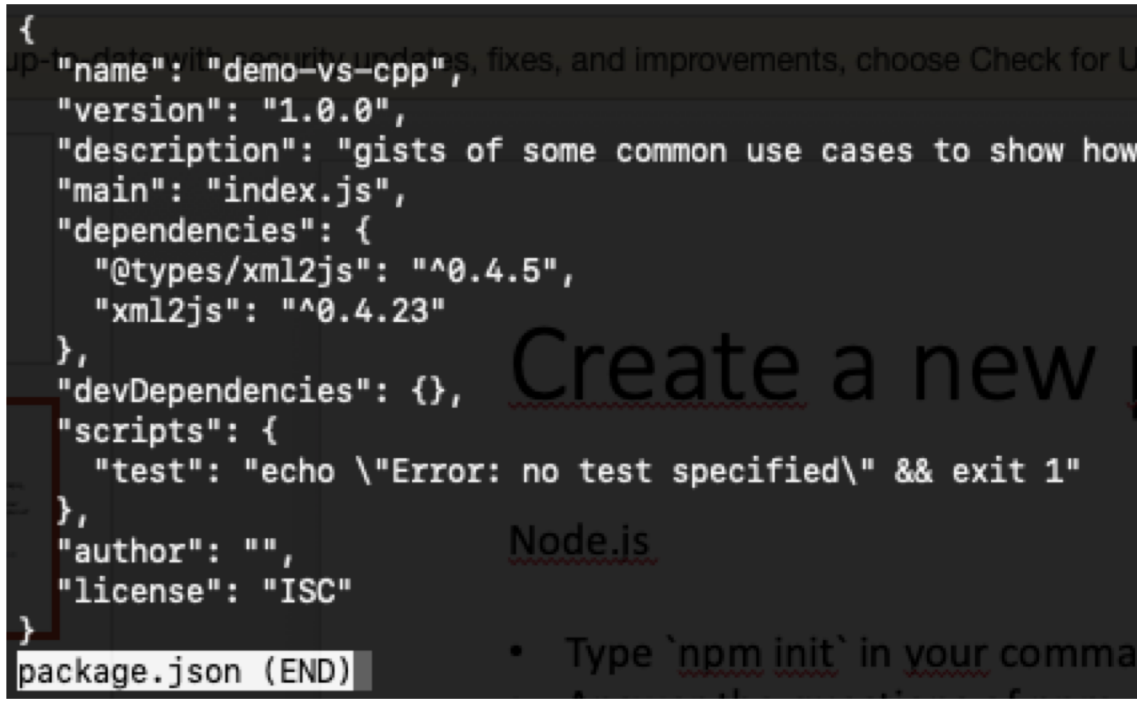
Node.JS && C++

&& TypeScript

Create a new project

Node.js

- Type `npm init` in your command line
- Answer the questions of npm
- // You're done



C++

- Create a new project configuration (vcxproj, Makefile, CMakeLists, …etc)
  - Either fill an assistant GUI or fill your project configuration or write it yourself and enjoy! :p
- Example of fresh configuration:

```
1  cmake_minimum_required(VERSION 3.9)
2  project(HelloWorld)
3
4  set(CMAKE_CXX_STANDARD 11)
5
6  add_executable(HelloWorld
7          main.cpp)
8
```

Your entry point

Node.js

```
/* Nothing to write, your code will be executed
from the first line */
console.log(‘Hello, World!’)
```

C++

```
int main(int argc, char **argv, char **env) {
  // Write your start code here
  printf(‘’Hello, World!’’);
  return 0; // End your program
}
```

Add a library

Node.js

- Type `npm i [name_of_your_library]` in your command line
- Use it! =)

C++

- No native packager, you have to use one of these solution:
  - Copy/Paste code in your project and add it to your compiler input
  - Compile the library and add it to your linker input
  - Use alternative packager / dependency resolver (CMake, vcpkg, Conan, …etc)

Compile your source

Node.js

- No compilation

TypeScript

- `tsc [file.ts]` => Will produce a ‘’file.js’’ script

C/C++

- `gcc [file.c]` or `g++ [file.cpp]`
- => Will produce a ’’a.out’’ executable

Build your project

Node.js

- `npm build`

C/C++

- `make` or `cmake . && make`, …etc depending on the build system

Number of build environments

Node.js

- 1
- No need to rebuild for each platforms
- External native dependencies are not tied to scripts, so cross-build is easy

C/C++

- At least the result of this formula: A * O
- A = Number of processor architectures
- O = Number of Operating Systems
- Cross-Compilation is not easy
  - Clang + LLVM can produce binaries for different processors but are still tied to the current OS
  - GCC need to be rebuilt for each cross-compilation target if the host can support it

Base project build size (Hello, World!)

Node.js

- With global interpreter -> Size of the script (30B) ☺
- With embedded interpreter
  - No optimization: ~38MB ☹☹☹
  - Stripped interpreter binary: ~26MB ☹☹
  - Compressed package: ~10MB ☹

C/C++

- Base g++ invocation:
  - Base size: 8,2KB ☺
  - Stripped size: 8,2KB ☺
  - Compressed: error (too small)
- G++ static build without libc (using inline assembly instead): 4KB

Memory management

Node.js

```ts
nodejs > TS dynamic-allocation.ts > ...
interface Toto {
    toto: string;
}

function overwrite(a : Toto) : void {
    a.toto = "I overwrote it!";
    // Previous value in a has been freed
}

function init_my_toto(a : Toto) : void {
    a.toto = "Hello, World!".toLowerCase();
}
var a : Toto = {toto: null};
console.log(a);
init_my_toto(a);
console.log(a);
overwrite(a);
console.log(a);
```

C++

```cpp
nodejs > dynamic-allocation.cpp
#include <string.h>
#include <string>
#include <iostream>
#if __cplusplus >= 201400L
#include <memory>
using Toto = std::unique_ptr<char *>;
#else
typedef char * Toto;
#endif

void overwrite(Toto & a) {
#if __cplusplus >= 201400L
    char * b = *a;
    a = std::make_unique<char*>(strdup("I overwrote it!"));
    // delete b; // OK! already freed
#else
    // delete a; // Oops! memory leak if forgot
    a = strdup("I overwrote it!");
#endif
}

// absent from standard library standard :(
static char * str2lower(const char * s) {
    char * ret = strdup(s);
    for (int i = 0; i < strlen(s); i++) {
        if (ret[i] >= 'A' && ret[i] <= 'Z') ret[i] += 32;
    }
    return ret;
}

void init_my_toto(Toto & a) {
#if __cplusplus >= 201400L
    a = std::make_unique<char*>(str2lower("Hello, World!"));
#else
    a = str2lower("Hello, World!");
#endif
}

int main(void) {
#if __cplusplus >= 201400L
    Toto a = std::make_unique<char *>(strdup(""));
    std::cout << *a << std::endl;
    init_my_toto(a);
    std::cout << *a << std::endl;
    overwrite(a);
    std::cout << *a << std::endl;
#else
    Toto a = "";
    std::cout << a << std::endl;
    init_my_toto(a);
    std::cout << a << std::endl;
    overwrite(a);
    std::cout << a << std::endl;
#endif
    return 0;
}
```

Callbacks

Node.js

nodejs > TS callback.ts > ...

```ts
function call(toto : Function) {
    toto();
}
call(() => { console.log("Hello, World!"); })
```

C++

nodejs > callback.cpp > main(void)

```cpp
#include <iostream>
#if __cplusplus >= 201103L
#include <functional>
void call(std::function<void ()> toto) {
    toto();
}
#else
void call(void (*toto)()) {
    toto();
}
void titi() { std::cout << "Hello, C" << std::endl; }
#endif
int main(void) {
#if __cplusplus >= 201103L
    call([]() { std::cout << "Hello, C++11!" << std::endl; });
#else
    call(&titi);
#endif
}
```

String formatting

Node.js

nodejs > TS string-formatting.ts > b

```ts
const a = 42;
const b = 3.14;
console.log(`Hello, World n${a},
you're ${b} magic number`);
```

C++

nodejs > string-formatting.cpp > toto()

```cpp
#include <sstream>

std::string toto() {
    std::ostringstream ss;
    ss << "Hello, World n" << 42 << ','
    << std::endl << "you're the " << 3.14
    << " magic number" << std::endl;
    // C++ way until std::fmt in C++20
    char buffer[512];
    sscanf(buffer, "Hello, World n%d,\n"
    "you're the %f magic number\n",
    42, 3.14); // C way
    return ss.str();
}
```

Real World example: JSON parsing

Node.js

nodejs > TS parse-json.ts

```
1   console.log(JSON.parse(`
2   {
3       "tata":42,
4       "titi": [
5           {
6               "foo": "Hello, World!",
7               "bar":3.14
8           },
9           "I am a string!",
10          null
11      ]
12  }`));
```

C++

- The good: C++11 libraries using STL and initializer lists (for ex: nlohmann/json for example)
- The ok: C++98 libraries where every fields are accessed manually (for ex: ArduinoJson
- The bad: code generators required based libraries because of their conception (for ex: gsoap)
- <= You can do the same even in C++98 with good libraries

Real World example: JSON serialization

Node.js

```
nodejs > TS dump-json.ts > ...
1   const toto = {
2       tata: 42,
3       titi: [
4           {
5               foo: 'Hello, World!',
6               bar: 3.14
7           },
8           "I am a string!",
9           null
10      ]
11  };
12
13  console.log(JSON.stringify(toto));
```

C++

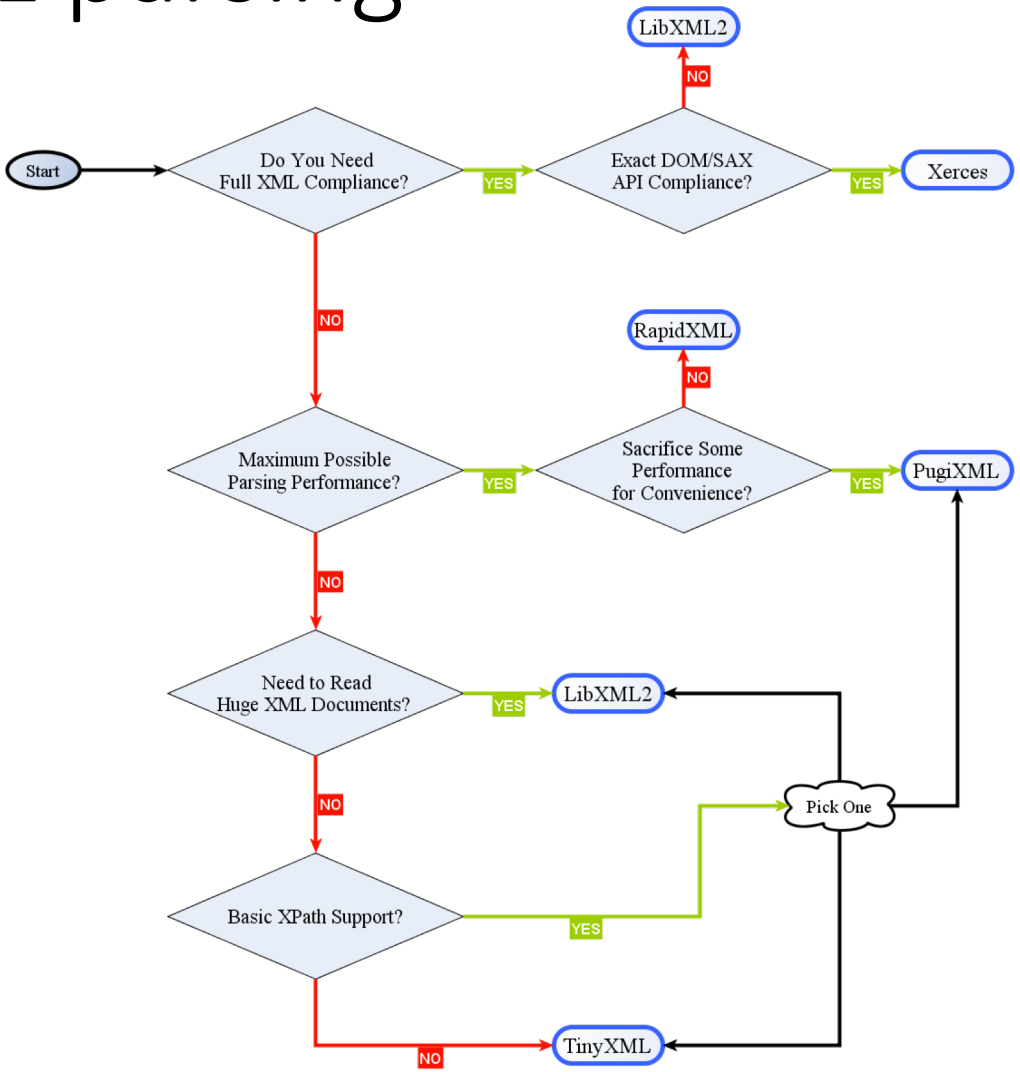
- The good: C++11 libraries using STL and initializer lists (for ex: nlohmann/json for example)
- The ok: C++98 libraries where every fields are accessed manually (for ex: ArduinoJson
- The bad: code generators required based libraries because of their conception (for ex: gsoap)
- <= You can do the same in C++11 if using only STL containers and types with good libraries, otherwise you'll have to implement your own serialization like in C++98

Real World example: XML parsing

Node.js

```
nodejs > TS parse-xml.ts > ...
1   import { parseString } from 'xml2js';
2
3   parseString(`<toto>
4   <tata>42</tata>
5   <titi>[
6       <ano>
7           <foo>Hello, World!</foo>
8           <bar>3.14</bar>
9       </ano>,
10      I am a string,
11      null
12  ]</titi>
13  </toto>`, (_, result) => console.log(result));
```

C++ (source: StackOverflow)



Real World example: XML serialization

Node.js

C++: Ouch :’(

Real World example: SQL requests

Node.js

nodejs > TS sql-demo.ts > then() callback

```typescript
import { Entity, PrimaryGeneratedColumn, Column,
createConnection, Not, Equal } from 'typeorm';

export interface IToto {
    id: number;
    tata: string;
}

@Entity()
export class Toto implements IToto {
    @PrimaryGeneratedColumn()
    id: number;
    @Column()
    tata: string;
};

createConnection({ type: 'sqlite', database: 'toto.db', entities: [ Toto ]
}).then(async (db) => {
    var toto = new Toto();
    var repo = db.getRepository(Toto);
    toto.tata = 'Hello, World!';
    await repo.save(toto);
    var tata : IToto[] = await repo.find({
        tata: Not(Equal('Hello, World!'))
    });
    console.log(tata);
});
```

C++:

- Code generators
- Raw SQL requests throug DB connectors
- Sqlpp11 and sqlpp17

Fork and IPC

Node.js

nodejs > TS ipc.ts > ...

```ts
import { fork } from 'child_process';

var worker = fork('./my_worker.js', ['1', '1', '1', 'Houston']);
worker.on('message', (message) => {
    // do something on parsed message
});
worker.send({ toto: "Hello, World!", tata: [1, 2, 3] }, (error) => {
    // fail :(
});

// Worker side:
process.on('message', (message) => {
    const ret: Boolean = process.send('Answer something',
    null, null, (error) => {
        // fail :(
    });
});
```

```c
#include <unistd.h>

int main(void) {
    char buffer[32768];
    pid_t pid;
    int master2worker[2];
    int worker2master[2];
    pipe(master2worker);
    pipe(worker2master);
    if ((pid = fork()) == 0) {
        // worker
        close(master2worker[1]);
        close(worker2master[0]);
        write(worker2master[1], "Hello, I'm worker!", 19);
        while (1) {
            read(master2worker[0], buffer, 32768);
            // do something
        }
    } else {
        // master
        close(worker2master[1]);
        close(master2worker[0]);
        write(master2worker[1], "Hello, I'm master!", 19);
        while (1) {
            read(worker2master[0], buffer, 32768);
            // do something
        }
    }
    return 0;
}
```

Calling C++

nodejs > JS call-cpp.js > ...

```
1   const lrt = require('./ng_lrt'); // Import C++ .node object
2   lrt.initLibrary('666'); // Call C function `int initLibrary(const char *);`
3   // Create a new C++ CDelivery_Receipt_Instruction instance:
4   dri = new lrt.DeliveryReceiptInstruction();
5   dri.setMember('DRI_CODE', '12345678'); // call C++ SetMember method
6   dri.setMember('DRI_LINK', 'A001'); // call C++ SetMember method
7   dri.load(); // call C++ COM_Select method
8   ext = dri.getExtension(); // call C++ getExtension method
9   status = dri.getMember('IL_STA_ID'); // call C++ sGetMember method
10  dri.create(); // call C++ SIB engine (init, phases 1, 2, 3, ...etc)
```

When to use C++ in a Node.js project?

- Binary operations (e.g binary masks, packing/unpacking, hardware operations, …etc)
- Exposing a native library functions to Node.js
- Optimizing local code speed - scientific algorithms (e.g gravity simulations, collision engines, rasterization, ray tracing, …etc)
- Deporting code on accelators (GPUs, FPGAs, custom ASICs, …etc)
- Process manipulation (system calls, signals handling, memory manipulation, custom loading procedures)
- Device management (for example USB/PCI(e) device drivers / access in)
- Network protocols (at the low-level implementation)
- Strict resource management (for example in embedded world with restrictions such as available RAM in bytes or program memory available also in the bytes or kilo-bytes level)
- Kernel / Firmware code
- Machine Learning
- Big Data
- Signal Processing
- Many others…
- Don’t forget inline assembly ;)