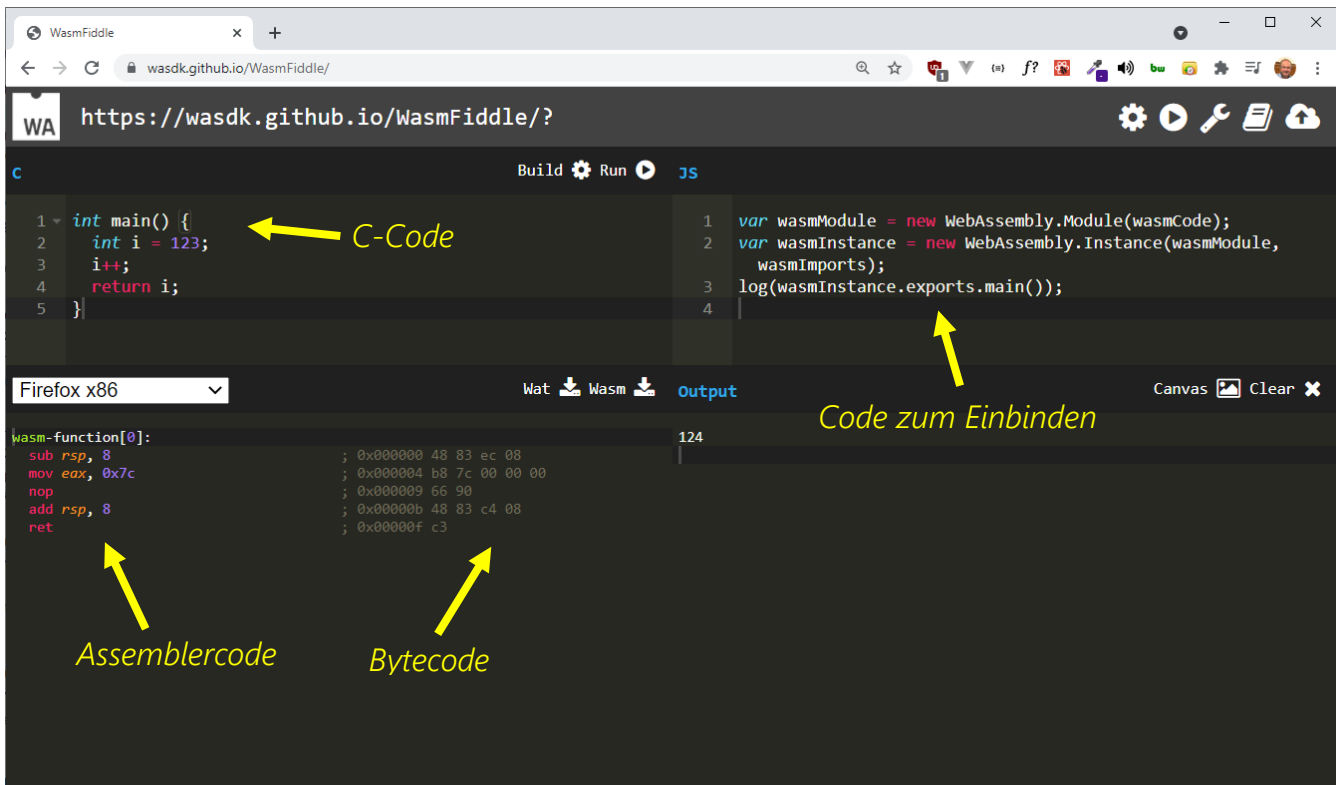
WebAssembly wird bereits in Firefox, Chrome, Safari und Edge ausgeliefert.

## Praktischer Teil

Da wir für das Beispiel C verwenden, ist es gut, wenn du zumindest die C-Datentypen kennst. Hier ist eine Liste:

<https://www.programiz.com/c-programming/c-data-types>

1. Wir beginnen zunächst mit einem Online Tool, welches für uns den Bytecode erstellt (später sehen wir uns den Compiler Emscripten an): <https://wasdk.github.io/WasmFiddle/>

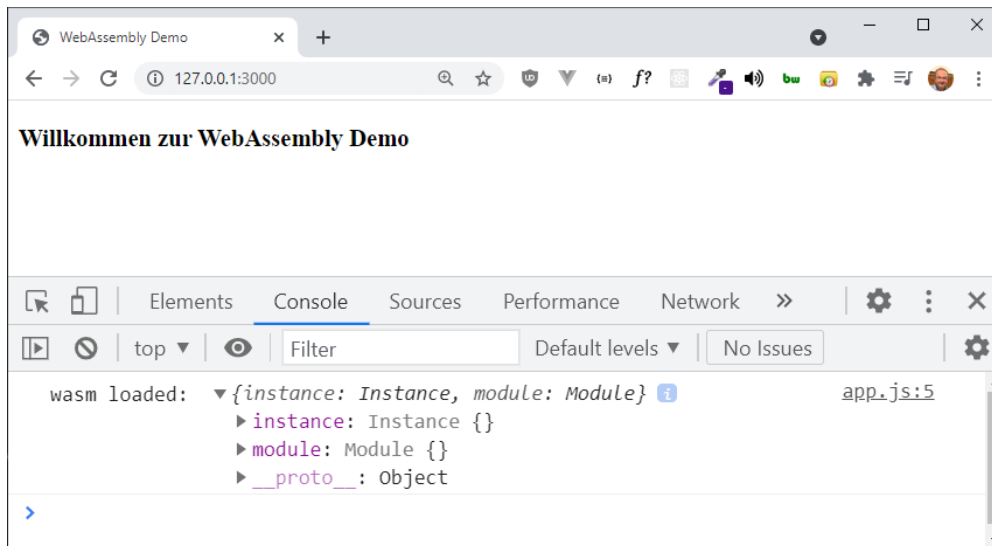


2. Entzippe die Datei **server start.zip** und lade den **.wasm** Code herunter.

Kopiere `program.wasm` nach `/public` und lade in `app.js` den Code. Wir verwenden `instantiateStreaming`, da dies die effizienteste Methode ist. `fetch` ist ein JavaScript Function ähnlich wie **Axios**.

```
const wasm = await WebAssembly.instantiateStreaming(fetch('/program.wasm'));
console.log('wasm loaded: ', wasm);
```

So sollte die Ausgabe aussehen:

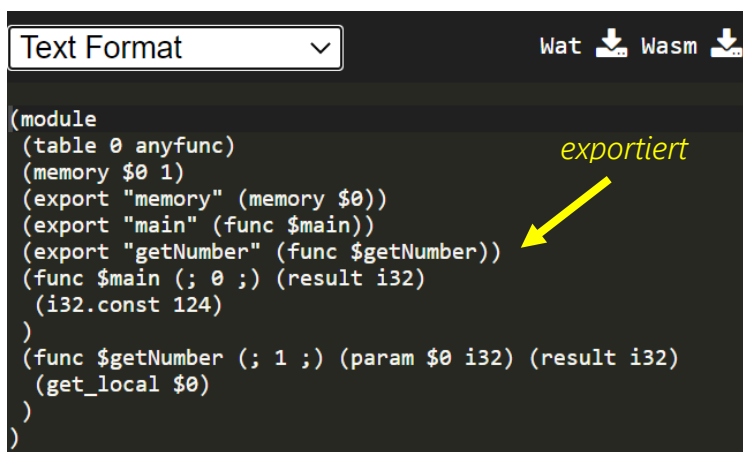


Der nächste Schritt ist nun eine C-Funktion von JavaScript aus aufzurufen.

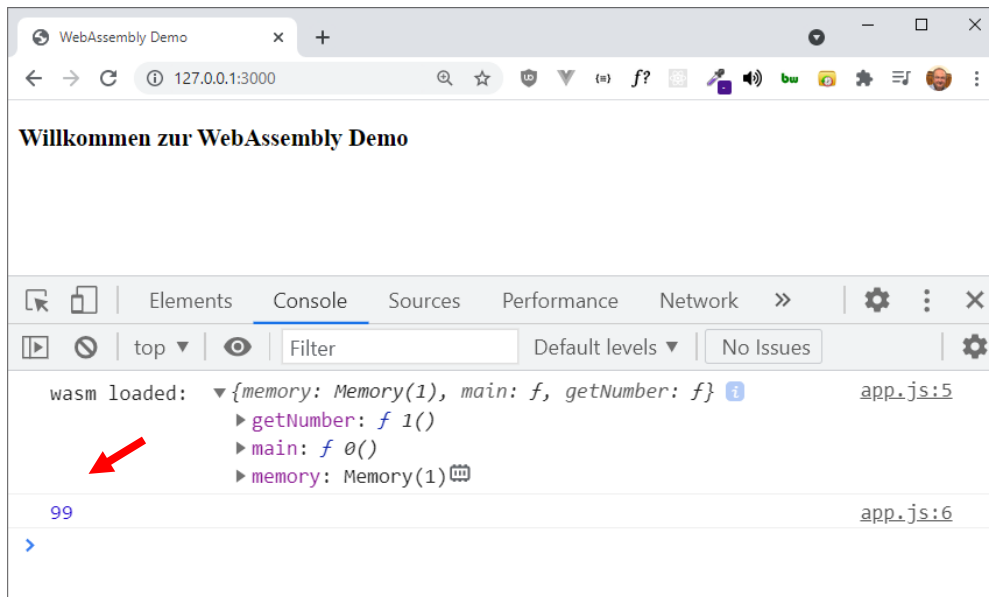
3. Füge noch eine Funktion zu `main` hinzu und führe Build aus:

```
int main() {
  int i = 123;
  i++;
  return i;
}

int getNumber(num) {
  return num;
}
```



4. Gib nun mittels `wasm.instance.exports` die exportierten Funktionen aus und rufe `getNumber` mit der Zahl 99 auf:



Wie sieht es aus, wenn wir JavaScript von C aus aufrufen? Als Beispiel verändern wir unsere C-Funktion, sodass die Zahl, die als Argument übergeben wird, quadriert wird. Wie können wir nun `console.log` in der C-Funktion aufrufen? Dazu müssen wir nach `fetch` als zweiten Parameter ein `imports` Objekt definieren.

5. Das Objekt muss ein Modul enthalten. Oft wird hier `env` verwendet, welches das Defaultmodul ist.

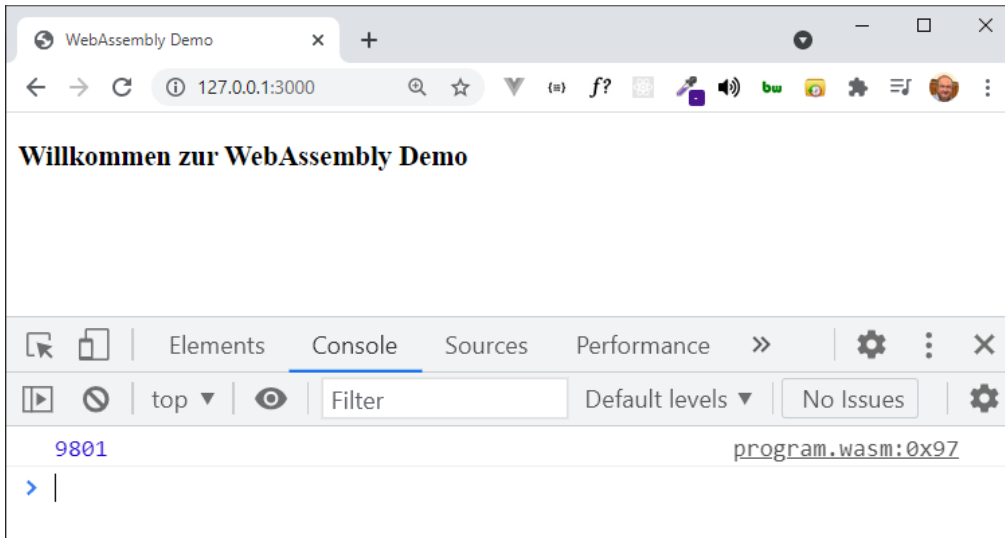
```
const imports = {
  env: {
    consoleLog: console.log,
  },
};
const wasm = await WebAssembly.instantiateStreaming(fetch('/program.wasm'), imports);
```

Im C-Code müssen wir die Funktionssignatur (Name, Returnwert, Parameter) definieren.

```
void consoleLog(int);
int main() {
}
void getNumberSquared(num) {
  consoleLog(num*num);
}
```

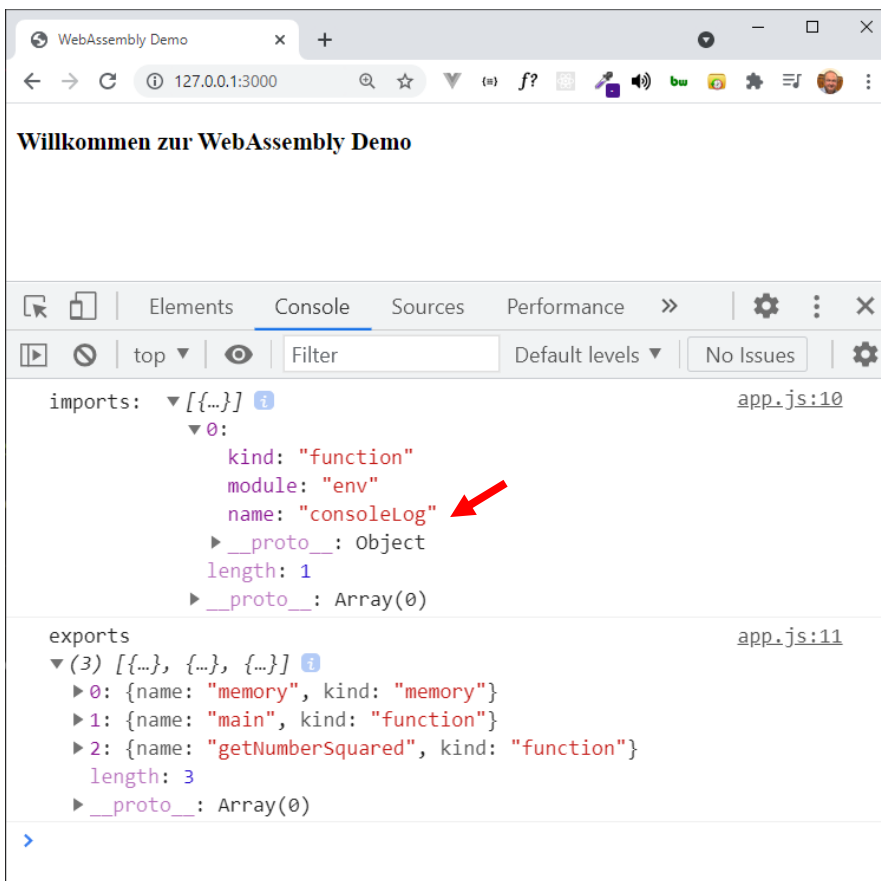
Nun noch die Funktion in `app.js` aufrufen.

```
wasm.instance.exports.getNumberSquared(99);
```



Exports und Imports sind im .wasm Modul definiert.

```
console.log('imports: ', WebAssembly.Module.imports(wasm.module));  
console.log('exports', WebAssembly.Module.exports(wasm.module));
```



Der nächste Teil ist etwas kompliziert und daher optional, aber du erfährst auch einige zusätzliche Aspekte von JavaScript wie **Typed ArrayBuffer** und **Custom Events**!



## Optionaler Teil

Nun die schockierende Wahrheit: WebAssembly kennt nur **int** und **floats**! Was, wenn wir Strings austauschen wollen? Dazu müssen wir den Memory-Buffer von WebAssembly verwenden. Wie du oben siehst, wird **memory** auch exportiert.

6. Erstelle zunächst eine weitere C-Funktion. Diese soll eine JavaScript Function aufrufen, die einen String in der Console ausgibt. Da Strings nicht Teil von WebAssembly sind, müssen wir uns selbst darum kümmern. Wir brauchen also eine eigene Funktion für Strings in JavaScript. Diese werden wir in das C-Programm importieren und dort aufrufen. Nennen wir diese Funktion **printStrConsole**.

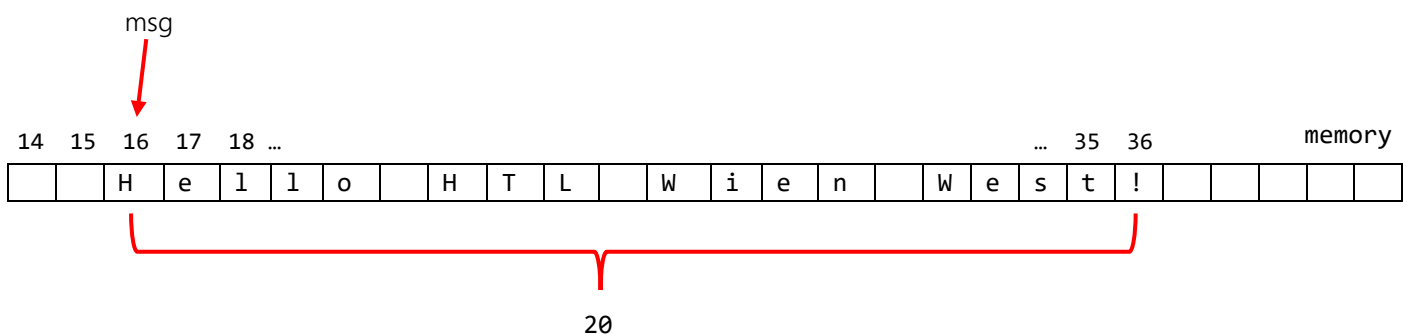
Erstelle in **WasmFiddle** folgende Funktion:

```
void printStrConsole(char * msg, int len);

int main() {
}

void greet() {
    printStrConsole("Hello HTL Wien West!", 20);
}
```

In dem obigen Beispiel ist **msg** ein Pointer (Zeiger) auf eine Adresse im Speicherbereich **memory**, ab der die Zeichen gespeichert sind. Da **memory** exportiert wird, kann die JavaScript Function **printStrConsole** darauf zugreifen. Allerdings müssen wir auch angeben wie lange die Zeichenkette in **memory** ist.

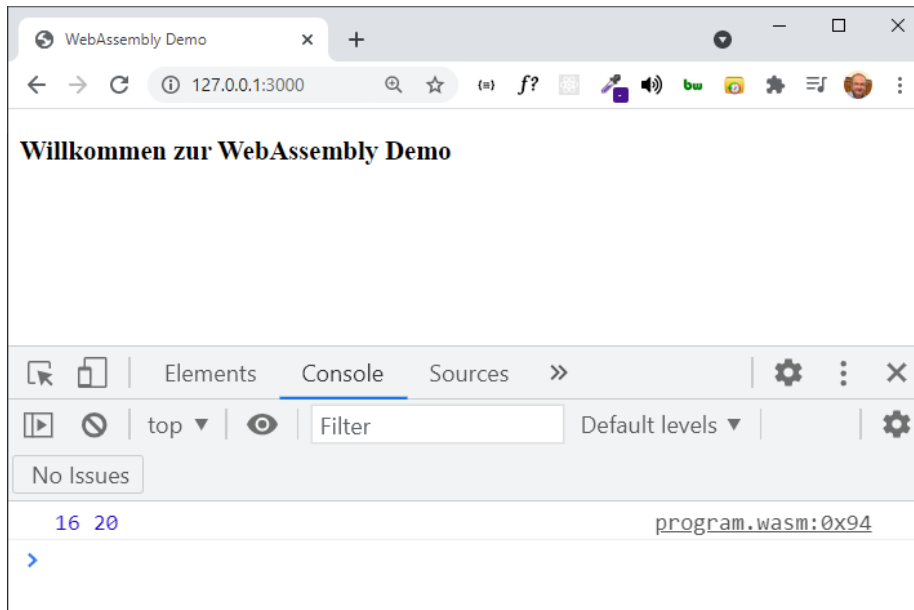


Erstelle **program.wasm** und kopiere die Datei nach **\public**. Ändere **app.js** so, dass **greet** aufgerufen wird:

```
const imports = {
  env: {
    printStrConsole: console.log,
  },
};

const wasm = await WebAssembly.instantiateStreaming(fetch('/program.wasm'), imports);
wasm.instance.exports.greet();
```

Aufruf:



Es werden nur die Zahlen 16 und 20 ausgegeben. Offensichtlich ist 16 die Position in **memory** und 20 ist die Länge, die wir übergeben haben.

Wie können wir in JavaScript auf **memory** zugreifen?

```
wasm.instance.exports.memory.buffer;
```

**buffer** ist ein **ArrayBuffer**, ein JavaScript Datentyp, der mit ES2015 eingeführt wurde<sup>3</sup>.

So ein ArrayBuffer ist einfach eine Folge von Bytes mit der fixen Länge von 65536 (= 64 kB, in WebAssembly: 1 Page). Da die Folge alles Mögliche sein kann, muss man spezifizieren, um welche Art von ArrayBuffer es sich handelt, bevor man auf die Bytes zugreifen kann.

Lies dazu [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/TypedArray](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypedArray).

Die Zeichen werden als eine Folge von **unsigned 8 Bit Integer** Werten von WebAssembly in **memory** geschrieben und folgen der UTF-8 Kodierung, die in unserem Fall im Prinzip ASCII Code<sup>4</sup> ist.

Hello HTL Wien West! → 72 101 108 108 111 32 72 84 76 32 87 105 101 110 32 87 101 115 116 33

Siehe: <https://gc.de/gc/ascii/>

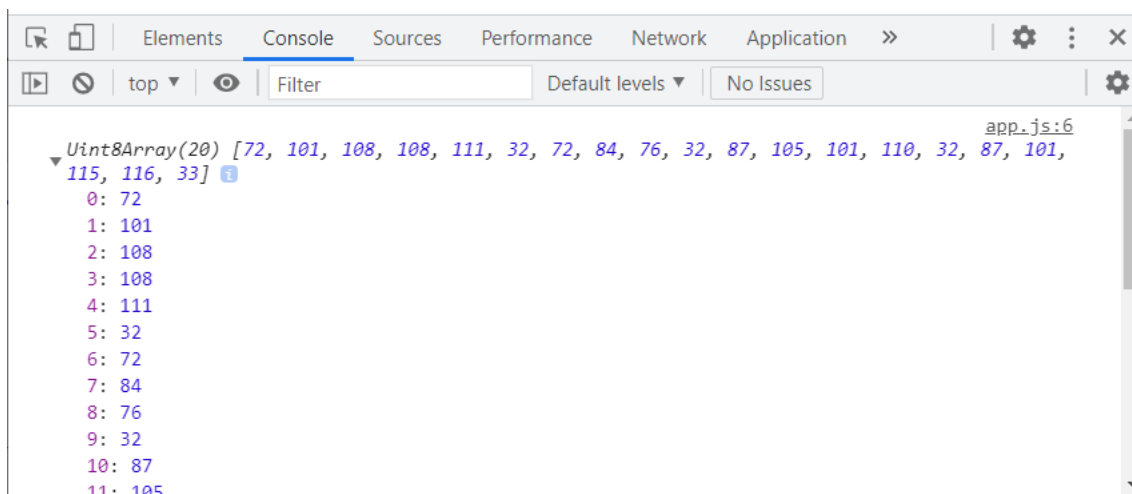
Daher bekommen wir hier ein Uint8Array via **memory** nach JavaScript.

Klar ist, **console.log** wird hier nicht brauchbar sein. wir müssen eine eigene Function schreiben, die auf **memory** zugreift und den ArrayBuffer umwandelt:

<sup>3</sup> Siehe: [https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global\\_Objects/ArrayBuffer](https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/ArrayBuffer)

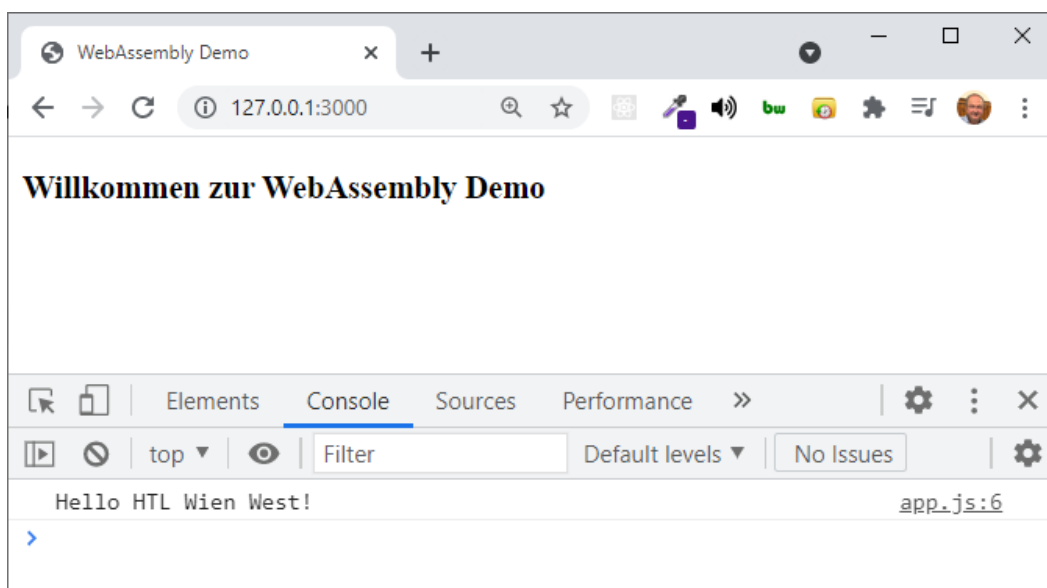
<sup>4</sup> Siehe: [https://de.wikipedia.org/wiki/American\\_Standard\\_Code\\_for\\_Information\\_Interchange](https://de.wikipedia.org/wiki/American_Standard_Code_for_Information_Interchange)

```
(async () => {  
  const readStr = (offset, len) => {  
    const strBuffer = new Uint8Array(wasm.instance.exports.memory.buffer, offset, len);  
    console.log(strBuffer);  
  };  
  
  const imports = {  
    env: {  
      printStrConsole: readStr,  
    },  
  };  
  
  const wasm = await WebAssembly.instantiateStreaming(fetch('/program.wasm'), imports);  
  wasm.instance.exports.greet();  
})();
```



Mit Hilfe der JavaScript Function `TextDecoder` können wir den String wieder in Zeichen umwandeln<sup>5</sup>.

```
console.log(new TextDecoder().decode(strBuffer));
```



<sup>5</sup> Siehe: <https://developer.mozilla.org/en-US/docs/Web/API/TextDecoder>

Toll, aber den String können wir nicht wirklich in **app.js** verwenden, da **ReadStr** ja in WebAssembly importiert wird und dort aufgerufen wird. Also ist es nicht möglich einfach einen Returnwert zu definieren. Aber wir können einen Event erzeugen. und zwar einen Custom Event und uns für diesen mit **addEventListener** registrieren!

```
const readStr = (offset, len) => {  
  const strBuffer = new Uint8Array(wasm.instance.exports.memory.buffer, offset, len);  
  const str = new TextDecoder().decode(strBuffer);  
  window.dispatchEvent(new CustomEvent('strArrived', { detail: str }));  
};
```

7. Erzeuge mittels der obigen Erklärung folgende Ausgabe:

