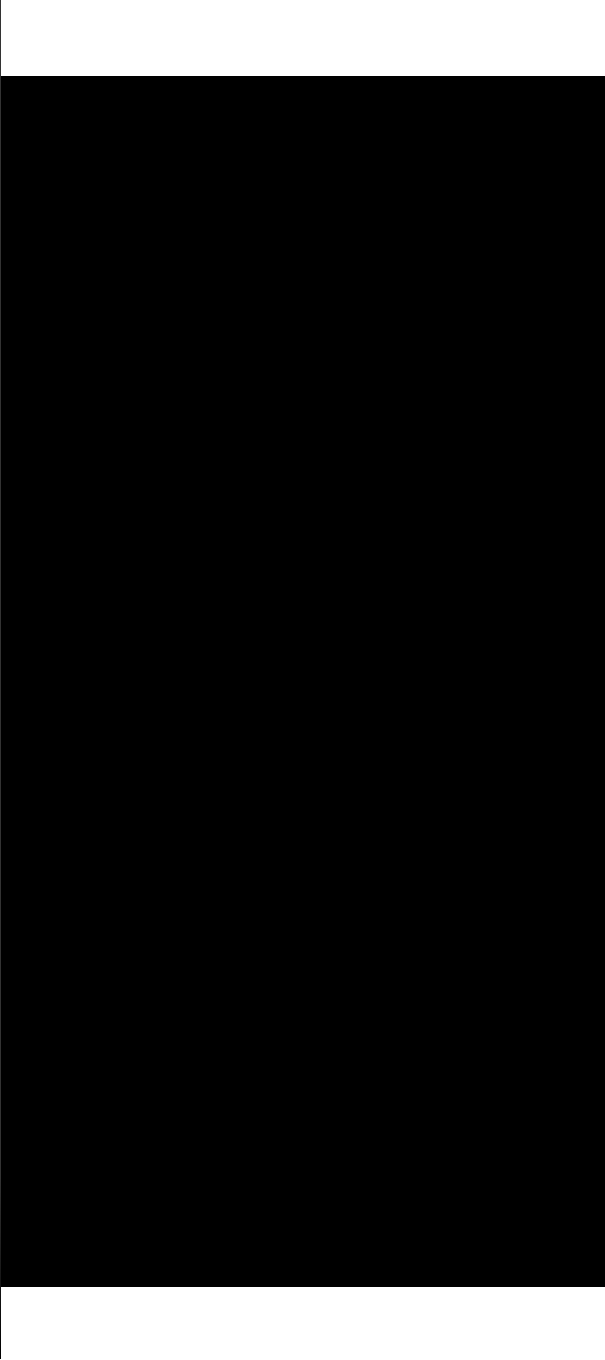
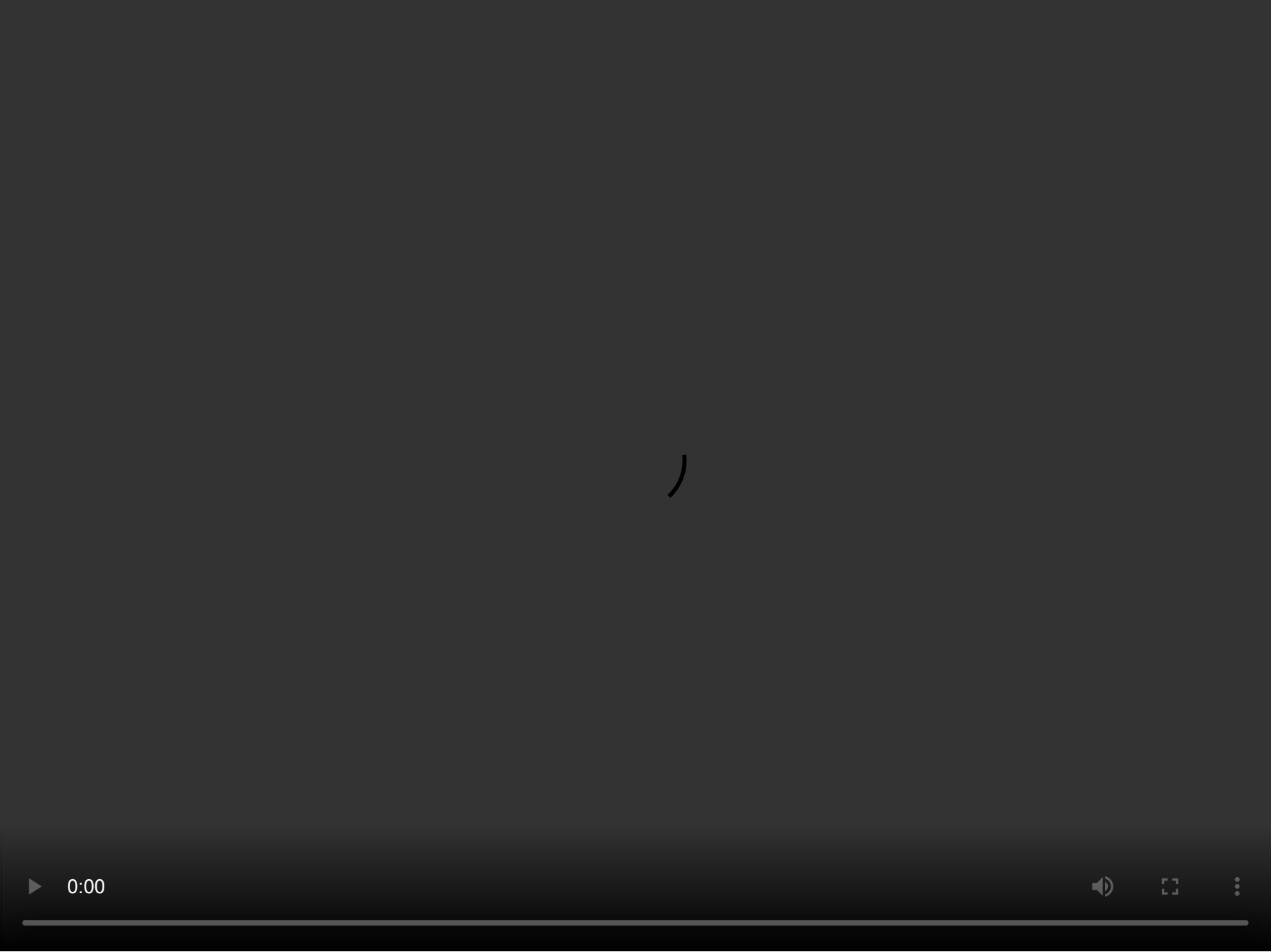


typescript

Generating Type-Safe JavaScript bindings for emscripten

Sebastian Theophil, think-cell Software, Berlin

stheophil@think-cell.com

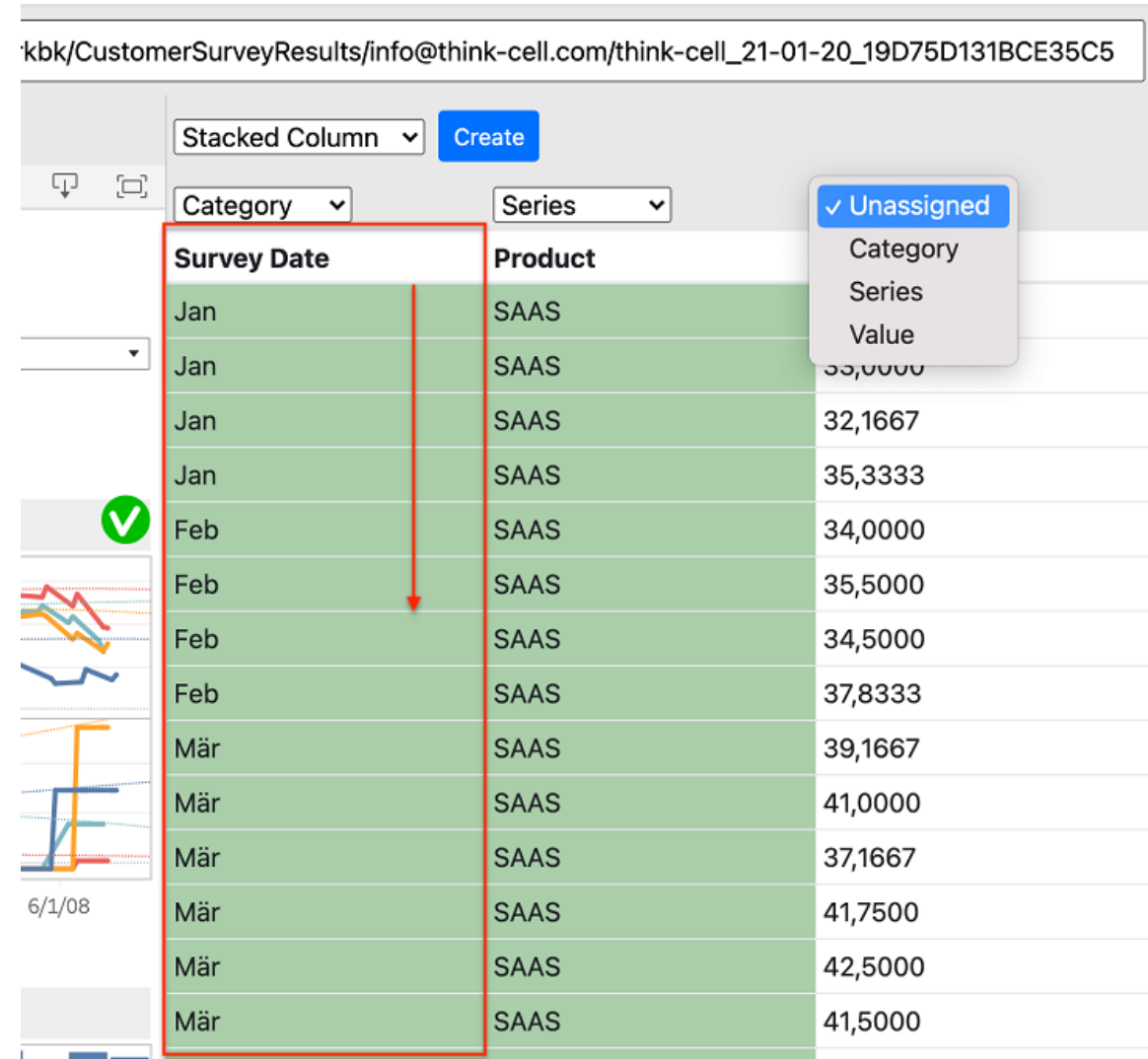


JavaScript for Beginners

One simple problem:

Transform data into tabular format

- data could be `number|string|date`
- sort, make unique, do binary search

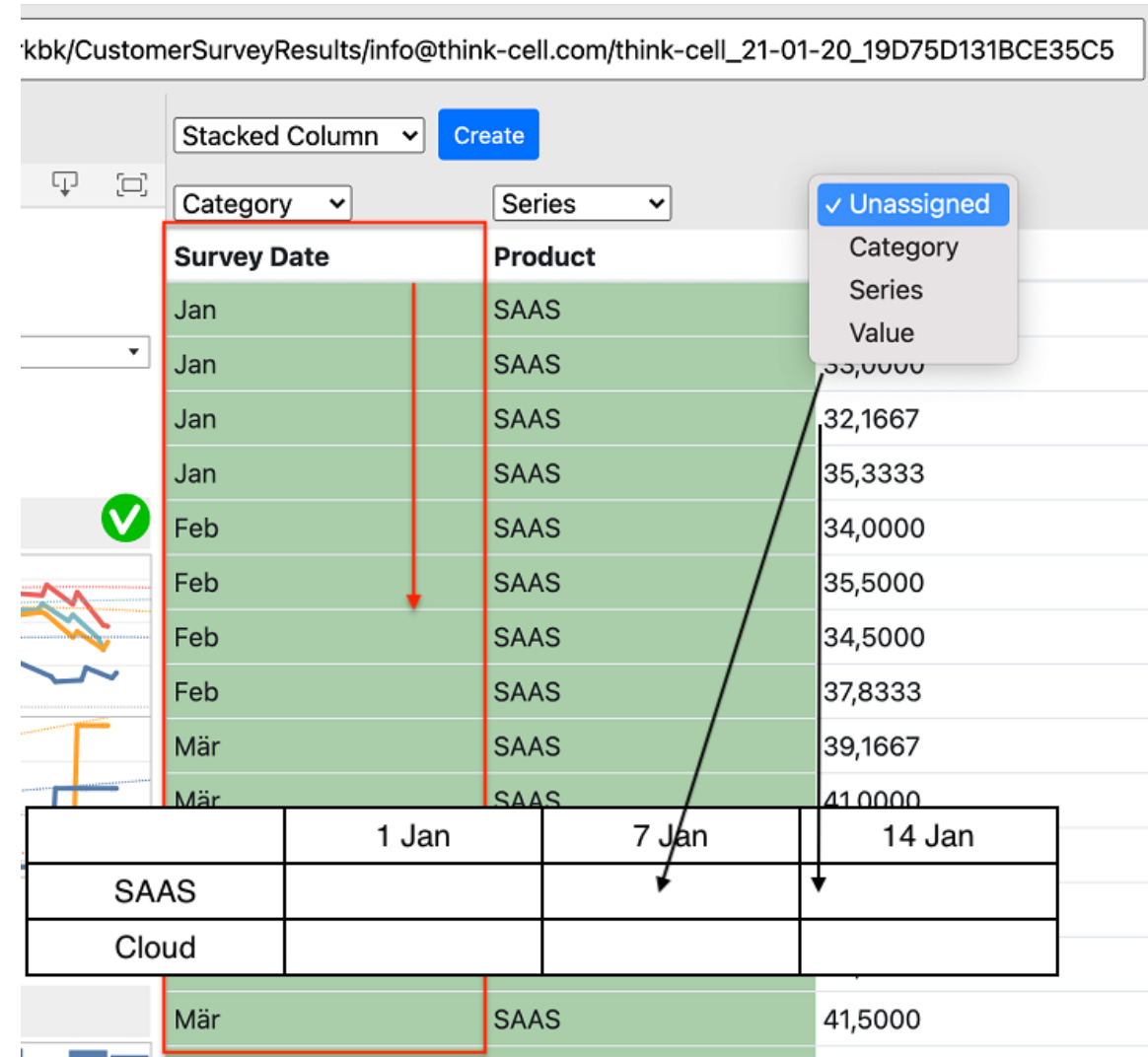


JavaScript for Beginners

One simple problem:

Transform data into tabular format

- data could be **number|string|date**
- sort, make unique, do binary search



MDN Web Docs: [Array.sort](#)


"The default sort order is ascending, built upon converting the elements into strings, then comparing their sequences of UTF-16 code units values.


The time and space complexity of the sort cannot be guaranteed as it depends on the implementation."

⊘ No unique

⊘ No binary search

JavaScript for Beginners





Search

Sign Up

Sign In

316 packages found

1

2

3

...

16

»

Sort Packages

Optimal

Popularity

Quality

Maintenance

functional-red-black-tree

A fully persistent balanced binary search tree

functional

red

black

tree

binary

search

balance


persistent

fully

dynamic

data

structure

 mikolalysenko published 1.0.1 • 7 years ago

binary-search-bounds

Better binary searching

binary

search


bounds

least

lower

greatest

upper

 mikolalysenko published 2.0.5 • 9 months ago

Meanwhile, in C++ land

Everything could be so easy:

```
using datavalue = std::variant<double, std::string, std::chrono::time_point>
std::vector<datavalue> vecdata;
// Fill vecdata

auto const rng = std::ranges::unique(std::ranges::sort(vecdata));
std::ranges::binary_search(rng, x)
```

What Do I Need?

Compile C++ for the Web

Call JavaScript from C++

Type-safe calls to JS

CppCon 2014: Alon Zakai "Emscripten and asm.js: C++'s role in the modern web"

CppCon 2014: Chad Austin "Embind and Emscripten: Blending C++11, JavaScript, and the Web Browser"

CppCon 2016: Dan Gohman "C++ on the Web: Let's have some serious fun."

CppCon 2017: Lukas Bergdoll "Web | C++"

CppCon 2018: Damien Buhl "C++ Everywhere with WebAssembly"

CppCon 2019: Ben Smith "Applied WebAssembly: Compiling and Running C++ in Your Web Browser"

CppCon 2019: Borislav Stanimirov "Embrace Modern Technology: Using HTML 5 for GUI in C++"

The shortest intro to WebAssembly

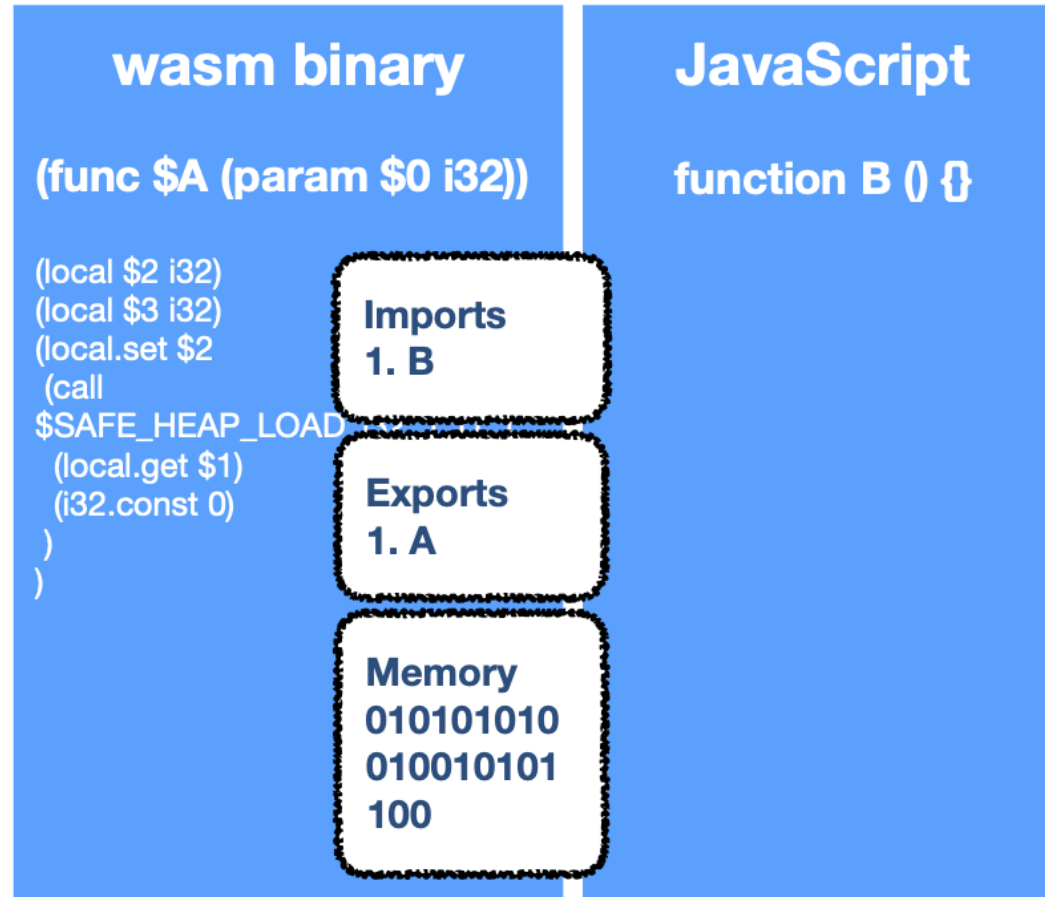
- compiled, binary format, standardised and supported by all major browser vendors
- fast and compact
- low level data types: integer and floating point numbers
- secure per-application sandbox, runs in browser VM



WEBASSEMBLY

Enter WebAssembly

WebAssembly is instantiated from and interacts with JavaScript



Enter WebAssembly

WebAssembly is instantiated from and interacts with JavaScript

```
function _abort() { abort(); }

function _handle_stack_overflow() { abort("stack overflow"); }

var imports = {
  "_handle_stack_overflow": _handle_stack_overflow,
  "abort": _abort
}

var instance = WebAssembly.instantiate(
  binary,
  {"env": imports}
).instance;

instance.exports["exported_func"]();
```

<https://hacks.mozilla.org/2018/10/calls-between-javascript-and-webassembly-are-finally-fast-🎉/>

```
(func $strcmp (param $0 i32) (param $1 i32) (result i32)
(local $2 i32)
(local $3 i32)
(local.set $2
  (call $SAFE_HEAP_LOAD_i32_1_U_1
    (local.get $1)
    (i32.const 0)
  )
)
)
(block $label$1
  (br_if $label$1
    (i32.eqz
      (local.tee $3
        (call $SAFE_HEAP_LOAD_i32_1_U_1
          (local.get $0)
          (i32.const 0)
        )
      )
    )
  )
)
)
...

```

Compiling C++ for the Web

- Toolchain based on clang/llvm with WebAssembly backend
- Simple DirectMedia Layer API (SDL) for input device access and graphics output
- Access to OpenGL API and HTML5 input events
- Virtualized file system



Easy to compile portable C or C++ to WebAssembly and run it in browser



✓ Compile C++ for the Web — **WebAssembly & emscripten**

Call JavaScript from C++

Type-safe calls to JS

How to call JavaScript?

1. Implement C functions in JS

- WebAssembly imports or exports
- limited to WebAssembly supported types, integers or floating point

2. Direct Embedding

```
int x = EM_ASM_INT({  
    console.log('I received: ' + $0);  
    return $0 + 1;  
}, 100);  
printf("%d\n", x);
```

- `int` or `double` return values

emscripten::val "transliterates JavaScript" to C++

```
using namespace emscripten;

int main() {
    val AudioContext = val::global("AudioContext");
    val context = AudioContext.new_();
    val oscillator = context.call<val>("createOscillator");
    oscillator.set("type", val("triangle"));
    oscillator["frequency"].set("value", val(261.63)); // Middle C
}
```

emscripten::val "transliterates JavaScript" to C++

```
using namespace emscripten;

int main() {
    val AudioContext = val::global("AudioContext");
    val context = AudioContext.new_();
    val oscillator = context.call<val>("createOscillator");
    oscillator.set("type", val("triangle"));
    oscillator["frequency"].set("value", val(261.63)); // Middle C
}
```

emscripten::val "transliterates JavaScript" to C++

```
using namespace emscripten;

int main() {
    val AudioContext = val::global("AudioContext");
    val context = AudioContext.new_();
    val oscillator = context.call<val>("createOscillator");
    oscillator.set("type", val("triangle"));
    oscillator["frequency"].set("value", val(261.63)); // Middle C
}
```

Pro: Convenient interaction with JS objects

Con: Combines disadvantages of both languages:

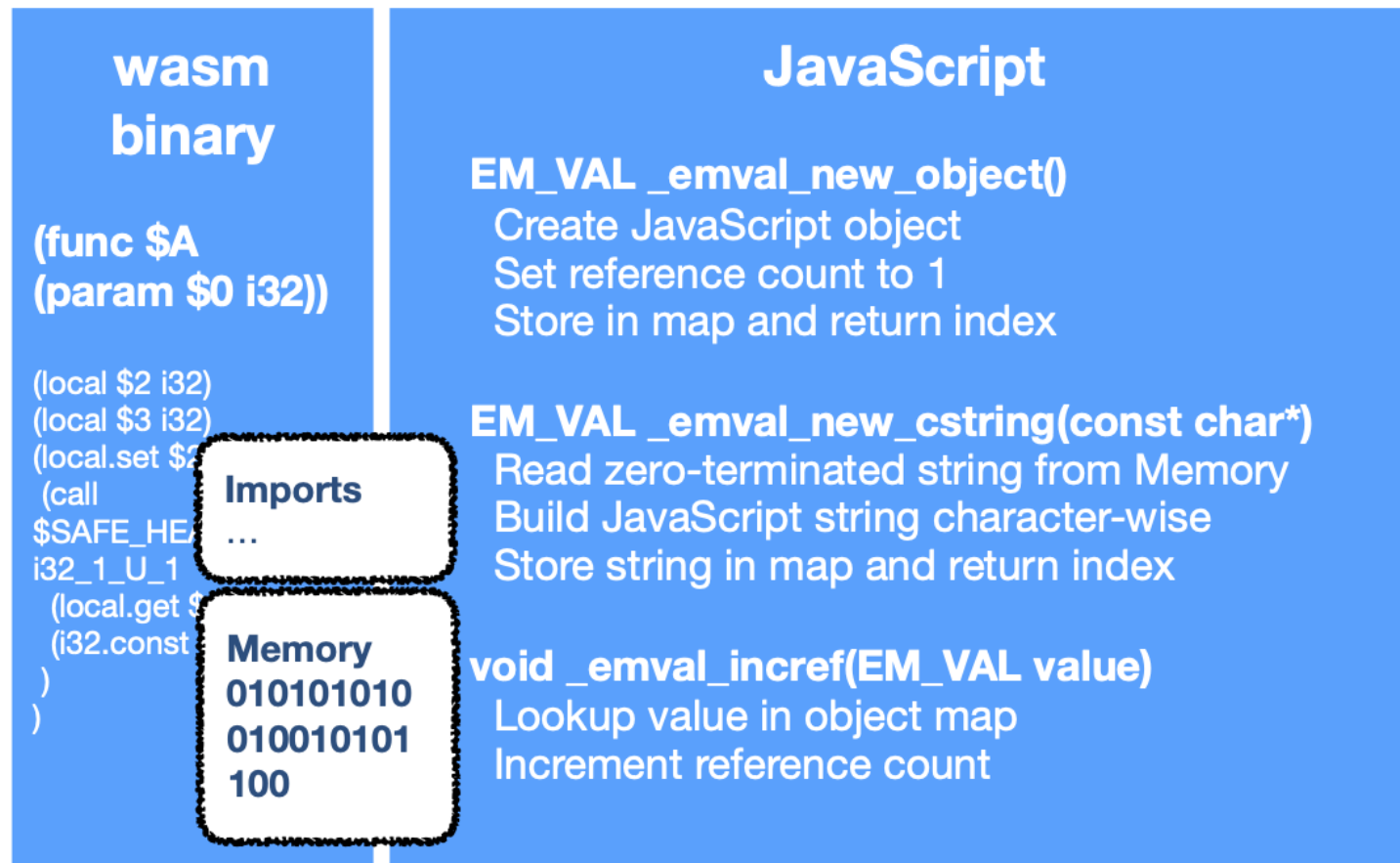
1. Compiled
2. Not type-safe

emscripten::val based on WebAssembly imports implemented in JS

```
EM_VAL _emval_new_object();
EM_VAL _emval_new_cstring(const char*);

void _emval_incref(EM_VAL value);
void _emval_decref(EM_VAL value);

void _emval_call_void_method(
    EM_METHOD_CALLER caller,
    EM_VAL handle,
    const char* methodName,
    EM_VAR_ARGS argv);
```



EM_VAL = reference to JavaScript object stored in a table, possibly with reference count

- ✓ Compile C++ for the Web — **WebAssembly & emscripten**
- ✓ Call JavaScript from C++ — **emscripten**

Type-safe calls to JS

Live Coding

1. Download TypeScript interface definition



```
npm install @types/tableau
```



2. Call typescripten compiler



3. Use C++ header definitions to call JavaScript API

Type definition libraries:

```
interface Document extends Node, NonElementParentNode, DocumentOrShadowRoot {  
  readonly URL: string;  
  
  readonly activeElement: Element | null;  
  
  readonly anchors: HTMLCollectionOf<HTMLAnchorElement>;  
  
  title: string;  
  
  createElement<K extends keyof HTMLElementTagNameMap>(  
    tagName: K, options?: ElementCreationOptions  
  ): HTMLElementTagNameMap[K];  
}
```

Type definition libraries:

```
interface Document extends Node, NonElementParentNode, DocumentOrShadowRoot {  
  readonly URL: string;  
  
  readonly activeElement: Element | null;  
  
  readonly anchors: HTMLCollectionOf<HTMLAnchorElement>;  
  
  title: string;  
  
  createElement<K extends keyof HTMLElementTagNameMap>(  
    tagName: K, options?: ElementCreationOptions  
  ): HTMLElementTagNameMap[K];  
}
```

<https://github.com/DefinitelyTyped/DefinitelyTyped>

Repository for over 7000 JavaScript libraries, e.g, AngularJS, bootstrap, tableau.com

TypeScript ships with **super convenient** parser and resolver API:

```
function transform(file: string) : void {
  let program = ts.createProgram([file]);
  const sourceFile = program.getSourceFile(file);

  ts.forEachChild(sourceFile, node => {
    if (ts.isFunctionDeclaration(node)) {
      // do something
    } else if (ts.isVariableStatement(node)) {
      // do something else
    }
  });
}
```

typescripten — <https://github.com/think-cell/typescripten>

- Compiles TypeScript interface declarations to C++ interfaces
- i.e. type-safe, idiomatic calls to JavaScript libraries via emscripten

JavaScript:

```
document.title = "Hello World from C++";
```

C++:

```
using namespace tc;  
js::document()->title(js::string("Hello World from C++!"));
```

```
namespace tc::js {
    struct object_base {
        emscripten::val m_emval;
    };

    struct Document : virtual Node, ... {
        auto URL() noexcept;
        auto activeElement() noexcept;
        auto title() noexcept;
        void title(string v) noexcept;
        // ...
    };

    inline auto Document::title() noexcept { return m_emval["title"].template as<string>(); }
    inline void Document::title(string v) noexcept { m_emval.set("title", v); }

    inline auto document() noexcept {
        return emscripten::val::global("document").template as<Document>();
    }
}
```

```
namespace tc::js {
    struct object_base {
        emscripten::val m_emval;
    };

    struct Document : virtual Node, ... {
        auto URL() noexcept;
        auto activeElement() noexcept;
        auto title() noexcept;
        void title(string v) noexcept;
        // ...
    };

    inline auto Document::title() noexcept { return m_emval["title"].template as<string>(); }
    inline void Document::title(string v) noexcept { m_emval.set("title", v); }

    inline auto document() noexcept {
        return emscripten::val::global("document").template as<Document>();
    }
}
```

Do we support all TypeScript constructs?

Do we support all TypeScript constructs?

```
interface A {  
    func(a: { length: number }) : void;  
}
```

Do we support all TypeScript constructs?

```
interface A {  
    func(a: { length: number }) : void;  
}
```

No.

Need to support common constructs in interface definition files.

Do we support all TypeScript constructs?

```
interface A {  
    func(a: { length: number }) : void;  
}
```

No.

Need to support common constructs in interface definition files.

```
interface A {  
    func(a: TypeWithLengthProperty) : void;  
}
```

Supported TypeScript constructs

- Implementation of built-in types `tc::js::any`, `tc::js::undefined`, `tc::js::null`, `tc::js::string`
- Optional members, type guards
- Support for union types `A|B|C` as `tc::js::union_t<A, B, C>`
- Mixed enums like

```
enum E {  
    a,  
    b = "that's a string",  
    c = 1.0  
}
```

- Passing function callbacks and lambdas to JavaScript as `tc::js::function<R (Args...)>`
- Generic types, e.g., `tc::js::Array<T>` or `tc::js::Record<K, V>`

Supported TypeScript constructs

- Implementation of built-in types `tc::js::any`, `tc::js::undefined`, `tc::js::null`, `tc::js::string`
- Optional members, type guards
- Support for union types `A|B|C` as `tc::js::union_t<A, B, C>`
- Mixed enums like

```
enum E {  
    a,  
    b = "that's a string",  
    c = 1.0  
}
```

- Passing function callbacks and lambdas to JavaScript as `tc::js::function<R (Args...)>`
- Generic types, e.g., `tc::js::Array<T>` or `tc::js::Record<K, V>`

Self-hosting, i.e., compiles interface definition for TypeScript API that it uses itself

typescripten itself uses generated interfaces to TypeScript API

```
function transform(file: string) : void {
  let program = ts.createProgram([file]);
  const sourceFile = program.getSourceFile(file);

  ts.forEachChild(sourceFile, node => {
    if (ts.isFunctionDeclaration(node)) {
      // do something
    } else if (ts.isVariableStatement(node)) {
      // do something else
    }
  });
}
```

typescriptn itself uses generated interfaces to TypeScript API

```
void transform(js::string const& file) noexcept {
    js::Array<js::string> files(jst::create_js_object, tc::single(file));

    auto const program = js::ts::createProgram(files, ...);
    auto const sourceFile = program->getSourceFile(file);

    js::ts::forEachChild(sourceFile,
        js::lambda(
            [] (js::ts::Node node) noexcept {
                if (js::ts::isFunctionDeclaration(node)) {
                    // do something
                } else if (js::ts::isVariableStatement(node)) {
                    // do something else
                }
            }
        )
    );
}
```

Declaration order does not matter in TypeScript

```
type FooBar = test.Foo | test.Bar;

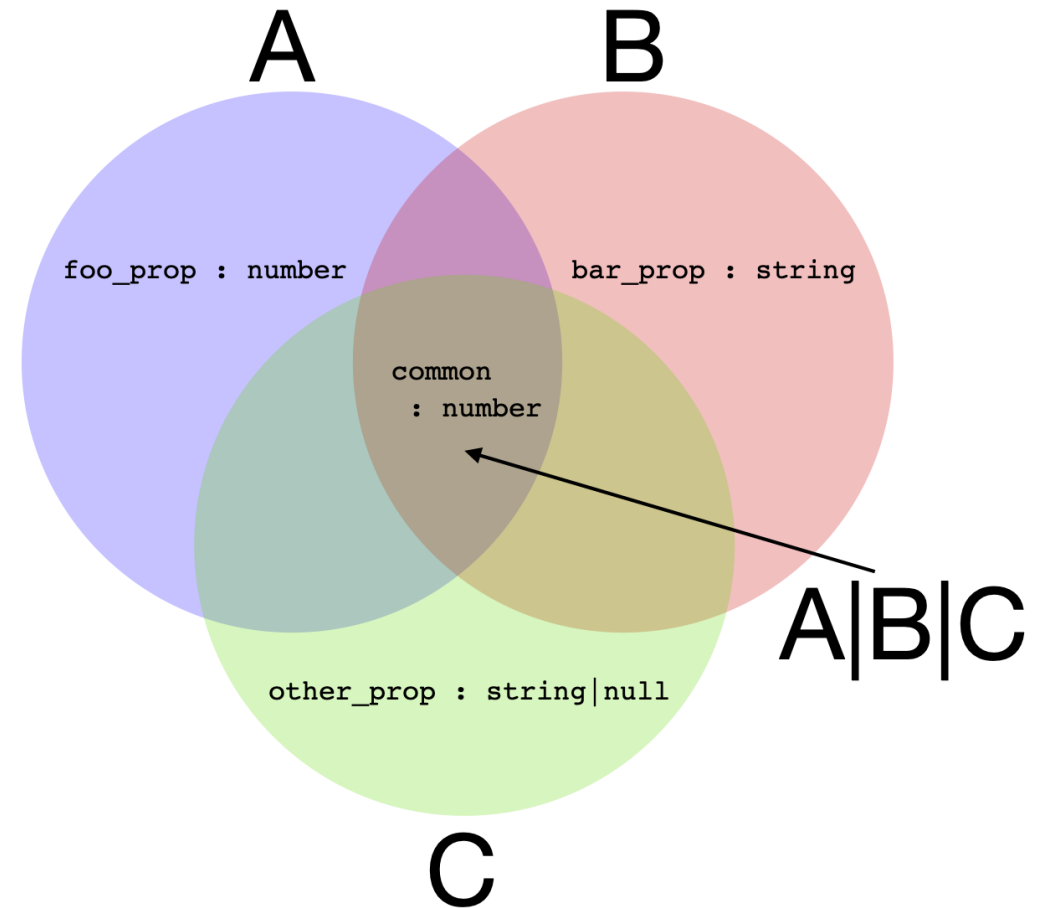
declare namespace test {
  export interface Foo {
    a: string;
  }

  export interface Bar {
    b: number;
  }
}
```


Union types are not like C++ unions

- don't have a discriminating enumeration value
- instead, intersection of properties

`A|B|C` has members that are *in the intersection* of members of A, B and C



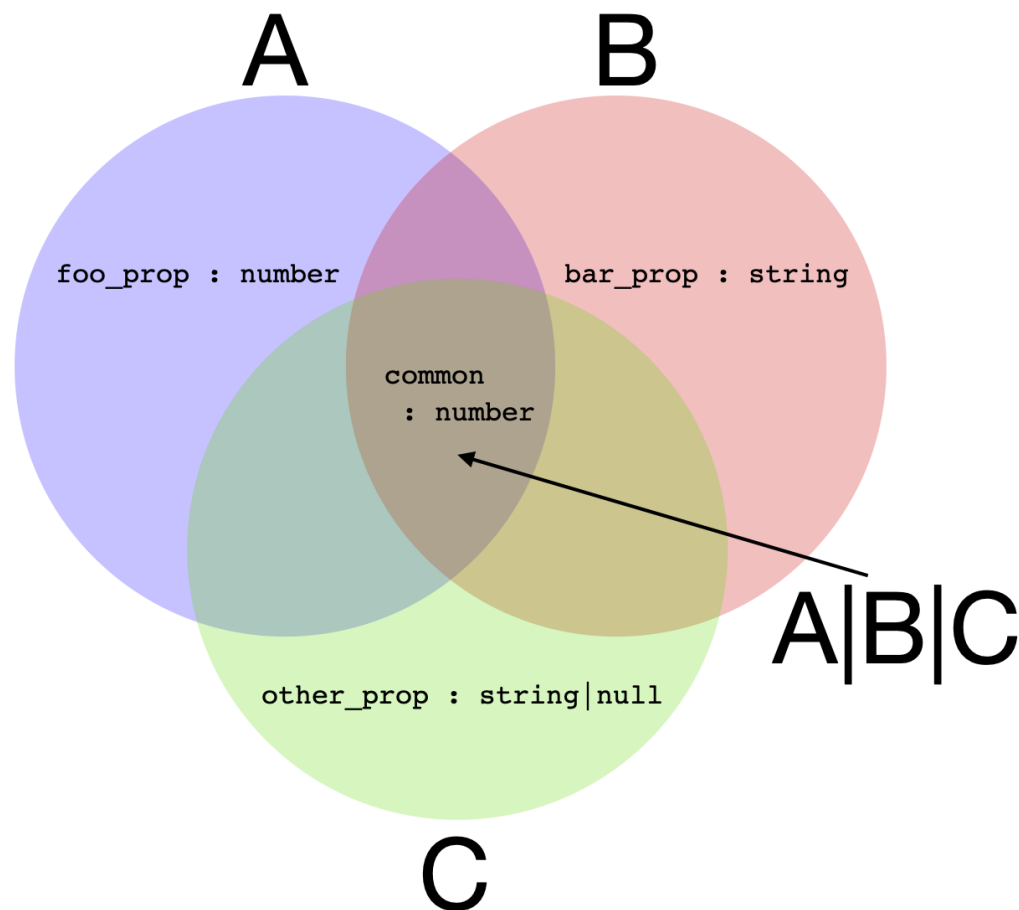
Union types are not like C++ unions

- don't have a discriminating enumeration value
- instead, intersection of properties

A|B|C has members that are *in the intersection* of members of A, B and C

A|B|C constructible from any value that has all members shared by A, B and C

```
class D {  
    common: number = 0.0;  
}  
  
let u : A|B|C = new D();
```



Union types are not like C++ unions

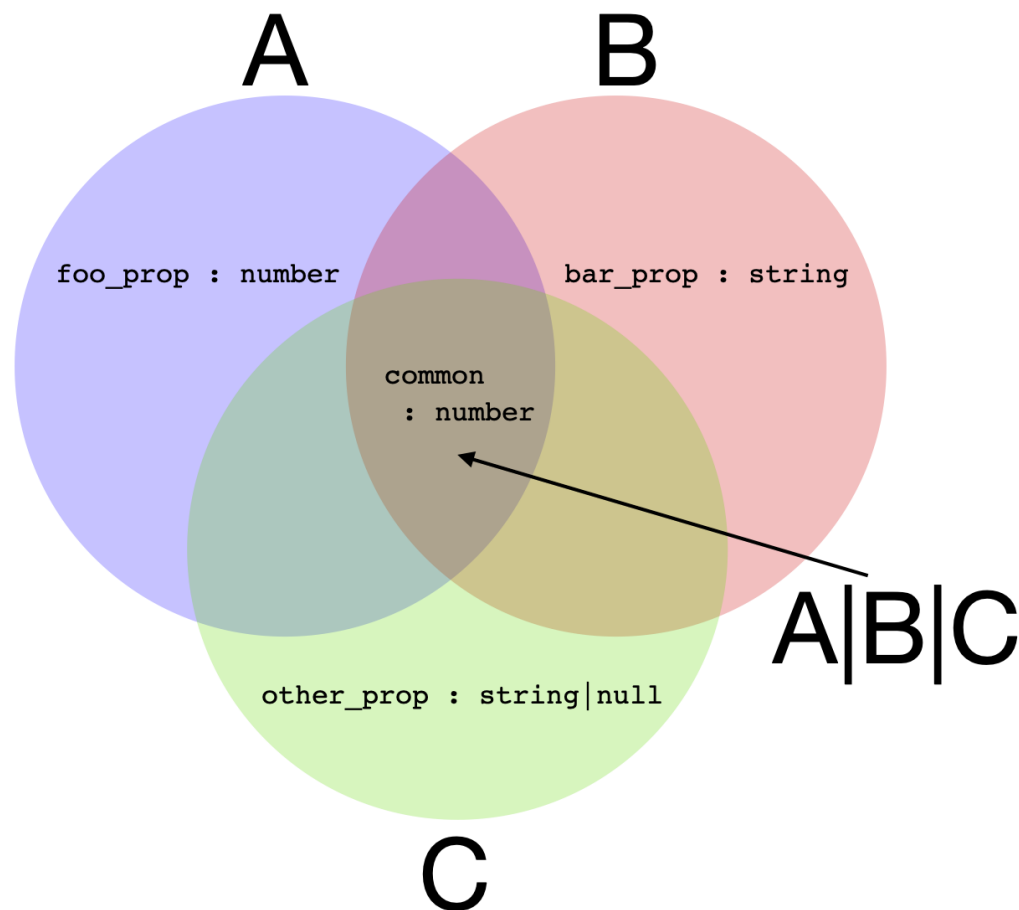
- don't have a discriminating enumeration value
- instead, intersection of properties

$A|B|C$ has members that are *in the intersection* of members of A, B and C

$A|B|C$ constructible from any value that has all members shared by A, B and C

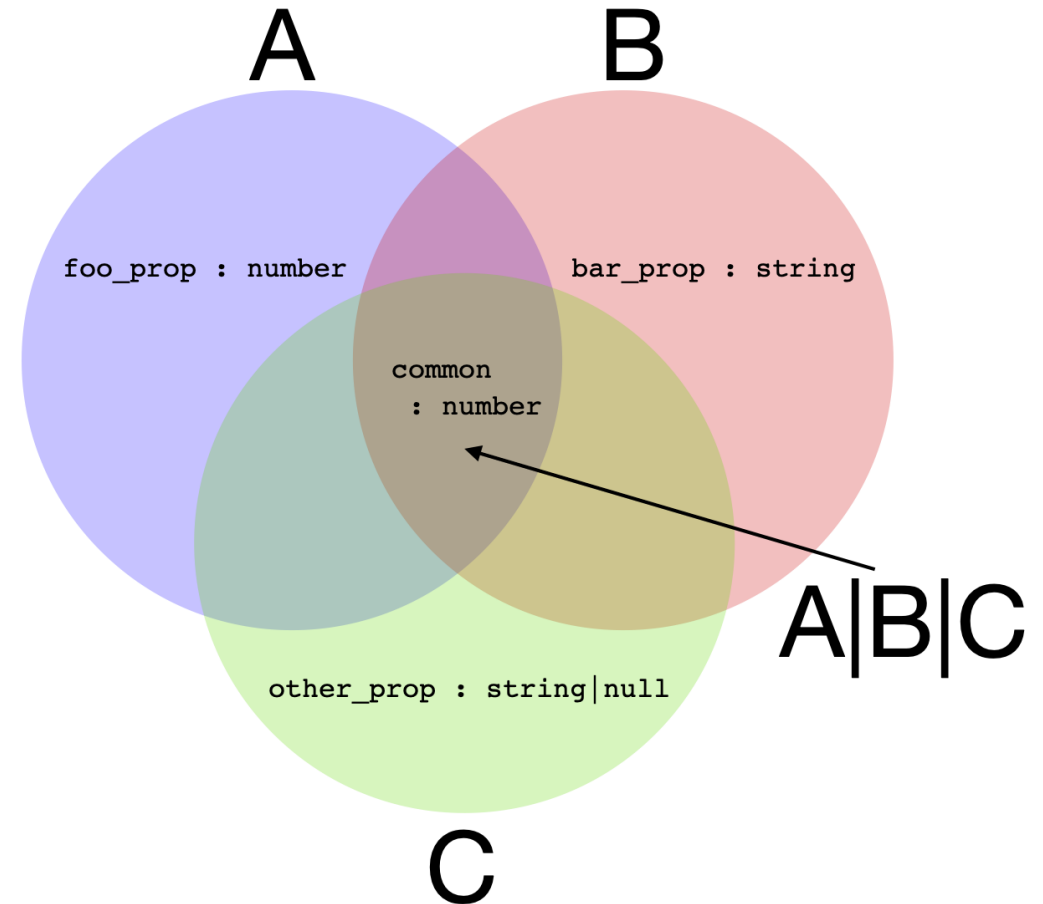
```
class D {  
    common: number = 0.0;  
}  
  
let u : A|B|C = new D();
```

A type is just a set of properties = structural typing



C++ does not support structural typing

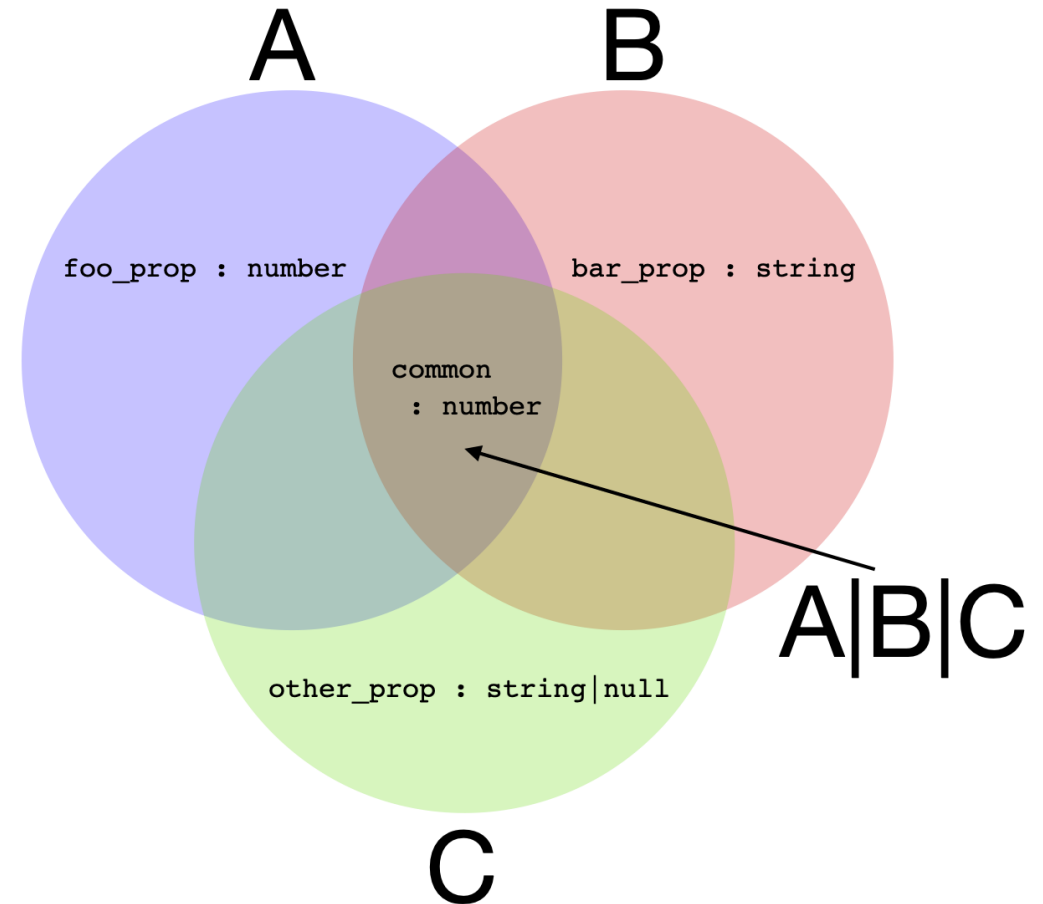
`union_t<A,B,C>` converts to *common base classes* of A, B and C



C++ does not support structural typing

`union_t<A,B,C>` converts to *common base classes* of A, B and C

`union_t<A,B,C>` converts to wider union `union_t<A,B,C,D>`

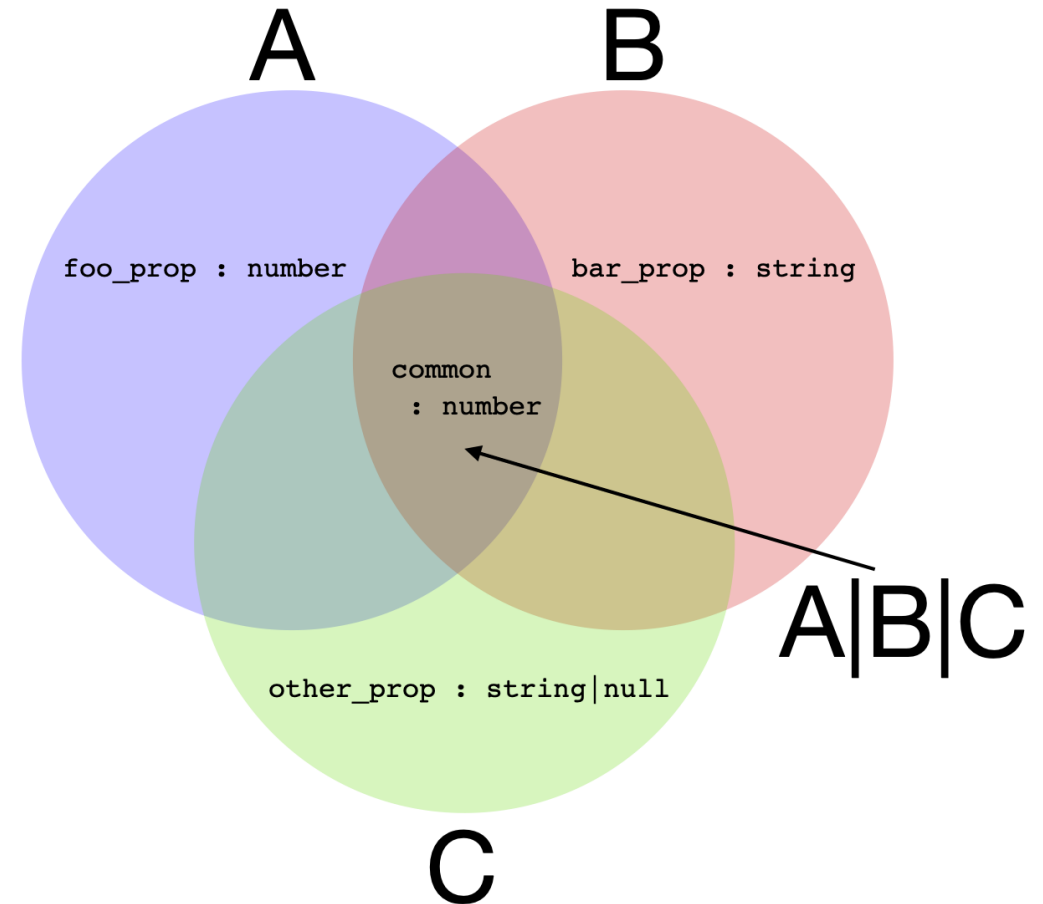


C++ does not support structural typing

`union_t<A,B,C>` converts to *common base classes* of A, B and C

`union_t<A,B,C>` converts to wider union `union_t<A,B,C,D>`

`union_t<A,B,C>` constructible from anything that converts to A, B or C



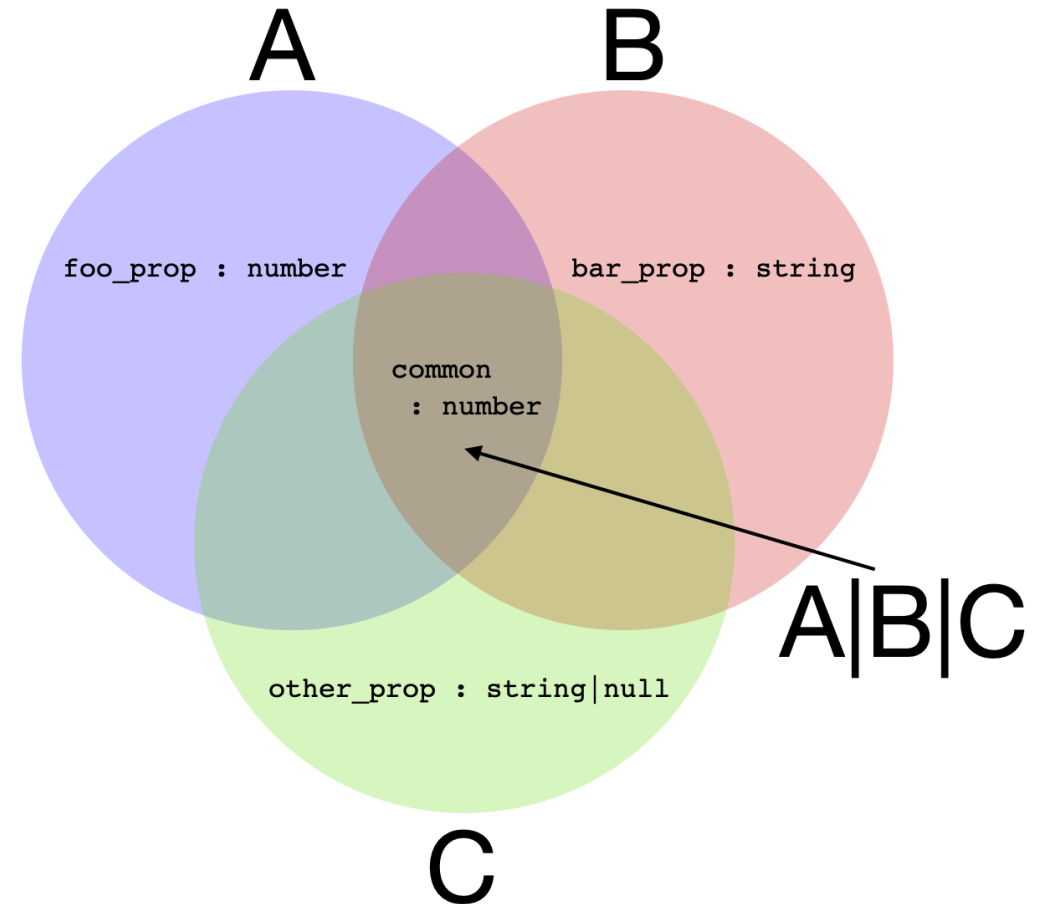
C++ does not support structural typing

`union_t<A,B,C>` converts to *common base classes* of A, B and C

`union_t<A,B,C>` converts to wider union `union_t<A,B,C,D>`

`union_t<A,B,C>` constructible from anything that converts to A, B or C

Not as limiting as it sounds



C++ does not support structural typing

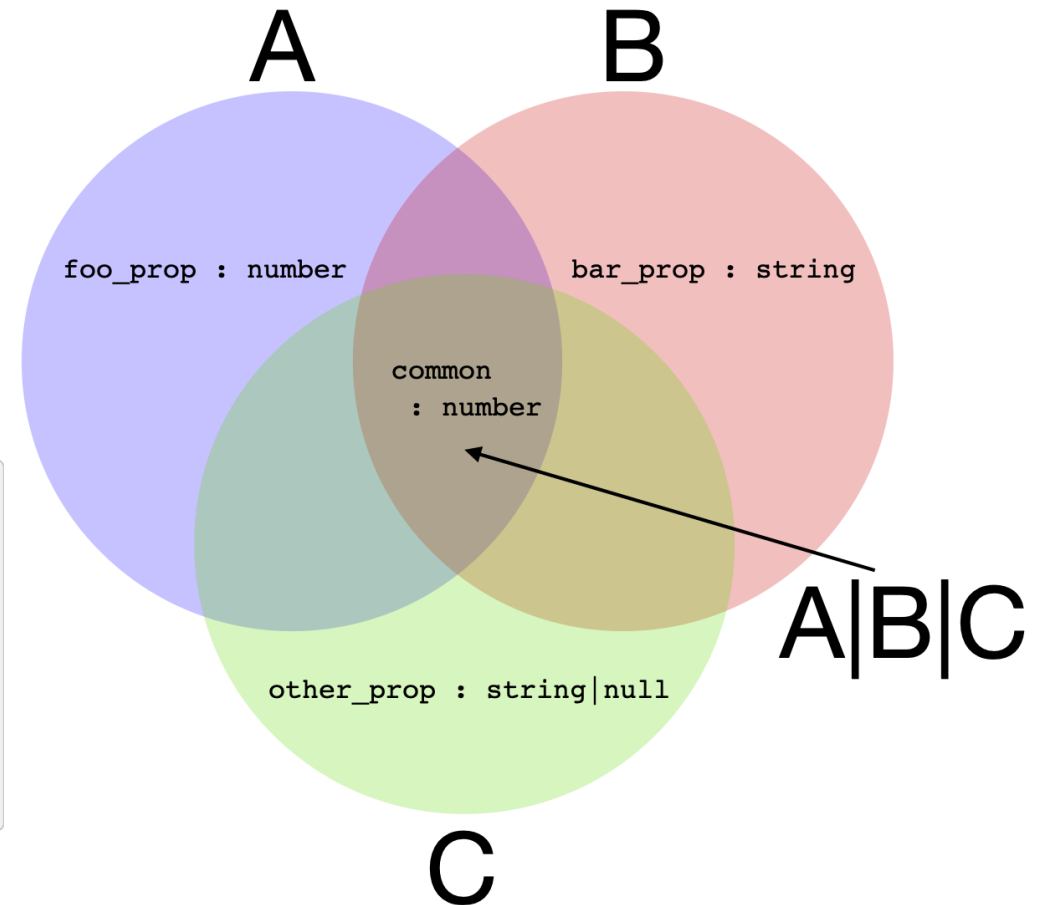
`union_t<A,B,C>` converts to *common base classes* of A, B and C

`union_t<A,B,C>` converts to wider union `union_t<A,B,C,D>`

`union_t<A,B,C>` constructible from anything that converts to A, B or C

Not as limiting as it sounds

```
interface HasCommonProp {  
  common: number;  
};  
  
interface A extends HasCommonProp {}  
interface B extends HasCommonProp {}  
interface C extends HasCommonProp {}
```



Mixed enumerations with custom marshaling

```
export enum FunnyEnum {  
    foo = "foo",  
    bar = 1.5  
}
```

Mixed enumerations with custom marshaling

```
export enum FunnyEnum {  
    foo = "foo",  
    bar = 1.5  
}
```

```
enum class FunnyEnum { foo, bar };  
  
template<> struct MarshalEnum<FunnyEnum> {  
    static inline auto const& Values() {  
        static tc::dense_map<FunnyEnum, js::any> vals{  
            {FunnyEnum::foo, js::string("foo")},  
            {FunnyEnum::bar, js::any(1.5)}  
        };  
        return vals;  
    }  
};
```

Mixed enumerations with custom marshaling

```
export enum FunnyEnum {  
    foo = "foo",  
    bar = 1.5  
}
```

```
enum class FunnyEnum { foo, bar };  
  
template<> struct MarshalEnum<FunnyEnum> {  
    static inline auto const& Values() {  
        static tc::dense_map<FunnyEnum, js::any> vals{  
            {FunnyEnum::foo, js::string("foo")},  
            {FunnyEnum::bar, js::any(1.5)}  
        };  
        return vals;  
    }  
};
```

Code Example #2

Reference-counted function objects are complicated

```
class SomeButton {  
  constructor() {  
    const button = document.createElement(...);  
    button.addEventListener("click", () => this.OnClick());  
  }  
  
  function OnClick(ev: MouseEvent) : void {  
    /* do something */  
    /* but in which states will this be called? */  
  }  
}
```

- No deterministic destruction
- On ownership of reference-counted objects
- makes thinking about states complicated

Ugly syntax but simple state machine

```
struct SomeButton {  
    SomeButton() {  
        const button = js::document()->createElement(...);  
        button->addEventListener("click", OnClick);  
    }  
  
    ~SomeButton() {  
        button->remove();  
        // Our callback will also be destroyed! 🎉  
    }  
  
    TC_JS_MEMBER_FUNCTION(S, OnClick, void, (js::MouseEvent ev)) {  
        // do something  
    }  
};
```

```
// 1. Create RAII wrapper OnClick
static emscripten::val OnClickWrapper(void* pvThis, emscripten::val const& emvalThis,
    emscripten::val const& emvalArgs) noexcept;

jst::function<void (js::MouseEvent)> OnClick{&OnClickWrapper, this};
```

```
// 1. Create RAII wrapper OnClick
static emscripten::val OnClickWrapper(void* pvThis, emscripten::val const& emvalThis,
    emscripten::val const& emvalArgs) noexcept;

jst::function<void (js::MouseEvent)> OnClick{&OnClickWrapper, this};
```

```
// 2. jst::function ctor calls to JS and creates JS function object
Module.CreateJsFunction = function(iFuncPtr, iThisPtr) {
    const fnWrapper = function() {
        if(iFuncPtr !== null) {
            return Module.tc_js_CallCpp(iFuncPtr, iThisPtr, this, arguments);
        }
    };
    fnWrapper.detach = function() {
        iFuncPtr = null;
    }
    return fnWrapper;
}
// 3. JS function object held as emscripten::val
```



```
// 1. Create RAII wrapper OnClick
static emscripten::val OnClickWrapper(void* pvThis, emscripten::val const& emvalThis,
    emscripten::val const& emvalArgs) noexcept;

jst::function<void (js::MouseEvent)> OnClick{&OnClickWrapper, this};
```

```
// 2. jst::function ctor calls to JS and creates JS function object
Module.CreateJsFunction = function(iFuncPtr, iThisPtr) {
    const fnWrapper = function() {
        if(iFuncPtr != null) {
            return Module.tc_js_CallCpp(iFuncPtr, iThisPtr, this, arguments);
        }
    };
    fnWrapper.detach = function() {
        iFuncPtr = null;
    }
    return fnWrapper;
}
// 3. JS function object held as emscripten::val
```

```
// 1. Create RAII wrapper OnClick
static emscripten::val OnClickWrapper(void* pvThis, emscripten::val const& emvalThis,
    emscripten::val const& emvalArgs) noexcept;

jst::function<void (js::MouseEvent)> OnClick{&OnClickWrapper, this};
```

```
// 2. jst::function ctor calls to JS and creates JS function object
Module.CreateJsFunction = function(iFuncPtr, iThisPtr) {
    const fnWrapper = function() {
        if(iFuncPtr != null) {
            return Module.tc_js_CallCpp(iFuncPtr, iThisPtr, this, arguments);
        }
    };
    fnWrapper.detach = function() {
        iFuncPtr = null;
    }
    return fnWrapper;
}
// 3. JS function object held as emscripten::val
```

```
// 1. Create RAII wrapper OnClick
static emscripten::val OnClickWrapper(void* pvThis, emscripten::val const& emvalThis,
    emscripten::val const& emvalArgs) noexcept;

jst::function<void (js::MouseEvent)> OnClick{&OnClickWrapper, this};
```

```
// 2. jst::function ctor calls to JS and creates JS function object
Module.CreateJsFunction = function(iFuncPtr, iThisPtr) {
    const fnWrapper = function() {
        if(iFuncPtr != null) {
            return Module.tc_js_CallCpp(iFuncPtr, iThisPtr, this, arguments);
        }
    };
    fnWrapper.detach = function() {
        iFuncPtr = null;
    }
    return fnWrapper;
}
// 3. JS function object held as emscripten::val
```

```
// 4. When called, JS function object passes function pointer back to generic C++ function
emscripten::val Call(PointerNumber iFuncPtr, PointerNumber iArgPtr,
    emscripten::val emvalThis, emscripten::val emvalArgs) noexcept {
    // 5. Casts function pointer to correct signature and calls it
}
```

```
// 4. When called, JS function object passes function pointer back to generic C++ function
emscripten::val Call(PointerNumber iFuncPtr, PointerNumber iArgPtr,
    emscripten::val emvalThis, emscripten::val emvalArgs) noexcept {
    // 5. Casts function pointer to correct signature and calls it
}
```

```
static emscripten::val OnClickWrapper(void* pvThis, emscripten::val const& emvalThis,
    emscripten::val const& emvalArgs) noexcept
{
    // 6. Cast this pointer, unpack arguments from emvalArgs and call OnClickImpl
}
```

```
// 4. When called, JS function object passes function pointer back to generic C++ function
emscripten::val Call(PointerNumber iFuncPtr, PointerNumber iArgPtr,
    emscripten::val emvalThis, emscripten::val emvalArgs) noexcept {
    // 5. Casts function pointer to correct signature and calls it
}
```

```
static emscripten::val OnClickWrapper(void* pvThis, emscripten::val const& emvalThis,
    emscripten::val const& emvalArgs) noexcept
{
    // 6. Cast this pointer, unpack arguments from emvalArgs and call OnClickImpl
}
```

```
void OnClickImpl(js::MouseEvent ev) noexcept {
    /* ... user code */
}
```

TypeScript supports generic classes

```
js::HTMLCollectionOf<js::Element> htmlcollection =  
    js::document()->body()->getElementsByTagName(js::string("div"));
```

TypeScript supports generic classes

```
js::HTMLCollectionOf<js::Element> htmlcollection =  
    js::document()->body()->getElementsByTagName(js::string("div"));
```

Generic classes are translated to C++ templates

```
interface Array<T> {} translates to template<typename T> struct Array {}
```


TypeScript supports generic classes

```
js::HTMLCollectionOf<js::Element> htmlcollection =  
    js::document()->body()->getElementsByTagName(js::string("div"));
```

Generic classes are translated to C++ templates

`interface Array<T> {}` translates to `template<typename T> struct Array {}`

Generic classes can have constraints

```
enum Enum {}  
interface A<T extends Enum> {}
```

TypeScript supports generic classes

```
js::HTMLCollectionOf<js::Element> htmlcollection =  
    js::document()->body()->getElementsByTagName(js::string("div"));
```

Generic classes are translated to C++ templates

`interface Array<T> {}` translates to `template<typename T> struct Array {}`

Generic classes can have constraints

```
enum Enum {}  
interface A<T extends Enum> {}
```

Expressible as non-type template parameter

```
template<Enum E>  
struct A {};
```

Generic classes support many kinds of constraints

```
class Node {}  
interface A<T> extends Node> {}
```

Generic classes support many kinds of constraints

```
class Node {}  
interface A<T extends Node> {}
```

might be expressed as

```
<typename T, std::enable_if_t<std::is_base_of<tc::js::ts::Node, T>::value>* = nullptr>  
struct A {};
```

Again, the semantics are not identical.

Live Coding #3

- ✓ Compile C++ for the Web — **WebAssembly & emscripten**
- ✓ Call JavaScript from C++ — **emscripten**
- ✓ Type-safe calls to JS — **typescripten**

typescripten will be superseded by *WebAssembly Interface types*

Still in proposal phase <https://github.com/WebAssembly/interface-types>

Longer Introduction: <https://hacks.mozilla.org/2019/08/webassembly-interface-types/>

As in ISO C++, maybe good idea to experiment with implementation

Performance Test

1.000.000 function calls WebAssembly to JavaScript

JS function increments a number

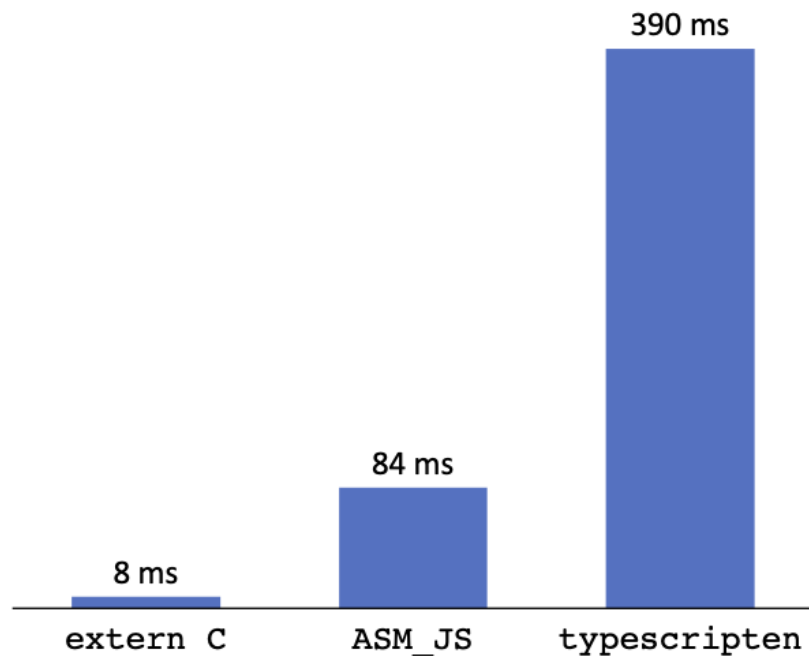
- `extern "C" function` from WebAssembly to JavaScript
- `EM_ASM_DOUBLE` embedded JS code
- `typescriptn` call via `emscripten::val`

```
inline auto _impl_js_j_qMyLib_q::_tcjs_definitions::next() noexcept {  
    return emscripten::val::global("MyLib")["next"]().template as<double>();  
}
```


Performance Test

1.000.000 function calls WebAssembly to JavaScript

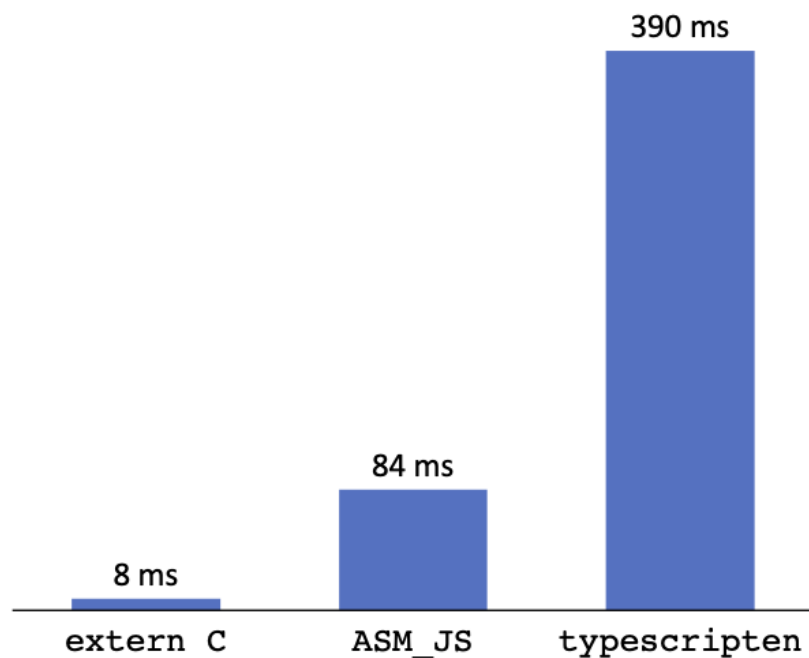
JS function increments a number



Performance Test

1.000.000 function calls WebAssembly to JavaScript

JS function increments a number



Cost of converting C-strings to JavaScript strings

Next Challenges:

- Generic constraints

```
interface HTMLCollectionOf<T extends Element> extends HTMLCollectionBase {  
    item(index: number): T | null;  
}
```

Next Challenges:

- Generic constraints

```
interface HTMLCollectionOf<T extends Element> extends HTMLCollectionBase {  
  item(index: number): T | null;  
}
```

- Indexed access types

```
interface DocumentEventMap {  
  "click": MouseEvent;  
  "keydown": KeyboardEvent;  
}  
addEventListener<K extends keyof DocumentEventMap>(  
  type: K, listener: (this: Document, ev: DocumentEventMap[K]) => any, ...  
) : void;
```

Check it out at <https://github.com/think-cell/typesripten>

Contributors are very welcome

Thank you!

Now to your questions!

Sebastian Theophil, think-cell Software, Berlin

stheophil@think-cell.com

keyof operator returns the names of class properties

```
interface K {  
    foo: string;  
    bar: string;  
}  
interface A<T extends keyof K> {} // T can be "foo" or "bar"
```

maybe best expressed as

```
enum class KeyOfK {  
    foo, bar  
};  
template<KeyOfK e> struct A {}
```

