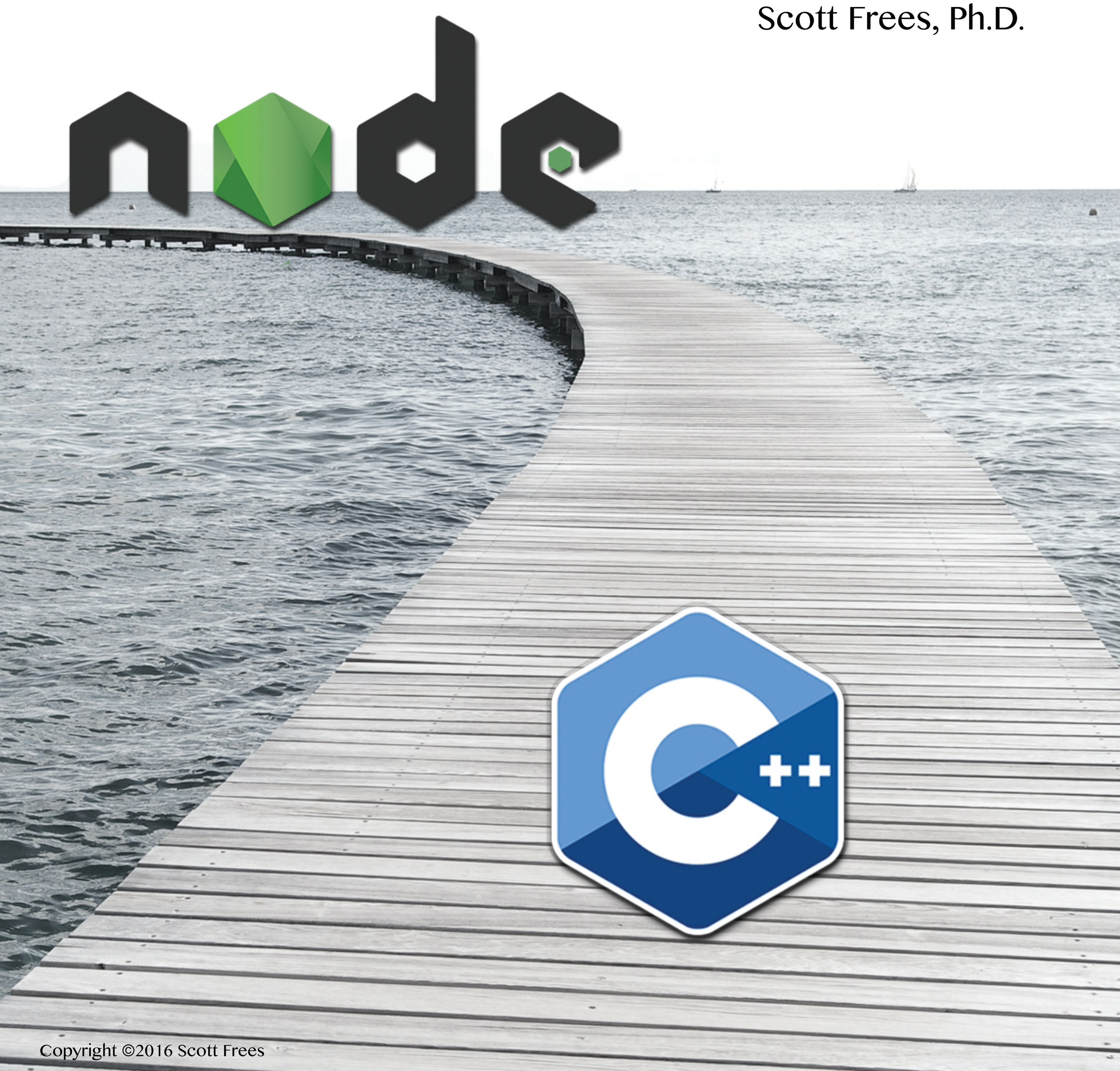
C++ and Node.js Integration

Handbook for creating Node.js C++ addons

Scott Frees, Ph.D.



Copyright ©2016 Scott Frees

C++ and Node.js Integration

Handbook for creating Node.js C++ addons

Scott Frees, Ph.D

© 2016 Scott Frees. All rights reserved.

Contents

About this book

I began programing in C++ as a student, almost 20 years ago and I haven't stopped using it both professionally and as a professor ever since. Most of my professional work involves web development however - and after doing many projects in many different languages and frameworks, no platform has made me more happy than Node.js. In 2014, while working on a project to integrate a legacy C++ tool (genomic sequence alignment) with a new web app, I stumbled upon C++ addons for Node.js. The chance to integrate two languages that I enjoy working with was exciting, so I began to teach myself how to write addons from resources online.

It was painful!

Not long after I put together my first few addons, I decided to write this book. Over the course of about a year and a half, I've written dozens of addons, researched the way V8 and Node.js work, I read pretty much every SO and blog post on the internet dealing with the topic. This book is my effort to synthesis all that I've learned into a complete handbook - to save you the trouble I went through getting up to speed.

This book is *not* a reference manual. You won't find exhaustive API listings here, the book is written around examples - focusing on *concepts* - not facts that you can find in API docs. I want this to be a *practical guide* for creating addons - I hope you find it helpful. As you read the book, you'll no doubt want to keep V8 and Node.js documentation handy, as they are indispensable.

This book contains a *lot* of source code listings. In order to make things look ok for a variety of ebook formats, I've severely limited the number of columns I use for code - which at times makes things look a bit strange. I highly recommend that you visit my github repository, `nodecpp-demo`, which contains nearly all the code presented in this book. In each chapter, I've referred you to the specific directories within that demo repository where you can find the code being discussed in the text.

You can grab the source code at https://github.com/freezer333/nodecpp-demo.

Who is this book for?

This book is for anyone wanting to start out with or learn a bit more about Node.js C++ addons. Although nowhere in the book do I go out of my way to use advanced features of C++ or JavaScript, I do expect the reader to be fairly well familiar with these languages. Towards the middle of the book (Chapter 4), the book takes a more technical turn - dealing with issues such as threading and shared memory. I try to explain these topics a bit, however generally speaking the reader should already have a bit of multi-threaded programming experience before trying to absorb those examples. While I've used some new(ish) C++ features in my addons, most anyone with a moderate level of C++ experience should have little problem digesting the code.

This book is *not* for someone trying to learn Node.js, nor is it for someone whose never programmed in C++. While all the source code is freely available at the book's github repository, please keep in mind that I've creating the examples for educational purposes - I'm quite positive there are ways to optimize the code or make the code more "production ready", but when in doubt I've opted for simplicity. Further along those lines, while using C++ for really heavy lifting sometimes gives you a performance boost over pure JavaScript - none of the examples shown in this book are likely to be justified by a performance increase. They are specifically designed to be simple demonstrations to teach you how to integrate the two languages.

Node.js has many versions, at the time of this writing version 6.0 is readying for release. I've geared most examples and discussion for Node.js versions 0.12 and above, and all examples have been tested on Node.js v5 and below.

Regarding versions, and creating "production ready" code - I'd be remiss if I didn't explain my choice of introducing NAN, an abstraction library that shields developers from version changes, so late in the book (Chapter 6). My goal for this book is to *educate the concepts* of addon development, and I found that by learning the basic V8-level API first (which I did, somewhat accidentally), one is better able to understand NAN and it's higher level abstractions. As you progress through the book, you'll see that the more advanced concepts (progress async workers, streaming) are discussed *after* NAN, so we can use the simplifications it provides. My advice to anyone learning addons is to learn the "V8 way" first, but to write final production code using NAN.

About the author

Scott Frees is a Professor of Computer Science at Ramapo College of New Jersey. His research background is in immersive virtual environments, user interface design, and more recently bioinformatics and web applications. He teaches introductory programming courses in C++ and Data Structures, along with upper level courses in Web Development, Computer Graphics, Operating Systems, and Database Systems at Ramapo College.

In addition to his post at Ramapo College, Scott is an active professional software consultant- with over 11 years of experience working as an application developer. Scott worked as a systems developer and consultant in the oil and gas industry for eight years, which included developing thermodynamic simulations used by General Electric and helping the National Institute of Standards and Technology create B2B electronic data exchange standards for capital facilities. He has developed custom desktop applications and web server back-ends for clients using C++, C#/ASP.NET MVC, Java, PHP, and Node.js.

You can see more about Scott's work at Ramapo at his college web page: http://pages.ramapo.edu/~sfrees

More information about his professional work can be found at: https://scottfrees.com

Acknowledgments

I'd like to first thank the hundreds of fellow developers who have make their code, their questions, and their blog articles freely available online - without their willingness to put the information out there I can't imagine I would have been able to learn this subject or write this book.

I'd like to thank all the people who signed up for updates about the development of this book at my website. The steady stream of signups, blog post feedback, and just general engagement gave me the motivation I needed to finish this book.

I'd of course also like to thank my family - my wife Sarah and my daughter Abigail - for putting up with me working on yet another project!

Chapter 1 - Introduction

In this first chapter we're going to dive right into making a few really simple Node.js addons in C++. The examples are limited, so we don't need to tackle too much of the V8 API too soon. They really serve as a launching point into the rest of the book, as we go through them I'll point you to different chapters throughout the book where you can get more details on various topics.

I assume you have a working knowledge of C++ and JavaScript - at least from a syntax perspective. If you are a JavaScript developer who has never written C++, some of this content is going to take some time to digest. If you are C++ developer, with little JavaScript experience, you likely will have it a bit easier - simply because most of our code will be C++. Nevertheless, this chapter will be important for you since it will give you some insight as to how JavaScript actually works with Node.js and the V8 engine.

Why build a C++ addon?

Before we start - let's answer the most obvious question... why would we want to build a C++ addon and call it from JavaScript?

First off, recognize that while raw *speed* is probably the knee-jerk reaction - especially if you are C++ developer - make sure you carefully consider this. C++ certainly runs faster than JavaScript - but it doesn't outperform it as much as some people like to believe. JavaScript doesn't have a "speed". The JavaScript *interpreter*, which translates the code into executable binary instructions, determines the speed - and that program is usually a web browser. Up until around 2008, execution time of JavaScript was extremely slow compared to C++ - however things have changed since then. Modern JavaScript engines (we don't like calling them *interpreters*, since they are actually compiling code and doing all sorts of optimization that we don't normally associate with that term) have achieved dramatic speed gains - and can at times achieve performance rivaling C++ for certain tasks. While at the time of this writing (2016) Node.js only runs on the V8 JavaScript engine from Chrome, other JavaScript engines such as Mozilla's SpiderMonkey and Microsoft's Chakra are pushing the speed boundaries even further on the front end and there are projects aimed at targeting them for Node.js as well. Don't jump to conclusions - before writing a C++ addon for speed, do some testing in pure JavaScript first.. you might be surprised!

All that said, C++ should deliver at least a 2x runtime improvement over JavaScript for most CPU-bound tasks. This can be even more so if your algorithms can run faster by optimizing on low level architecture features (cache size, etc.) that aren't available in JavaScript. So as a first reason... yes, if you've got a critical CPU-heavy task to complete in a Node.js application -

building a C++ addon is likely a great choice[1].

A second reason for writing a C++ addon - and one that I'd argue is likely *more* common - is that you *already have some C or C++ code*, that you want to integrate into Node.js. Node.js is a great platform for building web applications, micro-services, and other I/O tasks - but sometimes when adopting Node.js you need to migrate over C++ code that is already important to your application or business. You don't need to rewrite it in JavaScript - you can turn it into an addon! If you are in this situation, be sure sure to review Appendix A as well as the core chapters in this book - they cover alternative ways of integrating existing C++ code.

What is a Node.js Module?

So we know we want to develop some C++ code that can be added into (it's an *addon*, after all) our JavaScript code. What mechanism does JavaScript have for organizing packages? If you've been paying attention to the JavaScript world on the front-end, you know that while standards are emerging for this, JavaScript has always been plagued by a lack of packaging mechanism. Node.js, however, already has such a mechanism built in[2].

In Node.js, modules represent individual units of code. Modules have a 1:1 relationship with files - one module goes into one file. Any JavaScript file (when run in Node.js) can be a module, all it needs to do is add properties (or functions) to the `exports` property, which is implicitly defined by Node.js.

```
// hello_module.js
exports.say_hello = function() {
    return "hello world";
}
```

To import this module into another script file, we utilize the `require` method, with a relative path to the module (without a file extension).

```
// hello.js
var hello = require("./hello_module");
console.log(hello.say_hello());
```

[1] There is another reason using C++ may not directly lead to performance increases - and that is the cost of data copying. Chapter 2 will describe exactly why we must always copy data out of V8 before utilizing it fully, and this will be extended in Chapter 4 when we look as asynchronous addons especially. Typically our addon needs to be *long running* before it's runtime performance begins to outweigh the cost of copying memory.

[2] The module/require pattern Node.js uses is actually a partial implementation of the CommonJS module pattern. EMCAScript6 has standardized a similar strategy as well. For the purposes of this book, since we are not dealing with "front-end" JavaScript, we'll pretend the Node.js way is the only way.

The returned object from require *is* the `exports` object - so we can now call `say_hello` on that object.

```
$ node hello.js
hello world
```

One of the things that makes this import/module system all the more powerful is `npm` - the Node.js package manager. With an account on npmjs.com, you can publish your own modules, and by utilizing `npm install` you can easily pull down any one of the tens of thousands of open source modules that make up the Node.js ecosystem. If you've built Node.js apps before you've likely installed modules using `npm` many times. We won't talk to much more about `npm` until Chapter 8 - but if you are unfamiliar with it - you might want to read up on it before you continue.

What is a Node.js Addon?

OK, so if you didn't already know how all those module's you've been plugging into your Node.js apps were built - now you do! What you might have also observed when including modules is that some of them seem to require `npm` to do a bunch of work during the install process. A good example of this is `bcrypt`, a library for doing cryptographic hashing:

```
$ npm install bcrypt
-
> bcrypt@0.8.5 install /home/sfrees/projects/tmp/node_modules/bcrypt
> node-gyp rebuild
>
make: Entering directory `/home/sfrees/projects/tmp/node_modules/bcrypt/build'
  CXX(target) Release/obj.target/bcrypt_lib/src/blowfish.o
  CXX(target) Release/obj.target/bcrypt_lib/src/bcrypt.o
  CXX(target) Release/obj.target/bcrypt_lib/src/bcrypt_node.o
  SOLINK_MODULE(target) Release/obj.target/bcrypt_lib.node
  COPY Release/bcrypt_lib.node
make: Leaving directory `/home/sfrees/projects/tmp/node_modules/bcrypt/build'
bcrypt@0.8.5 node_modules/bcrypt
|__ bindings@1.2.1
|__ nan@2.0.5
```

A quick look at that output and you'll see the tell tale signs of a **Node.js** native addon written in C++. The third line of the output shows us that bcrypt is issuing a `node-gyp rebuild` command, and we can see that a `Makefile` is being executed to build CXX targets. `bcrypt` is largely written in C++ .

Using `bcrypt` is easy - there is no difference between using a native C++ addon and a normal JS module - you just `require` it.

```
var bcrypt = require('bcrypt');
bcrypt.genSalt(10, function(err, salt) {
    bcrypt.hash('B4c0/\/', salt, function(err, hash) {
        // Store hash in your password DB.
    });
})
```

Of course, in order for *your own* addon (or JS package) to be installed and included like this, you need to package it up and publish it - which I'll show you how to do in Chapter 8.

Node.js and the V8 Engine

We know why we'd like to write addons in C++, and we know there are plenty of modules out there that can be seamlessly integrated into our Node.js applications. So the next question is, how is this done? What is providing this nice bridge between these two *very* different languages? The answer, of course, is the **Google V8 JavaScript library** - the *open source* JavaScript execution engine that ships with the Google Chrome web browser. V8 is a *C++ library* which provides a high performance execution environment for JavaScript code. When V8 shipped as part of Chrome in 2008 it was envisioned as a library for executing JavaScript loaded in the browser, at the request of the loaded web document - but things changed in 2009 when Ryan Dahl created Node.js.

Node.js has brought JavaScript server-side development into the mainstream. It is built on top of V8, but instead of being a web browser, it provides access to the underlying operating system. Before you put your security hat on, remember that we aren't talking about *arbitrary* JavaScript coming from the web - we're talking about locally developed (or at least willfully copied) code. Essentially, Node.js gives JavaScript the system API calls that languages like C, C++, Java, Python, etc. have always had.

Node.js is really just glue and support code around a few key components:

1. Google's V8 JavaScript engine
2. C++ modules for low-level I/O operations (files, threads, sockets, etc.)
3. An event-loop (much more on this later)

Of course, that's a simplification - but the idea is that Node.js plays a similar role as the Chrome web browser does (see Figure 1). Where a web browser like Chrome (written in C++) exposes host objects like `window` and the HTML DOM to JavaScript executing within V8, Node.js exposes low-level I/O - such as the file system, network, threading, etc. This is why Node.js is perfect for *server* development - it has provided access to the OS from JavaScript.

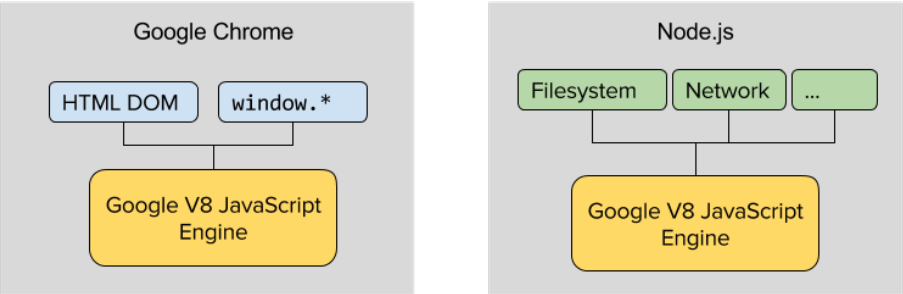


Figure 1: Node.js provides a "host" for JavaScript executing in the V8 Engine, just like Chrome does for the web.

So now we see that the JavaScript interpreter (actually, it's more like a JIT compiler) is just a C++ library. You can actually embed V8 in your own C++ applications to execute JavaScript code contained in regular C++ strings - without Node.js. If you want to take a look - check out the examples in Google's Embedder's Guide.

While nice (and extremely useful in many situations), we're not really interested in using V8 directly like this. We are more interested in the fact that V8 provides a C++ API *into* the JavaScript that is executing. This API allows us to write C++ code that can be loaded by Node.js - and this C++ code becomes fully callable from your JavaScript code. Along the same lines, the API allows us to hold resources allocated by JavaScript in C++, and even call JavaScript functions from C++.

Let's get started!

Hello World - Returning data from C++ to JavaScript

I'm going to start off with a few examples that are currently on the official Nodejs.org API manual. Don't worry - you weren't tricked into purchasing a book that just turns a bunch of freely available examples in to a PDF - I'll be going into far more detail in subsequent chapters. For now though, nothing beats a few "hello worlds" to get things going - so I'm not reinventing the wheel.

There are going to be three essential components to each of our initial examples - (1) a C++ file that holds our C++ addon, (2) a binding.gyp file that sets up our build, and (3) our JavaScript file that actually calls the addon. For simplicity, I'm putting all three in a single directory - but more exotic layouts are frequently used (and often addons are developed without a JavaScript file that calls it). We'll defer the discussion on `binding.gyp` for a bit - let's just start out with the code.

As our first example, our C++ addon will expose a single method - `hello` that returns the string `world`. We'd use it from JavaScript like this, which would be in `hello.js`:

```
// hello.js
const addon = require('<path to addon>');

console.log(addon.hello()); // 'world'
```

Note `<path to addon>` will get filled in later in this discussion, after we build the addon. As you can see though, our C++ addon will be callable just as easily as modules like `bcrypt` and pure JS modules are called.

Now let's take a look at the C++ code, which would be stored in 'hello.cpp'.

```
1  #include <node.h>
2
3  using namespace v8;
4
5  void Method(const FunctionCallbackInfo<Value>& args) {
6    Isolate* isolate = args.GetIsolate();
7    Local<String>  retval = String::NewFromUtf8(isolate, "world");
8    args.GetReturnValue().Set(retval);
9  }
10
11 void init(Local<Object> exports) {
12   NODE_SET_METHOD(exports, "hello", Method);
13 }
14
15 NODE_MODULE(hello_addon, init)
```

All of the code above, and in the rest of this chapter is available in full in the `nodecpp-demo` repository at https://github.com/freezer333/nodecpp-demo, under the "Basics" section.

Before we go any further, let's get a few of the questionable data types and methods clarified. On lines 1 and 3 you see we're just including our headers and declaring the v8 namespace.

Where is `node.h` you ask? Well - be patient... we'll see that we'll use a build tool specially designed to pull in Node.js dependencies in a moment. For now, just understand that by including `node.h` we've also pulled in headers for V8 - which we'll start using now.

From the bottom up - registering the module

First, let's remember that hello.cpp aims to be a Node.js module. If we look at line 15, we see `NODE_MODULE(addon, init)`. This is a C++ `macro` - it's not a function call. This macro creates code that will (when loaded by Node.js) register a module named "hello_addon" with the JavaScript context (V8). In addition, it will ensure that a function called `init` is called when the module is `required`. That `init` function is on line 11, and repeated here.

```
11 void init(Local<Object> exports) {
12   NODE_SET_METHOD(exports, "hello", Method);
13 }
```

Remember that when creating a JavaScript module, your `js` file usually adds properties (objects, methods) to the `module.exports` object. The parameter of the `init` function is that very same `module.exports` object - and we are adding a method to it. The data type is `Local<Object>`, which means it's a JavaScript object, defined only within the scope of this function call (we'll get into this more in a moment). The call to `NODE_SET_METHOD` adds a function `Method` to the `exports` object, and makes it callable by the name `hello`.

The `Isolate` object, `Locals`, and `FunctionCallbackInfo`

Now we get to the definition of the actual C++ function being made callable from JavaScript - `Method`. It's a really simple function - it just returns the string "world" - but the method below seems to have some pretty complicated stuff in it...

```
6  void Method(const FunctionCallbackInfo<Value>& args) {
7    Isolate* isolate = args.GetIsolate();
8    Local<String>  retval = String::NewFromUtf8(isolate, "world");
9    args.GetReturnValue().Set(retval);
10 }
```

As will be described in much further detail in Chapter 2, V8's core job is to manage the execution of JavaScript code - and that principally means it needs to manage all the memory that your JavaScript (Node.js) code allocates and de-allocates. The fact is, the JavaScript running in V8 (which will call this method) doesn't have direct access to "memory" like C or C++ does - it's memory is completely managed by V8; JavaScript variables/object occupy chunks of memory *within* V8 address space.

This issue is the real reason returning "world" to JavaScript appears complex. Specifically, the memory that holds the character string "world" cannot simply

be allocated like a C++ string (on *actual* main memory) - it needs to be allocated within V8's memory store which is serving JavaScript. The V8 API exposes ways of allocating these variables from C++ in a variety of ways (the details of which will talk more about in Chapter 2) - but a common theme is that we must do so through an `Isolate` object.

An `Isolate` object represents the V8 instance itself. The name comes from the fact that the execution engine, when embedded in a web browser like Chrome, could be executing many JavaScript environments simultaneously - associated with web pages running in separate tabs and windows. When we have multiple pages open, each page's JavaScript instance must be *isolated* from the others - otherwise one page's script could access/modify another's global objects (like `window.location` or `window.document`)... which would be **bad**. V8 uses the notion of an `Isolate`, combined with a `Context`, to provide these parallel execution environments.

On line 7, we see the first (of a few) ways we can obtain an instance of an `Isolate`. The parameter passed to `Method` represents the calling context of the function, from JavaScript. Among other things, it contains the current scope, parameters, and return address/site of what will be returned. It of course also knows very well which instance of V8 it was called from - so the availability of the `args.GetIsolate()` method seems reasonable.

On line 8 we use the isolate when invoking a factory call on `v8::String`'s static `NewFromUtf8` method. This method allocates room for "world" within the memory associated with the `isolate`. Exactly *where* this variable is allocated is important - and we get a clue from the data type the return value is being stored in.

V8's mechanism for allowing your C++ code to create and hold variables accessible to JavaScript is the `Handle` - which comes in two general flavors - `Local` and `Persistent`. Chapter 2 will deal with this in greater detail, for now let's be content with the idea that `Local` handles manage memory scoped to the current function call. Handles are template classes, capable of holding any V8 basic data type (Value, Object, Array, Date, Number etc.).

```
6  void Method(const FunctionCallbackInfo<Value>& args) {
7    Isolate* isolate = args.GetIsolate();
8    Local<String>  retval = String::NewFromUtf8(isolate, "world");
9    args.GetReturnValue().Set(retval);
10 }
```

Now that we have a variable `retval` containing "world", we're ready to return it to the calling JavaScript code. Remember again though, V8 has called this function - the JavaScript only caused V8 to call this - it didn't call it itself!. All methods that V8 calls return `void` - the mechanism of setting return values is through the `FunctionCallbackInfo` class.

The `FunctionCallbackInfor` class contains all the information one would need to know about how the method was invoked from JavaScript. It will contain the parameters of the call, the `isolate`, the `this` object you'd normally expect to have if this were a JavaScript function, and finally... a slot to store what will be returned to JavaScript by V8. The Set method on the `v8::ReturnValue` is overloaded to accept any of the standard V8 data types.

Building Addons with `node-gyp`

C++ code needs to be compiled into executable form - whether it be as an object file to be linked with others, a shared library, or a standalone executable. I assume you are somewhat familiar with building C++ on your platform/OS of choice (or all of them!) - but to build our addons we'll want to use a tool chain specifically setup for Node.js addons. One reason for this is that we need to link to the Node.js dependencies and headers correctly. Secondly, when we publish our modules (Chapter 8), we'll need a cross-platform way for `npm` to build our C++ source into binary for the target platform.

At time of this writing (2016), `node-gyp` is the de-facto standard build tool for writing Node.js addons. It's based on Google's `gyp` build tool, which abstracts away many of the tedious issues related to cross platform building. While `gyp` has little to do with Node.js, `node-gyp` makes creating addons a relative breeze by setting up dependencies and target formats (addons will be binary files with a `.node` extension) for us automatically.

Pre-requisites for building

While `node-gyp` handles orchestrating builds, we still need to have an appropriate C++ tool chain installed on our machine to build (or install, by the way!) our addon. Before moving forward, you'll need to make sure you have the following:

1. Python 2.7+, but **not Python 3.x**. `node-gyp` uses Python for part of it's build process, but it is not compatible with Python 3 - you must have a 2x version installed on your machine. They can be installed side-by-side, so if you want Python3 too - it's not a problem.
2. Build tools for your platform - if you are on Linux/Mac OX you'll need `make` and `clang++`. On OS X, these are installed when you install XCode. On Windows, you need Visual Studio installed on your system.

Once you have those setup (and on your path), you can install `node-gyp` using `npm`. Be sure to install it globally!

```
$ npm install node-gyp -g
```

Configuring the build

We configure the build process by creating a `binding.gyp` file in the same directory as our `hello.cpp` file. This binding file's primary purpose is to tell `node-gyp` which source code files to build, and what to name the output.

```
{
  "targets": [
    {
      "target_name": "hello_addon",
      "sources": [ "hello.cpp" ]
    }
  ]
}
```

The `sources` array is self explanatory - however one important thing to note is that the `target_name` is **not arbitrary**, it must specifically match up with what you named your module on line 15 of the addon - the `NODE_MODULE(hello_addon, init)` call. You've been warned.

Now it's time to build. Within the directory of your `hello.cpp` and `binding.gyp` file, issue the configure and build command:

```
$ node-gyp configure build
```

As long as you don't have any syntax errors in your C++ source code, this command will successfully produce build files (`Makefile` or Visual Studio project files) and build the `hello_addon.node` file. It will be located in `build/Release`.

There's a lot more we can do with `node-gyp`. Many of the examples to follow will add compiler options to enable C++11 and C++14 features, and we'll push further with it still in Chapter 8 and Appendix A, where we'll see how we can build standalone applications and shared libraries with `node-gyp` as well.

But for now... let's run our JavaScript and call our Addon!

Calling your Addon from JavaScript

At this point, your directory structure should be something like this (I've left out a bunch of files `node-gyp` creates along the way):

```
/project_root
  |__ .
  |__ ..
  |__ build
```

```
|     |__ Release
|           |__ hello_addon.node
|__ hello.cpp
```

At the project's root directory, right next to `hello.cpp`, create a new JavaScript file called `hello.js`. This file will call our addon by first `require`-ing it using *relative path* syntax (see Chapter 8 for actually publishing addons and then being able to require them like normal modules).

```
// hello.js
const addon = require('./build/Release/hello_addon');

console.log(addon.hello()); // 'world'
```

Remember that we attached `Method` (line 5 in `hello.cpp`) to the module's `exports` object when we called `NODE_SET_METHOD(exports, "hello", Method)` on line 12 - but we called it `hello` (the second parameter). Thus, holding the `addon` object (the `exports` object) returned from require, we can simply call the `hello` function to invoke our C++ code.

```
$ node hello.js
world
```

Passing Parameters from JavaScript to C++

In the first "hello world" example we simply returned a string to the calling JavaScript. The return value was set by accessing the `FunctionCallbackInfo` data structure V8 hands us when our addon's method (`Method`) is called. It likely comes as no surprise that this `args` object is also where we'll get the parameters/arguments our addon's method is called with from.

Let's turn our attention to our next example, where we build a C++ addon that adds two numbers and returns the sum. We'd call the addon from JavaScript like this:

```
const cmath = require('./build/Release/cmath');
const five = cmath.add(2, 3);
console.log( '2 + 3 is ', five );
```

Let's create this example in a folder called `cmath`, and start with `cmath.cpp`. Defining the addon's entry point and initialization code is virtually identical to that of our first "hello world" example - which becomes a bit of a recurring theme!

```
#include <node.h>
using namespace v8;

// Called when addon is require'd from JS
void Init(Local<Object> exports) {
    NODE_SET_METHOD(exports, "add", Add);  // we'll create Add in a moment...
}

// The addon's name is cmath, this tells V8 what to call when it's require'd
NODE_MODULE(cmath, Init)
```

The code above is pretty much "boilerplate" code to bootstrap an addon - we've invoked the `NODE_MODULE` macro to tell V8 that when someone loads an addon called `cmath` our `Init` function should be called. Our `Init` function registers a single method on the `exports` object (as we'll see, we can register many if we wish). Now let's look at the `Add` method itself:

```
1  void Add(const FunctionCallbackInfo<Value> & args) {
2      Isolate * isolate = args.GetIsolate();
3
4      // value is created on the C++ runtime stack, not as part of
5      // the JavaScript execution context...
6      double value= args[0]->NumberValue() + args[1]->NumberValue();
7
8      // Now we create it in the JS execution context so we can return it
9      Local<Number> num = Number::New(isolate, value);
10     args.GetReturnValue().Set(num);
11 }
```

Line 2 gets a reference to our now familiar `Isolate` - which represents the execution context the calling JavaScript is running within. Line 6 is where we extract the arguments from the `args` object. It's also our first run-in with one of the various ways we can cast V8 `Value` types. The `args` object overloads the `[]` operator and returns `Local<Value>` objects corresponding to the parameters the method was called with. `Local<Value>` is a wrapper around a `Value` object - locally bound to the current JS function invocation.

As Chapter 2 will discuss, `V8::Value` acts as a base type for the data types V8 provides to map against JavaScript data types. For the C++ developers reading this - `V8::Value` sits at (close to) the top of an inheritance hierarchy which includes `Object`, `Array`, `Date`, `Function`, JS primitives like `Boolean`, `Number`, `String`, and many more. In JavaScript, there is little distinction between integers and float/doubles in the basic language, and thus the `Number` type - which holds floats. Incidentally, in V8, we can also use `Integer` types if we wish - they are subclasses of `Number`. Chapters 2 and 3 will also get into using `Objects` and `Arrays` in increasing detail.

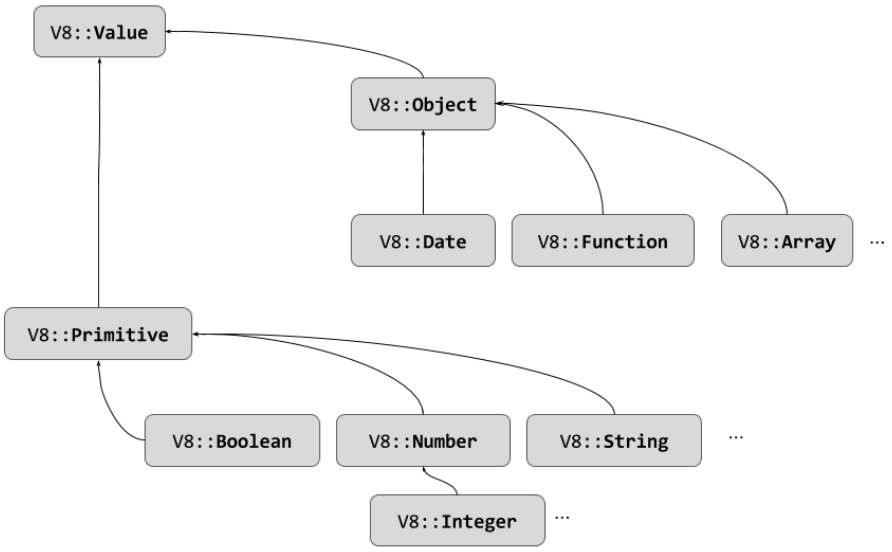


Figure 2: Simplified inheritance hierarchy for V8 data types.

`Value` objects can can hold data corresponding to any of their subclasses. Assuming you know their underlying data type, you may use a corresponding accessor to retrieve the value. So, for example, if we know arguments 0 and 1 are numbers, we can call `NumberValue` on them to return a pure C++ double value representing the parameter. If we wanted to simply cast the `Value` to the V8 representation (and thus, work directly with the data held in the V8 execution context as a `Local`), we could call `ToNumber` instead - which returns `Local<Number>`.

Line 6 adds the pure C++ primitives retrieved by calling `NumberValue` and stores it in a plain-old C++ primitive. Importantly, this variable is just a normal stack variable in C++ - we can't return it to JavaScript. To allocate this data in the V8 execution context, we wrap it in a new `Local<Number>` object using a static `New` method defined on the `Number` data type. This call (line 9) accepts the `isolate` and allocates / assigns the value in the V8 runtime. Finally, we can return the value the same way we returned our "hello world" string - as shown on line 10.

Inspecting the arguments before using them

If directly calling `NumberValue` on `args[0]` and `args[1]` causes you some pause... it should! Unlike C++, JavaScript doesn't force callers to use the "correct" number of parameters when calling a function. Even though we'd like our `Add` function to accept two numeric parameters - we don't have any way

of ensuring this. The following calls to `add` from JavaScript are all perfectly legitimate (even though they might not make sense):

```
const cmath = require('./build/Release/cmath');

var x = cmath.add();
var y = cmath.add(5);
var z = cmath.add(5, "Confused");
```

Knowing this, we'd be a bit foolish to simply assume (1) that the caller has provided us both `args[0]` and `args[1]`, and that (2) they are both numeric values! Thankfully, the V8 API provides us with some easy ways to inspect our `args` object before blissfully extracting our numbers. Let's revise our `Add` method to first check that there are the correct number of arguments:

```
1  void Add(const FunctionCallbackInfo<Value> & args) {
2      Isolate * isolate = args.GetIsolate();
3
4      if (args.Length() < 2) {
5          return;
6      }
7
8      // value is created on the C++ runtime stack, not as part of
9      // the JavaScript execution context...
10     double value= args[0]->NumberValue() + args[1]->NumberValue();
11
12     // Now we create it in the JS execution context so we can return it
13     Local<Number> num = Number::New(isolate, value);
14     args.GetReturnValue().Set(num);
15 }
```

One line 5, we simply return if the caller did not specify enough arguments. **The actual result of our call will show up as `undefined` in JavaScript** - since we never called `args.GetReturnValue().Set(...)`. If you instead want to really punish the caller for using your addon incorrectly, you can throw an exception - the choice is yours (and probably depends on the usage of your addon):

```
// alternative to returning undefined
if (args.Length() < 2) {
    // Throw an Error that is passed back to JavaScript
    isolate->ThrowException(Exception::TypeError(
        String::NewFromUtf8(isolate, "Wrong number of arguments")));
    return;
}
```

It's not enough to know there were two arguments though, we also need to ensure both arguments were numbers. As shown above, the `Value` object has accessor functions to return `Local` handles that wrap the underlying subclass (i.e. `ToNumber`) as well as accessors to return the corresponding C++ primitive (i.e. `NumberValue`). `Value` also has query methods to determine if the data belongs to a corresponding subclass. We can inspect each argument to see if it's actually a number as follows:

```
if (!args[0]->IsNumber() || !args[1]->IsNumber()) {
    return; // undefined goes back to the caller
}
```

Similar query methods exist for all subclasses of `Value`. Similarly, we could opt to throw an exception in this case as well.

Calling JavaScript functions from C++

As a last "tease" for the remainder of the book, the introduction wouldn't be complete without dealing with the ever-present JavaScript "callback". A callback is nothing more than a function that you pass to some other part of your code, with the expectation that the recipient code will "call" the function later... essentially "calling you back". Callbacks exist in virtually every language (although more directly in *functional languages*) - but in JavaScript (or more specifically, Node.js), they play a very large role in development.

In order for C++ to call a JavaScript function (callback), the callback must be passed into the C++ module. Since functions are just normal data in JavaScript, the V8 API defines a `Function` subclass of `Value`. This subclass can be used in a similar way to the `Number` subclass we saw in the previous example. In this current example, we'll build a simple addon that instead of *returning* data, sends a result back to JavaScript by calling the *callback* it is provided.

Here's what the calling JavaScript might look like:

```
const callback = require('./build/Release/callback');

var callme = function() {
    console.log("I've been called!");
}

callback.callthis(callme);
```

When executed, we'll see "I've been called!" print out, because our C++ calls the function itself. I'll leave out the `NODE_MODULE` and `Init` boilerplate, as it's basically the same as the previous examples:

```
1  void CallThis(const FunctionCallbackInfo<Value>& args) {
2    Isolate* isolate = args.GetIsolate();
3    Local<Function> cb = Local<Function>::Cast(args[0]);
4    cb->Call(Null(isolate), 0, nullptr);
5    return;  // undefined returned to JavaScript
6  }
```

On line 2 we get a reference to the V8 execution context - `isolate`. Line 3 casts `arg[0]` (a `Local<Value>`) to a function object - `cb`. In a bit of a V8 quirk, the API is a little inconsistent here. Instead of calling something like `ToNumber` as we would to convert the `Local<Value>` to a `Local<Number>`, we must use the casting syntax here. There is good reason for this (involving type coercion), but let's defer that to Chapter 2 where we'll dive into the data types further.

Line 4 is where we invoke the callback - using the `Call` method. `Call` accepts three parameters:

1. Receiver - which is the JavaScript object that `this` will be bound to when this JavaScript callback is invoked. We are setting this to `null`, which will just set `this` to whatever the default is within the JavaScript code.
2. Number of Arguments - zero in this case, but would correspond to the number of parameters we will pass to the callback when it is executed.
3. Arguments - `null` here because we aren't passing any arguments - otherwise we'd be passing in an array of `Local<Value>` objects.

On execution, we'll call whatever function was passed into `CallThis`'s first parameter.

Calling callbacks with parameters

A slight variation on the example above would actually make use of the ability to call function callbacks with parameters. Let's write an addon that could be used like this:

```
const callback = require('./build/Release/callback');

var callme = function(message) {
    if ( message ) {
        console.log(message);
    }
    else {
        console.log("I've been called!");
    }
}
```

```
callback.callthis(callme);
callback.callthis_withthis(callme, "This is an important message");
```

The first thing you should notice is that there are two methods being exported by the `callback` addon from the last example. Your addons can export any number of methods - you just need to make additional calls to `NODE_SET_METHOD` within your `Init` function.

```
void Init(Local<Object> exports, Local<Object> module) {
  NODE_SET_METHOD(exports, "callthis", CallThis);
  NODE_SET_METHOD(exports, "callthis_withthis", CallThisWithThis);
}
```

Here's the implementation of `CallThisWithThis` - which simply passes `arg[1]` through to the callback function defined in `arg[0]`.

```
1  void CallThisWithThis(const FunctionCallbackInfo<Value>& args) {
2    Isolate* isolate = args.GetIsolate();
3    Local<Function> cb = Local<Function>::Cast(args[0]);
4
5    // Create an array with only the argument passed in (the message)
6    Local<Value> argv[1] = {args[1]};
7    cb->Call(Null(isolate), 1, argv);
8  }
```

Of course, we could consolidate this into a single C++ function by checking the number of parameters passed in instead - but I'll leave that as an exercise for the reader.

It is extremely important to note that the callback examples we've just seen execute callbacks **synchronously**, meaning that when we call `callthis` from JavaScript, `callthis` does not actually return until *after* the C++ invokes the callback and the callback returns. This is most definitely *not* the way we expect callbacks to work in Node.js typically! Chapter 4 and 7 will deal with this topic in more detail.

Take-aways and what's coming next...

Hopefully this chapter has wet your appetite for learning how to write C++ addons for the Node.js platform. We've seen some really basic examples, but the concepts introduced will be leveraged again and again throughout this book. This chapter also serves as a bit of an outline for the rest of the text - in the next chapter we will look at the the V8 data types in much more details, and then in Chapter 3 we'll put that knowledge to use by seeing practical ways to

work with Arrays and Objects. In Chapter 5 we'll see how we can wrap C++ objects so they can be used directly from JavaScript. By the time we work through all those details, you'll have a very clear picture of just how complex the V8 API can get, and we'll use `NAN` in Chapter 6 to simplify some things. Chapters 7 will then look more closely at function callbacks, specifically focusing on *asynchronous* callbacks and streaming patterns. In Chapter 8, we'll conclude by learning how to publish our native addons - which will bring the details on `npm`, `node-gyp`, and the build system into more focus.

Chapter 2 - Understanding the V8 API

In Chapter 1 we learned just enough of the V8 API to be dangerous - *which is not where we want to be*! In order to make use of the V8 documentation online, you'll need to have a fundamental grasp of how the system works. To really start developing powerful addons to serve all of your use cases, you'll need to learn the details of the memory management and data type facilities V8 provides. These topics are the focus of this chapter. This chapter serves as the foundation of the next few chapters as well, which will show you how to deal with arrays and objects. The topics in this chapter also play a critical role in understanding how asynchronous addons function, as we'll highlight the different ways we can allocate data in C++ through V8.

JavaScript memory allocation

The key to understanding V8 data types - and how we use them inside our C++ addons - is to understand how V8 stores the memory associated with the variables JavaScript code creates. A few key points are worth remembering, especially if you are relatively new to Node.js:

1. Node.js is a C++ program. It uses two libraries to provide much of it's structure - V8, which is the JavaScript code runtime; and libuv, which implements the event loop (much more on this in Chapter 4 and 7).
2. V8 is a C++ library. It's your JavaScript's "world" so to speak - when your JavaScript code creates variables, the space created for them is created *inside* V8's address space - in it's heap. We'll call the memory associated with a JavaScript variable a **storage cell** to keep this distinction clear.
3. JavaScript variables are *garbage collected* - meaning V8 must implement a garbage collector (there are tons of resources online that show you how it does this). To do this, V8 must keep track of how many references are pointing to each **storage cell**.
4. V8's C++ data types allow your C++ code (addon) to access the very same **storage cells** that the executing JavaScript code can.
5. Lastly, **storage cells** are only eligible for garbage collection when there are no references to them in JavaScript code, *but also* no references to them in any C++ code. This point is critical to understanding `Local` vs. `Persistent` handles later in this chapter!

The take-away here is that there is a *huge* difference between a variable your C++ code creates in standard fashion (`double x`) and a `V8::Number`. The `double` is getting created on the stack (or heap) of the C++ program (essentially, Node.js), while the `V8::Number` is a C++ object which contains a reference to a **storage cell**, managed entirely by the V8 runtime (and of course, stored on the C++ program's heap as well).

Isolates

Before tackling how actual memory is allocated and accessed, let's investigate the idea of *isolates* a bit further than we did in Chapter 1 - as references to isolates are virtually everywhere in the V8 API. An `Isolate` is an independent instance of the V8 runtime - complete with execution context (control flow), memory management/garbage collection, etc. A C++ program can create multiple *Isolate* objects in order to provide multiple, parallel, and *isolated* execution environments for scripts to run. Most importantly, each `Isolate` contains it's own heap - *it's own pool of memory from which storage cells for the variables created by the JavaScript executing within it.*

Node.js creates a single `Isolate`, and there is no API for creating additional ones via JavaScript (not sure if that even makes sense!). If you were embedding V8 in your own C++ application, only then would creating multiple `Isolate` objects be possible. Isolates have an important role in the limitations placed on multi-threaded code. There is a strict *single thread* rule for a given `Isolate` - meaning only one thread is allowed to access an `Isolate` and it's resources at a given time. In the Node.js environment, this thread is already spoken for (it's the event-loop, your JavaScript code - or the C++ called from it) - accessing `Isolate` resources from other C++ threads is impossible[3].

If you were embedding V8 in your own C++, you *could* access a single `Isolate` using multiple threads. You'd need to perform synchronization to ensure no two threads accessed the `Isolate` simultaneously however. This synchronization must be done through V8's `Locker` and `Unlocker` classes. V8 enforces this rule strictly - if you try get around it, you'll take an unexpected detour to `segmentation fault` city - regardless of if an actual race conditions exits or not.

The threading rules associated with `Isolate` objects is critical to understanding the limitations of C++ addons and multi-threading - which will be discussed in further detail towards the end of this chapter, and in Chapter 4.

Contexts

While an `Isolate` encapsulates an execution environment and heap, it does not fully provide the requirements for running JavaScript code. The running of JavaScript requires a `global` object. The `global` object will have properties attached to it such as `Infinity`, `undefined`, `NAN`, and `null`. It has functions associated with it - like `eval`, `isNan`, `parseFloat`, etc. It also contains the

[3]Specifically, the reason it's impossible is because Node locks it's isolate using `V8::Locker` before entering it's main message pump loop, and never relinquishes it. If you attempt to unlock (using `V8::Unlocker`) from the event-loop thread, Node will never attempt to re-acquire the lock and you'll crash the program. At time of this writing, you can plainly see this taking place in `StartNodeInstance` within `node.cc` in Node's source code - found at https://github.com/nodejs

Objects we are familiar with (`Object`, `Function`, `Boolean`, etc). These are all defined in the EMCAScript standard.

A full execution environment also has a *global scope* - which will contain objects created by user scripts or provided by the host (in this context, the host is the C++ program instantiating V8). In the browser world, globally scoped objects would include `window`, `document`, `alert`, `setTimeout()`, etc. Node.js provides global objects/function such as `console`, `exports`, `module`, `process`, `require()`,. A complete list can be found at https://nodejs.org/api/globals.html.

V8 represents the "global" idea as a `V8::Context`. A `Context` object corresponds to a global object in the `Isolate`'s heap. An `Isolate` may have any number of `Contexts` active, each operating with their own independent globals. In a web browser, the C++ browser code will create separate `Context` objects for each tab - as each tab's JavaScript should execute independently. Any changes to objects in the global space of one tab would not affect another's (such as changing `window.location` or `document`!).

In Node.js, we typically only have a single implicit `Context`. You can actually compile and execute JavaScript in new contexts through the `vm` object however. While sometimes useful, use cases for this functionality is relatively rare and won't be discussed much further in this book.

From this point forward, unless we must make the distinction in code, when we speak of an `Isolate` we are speaking of the isolate and the context created by Node.js. This single execution environment executes all of our Node.js JavaScript code, and will be accessed via the V8 API by our C++ addons when we need to allocate / read / or write to JavaScript memory.

JavaScript variables, C++ Handles

Now let's look at JavaScript variables, from the perspective of Node.js and V8. When we create an object in JavaScript, `var obj = {x : 5};` V8 dutifully creates a storage cell in the `Isolate`'s heap - as shown in Figure 3.

Now let's take a look at how this particular variable could be accessed by a Node.js C++ addon. Aside from trivial use cases, addons will need to either accepts parameters sent from JavaScript or return data to JavaScript - and likely they will do both. Let's build a simple addon now that exposes a single method - `mutate` which accepts an object to change via a parameter:

```
1  void Mutate(const FunctionCallbackInfo<Value>& args) {
2      Isolate * isolate = args.GetIsolate();
3      Local<Object> target = args[0]->ToObject();
4      target->Set(
5          String::NewFromUtf8(isolate, "x"),
6          Number::New(isolate, 42));
7  }
```

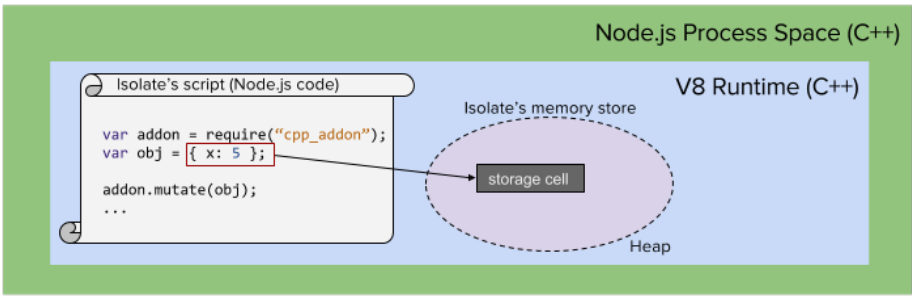


Figure 3: JavaScript variables being allocated as storage cell inside an isolate's heap.

In the code above, I've left out the necessary includes, initialization (NODE_SET_METHOD) and the text of the binding.gyp file - see Chapter 1 for details. The interesting stuff is happening in the `Mutate` function. When our Node.js code invokes `Mutate`, the Node.js runtime creates a `FunctionCallbackInfo` object and passes it to our function. On line 2, the first action is to obtain a reference to the current `Isolate`. This is the way we can begin to access anything within the running JavaScript program - without it we are limited to just normal C++ stack and heap variables.

On line 3, we obtain a `Local` handle to an object passed in as `arg[0]`. We'll cover type conversion in a moment (`ToObject`), but for now let's focus on the what exactly `arg[0]` is. V8 defines data types that correspond to JavaScript variables stored in storage cells - and since JavaScript defines many data types, the C++ V8 engine models this as a type *hierarchy* - with `v8::Value` at the top of the inheritance chain. `arg[0]` is of type `v8::Value`, which represents a *reference* into V8's storage cells to a particular cell where the first parameter is held.

In theory, we should be able to manipulate the underlying JavaScript object directly - yet the V8 API requires us to wrap this `v8::Value` in a *handle* instead. Recall that V8 is responsible for *garbage collection* - thus it is extremely concerned with knowing precisely how many references exist to storage cells within an isolate. A *handle* is an object that represents the existence of such a reference *in C++* - as V8 not only needs to keep track of references to storage cells originating from JavaScript, but also in the addons JavaScript may invoke.

There are two types of handles - `Local` and `Persistent`. A `Local` handle is allocated on the C++ call stack, and wraps a given `v8::Value`. The creation of a `Local` implicitly invokes the `Local`'s constructor - which in turn allows the necessary V8 bookkeeping to take place, ensuring V8 knows there is now a new outstanding reference to this particular storage cell.

`Local` handles are by far the most common type of handle we see in addon development. Once we have one, we may access and manipulate the underlying

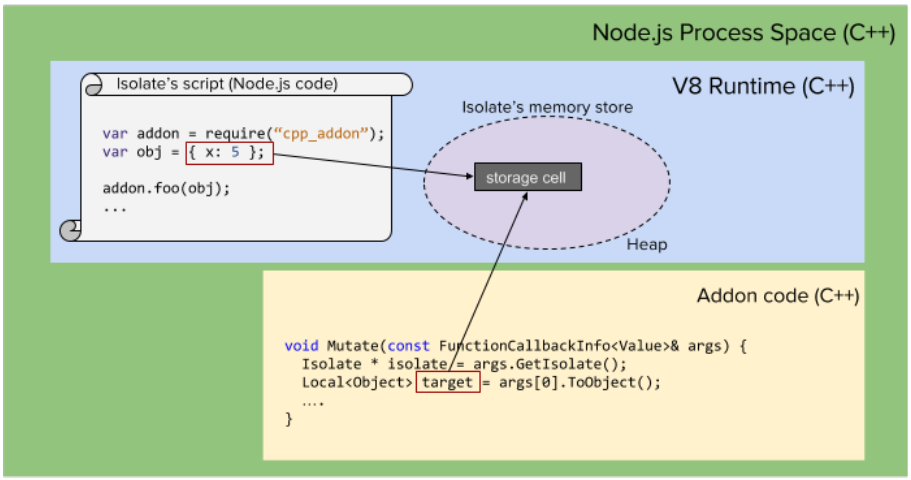


Figure 4: Code within a C++ addon accessing storage cells within an isolate's heap.

data it wraps - typically parameters sent into our addon. As we'll see below, we may also allocate new storage cells - typically for the purposes of returning references to them back to JavaScript. As is the case in any C++ function, when our addon function returns (back to Node.js/V8), the *destructor* of any `Local` handles created within the function is called - which will notify V8 that *this* particular reference to the storage cell is no longer valid.

HandleScope

Actually - while the above paragraph serves as a reasonable mental model of what is going on in typically Node.js addons - it's actually *a bit* more complicated than this... `Local` handle objects do not directly perform bookkeeping in their constructors and destructors - there is a level of indirection that goes unseen in synchronous addons, but is a critical part of V8. The indirection is that `Local` handles are associated with a `HandleScope` object - which is a container for bunches of `Local` handles.

At a given time, there (must) always be one `HandleScope` object active within a given isolate. Whenever `Local` handles are created, they are added to the currently active `HandleScope` object. Each new `HandleScope` that is created automatically becomes "active" and all `Local` handles created afterwards will be associated with it. When a `HandleScope` is deleted, the previously active `HandleScope` is made active. It is only when the `HandleScope` object's destructor is called that storage cells referenced by the contained handles are considered eligible for garbage collection (assuming no other handles or JavaScript references point to them).

At the time of this writing, `HandleScope` objects are front and center Google's V8 Embedder's Guide - however for a simple synchronous Node.js addon - the concept is never used, which leaves many a bit confused. To resolve the confusion, we must again remember that JavaScript most definitely *does not* call our addon function *directly*. When JavaScript code invokes an addon - its Node.js (C++) that will actually take the V8 objects associated with parameters and package them up into the `FunctionCallbackInfo` and call your addon. It is here, **before** your C++ addon function is called, that a `HandleScope` is created. This means that for C++ addon functions called from JavaScript, it is not mandatory to create a new `HandleScope` (although you may, if you wish). As will be described when we discuss *asynchronous* callbacks - C++ code that is going to use V8 but is not directly called by JavaScript **will need** to create a `HandleScope` of it's own however.

I'm going to defer discussing `HandleScope`s much further, and save more of the discussion for when we cover asynchronous callbacks and are *forced* to deal with the issue. For now, we can simply remember that when our C++ addon is invoked, any local handle it creates will implicitly be associated with a `HandleScope` already created - **which will be deleted** when we return back to JavaScript.

V8 Data Types

Now that we have a bit of an understanding of handles, and how they allow us to interact with storage cells - let's look at what sort of storage cells we can actually work with. This means taking a look at what data types V8 supports - which unsurprisingly match right up with JavaScript data types!

Rather than simply listing out data types (I assume you know what a string is!), let's look at these data types from the context of being passed in as arguments to a C++ addon and returning modified values back to C++. All of the code for this chapter is available in full in the `nodecpp-demo` repository at https://github.com/freezer333/nodecpp-demo, under the "Conversions" section.

Primitives - Strings, Numbers, and Booleans

JavaScript primitives include Strings, Numbers, and Booleans - along with `null` and `undefined`. V8 uses an inheritance hierarchy in which `Primitive` extends `Value`, and all the individual primitives subclass `Primitive`. In addition to the standard JavaScript primitives, V8 also supports integers (`Int32` and `Uint32`).

JavaScript has a very flexible type casting system - i.e. using the string "42" as an number automatically converts it to a number primitive and using `undefined` as a string will result in it turning into the literal "undefined". This functionality, while foreign to a C++ developer, is fully supported by V8.

As we go through the API for working with primitives, you will notice that there is no facility for assignment - which at first may seem odd! In fact, it makes a lot of sense however - for three reasons:

1. JavaScript primitives are *immutable* - which means from an underlying storage mechanism, variables "containing" primitives are just pointing to unchanging (and created on-demand) storage cells. Assignment of a `var x = 5;` makes `x` point to a storage cell with 5 in it - reassigning `x = 6` does not change this storage cell (at least, in concept - the interpreter can "cheat") - it simply makes `x` point to another storage cell that contains 6. If `x` and `y` are both assigned the value of `10`, they both point to the same storage cell. The same holds for strings and booleans.
2. Function calls are pass-by-value, so whenever JavaScript calls a C++ addon with a parameter - if that parameter is a primitive, it is always a distinct *copy* - changing it's value has no effect in the calling code.
3. Handles, as described, are references to *storage cells*. Thus, given #1 above, it doesn't make sense to allow handle values to change - since primitives don't change!

Hopefully that makes some sense - however it's still likely you'll need to *modify* V8 variables... we'll just need to do this by *creating* new ones and assigning the new value to them.

Now let's look at the common primitive types - `Number`, `String`, and `Boolean`. For each, we'll build an addon function that accepts a single primitive, transforms it in some way, and returns the transformed value. Along the way, we'll look at how V8 handles casting, the creation of new primitives, and how we can detect alternative conditions - such as when `null`, `undefined`, or an `Object` type is passed in.

Converting a V8 Number to a C++ double

Let's start with a simple conversion of a JavaScript number being passed into an addon function.

```
void PassNumber(const FunctionCallbackInfo<Value>& args) {
    Isolate * isolate = args.GetIsolate();

    Local<Number> target = args[0]->ToNumber();
    double value = target->NumberValue();

    // value is now OUTSIDE of V8 - we can use it in
    // all the C++ standard ways.
}
```

On line 4 we convert the first argument passed to C++ into a number using the `ToNumber` method on `v8::Value`. Note here that the `->` syntax is be used on `Local<Value>` objects to call methods on their underlying V8 data types - the `Local` object overloads the pointer dereferencing operator.

The first conversion to `Local<Number>` is actually unnecessary though - since the `NumberValue` function is actually (somewhat oddly[4]) defined on `v8::Value`. So the code below is perfectly equivalent:

```
1  void PassNumber(const FunctionCallbackInfo<Value>& args) {
2      Isolate * isolate = args.GetIsolate();
3
4      // We don't need to convert to `Local<Number>` explicitly...
5      double value = args[0]->NumberValue();
6
7      // value is now OUTSIDE of V8 - we can use it in
8      // all the C++ standard ways.
9  }
```

Creating new V8 Numbers and assignment

Now that we have a C++ value, we can modify it - perhaps by adding a fixed value to it. To return new value, we need to create a new storage cell and return a reference back to JavaScript.

Creating new V8 numbers is straightforward - it is accomplished using the static factory method `New` defined on `v8::Number`.

```
1  void PassNumber(const FunctionCallbackInfo<Value>& args) {
2      Isolate * isolate = args.GetIsolate();
3
4      double value = args[0]->NumberValue();
5
6      value+= 42;
7
8      Local<Number> retval = Number::New(isolate, value);
9
10     args.GetReturnValue().Set(retval);
11 }
```

Line 8 creates the storage cell (note that the `isolate` must be passed into the `New` method). We simultaneously assign the value as we allocate it - again,

[4] It is, in fact, not that odd. Values in JavaScript can be cast to different primitives in a lot of ways. It actually makes a lot of sense that the responsibility for casting a value into a specified primitive rests with `v8::Value`. Unlike C++ - there is no "true" type that we should be casting to given a value. An instance of Value is legitimate, and it can be transformed into something like a number or string on demand.

since primitives are immutable in JavaScript. Line 10 then returns it, using the standard method discussed in Chapter 1.

Calling from JavaScript results in the expected results for standard number values.

```
1  const addon = require('./build/Release/primitive');
2
3  var number_returned = addon.pass_number(23);
4  console.log(number_returned); // prints 65
5
6  var number_returned = addon.pass_number(0.5);
7  console.log(number_returned); // prints 42.5
```

Safety checks - making sure it's a number

Of course, there is no guarantee that the first argument is indeed a JavaScript number. Let's investigate what happens should we invoke this code in a variety of different use cases.

As a first check - what happens if a first argument isn't even specified?

```
var number_returned = addon.pass_number();
console.log(number_returned); // NAN!
```

We're getting our first glimpse of V8's ability to "do the JavaScript thing". If our addon was a normal JavaScript function, `args[0]` would be `undefined`. When used in a mathematical expression, `undefined` ends up giving you `NAN` - so we are returning `NAN + 42`. So... lesson learned: if you call `NumberValue()` on a `Local<Value>` that stores `undefined`, you get `NAN`.

Now is a good time to review how you could preemptively check for this though. First, `FunctionCallbackInfo` exposes an `Length` function that can let us catch situations where we don't have the arguments we expect. In addition, `v8::Value` has query functions defined on it so you can interrogate the value to determine it's underlying type. There is an `IsArray`, `IsBoolean`, `IsDate`, `IsFunction`, `IsInt32`, `IsNull`, `IsNumber`, `IsObject`... you get the idea! They all return boolean results. If you prefer to throw exceptions or return error values if you don't get the right parameters in your addon, you may certainly do so.

```
void PassNumber(const FunctionCallbackInfo<Value>& args) {
    Isolate * isolate = args.GetIsolate();

    if ( args.Length() < 1 ) {
        args.GetReturnValue().Set(Number::New(isolate, 0));
        return;
```

```
    }
    // This would catch anything that can't be a Number
    if ( !args[0]->IsNumber()) {
        args.GetReturnValue().Set(Number::New(isolate, -1));
        return;
    }
    ...
```

What if we stretch this example a bit, and pass in an argument from JavaScript that aren't *necessarily* true number objects? We can get a sense of how V8 handles these situations if we *do not* check for non-numbers. Let's return to our naive addon:

```
void PassNumber(const FunctionCallbackInfo<Value>& args) {
    Isolate * isolate = args.GetIsolate();

    // naively assume args[0] is a number
    double value = args[0]->NumberValue();

    Local<Number> retval = Number::New(isolate, value + 42);
    args.GetReturnValue().Set(retval);
}
```

Let's call this with a bunch of non-numbers.

```
number_returned = addon.pass_number("23");
console.log(number_returned);
// Prints 65 - string is parsed

number_returned = addon.pass_number(null);
console.log(number_returned);
// Prints 42 - null is 0

number_returned = addon.pass_number(undefined);
console.log(number_returned);
// Prints NaN, undefined -> NaN

number_returned = addon.pass_number("this is not a number");
console.log(number_returned);
// Prints NaN, string can't be parsed

number_returned = addon.pass_number({x: 5});
console.log(number_returned);
// Prints NaN, object cannot be cast
```

If you wrote a pure JavaScript counterpart for our addon, you'd get all the same results as the addon above produces - which is exactly the way we'd want things to work! In all of these cases, if we checked `args[0]->IsNumber`, it would have returned false. The takeaway here is that you have a choice - you can write your addons to rigidly accept true numbers, or you can allow V8 to do "the JavaScript thing" - which often times is the best approach.

Converting a V8 Number to a C++ int

While JavaScript doesn't necessarily force us to distinguish between integers and real numbers, C++ of course does. If we wish to use signed or unsigned integers in our addon, we could (1) simply convert our C++ `double` into an `int` using standard C++ methods - or (2) we can use some of V8's API to do the work.

```
void PassInteger(const FunctionCallbackInfo<Value>& args) {
    Isolate * isolate = args.GetIsolate();

    int value = args[0]->Int32Value();

    Local<Number> retval = Int32::New(isolate, value + 42);
    args.GetReturnValue().Set(retval);
}
```

When called from JavaScript with an integer, things are pretty mundane. If JavaScript calls this function with say `5.7` as a parameter, V8 automatically cuts off the precision and `Int32Value` still returns 5 - resulting in 47 being returned. Alternatively, we can use `IsInt32()`, which would return false given `5.7`, allowing us to take corrective action.

Using `Uint32` instead of `Int32` allows you to work with unsigned ints in the same manner.

Working with Boolean parameters and return values

The API for converting from V8 to C++ boolean values is quite similar to numbers.

```
void PassBoolean(const FunctionCallbackInfo<Value>& args) {
    Isolate * isolate = args.GetIsolate();
    bool value = args[0]->BooleanValue();

    Local<Boolean> retval = Boolean::New(isolate, !value);
    args.GetReturnValue().Set(retval);
}
```

Just like with numbers, we can test to see if the given argument is a boolean or not using `IsBoolean`.

Truthy vs. Falsey values

The JavaScript rules for converting other data into booleans is fairly straightforward. `Null`, `Undefined`, 0, and empty strings are always converted to `false`. Everything else (including empty objects and arrays) are considered `true`. The `BooleanValue` function honors this specification and will convert anything referenced by the `v8::Value` to a boolean in the expected manner.

Converting a V8 String to a C++ string

In some respects, accepting strings into an addon is simpler than number values, since we're usually a lot more permissive about what can be represented as a string. On the other hand, the V8 API is a little funny with strings - rather than having a simple analog to the `NumberValue()` method on `v8::Value`, we instead need to instantiate a `v8::String::Utf8Value` object from a given handle. This instance is a wrapper around a null-terminated `char *`, and can be dereferenced to obtain the underlying `char *`. This `char *` can then be used directly, or wrapped in a standard C++ string.

Let's look at an addon that reverses a given string and returns it.

```
void PassString(const FunctionCallbackInfo<Value>& args) {
    Isolate * isolate = args.GetIsolate();

    v8::String::Utf8Value s(args[0]);

    // wrap in c++ string instead
    std::string str(*s);
    std::reverse(str.begin(), str.end());

    /// need to return this...
```

Creating new V8 strings and assignment

Just as with `Number` objects, strings are immutable - we are manipulating the C++ representation of the string - not the V8 storage cell. To return the transformed string, we must move it back into V8 by creating a new `String`. Older versions of V8 also supported `AsciiValue`, however this is no longer a supported type.

```
void PassString(const FunctionCallbackInfo<Value>& args) {
    Isolate * isolate = args.GetIsolate();

    v8::String::Utf8Value s(args[0]);
    std::string str(*s);
    std::reverse(str.begin(), str.end());
```

```
    // Create a new V8 string out of the char *
    Local<String> retval =
        String::NewFromUtf8(isolate, str.c_str());
    args.GetReturnValue().Set(retval);
}
```

Note that the `String::NewFromUtf8` factory method accepts a `const char *`, not a C++ string - so we've accessed it via the standard `c_str()` method.

String representations of other JavaScript types

Just like with numbers, if we receive a value from JavaScript we can always check to see if it truly is a string - this time by using the `IsString` method.

```
void PassString(const FunctionCallbackInfo<Value>& args) {
    Isolate * isolate = args.GetIsolate();

    if ( !args[0]->IsString() ) {
        // uh oh... this isn't a string
    }
    ...
}
```

Of course, we can just ignore this little problem, and ask V8 to convert the given argument to a string. Much like numbers, this works just like the JavaScript spec. If we pass the following values into the addon, we can see how non-strings get returned (remember, the addon is reversing the string before it returns it).

```
var string_returned = addon.pass_string("The truth is out there");
console.log(string_returned);  // prints "ereht tuo si hturt ehT"

string_returned = addon.pass_string(42);
console.log(string_returned);
// prints 24 - the 42 is easily made to a string

string_returned = addon.pass_string(null);
console.log(string_returned);
// prints llun - "null" backwards!

string_returned = addon.pass_string(undefined);
console.log(string_returned);  // prints "denifednu"

string_returned = addon.pass_string({x: 5});
console.log(string_returned);
// prints ]tcejbO tcejbo[ - "[Object]" backwards
```

In most situations, converting numbers and booleans to strings implicitly likely makes sense. Converting `null`, `undefined`, and objects may end up being undesirable in some cases though - but we can easily check for these:

```
void PassString(const FunctionCallbackInfo<Value>& args) {
    Isolate * isolate = args.GetIsolate();

    if ( args[0]->IsNull() ) {
        args.GetReturnValue().Set(Null(isolate));
        return;
    }
    else if ( args[0]->IsUndefined() ) {
        // if we never set the return value, it will be undefined.
        return;
    }
    else if ( args[0]->IsObject() ) {
        // we can just return the same reference to the object, rather
        // than reversing it...
        args.GetReturnValue().Set(args[0]);
        return;
    }

    v8::String::Utf8Value s(args[0]);
    std::string str(*s);
    std::reverse(str.begin(), str.end());

    Local<String> retval =
        String::NewFromUtf8(isolate, str.c_str());
    args.GetReturnValue().Set(retval);
}
```

In the code above, functions, arrays, regular expressions, and all other JavaScript object types will get caught by the `IsObject` function. There are also more specific queries if you want to handle those differently.

A common thread between all primitives is that we must copy data out of V8 storage cells and into C++ address space to manipulate them. We must then allocate new V8 storage cells in order to return our manipulated data. This work flow is predicated on the fact that primitives are *immutable* - a property that JavaScript objects do not share.

Objects

Objects represent our first *mutable* data type. In JavaScript, the term "object" has dual meaning - there is a standard `Object` type, and there are other *object*

sub-types that extend `Object`, such as `Array` and `Function`. For now, let's just focus on the typical, standard object:

```
var obj = {
    x : 5,
    y : "hello",
    z : true
}
```

Accessing JavaScript objects passed to C++

When objects are passed from JavaScript we can create local handles to them in much the same way as we've done with primitives. In the example below, we'll receive an object and set it's `y` property to a fixed value of `10` before returning it back to JavaScript.

```
void PassObject(const FunctionCallbackInfo<Value>& args) {
    Isolate * isolate = args.GetIsolate();
    Local<Object> target = args[0]->ToObject();

    target->Set(String::NewFromUtf8(isolate, "y"),
        Number::New(isolate, 10));

    args.GetReturnValue().Set(target);
}
```

Once we have a `Local` handle wrapping the object, we can begin manipulating properties within the referenced object using `Set` and `Get`. Note that the `Set` (and `Get`) methods **require us to refer to property names using V8 strings**, not C++ strings. If we were to call this from JavaScript, we'd see that a new property `y` is now on the returned object:

```
var retval = addon.pass_object({x : 3});

console.log(retval);  // prints {x : 3, y: 10}
```

We can of course overwrite an existing property as well, so passing in an object with `y` already present behaves as expected:

```
var retval = addon.pass_object({x : 3, y : "hello"});

console.log(retval);  // still prints {x : 3, y: 10}
```

At this point, it is instructive to note the second parameters of the `Set` method (Line 5 of C++ code above, repeated below).

```
target->Set(String::NewFromUtf8(isolate, "y"), Number::New(isolate, 10));
```

While objects are *mutable* - meaning their properties may be changed - the properties *themselves* could be primitives (which is the case here), which means they are **immutable**. Therefore, to set `y` we must always pass in a newly created (or previously existing) V8::Value - we are not simply *overwriting* `y` - we are reassigning it. We'll cover nested objects in Chapter 3, but for now, know that we could of course assign a object to `y` as well.

To modify a property based on it's current value, we must perform similar steps as we did previously with primitives: we must (1) copy the V8 property into a C++ variable, (2) manipulate the data, and (3) copy the C++ data into a new storage cell and assign as appropriate. In the code listing below we use the `Get` method to retrieve a `Handle` to the `y` property and copy the data into C++ for manipulation.

```
void PassObject(const FunctionCallbackInfo<Value>& args) {
    Isolate * isolate = args.GetIsolate();
    Local<Object> target = args[0]->ToObject();

    Local<String> prop = String::NewFromUtf8(isolate, "y");

    Local<Number> y_handle = target->Get(prop)->ToNumber();
    double y = y_handle->NumberValue();

    target->Set(prop, Number::New(isolate, y + 42));

    args.GetReturnValue().Set(target);
}
```

The same can be achieved more succinctly using `NumberValue` directly on the handle returned by `Get`. More generally, the return value of `Get` is simply a `Local<Value>` - which can be used in all the same ways it was used in the sections earlier in this chapter.

```
1  void PassObject(const FunctionCallbackInfo<Value>& args) {
2      Isolate * isolate = args.GetIsolate();
3      Local<Object> target = args[0]->ToObject();
4
5      Local<String> prop = String::NewFromUtf8(isolate, "y");
6
7      double y = target->Get(prop)->NumberValue();
8
9      target->Set(prop, Number::New(isolate, y + 42));
10
```

```
11      args.GetReturnValue().Set(target);
12  }
```

Assuming `y` already exists as a number within the object passed from JavaScript, the returned object will have a `property` which is incremented. If the object passed does not have a `y` property, or it's `y` property is not a number type, then the mathematics will reflect the rules outlined in the `Number` discussion above.

```
var retval = addon.pass_object({x : 3, y: 10});
console.log(retval);  // prints {x : 3, y: 52}

var retval = addon.pass_object({x : 3, y: "hello"});
console.log(retval);  // prints {x : 3, y: NaN}

var retval = addon.pass_object({x : 3});
console.log(retval);  // prints {x : 3, y: NaN}
```

Note that `Get` returns `Undefined` objects if a requested property does not exist. If we want to avoid getting `NaN`, we could detect that the property is undefined, and preemptively set it to 0 before accessing it again to do the computation.

```
// before converting y to a number, init to 0 if it doesn't exist...
if ( target->Get(prop)->IsUndefined() ) {
   target->Set(prop, Number::New(isolate, 0));
}

// continue to work with "prop"
```

Creating and returning new JavaScript objects from C++

Much like primitives, we can create new objects on the V8 heap using a factory method. Properties can be added using the `Set` method just like on preexisting objects. The following example accepts an object with `x` and `y` numeric properties and returns a **new** object containing the sum and product.

```
1  void PassObject(const FunctionCallbackInfo<Value>& args) {
2      Isolate * isolate = args.GetIsolate();
3      Local<Object> target = args[0]->ToObject();
4
5      Local<String> x_prop = String::NewFromUtf8(isolate, "x");
6      Local<String> y_prop = String::NewFromUtf8(isolate, "y");
7      Local<String> sum_prop = String::NewFromUtf8(isolate, "sum");
8      Local<String> product_prop = String::NewFromUtf8(isolate, "product");
9
```

```
10      if ( !target->Get(x_prop)->IsNumber() ) {
11          target->Set(x_prop, Number::New(isolate, 0));
12      }
13      if ( !target->Get(y_prop)->IsNumber() ) {
14          target->Set(y_prop, Number::New(isolate, 0));
15      }
16
17      double x = target->Get(x_prop)->NumberValue();
18      double y = target->Get(y_prop)->NumberValue();
19
20      // Create a new object to return to V8
21      Local<Object> obj = Object::New(isolate);
22      obj->Set(sum_prop, Number::New(isolate, x + y));
23      obj->Set(product_prop, Number::New(isolate, x * y));
24
25      args.GetReturnValue().Set(obj);
26  }
```

As a final note, keep in mind that JavaScript object types do not (and cannot) automatically map into C++ objects. In Chapter 5 we will learn how to approximate this idea using `ObjectWrap` - however we must always remember that the two concepts are fundamentally different.

Arrays

Arrays in JavaScript are simply objects, with special treatment of properties that represent whole numbers. Arrays can have non-indexed properties, and they can also have methods (`Functions`) such as `length()`. Values held at indexes can be of any data type, and they need not be all of the same type. Needless to say - they *are not much like C++* arrays - they just appear so!

Accessing Arrays passed from JavaScript

Somewhat surprisingly, converting a `Local<Value>` into a `Local<Array>` breaks the pattern we've seen being established. While you may expect `Value` to have a `ToArray` method like it's `ToNumber`, `ToObject`, etc. - it does not. Rather, we must explicitly cast it using `Array`'s static `Cast` function.

```
Local<Array> array = Local<Array>::Cast(args[0]);
```

Happily, accessing index properties in arrays is quite similar to when dealing with objects - we just use integers to access elements. We can also use the built in `Length` function that is defined on the `v8::Array` object as a way to loop through the array. The below addon method increments each numeric value stored at an array index:

```
void IncrementArray(const FunctionCallbackInfo<Value>& args) {
    Isolate * isolate = args.GetIsolate();
    Local<Array> array = Local<Array>::Cast(args[0]);

    for (unsigned int i = 0; i < array->Length(); i++ ) {
      double value = array->Get(i)->NumberValue();
      array->Set(i, Number::New(isolate, value + 1));
    }

    // Like all objects, our changes will be reflected even if we
    // don't return - as objects (and array) are mutable.
}
```

Passing in an array of mixed types will result in some `NaN` results, which could be avoided using the techniques (i.e. `IsNumber`) described in previous sections.

```
var data = [1, 2, "hello", "world", 3];
addon.increment_array(data);
console.log(data);  // prints [ 2, 3, NaN, NaN, 4 ]
```

It's important to note that `increment_array` has not returned anything - yet the results of the array manipulation that took place within the addon are reflected in our JavaScript printout. This shouldn't be a surprise - objects (and arrays are objects) are reference types. When reference types are passed to JavaScript functions, only references are passed - not copies of the objects. By and large, C++ addons behave exactly like JavaScript functions - so we see that the addon is working directly with the underlying storage cells associated with the array it was given. The same could be done with `Objects` as well.

Unlike C++, JavaScript arrays may be sparse - which we should consider when looping through an array in our addon. The addon code below avoids processing any indices which have not already been set in JavaScript.

```
void IncrementArray(const FunctionCallbackInfo<Value>& args) {
    Isolate * isolate = args.GetIsolate();
    Local<Array> array = Local<Array>::Cast(args[0]);

    for (unsigned int i = 0; i < array->Length(); i++ ) {
      if (array->Has(i)) {
        double value = array->Get(i)->NumberValue();
        array->Set(i, Number::New(isolate, value + 1));
      }
    }
}
```

Note that the `Has` method could also be used on objects - with a property string (`Local<Value>` more accurately) rather than an index.

Creating and returning new Arrays to C++

Once again, creating Arrays and returning them follows a similar pattern as other data types. `Array` has a static `New` factory method, and we may set any indices we wish.

```
Local<Array> a = Array::New(isolate);
a->Set(0, Number::New(isolate, 10));
a->Set(2, Number::New(isolate, 20));

args.GetReturnValue().Set(a);
```

Arrays with non-index properties

JavaScript arrays can have non-array properties, since ultimately they are just objects. We can easily access these, using the same methods we used with Objects due to V8's inheritance hierarchy - `v8::Array` is just a subtype of `v8::Object`.

```
Local<String> prop = String::NewFromUtf8(isolate, "not_index");
Local<Array> a = Array::New(isolate);
a->Set(0, Number::New(isolate, 10));
a->Set(2, Number::New(isolate, 20));

// get a regular property out of array passed into addon
a->Set(1, array->Get(prop));
```

The code above retrieves the value of the "not_index" property of the array passed from JavaScript. It set's that value to be the value also stored at index 1.

Other data types

The next chapter will have a lot of examples using the primitive data types discussed above, and will delve into more of the details of working with objects and arrays. These aren't the only JavaScript data types that V8 exposes to C++ addons though. Perhaps the most important of the remaining is the `Function` object - which we'll cover in Chapter 4 and utilize extensively when we look at streaming in Chapter 7. Less commonly used (in addons) data types include `Date`, `RegExp`, and newer types just making their way into V8 like `Promise` and `Set` are also available.

Passing Local Handles and using Escapable Handles

To this point, we've only looked at using `Local` handle objects - which are scoped by the active `HandleScope` object. When the active `HandleScope` object

is destroyed, all `Local` handles created within it are marked as deleted - which means they can no longer be used from C++ to access a storage cell. Likewise, if the `Local` is the only reference to a given storage cell, that storage cell is eligible to be garbage collected. In all of the examples above, we have a implicit `HandleScope` that is active as long as we are within the code directly called by JavaScript. This is because Node.js creates a `HandleScope` object before calling our addon.

This implies that we can create `Local` handles to storage cells within our addon and pass those `Local` handles to different functions.

```
Local<Value> make_return(Isolate * isolate, const Local<Object> input ) {
    Local<String> x_prop = String::NewFromUtf8(isolate, "x");
    Local<String> y_prop = String::NewFromUtf8(isolate, "y");
    Local<String> sum_prop = String::NewFromUtf8(isolate, "sum");
    Local<String> product_prop = String::NewFromUtf8(isolate, "product");

    double x = input->Get(x_prop)->NumberValue();
    double y = input->Get(y_prop)->NumberValue();

    Local<Object> obj = Object::New(isolate);
    obj->Set(sum_prop, Number::New(isolate, x + y));
    obj->Set(product_prop, Number::New(isolate, x * y));

    return obj;
}

void PassObject(const FunctionCallbackInfo<Value>& args) {
    Isolate * isolate = args.GetIsolate();
    Local<Object> target = args[0]->ToObject();

    Local<Value> obj = make_return(isolate, target);

    args.GetReturnValue().Set(obj);
}
```

The listing above would begin with the call into `PassObject`. Prior to being called, a `HandleScope` is created by Node.js and remains active when `make_return` is called. Importantly, the `obj` handle created in `make_return` will not be destroyed when that function returns - as it belongs to the original `HandleScope` and will remain "live" until that scope is destroyed[5].

[5]Note that we're using the term "destroyed" a bit loosely. V8 doesn't change C++, the actual `Local<Object>` created in `make_return` is a best thought of as a pointer. The pointer itself is of course destroyed when `make_return` returns, along with the other C++ stack variables. The storage cell pointed to by `obj` is not deleted however - it is only deleted when the `HandleScope` is destroyed. In this way, you can think of a `Local` handle as a smart pointer

We rarely need to consider anything but `Local` handles when dealing with simple addons that immediately (or at least directly) return to C++. An exception to this is if we for some reason create our own `HandleScope` within our addon. In these cases, we must take care when passing handles to other functions, as the `Local` handle may become invalid.

```
Local<Value> make_return(Isolate * isolate, const Local<Object> input ) {
    .. same as above...

    HandleScope scope(isolate); // DANGER!  All handles contained in this now.
    Local<Object> obj = Object::New(isolate);
    obj->Set(sum_prop, Number::New(isolate, x + y));
    obj->Set(product_prop, Number::New(isolate, x * y));

    return obj;
}
```

In the case above, the creation of `scope` inside `make_return` leaves us in a dangerous situation. When `make_return` completes, the `scope` variable is destroyed, and V8 is free to delete storage cells pointed to by any handles associated with that `HandleScope` - as long as no other references exist to them. In this situation, V8 *could* garbage collect the storage cells for `obj` at the time this function returns, as there are no handles in the original `HandleScope` that refer to them.

The safe way to handle these situations is to use `EscapeableHandle` to allow a `Local` handle to *leak* out to another `HandleScope`. We would create an `EscapableHandle` in `make_return` and explicitly allow our returned handle to survive.

```
1  Local<Value> make_return(Isolate * isolate, const Local<Object> input ) {
2      .. same as above...
3
4      EscapableHandleScope scope(isolate);
5
6      Local<Object> obj = Object::New(isolate);
7      obj->Set(sum_prop, Number::New(isolate, x + y));
8      obj->Set(product_prop, Number::New(isolate, x * y));
9
10     return scope.Escape(obj);
11 }
```

While the official Google documentation promotes the idea of using new `HandleScope`s or `EscapableHandleScope`s for each function, this likely only

- one whose memory (storage cell) will be garbage collected when `HandleScope` goes out of scope - not the `Handle`.

makes sense if you are doing lots of function calling, with lots of temporary allocation in V8. The use of scopes would thus limit overall V8 memory consumption by allowing it to aggressively collect dead storage between function calls. While developing the "average" Node.js addon though, this is most likely overkill - you do not need to feel obligated to use a new `scope` in each function call if you are only using very small amounts of V8 storage!

Persistent Handles

There are use cases (as we'll see in Chapter 4) where we use asynchronous addons. In these situations, handles to data created in the C++ addon entry point will no longer be usable from other threads once the initial addon entry point has returned - since the `HandleScope` will be destroyed. Anytime we wish to hold on to references to storage cells beyond the scope of an addon's entry point we must consider using a completely different type of handle - `Persistent`. `Persistent` handles are not scoped to a `HandleScope` - you have complete control over when the handle will be destroyed/invalidated.

While this is most useful in asynchronous addon development, we can simulate a use case for `Persistent` handles by developing an addon that gets initialized with a JavaScript variable and holds on to the variable over the course of it's entire lifetime - spanning multiple calls into the addon from JavaScript.

For example, suppose we had an addon that could be used like this:

```
var target = {  x: 3  };

addon.init(target, 10);
console.log(target);

for ( var i = 0; i < 3; i++ ) {
    addon.increment();
    console.log(target);
}
```

This addon stores state - we initialize it by calling `init` and passing in an object with a numeric property called `x` and a desired increment value - in this case `10`. Subsequent calls to `increment` will increment `x` by `10`. It's a silly example - but it demonstrates an addon that is holding a reference to V8 objects (`target`) for longer than a given function call into it. The output of the code would be something like this:

```
{ x: 3 }
{ x: 13 }
{ x: 23 }
{ x: 33 }
```

The key to making this work is `Persistent` handles. Let's look at the associated addon code.

```
1  Persistent<Object> persist;
2  int increment;
3
4  void Increment(const FunctionCallbackInfo<Value>& args) {
5      Isolate * isolate = args.GetIsolate();
6      Local<Object> target = Local<Object>::New(isolate, persist);
7      Local<String> prop = String::NewFromUtf8(isolate, "x");
8      double x = target->Get(prop)->NumberValue();
9      target->Set(prop, Number::New(isolate, x + increment));
10 }
11 void Initialize(const FunctionCallbackInfo<Value>& args) {
12     Isolate * isolate = args.GetIsolate();
13
14     // set global variables so they survive returning from this call
15     persist.Reset(isolate, args[0]->ToObject());
16     increment = args[1]->NumberValue();
17 }
```

`Persistent` handles have very strict rules - they cannot be copied for any reason, and you must explicitly free/change their storage cells through a call to `Reset`. The `Initialize` (initialize in JS) function takes the first argument (a `Local<Object>`) and creates a new `Persistent` handle on line 15. Note that `persist` is actually a global variable though - so it remains in scope even when `Initialize` returns. We do something similar with `increment` on line 16, but in this case we don't hold a V8 reference - just a C++ variable.

Subsequent calls into `Increment` now convert `persist` into a `Local` handle. This is required because the API for `Persistent` handle is quite limited (in past versions of V8 it was a subclass of `Handle`, but it no longer is). From there, we modify the `target` in the same way we've done previously. Note that since `persist` still points to the same thing `target` does - the reference is preserved over many calls to `Increment`.

While this makes for a nice little toy example, it's not likely that you'll be using lots of `Persistent` handles in simple synchronous addons - holding state inside your addons is likely an anti-pattern - as it promotes programming by side-effect. In this case, you'd be better off passing the object and increment into the `increment` function each time you call it. As we begin to look at asynchronous addons however, this persistent concept will be incredibly valuable.

Chapter 3 - Basic Integration Patterns

In the last chapter we did a superficial overview of how to use the V8 API to build Node.js C++ addons. While Chapter 2 is designed to be "reference-like", it doesn't provide much context to addon development - that's what this Chapter is for! In this chapter, we'll present the first (of a few) example Node.js addons - using C++ to calculate some simple statistics about rainfall in a particular region. The example is designed to have just enough depth to show you the patterns and concepts involved when building an addon - but not so much depth as to get us bogged down in C++ rather than the integration of C++. The aim is to take what we've learned in Chapters 1 and 2 and begin to *apply* it to a more concrete example. The main concepts presented in this chapter include exchanging data (including objects and arrays) between JavaScript and Node.js. Chapter 4 and 5 will enhance these examples to use other common integration patterns such as object wrapping and asynchronous processing.

Node Versions

It's worth a quick reminder at this point (since we're going to be presenting a lot of code) that the V8 API, along with Node.js in general, does undergo changes. This book is compatible with version 0.12, 4.x, and 5.x of Node.js and the V8 API versions that are packaged with them. If you are still using version 0.10 (at time of this writing, some hosting platforms like Microsoft Azure still default to this), the example code presented will not work!

If you are struggling with keeping track of Node.js versions, it's highly recommended to use a tool such as `nvm` (Node.js Version Manager) - which allows you to install and flip between any version of Node.js. Using such a tool will greatly improve your ability to stay on top of API inconsistencies. You can find `nvm` for Linux and Mac OS X at https://github.com/creationix/nvm. For Microsoft Windows, there are few different distributions, one of which is https://github.com/coreybutler/nvm-windows.

It's important to note that while everything presented in the next few chapters uses the V8 API directly, it is also possible to shield yourself from API changes using *Native Abstractions for Node* - `NAN`. `NAN` is the subject of Chapter 6.

Integration Pattern: Data transferring

The most straightforward integration pattern involves passing data between JavaScript and C++ as simple copies. For primitives, this is really the most obvious (and only) alternative, however we'll also use this copying mechanism for objects and arrays. In particular, the pattern presented here does not share class structure between JavaScript and C++, and we are specifically not using V8 class wrapping strategies.

Our integrations will follow a simple three step process when invoked from JavaScript:

1. The C++ addon code will transfer JavaScript/V8 data into native C++ data structures (classes, STL containers, etc.).
2. The C++ addon will perform the necessary calculations/processing on the data and store it's results in native C++ data structures
3. Before returning, the C++ addon will transfer the results held in the native C++ data structures into the appropriate JavaScript/V8 data structures.

The main reason for using this approach is simplicity. For one, the V8 API impact on data structures in C++ (objects, arrays, etc) will minimized. Using this approach, any existing C++ code can be used without modification, which is an enormous advantage. In addition, there will be a complete decoupling of the JavaScript and C++ code - held together by the data "transfer" code responsible for steps 1 and 3.

From an organization standpoint, this simplicity also allows us to completely separate the "transfer" code from the core C++ logic we are implementing. As you might have guessed from Chapters 1 and 2, the transfer logic won't be the "cleanest" looking code - the V8 API is cumbersome. In the author's experience, keeping the cumbersome V8 API integration code isolated from core C++ business logic is a huge win.

There are some downsides to the "transfer" approach. One is that it does indeed require you write the transfer boilerplate logic - nothing is done automatically for us. While certainly a negative, as you'll see when we look at other strategies later in this book, there really isn't much avoiding this logic - the complexity is simply inherent in the V8 API itself. Using techniques such as ObjectWrap and Buffers still require lots of "boilerplate" as well. The second downside is performance - copying data takes time. As discussed a bit in Chapter 2, since data held in V8 is fundamentally different than normal C++ data, avoiding this penalty is much more difficult than it might initially seem. We'll discuss this issue a bit more at the end of the chapter, but for now know that under normal circumstances, this performance penalty is often not significant when building real-world addons.

Rainfall Data Example

Over the course of this chapter we will be constructing an addon that can accept rainfall data from JavaScript code (i.e. amount of rainfall over periods of time at given locations). The JavaScript program will send an object containing rain fall sample data to C++ for processing. The sample data will contain a list of `locations`, marked by their latitude and longitude. Each `location` also has a list of `samples` containing the date when the measurement was taken and the amount of rainfall in cm. Below is an example.

```
{
  "locations" : [
    {
      "latitude" : "40.71",
      "longitude" : "-74.01",
      "samples" : [
          {
             "date" : "2014-06-07",
             "rainfall" : "2"
          },
          {
             "date" : "2014-08-12",
             "rainfall" : "0.5"
          },
          {
             "date" : "2014-09-29",
             "rainfall" : "1.25"
          }
       ]
    },
    {
      "latitude" : "42.35",
      "longitude" : "-71.06",
      "samples" : [
          {
             "date" : "2014-03-03",
             "rainfall" : "1.75"
          },
          {
             "date" : "2014-05-16",
             "rainfall" : "0.25"
          },
          {
             "date" : "2014-03-18",
             "rainfall" : "2.25"
          }
       ]
    }
  ]
}
```

All of the code above, and in the rest of this chapter is available in full in the `nodecpp-demo` repository at https://github.com/freezer333/nodecpp-demo, under the "Rainfall" section.

The JavaScript code will call a C++ addon to calculate average and median

rainfall for each location. Initially, we'll just deal with one location at a time, but we'll gradually extend the example to cover lists (arrays) and the return of more complex data structures.

Please note, average/median is not exactly a "heavy compute" task - this is just for demonstration purposes. The reader should always keep in mind that there should be a clear reason for developing a C++ addon rather than just staying in JavaScript. Dropping into C++ for relatively short calculations (such as this) ***is not*** *going to yield performance over JavaScript because of the cost of integration. Addons make the most sense when (1) there are very heavy computation tasks being performed (minutes, not seconds) or (2) when an existing implementation in C/C++ already exists and must be leveraged.*

Organizing the source code

All examples in this book will follow a straight forward file/folder layout - at the top level of the addon directory we'll place all JavaScript code. A subdirectory titles `cpp` will contain our C++ addon code along with our `binding.gyp` file that controls the build process. The build process will create several artifacts (platform specific), but the resulting addon will always be placed in a child directory `Debug` or `Release`.

```
- /addon_folder
  - all JavaScript source code files
  - /cpp
    - binding.js
    - all c++ source code files
    - /build
      - build artifacts
      - /Debug
      - /Release
```

JavaScript code running at the top level will be able to include the created addon by issuing a `require(/cpp/build/Release/addon_name)`, where `addon_name` will be `rainfall` for our examples in this chapter. While this folder structure is sufficient for examples, note that later in the book (Chapter 8) we'll examine how to organize addons so they can easily be redistributed through `npm`.

Rainfall C++ data models

We're going to start addon development from a *bottom up* viewpoint - building out the C++ logic first. At the core of the C++ logic is two classes that model the domain. The first, `sample`, will represent a distinct rainfall data sample, consisting of the date and the amount of rainfall. For now, we'll model `date`

with a simple string, but we could improve this using V8's actual `Data` object as well. The second class represents a given location. `location` contains doubles describing latitude and longitude along with a list of `sample` objects. The code below would be placed in `/cpp/rainfall.h`.

```
class sample {
public:
  sample (); // in rainfall.cc
  sample (string d, double r) ; // in rainfall.cc
  string date;
  double rainfall;
};

class location {
public:
  double longitude;
  double latitude;
  vector<sample> samples;
};

// Will return the average (arithmetic mean) rainfall for the give location
double avg_rainfall(location & loc); // code in rainfall.cc
```

Note that the code implementing these function and method prototypes will be found in `rainfall.cc`. This file will be 100% standard C++ - neither `rainfall.h` and `rainfall.cc` will contain any V8/Node.js API dependencies at all. We'll see the implementation shortly.

Thinking about JavaScript

Given some logic, we must start to think about what will be called accessed from JavaScript. The core logic that we created above is the `avg_rainfall`, which would return the average rainfall amounts across all samples found in a location. Note however, this function *cannot* be called directly from JavaScript, because it's input is a native C++ object. Our addon code will need to expose a *similar* function to JavaScript, and the use the provided data (likely a JavaScript object representing a location/sample set) to build a true `location` object. We'd imagine our JavaScript code could look something like this:

```
var location = {
    latitude : 40.71, longitude : -74.01,
       samples : [
          { date : "2016-06-07", rainfall : 2 },
          { date : "2016-08-12", rainfall : 0.5}
```

```
        ] };

var average = addon.avg_rainfall(location);
```

Creating the Rainfall Addon

As described in Chapter 1, creating the addon requires some standard boilerplate code that enables Node.js to load our module and begin to call functions within it. Our V8 API code will all be placed in a separate C++ file called `rainfall_node.cc`. This file includes `node.h` and `v8.h` headers. Next, it defines an entry point for our addon - achieved by creating a function and registering it via a macro provided by the node/v8 headers.

```
#include <node.h>
#include <v8.h>
#include "rainfall.h"

using namespace v8;

void init(Handle <Object> exports, Handle<Object> module) {
 // we'll register our functions here..
}

// associates the module name with initialization logic
NODE_MODULE(rainfall, init)
```

As described in Chapter 1, inside the `init` function (we can name it anything, as long as we associate it in the NODE_MODULE macro) we will define which functions are going to be exposed to Node.js when are module is included/required. The wrapper code to do all this gets a little ugly, which is why its great to keep your clean C++ code (the rainfall.h/cc files) separate from all this.

So the first thing we'll do is expose the `avg_rainfall` method from rainfall by creating a new function in `rainfall_node.cc`.

```
void AvgRainfall(
    const v8::FunctionCallbackInfo<v8::Value>& args){

  Isolate* isolate = args.GetIsolate();

  Local<Number> retval = v8::Number::New(isolate, 0);
  args.GetReturnValue().Set(retval)
}
```

The return value is set at the last line of the function. As currently written, the function always just returns 0 as the average rainfall - we'll fix that soon...

Now lets make this function callable from node, by registering it within the `init` function from earlier.

```
void init(Handle <Object> exports, Handle<Object> module) {
  NODE_SET_METHOD(exports, "avg_rainfall", AvgRainfall);
}
```

The init function is called when the module is first loaded in a node application; it is given an export and module object representing the module being constructed and the object that is returned after the `require` call in JavaScript. The `NODE_SET_METHOD` call is adding a method called `avg_rainfall` to the exports, associated with our actual `AvgRainfall` function from above. From JavaScript, we'll see a function called "avg_rainfall", which at this point just returns 0.

We can now build the addon and try calling the `avg_rainfall`, even though it will just return 0.

Building the C++ addon

Our `binding.gyp` file will be placed in the same directory as the C++ code we already have. In this file, we must specify the name of our addon (`rainfall`), the source code that must be compiled (we can omit headers), and optional build flags and settings. The target name is the addon/module name - **it must match the name you gave in NODE_MODULE macro in the `rainfall_node.cc` file!**.

```
{
  "targets": [
    {
      "target_name": "rainfall",
      "sources": [ "rainfall.cc" , "rainfall_node.cc" ],
      "cflags": ["-Wall", "-std=c++11"],
      "conditions": [
        [ 'OS=="mac"', {
            "xcode_settings": {
                'OTHER_CPLUSPLUSFLAGS' :
                    ['-std=c++11','-stdlib=libc++'],
                'OTHER_LDFLAGS': ['-stdlib=libc++'],
                'MACOSX_DEPLOYMENT_TARGET': '10.7' }
            }
        ]
      ]
```

```
    }
  ]
}
```

You'll notice this binding file is more complex compared to the one presented in Chapter 1. For one, we've included some cflags - which are compiler options that will be passed into whatever compiler is being used during the build. The flags presented force the compiler to report all warnings and to use the C++11 standard. These flags are generic, they work just fine if you are using Linux and clang++. If built on Windows, `node-gyp` will use Visual Studio, which does not use the `-Wall` and `-std` flags - although it automatically uses the C++11 standard. The oddball is the Mac OS X environment - it doesn't play quite as nicely. The "conditions" group in the binding file above allows us to specify some additional, conditional, flags. In this case, when the compiler is running on Mac OS X, we are specifying build flags specifically designed for Xcode. These are unfortunately required (at least at this time) in order to utilize C++11 and above.

With this is place, you can build your module by moving your terminal/command prompt into the `/cpp` directory and invoking `node-gyp`:

```
> node-gyp configure build
```

If all goes well here you will have a `/build/Release` folder created right alongside your C++ code files. Within that folder, there should be a `rainfall.node` output file. **This is your addon**... ready to be required from node.

At this point, it makes sense to ensure the addon is working as expected. Create a new file in the addon's root directory (a sibling of `/cpp`) called `rainfall.js`. Let's fill in some details not initially present when we first looked at our calling JavaScript code:

```
var addon = require("./cpp/build/Release/rainfall");

var location = {
    latitude : 40.71, longitude : -74.01,
       samples : [
          { date : "2016-06-07", rainfall : 2 },
          { date : "2016-08-12", rainfall : 0.5}
       ] };

var average = addon.avg_rainfall(location);

// average will be 0 at this point,
// since the actual C++ code isn't complete
```

Mapping JavaScript objects to C++ classes

With the addon established, we can connect the `AvgRainfall` C++ addon function to the actual C++ logic defined in `rainfall.h`. To do this, we need some additional code to extract the object properties and instantiate C++ objects. We'll pack this transfer code into a separate function called within the newly revised `AvgRainfall` function:

```
void AvgRainfall(
  const v8::FunctionCallbackInfo<v8::Value>& args) {

  Isolate* isolate = args.GetIsolate();

  location loc = unpack_location(isolate, args);
  double avg = avg_rainfall(loc);

  Local<Number> retval = v8::Number::New(isolate, avg);
  args.GetReturnValue().Set(retval);
}
```

The `unpack_location` function accepts the VM instance and the argument list, and unpacks the V8 object into a new `location` object - and returns it.

```
location unpack_location(Isolate * isolate,
                const v8::FunctionCallbackInfo<v8::Value>& args) {
  location loc;
  Handle<Object> location_obj = Handle<Object>::Cast(args[0]);
  Handle<Value> lat_Value =
                location_obj->Get(
                  String::NewFromUtf8(isolate,"latitude"));
  Handle<Value> lon_Value =
                location_obj->Get(
                  String::NewFromUtf8(isolate,"longitude"));

  loc.latitude = lat_Value->NumberValue();
  loc.longitude = lon_Value->NumberValue();

  Handle<Array> array =
       Handle<Array>::Cast(location_obj->Get(
                String::NewFromUtf8(isolate,"samples")));

  int sample_count = array->Length();
  for ( int i = 0; i < sample_count; i++ ) {
    sample s = unpack_sample(
      isolate, Handle<Object>::Cast(array->Get(i)));
```

```
    loc.samples.push_back(s);
  }
  return loc;
}
```

The `unpack_sample` function is similar - this is all a matter of unpacking the data from V8's data types.

```
sample unpack_sample(Isolate * isolate,
  const Handle<Object> sample_obj) {

  sample s;
  Handle<Value> date_Value = sample_obj->Get(
    String::NewFromUtf8(isolate, "date"));
  Handle<Value> rainfall_Value = sample_obj->Get(
    String::NewFromUtf8(isolate, "rainfall"));

  v8::String::Utf8Value utfValue(date_Value);
  s.date = std::string(*utfValue);

  // Unpack numeric rainfall amount from V8 value
  s.rainfall = rainfall_Value->NumberValue();
  return s;
}
```

Completed Node.js File - Average Rainfall

Below is a slightly modified JavaScript listing from above. Executing the file with `node`, we'll see the actual average, as computed in C++, is returned.

```
var addon = require("./cpp/build/Release/rainfall");

var location = {
    latitude : 40.71, longitude : -74.01,
       samples : [
          { date : "2016-06-07", rainfall : 2 },
          { date : "2016-08-12", rainfall : 0.5}
       ] };

var average = addon.avg_rainfall(location);

console.log("Average rain fall = " + average + "cm");
```

You should be able to run it - and see that your C++ module has been called!

```
> node rainfall.js
Average rain fall = 1.25cm
```

Returning Objects - a collection of statistics

We now have a fully functional node application calling C++. We've successfully transformed a single JavaScript object into a C++ object. Now let's return more than just an average - let's return an object containing several statistics describing the sample we get from JavaScript.

```
class rain_result {
   public:
       float median;
       float mean;
       float standard_deviation;
       int n;
};
```

Just like before, we'll keep the "business" part of the C++ code completely separate from the code dealing with V8 integration. The class above has been added to the `rainfall.h / rainfall.cc` files.

Now we're going to create a new callable function for the Node addon. So, in the rainfall_node.cc file (where we put all our V8 integration logic), Lets add a new function and register it with the module's exports.

```
void RainfallData(
  const v8::FunctionCallbackInfo<v8::Value>& args) {

  Isolate* isolate = args.GetIsolate();

  location loc = unpack_location(isolate, args);
  rain_result result = calc_rain_stats(loc);

/*
 .... return the result object back to JavaScript  ....
*/
}
```

The `unpack_location` function is just being reused, it's where we extract the location (and rainfall samples) from the JavaScript arguments. We now also have a new function, defined in `rainfall.h / rainfall.cc`, called `calc_rain_stats` which returns a `rain_result` instance based on the `location` instance it is given. It computes mean/median/standard deviation.

```
rain_result calc_rain_stats(location &loc) {
  rain_result result;
  double ss = 0;
  double total = 0;

  result.n = loc.samples.size();

  for (const auto &sample : loc.samples) {
     total += sample.rainfall;
  }
  result.mean = total / loc.samples.size();

  for (const auto &sample : loc.samples) {
     ss += pow(sample.rainfall - result.mean, 2);
  }
  result.standard_deviation = sqrt(ss/(result.n-1));

  std::sort(loc.samples.begin(), loc.samples.end());
  if (result.n %2 == 0) {
    result.median = (loc.samples[result.n / 2 - 1].rainfall +
                     loc.samples[result.n / 2].rainfall) / 2;
  }
  else {
    result.median = loc.samples[result.n / 2].rainfall;
  }
  return result;
}
```

The `RainfallData` function is exported by adding another call to `NODE_SET_METHOD` inside the `init` function in `rainfall_node.cc`.

```
void init(Handle <Object> exports, Handle<Object> module) {
  NODE_SET_METHOD(exports, "avg_rainfall", AvgRainfall);
  NODE_SET_METHOD(exports, "data_rainfall", RainfallData);
}
```

Building the JavaScript object and returning it

After unpacking the `location` object inside the `RainfallData` function, we get a `rainfall_result` object:

```
rain_result result = calc_rain_stats(loc);
```

Now its time to return that - and to do so we'll create a new V8 object, transfer the rain_result data into it, and return it back to JavaScript. The API here is

all introduced in Chapter 2 - we're just applying what we learned to build up the objects now.

```
void RainfallData(
  const v8::FunctionCallbackInfo<v8::Value>& args) {

  Isolate* isolate = args.GetIsolate();

  location loc = unpack_location(isolate, args);
  rain_result result = calc_rain_stats(loc);

  // Creates a new Object on the V8 heap
  Local<Object> obj = Object::New(isolate);

  // Transfers the data from result, to obj (see below)
  obj->Set(
    String::NewFromUtf8(isolate, "mean"),
    Number::New(isolate, result.mean));
  obj->Set(
    String::NewFromUtf8(isolate, "median"),
    Number::New(isolate, result.median));
  obj->Set(
    String::NewFromUtf8(isolate, "standard_deviation"),
    Number::New(isolate, result.standard_deviation));
  obj->Set(
    String::NewFromUtf8(isolate, "n"),
    Integer::New(isolate, result.n));

  // Return the object
  args.GetReturnValue().Set(obj);
}
```

Notice the similarities between this function and the AvgRainfall function from earlier. They both follow the similar pattern of creating a new variable on the V8 heap and returning it by setting the return value associated with the `args` variable passed into the function. The difference now is that actually setting the value of the variable being returned is more complicated. In AvgRainfall, we just created a new `Number`:

```
Local<Number> retval = v8::Number::New(isolate, avg);
```

Now, we have we instead move the data over one property at time:

```
Local<Object> obj = Object::New(isolate);
obj->Set(
```

```
  String::NewFromUtf8(isolate, "mean"),
  Number::New(isolate, result.mean));

obj->Set(
  String::NewFromUtf8(isolate, "median"),
  Number::New(isolate, result.median));

obj->Set(
  String::NewFromUtf8(isolate, "standard_deviation"),
  Number::New(isolate, result.standard_deviation));

obj->Set(
  String::NewFromUtf8(isolate, "n"),
  Integer::New(isolate, result.n));
```

While its a bit more code - the object is just being built up with a series of named properties - its pretty straightforward.

Calling from JavaScript

Now that we've completed the C++ side, we need to rebuild our addon:

```
> node-gyp configure build
```

In JavaScript, we can now call both methods, and we'll see the object returned by our new data_rainfall method returns a real JavaScript object.

```
//rainfall.js
var rainfall = require("./cpp/build/Release/rainfall");
var location = {
    latitude : 40.71, longitude : -74.01,
       samples : [
          { date : "2015-06-07", rainfall : 2.1 },
          { date : "2015-06-14", rainfall : 0.5},
          { date : "2015-06-21", rainfall : 1.5},
          { date : "2015-06-28", rainfall : 1.3},
          { date : "2015-07-05", rainfall : 0.9}
       ] };

var avg = rainfall.avg_rainfall(location)
console.log("Average rain fall = " + avg + "cm");

var data = rainfall.data_rainfall(location);
```

```
console.log("Mean = " + data.mean)
console.log("Median = " + data.median);
console.log("Standard Deviation = " + data.standard_deviation);
console.log("N = " + data.n);
```

```
> node rainfall.js
Average rain fall = 1.26cm
Mean = 1.2599999904632568
Median = 1.2999999523162842
Standard Deviation = 0.6066300272941589
N = 5
```

Receiving and Returning Arrays of Rainfall Data

Now that we have the ability to move individual samples and individual statistics between JavaScript and C++, extending this to support arrays of data is actually pretty simple - again using the principles introduced in Chapter 2.

Registering the callable addon function

As always, we start by writing a C++ function in `/cpp/rainfall_node.cc` that will be callable from Node.js.

```
void CalculateResults(
  const v8::FunctionCallbackInfo<v8::Value>&args) {

    Isolate* isolate = args.GetIsolate();
    std::vector<location> locations;  // we'll get this from Node.js
    std::vector<rain_result> results; // we'll build this in C++

    // we'll populate this with the results
    Local<Array> result_list = Array::New(isolate);

    // ... and send it back to Node.js as the return value
    args.GetReturnValue().Set(result_list);
}
....
void init(Handle <Object> exports, Handle<Object> module) {

  NODE_SET_METHOD(exports, "avg_rainfall", AvgRainfall);
  NODE_SET_METHOD(exports, "data_rainfall", RainfallData);
  NODE_SET_METHOD(exports, "calculate_results", CalculateResults);
}
```

The `CalculateResults` function will extract a list of location objects from the parameters (`args`) and eventually return a fully populated array of results. We make it callable by calling the `NODE_SET_METHOD` in the `init` function - so we can call `calculate_results` in JavaScript.

Before we implement the C++, lets look at how this will all be called in JavaScript. First step is to rebuild the addon from the `cpp` directory:

```
>  node-gyp configure build
```

In the rainfall.js, we'll construct an array of locations and invoke our addon:

```
// Require the Addon
var rainfall = require("./cpp/build/Release/rainfall");

var makeup = function(max) {
    return Math.round(max * Math.random() * 100)/100;
}

// Build some dummy locations
var locations = []
for (var i = 0; i < 10; i++ ) {
    var loc = {
        latitude: makeup(180),
        longitude: makeup(180),
        samples : [
            {date: "2015-07-20", rainfall: makeup(3)},
            {date: "2015-07-21", rainfall: makeup(3)},
            {date: "2015-07-22", rainfall: makeup(3)},
            {date: "2015-07-23", rainfall: makeup(3)}
        ]
    }
    locations.push(loc);
}

// Invoke the Addon
var results = rainfall.calculate_results(locations);

// Report the results from C++
var i = 0;
results.forEach(function(result){
    console.log("Result for Location " + i);
    console.log("--------------------------");
    console.log("\tLatitude:         "
        + locations[i].latitude.toFixed(2));
    console.log("\tLongitude:        "
```

```
        + locations[i].longitude.toFixed(2));
    console.log("\tMean Rainfall:    "
        + result.mean.toFixed(2) + "cm");
    console.log("\tMedian Rainfall:  "
        + result.median.toFixed(2) + "cm");
    console.log("\tStandard Dev.:    "
        + result.standard_deviation.toFixed(2) + "cm");
    console.log("\tNumber Samples:   "
        + result.n);
    console.log();
    i++;
})
```

When you run this with `node rainfall` you'll get no output, only because the C++ function is returning an empty array at this point. Try putting a `console.log(results)` in, you should see `[]` print out.

Extracting the Array in C++

Now lets skip back to our `CalculateResults` C++ function. We've been given the function callback arguments object, and our first step is to cast it to a V8 array.

```
void CalculateResults(
  const v8::FunctionCallbackInfo<v8::Value>&args) {

    Isolate* isolate = args.GetIsolate();
    ... (see above)...
    Local<Array> input = Local<Array>::Cast(args[0]);
    unsigned int num_locations = input->Length();
```

With the V8 array `input`, we'll now loop through and actually create a POCO `location` object using the `unpack_location` function we've been using throughout this chapter. The return value from `unpack_location` is pushed onto a standard C++ vector.

```
for (unsigned int i = 0; i < num_locations; i++) {
  locations.push_back(
       unpack_location(isolate, Local<Object>::Cast(input->Get(i)))
  );
}
```

Of course, now that we have a standard vector of `location` objects, we can call our existing `calc_rain_stats` function on each one and build up a vector of `rain_result` objects.

```
results.resize(locations.size());
std::transform(
     locations.begin(),
     locations.end(),
     results.begin(),
     calc_rain_stats);
```

Building an Array to return back from C++

Our next step is to move the data we've created into the V8 objects that we'll return. First, we create a new V8 Array:

```
Local<Array> result_list = Array::New(isolate);
```

We can now iterate through our `rain_result` vector and use the `pack_rain_result` function to create a new V8 object and add it to the `result_list` array.

```
for (unsigned int i = 0; i < results.size(); i++ ) {
      Local<Object> result = Object::New(isolate);
      pack_rain_result(isolate, result, results[i]);
      result_list->Set(i, result);
    }
```

And... we're all set. Here's the complete code for the `CalculateResult` function:

```
void CalculateResults(
  const v8::FunctionCallbackInfo<v8::Value>&args) {

    Isolate* isolate = args.GetIsolate();
    std::vector<location> locations;
    std::vector<rain_result> results;

    // extract each location (its a list)
    Local<Array> input = Local<Array>::Cast(args[0]);
    unsigned int num_locations = input->Length();
    for (unsigned int i = 0; i < num_locations; i++) {
      locations.push_back(
             unpack_location(isolate,
              Local<Object>::Cast(input->Get(i))));
    }

    // Build vector of rain_results
```

```
    results.resize(locations.size());
    std::transform(
          locations.begin(),
          locations.end(),
          results.begin(),
          calc_rain_stats);


    // Convert the rain_results into Objects for return
    Local<Array> result_list = Array::New(isolate);
    for (unsigned int i = 0; i < results.size(); i++ ) {
      Local<Object> result = Object::New(isolate);
      pack_rain_result(isolate, result, results[i]);
      result_list->Set(i, result);
    }

    // Return the list
    args.GetReturnValue().Set(result_list);
}
```

Do another `node-gyp configure build` and re-run `node rainfall.js` and you'll see the fully populated output results from C++.

```
Result for Location 0
-------------------------
    Latitude:         145.45
    Longitude:        7.46
    Mean Rainfall:    1.59cm
    Median Rainfall:  1.65cm
    Standard Dev.:    0.64cm
    Number Samples:   4

Result for Location 1
-------------------------
    Latitude:         25.32
    Longitude:        98.64
    Mean Rainfall:    1.17cm
    Median Rainfall:  1.24cm
    Standard Dev.:    0.62cm
    Number Samples:   4
....
```

About efficiency

You might be wondering, aren't we wasting a lot of memory by creating POCO copies of all the V8 data? Its a good point, for all the data being passed into the C++ Addon, the V8 objects (which take up memory) are being moved into new C++ objects. Those C++ objects (and their derivatives) are then copied into new V8 objects to be returned... we're doubling memory consumption and its also costing us processing time to do all this!

For most use cases you'll end up working with, the overhead of memory copying (both time and space) is dwarfed by the actual execution time of the algorithm and processing that I'm doing in C++. If you are going through the trouble of calling C++ from Node, its because the actual compute task is *significant*!

For situations where the cost of copying input/output isn't dwarfed by your actual processing time, it would probably make more sense to use V8 object wrapping API instead (Chapter 5), or to use Buffers (Appendix B) to directly allocate data *outside* V8 in the first place.

Chapter 4 - Asynchronous Addons

In Chapter 3 we built up a real-world example of a functioning C++ addon - however there is something a bit unsatisfying about it. If you've programmed with Node.js (or even client-side JavaScript), you know that often functions don't *return* results - they call a *callback* when they have completed, passing the results in as parameters.

Instead of using Chapter 3's rainfall addon like this:

```
var results = rainfall.calculate_results(locations);

results.forEach(function(result){
    // .. print the result
});
```

... the experienced JavaScript developer might instead expect to use it like this:

```
rainfall.calculate_results(locations, function(results){
  results.forEach(function(result){
    // .. print the result
  });
});
```

Why muddy things up with callbacks? The answer is both simple and complicated. The simple answer is that they allow us to implement a function *asynchronously*. Behind the scenes, what this really means is that functions that accept callbacks often create worker threads to do their computation. Once the worker threads are started, the function returns and allows the calling code to go about it's business (executing the next line of code). When the worker threads complete, the callback is invoked. It's a complex work flow that often confuses new developers and has given rise to all sorts of alternative ways of working with callbacks - such as promises.

Why are these functions asynchronous? Client side, within the web browser, there is clear need to avoid having JavaScript tying up a lot of time (either waiting for I/O or using the CPU), since while JavaScript is executing the browser cannot render or react to user input. In Node.js, the same philosophy holds - the event loop (implemented by a library called `libuv`) should never stall on I/O or heavy CPU because, in theory, there is likely other work to be done instead. For example, if using Node.js to implement a web server, tying up the event loop would preclude the web server from serving additional incoming web requests.

Addons are no different in this regard - and in fact they are almost always strong candidates for being implemented asynchronously. Remember, one of the

reasons you typically use a C++ addon is to perform a long running CPU task - where C++ will outperform JavaScript. By definition, these long CPU tasks are exactly the sort of thing that should be executed in a background thread!

We are going to "creep up" on the asynchronous model by first learning about the V8 Function API and seeing how we can modify our rainfall addon to use callback functions (synchronously) first. We'll then move to the asynchronous model, while investigating how V8 deals with worker threads and memory management in more detail.

All of the code for this chapter is available in full in the `nodecpp-demo` repository at https://github.com/freezer333/nodecpp-demo, under the "Rainfall" section.

V8 Function API

Chapter 2 described how to use the most common JavaScript data types through the V8 API. We covered primitives and objects in detail - but we didn't go much beyond standard objects and arrays. In JavaScript, functions are objects too - and V8 has a specific `Function` type that let's us access (and call) JavaScript functions from C++.

The API for calling JavaScript functions from addons was outlined in Chapter 1, and in fact there really isn't much too it! Lets review it quickly, starting with a simple addon function that accepts two arguments - a function from JS and an argument to invoke that function (callback) with:

```
void CallThisWithThis(
  const FunctionCallbackInfo<Value>& args) {

  Isolate* isolate = args.GetIsolate();
  Local<Function> cb = Local<Function>::Cast(args[0]);

  // Create an array with only the message passed in
  Local<Value> argv[1] = {args[1]};
  cb->Call(Null(isolate), 1, argv);
}

void Init(Local<Object> exports, Local<Object> module) {
  NODE_SET_METHOD(exports,
      "callthis_withthis", CallThisWithThis);
}
```

As you can see, a `Local<Function>` object can be cast from arguments just like any of the other data types we've seen. To execute a function we can utilize the `Call` method. It requires three arguments - the first is the "this" object that the JavaScript function will be called with. Specifying null tells V8 to use the default

value (depends on invocation context). The second and third parameters are for arguments the function will be called with - in this case just one argument, which is what was passed into the addon itself.

The JavaScript to use the addon is pretty straightforward:

```
const callback = require('./build/Release/callback');

var callme = function(message) {
    console.log(message);
}
back.callthis_withthis(callme, "This is an important message");
```

Synchronous addons with a Callback

Now let's adopt this style in our rainfall addon. We'll call this function `CalculateResultsSync` because while it uses callbacks, it is very much *synchronous*.

```
// in node_rainfall.cc
void CalculateResultsSync(
  const v8::FunctionCallbackInfo<v8::Value>&args) {

    Isolate* isolate = args.GetIsolate();
    std::vector<location> locations;
    std::vector<rain_result> results;

    // extract each location (its a list)
    Local<Array> input = Local<Array>::Cast(args[0]);
    unsigned int num_locations = input->Length();
    for (unsigned int i = 0; i < num_locations; i++) {
      locations.push_back(unpack_location(isolate,
          Local<Object>::Cast(input->Get(i))));
    }

    // Build vector of rain_results
    results.resize(locations.size());
    std::transform(locations.begin(),
      locations.end(),
      results.begin(),
      calc_rain_stats);

    // Convert the rain_results into Objects for return
    Local<Array> result_list = Array::New(isolate);
    for (unsigned int i = 0; i < results.size(); i++ ) {
```

```
27        Local<Object> result = Object::New(isolate);
28        pack_rain_result(isolate, result, results[i]);
29        result_list->Set(i, result);
30      }
31
32      Local<Function> callback = Local<Function>::Cast(args[1]);
33      Handle<Value> argv[] = { result_list };
34      callback->Call(
35        isolate->GetCurrentContext()->Global(), 1, argv);
36
37      std::cerr << "Returning from C++ now" << std::endl;
38
39      args.GetReturnValue().Set(Undefined(isolate));
40  }
41
42  ...
43
44  void init(Handle <Object> exports, Handle<Object> module) {
45    ...
46     NODE_SET_METHOD(exports,
47         "calculate_results_async", CalculateResultsAsync);
48  }
```

What do we mean by "synchronous"? On line 31 we invoke the callback sent in from JavaScript.

```
var print_rain_results = function(results) {
  results.forEach(function(result, i){
      console.log("Result for Location " + i);
      console.log("-------------------------");
      console.log("\tLatitude:          "
          + locations[i].latitude.toFixed(2));
      console.log("\tLongitude:         "
          + locations[i].longitude.toFixed(2));
      console.log("\tMean Rainfall:     "
          + result.mean.toFixed(2) + "cm");
      console.log("\tMedian Rainfall:   "
          + result.median.toFixed(2) + "cm");
      console.log("\tStandard Dev.:     "
          + result.standard_deviation.toFixed(2) + "cm");
      console.log("\tNumber Samples:    "
          + result.n + "\n");
  });
}
```

```
// Execute the synchronous callback.
rainfall.calculate_results_sync(locations, print_rain_results);

console.log("JavaScript program has completed.")
```

Notice the `cerr` output afterwards - when will it execute? It executes AFTER the JavaScript function we are calling completes. The steps are like so:

1. JavaScript calls our addon (`calculate_results_sync`)
2. C++ code unpacks samples, calculates statistics
3. C++ invokes `print_rain_results`
4. JavaScript prints the rains results to the screen
5. Control returns to C++ addon, which prints out "Returning from C++ now"
6. JavaScript prints out "JavaScript program has completed".

Everything is happening in lock step - while the C++ is executing nothing is happening in JavaScript unless explicitly called *from C++*. We'd prefer to see something like this:

1. JavaScript calls our addon (`calculate_results_sync`)
2. C++ code unpacks samples, starts thread, and returns.
3. JavaScript executes next line(s) (prints out "JavaScript program has completed")

. . . after some time..

1. C++ finishes the calculations and invokes `print_rain_results`
2. JavaScript prints the rains results to the screen

Moving to Asynchronous with Worker Threads

Lets do a quick overview of how worker threads work in V8. In our model, there are **two threads**. The first thread is the *event loop thread* - its the thread that our JavaScript code is executing in, and its the thread we are **still in** when we cross over into the C++ addon. This is the thread that we *don't* want to stall by doing heavy calculations! The second thread (to be created) will be a worker thread managed by libuv, the library that supports asynchronous I/O in Node.

Hopefully you're pretty familiar with threads - the key point here is that each thread has it's own stack - you can't share stack variables between the event loop thread and the worker thread! Threads do share the same heap though - so that's where we are going to put our input and output data, along with state information.

Asynchronous Memory Headaches

Worker threads are a extremely useful concept, but it comes at a price. JavaScript is implicitly single threaded, and V8 is built around the notion that data within JavaScript is strictly accessible by one thread at a time (even though technically, they are on the heap!). There is sort of a "golden rule" in asynchronous addon development:

> *you can't access V8 memory outside the event-loop's thread.*

This essentially means if you want the *asynchronous* part of your addon to be able to (1) access input data sent from JavaScript and/or (2) return data to JavaScript then you need to create copies of the input/output data. You might notice however that this is *exactly* what we are already doing with our unpacking/packing strategy. At the end of this chapter, we'll dive a bit deeper into why copying data is nearly unavoidable when dealing with asynchronous execution.

The C++ addon code

Our first step is to create yet another C++ function, and register it with our module.

```
// in node_rainfall.cc
void CalculateResultsAsync(const v8::FunctionCallbackInfo<v8::Value>&args) {
    Isolate* isolate = args.GetIsolate();

     // we'll start a worker thread to do the job
     // and call the callback here...

    args.GetReturnValue().Set(Undefined(isolate));
}
...
void init(Handle <Object> exports, Handle<Object> module) {
  ...
   NODE_SET_METHOD(exports, "calculate_results_async", CalculateResultsAsync);
}
```

The `CalculateResultsAsync` function is where we'll end up kicking off a worker thread using libuv - but notice what it does right away: it *returns*! Nothing we fill into this function will be long running, all the real work will be done in the worker thread.

On the C++ side of things, we're going to utilize **three functions** and a **struct** to coordinate everything:

1. `Worker Data` (struct) - will store plain old C++ input (locations) and output (rain_results) and the callback function that can be invoked when work is complete
2. `CalculateResultsAsync` - executes in event-loop thread, extracts input and stores it on the heap in *worker data*.
3. `WorkAsync` - the function that the worker thread will execute. We'll launch this thread from `CalculateResultsAsync` using the libuv API
4. `WorkAsyncComplete` - the function that libuv will invoke when the worker thread is finished. This function is executed on the *event loop thread*, **not** the worker thread. Figure 5 outlines the workflow we'll use.

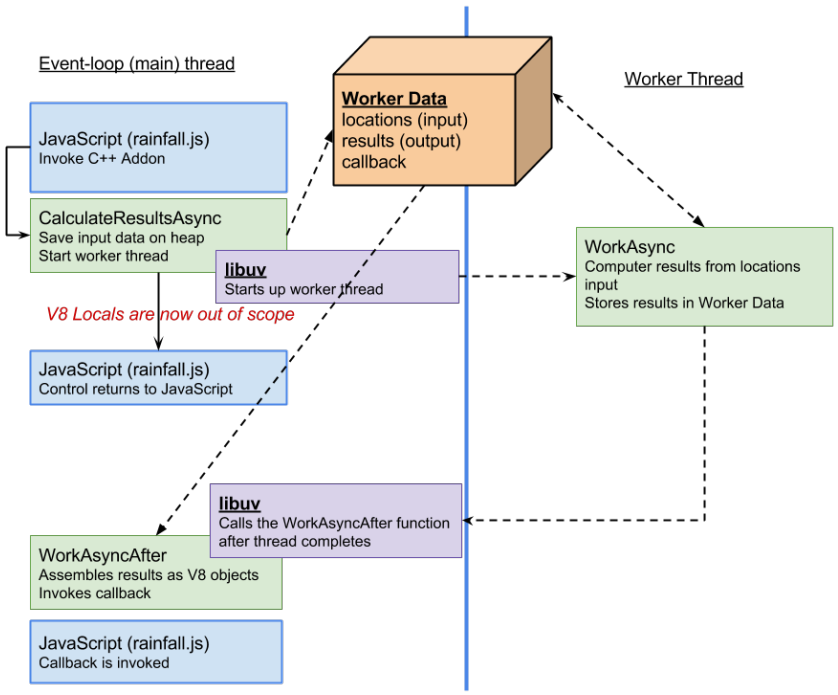


Figure 5: Node.js asynchronous model.

Lets look at the C++ code, starting with our Work data structure:

```
struct Work {
  uv_work_t  request;
  Persistent<Function> callback;

  std::vector<location> locations;
  std::vector<rain_result> results;
};
```

The vector will store our input and output. The request object is a handle that will actually loop back to the work object - the libuv API accepts pointers of type `uv_work_t` when starting worker threads. The `callback` variable is going to store the JavaScript callback. Importantly, its `Persistent<>`, meaning it will be kept in scope by V8 and not garbage collected. This seems confusing, since the callback will be executed in the event-loop thread, but we need to make it `Persistent` because when we initially return to JavaScript, all V8 locals are destroyed. A new Local context is created when we are about to call the JavaScript callback after the worker thread completes.

Now lets look at the `CalculateResultsAsync` function

```
void CalculateResultsAsync(
  const v8::FunctionCallbackInfo<v8::Value>&args) {

    Isolate* isolate = args.GetIsolate();

    Work * work = new Work();
    work->request.data = work;
```

Notice that the Work struct is created on the heap. Remember, local variables (and V8 Local objects) will be destroyed when this function returns - even though our worker thread will still be active. Here we also set the `uv_work_t` data pointer to point right back to the `work` struct so libuv will pass it back to us on the other side.

```
    ...
    // extract each location (its a list) and
    // store it in the work package
    // work (and thus, locations) is on the heap,
   // accessible in the libuv threads
    Local<Array> input = Local<Array>::Cast(args[0]);
    unsigned int num_locations = input->Length();
    for (unsigned int i = 0; i < num_locations; i++) {
      work->locations.push_back(
          unpack_location(isolate,
             Local<Object>::Cast(input->Get(i)))
      );
    }
```

Where earlier we now just went ahead and processed the rainfall data, now we'll kick off a worker thread using libuv. First we store the callback sent to use from JavaScript, and then we're off. Notice as soon as we call `uv_queue_work`, we return - the worker is executing in its own thread (`uv_queue_work` returns immediately).

```
    // store the callback from JS in the work
    // package so we can invoke it later
    Local<Function> callback =
        Local<Function>::Cast(args[1]);
    work->callback.Reset(isolate, callback);

    // kick of the worker thread
    uv_queue_work(uv_default_loop(),&work->request,
        WorkAsync,WorkAsyncComplete);

    args.GetReturnValue().Set(Undefined(isolate));

}
```

Notice the arguments to `uv_queue_work` - its the work->request we setup at the top of the function, and two functions we haven't seen yet - the function to start the thread in (`WorkAsync`) and the function to call when it's complete (`WorkAsyncComplete`).

At this point, control is passed back to Node (JavaScript). If we had further JavaScript to execute, it would execute now. Basically, from the JavaScript side, our addon is acting the same as any other asynchronous call we typically make (like reading from files).

The worker thread

The worker thread code is actually really simple. We just need to process the data - and since its already extracted out of the V8 objects, its pretty vanilla C++ code. Notice our function has been called with the libuv work request parameter. We set this up above to point to our actual work data.

```
static void WorkAsync(uv_work_t *req)
{
    Work *work = static_cast<Work *>(req->data);

    // this is the worker thread, lets build up the results
    // allocated results from the heap because we'll need
    // to access in the event loop later to send back
    work->results.resize(work->locations.size());
    std::transform(work->locations.begin(),
             work->locations.end(),
             work->results.begin(),
             calc_rain_stats);
```

```
    // that wasn't really that long of an operation,
    // so lets pretend it took longer...
    std::this_thread::sleep_for(chrono::seconds(3));
}
```

Note - the code above also sleeps for extra effect, since the rainfall data isn't really that large.

When the worker completes...

Once the worker thread completes, libuv handles calling our `WorkAsyncComplete` function - passing in the work request object again - so we can use it!

```
// called by libuv in event loop when async function completes
static void WorkAsyncComplete(uv_work_t *req,int status)
{
    Isolate * isolate = Isolate::GetCurrent();
    v8::HandleScope handleScope(isolate);

    Work *work = static_cast<Work *>(req->data);

    // the work has been done, and now we pack the results
    // vector into a Local array on the event-thread's stack.
    Local<Array> result_list = Array::New(isolate);
    for (unsigned int i = 0; i < work->results.size(); i++ ) {
      Local<Object> result = Object::New(isolate);
      pack_rain_result(isolate, result, work->results[i]);
      result_list->Set(i, result);
    }

    ...
```

The first part of the function above is pretty standard - we get the work data, and we package up the results into V8 objects rather than C++ vectors. Next, we need to invoke the JavaScript callback that was originally passed to the addon. **Note, this part is a lot different in Node 0.11 than it was in previous versions of Node because of recent V8 API changes.** If you are looking for ways to be a little less dependent on V8, take a look at Chapter 6 covering `Nan` - which abstracts away a lot of these issues.

```
    // set up return arguments
    Handle<Value> argv[] = { Null(isolate), result_list };

    // execute the callback
```

```
    Local<Function>::New(isolate, work->callback)->
      Call(isolate->GetCurrentContext()->Global(), 2, argv);

   // Free up the persistent function callback
    work->callback.Reset();

    delete work;
}
```

Once you call the callback, you're back in JavaScript! The `print_rain_results` function will be called...

```
rainfall.calculate_results_async(locations,
  function(err, result) {
    if (err ) {
      console.log(err);
    }
    else {
      print_rain_results(result);
    }
  });
```

Note that since we've used the standard call signature for the callback (err, data), you could use a standard promise library like bluebird to promisify your addon call.

Understanding V8 Memory and Worker Threads

As discussed earlier in the chapter, the worker thread API and programming model largely driven by the need to avoid accessing V8 data from our worker threads. In the event loop thread we've constructed C++ objects containing copies of the V8 data sent as input. We've performed calculations in the worker thread using only these C++ objects, and then repackaged them into V8 when we get back to the event loop thread.

For situations where input data and output data is relatively small, this poses absolutely no issue - and these are probably the most common cases. However - what if your async addon is going to do a lot of computation over a large input? What if it generates a huge amount of data? Note that this **copying input and output data is being done in the event loop** - meaning if it takes a long time, we're blocking the event loop (which we hate doing!).

So we have two problems:

1. Copying data might be a waste of memory

2. Copying data might take long, which ties up the Node.js event loop.

Ideally, we'd prefer a way to directly access V8 data from our worker threads. This is something that the official Node.js and V8 documentation both specifically say we can't do... however I think it's instructive to see why (and documentation covering the "why" is hard to come by!). The remainder of this chapter is about proving it won't work, and the ways we can mitigating the problem of copying within the event loop.

Warning: most of the following contains *anti-patterns* for asynchronous addons with Node.js.

Handles - Local and Persistent

How does C++ access JavaScript data in the first place? A lot of this is actually covered in Chapter 1 and 2 - but let's briefly review it here. When your Node.js JavaScript code is executing and allocating variables, it's doing so within the V8 JavaScript engine. V8 allocates JavaScript variables within it's own address space inside what we'll call "storage cells". When JavaScript calls into a C++ addon, the C++ code may obtain references to these storage cells by creating *handles* using the V8 API.

The most common handle type is `Local` - meaning it's scope is tied specifically to the current *handle scope*. The scope of a handle is critical, as V8 is charged with implementing garbage collection - and to do this it must keep track of how many references point to given storage cells. These references are normally within JavaScript, but the handles in our C++ addons must be kept track of too.

The most basic form of accessing pre-existing JavaScript variables occurs when we access arguments that have been passed into our C++ addon's when a method is invoked from JavaScript:

As you can see in Figure 6, the `Local<Object>` handle we create (`target`) will allow us to access V8 storage cells. `Local` handles only remain valid while the `HandleScope` object active when they are created is in scope. The V8 Embedder's Guide is once again the primary place to learn about `HandleScope` objects - however put simply, they are containers for handles. At a given time, only one `HandleScope` is active within V8 (or more specifically, a V8 `Isolate`). Where's the `HandleScope` in the above example? Good question!

It turns out that Node.js creates a `HandleScope` right before it calls our addon on the JavaScript code's behalf. This `HandleScope` is destroyed when the C++ addon function returns. Thus, any `Local` handle created inside our addon's function only survives until that function returns - meaning `Local` handles can never be accessed in worker threads when dealing with async addons - the worker threads clearly outlive the initial addon function call!

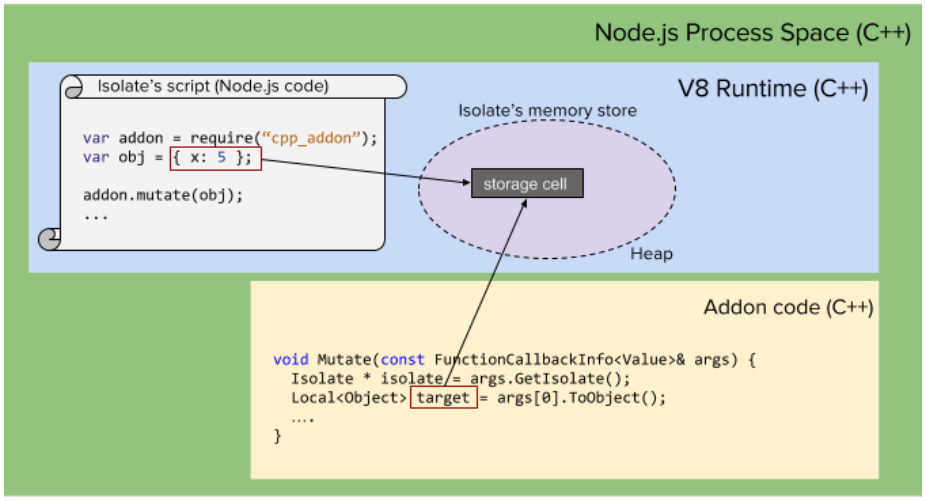


Figure 6: Local handles and V8 storage cells

Are `Persistent` handles the answer?

Maybe! As the name implies, `Persistent` handles are not automatically destroyed using `HandleScopes` - they can hang around indefinitely. Once you've created a persistent handle in your C++ code, V8 will honor that reference (and make sure you can still access the storage cells it points to) until you explicitly release it (you do this by calling the handle's `Reset` method). At first glance, this appears to be precisely the tool that would allow a long-lived C++ worker thread to access V8 data.

As a first experiment, lets maintain a reference to a JavaScript variable across C++ function calls by setting up a simple (non-threaded) example. Let's build a quick addon that allows JavaScript to pass in an object that the C++ hangs on to. Subsequent calls to the addon will *mutate* the JavaScript object originally passed into it - and we'll see these changes in JavaScript after the C++ returns.

```
#include <node.h>
using namespace v8;

// Stays in scope the entire time the addon is loaded.
Persistent<Object> persist;

void Mutate(const FunctionCallbackInfo<Value>& args) {
  Isolate * isolate = args.GetIsolate();
  Local<Object> target = Local<Object>::New(isolate, persist);

  Local<String> key = String::NewFromUtf8(isolate, "x");
```

```
  // pull the current value of prop x out of the object
  double current = target->ToObject()->Get(key)->NumberValue();
  // increment prop x by 42
  target->Set(key, Number::New(isolate,  current + 42));
}

void Setup(const FunctionCallbackInfo<Value>& args) {
  Isolate * isolate = args.GetIsolate();
  // Save a persistent handle for later use in Mutate
  persist.Reset(isolate, args[0]->ToObject());
}

void init(Local<Object> exports) {
  NODE_SET_METHOD(exports, "setup", Setup);
  NODE_SET_METHOD(exports, "mutate", Mutate);
}

NODE_MODULE(mutate, init)
```

Notice the two functions exposed by the addon. The first, `Setup`, creates a (global) `Persistent` handle to the object passed in from JavaScript. This is a pretty dubious use of global variables within an addon - it's probably a bad idea for non-trivial stuff - but this is just for demonstration. The point is that `Persistent<Object> target`'s scope is **not** tied to `Setup`.

The second function exposed by the addon is `Mutate`, and it simply adds 42 to `target`'s only property - `x`. Now let's look at the calling Node.js program.

```
const addon = require('./build/Release/mutate');

var obj = {   x: 0  };

// save the target JS object in the addon
addon.setup(obj);
console.log(obj);  // should print 0

addon.mutate(obj);
console.log(obj);  // should print 42

addon.mutate(obj);
console.log(obj);  // should print 84
```

When we run this program we'll see `obj.x` is initially 0, then 42, and then 84 when printed out. Living proof we can hang on to V8 within our addon across invocations... we're on to something!

```
> node mutate.js
{ x: 0 }
{ x: 42 }
{ x: 84 }
```

Bring on the worker threads!

Let's simulate a use case where a worker thread spends a long time modifying data iteratively. We'll modify the addon from above such that instead of needing JavaScript to call `Mutate`, it repeatedly changes `target`'s x value every 500ms in a worker thread.

```
#include <node.h>
#include <chrono>
#include <thread>
using namespace v8;

// Stays in scope the entire time the addon is loaded.
Persistent<Object> persist;

void mutate(Isolate * isolate) {
  while (true) {
    std::this_thread::sleep_for(std::chrono::milliseconds(500));
    // we need this to create a handle scope, since this
    // function is NOT called by Node.js
    v8::HandleScope handleScope(isolate);
    Local<String> key = String::NewFromUtf8(isolate, "x");
    Local<Object> target = Local<Object>::New(isolate, persist);
    double current = target->ToObject()->Get(key)->NumberValue();
    target->Set(key, Number::New(isolate,  current + 42));
    }
}

void Start(const FunctionCallbackInfo<Value>& args) {
  Isolate * isolate = args.GetIsolate();
  persist.Reset(isolate, args[0]->ToObject());

  // spawn a new worker thread to modify the target object
  std::thread t(mutate, isolate);
  t.detach();
}

void init(Local<Object> exports) {
  NODE_SET_METHOD(exports, "start", Start);
}
```

```
NODE_MODULE(mutate, init)
```

Note - we're not using libuv or any of the (best) common practice you'd normally see in an async addon; just ordinary C++ threads. Let's update the JavaScript to print out `obj` each second so we can see the fabulous work our C++ thread is doing.

```
const addon = require('./build/Release/mutate');

var obj = {   x: 0  };

addon.start(obj);

setInterval( () => {
  console.log(obj)
}, 1000);
```

If you've tried this before, you know what's coming next!

```
> node mutate.js
Segmentation fault: 11
```

Lovely. It turns out V8 doesn't allow what we're trying to do. Specifically, a single V8 instance (represented by an `Isolate`) can never be accessed by two threads. That is unless, of course, we use the built in V8 synchronization facilities in the form of `v8::Locker`. By using `v8::Locker`, we can prove to the V8 `isolate` that we are switching between threads - but that we never allow simultaneous access from multiple threads.

Understanding V8 Locker objects

Viewing V8 `isolate` as a shared resource, anyone familiar with thread synchronization can easily understand `v8::Locker` through the lens of a typical synchronization primitive in C++ - such as a mutex. The `v8::Locker` object is a lock, and we use RAII to use it. Specifically, the creation of a `v8::Locker` object (constructor) automatically blocks and waits until no other thread is within the `isolate`. Destruction of the `v8::Locker` (when it goes out of scope) automatically releases the lock - allowing other threads to enter.

We have two threads: (1) the Node.js (libuv) event loop and (2) the worker thread. Let's first look at adding locking to the worker:

```
void mutate(Isolate * isolate) {
  while (true) {
    std::this_thread::sleep_for(std::chrono::milliseconds(500));

    std::cerr << "Worker thread trying to enter isolate" << std::endl;
    v8::Locker locker(isolate);
    isolate->Enter();

    std::cerr << "Worker thread has entered isolate" << std::endl;
    // we need this to create local handles, since this
    // function is NOT called by Node.js
    v8::HandleScope handleScope(isolate);
    Local<String> key = String::NewFromUtf8(isolate, "x");
    Local<Object> target = Local<Object>::New(isolate, persist);
    double current = target->ToObject()->Get(key)->NumberValue();
    target->Set(key, Number::New(isolate,  current + 42));

    // Note, the locker will go out of scope here, so the thread
    // will leave the isolate (release the lock)
  }
}
```

At this point, since we haven't added locking anywhere else, you might think this would have very little effect if we run our program now. After all, there is seemingly no contention on the V8 lock, since the the worker is the only thread trying to lock it.

```
> node mutate.js
Worker thread trying to enter isolate
{ x: 0 }
{ x: 0 }
{ x: 0 }
{ x: 0 }
{ x: 0 }
{ x: 0 }
{ x: 0 }
^C
```

This is not what we see though... instead, we see that our worker thread *never acquires the lock!*. This implies our event loop thread owns the `isolate`. Diving into the Node.js source code, we can see this is correct!. Inside the `StartNodeInstance` method in `src/node.cc` (at time of this writing, around line 4096), a Locker object is created about 20 lines or so before beginning to process the main message pumping loop that drives every Node.js program.

```
// Excerpt from
// https://github.com/nodejs/node/blob/master/src/node.cc#L4096
static void StartNodeInstance(void* arg) {
  ...
  {
    Locker locker(isolate);
    Isolate::Scope isolate_scope(isolate);
    HandleScope handle_scope(isolate);

  //... (lines removed for brevity...)

    {
      SealHandleScope seal(isolate);
      bool more;
      do {
        v8::platform::PumpMessageLoop(default_platform, isolate);
        more = uv_run(env->event_loop(), UV_RUN_ONCE);

        if (more == false) {
          v8::platform::PumpMessageLoop(default_platform, isolate);
          EmitBeforeExit(env);
          more = uv_loop_alive(env->event_loop());
          if (uv_run(env->event_loop(), UV_RUN_NOWAIT) != 0)
            more = true;
        }
      } while (more == true);
    }
    ....
```

Node.js acquires the `isolate` lock *before* beginning the main event loop - *and it never relinquishes it.* This is where we might begin to realize our goal of accessing V8 data from a worker thread is impractical with Node.js. In other V8 programs, you might very well allow workflows like this, however Node.js specifically disallows multi-threaded access by holding the `isolate`'s lock throughout the entire lifetime of your program.

What about `Unlocker`?

If you look at the V8 documentation, you will find a counterpart to `Locker` though - `Unlocker`. Similar to `Locker`, it uses an RAII pattern - whenever it is in scope you've effectively unlocked the isolate. Perhaps we could use this in the event loop thread to unlock the isolate...

```
// Remember - this is called in the event loop thread
void Start(const FunctionCallbackInfo<Value>& args) {
```

```
  Isolate * isolate = args.GetIsolate();
  persist.Reset(isolate, args[0]->ToObject());

  // spawn a new worker thread to modify the target object
  std::thread t(mutate, isolate);
  t.detach();

  // This will allow the worker to enter...
  isolate->Exit();
  v8::Unlocker unlocker(isolate);
}
```

Running this ends in disaster though - segmentation fault as soon as `Start` returns - the worker thread never even gets a chance. This really should not have been a surprise. When `Start` returns, it's relinquished the lock it had on V8 and actually exited the isolate - but this is the thread that actually runs our JavaScript - a clear conflict in logic! The `seg fault` is a result of Node.js calling into V8 (to return control back to JavaScript) after it's exited the isolate. If we delay the return of `Start`, we can see that the worker thread *is able to access V8 data.*

```
void Start(const FunctionCallbackInfo<Value>& args) {
  ...

  isolate->Exit();
  v8::Unlocker unlocker(isolate);

  // as soon as we return, Node's going to access V8 which
  // will crash the program.  So we can stall...
  while (1);
}
```
```
```
```
> node mutate.js
Worker thread trying to enter isolate
Worker thread has entered isolate
Worker thread trying to enter isolate
Worker thread has entered isolate
...
```

In theory, we could force JavaScript to call into the addon to periodically release the isolate to allow the worker thread to access it for a while.

```
void Start(const FunctionCallbackInfo<Value>& args) {
  Isolate * isolate = args.GetIsolate();
```

```
  persist.Reset(isolate, args[0]->ToObject());

  // spawn a new worker thread to modify the target object
  std::thread t(mutate, isolate);
  t.detach();
}

void LetWorkerWork(
  const FunctionCallbackInfo<Value> &args) {

  Isolate * isolate = args.GetIsolate();
  {
    isolate->Exit();
      v8::Unlocker unlocker(isolate);

    // let worker execute for 200 seconds
    std::this_thread::sleep_for(std::chrono::seconds(2));
  }
  //v8::Locker locker(isolate);
  isolate->Enter();
}
```

And here's the JavaScript, graciously calling `LetWorkerWork`.

```
const addon = require('./build/Release/mutate');

var obj = {   x: 0  };

addon.start(obj);

setInterval( () => {
  addon.let_worker_work();
  console.log(obj)
}, 1000);
```

As you can see - this *does work* - we an access V8 data from a worker thread while the event loop is asleep, with an `Unlocker` in scope.

```
> node mutate.js
Worker thread trying to enter isolate
Worker thread has entered isolate
Worker thread trying to enter isolate
Worker thread has entered isolate
Worker thread trying to enter isolate
{ x: 84 }
```

```
Worker thread has entered isolate
Worker thread trying to enter isolate
Worker thread has entered isolate
{ x: 168 }
Worker thread trying to enter isolate
Worker thread has entered isolate
Worker thread trying to enter isolate
Worker thread has entered isolate
{ x: 252 }
...
```

While this makes for a nice thought experiment - it's not practical. The purpose of this entire exercise is to allow a worker thread to asynchronously work with JavaScript data. Further, we wanted to do this without copying lots of data which would slow down (block) the event loop thread. Keeping those goals in mind, the unlocker approach fails miserably - the worker thread can only access V8 data when the event loop is sleeping!

Why typical async addons work

If you've written async addons in C++ already, you surely have used `Persistent` handles to callbacks. You've gone through the trouble of copying input data into plain old C++ variables, used `libuv` or `Nan`'s wrappers to queue up a work object to be processed in a worker thread, and then copied the data back to V8 handles and invoked a callback.

It's a complicated workflow - but hopefully the above discussion highlights why its so important. Callback functions (passed in as arguments to our addon when it's invoked) must be held in `Persistent` handles, since we must access them when our work thread is completed (and well after the initial C++ function returns to JavaScript). Note however that we *never* invoke that callback from the worker thread. We invoke the callback (accessing the `Persistent` handle) when our C++ "completion" function is called by `libuv` - **in the event loop thread**.

Once you understand the threading rules, I think the elegance of the typical async solution pattern becomes a lot more clear.

Workarounds and Compromises

Our premise was that we could have a C++ addon that used data *from JavaScript* as input. Since we now know making a copy is necessary - we come back to the issue of copying *a lot* of data. If you find yourself in this situation, there may be some very simple ways to mitigate the problem.

First - does your input data *really need* to originate from JavaScript? If your addon is using a lot of data as input, where is that data *actually* coming from?

Is it coming from a database? *If so, then retrieve it in C++ - not JavaScript!*

Is it coming from a file (or an upload)? *If so, can you avoid reading it into JavaScript variables and just pass the filename/location into C++?*

If your data doesn't sit somewhere already, and JavaScript code is actually building it up incrementally over time (before asking a C++ addon to use it), then another option would be to store the data in the C++ addon directly (as plain old C++ variables). As you build data, instead of creating JavaScript arrays/objects - make calls into the C++ addon to copy data (incrementally) into a C++ data structure that will remain in scope.

There are few really good ways to do this. One is through the ObjectWrap API, which allows you to utilize C++ object directly from JavaScript - which is covered in the next chapter. JavaScript can work with a C++ addon's result data like this too - the worker thread can build up a C++ data structure wrapped in V8 accessors and your JavaScript can pull (copy) data from the structure when ever it needs to. While this approach does not avoid copying data, when used correctly it can avoid holding two copies of large input/output data - it only requires JavaScript to have copies of the parts of the shared data structure it needs "at the moment".

Another method involves building up Node.js Buffer objects and using those structures in your C++ addons. This technique works because Buffer objects allocate data *outside* of V8 implicitly - so they are indeed accessible from work threads.

Chapter 5 - Object Wrapping

We've now seen a lot of the underlying mechanics of exposing C++ *functions* to JavaScript programs. We've also seen how V8 data can be constructed in C++ addons and *returned* to JavaScript. There's a missing piece though - C++ objects. Classes (objects) serve the primary purpose of organizing data and methods into a single programming unit. Classes contain state (member variables), and methods (member functions) that can operate on them. JavaScript of course has classes too, but unfortunately there is no *seamless* way of simply taking an existing C++ class/object and letting JavaScript work with it. This means that C++ classes in your addon are essentially inaccessible, you can create them (on Node's C++ heap, not in V8 memory), but you can't easily return them to JavaScript. It's also impossible to let JavaScript directly access regular C++ objects created in your addon, since they don't reside in V8 memory.

All that said, there *is* a way to build a bridge between JavaScript and your addon such that the above mentioned use cases are possible - however it requires you to create a *new type of object* - a `node::ObjectWrap`. `ObjectWrap` is a base class provided by Node.js that includes the plumbing to connect JavaScript code to a C++ object, however classes that extend `ObjectWrap` don't necessarily look like plain-old C++ objects (POCOs). Classes extending `ObjectWrap` can be instantiated from JavaScript using the `new` operator, and their methods (which have V8-oriented call signatures) can be directly invoked from JavaScript.

The name `ObjectWrap` is sometimes misleading. Given an *existing* C++ object, you'd be forgiven for thinking `ObjectWrap` will allow you to magically decorate your existing C++ class such that it's accessible to JavaScript. Unfortunately, the *wrap* part really refers to a way to group methods and state, not decorate existing classes. Should you wish to use an existing C++ class from JavaScript, you will certainly want to use `ObjectWrap`, however you need to write custom code to bridge each of your C++ class methods.

Example: Polynomial Class

Let's start with a wish list. We want a second degree polynomial class (Ax^2 + Bx + C), which we can calculate roots (using the quadratic formula), calculate f(x), and set/get the three coefficients at anytime. From JavaScript, usage of the class would look like this:

```
var poly = new Polynomial(1, 3, 2);
console.log(poly.roots()) // prints [-1, -2]
console.log(poly.at(4)) // prints 30

poly.c = 0;
console.log(poly.at(4)) // prints 28
```

```
//prints 1, 3, 0
console.log(poly.a + ", " + poly.b + " " + poly.c);
```

Since we don't like doing things the easy way, we want to implement this class's logic in C++. We'll assume we are starting from a clean slate - we don't already have a suitable C++ implementation for Polynomial. Let's start with the class prototype for our Polynomial, along with the V8 bootstrapping code to export the class.

The code for this chapter is available in full in the `nodecpp-demo` repository at https://github.com/freezer333/nodecpp-demo, under the "ObjectWrap" section.

```
class WrappedPoly : public node::ObjectWrap {
 public:
  static void Init(v8::Local<v8::Object> exports);

 private:
  explicit WrappedPoly(
      double a = 0, double b = 0, double c = 0);

  ~WrappedPoly();

  static void New(
      const v8::FunctionCallbackInfo<v8::Value>& args) ;

  static v8::Persistent<v8::Function> constructor;
  double a_;
  double b_;
  double c_;
};

void InitPoly(Local<Object> exports) {
  WrappedPoly::Init(exports);
}

NODE_MODULE(polynomial, InitPoly)
```

This is the bare minimum setup for an `ObjectWrap`. Let's dissect the role of each method before looking at the code, as it's important to understand what's happening conceptually first - the code is convoluted. First notice that neither the constructor or destructor is public, they will not be called directly from outside the class (much less from JavaScript!).

The only public method created inside `WrappedPoly` is a `static` method - `Init`. `Init` is responsible for adding all class functions that will be called

from JavaScript to the `exports` object. `Init` is called by `InitPoly`, which is the entry point to our addon - it's called as soon as the addon is `required`. In this preliminary implementation, the only class function exposed will be a constructor.

```
static void Init(v8::Local<v8::Object> exports) {
  Isolate* isolate = exports->GetIsolate();

  // Prepare constructor template
  Local<FunctionTemplate> tpl =
    FunctionTemplate::New(isolate, New);

  tpl->SetClassName(
        String::NewFromUtf8(isolate, "Polynomial"));

  tpl->InstanceTemplate()->SetInternalFieldCount(1);

  exports->Set(
      String::NewFromUtf8(isolate, "Polynomial"),
      tpl->GetFunction());
}
```

`Init` first creates a *new JavaScript function* using a `FunctionTemplate`. This function is going to serve as a constructor function in JavaScript - it will be what is initially invoked when JavaScript executes `var poly = new Poly()`. Function templates are used in V8 to create functions within the V8 runtime, they can be associated with functions in the addon - in this case `New`. We'll look at `New`'s implementation next, but for now you just need to know that it will be responsible for actually creating a new instance of `WrappedPoly`. `Init` associates the new function with a class called `Polynomial`, which only serves to ensure that objects created using this function show up with the correct name in JavaScript. `InstanceTemplate()` returns the `ObjectTemplate` associated with this function.

Note that the call to `SetInternalFieldCount` is simply to allow for a single instance of a C++ object (the `WrappedPoly`) to be associated with the `ObjectTemplate`. The internal field count has nothing to do with the number of properties inside `WrappedPoly` or the JavaScript object that ends up being created. Typically, you'll keep this at 1, however this can also be used in advanced situations to associate additional data with the V8 memory associated with the template.

Finally, we explicitly export the function created by the template as `Polynomial` to the module's export object. Now let's look at the actual details of `New`, which is mapped to any call in JavaScript to `Polynomial`.

```
static void New(const v8::FunctionCallbackInfo<v8::Value>& args) {
  Isolate* isolate = args.GetIsolate();
  double a = args[0]->IsUndefined() ? 0 :
    args[0]->NumberValue();
  double b = args[1]->IsUndefined() ? 0 :
    args[1]->NumberValue();
  double c = args[2]->IsUndefined() ? 0 :
    args[2]->NumberValue();
  WrappedPoly* obj = new WrappedPoly(a, b, c);
  obj->Wrap(args.This());
  args.GetReturnValue().Set(args.This());
}
```

As you can see, this is basically like any other function we've exposed in other addons. The difference is that we've explicitly associated this function with a JavaScript function `Polynomial`, which should be called within the context of `new Polynomial(...)`. The function examines the callers arguments and instantiates a new `WrappedPoly` object. The new C++ object then get's some Node.js magic sprinkled into is by calling it's `Wrap` method (inherited from `ObjectWrap`). Notice that it takes `args.This()` as it's parameter - it is setting things up such that the object representing `this` inside the `Polynomial` function is truly the newly created wrapped polynomial. Finally, the return value is set (again, the `this` variable in the JavaScript call to `Polynomial`).

We can now build this addon and actually instantiate `Polynomial` from JavaScript:

```
const addon = require('./build/Release/polynomial');

var poly = new addon.Polynomial(1, 3, 2);
console.log(poly);
```

The resulting output will be:

```
Polynomial {}
```

The class name `Polynomial` is a direct result of the `tpl->SetClassName` call in `Init`. While the code is working OK, it's not robust. If our calling JavaScript uses the `Polynomial` function as a normal function instead of a constructor, things go badly:

```
var poly = addon.Polynomial(1, 3, 2);
console.log(poly);
```

... results in:

```
Assertion failed: (handle->InternalFieldCount() > 0), function Wrap,
file /Users/sfrees/.node-gyp/5.10.1/include/node/node_object_wrap.h,
Abort trap: 6
```

One might argue that the caller should not do such a thing, but we can capture this and still allow the call to return a new instance - we just need to detect if the function has been called using constructor syntax within the `New` function in the addon. If it hasn't, then we explicitly call the constructor (which actually just recalls `New` using constructor syntax!). Here's the complete listing, note the `Persistant` function that has been added called `constructor` - it's a persistent reference to the constructor we create in `Init` so it can be called from elsewhere (`New`) in the addon.

```
class WrappedPoly : public node::ObjectWrap {
 public:
  static void Init(v8::Local<v8::Object> exports) {
      Isolate* isolate = exports->GetIsolate();

      // Prepare constructor template
      Local<FunctionTemplate> tpl =
            FunctionTemplate::New(isolate, New);

      tpl->SetClassName(
            String::NewFromUtf8(isolate, "Polynomial"));

      tpl->InstanceTemplate()->SetInternalFieldCount(1);

      ////////////////////////////////////////
      // Store a reference to this constructor
      // so it can be called in New if New is called
      // without constructor syntax.

      constructor.Reset(isolate, tpl->GetFunction());
      ////////////////////////////////////////

      exports->Set(
          String::NewFromUtf8(isolate, "Polynomial"),
          tpl->GetFunction());
  }

 private:
  explicit WrappedPoly(
      double a = 0, double b = 0, double c = 0)
    : a_(a), b_(b), c_(c) {}
  ~WrappedPoly() {}
```

```
  static void New(
      const v8::FunctionCallbackInfo<v8::Value>& args) {

      Isolate* isolate = args.GetIsolate();

      if (args.IsConstructCall()) {
        // Invoked as constructor: `new Polynomial(...)`
        double a = args[0]->IsUndefined() ? 0 :
            args[0]->NumberValue();
        double b = args[1]->IsUndefined() ? 0 :
            args[1]->NumberValue();
        double c = args[2]->IsUndefined() ? 0 :
            args[2]->NumberValue();
        WrappedPoly* obj = new WrappedPoly(a, b, c);
        obj->Wrap(args.This());
        args.GetReturnValue().Set(args.This());
      } else {
        // Invoked as plain function `Polynomial(...)`,
        // turn into construct call.
        const int argc = 3;
        Local<Value> argv[argc] = { args[0] , args[1], args[2]};
        Local<Function> cons =
            Local<Function>::New(isolate, constructor);
        args.GetReturnValue().Set(
            cons->NewInstance(argc, argv));
      }
  }

  static v8::Persistent<v8::Function> constructor;
  double a_;
  double b_;
  double c_;
};

Persistent<Function> WrappedPoly::constructor;
```

Now we'll get the same results if we do `var poly = addon.Polynomial(1, 2, 3);` as we get with `var poly = new addon.Polynomial(1, 2, 2);`.

Adding methods

In our original hypothetical JavaScript program we included a `roots` function and a `at` function. We can now add these to the object that gets returned by the `Polynomial` constructor. This is achieved by adding methods to the *prototype* of

the instance associated with calls to the `Polynomial` constructor. For JavaScript beginners:

> All JavaScript objects inherit the properties and methods from their prototype. Objects created using an object literal, or with new Object(), inherit from a prototype called Object.prototype. Objects created with new Date() inherit the Date.prototype. The Object.prototype is on the top of the prototype chain. – *www.w3schools.com*

In this situation, we are going to add a function to `Polynomial.prototype`, which is achieved through the Node.js macro `NODE_SET_PROTOTYPE_METHOD`. This macro uses the function template (which, recall, is a constructor for types `Polynomial`), a name (to be used from JavaScript), and an addon function. This is all done inside `Init`, where we configure the function template.

```
static void Init(v8::Local<v8::Object> exports) {
  Isolate* isolate = exports->GetIsolate();

  // Prepare constructor template
  Local<FunctionTemplate> tpl =
      FunctionTemplate::New(isolate, New);
  tpl->SetClassName(
      String::NewFromUtf8(isolate, "Polynomial"));
  tpl->InstanceTemplate()->SetInternalFieldCount(1);

  ////////////////////////////////////////////////
  // Add the "at" function to the object prototype
  ////////////////////////////////////////////////
  NODE_SET_PROTOTYPE_METHOD(tpl, "at", At);

  constructor.Reset(isolate, tpl->GetFunction());
  exports->Set(String::NewFromUtf8(isolate, "Polynomial"),
           tpl->GetFunction());
}
```

`At`'s implementation is straightforward, it's just a normal addon function, with the exception of how it gains access to the underlying `WrappedPoly` object.

```
void WrappedPoly::At(
  const v8::FunctionCallbackInfo<v8::Value>& args) {

  Isolate* isolate = args.GetIsolate();
  double x = args[0]->IsUndefined() ? 0 :
```

```
    args[0]->NumberValue();
  WrappedPoly* poly =
    ObjectWrap::Unwrap<WrappedPoly>(args.Holder());

  double results =  x * x * poly->a_ +
                    x * poly->b_ +
                    poly->c_;

  args.GetReturnValue().Set(
    Number::New(isolate, results));
}
```

When JavaScript calls `poly.at`, based on the rules of JavaScript, whatever is on the left side of the `.` operator will be used as `this` in the function (`at`). In V8, `args.Holder()` refers to the `this` property at the time the function is invoked. Therefore, `args.Holder()` is returning our `WrappedPoly` - or at least the wrapper around it. Utilizing the static `Unwrap` method, we can extract the actual C++ object that was created when `New` was actually called. The rest of the code simply calculates the result and returns the value. The implementation of the `roots` function will follow the exact same pattern as `at`, only returning a `v8::Array` instead. We'll defer that until the complete code is shown at the end of this section.

Adding properties (accessor/mutators)

In JavaScript there is no concept of private member variables, object simply have properties. The following code JavaScript code will have strange results when using our addon "as is" though:

```
var poly = new Polynomial(1, 3, 2);
console.log(poly.at(4)) // prints 30

poly.c = 0; // no error

// should print 28, but prints 30!
console.log(poly.at(4))

console.log(poly);
```

The strange bit is that setting poly.c should (in theory) make the polynomial evaluate to 28 - however the addon reports 30 still. The reason is likely obvious: we haven't built any sort of bridge between `c` in the JavaScript object returned by `Polynomial` and the `c_` member variable held inside the `WrappedPoly` instance that is hiding behind it. Thankfully, we can fix this by setting accessors on the

`ObjectTemplate` we are using when creating the `Polynomial` constructor. We do this by revisiting `Init` once again:

```
static void Init(v8::Local<v8::Object> exports) {
  Isolate* isolate = exports->GetIsolate();

  // Prepare constructor template
  Local<FunctionTemplate> tpl =
    FunctionTemplate::New(isolate, New);

  tpl->SetClassName(
        String::NewFromUtf8(isolate, "Polynomial"));

  tpl->InstanceTemplate()->SetInternalFieldCount(1);

  NODE_SET_PROTOTYPE_METHOD(tpl, "at", At);

  ////////////////////////////////
  // Add an accessor - get/set
  ////////////////////////////////
  tpl->InstanceTemplate()->SetAccessor(
      String::NewFromUtf8(isolate, "c"), GetC, SetC);

  constructor.Reset(isolate, tpl->GetFunction());

  exports->Set(
      String::NewFromUtf8(isolate, "Polynomial"),
      tpl->GetFunction());
  }
```

The call to `SetAccessor` takes a property name and then callbacks to invoke when that particular property is accessed or mutated in JavaScript. The call signatures are a bit different than standard addon functions - they have extra parameters so you can inspect which property name is being accessed (likely redundant information, but can be useful for certain designs) and (for setters) the value being assigned. Here's the call signatures for GetC and SetC, which are defined in `WrappedPoly`:

```
static void GetC(Local<String> property,
    const PropertyCallbackInfo<Value>& info)

static void SetC(Local<String> property,
    Local<Value> value,
    const PropertyCallbackInfo<void>& info);
```

The implementations simply set/return the correct property on the unwrapped polynomial object, much in the same way as the `at` function did:

```
void WrappedPoly::GetC(Local<String> property,
  const PropertyCallbackInfo<Value>& info) {

  Isolate* isolate = info.GetIsolate();
  WrappedPoly* obj = ObjectWrap::Unwrap<WrappedPoly>(info.This());
  info.GetReturnValue().Set(Number::New(isolate, obj->c_));
}

void WrappedPoly::SetC(Local<String> property,
  Local<Value> value,
  const PropertyCallbackInfo<void>& info) {

  WrappedPoly* obj = ObjectWrap::Unwrap<WrappedPoly>(info.This());
  obj->c_ = value->NumberValue();
}
```

Now when we manipulate properties from JavaScript we'll see the correct results. When we print our object, we also see the actual properties inside the `Polynomial` object itself.

```
var poly = new Polynomial(1, 3, 2);
console.log(poly.at(4)) // prints 30

poly.c = 0;


console.log(poly.at(4))  // prints 28

console.log(poly);  // Polynomial {c:0}
```

Completed Wrapped Polynomial class

To round out the implementation, here is the complete code listing for the polynomial class. There are a few changes from what has been shown above - namely the implementation of the accessors. Since the polynomial class has three properties (a, b, c) one might expect to see three sets of accessors (getters and setters) - however this would require a lot of duplication of V8 boilerplate code. Instead, I've created one set of accessors - `GetCoeff` and `SetCoeff`, which inspect the property parameter to determine which coefficient to return or change. This is a matter of personal preference, you could certainly elect to create 6 distinct functions. In addition, the `roots` function is implemented as well.

```
#include <node.h>
#include <node_object_wrap.h>
#include <iostream>
#include <cmath>
using namespace std;
using namespace v8;

class WrappedPoly : public node::ObjectWrap {
 public:
  static void Init(v8::Local<v8::Object> exports) {
      Isolate* isolate = exports->GetIsolate();

      Local<FunctionTemplate> tpl =
        FunctionTemplate::New(isolate, New);
      tpl->SetClassName(
            String::NewFromUtf8(isolate, "Polynomial"));
      tpl->InstanceTemplate()->SetInternalFieldCount(1);

      NODE_SET_PROTOTYPE_METHOD(tpl, "at", At);
      NODE_SET_PROTOTYPE_METHOD(tpl, "roots", Roots);

      tpl->InstanceTemplate()->SetAccessor(
          String::NewFromUtf8(isolate, "a"), GetCoeff, SetCoeff);
      tpl->InstanceTemplate()->SetAccessor(
          String::NewFromUtf8(isolate, "b"), GetCoeff, SetCoeff);
      tpl->InstanceTemplate()->SetAccessor(
          String::NewFromUtf8(isolate, "c"), GetCoeff, SetCoeff);

      constructor.Reset(isolate, tpl->GetFunction());
      exports->Set(String::NewFromUtf8(isolate, "Polynomial"),
               tpl->GetFunction());
  }

 private:
  explicit WrappedPoly(double a = 0, double b = 0, double c = 0)
    : a_(a), b_(b), c_(c) {}
  ~WrappedPoly() {}

  static void New(const v8::FunctionCallbackInfo<v8::Value>& args) {
      Isolate* isolate = args.GetIsolate();

      if (args.IsConstructCall()) {
        // Invoked as constructor: `new Polynomial(...)`
        double a = args[0]->IsUndefined() ? 0 :
            args[0]->NumberValue();
        double b = args[1]->IsUndefined() ? 0 :
```

```
            args[1]->NumberValue();
        double c = args[2]->IsUndefined() ? 0 :
            args[2]->NumberValue();
        WrappedPoly* obj = new WrappedPoly(a, b, c);
        obj->Wrap(args.This());
        args.GetReturnValue().Set(args.This());
      } else {

        // Invoked as plain function `Polynomial(...)`,
        // turn into construct call.
        const int argc = 3;
        Local<Value> argv[argc] = { args[0] , args[1], args[2]};
        Local<Function> cons =
            Local<Function>::New(isolate, constructor);
        args.GetReturnValue().Set(cons->NewInstance(argc, argv));
      }
  }
  static void At(const v8::FunctionCallbackInfo<v8::Value>& args);
  static void Roots(const v8::FunctionCallbackInfo<v8::Value>& args);

  static void GetCoeff(Local<String> property,
    const PropertyCallbackInfo<Value>& info);
  static void SetCoeff(Local<String> property,
    Local<Value> value, const PropertyCallbackInfo<void>& info);

  static v8::Persistent<v8::Function> constructor;
  double a_;
  double b_;
  double c_;
};

Persistent<Function> WrappedPoly::constructor;

void WrappedPoly::At(
    const v8::FunctionCallbackInfo<v8::Value>& args) {

    Isolate* isolate = args.GetIsolate();
    double x = args[0]->IsUndefined() ? 0 : args[0]->NumberValue();
    WrappedPoly* poly = ObjectWrap::Unwrap<WrappedPoly>(args.Holder());

    double results = x * x * poly->a_ + x * poly->b_ + poly->c_;

    args.GetReturnValue().Set(Number::New(isolate, results));
}

void WrappedPoly::Roots(
```

```
    const v8::FunctionCallbackInfo<v8::Value>& args) {

    Isolate* isolate = args.GetIsolate();
    WrappedPoly* poly = ObjectWrap::Unwrap<WrappedPoly>(args.Holder());

    Local<Array> roots = Array::New(isolate);
    double desc = poly->b_ * poly->b_ - (4 * poly->a_ * poly->c_);
    if (desc >= 0 ) {
        double r = (-poly->b_ + sqrt(desc))/(2 * poly->a_);
        roots->Set(0,Number::New(isolate, r));
        if ( desc > 0) {
            r = (-poly->b_ - sqrt(desc))/(2 * poly->a_);
            roots->Set(1,Number::New(isolate, r));
        }
    }
    args.GetReturnValue().Set(roots);
}

void WrappedPoly::GetCoeff(Local<String> property,
    const PropertyCallbackInfo<Value>& info) {

    Isolate* isolate = info.GetIsolate();
    WrappedPoly* obj = ObjectWrap::Unwrap<WrappedPoly>(info.This());

    v8::String::Utf8Value s(property);
    std::string str(*s);

    if ( str == "a") {
        info.GetReturnValue().Set(Number::New(isolate, obj->a_));
    }
    else if (str == "b") {
        info.GetReturnValue().Set(Number::New(isolate, obj->b_));
    }
    else if (str == "c") {
        info.GetReturnValue().Set(Number::New(isolate, obj->c_));
    }

}

void WrappedPoly::SetCoeff(Local<String> property,
    Local<Value> value, const PropertyCallbackInfo<void>& info) {

    WrappedPoly* obj = ObjectWrap::Unwrap<WrappedPoly>(info.This());

    v8::String::Utf8Value s(property);
    std::string str(*s);
```

```
    if ( str == "a") {
        obj->a_ = value->NumberValue();
    }
    else if (str == "b") {
        obj->b_ = value->NumberValue();
    }
    else if (str == "c") {
        obj->c_ = value->NumberValue();
    }
}

void InitPoly(Local<Object> exports) {
  WrappedPoly::Init(exports);
}

NODE_MODULE(polynomial, InitPoly)
```

Wrapping existing objects

What if we already had a Polynomial class in C++ code, which was just a normal C++ class unaware of V8 and Node? Unfortunately, we still need to do all the same steps as we've seen above. You would still want to create a new `WarppedPoly` class, and create all the same V8 functions to expose features such as `roots`, `at`, and property accessors. The difference in this scenario is that `WrappedPolynomial` would likely have an instance of your `Polynomial` POCO instead of three coefficients of it's own. V8-based functions like `roots` and `at` would simply forward the calls on to the POCO `Polynomial` implementation. It's not the most satisfying solution - since it means duplicating class interfaces - but it will allow you to export your POCO functionality to your Node.js programs.

Wrapped Objects as Arguments

As a final example, let's take a look at situations where instances of your *wrapped* class are passed *back* into C++ addon functions from JavaScript. An example of this might be an addon with a standalone `add` function which accepts two polynomials. Here's what it might look like from the JavaScript perspective:

```
var polyA = new addon.Polynomial(1, 3, 2);
var polyB = new addon.Polynomial(8, -2, 4);
var polyC = addon.add(polyA, polyB);

// prints 9, 1, -2
console.log(polyC.a + ", " + polyC.b + ", " + polyC.c);
```

Here, the `add` function is a static member of Polynomial.

```
// defined within the WrappedPoly class
static void Add(const FunctionCallbackInfo<Value>& args) {
    Isolate* isolate = args.GetIsolate();

    WrappedPoly* obj1 =
        node::ObjectWrap::Unwrap<WrappedPoly>(
        args[0]->ToObject());
    WrappedPoly* obj2 =
        node::ObjectWrap::Unwrap<WrappedPoly>(
        args[1]->ToObject());

    double a = obj1->a_ + obj2->a_;
    double b = obj1->b_ + obj2->b_;
    double c = obj1->c_ + obj2->c_;

    Local<Value> argv[3] = {
        Number::New(isolate, a),
        Number::New(isolate, b),
        Number::New(isolate, c)};

    Local<Function> cons =
        Local<Function>::New(isolate, constructor);
    args.GetReturnValue().Set(cons->NewInstance(3, argv));
 }
```

The method is exported from the module using standard V8 style - it's not exported as part of the JS Polynomial class.

```
void InitPoly(Local<Object> exports) {
  WrappedPoly::Init(exports);
  NODE_SET_METHOD(exports, "add", WrappedPoly::Add);
}
```

Chapter 6 - Native Abstractions for Node (NAN)

Throughout the first five chapters of this book we've been operating under the assumption that the addons we are creating target Node.js versions 0.12 through 6.0. That's a pretty broad version set, but there are earlier versions of Node.js where our addons won't compile/work. This can be a significant issue when using cloud hosting services, which might be using an earlier version of Node.js. It also poses problems when distributing your addons over `npm` - since your addon can't be distributed to all users.

Why won't our addons work in earlier versions? Why should we worry that future versions of Node.js could break our addons? The answer is that it's not Node.js that is defining the API for which we inter-operate with JavaScript, it's V8! The V8 API has changed over time, and there is no guarantee that it won't again. While the V8 developers do attempt to keep the API stable, frequently there will be some (perhaps small) breaking changes introduced in each new release. As new V8 releases are moved into Node.js releases, we run the constant danger of our addons needing to be modified.

Given the prevalence of Node.js runtime versions (many applications are still running on Node.js v0.10, v0.12, v4.0, etc.), there is a real need for some form of abstraction, so that we might target a more stable (and clean!) API. Native Abstractions for Node (NAN) is exactly that.

NAN is governed by the io.js working group and aims to provide a set of macros and utilities that achieve two goals: 1) maintaining a stable API across all Node.js versions, and 2) providing a set of utilities to help simplify some of the most difficult parts of the Node.js and V8 API's (such as asynchronous addon development). While developing with NAN is only required if you intend to support addons across v0.10 - 4.0 of Node.js, it is a good idea to become familiar with it and use it on any project where you expect to have wide variety of users over a long period of time.

It should be noted that NAN is *not* a high level API. While there are few utilities to simplify asynchronous addon development, the majority of NAN simply provides replacement macro calls for standard V8 operations, such as creating new primitives/objects in the JavaScript heap, defining exported functions, and Node.js operations like object wrapping. The underlying rules of V8 programming which we've learned so far all still apply - NAN simply provides a slightly different API for doing the same things we've done before.

In this chapter, I'll move quickly to show the corresponding "NAN way" of doing the things we've accomplished in Chapters 1-5. In later chapters, we'll use NAN nearly exclusively - as it's extremely helpful for complex addon development and makes a lot of sense when publicly publishing addons to `npm`.

All of the code for this chapter is available in full in the `nodecpp-demo` repository at https://github.com/freezer333/nodecpp-demo, under the "Conversions_NAN", "primes", and "ObjectWrap_NAN" sections.

Basic Functionality

To demonstrate simple functionality, let's revisit the addon examples from Chapter 2, specifically where we passed primitives and objects/arrays to and from JavaScript and C++. Here's the function we created within the addon:

```
// adds 42 to the number passed and returns result
NODE_SET_METHOD(exports, "pass_number", PassNumber);

// adds 42 to the integer passed and returns result
NODE_SET_METHOD(exports, "pass_integer", PassInteger);

// reverses the string passed and returns the result
NODE_SET_METHOD(exports, "pass_string", PassString);

// nots the boolean passed and returns the result
NODE_SET_METHOD(exports, "pass_boolean", PassBoolean);

// extracts the x/y numeric properties in object
// passed to it and return a new object with sum
// and product
NODE_SET_METHOD(exports, "pass_object", PassObject);

// increments each numeric value in the input array, then returns
// an array a with a[0] equal to input[0]+1, a[1] equal to the
// "none_index" property defined on the input array, and a[2] equal
// to input[2]+1.
NODE_SET_METHOD(exports, "increment_array", IncrementArray);
```

These example functions were basic, but they allowed us to see all the typical use cases of V8 variables. Now let's create a new addon, with the same functionality, only using NAN instead. We'll examine each one side by side, so we can get a really clear picture on the differences between the raw V8 API and NAN's.

Build setup

Before we can start, we need to configure our addon to use NAN. When creating an addon using NAN, NAN becomes a dependency of your module. Thus, within the `package.json` file you must declare NAN as a dependency - `$ npm install --save nan`.

Unlike most modules you install with `npm` however, doing the install does not install a JavaScript package, it simply downloads the C++ (mostly headers) distribution of NAN. You will not need to reference NAN from JavaScript -

but you will need to reference it from your C++ code, specifically by including `nan.h`.

To get the dependencies into your addon, we add a bit of `node-gyp` magic into our `binding.gyp` file:

```
"include_dirs" : [
    "<!(node -e \"require('nan')\")"
]
```

This directive will result in the `nan` module's header files being on the build path when we compile our C++ addon.

Creating functions

Let's start by implementing the most simple function in our example - `pass_number`. We'll create a folder to contain our addon, create a package.json, install NAN by doing an `npm install nan --save`, and configure the following `binding.gyp` file:

```
{
  "targets": [
    {
      "target_name": "basic_nan",
      "sources": [ "basic_nan.cpp" ],
      "include_dirs" : [
        "<!(node -e \"require('nan')\")"
      ]
    }
  ]
}
```

We'll now create `basic_nan.cpp` and setup our addon.

```
#include <nan.h>

using namespace Nan;
using namespace v8;

NAN_METHOD(PassNumber) {
    // do nothing ... for now.
}

NAN_MODULE_INIT(Init) {
```

```
    Nan::Set(target,
      New<String>("pass_number").ToLocalChecked(),
      GetFunction(New<FunctionTemplate>(PassNumber))
       .ToLocalChecked());
}

NODE_MODULE(basic_nan, Init)
```

Compare this to the setup for the same function using the standard V8 API:

```
#include <node.h>

using namespace v8;

void PassNumber(const FunctionCallbackInfo<Value>& args) {
    // do nothing ... for now.
}

void Init(Local<Object> exports) {
  NODE_SET_METHOD(exports, "pass_number", PassNumber);
}

NODE_MODULE(basic_nan, Init)
```

Let's start at the very bottom actually - notice the `NODE_MODULE` command is exactly the same. `NODE_MODULE` is a macro already - NAN need not wrap it to preserve compatibility across Node.js version. As will be a pattern with NAN, the authors haven't invented new ways of doing things unless there is reason to believe API changes in V8/Node will/have created problems.

Working our way up, we have the `Init` method. In the pure V8 approach, we have a specific call signature for the initialization function used in `NODE_MODULE` - it must have a return type of `void` and accept `exports` (and the module, optionally). In NAN, a macro is provided called `NAN_MODULE_INIT`. The macro is parameterized, so we still get to name the initialization routine (`Init`), however the macro shields us from any inconsistencies to the call signature.

Inside `Init` we start seeing some major changes, and oddly it actually gets slightly more complicated with NAN! Instead of using the NODE_SET_METHOD macro, we instead utilize the `Set` function within `NAN`. Notice there is a "magic" variable in the NAN version of `Init`, called `target`. This is supplied by the `NAN_MODULE_INIT` macro, and is indeed to `exports` object we are accustomed to working with. To set a value on any object using `NAN`, we must utilize actual JavaScript strings and functions - thus the extra code to convert "pass_numbers" into a V8 string and reference `PassNumber` as a JavaScript function. We'll cover the details of these conversions in a moment, as they use the standard `New` and `ToLocalChecked` methods that `NAN` relies quite heavily on.

Moving up again, we look at the `PassNumber` function. In raw V8 code, we must have a specific call signature - where `void` is returned and we accept a `FunctionCallbackInfo` object containing calling information. This signature in particular has changed rather drastically over V8 versions, so it's no surprise NAN provides us with a macro to stamp this out. `NAN_METHOD` creates an appropriate function signature given the Node.js version installed. As we'll see in a moment, it also creates an `info` parameter which will allow us to get function arguments and the holder (this) in a V8 version agnostic way.

All the way at the top of the two listings, you'll notice we are now using an additional namespace `Nan`. The `v8` namespace is still used, and `node.h` is already included through `nan.h`.

Working with Primitives

Let's remain on `PassNumber`. In it's simplest form, without error checking, the raw V8 implementation looked like this:

```
void PassNumber(const FunctionCallbackInfo<Value>& args) {
    Isolate * isolate = args.GetIsolate();

    double value = args[0]->NumberValue();

    Local<Number> retval = Number::New(isolate, value + 42);
    args.GetReturnValue().Set(retval);
}
```

The conversion to using NAN couldn't be easier. The main changes will:

1. No need to obtain an `Isolate`.
2. We will use the built-in `info` parameters, supplied by the `NAN_METHOD` macro instead of `FunctionCallbackInfo` directly.
3. We'll use `Nan::New` to create a new number, rather than V8::Number (this is why we won't need `Isolate`.

```
NAN_METHOD(PassNumber) {
    double value = info[0]->NumberValue();

    Local<Number> retval = Nan::New(value + 42);
    info.GetReturnValue().Set(retval);
}
```

Performing error checking is straightforward, since the `info` object has much the same functions available as `FunctionCallbackInfo<Value> args` has in the original V8 examples:

```
if ( info.Length() < 1 ) {
   info.GetReturnValue().Set(Nan::New(0));
   return;
}

if ( !info[0]->IsNumber()) {
  info.GetReturnValue().Set(Nan::New(-1));
  return;
}
```

The integer and boolean variations of these functions follow the same pattern - the `Nan::New` method is overloaded to create the appropriate values, and `info` has `IntegerValue` and `BooleanValue`.

```
NAN_METHOD(PassInteger) {
    if ( info.Length() < 1 ) {
        return;
    }
    if ( !info[0]->IsInt32()) {
        return;
    }
    int value = info[0]->IntegerValue();
    Local<Integer> retval = Nan::New(value + 42);
    info.GetReturnValue().Set(retval);
}

NAN_METHOD(PassBoolean) {
    if ( info.Length() < 1 ) {
        return;
    }
    if ( !info[0]->IsBoolean()) {
        return;
    }
    bool value = info[0]->BooleanValue();
    Local<Boolean> retval = Nan::New(!value);
    info.GetReturnValue().Set(retval);
}
```

Maybe types and `ToLocalChecked`

If you read the NAN documentation, you will quickly notice there is a large emphasis on the `Maybe` type concept, which is a relatively new feature of V8. Likewise, while our examples above work with arguments and new primitives in a concise format, much of the NAN examples instead use `ToLocalChecked` when creating JavaScript primitives and objects (including functions). In fact,

we saw this style in the `Init` method itself. Let's look into what these concepts mean, and why you'd want to use them.

Let's imagine we have our `PassNumber` method exported and call it with JavaScript code like this:

```
console.log( addon.pass_number("xyz") );
```

In our original code, `info[0]` is a `Local<Value>` and it was trivial to get the "correct" number value given the argument passed to the add, `info[0]->NumberValue()` will return `NaN` if the argument is not a valid Number. There is an additional way to perform the conversion to a Number though, using `Nan::To` and `ToLocalChecked`.

```
NAN_METHOD(PassNumber) {

    Nan::MaybeLocal<Number> value = Nan::To<Number>(info[0]);

    // will crash if value is empty
    Local<Number> checked = value.ToLocalChecked();
    Local<Number> retval = Nan::New(checked->Value() + 42);
    info.GetReturnValue().Set(retval);
}
```

Here we see a (trivial) use of `Maybe` types, which may or may not hold values. `Nan::To` returns `Maybe` types - and converting into a true local is achieved by calling `ToLocalChecked`. Note that if `value` was actually empty, `ToLocalChecked` would crash the program. Interestingly though, `value` will always have an actual value because the conversion to `Number` is well defined in JavaScript - anything that isn't an number is `NaN`. In this case, no matter what we pass to `PassNumber`, `ToLocalChecked` will always succeed.

We could also move directly to a `double`, instead of creating a `Local<Number>`.

```
NAN_METHOD(PassNumber) {
    Nan::Maybe<double> value = Nan::To<double>(info[0]);
    Local<Number> retval = Nan::New(value.FromJust() + 42);
    info.GetReturnValue().Set(retval);
}
```

We'll get the same results with this function as we've gotten before - here we've used the `Maybe` type's `FromJust` method to get the actual double value. Again, since numbers *always* have a defined value from any other JavaScript type - these `Maybe` examples are a bit superfluous - they never crash! Nevertheless, you'll see this style used in many places.

Working with Strings

Extracting strings from arguments is similar to numbers and booleans, however converting them to actual C++ strings requires the extra step of using `Nan::Utf8String`, just like we needed to use `v8::String::Utf8Value` when dealing directly with V8.

```
NAN_METHOD(PassString) {
    Nan::MaybeLocal<String> tmp = Nan::To<String>(info[0]);
    Local<String> local_string = tmp.ToLocalChecked();

    Nan::Utf8String val(local_string);

    std::string str (*val);
    std::reverse(str.begin(), str.end());

    info.GetReturnValue().Set(
         Nan::New<String>(str.c_str()).ToLocalChecked());
}
```

When creating a new string object, we are now forced to deal directly with `Maybe` types again, unlike when creating new numbers. This is because while `Nan::New` returns `Local` for most JavaScript types, it returns `MaybeLocal` when creating strings, dates, regular expressions, and script objects. This inconsistency is driven by the underlying V8 API. As we saw above however, converting to a `Local` merely requires us to use `ToLocalChecked` on the return value of `Nan::New<String>`. Since we are specifically passing something known to be a string, we need not worry about `ToLocalChecked` actually failing.

There is a slightly more convenient method of getting to a `Utf8String` from the `info` object as well, using the `ToString` method defined on `Local<Value>`.

```
NAN_METHOD(PassString) {
    v8::String::Utf8Value val(info[0]->ToString());
    std::string str (*val);
    std::reverse(str.begin(), str.end());

    info.GetReturnValue().Set(
         Nan::New<String>(str.c_str()).ToLocalChecked());
}
```

Working with Objects

So far you might have noticed that the real changes when moving to NAN are centered around the use (or lack) of `Isolate` and how new variables are allocated.

NAN does not replace the direct usage of much of the V8 API - specifically `Local<Value>`. As we move to working with objects, this becomes even more apparent. Let's look at the first, pure V8 implementation of our `PassObject` addon function from Chapter 2.

```
Local<Value> make_return(
    Isolate * isolate, const Local<Object> input ) {

    Local<String> x_prop = String::NewFromUtf8(isolate, "x");
    Local<String> y_prop = String::NewFromUtf8(isolate, "y");
    Local<String> sum_prop =
        String::NewFromUtf8(isolate, "sum");
    Local<String> product_prop =
        String::NewFromUtf8(isolate, "product");

    double x = input->Get(x_prop)->NumberValue();
    double y = input->Get(y_prop)->NumberValue();

    HandleScope scope(isolate); // bad..
    Local<Object> obj = Object::New(isolate);
    obj->Set(sum_prop, Number::New(isolate, x + y));
    obj->Set(product_prop, Number::New(isolate, x * y));

    return obj;
}

void PassObject(const FunctionCallbackInfo<Value>& args) {
    Isolate * isolate = args.GetIsolate();
    Local<Object> target = args[0]->ToObject();
    Local<Value> obj = make_return(isolate, target);
    args.GetReturnValue().Set(obj);
}
```

The function expects an object to be passed in with numeric properties `x` and `y`. It builds a return object containing `sum` and `product` and returns it to JavaScript. The changes to utilize NAN are quite straightforward.

```
Local<Value> make_return(const Local<Object> input ) {
    Local<String> x_prop =
        Nan::New<String>("x").ToLocalChecked();
    Local<String> y_prop =
        Nan::New<String>("y").ToLocalChecked();
    Local<String> sum_prop =
        Nan::New<String>("sum").ToLocalChecked();
    Local<String> product_prop =
```

```
        Nan::New<String>("product").ToLocalChecked();

    double x = input->Get(x_prop)->NumberValue();
    double y = input->Get(y_prop)->NumberValue();

    Local<Object> obj = Nan::New<Object>();
    obj->Set(sum_prop, Nan::New<Number>(x + y));
    obj->Set(product_prop, Nan::New<Number>(x * y));

    return obj;
}

NAN_METHOD(PassObject) {
    if ( info.Length() > 0 ) {
        Local<Object> target = info[0]->ToObject();
        Local<Value> obj = make_return(target);
        info.GetReturnValue().Set(obj);
    }
}
```

Note that the syntax to pull a property from the `input` object is changed, since we are using `Nan::New` to build the string property names. We are also utilizing `Nan::New<Object>` to create the new object to be returned. All references to `Isolate` have been removed, as this is being handled for us behind the scenes by NAN. In the example above, we are still using the native V8 `Set` method on `Local<Object>` to set the sum and product properties. NAN also has it's own property setting method as well, which when used would further shield you from V8 API version changes:

```
Nan::Set(obj, sum_prop, Nan::New<Number>(x+y));
Nan::Set(obj, product_prop, Nan::New<Number>(x*y));
```

Likewise, we can use `Nan::Get` to retrieve `x` and `y` instead of using the raw V8 API. Here's the version of `make_return` in it's full NAN glory:

```
Local<Value> make_return(const Local<Object> input ) {
    Local<String> x_prop =
        Nan::New<String>("x").ToLocalChecked();
    Local<String> y_prop =
        Nan::New<String>("y").ToLocalChecked();
    Local<String> sum_prop =
        Nan::New<String>("sum").ToLocalChecked();
    Local<String> product_prop =
        Nan::New<String>("product").ToLocalChecked();
```

```
    Local<Object> obj = Nan::New<Object>();
    double x = Nan::Get(input, x_prop)
        .ToLocalChecked()->NumberValue();
    double y = Nan::Get(input, y_prop)
        .ToLocalChecked()->NumberValue();

    Nan::Set(obj, sum_prop, Nan::New<Number>(x+y));
    Nan::Set(obj, product_prop, Nan::New<Number>(x*y));
    return obj;
}
```

Now seems like a good time to point out what may already be obvious... using NAN and V8 API's is not an "either or" proposition, there is no problem with mixing. That said, if you are using NAN at all, it likely makes sense to write everything you can using the "NAN way" - this way you'll get the full advantage of version compatibility.

Working with Arrays

Rounding out our conversion of the basic Chapter 2 examples, let's take a look at changes when dealing with arrays. Recall, we build a sort of silly array addon function which accepted an array of numerics, but also had a property called "not_index" to demonstrate how we could still access named properties on an array. The addon returned an array containing 3 elements - the first and last were simply the first and last elements from the input array, incremented by one. The second index in the returned array was set to the value found in "not_index". Here's what we had using V8:

```
void IncrementArray(const FunctionCallbackInfo<Value>& args) {
    Isolate * isolate = args.GetIsolate();
    Local<Array> array = Local<Array>::Cast(args[0]);

    for (unsigned int i = 0; i < array->Length(); i++ ) {
      if (array->Has(i)) {
        double value = array->Get(i)->NumberValue();
        array->Set(i, Number::New(isolate, value + 1));
      }
    }


    Local<String> prop =
        String::NewFromUtf8(isolate, "not_index");
    Local<Array> a = Array::New(isolate);
    a->Set(0, array->Get(0));
    a->Set(1, array->Get(prop));
```

```
    a->Set(2, array->Get(2));

    args.GetReturnValue().Set(a);
}
```

Let's jump right to the full NAN implementation. The key changes will of course be creating the new array, the new values, and using NAN getters and setters to access the array indexes and properties. Note the use of an integer with `Nan::Set` and `Nan::Get`, which allows you to index the array.

```
NAN_METHOD(IncrementArray) {
    Local<Array> array = Local<Array>::Cast(info[0]);

    for (unsigned int i = 0; i < array->Length(); i++ ) {
      if (Nan::Has(array, i).FromJust()) {
        double value = Nan::Get(array, i)
            .ToLocalChecked()->NumberValue();
        Nan::Set(array, i, Nan::New<Number>(value + 1));
      }
    }

    Local<String> prop =
        Nan::New<String>("not_index").ToLocalChecked();
    Local<Array> a = New<v8::Array>(3);
    Nan::Set(a, 0, Nan::Get(array, 0).ToLocalChecked());
    Nan::Set(a, 1, Nan::Get(array, prop).ToLocalChecked());
    Nan::Set(a, 2, Nan::Get(array, 2).ToLocalChecked());

    info.GetReturnValue().Set(a);
}
```

In the above example, you may notice one glaring inconsistency - we are using the basic V8 `Local<Array>::Cast` function to convert our single function parameter into an array. Curiously, unlike primitives and objects, NAN does not currently provide a "NAN way" of doing the casting to arrays. Thankfully, the code above will work in all versions of Node.js - which is likely why a "NAN way" has never made it into the development pipeline.

Callbacks and Asynchronous Patterns

We've seen several examples throughout this book that involve JavaScript functions being sent to C++ addons, and then invoked from the C++. In particular, we saw both synchronous and asynchronous callbacks in Chapter 4, where we calculated rainfall statistics in C++. One area where NAN goes a bit beyond

simple macros and API redefinition is when dealing with callbacks and asynchronous workers. For these subjects, NAN actually provides some extra utility classes that make accomplishing things *significantly* easier (and different) than raw V8.

Instead of resurrecting the rainfall example from Chapter 4, which is fairly large, let's focus on a more succinct addon function, which computes prime numbers. We'll start out with a synchronous version, and then reconfigure it to create an asynchronous version. In both cases, we'll export a single function called `prime` that accepts two parameters. The first will be an integer, `N`, for which we will generate all prime numbers `<` `N`. The second parameter will be a *callback* function that will be called once the prime numbers are calculated. The callback should accept a `primes` array, which contains the results. We'll use it like this:

```
addon.primes(20, function (primes) {
    // prints 2, 3, 5, 7, 11, 13, 17, 19
    console.log("Primes less than 20 = " + primes);
});
```

Let's first start out creating a standard C++ function that accepts `N` and a `vector` to fill with prime numbers - we'll use this in both the synchronous and asynchronous version of our addon. `find_primes` implements a simplified prime sieve algorithm.

```
void find_primes(int limit, vector<int> & primes) {
    std::vector<int> is_prime(limit, true);
    for (int n = 2; n < limit; n++ ) {
        if (is_prime[n] ) primes.push_back(n);
        for (int i = n * n; i < limit; i+= n) {
            is_prime[i] = false;
        }
    }
}
```

Let's now take a look at what the addon code would look like in a synchronous version:

```
NAN_METHOD(Primes) {
    int limit = info[0]->IntegerValue();
    Callback *callback = new Callback(info[1].As<Function>());

    vector<int> primes;
    find_primes(limit, primes);
```

```
    Local<Array> results = New<Array>(primes.size());
    for ( unsigned int i = 0; i < primes.size(); i++ ) {
        Nan::Set(results, i, New<v8::Number>(primes[i]));
    }

    Local<Value> argv[] = { results };
    callback->Call(1, argv);
}

NAN_MODULE_INIT(Init) {
   Nan::Set(target, New<String>("primes").ToLocalChecked(),
        GetFunction(New<FunctionTemplate>(Primes))
        .ToLocalChecked());
}

NODE_MODULE(primes, Init)
```

The most interesting aspect of the addin is how we are using NAN to help us deal with the callback, which must be invoked once we have our results. Note the use of the `Callback` object, it is a NAN class that wraps a standard `v8::Function`. In the synchronous code above, it provides limited added value over using `v8::Function` directly - however `Callback` protects functions it wraps from garbage collection (utilizing `Persistent` handles). This functionality greatly simplifies callback usage when working with asynchronous addons, so they are good to get in the habit of using.

Most addons that accept callbacks are asynchronous (otherwise, why not just return the results directly!). In Chapter 4 we dove right into working with `libuv`, creating a multi-threaded addon where we managed the movement of data between the worker thread and Node event loop thread ourselves. It was a lot of work, and it's easy to make mistakes. NAN offers several utility classes that abstract the concept of moving data between the event loop and worker threads, which lets us focus on actual code a bit more quickly.

The key class NAN provides is `Nan::AsyncWorker`. This class allows you to create a new worker thread and facilitates storing data persistently, invoking the worker thread code, and invoking the persistent callback when you are done. Your first step is to setup a class that inherits from `AsyncWorker` and implement the abstract `Execute` method. Typically input to your worker thread will be saved within your class, generally passed in through it's constructor. When your `Execute` function completes (we'll see how to get it to run in a moment), `AsyncWorker` will call it's `HandleOKCallback` - which typically you'll override to invoke a callback to send the results back to JavaScript. `AsyncWorker` ensures `Execute` runs in a background thread and `HandleOKCallback` (and `HandleErrorCallback`, if necessary) are executed as part of the event loop.

The code below sets up a basic `AsyncWorker` implementation that works with our prime number code:

```
class PrimeWorker : public AsyncWorker {
    public:
    PrimeWorker(Callback * callback, int limit)
        : AsyncWorker(callback), limit(limit) {

    }
    // Executes in worker thread
    void Execute() {
        find_primes(limit, primes);
    }
    // Executes in event loop
    void HandleOKCallback () {
        Local<Array> results = New<Array>(primes.size());
        for ( unsigned int i = 0; i < primes.size(); i++ ) {
            Nan::Set(results, i, New<v8::Number>(primes[i]));
        }
        Local<Value> argv[] = { results };
        callback->Call(1, argv);
    }
    private:
    int limit;
    vector<int> primes;
};
```

Note that in `PrimeWorker`'s constructor, we call the base class's constructor to pass in the callback - which `AsyncWorker` will store persistently. The actual work is being done in `Execute`, which uses member variables `limit` and `primes` to compute the results. Once `Execute` completes, `HandleOKCallback` is called (on the event loop thread) and we invoke the callback. The code is virtually identical to the synchronous version, just spread out over two member functions.

Of course, this class must be instantiated. We'll do so in our addon function, but we won't directly invoke `Execute`. Instead, NAN provides a utility method called `AsyncQueueWorker` which accepts a `AsyncWorker` and queue's it's execution up such that `Execute` is run in a worker thread. This frees us from dealing directly with `libuv` entirely.

```
NAN_METHOD(Primes) {
    int limit = To<int>(info[0]).FromJust();
    Callback *callback = new Callback(info[1].As<Function>());

    AsyncQueueWorker(new PrimeWorker(callback, limit));
}
```

Invoking the synchronous and asynchronous versions of the prime addon from JavaScript works the same way, and we'll get identical results - however the asynchronous is far superior in that the heavy CPU computation being done for large `N` values won't tie up the event loop from working on other things!

Sending Progress Updates from async addons

In the prime addon above, we generate prime numbers within a core `for` loop, which could take a while to complete. We collect all the prime numbers in a `vector` and then dump it all back at once at the end. It would be nice if we could display the current progress we've made, so a user isn't left in the dark about how long the process might take. While we could do this with `libuv` directly, NAN really helps us out in this case - by providing a subclass of `AsyncWorker` called `AsyncProgressWorker`. `AsyncProgressWorker` builds out a convenient pattern for sending progress updates back to JavaScript during asynchronous execution.

The core principle behind `AsyncProgressWorker` is an additional parameter passed into our `Execute` function. The parameter, of type `ExecutionProgress` is our window into queuing up calls to a second callback we must now implement - `HandleProgressCallback`. `HandleProgressCallback` will be executed in the event loop, and `ExecutionProgresss` provides a method that allows us to store data (persistently), and add a call to `HandleProgressCallback` to the queue. This method, `progress.Send` is simple - it accepts a pointer to data, and it's size. The data is stored in a `Persistent` handle within `ExecutionProgress` until the callback is invoked in the event loop.

Below is an adaptation of the `PrimeWorker` code from above that now sends progress updates indicating how close it is to completing. In the `Execute` function we are actually computing prime numbers, instead of delegating to the original `find_primes`, since we need to send progress updates (you could, of course, modify `find_primes` to accept a `progress` object as well). Note that we are also sleeping after each turn around the loop, just for effect (otherwise we'd be done quite quickly!). The sleep code requires C++11, and inclusion of `<thread>` and `<chrono>` libraries.

```
class PrimeProgressWorker : public AsyncProgressWorker {
    public:
    PrimeProgressWorker(Callback * callback,
        Callback * progress, int limit)
        : AsyncProgressWorker(callback),
            progress(progress), limit(limit) {

    }
    // Executes in worker thread
    void Execute(
```

```
        const AsyncProgressWorker::ExecutionProgress & progress) {

        std::vector<int> is_prime(limit, true);
        for (int n = 2; n < limit; n++ ) {
            double p = (100.0 * n) / limit;
            progress.Send(reinterpret_cast<const char*>(&p),
                sizeof(double));
            if (is_prime[n] ) primes.push_back(n);
            for (int i = n * n; i < limit; i+= n) {
                is_prime[i] = false;
            }
            std::this_thread::sleep_for(
                    chrono::milliseconds(100));
        }
    }
    // Executes in event loop
    void HandleOKCallback () {
        Local<Array> results = New<Array>(primes.size());
        for ( unsigned int i = 0; i < primes.size(); i++ ) {
            Nan::Set(results, i, New<v8::Number>(primes[i]));
        }
        Local<Value> argv[] = { results };
        callback->Call(1, argv);
    }

    void HandleProgressCallback(const char *data, size_t size) {
        // Required, this is not created automatically
        Nan::HandleScope scope;

        Local<Value> argv[] = {
            New<v8::Number>(*reinterpret_cast<double*>(
                const_cast<char*>(data)))
        };
        progress->Call(1, argv);
    }
    private:
    Callback *progress;
    int limit;
    vector<int> primes;
};
```

In an odd quirk, unlike `HandleOKCallback`, a `HandleScope` is *not* created automatically for us before `HandleProgressCallback` is invoked - so it's our responsibility to create one before allocating new V8 memory. Note that `AsyncProgressWorker` handles "progress updates" as overwrite-able - meaning each time a progress message is logged it overwrites the one previously logged.

In our example above, since we are waiting 100 milliseconds on each turn around the for loop, it's likely that you'll see 0%, 1%, 2%... 100% appear on the screen. If we were to remove the 100 millisecond wait however, you might be surprised to simply see the 100% complete message - and likely none of the other progress updates. This is because each time you invoke `SendProgress` a job is queued to the event loop (the job of calling `HandleProgressCallback`). There is no guarantee that that job will execute before your worker thread sends another progress message - and in our case, for relatively small `N` values, it's likely ALL the progress updates will be logged before `libuv` ever actually gets around to calling `HandleProgressCallback`.

For progress updates, the behavior described make sense, since there shouldn't be an harm in dropping obsolete progress updates. For sending actual data to JavaScript (like partial results) however, this functionality is disastrous! Ultimately, `AsyncWorker` and `AsyncProgressWorker` are really meant as examples, they shouldn't be looked at as representative of the scope of what we can do. In Chapter 7, we'll see the idea of progress updates extended to provide streaming interfaces to and from addons. We'll utilize additional queues to ensure that no "progress" messages are ever overwritten, making our implementations suitable for sending real data.

As an aside, the prime number example we've just seen is also the subject of Appendix A, where alternative methods of integrating existing C++ code are discussed.

ObjectWrap

The last piece of the C++ addon puzzle we talked about before hitting NAN was `ObjectWrap`, which allowed us to pass complete C++ classes to JavaScript, provided we decorated them with a lot of Node/V8 boilerplate. That Node/V8 boilerplate code leaves us susceptible to version issues - `ObjectWrap` API in fact has changed through several versions of Node.js.

Thankfully, learning the "NAN" way to do object wrapping is pretty trivial, since Chapter 5's discussion was based on recent versions (v4+) of Node.js, you'll notice that the NAN `ObjectWrap` implementation is nearly identical to what we've already learned. The benefit of using it, rather than the API in Chapter 5, is derived from the ability to target earlier versions of Node.js.

Rather than repeating lots of code, the reader should refer back to the final implementation of `WrappedPoly` in Chapter 5. You'll notice that the class definition extended `node::ObjectWrap`. To use the NAN implementation, simply extend `Nan::ObjectWrap` instead (of course, you must include `nan.h` too). Most changes that you'll end up making while moving to NAN for an ObjectWrap stem from the typical NAN changes (`New`, no `Isolate`, etc.), however there are some changes specific to `Nan::ObjectWrap`.

First let's look at the `WrappedPoly` class declaration from Chapter 5 and highlight the changes we'll want to make:

```
class WrappedPoly : public node::ObjectWrap {
 public:
  static void Init(v8::Local<v8::Object> exports);

 private:
  explicit WrappedPoly(double a = 0, double b = 0, double c = 0)
    : a_(a), b_(b), c_(c) {}
  ~WrappedPoly() {}

  static void New(
        const v8::FunctionCallbackInfo<v8::Value>& args) ;
  static void At(
        const v8::FunctionCallbackInfo<v8::Value>& args);
  static void Roots(
        const v8::FunctionCallbackInfo<v8::Value>& args);

  static void GetCoeff(Local<String> property,
    const PropertyCallbackInfo<Value>& info);
  static void SetCoeff(Local<String> property,
    Local<Value> value, const PropertyCallbackInfo<void>& info);

  static v8::Persistent<v8::Function> constructor;
  double a_;
  double b_;
  double c_;
};
```

When moving to `Nan::ObjectWrap` the major changes that take place center on method declarations. It is wise in this case to use NAN macros for the `Init`, accessors, and member methods. In addition, due to namespacing issues, the `constructor` object must be moved outside of the `WrappedPoly`, and modified from `v8::Persistent<v8::Function>` to `Nan::Persistent<v8::FunctionTemplate>`. Here's the full declaration:

```
static Persistent<v8::FunctionTemplate> constructor;

class WrappedPoly : public Nan::ObjectWrap {
 public:
  static NAN_MODULE_INIT(Init) ;

 private:
  explicit WrappedPoly(double a = 0, double b = 0, double c = 0)
```

```
    : a_(a), b_(b), c_(c) {}
  ~WrappedPoly() {}

  static NAN_METHOD(New) ;
  static NAN_METHOD(At) ;
  static NAN_METHOD(Roots) ;

  static NAN_GETTER(GetCoeff);
  static NAN_SETTER(SetCoeff);

  double a_;
  double b_;
  double c_;
};
```

Within the methods, changes mostly involve `Nan::New` and the like, however there are some interesting changes in `Init`. Here is the original implementation from Chapter 5:

```
static void Init(v8::Local<v8::Object> exports) {
    Isolate* isolate = exports->GetIsolate();

    Local<FunctionTemplate> tpl =
        FunctionTemplate::New(isolate, New);
    tpl->SetClassName(String::NewFromUtf8(isolate, "Polynomial"));
    tpl->InstanceTemplate()->SetInternalFieldCount(1);

    NODE_SET_PROTOTYPE_METHOD(tpl, "at", At);
    NODE_SET_PROTOTYPE_METHOD(tpl, "roots", Roots);

    tpl->InstanceTemplate()->SetAccessor(
            String::NewFromUtf8(isolate, "a"),
            GetCoeff, SetCoeff);
    tpl->InstanceTemplate()->SetAccessor(
            String::NewFromUtf8(isolate, "b"),
            GetCoeff, SetCoeff);
    tpl->InstanceTemplate()->SetAccessor(
            String::NewFromUtf8(isolate, "c"),
            GetCoeff, SetCoeff);

    constructor.Reset(isolate, tpl->GetFunction());
    exports->Set(String::NewFromUtf8(isolate, "Polynomial"),
               tpl->GetFunction());
}
```

Moved to NAN, we need to use different methods to set prototype methods and

accessors:

```
static NAN_MODULE_INIT(Init) {
    // Prepare constructor template
    v8::Local<v8::FunctionTemplate> tpl =
        Nan::New<v8::FunctionTemplate>(WrappedPoly::New);

    tpl->SetClassName(
            Nan::New<v8::String>("Polynomial").ToLocalChecked());
    tpl->InstanceTemplate()->SetInternalFieldCount(1);

    // Prototype
    SetPrototypeMethod(tpl, "at", WrappedPoly::At);
    SetPrototypeMethod(tpl, "roots", WrappedPoly::Roots);

    v8::Local<v8::ObjectTemplate> itpl = tpl->InstanceTemplate();
    SetAccessor(itpl, Nan::New<v8::String>("a").ToLocalChecked(),
        WrappedPoly::GetCoeff, WrappedPoly::SetCoeff);
    SetAccessor(itpl, Nan::New<v8::String>("b").ToLocalChecked(),
        WrappedPoly::GetCoeff, WrappedPoly::SetCoeff);
    SetAccessor(itpl, Nan::New<v8::String>("c").ToLocalChecked(),
        WrappedPoly::GetCoeff, WrappedPoly::SetCoeff);

    constructor.Reset(tpl);
    Set(target,
        Nan::New<v8::String>("Polynomial").ToLocalChecked(),
        tpl->GetFunction());
}
```

Viewed side by side, you can see the changes are mainly "cosmetic", involving new NAN methods `SetAccessor` and `SetPrototypeMethod`. The rest of the changes to the `WrappedPoly` class are self-explanatory, they are just converting variable allocations/conversions to NAN syntax. Below is the full code listing of a completed example using `Nan`.

```
#include <cmath>
#include <nan.h>
#include <string>
using namespace Nan;

static Persistent<v8::FunctionTemplate> constructor;

class WrappedPoly : public Nan::ObjectWrap {
 public:
  static NAN_MODULE_INIT(Init) {
```

```
      v8::Local<v8::FunctionTemplate> tpl =
        Nan::New<v8::FunctionTemplate>(WrappedPoly::New);
      constructor.Reset(tpl);
      tpl->SetClassName(
            Nan::New<v8::String>("Polynomial").ToLocalChecked());
      tpl->InstanceTemplate()->SetInternalFieldCount(1);
      SetPrototypeMethod(tpl, "at", WrappedPoly::At);
      SetPrototypeMethod(tpl, "roots", WrappedPoly::Roots);
      v8::Local<v8::ObjectTemplate> itpl = tpl->InstanceTemplate();
      SetAccessor(itpl, Nan::New<v8::String>("a").ToLocalChecked(),
        WrappedPoly::GetCoeff, WrappedPoly::SetCoeff);
      SetAccessor(itpl, Nan::New<v8::String>("b").ToLocalChecked(),
        WrappedPoly::GetCoeff, WrappedPoly::SetCoeff);
      SetAccessor(itpl, Nan::New<v8::String>("c").ToLocalChecked(),
        WrappedPoly::GetCoeff, WrappedPoly::SetCoeff);
      Set(target,
        Nan::New<v8::String>("Polynomial").ToLocalChecked(),
        tpl->GetFunction());
  }

 private:
  explicit WrappedPoly(double a = 0, double b = 0, double c = 0)
    : a_(a), b_(b), c_(c) {}
  ~WrappedPoly() {}

  static NAN_METHOD(New) {
      double a = info[0]->IsUndefined() ? 0 :
        info[0]->NumberValue();
      double b = info[1]->IsUndefined() ? 0 :
        info[1]->NumberValue();
      double c = info[2]->IsUndefined() ? 0 :
        info[2]->NumberValue();
      WrappedPoly* obj = new WrappedPoly(a, b, c);
      obj->Wrap(info.This());
      info.GetReturnValue().Set(info.This());
  }
  static NAN_METHOD(At) ;
  static NAN_METHOD(Roots) ;

  static NAN_GETTER(GetCoeff);
  static NAN_SETTER(SetCoeff);

  double a_, b_, c_;
};

NAN_METHOD(WrappedPoly::At){
```

```
    double x = info[0]->IsUndefined() ? 0 :
        info[0]->NumberValue();
    WrappedPoly* poly =
        ObjectWrap::Unwrap<WrappedPoly>(info.Holder());
    double results = x * x * poly->a_ +
        x * poly->b_ +
        poly->c_;

    info.GetReturnValue().Set(
            Nan::New<v8::Number>(results));
}

NAN_METHOD(WrappedPoly::Roots){
    WrappedPoly* poly =
        ObjectWrap::Unwrap<WrappedPoly>(info.Holder());
    v8::Local<v8::Array> roots = Nan::New<v8::Array>();
    double desc = poly->b_ * poly->b_ - (4 * poly->a_ * poly->c_);
    if (desc >= 0 ) {
        double r = (-poly->b_ + sqrt(desc))/(2 * poly->a_);
        roots->Set(0,Nan::New<v8::Number>(r));
        if ( desc > 0) {
            r = (-poly->b_ - sqrt(desc))/(2 * poly->a_);
            roots->Set(1,Nan::New<v8::Number>(r));
        }
    }
    info.GetReturnValue().Set(roots);
}

NAN_GETTER(WrappedPoly::GetCoeff) {
    v8::Isolate* isolate = info.GetIsolate();
    WrappedPoly* obj = ObjectWrap::Unwrap<WrappedPoly>(info.This());
    v8::String::Utf8Value s(property);
    std::string str(*s);
    if ( str == "a")
        info.GetReturnValue().Set(
            v8::Number::New(isolate, obj->a_));
    else if (str == "b")
        info.GetReturnValue().Set(
            v8::Number::New(isolate, obj->b_));
    else if (str == "c")
        info.GetReturnValue().Set(
            v8::Number::New(isolate, obj->c_));
}

NAN_SETTER(WrappedPoly::SetCoeff) {
    WrappedPoly* obj = ObjectWrap::Unwrap<WrappedPoly>(info.This());
```

```
    v8::String::Utf8Value s(property);
    std::string str(*s);

    if ( str == "a") obj->a_ = value->NumberValue();
    else if (str == "b") obj->b_ = value->NumberValue();
    else if (str == "c") obj->c_ = value->NumberValue();
}

void InitPoly(v8::Local<v8::Object> exports) {
  WrappedPoly::Init(exports);
}

NODE_MODULE(polynomial, InitPoly)
```

Nan - conclusions

NAN aims to never fundamentally alter the programming model around C++ addon, only to provide a thin API layer around V8 to ensure backwards and forwards compatibility. Moving forward, the general consensus is to use NAN wherever possible - it's simply a safer bet than trying to keep up with versions of V8 yourself. Moving forward with the remainder of this book, including appendices, we'll stick largely with NAN - although sometimes my old V8 habits do find their way into the code listings!

Chapter 7 - Streaming between Node and C++

We've come a long way now! Not only have we learned how to deal with basic Node.js C++ integrations using V8, but we're now armed with the tools to wrap objects that can maintain state. We know how to abstract away much of the nastiness of asynchronous execution and V8 versions using `nan`. Now it's time to put it all together and develop addons that offer more sophisticated ways of interacting with C++ from JavaScript. In this chapter we'll build addons that expose interfaces that mimic `EventEmitter`s and streams. These interfaces are particularly well suited for long running C++ worker tasks that continuously either send input back to JavaScript or receive a series of inputs from it.

The concepts discussed in this chapter are advanced, if you merely skimmed the last few chapters you might want to go back and read them more carefully. To get our streaming interface, we'll make heavy use of `nan`, asynchronous execution, and even `ObjectWrap`. I'll cover the concepts through a series of distinct examples, and then meld the code together a bit to end up with a reusable SDK for developing streaming C+++ addons.

As a first step, we'll focus on exposing our C++ addons via an `EventEmitter`-like interface rather than function calling. We'll build from there, wrapping those event emitters in an JS adapter module to expose streaming interfaces.

If you are following along with the `nodecpp-demo` code, head over to the `streaming` directory in that repository to look at the full, completed code for this chapter.

Emitting events from C++

Sometimes we use C++ to do some heavy compute task that actually generates partial results over time. One way to work with addons like this would be to make it asynchronous, and when the results are completed, we could make the addon return the entire result (with partial results included, perhaps). Alternatively, we could implement the asynchronous addon such that all partial results were returned immediately, through invoking a JS callback. Moving further, the "Node way" of capturing a sequence of data is typically through the `EventEmitter` interface.

One problem that fits (sort of) this model is prime factorization. Factoring a number N, especially when it is large, is a bit time consuming[6] - however we compute factors in series, not all at once. Let's try to develop a C++ addon *emits* factors as they are computed - yielding us a Node.js program that looks something like this:

[6]To be clear, prime factorization is likely *not* a heavy enough computation task to warrant building out a complete C++ addon, let along a streaming interface. However, like most of the examples in this book, it's a simple enough problem that it doesn't require lots of explanation - yet demonstrates the programming model nicely.

```
// factorization.js
var factor = require(<path to addon>);

const factorization = factor({n:9007199254740991});

factorizer.on('factor', function(factor){
    console.log("Factor:  " + factor);
});

factorizer.on('close', function() {
    console.log("Factorization complete");
});
```

We'd end up with the following output:

```
Factor:  6361
Factor:  69431
Factor:  20394401
Factorization complete
```

As you can see, this hypothetical addon emits events of type `factor`, along with the standard `close` event (and probably `error` too!).

Let's build it.

Building the C++ addon

The core of the C++ addon code is the factorization process. Before we get into all the addon boilerplate and thread safety, we can model what the C++ code should really look like:

```
void factorize(uint32_t n) {
    while (n%2 == 0) {
        writeToNode(2);
        n = n/2;
    }
    for (uint32_t i = 3; i <= n; i = i+2) {
        while (n%i == 0) {
            writeToNode(i);
            n = n/i;
        }
    }
```

The key thing here is that the `send_to_node` method has to (1) get the integer factor across the thread boundary, from the worker to the event loop thread, and

(2) end up getting mapped to a proper event emitter. Notice that `send_to_node` is essentially the "send progress" problem though - we have an addon that is long running, and sends frequent updates. As we saw in Chapter 6, `nan` provides an excellent starting point for such and addon, `AsyncProgressWorker`. We are going to largely hijack this class, with some important changes to ensure our addon receives *every* message we send it [7].

Let's first create a standard implementation of AsyncProgressWorker that computes factorizations. Since we've already covered this topic in Chapter 6, it should be pretty straightforward at this point:

```
class Factorizer : public AsyncProgressWorker {
 public:
  Factorizer(Callback *progress, Callback *callback, uint32_t n)
    : AsyncProgressWorker(callback), progress(progress), n(n)
    {}
  ~Factorizer() {}

  // Executes in the new worker thread (background)
  void Execute (
      const AsyncProgressWorker::ExecutionProgress& progress) {
    uint32_t factor = 2;
    while (n%2 == 0) {
        progress.Send(
            reinterpret_cast<const char*>(&factor),
            sizeof(uint32_t));
        n = n/2;
    }
    for (uint32_t i = 3; i <= n; i = i+2) {
        while (n%i == 0) {
            progress.Send(
                reinterpret_cast<const char*>(&i),
                sizeof(uint32_t));
            n = n/i;
        }
    }
  }

  // Executes in the event-loop thread
  void HandleProgressCallback(const char *data, size_t size) {
    HandleScope scope;
```

[7] `AsyncProgressWorker` collapses consecutive progress updates if they haven't been processed by JavaScript quickly. For example, if progress is reported at 0, 50, 100% in the worker code all before the next event loop cycle, the Node.js will only receive a single progress indicator - 100%. This makes a lot of sense from a progress report standpoint - but not for message passing!

```
    v8::Local<v8::Value> argv[] = {
          New<v8::Integer>(
              *reinterpret_cast<int*>(const_cast<char*>(data)))
    };
    progress->Call(1, argv);
  }

 protected:
  Callback *progress;
  uint32_t n;
};

NAN_METHOD(Factor) {
  Callback *progress = new Callback(info[1].As<v8::Function>());
  Callback *callback = new Callback(info[2].As<v8::Function>());
  AsyncQueueWorker(new Factorizer( callback, progress
    , To<uint32_t>(info[0]).FromJust()
  ));
}

NAN_MODULE_INIT(Init) {
  Set(target
    , New<v8::String>("factorize").ToLocalChecked()
    , New<v8::FunctionTemplate>(Factor)->GetFunction());
}

NODE_MODULE(factorization, Init)
```

The code listing above creates the addon, now we can use it as follows (assuming this js file is in the same directory as the cpp / gyp files.

```
"use strict";

const path = require("path");

var addon_path = path.join(
    __dirname, "build/Release/factorization");
const worker = require(addon_path);

worker.factorize(9007199254740991, function() {
    console.log("Factorization Complete");
}, function(factor){
    console.log("Factor:  " + factor);
})
```

Unfortunately, if you run this on a reasonably fast machine, you'll get a rather

disappointing output!

```
Factorization complete
```

The problem here is that factorization is actually pretty quick. If you look deep inside AsyncProgressWorker, you'll see that the code in fact declines to invoke the progress callback if the worker is already completed. In this case, we've finished the loop before the `lib_uv` event loop even gets a chance to run, so no progress is reported. As a quick fix, let's put a `Sleep(1000)` at the very end of the `Execute` function - which should be plenty of time for the event loop to wake up and process our updates:

```
void Execute (
    const AsyncProgressWorker::ExecutionProgress& progress) {
  uint32_t factor = 2;
  while (n%2 == 0) {
      progress.Send(
          reinterpret_cast<const char*>(&factor),
          sizeof(uint32_t));
      n = n/2;
  }
  for (uint32_t i = 3; i <= n; i = i+2) {
      while (n%i == 0) {
          progress.Send(
          reinterpret_cast<const char*>(&i),
          sizeof(uint32_t));
          n = n/i;
      }
  }
}
```

Our output likely won't be much better though:

```
Factor:  65537
Factorization complete
```

What happened to the other factors - such as, 3, 5, 17, and 257? Again, the `AsyncProgressWorker`'s implementation comes back to bite us here, as there is no functionality build in to handle multiple "progress" updates within one event loop cycle. We ended up calling `progress.Send` 5 times before the event loop got a chance to process our requests - each time overwriting the "current" progress. Only the last value was propagated.

We could fix this by re-implementing `AsyncProgressWorker`, but instead let's just build a queue to hold each message we send and then ensure that each

time our `HandleProgressCallback` callback is invoked (at least once, if the worker hasn't already completed) we drain the queue - sending each item to JavaScript via individual progress callback invocations. Note, I'm going to use a thread-safe queue implementation - not a standard C++ queue. This is critical, because in fact what we're creating is a producer-consumer queue model, where "progress" updates are produced by the background thread and consumed in the `HandleProgressCallback` (event loop thread).

Here's the queue, which will be used in the remaining examples as well, so I've templated it. We'll just use `readAll` right now, but in later examples we'll use `read` to get one item off the queue at a time as well. It uses C++11 synchronization primitives to ensure thread-safety.

```
template<typename Data>
class PCQueue
{
public:
    void write(Data data) {
        while (true) {
            std::unique_lock<std::mutex> locker(mu);
            buffer_.push_back(data);
            locker.unlock();
            cond.notify_all();
            return;
        }
    }
    Data read() {
        while (true)
        {
            std::unique_lock<std::mutex> locker(mu);
            cond.wait(locker, [this](){
                return buffer_.size() > 0;
            });
            Data back = buffer_.front();
            buffer_.pop_front();
            locker.unlock();
            cond.notify_all();
            return back;
        }
    }
    void readAll(std::deque<Data> & target) {
        std::unique_lock<std::mutex> locker(mu);
        std::copy(
            buffer_.begin(),
            buffer_.end(),
            std::back_inserter(target));
```

```
        buffer_.clear();
        locker.unlock();
    }
    PCQueue() {}
private:
    std::mutex mu;
    std::condition_variable cond;
    std::deque<Data> buffer_;
};
```

Now let's put one of these queues into our worker class (named `toNode`, type `PCQueue<uint32_t>`) and adapt the callbacks such that the queue is used to store progress messages. First off, let's create a private function `drainQueue` which reads all items and sends them to Node by invoking the provided callback function.

```
// private method within the Factorizer class
void drainQueue() {
    HandleScope scope;
    // drain the queue - since we might only get
    // called once for many writes
    std::deque<uint32_t> contents;
    toNode.readAll(contents);
    for(uint32_t & item : contents) {
        v8::Local<v8::Value> argv[] = {
          New<v8::Integer>(*reinterpret_cast<int*>(&item))
        };
        progress->Call(1, argv);
    }
}
```

We can now utilize this function in the `HandleProgressCallback` method, and also in `HandleOKCallback` - which is a virtual function in `AsyncProgressWorker` that we'll now override.

```
void HandleOKCallback() {
    drainQueue();
    callback->Call(0, NULL);
}

void HandleProgressCallback(const char *data, size_t size) {
  drainQueue();
}
```

Now let's work on getting our factors onto this queue. Let's create a wrapper function that accepts the `progress` object and the `uin32_t` data item and adds

the data to the queue and fires off a progress event. Notice, the queue is being sent in in `progress.Send`. In reality, we could send anything - we are effectively ignoring what is being sent into this function, since these calls will result in `drainQueue` being called. I'll leave code cleanup to the reader - there are some obvious simplifications we could make if we were willing to mess with the actual implementation of `AsynProgressWorker` itself.

```
// protected member of Factorizer
void writeToNode(
    const AsyncProgressWorker::ExecutionProgress& progress,
    uint32_t & factor){

  toNode.write(factor);
  progress.Send(
      reinterpret_cast<const char*>(&toNode), sizeof(toNode));
}
```

We'll replace calls to `progress.Send` in `Execute` with our new wrapper, and our output will now capture everything!

```
void Execute (
    const AsyncProgressWorker::ExecutionProgress& progress) {

  uint32_t factor = 2;
  while (n%2 == 0) {
     writeToNode(progress, factor);
     n = n/2;
  }
  for (uint32_t i = 3; i <= n; i = i+2) {
     while (n%i == 0) {
        writeToNode(progress, i);
        n = n/i;
     }
  }
}
```

```
Factor:  3
Factor:  5
Factor:  17
Factor:  257
Factor:  65537
Factorization Complete
```

Creating the EventEmitter interface

We have a working asynchronous addon, but our initial aim was to interact with it as an EventEmitter. This is really just an adaptation - it can be done in JavaScript. An EventEmitter should emit arbitrary named events, plus "error" and "close" events. Our addon currently invokes a specific callback when complete and another when there is progress. We must change the addon in 2 ways to facilitate adapting it into an EventEmitter - (1) we should be sending name/value pairs from the addon (event name, event data) instead of just data, and (2) we should have an error callback!

The first change is pretty easy - instead of sending instances of `uint32_t`, let's send a new object - `Message`. `Message` will be a name/value pair object - where it's data is a *string* to maximize flexibility. The use of templates instead of serializing data to strings is left as an exercise by the reader.

```
class Message {
public:
  string name;
  string data;
  Message(string name, string data) : name(name), data(data){}
};
```

The queue that we originally placed in the `Factorizer` will now hold `Message` objects rather than `uint32_t`. When sending data to node, we'll construct our message object with the name "factor".

The second change is simply the addition of an error callback. `AsyncProgressWorker` already has a built-in `HandleErrorCallaback` virtual function that we can override to invoke a JS callback when errors occur. We'll need to create a new protected member, `error_callback`, which will be passed to `Factorizer`'s constructor when launching the addon.

```
void HandleErrorCallback() {
    HandleScope scope;

    v8::Local<v8::Value> argv[] = {
        // ErrorMessage() is method of `AsyncWorker`,
        // `AsyncProgressWorker`'s base class.
        v8::Exception::Error(New<v8::String>(ErrorMessage())
            .ToLocalChecked())
    };
    error_callback->Call(1, argv);
  }
```

Here is the complete code listing, including the modified constructor and Node/nan boilerplate code to accommodate the new error callback.

```
class Factorizer : public AsyncProgressWorker {
 public:
  Factorizer(Callback *progress, Callback *callback,
    Callback *error_callback, uint32_t n)
    : AsyncProgressWorker(callback), progress(progress),
    error_callback(error_callback), n(n)
    {

    }
  ~Factorizer() {}

  void Execute (
      const AsyncProgressWorker::ExecutionProgress& progress) {

    uint32_t factor = 2;
    while (n%2 == 0) {
        send_factor(progress, factor);
        n = n/2;
    }
    for (uint32_t i = 3; i <= n; i = i+2) {
        while (n%i == 0) {
            send_factor(progress, i);
            n = n/i;
        }
    }
  }

  void HandleOKCallback() {
    drainQueue();
    callback->Call(0, NULL);
  }

  void HandleProgressCallback(const char *data, size_t size) {
    drainQueue();
  }

 void HandleErrorCallback() {
    HandleScope scope;
    v8::Local<v8::Value> argv[] = {
      v8::Exception::Error(New<v8::String>(ErrorMessage())
        .ToLocalChecked())
    };
    error_callback->Call(1, argv);
  }
```

```
 protected:
  Callback *progress;
  Callback *error_callback;
  uint32_t n;
  PCQueue<Message> toNode;

  void send_factor(
      const AsyncProgressWorker::ExecutionProgress& progress,
      long long factor) {

        Message tosend("factor", std::to_string(factor));
        writeToNode(progress, tosend);
  }

  void writeToNode(
      const AsyncProgressWorker::ExecutionProgress& progress,
      Message & msg){

    toNode.write(msg);
    progress.Send(
        reinterpret_cast<const char*>(&toNode), sizeof(toNode));
  }


  void drainQueue() {
      HandleScope scope;
      // drain the queue - since we might only get
      // called once for many writes
      std::deque<Message> contents;
      toNode.readAll(contents);

      for(Message & msg : contents) {
          v8::Local<v8::Value> argv[] = {
            New<v8::String>(msg.name.c_str()).ToLocalChecked(),
            New<v8::String>(msg.data.c_str()).ToLocalChecked()
          };
          progress->Call(2, argv);
      }
  }
};

NAN_METHOD(Factor) {
  Callback *progress = new Callback(info[1].As<v8::Function>());
  Callback *callback = new Callback(info[2].As<v8::Function>());
  Callback *error_callback =
    new Callback(info[3].As<v8::Function>());
```

```
  AsyncQueueWorker(new Factorizer(
      callback
    , progress
    , error_callback
    , To<uint32_t>(info[0]).FromJust()
  ));
}
```

Building the JS adapter

Our final piece is to wrap up this addon in an EventEmitter. For now, it will be a simple JavaScript function in factorization.js, but later we'll adapt this all so it's reusable.

```
const EventEmitter = require('events');

var make_factorizer = function(n) {
    var addon_path = path.join(
        __dirname, "build/Release/factorization");

    const worker = require(addon_path);
    var emitter = new EventEmitter();
    worker.factorize(n,
        function () {
            emitter.emit("close");
        },
        function(event, value){
            emitter.emit(event, value);
        },
        function(error) {
            emitter.emit("error", error);
        });

    return emitter;
}
```

Notice that we are not returning the factorizer worker - only the emitter that has been used in the complete, progress, and error callbacks passed to the worker. Our main JavaScript code can now interact with the addon via this emitter:

```
var addon = make_factorizer(9007199254740991);
```

```
addon.on('factor', function(factor){
    console.log("Factor:  " + factor);
});

addon.on('close', function() {
    console.log("Factorization is complete");
})
addon.on('error', function(e) {
    console.log(e);
});
```

We'll get the same output as last time, now with a EventEmitter interface.

Generalizing Event-based Addons

Before moving forward with more examples, it will be helpful to see how we can generalize this structure to make creating additional addons simpler. Any addon based on this interface is going to have some common features:

1. It must accept a complete, progress, and error callback
2. It likely will need some initialization values (for factorization, N - the number to factor).
3. It will send data back to Node using the writeToNode method created above.

To make this all reusable, we'll create a header file (`streaming-worker.h`). We'll define utility classes (`PCQueue` and `Message`) within it. In addition, we'll create our base class `StreamingWorker`, which inherits `AsyncProgressWorker` and provides the common facilities, such as the `HandleOkCallback`, `HandleProgressCallback`, `HandleErrorCallback`, and `sendToNode` methods. Our individual addons will basically just inherit from `StreamingWorker`, overriding the `Execute` method and providing the appropriate constructor. Here is the code listing for `StreamingWorker` - however I've only written prototypes below for methods that are the same as in the implementation of the Factorization example. Utilities like `Message` and `PCQueue` are also omitted, but would be the same as before.

```
class StreamingWorker : public AsyncProgressWorker {
 public:
  StreamingWorker(Callback *progress,  Callback *callback,
    Callback *error_callback)
    : AsyncProgressWorker(callback), progress(progress),
        error_callback(error_callback)
    {
```

```
      // Notice no 4th parameter - initialization variables
      // will be handled by classes extending this.
    }

  // same as in factorization example
  ~StreamingWorker();

  // same as in factorization example
  void HandleErrorCallback();

  // same as in factorization example
  void HandleOKCallback() ;

  // same as in factorization example
  void HandleProgressCallback(const char *data, size_t size) ;

 protected:

  // same as in factorization example
  void writeToNode(
      const AsyncProgressWorker::ExecutionProgress& progress,
      Message & msg);

  // same as in factorization example
  void drainQueue();

  Callback *progress;
  Callback *error_callback;
  PCQueue<Message> toNode;
  bool input_closed;
};
```

This refactors our implementation of the original Factorization class. The code below has the same functionality as the previous implementation, however the 4th parameter for creating the addon is a JavaScript object instead of a plain integer - in order to allow other initialization parameters to be used. This isn't useful for factorization, but as we'll see in a moment, the constructor signature will remain constant across addons - so it pays to be a bit flexible.

```
class Factorization : public StreamingWorker {
  public:
    Factorization(Callback *data, Callback *complete,
      Callback *error_callback,  v8::Local<v8::Object> & options)
          : StreamingWorker(data, complete, error_callback){
```

```
        N = -1;
        if (options->IsObject() ) {
          v8::Local<v8::Value> n_ =
            options->Get(New<v8::String>("n").ToLocalChecked());

          if ( n_->IsNumber() ) {
            N = n_->NumberValue();
          }
        }

        if ( N < 0 ) {
          SetErrorMessage("Cannot compute prime factorization "
            + "of negative numbers (overflowed long long?)!");
        }
    }

    void send_factor(
        const AsyncProgressWorker::ExecutionProgress& progress,
        long long factor) {

        Message tosend("factor", std::to_string(factor));
        writeToNode(progress, tosend);
    }

    void Execute (
        const AsyncProgressWorker::ExecutionProgress& progress) {

      long long n = N;
      while (n%2 == 0)
      {
        send_factor(progress, 2);
        n = n/2;
      }

      for (long long i = 3; i <= n; i = i+2) {
          while (n%i == 0) {
            send_factor(progress, i);
            n = n/i;
          }
      }
    }
  private:
    long long N;
};
```

One additional issue with how we have the C++ setup is that the only "entry"

point we've provided for is the constructor - there is no facility to control the addon after it has been created. We'll fix that by actually wrapping our addon classes using `Nan::ObjectWrap`, which will allow us to treat our addon like a real object, with methods (to be added in later examples). This will also help facilitate generic (reusable) Node/Nan boilerplate.

The first step is to create a StreamingWorkerWrapper class. This class will wrap *any* StreamingWorker addon - we won't have to extend it in any way. This code uses all the concepts we saw in Chapters 5 and 6. Note that in it's current state there are no fields (methods) associated with the wrapper - but that will change shortly.

```
class StreamWorkerWrapper : public Nan::ObjectWrap {
 public:
  static NAN_MODULE_INIT(Init) {
    v8::Local<v8::FunctionTemplate> tpl =
        Nan::New<v8::FunctionTemplate>(New);
    tpl->SetClassName(
        Nan::New("StreamingWorker").ToLocalChecked());
    tpl->InstanceTemplate()->SetInternalFieldCount(0);

    constructor().Reset(
        Nan::GetFunction(tpl).ToLocalChecked());

    Nan::Set(target,
        Nan::New("StreamingWorker").ToLocalChecked(),
        Nan::GetFunction(tpl).ToLocalChecked());
  }

 private:
  explicit StreamWorkerWrapper(StreamingWorker * worker)
    : _worker(worker) {}
  ~StreamWorkerWrapper() {}

  static NAN_METHOD(New) {
    if (info.IsConstructCall()) {
      Callback *data_callback =
        new Callback(info[0].As<v8::Function>());
      Callback *complete_callback =
        new Callback(info[1].As<v8::Function>());
      Callback *error_callback =
        new Callback(info[2].As<v8::Function>());
      v8::Local<v8::Object> options = info[3].As<v8::Object>();

      StreamWorkerWrapper *obj = new StreamWorkerWrapper(
            create_worker(data_callback, complete_callback,
```

```
            error_callback, options));

      obj->Wrap(info.This());
      info.GetReturnValue().Set(info.This());

      // start the worker
      AsyncQueueWorker(obj->_worker);

    } else {
      const int argc = 3;
      v8::Local<v8::Value> argv[argc] = {
        info[0], info[1], info[2]
      };

      v8::Local<v8::Function> cons = Nan::New(constructor());
      info.GetReturnValue().Set(cons->NewInstance(argc, argv));
    }
  }

  static inline Nan::Persistent<v8::Function> & constructor() {
    static Nan::Persistent<v8::Function> my_constructor;
    return my_constructor;
  }

  StreamingWorker * _worker;
};
```

The principal point of extension in this wrapper is the call to `create_worker` within the constructor method. This method is where individual addons will be able to instantiate the actual implementation of `StreamingWorker`. The prototype of `create_worker` will be defined in `streaming-worker.h`, however the *implementation* will be within the addon's own source. The function passes the 3 callbacks plus the 4th argument, which will be an object containing initialization parameters.

Here's the `create_worker` implementation for the factorization addon (note, it's not a member method of Factorizer, it's standalone).

```
StreamingWorker * create_worker(
    Callback *data, Callback *complete ,
    Callback *error_callback, v8::Local<v8::Object> & options) {
 return new Factorization(data, complete, error_callback, options);
}
```

The only other thing each addon needs to do is expose the wrapper class to JavaScript using typical `nan` boilerplate.

Here's a breif review of the organization:

In `streaming-worker.h` - intended to be reused in each addon

1. Implementation of `PQueue`, `Message` classes
2. Implementation of `StreamingWorker` abstract class - each addon will extend this and implement a constructor and `Execute` method
3. Implementation of `StreamingWorkerWrapper` which will be exposed to JavaScript by each addon

In an addon's source code, we'd include `streaming-worker.h` and then...

1. Extend `StreamingWorker`
2. Provide an implementation of `create_worker` which `StreamingWorkerWrapper` will call
3. `NODE_MODULE(<addon name>, StreamWorkerWrapper::Init)` to expose the wrapper to Node.js. Note, each addon will have a different name.

With the C++ refactored, we have some changes to make to the JavaScript adapter. Instead of calling `factor` on a worker object returned by the `require` of the addon, now the addon returns the new worker object (wrapper) constructor. The rest of the JavaScript adapter is already highly reusable - there is nothing about the `make_factorizer` function above that is tied to factorization in any way in fact. All we need to do is make it parameterized so it can load any C++ addon.

We'll create a new module, in a separate file (`index.js` for now). It will export a single factory function, quite similar to `make_factorizer` from earlier, however it will accept the path to the C++ addon, and an object for initialization variables (as opposed to only a number). As a final adjustment, instead of only returning the emitter, we'll return an object containing the emitter, so we have room to extend functionality.

```
module.exports = function(cpp_entry_point, opts) {
    const factory = require(cpp_entry_point);
    var emitter = new EventEmitter();
    var worker = new factory.StreamingWorker(
        function () {
            emitter.emit("close");
        },
        function(event, value){
            emitter.emit(event, value);
        },
        function(error) {
            emitter.emit("error", error);
```

```
        }, opts);
    var retval = {
        from = emitter
    }
    return retval;
}
```

Now the individual JavaScript programs will require the JS adapter code in index.js. The factorization.js file from earlier would look something like this:

```
// this contains the JS adapter code from above
const streaming_worker = require("./index.js");

var addon_path = path.join(
    __dirname, "build/Release/factorization");
const factorizer = worker(addon_path, {n: 9007199254740991});

factorizer.from.on('factor', function(factor){
    console.log("Factor:  " + factor);
});

factorizer.from.on('error', function(e) {
    console.log(e);
});
```

Emitting JSON from C++

Now that we have a fairly easy way to create EventEmitter interfaces to addons, lets look at how we can send more detailed data from our addon to JavaScript - using objects.

You surely noticed that the `Message` class created in the previous section is designed to only carry strings as data. While the C++ programmers among us might be tempted to convert `Message` to a templatized class, remember that these messages are destined to become events over on the JavaScript side. That means that any data we wish to send must be converted to V8 data types before they are sent back. For complex C++ objects, you surely see how this can start to become difficult. In reality, most use cases would be satisfied by just being able to send a simple (no functions) JavaScript object back from C++. We can do this by sending stringified JSON - and parsing it when it arrives in our JavaScript code using `JSON.parse`.

How do you create stringified JSON in C++? Well, you could do it yourself - but there are some really excellent C++ libraries out there that will do the work for you. I highly recommend `json`, written by Niels Lohmann. You can obtain the source code at https://github.com/nlohmann/json.

If you've ever worked with sensor/tracking devices (for example, a head mounted display reporting position/orientation of user's head), you know that often their primary SDK is written in C++. For applications using these devices, you either have to write them in C++ or find a way to bridge the languages. For virtual reality, for example, one might find themselves using Node.js driving an electron app rendering 3D scenes with three.js. A C++ addon to emit tracking data is a great solution in this case. As an exercise, let us now build a simulated tracking/sensor addon that reports random x/y/z coordinates (we'll ignore orientation) at regular intervals. The addon can emit JSON strings, which we'll parse into regular objects once they get to JavaScript.

Step 1 - Build the C++ Addon (extending StreamingWorker)

The first step is of course creating `binding.gyp` and `package.json`. These files are essentially the same as in the factorization example above, so I won't repeat them here - just remember to change the addon `target_name` to `sensor_sim`. In addition, make sure you've added the directory where `streaming-worker.h` from the previous example can be found. You can do this by adding an entry to the `include_dirs` entry. We get down to business when we create the actual C++ code, let's put that in `sensor_sim.cpp`.

We start out by including some libraries - `streaming-worker.h` and now also `json.hpp`, which needs to be downloaded into the local directory (https://github.com/nlohmann/json). I'm also including some C++ additional headers so we can emit randomized position data at set intervals of time.

```
#include <iostream>
#include <chrono>
#include <random>
#include <thread>

// Make sure binding.gyp can find this!
#include "streaming-worker.h"

//download from https://github.com/nlohmann/json
#include "json.hpp"

using namespace std;
using json = nlohmann::json;
...
```

Now let's create the worker class, and the required `create_worker` factory function:

```
class Sensor : public StreamingWorker {
  public:
```

```
    Sensor(Callback *data, Callback *complete,
        Callback *error_callback)
          : StreamingWorker(data, complete, error_callback){

       // no extra options, but certainly tracker name,
       // perhaps position offsets could be passed in here..
    }

    void send_sample(
        const AsyncProgressWorker::ExecutionProgress& progress,
        double x, double y, double z) {

        json sample;
        sample["position"]["x"] = x;
        sample["position"]["y"] = y;
        sample["position"]["z"] = z;
        Message tosend("position_sample", sample.dump());
        writeToNode(progress, tosend);
    }

    void Execute (
        const AsyncProgressWorker::ExecutionProgress& progress) {

      std::random_device rd;
      std::uniform_real_distribution<double> pos_dist(-1.0, 1.0);
      while (true) {
        send_sample(progress,
            pos_dist(rd), pos_dist(rd), pos_dist(rd));

        std::this_thread::sleep_for(chrono::milliseconds(50));
      }
    }
};

StreamingWorker * create_worker(Callback *data
    , Callback *complete
    , Callback *error_callback
    , v8::Local<v8::Object> & options) {
 return new Sensor(data, complete, error_callback);
}

NODE_MODULE(sensor_sim, StreamWorkerWrapper::Init)
```

The interesting code is happening in `send_sample`, which is called from `Execute`. The `json` library allows you to build JSON in a very convenient, map-like API. Once we have the `sample` constructed, we call `dump()` to serialize to a string.

Step 2 - Using sensor in JavaScript

Next we need to look at the JavaScript side. Again, since we put in so much effort in the previous example to create a reusable module, the code is really quite simple. Here's the complete listing for instantiating our sensor addon and reporting to the console any position samples emitted.

```
"use strict";

const worker = require("streaming-worker");
const path = require("path");

const addon_path = path.join(__dirname, "build/Release/sensor_sim");
const sensor = worker(addon_path);

sensor.from.on('position_sample', function(sample){
    console.log(JSON.parse(sample));
});
```

Our sensor never actually stops emitting position data, so to stop it you'll need to do a `ctrl+c` (we'll fix this in a bit). The output should look something like this:

```
{ position:
   { x: -0.770311412681532,
     y: -0.770311412681532,
     z: -0.770311412681532 }}
{ position:
   { x: 0.456215735702525,
     y: 0.456215735702525,
     z: 0.456215735702525 }}
....
```

Streaming C++ output

The `EventEmitter` interface is probably a good choice for the two examples above - however there are times where you'd prefer to treat the data created as a stream, perhaps because you want it to get written to a (log) file. In fact, our sensor example might be a good candidate for this - sensor data is often logged to files or steamed to network sockets! Thankfully, the effort of turning event emitters into streams is pretty modest - we can utilize the `event-stream` to easily create streams from a proper `EventEmitter`. Unlike previous examples, making these changes is just a JavaScript effort (we'll place it in the `streaming-worker` JS module) - the C++ doesn't change.

Recall from above, we've create a reusable module that builds the `EventEmitter` around an addon specified by a file path:

```
// inside index.js (streaming-worker module)
module.exports = function(cpp_entry_point, opts) {
    const factory = require(cpp_entry_point);
    var emitter = new EventEmitter();
    var worker = new factory.StreamingWorker(
        function () {
            emitter.emit("close");
        },
        function(event, value){
            emitter.emit(event, value);
        },
        function(error) {
            emitter.emit("error", error);
        }, opts);
    var retval = {
        from = emitter
    }
    return retval;
}
```

Let's now add a function to the `from` object to turn it into a stream. We could do this inside the factory method itself, however it's probably wasteful to create streams that won't be used, so we'll make it an explicit function call by the user of the module.

```
const emitStream = require('emit-stream');
const through = require('through');

module.exports = function(cpp_entry_point, opts) {
    ... configure the emitter...
    var retval = {
        from = emitter
    }

    retval.from.stream = function() {
       return emitStream(retval.from).pipe(
         through(function (data) {
           if ( data[0] == "close"){
              this.end();
           }
           else {
              this.queue(data);
```

```
            }
        }));

    }

    return retval;
}
```

The `emit-stream` library exports a function `emitStream` which accepts an `EventEmitter` (in this case `retval.from`) and returns a readable stream. You can see the details of `emit-stream` at https://github.com/substack/emit-stream. One relevant detail is that events (name/value pairs) are converted to raw arrays.

We know that when a 'close' event gets emitted from our addon the stream itself should close, so we use `through` to capture those events and actually close the stream. Now whenever our addon emits a 'close' event, the output stream will properly close.

In JavaScript, we can now create the output stream and use it just like any other readable stream. For example, if we want the position data streamed to a file (positions.log), we just create a writable file stream. Since file streams don't accept raw JavaScript objects (such as the arrays being put onto the stream by `emit-stream`, we can use the `JSONStream` module to transform the streamed arrays into JSON strings.

```
const worker = require("streaming-worker");
const through = require('through');
const JSONStream = require('JSONStream');
const path = require("path");

const addon_path = path.join(
    __dirname, "build/Release/sensor_sim");

const sensor = worker(addon_path);

var fs = require('fs');
var wstream = fs.createWriteStream('positions.log');
const out = sensor.from.stream();
out.pipe(JSONStream.stringify()).pipe(wstream);
```

Stopping a streaming addon

You'll notice that the output in positions.log isn't pretty - the whole file should be a JSON stringified array - however it's incomplete, missing the closing `]`. That's because our sensor addon never stops (I assume you stopped it with ctrl+c), and the JSON stream is never properly terminated. We need a way

to stop an addon - so let's take advantage of our `StreamingWorkerWrapper` to build in a `close` function, which sets a flag in our `StreamingWorker` (or in this case, `Sensor`). Inside `streaming-worker`, let's add the boolean flag first:

```
class StreamingWorker : public AsyncProgressWorker {
 public:
    ...

    void close() {
      closed = true;
    }

    ...
 protected:
    bool isClosed() {
      return closed;
    }

 private:
    bool closed;
```

We want to allow JavaScript to call `close`, so let's add that to the wrapper.

```
class StreamWorkerWrapper : public Nan::ObjectWrap {
 public:
  static NAN_MODULE_INIT(Init) {
    v8::Local<v8::FunctionTemplate> tpl =
        Nan::New<v8::FunctionTemplate>(New);
    tpl->SetClassName(
        Nan::New("StreamingWorker").ToLocalChecked());
    tpl->InstanceTemplate()->SetInternalFieldCount(1);

    SetPrototypeMethod(tpl, "close", closeInput);

    constructor().Reset(
        Nan::GetFunction(tpl).ToLocalChecked());

    Nan::Set(target,
        Nan::New("StreamingWorker").ToLocalChecked(),
        Nan::GetFunction(tpl).ToLocalChecked());
  }
  ... rest omitted, it's the same as before ...
};
```

```
static NAN_METHOD(closeInput) {
    StreamWorkerWrapper* obj =
        Nan::ObjectWrap::Unwrap<StreamWorkerWrapper>(info.Holder());
    obj->_worker->close();
}
```

Now we can alter our `Execute` function in `Sensor` to stop when `isClosed` returns true:

```
void Execute (
    const AsyncProgressWorker::ExecutionProgress& progress) {

  std::random_device rd;
  std::uniform_real_distribution<double> pos_dist(-1.0, 1.0);
  while (!closed()) {
    send_sample(progress,
        pos_dist(rd), pos_dist(rd), pos_dist(rd));

    std::this_thread::sleep_for(chrono::milliseconds(50));
  }
}
```

Let's also put a `close` method in the object returned by the factory method in the `streaming-worker` module:

```
....

retval.close = function (){
   worker.closeInput();
}

return retval;
```

Finally, let's close the sensor after 5 seconds in our JavaScript program:

```
const worker = require("streaming-worker");
const through = require('through');
const JSONStream = require('JSONStream');
const path = require("path");

const addon_path = path.join(__dirname, "build/Release/sensor_sim");
const sensor = worker(addon_path);

var fs = require('fs');
```

```
var wstream = fs.createWriteStream('positions.log');
const out = sensor.from.stream();
out.pipe(JSONStream.stringify()).pipe(wstream);

setTimeout(function(){sensor.close()}, 5000);
```

Now positions.log will be a nicely formatted JSON array. Of course, we could use `through` to further transform the output sent to this file - the sky is the limit!

As a side note, C++ has streams too... and you might be wondering if you could change the C++ addon interface to do something like this:

```
toNode << msg;
```

Instead of using the `writeToNode` method. The answer is - of course! To do so requires extending C++ i/o stream and stream buffer classes, which is a big topic in it's own right. I'll leave that as a C++ exercise for the ambitious reader.

Emitting events to C++

So far we've created addons that either take no input (sensor) or use only initialization variables to get information from JavaScript (factoring). What if we want to move lots of data, over time, to our addon? We can reverse the event and streaming API's and emit events/stream data from JavaScript *to C++* as well. We can leverage a lot of the work we've already done in fact (`Message` and `PCQueue`, along with `StreamingWorker` and `StreamingWorkerWrapper`).

Lets start by building the EventEmitter API. The most difficult part to this is the fact that we can no longer borrow functionality from `nan`'s `AsyncProgressWorker`, as it only supports sending periodic progress *from C++*, not to it. Our first step is to build the ability to move a message from JavaScript to C++, and we'll do this by exposing a method in our `StreamingWorkerWrapper` which adds a message to *another* queue inside `StreamingWorker` which will now be used to hold data flowing into the addon.

Here is the only change to `StreamingWorker` - adding a new queue object.

```
class StreamingWorker : public AsyncProgressWorker {
  public:
    ... most is exactly the same...

    PCQueue<Message> fromNode;

    ....
};
```

Purists would likely rather make this queue private and create an accessor, which is fine too. Here's the wrapper code, which now has 2 internal fields.

```
class StreamWorkerWrapper : public Nan::ObjectWrap {
 public:
  static NAN_MODULE_INIT(Init) {
    v8::Local<v8::FunctionTemplate> tpl =
        Nan::New<v8::FunctionTemplate>(New);
    tpl->SetClassName(
        Nan::New("StreamingWorker").ToLocalChecked());
    tpl->InstanceTemplate()->SetInternalFieldCount(1);

    SetPrototypeMethod(tpl, "close", closeInput);
    SetPrototypeMethod(tpl, "sendToAddon", sendToAddon);

    constructor().Reset(
        Nan::GetFunction(tpl).ToLocalChecked());
    Nan::Set(target,
        Nan::New("StreamingWorker").ToLocalChecked(),
        Nan::GetFunction(tpl).ToLocalChecked());
  }
  ... rest omitted, it's the same as before ...
};

static NAN_METHOD(sendToAddon) {
    v8::String::Utf8Value name(info[0]->ToString());
    v8::String::Utf8Value data(info[1]->ToString());
    StreamWorkerWrapper* obj =
        Nan::ObjectWrap::Unwrap<StreamWorkerWrapper>(info.Holder());
    obj->_worker->fromNode.write(Message(*name, *data));
  }
```

Let's now build another example that uses this new functionality - a C++ accumulator. For this example, our C++ addon will collect data sent from JavaScript and when a sentinel value is received (-1) it will emit a `sum` event with the sum of all numbers sent to it. We've been through building out new addons now a few times, so I'll only present the Accumulator class.

```
class Accumulate : public StreamingWorker {
  public:
    Accumulate(Callback *data, Callback *complete,
        Callback *error_callback,
        v8::Local<v8::Object> & options)
        : StreamingWorker(data, complete, error_callback){
```

```
        sum = 0;
        filter = "";
        if (options->IsObject() ) {
          v8::Local<v8::Value> filter_ = options->Get(
              New<v8::String>("filter").ToLocalChecked());

          if ( filter_->IsString() ) {
            v8::String::Utf8Value s(filter_);
            filter = *s;
          }
        }
    }
    ~Accumulate(){}

    bool filter_by_name(string name) {
      return ( filter.empty() || name == filter);
    }

    void Execute (const AsyncProgressWorker::ExecutionProgress& progress) {
      int value ;
      do {
        Message m = fromNode.read();
        value = std::stoi(m.data);
        if ( filter_by_name(m.name) || value <= 0) {
          if ( value > 0 ){
            sum += value;
          }
          else {
            Message tosend("sum", std::to_string(sum));
            writeToNode(progress, tosend);
          }
        }
      } while (value > 0);
    }
  private:
    int sum;
    string filter;
};
```

Note I've also added an optional filter to this class, so it will only accumulate events with a name that matches it's filter. If no filter is specified in the initialization object then all events will be accumulated. The accumulator's `Execute` function just sits in a loop and reads individual messages from the `fromNode` queue. Recall, the `read` method on `PCQueue` is blocking.

Now let's build a simple JavaScript program to send a few numbers to the addon.

We need to go into our `streaming-worker` module and expose a method to actually invoke the `sendToAddon` method we've added.

```
const emitStream = require('emit-stream');
const through = require('through');

module.exports = function(cpp_entry_point, opts) {
    const factory = require(cpp_entry_point);
    var emitter = new EventEmitter();
    var worker = new factory.StreamingWorker(
        function () {
            emitter.emit("close");
        },
        function(event, value){
            emitter.emit(event, value);
        },
        function(error) {
            emitter.emit("error", error);
        }, opts);

    var retval = {
        from = emitter
    }

    retval.from.stream = function() {
       return emitStream(retval.from).pipe(
         through(function (data) {
           if ( data[0] == "close") this.end();
           else this.queue(data);
        }));
    }

    ////////////////////////////
    // Create a to object to similate
    // an emitter
    retval.to = {
      emit: function(name, data) {
         worker.sendToAddon(name, data);
      }
    return retval;
}
```

The `to` object we've added to the factory's return object isn't an actual `EventEmitter`, if you are looking to improve this you might consider adapting this code to truly create an `EventEmitter` which overrides `emit` and `close`

to communicate with the addon. The implementation shown above is "good enough" for the purposes of these examples however.

Now in our actual JavaScript program we can utilize this new `to` object:

```
"use strict";

const worker = require("streaming-worker");
const path = require("path");

var addon_path = path.join(__dirname, "build/Release/accumulate");
const acc = worker(addon_path);

acc.to.emit("value", 3);
acc.to.emit("value", 16);
acc.to.emit("value", 42);
acc.to.emit("value", -1);

acc.from.on('sum', function(value){
    console.log("Accumulated Sum:  " + value);
});
```

When run, you'll get the expected answer of 61.

Streaming input to C++

It should be clear that now that we have an `EventEmitter`-like interface, we can use `emit-stream` to turn it into a writable stream. Once again, this is a JavaScript exercise, our C++ addons don't really care whether the JavaScript uses a streaming API, they only read from the `fromNode` queue [8] . Where creating a readable stream from an event emitter for sending data to JavaScript was pretty trivial, for the opposite direction we do need to consider what happens when our input stream closes. Each addon is likely to want to be notified that the stream has closed differently (for example, the accumulator detects -1 as a sentinel but other addons could use a 'close' message, or a different sentinel value). To allow for this flexibility, we'll bake a parameterized callback into the function we expose to create the input stream, allowing each program to decide what to do when the input stream closes. So, back into the `streaming-worker` module:

```
....
```

[8] As with sending data out of C++, a nice enhancement in C++ would be to create new input stream objects that extract messages from the `fromNode` queue, allowing an addon to do things like `fromNode >> msg;`. This is a purely C++ exercise though, and is left to the reader.

```
    ////////////////////////////
    // Create a to object to similate
    // an emitter
    retval.to = {
      emit: function(name, data) {
         worker.sendToAddon(name, data);
      }

      stream : function(name, end) {
         var input = through(function write(data) {
            if (Array.isArray(data)) {
               if ( data[0] == "close"){
                  this.end();
               }
               else {
                  retval.to.emit(data[0], data[1]);
               }
            }
            else {
               retval.to.emit(name, data);
            }
          }, end);
        return input;
      }
    return retval;
}
```

The `stream` function builds a `through` stream, which requires us to specify a `write` and optional `end` callback. The `write` function accepts data sent to the stream and calls the `emit` function we already created (rather than queuing it to the actual stream, which is typical). Notice that the `write` callback looks out for a `close` message and properly closes the input stream. It also handles arrays (`[name, value]`) rather than only objects (`{name:name, value:value}`);

Now we can revise accumulate to use a stream as input:

```
"use strict";

const worker = require("streaming-worker");
const path = require("path");
const streamify = require('stream-array');

var addon_path = path.join(__dirname, "build/Release/accumulate");
const acc = worker(addon_path);

const input = acc.to.stream("value",
```

```
    function () {
        acc.to.emit('value', -1);
    });

streamify([1, 2, 3, 4, 5, 6]).pipe(input);

acc.from.on('sum', function(value){
    console.log("Accumulated Sum:  " + value);
});
```

Summary

The goal of this chapter was to demonstrate how we are not locked into the basic interfaces provided by typical C++ addons. The `streaming-worker` and `StreamingWorker` models are meant as examples, there is considerable room for improvement. If you'd like to use the model presented in this chapter (or extend or contribute to it), please take a look at the full repository - https://github.com/freezer333/nodecpp-demo. The streaming models are in the `/streaming` directory.

Now that we've gone through so much trouble to create addons that can be plugged into Node.js applications in so many ways, we should turn our attention towards packaging and publishing them so the world can benefit... which is the subject of the next chapter!

Chapter 8 - Publishing Addons

As the final chapter in this book, we'll take a look at some of the basics of packaging and publishing Node.js addons. While much of the workflow follows the same patterns as publishing normal Node.js modules (JavaScript), there are some particulars - relating to compiling C++ code - that we must be aware of. As a first step, let's review some of the important aspects of our build tool of choice - `node-gyp` - and see how to publish these addons. We'll then take a look at how to ensure our addons compile and install on a wide variety of end-user machines. Finally, we'll take a look at some of the alternatives to `node-gyp`.

The code above in this chapter is available in full in the `nodecpp-demo` repository at https://github.com/freezer333/nodecpp-demo, under the "Packaging" section.

Review of `node-gyp` basics

Throughout this book we've built all our addons using `node-gyp`, which is a Node.js program that wraps the `gyp` build tool system created by Google. `gyp` itself (Generate your own project) is geared towards allowing developers to setup projects/dependencies in a platform agnostic way - rather than maintaining project files for Visual Studio (projects) and Linux/OS X toolchains. The `node` in `node-gyp` is what builds in the support for setting up projects that output actual Node.js addons.

If you've made it this far, hopefully you've built a few (at least!) of the addons described in previous chapters - which means not only have you installed `node-gyp`, but you've also installed the proper dependencies for your platform.

1. Python 2.7+, but **not Python 3.x**. `node-gyp` uses Python for part of it's build process, but it is not compatible with Python 3 - you must have a 2x version installed on your machine. They can be installed side-by-side, so if you want Python3 too - it's not a problem.
2. Build tools for your platform - if you are on Linux/Mac OX you'll need `make` and `clang++`. On OS X, these are installed when you install XCode. On Windows, you need Visual Studio installed on your system.

Once you have those setup (and on your path), you can install `node-gyp` using `npm`. Be sure to install it globally!

```
$ npm install node-gyp -g
```

All our addons have a `binding.gyp` file, and we've already seen some rather complex ones. At it's core, the `binding.gyp` file simply specifies a target name and a list of C++ source code files to build.

```
{
  "targets": [
    {
      "target_name": "hello_addon",
      "sources": [ "hello.cpp" ]
    }
  ]
}
```

`node-gyp` can be used to simply create project build files (for the given platform) by issuing `node-gyp configure`. On Windows, this would create the necessary `.vsprojx` file for Visual Studio to compile the addon - on OS X or Linux it will create a `Makefile`. Issuing a `node-gyp build` will run the compiler and generate the addon. You of course can do both at the same time.

```
$ node-gyp configure build
```

We've been through this before, but what if we want to distribute our addon? We'll start with a really simple addon - one using no C++ library dependencies, standard (pre C++11) C++, and no platform specific API calls.

```
#include <node.h>

using namespace v8;

void Method(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();
  Local<String>  retval = String::NewFromUtf8(isolate, "hello");
  args.GetReturnValue().Set(retval);
}

void init(Local<Object> exports) {
  NODE_SET_METHOD(exports, "hello", Method);
}

NODE_MODULE(hello_addon, init)
```

Up until this point in the book, the next step has always been to immediately begin to create a JavaScript source file that started to use the addon. Let's take a step back however - addons should be *reusable* and *distributable* - right? So before writing JavaScript that calls the addon, let's package the addon up so we can treat it as a **standalone** module. To do that, we need a `package.json` file of course!

Let's create a real simple one:

```
{
  "name": "hello-world-nodecpp",
  "version": "1.0.0",
  "main": "./build/Release/hello_addon",
  "gypfile": true,
  "author": "Scott Frees <scott.frees@gmail.com> (http://scottfrees.com/)",
  "license": "ISC"
}
```

Notice the inclusion of `gypfile` however - this is important - it tells anything consuming the `package.json` file (such as `npm`) that there is a build process that must be performed before using the module. Notice also the entry for `main`, it is pointing `npm` to the actual location where the addon will be created - which is where we've been hardcoding `require` statements to throughout this book.

Currently there should be three files in a directory called `hello`:

```
/hello
 |
 |--- hello.cpp
 |--- binding.gyp
 |--- package.json
```

This is all we need to have a standalone module. Let's now create a new directory (at the same level as `/hello` called `/demo` and create a Node.js program that specifies the `hello-world-nodecpp` module we've created as a dependency. Instead of hardcoding filepaths in our JavaScript program, we'll utilize `npm`'s ability to use local module dependencies.

Let's create our directory and add an `index.js` and another `package.json` file for the demo program:

```
/hello
 |
 |--- hello.cpp
 |--- binding.gyp
 |--- package.json
/demo
 |
 |--- index.js
 |--- package.json
```

Inside `/demo/package.json` we will add a local dependency (supported by `npm` versions 2+:

```
{
  "name": "demo",
  "version": "1.0.0",
  "main": "index.js,
  "author": "Scott Frees <scott.frees@gmail.com> (http://scottfrees.com/)",
  "license": "ISC",
  "dependencies": {
    "hello-world-nodecpp": "file:../hello"
  }
}
```

Here's how we'd use the module in `index.js`:

```
var say = require('hello-world-nodecpp');
console.log( say.hello() );
```

While within the `demo` directory, let's do an `npm install`. `npm` will interrogate the `package.json` file and see the local `hello-world-nodecpp` dependency. `npm` will then read `/hello/package.json` and see that there is a `gypfile` flag, and build the addon. Note, this process will actually put the addon binary within `/demo/node_modules` directory. Now that the addon is built and copied into `/node_modules`, we can run the program with `node index.js` and we'll see our addon happily greets us.

```
$ node index.js
hello
$
```

The key takeaway from this segment is the creation of the `package.json` file which directly points to the addon at it's entry point. When consuming (`require`ing) programs declare the addon as a dependency, `npm` takes care of the rest (executing `node-gyp`). Our addon can now be required cleanly, without specifying file paths in our JavaScript code.

Publishing to `npm`

While we used `hello-world-nodecpp` as a local dependency above, there is nothing stopping us from publishing this addon now to the global `npm` registry. Let's go ahead and pollute the registry with yet another hello world module!

```
# From within `/hello`
$ npm publish
+ hello-world-nodecpp@1.0.0
```

The `npm publish` command requires some parameters from you, such as your account with `npmjs.org` credentials. If youv'e never published anything to `npm`, then you might have some configuration to do (see https://docs.npmjs.com/getting-started/publishing-npm-packages for help setting this up). You can head over to https://www.npmjs.com/package/hello-world-nodecpp to see the published result.

Now of course, we can change our `demo/package.json` to link to the npm registry itself.

```
{
  "name": "demo",
  "version": "1.0.0",
  "main": "index.js,
  "author": "Scott Frees <scott.frees@gmail.com> (http://scottfrees.com/)",
  "license": "ISC",
  "dependencies": {
    "hello-world-nodecpp": "^1.0.0"
  }
}
```

All the standard version conventions inherent to npm package management apply to addons - there is nothing new to learn. Now anytime anyone links to `hello-world-nodecpp` the C++ source code will be downloaded to their machines, built with `node-gyp` and their local C++ toolchain, and be usable as a native binary.

Distributing addons that use NAN

Most larger addons will likely make use of NAN to facilitate cross-node-version compatibility, which makes even more sense if you plan on distributing addons. Thankfully, declaring NAN as a dependency is fairly straightforward - we must indicate the dependency in two places - (1) our `binding.gyp` file so the compiler can use NAN and (2) our `package.json` file, which will instruct `npm` to fetch the NAN library on install.

Here is the C++ addon - a slight variant on the previous hello world example:

```
#include <nan.h>
using namespace Nan;
using namespace v8;

NAN_METHOD(Method) {
  info.GetReturnValue().Set(
    New<String>("nan hello").ToLocalChecked());
```

```
}

NAN_MODULE_INIT(Init) {
   Nan::Set(target,
    New<String>("hello").ToLocalChecked(),
    GetFunction(New<FunctionTemplate>(Method)).ToLocalChecked());
}

NODE_MODULE(hello_nan_addon, Init)
```

In Chapter 6 we saw how to include NAN in the `binding.gyp` file - here it is again:

```
{
  "targets": [
    {
      "target_name": "hello_nan_addon",
      "sources": [ "hello_nan.cpp" ],
      "include_dirs" : [
        "<!(node -e \"require('nan')\")"
      ]
    }
  ]
}
```

Finally, we add `nan` to the module's `package.json` file:

```
{
  "name": "hello-world-nan-nodecpp",
  "version": "1.0.0",
  "main": "./build/Release/hello_nan_addon",
  "gypfile": true,
  "author": "Scott Frees <scott.frees@gmail.com> (http://scottfrees.com/)",
  "license": "ISC",
  "dependencies": {
    "nan": "^2.3.3"
  }
}
```

Instead of further polluting the npm registry, let's just add a local dependency to our `/demo` program and utilize this addon also.

```
{
  "name": "demo",
```

```
  "version": "1.0.0",
  "main": "index.js",
  "author": "Scott Frees <scott.frees@gmail.com> (http://scottfrees.com/)",
  "license": "ISC",
  "dependencies": {
    "hello-world-nodecpp": "^1.0.0",
    "hello-world-nan-nodecpp": "file:../hellonan"
  }
}
```

```
var say = require('hello-world-nodecpp');
var nansay = require('hello-world-nan-nodecpp');
console.log( say.hello() );
console.log( nansay.hello() );
```

From the `/demo` directory, do another `npm install` to build the addon, and you are all set:

```
# From within `/demo`
$ npm install
.... will build the new addon...
$ node index.js
hello
nan hello
```

Distributing addons that use C++11

So far we've used basic C++ in our addons, but at many points in this book we utilized some features of C++ that are part of the C++ 11 (or 14, 17..) standard. The C++11 standard (especially) is common place, however many compilers require special incantations to enable it - which makes configuring `node-gyp` correctly for each platform pretty critical. Of course, the target machine where your add on is being installed must have a C++ toolchain that supports C++ 11 to make things work, but let's now look at what else we need to do. While there are undoubtedly many more possible configurations and options, I'll present the additions to `binding.gyp` that I've found to cover the most common platform configurations.

C++11 example

Let's start by creating a needlessly complicated `Add` function that utilizes a `lambda` and `auto`. I stress that this is a needlessly complicated way of adding two numbers - but it certainly requires C++11! I'll also continue to use NAN.

```
#include <nan.h>
using namespace Nan;

NAN_METHOD(Add) {

    auto sum = [](int x, int y) { return x + y; };

    int a = To<int>(info[0]).FromJust();
    int b = To<int>(info[1]).FromJust();

    Local<Number> retval = Nan::New(sum(a, b));
    info.GetReturnValue().Set(retval);
}

NAN_MODULE_INIT(Init) {
   Nan::Set(target, New<String>("add").ToLocalChecked(),
        GetFunction(New<FunctionTemplate>(Add)).ToLocalChecked());
}

NODE_MODULE(cpp11, Init)
```

Assuming we also create a standard `binding.gyp` and `package.json` file for this addon, let's add this to `/demo` and try to install. If you are compiling on OS X or Linux with the latest compilers, it's fairly likely that this code will indeed compile just fine. If you are using versions of Xcode or clang/g++ that are a bit old however, it's likely you will need to take some additional steps.

Building on Linux

For backwards compatibility, most versions of g++ and clang++ require c++11 to be specifically targeted. Generally I simply add a general entry under `cflags` in `binding.gyp` to ensure the standard is used. You could elect to put this in a condition specifically targeting Linux, however since it doesn't do any harm on any other platforms, I recommend simply adding it directly

```
"cflags": ["-std=c++11"],
```

You of course can add any number of additional flags here as well, it is an array.

Building on OS X

For users running OS X, you won't be able to count on them having the latest version of Xcode installed. For earlier versions of Xcode, C++ 11 and the standard library must be specifically included. `gyp` allows us to add options

specific for OS X as a conditional - simply include the following to allow a wide range of Xcode versions to successfully compiler your addon:

```
"conditions": [
  [ 'OS=="mac"', {
       "xcode_settings": {
         'OTHER_CPLUSPLUSFLAGS' : ['-std=c++11','-stdlib=libc++'],
         'OTHER_LDFLAGS': ['-stdlib=libc++'],
         'MACOSX_DEPLOYMENT_TARGET': '10.7' }
        }
  ]
]
```

Building on Windows

Happily, there isn't much for us to do for Windows. While we can certainly pass Visual Studio compiler flags in through `gyp` (`msvs_settings`), each version of Visual Studio will automatically apply the latest C++ features that it supports. Of course, if the version of Visual Studio installed on the target machine does not support the specific C++ feature you are using in your addon, there is little that `gyp` can do to help.

Rule of thumb: When developing addons, realize they will be *compiled* on the end-users' machine - needless use of advance language features might be fun, but it can haunt you!

Including multiple C++ files

Most of the addons in this book have been contained within one single C++ file. Many addons will be built from potentially dozens of C++ files, and this is fully *supported* by `node-gyp`, however be warned - if you have a complex build/dependency setup you may find `node-gyp` a bit uncooperative, it supports simple builds well, but can get tedious and difficult quickly. If you have a complex C++ build setup, you are likely using `cmake` or similar, and might find `node-gyp` frustrating - see the section below on `cmake-js` for some refuge.

Let's create a simple addon that adds two numbers passed from Node.js, but does this by utilizing a `sum` function that is contained in a separate C++ module (cpp file and header). Again - needlessly complicated, for demonstration purposes:

```
// contents of add.h
int sum(int, int);
```

```
// contents of add.cpp
#include "add.h"

int sum(int x, int y ) {
    return x+y;
}


// contents of addlib.cpp - entry point for addon
#include <nan.h>
#include "add.h"
using namespace Nan;
using namespace v8;

NAN_METHOD(Add) {
    int a = To<int>(info[0]).FromJust();
    int b = To<int>(info[1]).FromJust();

    Local<Number> retval = Nan::New(sum(a, b));
    info.GetReturnValue().Set(retval);
}

NAN_MODULE_INIT(Init) {
   Nan::Set(target, New<String>("add").ToLocalChecked(),
        GetFunction(New<FunctionTemplate>(Add)).ToLocalChecked());
}

NODE_MODULE(cpp11, Init)
```

With additional source code file, we just need to add the files to the `sources` array in the `binding.gyp` file:

```
{
  "targets": [
    {
      "target_name": "addlib",
      "sources": [ "add.cpp", "addlib.cpp" ],
      "include_dirs" : [
        "<!(node -e \"require('nan')\")"
      ]
    }
  ]
}
```

When publishing addons, all C++ source code that is needed to compile the addon must be published. In the example above, this happens automatically -

however if your addon has dependencies outside the working directory you can run into trouble. It's wise to brush up on how `npm` determines which files are part of the published package, which is described in `npm`'s documentation here: https://docs.npmjs.com/misc/developers#keeping-files-out-of-your-package.

Pre-compiled Addons

It's important to note that everything above is predicated on the fact that you intend to have your users "compile" the addon when they do an `npm install`. In most cases, this makes sense - Node.js is usually used in programs that are not meant to *directly* run on end users machines - in most cases your addon will be run on a machine that you can assume a developer has access to and can install the prerequisite toolchain on. Node.js is expanding though, and especially with the advent of desktop applications being developed with NW and electron, you could very well end up seeing your addon being used on "grandma's" computer too - and it's unlikely grandma has a C++ toolchain installed!

If you want to publish addons in pre-built form you must consider usage on Windows, Linux, and OS X. While you can role your own solution to this, there exist some helpful tools to simplify the process. `node-pre-gyp` is one such tool, which can be installed on the developer's machine and facilitates building and installing binary dependencies. While out of the scope of this book, it's worth taking a look at if you plan to create addons that may have non-technical users.

Node-gyp Alternative: cmake-js

`node-gyp` is built on Google's `gyp` tool. It hasn't always been the method of choice when creating addons, in earlier (pre v0.8) versions of Node a program called `node-waf` was used, and required different build files. While there is no reason to believe `node-gyp` is going away soon (as of mid-2016), there are no guarantees in life. Furthermore, `gyp` leaves a lot to be desired for complex C++ builds - not the least of which is documentation (`gyp` has documentation, but it's nowhere near as comprehensive as other C++ build tools). For these reasons, seasoned C++ developers wishing to support complex build systems may prefer alternatives to `node-gyp`.

The leading alternative is `cmake-js`. As it's name suggests, it uses the immensely popular and ubiquitous `CMake` build tool. The advantage of `cmake-js` is that rather than using `binding.gyp`, it uses `CMake` configuration files and is likely compatible with existing C++ developer workflows. `CMake` has it's own modules system, extremely comprehensive documentation, and is not dependent on Python. Perhaps the biggest advantage is that it makes including the Boost C++ library as a dependency to your addon fairly straightforward - something that is *extremely* difficult to pull off using `node-gyp`.

CMake isn't the focus of this book, but if you are an experienced CMake user or C++ developer it is likely a really good idea to invest some time learning more about CMake and `cmake-js`. It's a solid alternative to `node-gyp`.

Information about the tools can be found at the following links:

1. `cmake-js` on npm: https://www.npmjs.com/package/cmake-js
2. `cmake-js`'s wiki: https://github.com/cmake-js/cmake-js/wiki Contains tutorials for building addons with a variety of dependencies and targets (NW.js, Qt Creator)
3. `boost-lib` on npm: https://www.npmjs.com/package/boost-lib Utility for adding Boost to your Node.js C++ addons.

There are many more resources online as well. Be sure to check out Appendix A as well, were you'll see how to use `node-gyp` to build a variety of different target types such as executables and shared libraries.

Appendix A - Alternatives to Addons

Note: This appendix is largely taken from a series of blog posts on the author's website, blog.scottfrees.com, which covers the various ways one might choose to integrate an ***existing, legacy C/C++ program*** *into a Node.js web app. The series is standalone from this book, and as such, you might find some of this material to be redundant, however it illustrates a real world example of integrating legacy C++ code as opposed to the narrowly scoped examples in the book. The section also has a different github repository to download the code from than the other examples throughout this book. It is highly recommended that you do grab the code, as there is a lot of it that is not presented in the text!*

This book has focused 100% on Node.js C++ addons as the method of choice for integrating Node.js and C++. In many cases, addons are indeed the best alternative for doing this sort of integration, however there are other options - and this book would be incomplete without discussing them. Before doing so, let's take a step back and re-ask the question asked way back in Chapter 1: *Why integrate Node.js and C++*? To sharpen the discussion, **let's look at this from the perspective of having an existing C++/C program that you wish to get onto the web**.

Can't I just write a C++ web site?

Well... yes - you could! People have been writing parts of web applications in C++ for a very long time - using CGI. CGI isn't the most popular thing on the web these days though, it lacks a lot of productivity enhancements that makes web development so great today. More importantly, it introduces some significant performance and scalability issues. On the other hand, C++ has come a long way over the past few years in terms of expressiveness, and the C++14 standard has enabled a some really cool projects focused on writing modern MVC-styled web apps in pure C++. If that's your thing, check out Silicon.

The majority of web developers aren't C++ programmers, and frankly, unless ultra high performance from your web tier is critical, you are probably better off using languages providing a higher level of abstraction. The common players on the web being Ruby, Go, Node.js, Python, PHP along with many more...

Why Node.js?

Node.js has a number of advantages. For one, it **integrates** really nicely with C++ in several different ways - one of which of course we've seen throughout this book! In general, Node.js also has a lot of benefits that dovetail with why you'd be using C++ in the first place - it's highly portable, it promotes performance at scale, and has a thriving ecosystem.

Why not just rewrite the C++?

Ah... every developer's first instinct - "Let's rewrite this old code written in language X because language Y is so much cooler | better | faster | easier!". First

off - if you have some legacy C++ code that is simple, small, and doesn't really need to be high performance - this might very well be the best answer. However, if you are in that category, you probably aren't reading this - you're likely almost done rewriting your C++ code.

There are a few practical reasons not to rewrite code. **First**, you might not have the code! Believe it or not, if you are working for a company that uses legacy tools to support their business, source code for these tools are often lost to time. A derivative of this is when your legacy code uses third-party dependencies, which cannot be rewritten or modified.

Second, C/C++ can be complex - and if it's old, it might be really hard to decipher. Are you a web developer that is also a C++ guru? Are you *positive* you can completely recreate the precise inputs/outputs of this program? If it's a critical line of business tool, you are putting a **ton** of risk on your plate.

The **third reason** not to rewrite your C++ is because it might really want to be in C++! While Node.js has decent performance, it's simply not C/C++. If your application has extreme performance criteria, you aren't going to beat C++.

C++ and Node.js - your options

There are three general ways of integrating C++ code with a Node.js application - although there are lots of variations within each category:

1. **Automation** - call your C++ as a standalone app in a child process.
2. **Shared library** - pack your C++ routines in a shared library (dll) and call those routines from Node.js directly.
3. **Node.js Addon** - compile your C++ code as a native Node.js module/addon (*we know all about this now, right?)

Each of these options have their advantages and disadvantages, they primarily differ in the degree in which you need to modify your C++, the performance hit you are willing to take when calling C++, and your familiarity / comfort in dealing with Node.js and the V8 API.

How to choose

The most obvious question to ask first is **do you have access to the C++ source**, *or just the binary?* Without source code, you need to hope the C++ program is either a command line program or a dll/lib shared library. If you are looking at a program written with only a graphical user interface... well then you are in a world of pain. Its likely you are going to need to rewrite your application in order to make it work on the web.

Option 1 - Automation

If your C++ runs as a standalone from a command line, you don't need the source code to take advantage of Option 1 - the **automation option**. You can run your C++ program unaltered, using Node's child process API. This option works for bringing just about anything to the web - it really doesn't make a difference what language your command line program is written in if you are simply running it. If you are reading this hoping to get C code, Fortran code, or some other language onto the web - then this option is worth reading.

The automation option is not only for those *without* the underlying C++ code. If you have C++ code that either is currently, or could easily be turned into, a command line program - then this option is reasonable if you can live with the performance, and you don't really want to get into the hassles of language integration.

Option 2 - Shared Library / DLL

If you are dealing with a C++ dll/lib, or you have the C++ source code and can make modest modifications in order to create a shared library, then the **shared library** approach might work well for you. We'll detail how you can do this using the Foreign Function Interface module in this chapter. This option gives you more fine-grain control of how you integrate C++ into Node, because calls to C++ routines can be interleaved with Node.js code itself. While this approach brings you closer to full integration, you still have to deal with type conversions and blocking when calling C++. It's a great option if you want better integration, without investing a lot of time dealing with V8.

Option 3 - Node.js Addon

If you have the C++ source code, then a third option is creating a native Node.js module out of your C++. While this is the more challenging approach, you gain a ton of flexibility and performance. You also have the option to call your C++ asynchronously so you don't block your web application's event-loop while your C++ is crunching numbers. When we cover this part in this section, it will serve largely as a review of the material already presented in the main chapters of the book.

A running example - Prime Numbers

Throughout this section I'll be showing you examples of how to implement each of the options above. I want to use the same basic example in each. Prime numbers are extremely important for lots of stuff (i.e. cryptography), and their generation tends to be really computationally expensive. A quick search online

will direct you mostly towards C and C++ implementations - and the really efficient ones are *complicated*. Looking at their source, you'll instantly recognize that you probably don't' want to rewrite them - unless you are just looking for a challenge - which is fine:).

One of the more effective algorithms is called Sieve of Eratosthenes. There is a really popular C++ suite, primseieve - but it's pretty complicated to build. Instead, I found a simpler implementation that is more suitable for our purposes. You can find the source code for it at http://wwwhomes.uni-bielefeld.de/achim/prime_sieve.html, but it's also in the git repository - https://github.com/freezer333/cppwebify-tutorial for this section.

Getting Started - a simple Node.js Web app

Throughout this section I'll use the **exact** same Node.js web application. It's pretty bare-bones, there is a single HTML page with some JavaScript (AngularJS) that asks the web server for prime numbers under a user specified value. The web server responds with a JSON object containing the primes - computed using one of the several techniques I'll implement.

I assume the reader has some basic understanding of a Node.js web app. I created the app with Express on the backend and AngularJS on the frontend, but I stayed away from any complexity and eye/candy as to not distract from the purpose of these tutorials. It's also a great setup for an API into your C++ code - just ditch the UI!

To get started - clone my github repository and check out tag "start".

```
$ git clone https://github.com/freezer333/cppwebify-tutorial.git

$ git checkout start
```

You can poke around the web app yourself - but the relevant bits are the front end - found in /web/views and the backend, found in /index.js and /routes.

Let's take a quick look at /index.js. The first ten lines or so are just boilerplate express code:

```
var express = require('express');
var app = express();
var bodyParser = require('body-parser');

app.use(express.static('public'));
app.set('view engine', 'jade');

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));
```

The next line line is building an array of "types" - which will eventually hold entries for each example in the post series. For now, we only have one - a Node.js primesieve implementation.

```
var types = [
  {
    title: "pure_node",
    description: "Execute a really primitive " +
        "implementation of prime sieve in Node.js"
  }];
```

Each of the entries in types will correspond to a route, found in the /routes directory. These are loaded dynamically from index.js and the web server is started by the final hand full of lines.

```
types.forEach(function (type) {
    app.use('/'+type.title, require('./routes/' + type.title));
});

app.get('/', function (req, res) {
  res.render('index', { routes: types});
});

var server = app.listen(3000, function () {
  console.log('Web server listing at http://localhost:%s',
    server.address().port);
});
```

To launch the web server, navigate to the `/web` directory in your terminal and type the following:

```
$ npm install
... dependencies will be installed
$ node index
```

Now point your browser to `http://localhost:3000`. You'll get the index page, which lists the implementation options. For now, you'll just have one option - "pure_node". Click on it, and you'll see a page with a single number box. Type `100` and submit - and the Node.js implementation of primesieve will run and return all prime numbers under 100.

The primesieve implementation in Node.js is found in `routes/pure_node.js`. By comparison to the C implementation we'll use throughout the remainder of this series, it's mind-numbingly simple - but it gets the job done! The code that handles the actual response is the router's post method:

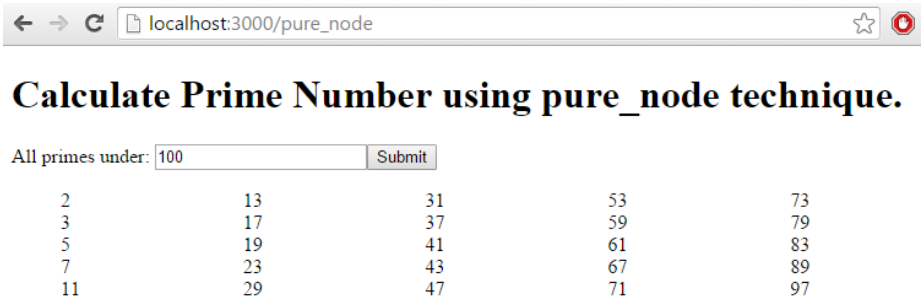


Figure 7: Results for primes under 100

```
router.post('/', function(req, res) {
    var under = parseInt(req.body.under); // from the user

    var primes = find_primes(under);

    res.setHeader('Content-Type', 'application/json');
    res.end(JSON.stringify({
      results: primes
    }));
});
```

Automating a C++ program from a Node.js Web app

If your C++ runs standalone from a command line - or can be made to do so - you can run it using Node's child process API. This option works for bringing just about anything to the web - it really doesn't make a difference what language your command line program is written in if you are simply running it.

Two features of automation make it attractive. First, since you are executing the C++ application in another process, you are essentially doing the C++ job *asynchronously* - which is a big win on the web since you can process other incoming HTTP traffic while the C++ app is working. Second, you really don't need to do a great deal of *language integration* or use sophisticated V8 API's - it's actually pretty easy!

For this particular section, checkout the **automation** tag from the git repository.

```
$ git checkout automation
```

Prime Sieve C/C++ implementation

As described above, we're building all the examples around a C implementation of the Sieve of Eratosthenes Prime number calculation strategy. It's a good example problem, because speed matters big time for prime numbers - and the C code that I'm using is not exactly the type of thing you'd be eager to rewrite! The example I'm using - http://wwwhomes.uni-bielefeld.de/achim/prime_sieve.html - is actually pretty simple, compared to more complex techniques that leverage CPU caching, among other things. Head over to primesieve.org to get an idea. For implementations of Prime Sieve, the user of the program must enter a maximum value, and the algorithm will output all prime numbers "under" this value. We'll call that input value "under" throughout most of this chapter.

To follow along, please take a look at the original primesieve.c code now, found at https://gist.github.com/freezer333/ee7c9880c26d3bf83b8e - although don't get too caught up in the details, we won't need to mess with it much (that's the whole point!).

Modifications to primesieve.c

When faced with integrating a legacy program, you might not have the luxury of accessing the code. For the purposes of this chapter, I'm going to simulate a few common integration scenarios - and I'll edit some bits of the original primesieve.c in order to allow for this.

- **Scenario 1:** An app that gets input only from command line arguments, and prints to standard out.
- **Scenario 2:** An app that gets input from the user (stdin), and prints to standard out.
- **Scenario 3:** An app that gets input from a file, and outputs to another file.

To simulate each scenario, we'll want to be able to pass a file descriptor into the main routine of primesieve.c, so the program doesn't *always* print to the console. Let's rename `main` to `generate_args` and add a third parameter for the file descriptor. We'll make specific use of this in Scenario 3.

```
// in cppwebify-tutorial/cpp/prime4standalone/prime_sieve.c,
// I've renamed int main(int argc, char *argv[])
// to:
int generate_args(int argc, char * argv[], FILE * out) {
    ... complicated prime number stuff ...
```

I'll write the entry point in a different file (`main.cpp`), so I'm also adding the declaration of `generate_args` to a header file called `prime_sieve.h`.

I'm creating a second function - `generate` which provides a simplified interface - it just accepts the "under" parameter instead of command line arguments. The definition is at the bottom of prime_sieve.c, and just transforms the parameter into character arguments and calls `generate_args`. This is just so I don't edit the original code much, and to make Scenario 2 below a little cleaner. Obviously, the imaginative reader can figure out better ways of doing all this :)

```
// at the bottom of cppwebify-tutorial/cpp/prime4standalone/prime_sieve.c,
// an adapter function for use when we aren't using command-line arguments
int generate(int under, FILE *out) {
  char * name = "primes";
  char param [50];
  sprintf(param, "%d", under);
  char * values[] = { name, param};
  generate_args(2, values, out);
}
```

So, we're left with the following prime_sieve.h header - using `extern C` to make sure our C functions can be integrated correctly with the C++ main files I'll use in the examples.

```
extern "C" {
    // the old main, renamed - with a third parameter"
    // to direct output to a file as needed
    int generate_args(int argc, char * argv[], FILE * out);

    // an adapter function when the caller hasn't
    // received under through command line arguments
    int generate(int under, FILE * out);
}
```

The Node.js Child Process API

Node.js contains a `child_process` module which exposes a robust API for creating and controlling processes. There are three basic calls for creating new child processes - each with their own use cases.

The first is `execFile`, which accepts (at a minimum) a file path to an executable program. You may pass an array of arguments that will be called with the program. The last parameter to the function is a callback to be executed when the program terminates. This callback will have an error, a stdout buffer, and a stderr buffer given to it, which can be used to interrogate the program's output. It's important to note that this callback is only called after the program executes. `execFile` also returns an object representing the child process, and you may write to it's stdin stream.

```
// standard node module
var execFile = require('child_process').execFile

// this launches the executable and returns immediately
var child = execFile("path to executable", ["arg1", "arg2"],
  function (error, stdout, stderr) {
    // This callback is invoked once the child terminates
    // You'd want to check err/stderr as well!
    console.log("Here is the complete output of the program: ");
    console.log(stdout)
});

// if the program needs input on stdin, you can write to it immediately
child.stdin.setEncoding('utf-8');
child.stdin.write("Hello my child!\n");
```

I find the `execFile` function is best when you have to automate an application that has well-defined input and operates in sort of a "single phase" - meaning once you give it some input it goes off for a while, and the dumps all of it's output. This is precisely the type of program the prime sieve program is - so we'll use execFile throughout this chapter.

The `child_process` module has two other functions to create processes - `spawn` and `exec`. `spawn` is a lot like `execFile`, it accepts an executable and launches it. The difference is that `spawn` will give you a streamable interface to stdout and stderr. This works really well for more complex I/O scenarios where there is a back and forth dialog between your node code and the C++ app. `exec` is again very similar to `execFile`, but is used for shell programs (ls, pipes, etc).

Synchronous options

In Node.js v0.12 a new set of API's was introduced which allows you to execute child applications *synchronously* - your program will block when you start the child process and resume when the child process terminates (and sends you back it's output). This is fantastic if you are creating shell scripts, but it's decidedly *not* for web applications. For our prime number demo, certainly when we get an HTTP request for prime numbers we need to wait for the complete output before serving the page of results to the browser - *but we should be able to continue serving* ***other HTTP*** *requests from other browsers* in the meantime! Unless you have a really specific reason, you'll want to stay away from `spawnSync`, `execSync`, and `execFileSync` when writing web servers.

Scenario 1: C++ Program with input from command-line arguments

The simplest type of program to automate is a program that will accept all of it's input as command line arguments and dump it's output to stdout - so we'll

start with this scenario.

So - let's "imagine" prime sieve works like this (actually, it basically already does!). To use the application, we might type:

```
$ primesieve 10
2
3
5
7
# {1 <= primes <= 10} = 4
0.000000000000  -3.464368964356
```

And we'd get all prime numbers under 10 printed out to the screen (one on each line) - plus some extra info printed by the program that we don't need.

I'll keep the output easy to parse in all my examples - obviously if your program spits out data in a tough-to-parse way, you'll have a bit more work to do.

Using node-gyp to compile the prime sieve C++

Our first step is to actually get an executable C++ application! The C++ code in `cpp/prime4standalone` doesn't have an entry point - it's just the prime number generation code, and it will be shared across all 3 of the scenarios we're covering here. In `cpp/standalone_stdio` I've created an entry point:

```
#include <iostream>
#include <stdio.h>
#include "prime_sieve.h"
using namespace std;

int main(int argc, char ** argvs) {
    generate_args(argc, argvs, stdout);
}
```

The next step is to build the C++ executable - compiling together all three files: 1. `cpp/standalone_stdio/main.cpp` 2. `cpp/prime4standalone/prime_sieve.h` 3. `cpp/prime4standalone/prime_sieve.c`

If you are familiar with building C++, you'll have no trouble doing this with whatever your favorite compiler platform is. We're going to eventually need to use `node-gyp` - so I've setup all the C++ examples this way.

```
$ node-gyp configure build
```

In `/cpp/standalone_stdio` you'll find a `binding.gyp` file. This contains all the information needed to build this particular example with node-gyp - think of it as a Makefile.

```
{
  "targets": [
    {
      "target_name": "standalone",
      "type": "executable",
      "sources": [ "../prime4standalone/prime_sieve.c",
        "main.cpp"],
      "cflags": ["-Wall", "-std=c++11"],
      "include_dirs" : ['../prime4standalone'],
      "conditions": [
        [ 'OS=="mac"', {
            "xcode_settings": {
                'OTHER_CPLUSPLUSFLAGS' : ['-std=c++11',
                  '-stdlib=libc++'],
                'OTHER_LDFLAGS': ['-stdlib=libc++'],
                'MACOSX_DEPLOYMENT_TARGET': '10.7' }
            }
        ]
      ]
    }
  ]
}
```

Lets cover a few basics. We only have one target defined ("standalone") - so it has become the default. It's `type` is critical here, because node-gyp can also compile shared libraries and (of course!) native Node.js addons. Setting `type` to `executable` tells node-gyp to create a standard runnable executable. The `sources` array contains our source (the header is not needed, but could be added). Since a lot of my C++ later in this section will make use of C++11, I'm also passing in a few compiler flags in the `cflags` property. I also pass along OS X specific stuff to make C++11 work on a Mac with XCode. These special options are included in the `conditions` property and are ignored under Linux and Windows. Finally, I've made sure the compiler can find the include file by adding in the path under the `include_dirs` property.

The result of our build operation - `node-gyp configure build` - should create an executable in `cpp/standalone_stdio/build/Release` called `standalone`. You should be able to run it directly from the command line. **Now let's run it from Node.js**.

Automating from Node.js

Earlier we setup a really simple Node.js web application that had a single route that could calculate prime numbers using a pure JavaScript prime sieve implementation. Now we'll create a second route that uses our C++ implementation.

In `cppwebify-tutorial/web/index.js` first we'll add a new entry in our `types` array for the new C++ route:

```
var types = [
  {
    title: "pure_node",
    description: "Execute a really primitive " +
      "implementation of prime sieve in Node.js"
  },
  {
    title: "standalone_args",
    description: "Execute C++ executable as a " +
                 "child process, using command " +
                 "line args and stdout.  " +
                 "Based on /cpp/standalone_stdio"
  }];
```

That type array is used to create the routes by looking for a file named the same as each `title` property in the `web/routes/` directory:

```
types.forEach(function (type) {
    app.use('/'+type.title, require('./routes/' + type.title));
});
```

Now let's add our route in `/web/routes/standalone_args`. If you take a look, lines 1-9 are basically the same as the `pure_node` example - line 11 is where we start respond to an actual user request for prime numbers by executing the C++ app:

```
router.post('/', function(req, res) {
    var execFile = require('child_process').execFile
    // we build this with node-gyp above...
    var program = "../cpp/standalone_stdio/build/Release/standalone";

    // from the browser
    var under = parseInt(req.body.under);
    var child = execFile(program, [under],
      function (error, stdout, stderr) {
        // The output of the prime_sieve function has
        // one prime number per line.
```

```
        // The last 3 lines are additional information,
        // which we aren't using here - so I'm slicing
        // the stdout array and mapping each line to an int.
        // You'll want to be more careful parsing your
        // program's output!
        var primes = stdout.split("\n").slice(0, -3)
                           .map(function (line) {
                             return parseInt(line);
                           });

        res.setHeader('Content-Type', 'application/json');
        res.end(JSON.stringify({
          results: primes
        }));

        console.log("Primes generated from " + type);
    });
});
```

While you'll likely need to be a bit more robust when handling program output (and dealing with input from the browser), as you can see it's pretty simple to call your child process and return a response to the browser. Go ahead and run the web app by typing `node index.js` in your terminal under `cppwebify-tutorial/web` and point your browser to `http://localhost:3000/`. Choose the "standalone_args" strategy, you can enter 100 to get all the primes under 100 - this time using a much faster C-based implementation!

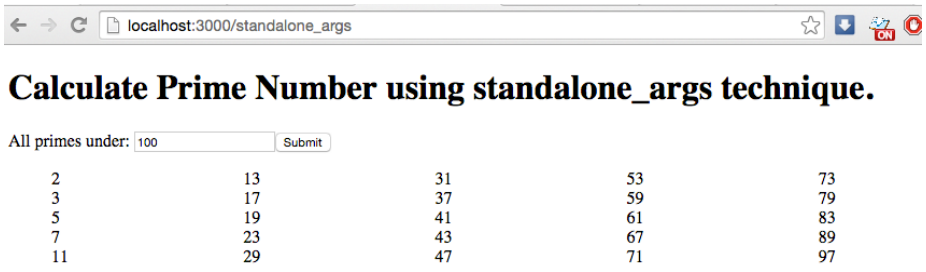


Figure 8: Results for primes under 100

Scenario 2: C++ Program that gets input from user (stdin)

Lots of programs ask an actual user for their input. If you have access to the code of your program, it's probably easy to change it so it accepts these inputs as command line args - which means you could just use the strategy in Scenario 1. Sometimes this won't work though - like if you don't even have the source

code! It also doesn't work when automating a program that actually has a bit of a dialog with the user, which you need to simulate through node. No worries though - writing to stdin is pretty straightforward, especially if you don't need to wait for any output from the child process first (if you do, check out `spawn` instead of `execFile` by the way).

Building the C++ example

In `cpp/standalone_usr` I've created a new entry point for a C++ program that simply asks the user for the `under` parameter the prime sieve algorithm needs.

```
#include <iostream>
#include <stdio.h>
#include "prime_sieve.h"
using namespace std;

int main(int argc, char ** argvs) {
    int max;
    cout << "Please enter the maximum number:  ";
    cin  >> max;
    generate_primes(max, stdout);
}
```

It's including the very same prime_sieve.h file as the code in Scenario 1, and is build with a strikingly similar `binding.gyp` file. Go ahead and build that with `node-gyp configure build` at the terminal from `cpp/standalone_usr`.

Writing to stdin to automate from Node.js

Now we've got a new executable build, which asks for input from a live user. We can now drop yet another route into our web app to automate this one too. In `web/index.js` we'll create another type entry:

```
var types = [
  {
    title: "pure_node",
    description: "Execute a really primitive " +
          "implementation of prime sieve in Node.js"
  },
  {
    title: "standalone_args",
    description: "Execute C++ executable as a " +
                " child process, using command line "+
                " args and stdout.  " +
                " Based on /cpp/standalone_stdio"
```

```
  },
  {
    title: "standalone_usr",
    description: "Execute C++ executable as a " +
                 " child process, using direct user input.  "+
                 " Based on /cpp/standalone_usr"
  }];
```

And we'll create a new route at `web/routes/standalone_usr.js`. In this file, our code will no longer pass `under` as a command line argument however, instead we'll write to stdin:

```
router.post('/', function(req, res) {
    var execFile = require('child_process').execFile
    // notice we're pointing this to the new executable
    var program =
      "../cpp/standalone_usr/build/Release/standalone_usr";

    var under = parseInt(req.body.under);
    // execFile will return immediately.
    var child = execFile(program, [],
      function (error, stdout, stderr) {
        // This function is executed once the program ends
        var primes = stdout.split("\n").slice(0, -3)
                       .map(function (line) {
                           return parseInt(line);
                       });

        res.setHeader('Content-Type', 'application/json');
        res.end(JSON.stringify({
          results: primes
        }));

        console.log("Primes generated from " + type);
    });

    // now we write "under" to stdin so the C++ program
    // can proceed (it's blocking for user input)

    child.stdin.setEncoding('utf-8');
    child.stdin.write(under + "\n");

    // Once the stdin is written, the C++ completes
    // and the callback above is invoked.
});
```

By now you probably have the idea.. fire up the web app again and now you'll have a third entry at the start page - go ahead and test it out!

Scenario 3: Automating a file-based C++ program

The last scenario I'll go over is where the program you are automating takes its input from a file, and dumps its output to a another file. Of course, your scenario might be a combination of the three scenarios discussed here - and your scenario might involved a fixed filename for input/output, or a user specified (via stdin, or command line arguments). Whatever your situation, you'll likely be able to apply what's here.

Modifications to prime sieve to use files

So the first step is to shape the prime sieve into something resembling a file-based program. If you take a look at `cpp/standalone_flex_file`, I've created a third entry point for prime sieve that accepts input/output filenames along the command line. The input file is assumed to simply have "under" on the first line. The output file will receive the same lines of results as previously went to stdin.

```
#include <iostream>
#include <stdio.h>
#include "prime_sieve.h"

using namespace std;

// Simulating a legacy app that reads
// it's input from a user-specified file via command line
// arguments, and outputs to a similarly specified file.
int main(int argc, char ** argvs) {
    FILE * in = fopen(argvs[1], "r");
    int i;
    fscanf (in, "%d", &i);
    fclose(in);

    FILE * out = fopen(argvs[2], "w");
    generate_primes(i, out);
    fprintf(stdout, "Output saved in %s\n", argvs[2]);
    fclose(out);
}
```

We can build this C++ program by issuing the familiar `node-gyp configure build` from `cpp/standalone_flex_file`. This will generate a target executable we can use from node.

Dealing with file-based program on the web

Before diving into the Node.js route for this scenario, lets talk about the challenge involved in a file-based program. Most applications never meant for the web will read a specified input file and write to an output file as if the application is the only thing running... and as if it's not running alongside another instance of the same program! This made sense when these applications were being run manually - but if you are placing them on the web you can easily have multiple simultaneous requests (from different browsers) coming in at the same time. It's critical that these simultaneous executions of your legacy C++ program don't collide with each other - you need to ensure they are reading from and writing to their own distinct files!

When you don't have access to the legacy source code, this can be easier said than done, especially if the app does not let the user specify the files (i.e. they are hardcoded in the program!). If they are hardcoded, but relative file paths, then you could play games with the current working directory, or create a copy of the executable (or a link to it) in a temporary directory on each incoming web requests. It's costly performance-wise, but it works. If the files paths are hard coded to absolute paths, you have quite a problem (find the code!).

I've simulated the easiest (but most common) situation, where the input and output files can be specified by the user (in this case, via command line arguments). All we need to do is make sure each web request that launches the C++ app picks unique filenames - and I usually do this by creating temporary directories on each web request, placing the input/output files within the temporary directory. This shields each running instance from the others, while keeping the input/output names consistent.

So now lets jump the Node.js route. At the top of `web/routes/standalone_file.js` I've required the `temp` module, which I use to handle the creation of temporary directories and files. It drops the temporaries in the appropriate location for your platform.

```
var temp = require('temp');
```

Below is the route code found in `web/routes/standalone_file.js`.

```
router.post('/', function(req, res) {
    var execFile = require('child_process').execFile
    var program =
      "../cpp/standalone_flex_file/build"+
      "/Release/standalone_flex_file";

    var under = parseInt(req.body.under);

    // Create a temporary directory, with
```

```
    // node_example as the prefix
    temp.mkdir('node_example', function(err, dirPath) {
      // build full paths for the input/output files
      var inputPath = path.join(dirPath, 'input.txt');
      var outputPath = path.join(dirPath, 'output.txt');

      // write the "under" value to the input files
      fs.writeFile(inputPath, under, function(err) {
        if (err) throw err;

        // once the input file is ready, execute the C++
        // app with the input and output paths
        // specified on the command line
        var primes = execFile(program,
          [inputPath, outputPath], function(error) {

            if (error ) throw error;
            fs.readFile(outputPath, function(err, data) {
              if (err) throw err;
              var primes = data.toString().split('\n')
                              .slice(0, -3)
                              .map(function (line) {
                                  return parseInt(line);
                              });
              res.setHeader('Content-Type', 'application/json');
              res.end(JSON.stringify({
                results: primes
              }));

              exec('rm -r ' + dirPath, function(error) {
                if (error) throw error;
                console.log("Removed " + dirPath);
              })
          });
        });
      });
    });
});
```

The above code first creates the temporary directory. It then writes the input file and launches the child process with the input and output file paths as command line arguments. Once the process completes, we read the output file to get the results, serving it back to the browser just like before. Finally, we clean up the temporary files by removing parent directory. This is important, since even though the `temp` module allows for tracking and automatic deletion of temporary files, it only cleans things up when the process terminates. Since this is a web

app, we would (hopefully!) be waiting a long time for this to happen.

As you can see, this code would benefit from better control flow patterns(async, promises, etc). I'm trying to stick to the bare minimum, I'll leave that to you :).

Aside from the route above, I've added this final scenario to the `types` array in `web/index.js` and you can start your web app and test this one out just like the others.

Calling Native C++ from Node.js as a DLL

This section focuses entirely on compiling your C++ as a shared library or DLL, and calling that code from Node.js using FFI. I'll also discuss some of the common issues you can run into when trying to convert a legacy C++ application into a callable shared library.

Why use a Shared Library / DLL?

When automating a C++ application, you have the advantage of having really clean separation between your JavaScript and C++. Automation also allows you to integrate with just about *every* programming language - as long as it can be automated through stdin/stdout or input and output files. One disadvantage though is that there is really only one entry point to your C++ - main. You could certainly develop complex coordination between your C++ and Node applications, but automation works best when you just want to send some input to C++ and wait for the results.

Often you want fine-grain control and coordination between Node.js and C++. You'd like to be able to call into C++ through *functions*, not just an executable's entry point. Further, you'd like to be able to get output from those functions as return values (or reference parameters), not by harvesting output from stdout or some output file.

A shared library (or DLL) is an excellent solution in this situation. If your C++ is already in a DLL, then you can get started right away - but if not, you can generally compile your legacy code into a DLL fairly easily - you just need to figure out which methods/functions you wish to expose to callers. Once you have a DLL, utilizing the interface through Node.js is pretty easy (read on!).

Converting a legacy C or C++ application into a DLL can be a good integration choice when automation is too cumbersome. It also lets you avoid the intricacies of developing for Node using the V8 API, which isn't always trivial.

For this particular section, checkout the **dll** tag

Once you've checked out the code, take a moment to survey the directory structure I've setup. The `/cpp` directory is where I've put all the C++ applications developed for the automation examples, with the shared source for prime number generation in `/cpp/prime4standalone`. Now we'll need to modify the

prime number code to allow it to work well as a DLL, and I'll put that code in `/cpp/prime4lib`. As was the case before, the sample web application is in `web`. We'll just be adding one route (`ffi`) in this post - for the shared library implementation.

Preparing the C++ as a Shared Library

If you are trying to integrate an existing shared library into Node.js, then you can basically skip this section - you are all set! If you have some legacy C++ code that was originally a standalone app (or part of one), you need to prepare your code to work as a shared library first. The major considerations when doing this is defining your API - the set of functions that should be callable by the host code (in our case, Node.js). Perhaps your C++ already is organized such that these functions are ready to go - but you may need to do a bit of reorganization.

Another main consideration is how you'll get your C++ code's output. For example, when automating, I ran a bunch of standalone primesieve applications from Node - each one either outputted prime numbers directly to standard out or to an output file. We don't want this for shared libraries though - we want the output *returned* to the caller. To do this, you might need to get a bit creative - I'll show you how I've done it in the section below.

Here's the API I want my shared library to support. Actually, it's not much of an API - it's just one function!

```
int getPrimes(int under, int primes[]);
```

The first parameter represents the maximum value - such that we'll find all prime numbers *under* this value. The prime numbers will be stuffed into the second parameter - an array. It is assumed that this array has enough space to store all the generated prime numbers (`under` is a good "maximum" size.). The function will return how many prime numbers were actually found.

Capturing the output

Now let's look at the code from the automation example. Inside `/cpp/prime4standalone`, the `primesieve.c` file had one main function:

```
int generate_args(int argc, char * argv[], FILE * out)
```

It also has an adapter function which replaces the argc/argv parameters with `under`. In both cases, notice that the output is being sent to `out` using `fprintf`. For our API, we want the output to be placed inside an array.

One approach might be to just start hacking away at the underlying primesieve implementation, replacing the `fprintf` calls with some code to load up an array.

This can work (especially if this is new C++ code, or at least C++ that is fairly straightforward), but it's not particularly scalable (what if you have a more complex set of actions you need to perform to capture the output?). I find making modifications to legacy programs goes best when you keep your changes simple - and that's what I'll do here.

Data Exchange utility class useable from C or C++

As with most things in life, keeping one thing simple often makes something else more complicated. My goal is to replace each `fprintf` statement in the existing primeseive code with a similarly simple function:

```
void pass(int prime);
```

I want that send function to be able to add the prime number into an array, which is sent in from the calling Node.js code:

```
// called from Node.js - calls to send should add prime to primes
int getPrimes(int under, int primes[]);
```

This seems simple enough, we could achieve something like this by making send a member method of an object that could have a reference to the array. This gets complicated by the fact that primeseive is straight C code though.

Let's start with the data exchange class, found in `exchange.h`:

```
#define _exchangeclass
#include <iostream>
#include <functional>
using namespace std;

class exchange {
public:
    exchange(const std::function<void (void * )> & c) {
        this->callback = c;
    }
    void send(int data){
       this->callback(&data);
    }
private:
    std::function<void (void * )> callback;
};

#include "c_exchange.h"
```

The first thing you'll note is that the class itself does not contain a reference to an array. To keep it general, I am simply having it hold a callback function - which will be responsible for storing the given value to the array in this example, but could do anything at all.

Notice the last line - I'm including a separate header file called `c_exchange.h`. The `send` member of exchange is not callable from C code (primesieve), and as you might have guessed, `c_exchange.h` contains a function to get around this problem. Lets take a look inside:

```
#ifdef _exchangeclass
extern "C" {
#endif

void pass(void * exchanger, int data);

#ifdef _exchangeclass
}
#endif
```

First off, this header is going to be included by C++ and C code. `exchange.h`, which declares the exchange class defines the `exchangeclass` symbol - so the first line is just detecting if that symbol is already there. If it is, the `pass function` - which will be called from C - is wrapped in an `extern` block.

The `pass` function accepts a pointer to an exchange object (`void *`, since the `exchange` class won't be visible to C callers). Within the definition, found in `exchange.cpp`, we see that this pointer is cast back to an `exchange` object and the send method is called:

```
void pass(void * exchanger, int data) {
    exchange * xchg = (exchange * ) exchanger;
    xchg->send(data);
}
```

It's a bit elaborate, but the `exchange` class and it's standalone `pass` helper function can be dropped into nearly any existing C++ or C legacy program, simply by getting a pointer to an `exchange` object into the legacy code and replacing output calls with `pass`. Let's do this with `primesieve.c`.

Modifying primesieve to use passing function

Inside `/cpp/prime4lib` I have a modified `primesieve.h` and `primesieve.c`. The **old** `primesieve.h` defined the following two functions:

```
// primeseive.h for standalone programs
int generate_args(int argc, char * argv[], FILE * out);
int generate_args(int under, FILE * out);
```

Now I've replaced these with the following signatures:

```
// primesieve.h for library calls
int generate_args(int argc, char * argv[], void * out);
int generate_args(int under, void * out);
```

Inside `primeseive.c` the old standalone code had a `#define` setup to use fprintf, on line 43 (Note, I am not the author of the original primsieve code - I do not know the history or intent behind the elaborate printing scheme. As with most legacy apps, sometimes those questions are better left un-asked!). We now replace the `fprintf(out, UL"\n",x)` call with a call to `pass(out, x)`.

The shared library entry point

Now we have a `primesieve.h/primeseive.c` implementation that uses `pass`, we just need to create a C++ entry point that creates an `exchange` object and calls the primesieve code. I have done this in `/cpp/lib4ffi/primeapi.h` and `/cpp/lib4ffi/primeapi.cpp`.

`primeapi.h` is the shared library entry point, it has the declaration for the library API function I wished for up above:

```
extern C {
    int getPrimes(int under, int primes[]);
}
```

The implementation uses the `exchange` class, with a lambda function as the callback. As you can see, the lambda function adds whatever data is sent to the array.

```
int getPrimes(int under, int primes[]) {
  int count = 0;
  exchange x(
        [&](void * data) {
            int * iptr = (int * ) data;
            primes[count++] = * iptr;
        }
  );

  generate_primes(under, (void*)&x);
  return count;
}
```

Now, when we call `generate_primes`, which is defined in `primesieve.h`, we pass in a reference to our exchange. Within `primesieve.c` that reference to the exchange object is called `out`. All calls to `pass(out, x)` in `primesieve.c` result in the pointer `out` being cast as an `exchange` object (in `exchange.cpp`), and the callback (the lambda) is fired. *The end result is that all values computed by `primesieve` are found in the `primes` array.*

Building the Shared Library with gyp

We need to build our shared library now. Luckily, the very same toolset we are used to - `node-gyp` - can help us here as well. Inside `/cpp/lib4ffi` you'll find another config file named `binding.gyp`. It's quite similar to the gyp files found in the standalone examples from the automation examples, but it links in the primesieve files from `/cpp/prime4lib` instead of `/cpp/prime4standalone` and it's build type is `shared_library` instead of `executable`.

Build the shared library by issuing the familiar `node-gyp configure build` from `cpp/lib4ffi`. This will generate a target shared library we can use from node. The shared library will be in `/cpp/lib4ffi/build/Release` - with an extension specific to your operating system (ie. prime.dylib on OS X, prime.dll on Windows).

Calling primelib with FFI

All that work and we have a shared library - now let's call it from Node.js. To do this, we'll use Node's Foriegn Function Interface (node-ffi). node-ffi is a Node.js addon for loading and calling dynamic libraries using pure JavaScript. You can find an excellent tutorial at https://github.com/node-ffi/node-ffi/wiki/Node-FFI-Tutorial which outlines it some more detail. In particular, checkout the async section, which shows you how to easily call shared library methods in their own threads using libuv so you don't block your main Node.js event loop!

One of the key parts of using `node-ffi` is mastering the `ref` module to build native data types on top of the Node.js `Buffer` object (the subject of Appendix B). These datatypes (int, arrays, etc.) allow you to interact with native functions found inside a shared library.

Our API has but one call, and it uses two integers and an integer array for return types and parameters:

```
int getPrimes(int under, int primes[]);
```

Simple integers don't require us to do much (node-ffi automatically converts to and from the JavaScript number type), but we do need to allocate an integer array to hold our results. Here's how we do it with `ref`

```
var ArrayType = require('ref-array');
var IntArray = ArrayType(int);
var a = new IntArray(10); // creates an integer array of size 10
```

Next, we use the `ref` data type identifiers and `node-ffi` to define the interface to our library:

```
var ffi = require('ffi')
var ref = require('ref')
var int = ref.types.int

var libprime =
  ffi.Library('../cpp/lib4ffi/build/Release/prime', {
  'getPrimes': [ int, [ int, IntArray] ]
})
```

The `libprime` variable now represents our getPrimes function found in the shared library we created in the previous section. We can call the function, and it's return type can be saved in a normal JavaScript number variable. We can use the returned count to extract the prime numbers out of the IntArray - giving us all the prime numbers under 10.

```
var count = libprime.getPrimes(under, a);
var primes = a.toArray().slice(0, count);
```

Putting it all together

Now that we have a shared library, and the Node.js code that can call it, let's wrap it all up into it's own route inside our growing web app example. Inside the `/web/index.js` file we are going to add another entry for a route called `ffi`.

```
var types = [
  {
    title: "pure_node",
    description: "Execute a really primitive " +
        " implementation of prime sieve in Node.js"
  },
  ... the entries for the automation example routes...
  {
    title: "ffi",
    description: "Using Node Foreign Function " +
        " Interface (ffi) to call C++ code.  Based on /cpp/lib4ffi"
  }
  ];
```

That type array is used to create the routes by looking for a file named after each `title` property in the `web/routes/` directory:

```
types.forEach(function (type) {
    app.use('/'+type.title, require('./routes/' + type.title));
});
```

Now let's add our route in `/web/routes/ffi.js`. The relevant post handler is below, and it looks a lot like the `ffi` example above:

```
router.post('/', function(req, res) {
    var ffi = require('ffi')
    var ref = require('ref')
    var ArrayType = require('ref-array')
    var int = ref.types.int
    var IntArray = ArrayType(int)

    // The under parameter is coming from
    /// the user input (form)
    var under = parseInt(req.body.under);
    var a = new IntArray(under);

    // Create the interface to our shared library
    var libprime = ffi.Library(
      '../cpp/lib4ffi/build/Release/prime', {
        'getPrimes': [ int, [ int, IntArray] ]
    })

    // call the prime number code and extract
    // the array of primes.
    var count = libprime.getPrimes(under, a);
    var primes = a.toArray().slice(0, count);

    // send the primes right back to the
    // browser for display
    res.setHeader('Content-Type', 'application/json');
    res.end(JSON.stringify({
      results: primes
    }));
});
```

Now fire up your web app by typing `node index.js` from `/web` and choose the ffi option. Type in 100 and click "submit" and you should see the prime numbers under 100 on your screen, this time generated by the DLL/shared library.

Building an Asynchronous C++ Addon for Node.js using Nan

This entire book is dedicated to C++ addons, so this section serves as yet another example - but it's a great where to compare a lot of alternatives. After reading this book, you are probably predisposed to taking the addon route, but there are indeed good reason *not to*. If you don't have access to the source code of your legacy C++ application, then automation is your best option - you won't be able to create the type of Node.js addon I'll describe here. Of course, if your legacy code is *not C or C++*, then automation might very well be your best bet as well (although there are indeed bindings from Node to other languages as well). If your C/C++ code is already in a dll or shared library, then of course it likely makes the most sense to use FFI - as described above.

For situations where you have complete access (and are comfortable editing) the C/C++ you are targeting though, creating a native addon is likely to be the most powerful approach. First, if your code is already reasonably well organized (clearly defined entry and exit/return points), it won't be too difficult to create the addon itself - especially using Nan. Second, addons are quite flexible - they can be blocking/synchronous or asynchronous, and support most use cases (i.e. passing/returning objects, arrays, etc.). Finally, when you create a Node.js addon, your JavaScript code is cleaner than when using the automation or shared library approaches - as you'll see by comparing JavaScript code in this post with the other posts in the series.

For this particular section, checkout the **addon** tag

Addon Code - blocking

Lets create our C++ addon file - `/cpp/nodeprime_sync/addon.cpp`. We are going to crate a wrapper around the `getPrimes` function found in `/cpp/prime4lib` and register it with V8 using macros defined in V8 and Nan. First, we'll include the headers for a primesieve code, the exchange class we use to collect data from the primeieve code, and V8/Nan:

```
#include <nan.h>  // includes v8 too
#include <functional>
#include <iostream>

// class to hold values returned from primesieve
#include "exchange.h"
#include "prime_sieve.h"

// bring in the required namespaces
using namespace Nan;
```

```
using namespace v8;
using namespace std;
```

Now let's create a function that will do the calculation. Much of it is familiar from the shared library post, we'll use the `exchange` class to collect output from primesieve. The main difference is that we'll be collecting the data into a V8 Local Array, which can be returned wholesale to the calling JavaScript code. Before diving into the C++, here's how the function will (almost) be used in JavaScript:

```
var primes = primenode.getPrimes(under);
// primes is now the array of all prime numbers less than under
```

Here's the C++:

```
NAN_METHOD(CalculatePrimes) {
    Nan:: HandleScope scope;

    int under = To<int>(info[0]).FromJust();
    v8::Local<v8::Array> results = New<v8::Array>(under);

    int i = 0;
    exchange x(
        [&](void * data) {
            Nan::Set(results, i,
              New<v8::Number>(*((int *) data)));
            i++;
       });

    generate_primes(under, (void*)&x);

    info.GetReturnValue().Set(results);
}
```

After we extract the necessary arguments and create an array for our results, we work with exchange class. We are creating a callback, which primesieve (`generate_primes`) will call each time a prime number is found. Here, instead of adding each prime number to a vector, we are adding it to the local V8 array we have declared. Note that the array will be "oversized", since if "under" is 100 there is clearly not 100 prime numbers less than 100! Each element that is not explicitly set will be set to `undefined` when accessed through JavaScript later. We now call the primesieve implementation, which executes and incrementally fills up the array with primes through the exchange object.

```
int i = 0;
exchange x(
    [&](void * data) {
        Nan::Set(results, i, New<v8::Number>(*((int *) data)));
        i++;
   });

generate_primes(under, (void*)&x);
```

Finally, we need to register this function (`CalculatePrimes`) with V8, which we do at the bottom of the file:

```
NAN_MODULE_INIT(Init) {
    Nan::Set(target,
      New<String>("getPrimes").ToLocalChecked(),
      GetFunction(
          New<FunctionTemplate>(CalculatePrimes))
          .ToLocalChecked());
}

NODE_MODULE(addon, Init)
```

Building the addon

We can use a familiar `binding.gyp` file to build the addon:

```
{
  "targets": [
    {
      "target_name": "nodeprime",
      "sources": [ "../prime4lib/prime_sieve.c",
                   "../prime4lib/exchange.cpp",
                   "addon.cpp"],
      "cflags": ["-Wall", "-std=c++11"],
      "include_dirs" : ['../prime4lib',
        "<!(node -e \"require('nan')\")"],
      "conditions": [
        [ 'OS=="mac"', {
            "xcode_settings": {
                'OTHER_CPLUSPLUSFLAGS' :
                  ['-std=c++11','-stdlib=libc++'],
                'OTHER_LDFLAGS': ['-stdlib=libc++'],
                'MACOSX_DEPLOYMENT_TARGET': '10.7' }
            }
        ]
```

```
      ]
    }
  ]
}
```

There are a few new things in this bindings file. First, notice there is no "type" property - by default node-gyp build a Node.js addon - so no need to specify anything. I've defined the target to be `nodeprime`, and the output of the build will end up being `nodeprime.node`. In addition to specifying the build files, and the include directory for primesieve, I've also added Nan to the set of include directories - utilizing a node shell command. The rest (conditions) is the same compiler stuff from the previous posts, mainly to enable C++ 11.

To build, do a `node-gyp configure build` from `/cpp/nodeprime_sync`. The `nodeprime.node` file will be located in `/cpp/nodeprimes_sync/build/Release` - which we'll link to in a moment.

Calling from JavaScript

Now comes the easy part! First, we need to require the module. We specify a path in the require command

```
var nodeprime = require("[relative path to code]" +
  + "/cpp/nodeprime_sync/build/Release/nodeprime")
```

Now to get prime numbers under 100, just call the function:

```
var retval = primes.getPrimes(100);
console.log(retval);
```

`retval` is now just a JavaScript array - however you'll see that there are 100 elements (mostly empty), since we over-allocated in C++. We can get rid of that pretty easily though:

```
var retval = primes.getPrimes(100)
             .filter(function(val) {
                 return val != undefined
             });
console.log(retval);
```

Addon Code - non-blocking

Synchronous code is a real problem if you are integrating a web application. If the JavaScript code above were executed in response to an HTTP request, no

other requests can be processed until the array is returned. The way we've coded the addon, the C++ code is executing in the Node.js event loop. It would be far better to use an asynchronous model!

I've created the asynchronous addon in `/cpp/nodeprime`. Within that folder, you'll see a package.json file (you need to do a `npm install`) that sets up Nan. You'll also see a similar binding.gyp file as before, and another addon.cpp file that contains the asynchronous addon.

First off, inside `addon.cpp`, you'll see the top part (includes/namespaces) and bottom part (`NAN_METHOD` and `NODE_MODULE`) are exactly the same. The change now is how `CalculatePrimes` is implemented, and the addition of the PrimeWorker class, which inherits `AsyncWorker` and contains all the logic for doing the work. Before diving in, let's look at what the calling JavaScript code will eventually look like:

```
// Asynchronously get all prime numbers under 100
nodeprime.getPrimes(100, function (err, primes) {
       console.log(primes);
});
```

Notice that `getPrimes` now gets two parameters, "under" and a callback function that receives the result when it's complete. That's where we'll start in the C++, because we need to get a reference to that callback so we can invoke it:

```
NAN_METHOD(CalculatePrimes) {
    int under = To<int>(info[0]).FromJust();
    Callback *callback = new Callback(info[1].As<Function>());

    AsyncQueueWorker(new PrimeWorker(callback, under));
}
```

Notice now that `CalculatePrimes` extracts two parameters - under and the callback. Now, instead of actually computing the prime numbers, we create a `AsyncQueueWorker` with an instance of our `PrimeWorker` class. Our `PrimeWorker` class is created with the callback and under parameter, since they will be used to process the work. `AsyncQueueWorker` returns *immediately* - it simply queues the worker. The C++ now returns control right back to the calling JavaScript code.

Now let's look at what is actually going on inside the `PrimeWorker` class. The constructor is pretty simple - most importantly it initializes the base class `AsyncWorker` with the callback sent in from JavaScript. The `under` value is also saved in `PrimeWorker`, and the primes vector that will hold our prime number results is initialized.

```
PrimeWorker(Callback *callback, int under)
        : AsyncWorker(callback), under(under), primes(0) {}
```

Here's where there is a big difference from the synchronous addon - we're going to save the prime numbers in a standard C++ vector as opposed to directly into a V8 Local Array. This is because the prime numbers are being calculated in a worker thread, not in the event loop.

Once we call `AsyncQueueWorker` from `CalculatePrimes`, libuv will dispatch our `PrimeWorker` object onto a worker thread and call it's `Execute` method - which is shown below:

```
void Execute () {
  exchange x(
    [&](void * data) {
      primes.push_back(*((int *) data));
    }
  );

  generate_primes(under, (void*)&x);
}
```

This is pretty much *exactly* what the synchronous version of `CalculatePrimes` did - it's just being executed in the worker thread. Once `Execute` completes, libuv will automatically call the `HandleOKCallback` methods on `PrimeWorker` - in the event loop.

```
void HandleOKCallback () {
  Nan:: HandleScope scope;

  v8::Local<v8::Array> results = New<v8::Array>(primes.size());
  int i = 0;
  for_each(primes.begin(), primes.end(),
    [&](int value) {
      Nan::Set(results, i, New<v8::Number>(value));
      i++;
    });


  Local<Value> argv[] = { Null(), results };
  callback->Call(2, argv);
}
```

Since this method is actually called in the Node event loop thread, we can allocate a V8 Local Array that will be returned back to JavaScript. We create a

scope, initialize an array (this time, exactly the right size, since we already have the vector with the primes). Next we use a `for_each` to fill the array.

The final step is to actually invoke the JavaScript callback that was sent in as the initial parameters to our addon. We pack an arguments array representing the parameters (Null first, since there is no error, and then the array). We end by executing the callback - at which time control is sent back to JavaScript again.

You'll need to do another `node-gyp configure build` to build this module, and now we can call it from Node.js.

Calling from JavaScript

As shown above, we just need to `require` the module now:

```
var nodeprime = require("[relative path to code]" +
  "/cpp/nodeprime/build/Release/nodeprime")
```

Now to get prime numbers under 100, just call the function - passing in a callback that will be invoked once the prime numbers are generated:

```
primes.getPrimes(100, function (err, primes){
  console.log(primes);
});
```

Putting it all together...

OK... so lets put this on the web app we've been developing. I'll just show you the asynchronous version, since the synchronous model really doesn't play well with the web at all.

Inside the `/web/index.js` file we are going to add another entry for a route called `ffi`.

```
var types = [
  {
    title: "pure_node",
    description: "Execute a really primitive " +
      "implementation of prime sieve in Node.js"
  },
  //... the entries for the automation and shared library
  {
    title: "addon",
    description: "Creating a Node Addon that can " +
      "be called like any other module.  Based on /cpp/nodeprime"
  }
  ];
```

That type array is used to create the routes by looking for a file named after each `title` property in the `web/routes/` directory:

```
types.forEach(function (type) {
    app.use('/'+type.title, require('./routes/' + type.title));
});
```

Now let's add our route in `/web/routes/addon.js`. The relevant post handler is below, and it looks a lot like the the code we've already seen:

```
router.post('/', function(req, res) {
    var under = parseInt(req.body.under);
    primes.getPrimes(under, function (err, primes) {
        res.setHeader('Content-Type', 'application/json');
        res.end(JSON.stringify({
            results: primes
        }));
    });
    console.log("Primes generated using " + type);
});
```

Fire up the web app (`node index` from `/web`) and try the link for `addon`. No surprises.

Appendix B - Buffers

Chapter 2 of this book covered using typical JavaScript data types when moving data to and from C++ addons. Node.js introduces a new data type, `Buffer`, which is not found in standard JavaScript (although new versions of JavaScript now have typed arrays, which provide much of the same functionality). Node.js `Buffer` objects are used to represent raw binary data, similar to a C++ array (in this case, integer array). Whenever you work with Node.js file I/O or TCP you are likely going to work with `Buffer` objects. While it is common to convert `Buffer` objects to JavaScript strings (by specifying the encoding of the integer data), often times you may wish to operate directly on the binary data as well.

`Buffer` objects are an interesting aspect OF C++ addon development, first because they are in fact *not* part of V8 - but part of Node.js. Secondly, `Buffer` object data is unique in that it is *not* allocated inside the V8 heap - an attribute that can allow us to sidestep some data copying when dealing with C++ addons and worker threads (which will be discussed below).

In this section, we'll look at how `Buffer` objects can be passed to and from C++ addons using NAN. NAN is used because the `Buffer` object API has actually undergone some significant changes recently, and NAN will shield us from these issues. We'll look at `Buffer` objects through the lens of an image converter - specifically converting binary png image data into bitmap formatted binary data.

All of the code for this section is available in full in the `nodecpp-demo` repository at https://github.com/freezer333/nodecpp-demo, under the "Buffers" section.

Example: PNG and BMP Image Processing

Image processing, in general, is anything that manipulates/transforms an image. An image of course is a big chunk of binary data - in it's most basic state an integer (or 3 or 4) can be used to represent each pixel in an image, and those integers can be stored in a file or held in a contiguously allocated array. Typically image data is not held in *raw* data form though, it's compressed/encoded into a image format standard such as png, gif, bmp, jpeg, and others.

Image processing is a good candidate for C++ addons, as image processing can often be time consuming, CPU intensive, and some processing technique have parallelism that C++ can exploit. For the example we'll look at now, we'll simply convert png formatted data into bmp formatted data[9]. There are a good number of existing, open source C++ libraries that can help us with this task, I'm going to use LodePNG as it is dependency free and quite simple to use. LodePNG can be found at http://lodev.org/lodepng/, and it's source code is

[9] Converting from png to bmp is *not* particularly time consuming, it's probably overkill for an addon - but it's good for demonstration purposes. If you are looking for a pure JavaScript implementation of image processing (including much more than png to bmp conversion), take a look at JIMP at https://www.npmjs.com/package/jimphttps://www.npmjs.com/package/jimp.

at https://github.com/lvandeve/lodepng. Many thanks to the developer, Lode Vandevenne for providing such an easy to use library!

Setting up the addon

For this addon, we'll create the following directory structure, which includes source code downloaded from https://github.com/lvandeve/lodepng, namely `lodepng.h` and `lodepng.cpp`.

```
/png2bmp
 |
 |--- binding.gyp
 |--- package.json
 |--- png2bmp.cpp  # the addon
 |--- index.js     # program to test the addon
 |--- sample.png   # input (will be converted to bmp)
 |--- lodepng.h    # from lodepng distribution
 |--- lodepng.cpp  # From loadpng distribution
```

To download the complete addon, head over to https://github.com/freezer333/nodecpp-demo, this particular example is in the `buffers` directory.

`lodepng.cpp` contains all the necessary code for doing image processing, and I will not discuss it's working in detail. In addition, the lodepng distribution contains sample code that allows you to specifically convert between png and bmp - I've adapted it slightly, and will put it in the addon source code file `png2bmp.cpp` which we will take a look at shortly. Let's first look at what the actual JavaScript program looks like though - before diving into the addon code itself:

```
'use strict';
const fs = require('fs');
const path = require('path');
const png2bmp = require('./build/Release/png2bmp');
var png_file = process.argv[2];
var bmp_file = path.basename(png_file, '.png') + ".bmp";
var png_buffer = fs.readFileSync(png_file);

png2bmp.saveBMP(bmp_file, png_buffer, png_buffer.length);
```

This program simply requires `fs`, `path`, and our addon, which will be located at `./build/Releast/png2bmp`. The program grabs an input (png) filename from the command line arguments and **reads the png into a `Buffer`**. It then sends the `Buffer` into the addon, which saves the converted BMP to the filename as

specified. Thus, the addon is converting png to BMP and saving the results to a file - returning nothing.

Here's the `package.json`, which is setting up `npm start` to invoke the `index.js` program with a command line argument of `sample.png`. It's a pretty generic image:

```
{
  "name": "png2bmp",
  "version": "0.0.1",
  "private": true,
  "gypfile": true,
  "scripts": {
    "start": "node index.js sample.png"
  },
  "dependencies": {
    "nan": "*"
  }
}
```

Finally, let's take a look at the `binding.gyp` file - which is fairly standard, other than the presence of a few compiler flags needed to compile lodepng. It also includes the requisite references to NAN.

```
{
  "targets": [
    {
      "target_name": "png2bmp",
      "sources": [ "png2bmp.cpp", "lodepng.cpp" ],
      "cflags": ["-Wall", "-Wextra", "-pedantic", "-ansi", "-O3"],
      "include_dirs" : ["<!(node -e \"require('nan')\")"]
    }
  ]
}
```

`png2bmp.cpp` will mostly contain V8/NAN code, however it does have one image processing utility function - `do_convert`, adapted from lodepng's png to bmp example code. The function accepts a `vector<unsigned char>` containing input data (png format) and a `vector<unsigned char>` to put it's output (bmp format) data into. That function in turn calls `encodeBMP`, which is straight from the lodepng examples. Here is the full code listing of these two functions. The details are not important to understanding addon `Buffer` objects, but are included here for completeness. Our addon entry point(s) will call `do_convert`.

```
/*
ALL LodePNG code in this file is adapted from lodepng's
```

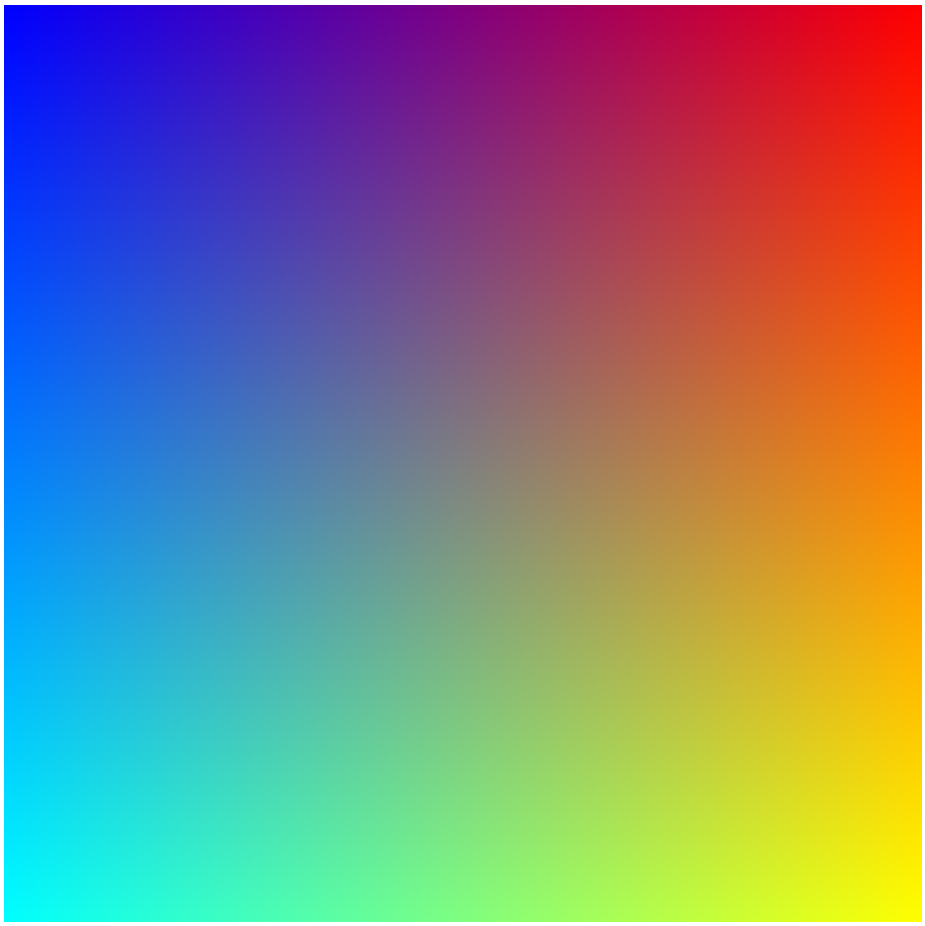


Figure 9: sample.png

```
examples, found at the following URL:
https://github.com/lvandeve/lodepng/blob/
master/examples/example_bmp2png.cpp'
*/

void encodeBMP(std::vector<unsigned char>& bmp,
  const unsigned char* image, int w, int h)
{
  //3 bytes per pixel used for both input and output.
  int inputChannels = 3;
  int outputChannels = 3;

  //bytes 0-13
  bmp.push_back('B'); bmp.push_back('M'); //0: bfType
  bmp.push_back(0); bmp.push_back(0);
  bmp.push_back(0); bmp.push_back(0);
  bmp.push_back(0); bmp.push_back(0); //6: bfReserved1
  bmp.push_back(0); bmp.push_back(0); //8: bfReserved2
  bmp.push_back(54 % 256);
  bmp.push_back(54 / 256);
  bmp.push_back(0); bmp.push_back(0);

  //bytes 14-53
  bmp.push_back(40); bmp.push_back(0);
  bmp.push_back(0); bmp.push_back(0);  //14: biSize
  bmp.push_back(w % 256);
  bmp.push_back(w / 256);
  bmp.push_back(0); bmp.push_back(0); //18: biWidth
  bmp.push_back(h % 256);
  bmp.push_back(h / 256);
  bmp.push_back(0); bmp.push_back(0); //22: biHeight
  bmp.push_back(1); bmp.push_back(0); //26: biPlanes
  bmp.push_back(outputChannels * 8);
  bmp.push_back(0); //28: biBitCount
  bmp.push_back(0); bmp.push_back(0);
  bmp.push_back(0); bmp.push_back(0);  //30: biCompression
  bmp.push_back(0); bmp.push_back(0);
  bmp.push_back(0); bmp.push_back(0);  //34: biSizeImage
  bmp.push_back(0); bmp.push_back(0);
  bmp.push_back(0); bmp.push_back(0);  //38: biXPelsPerMeter
  bmp.push_back(0); bmp.push_back(0);
  bmp.push_back(0); bmp.push_back(0);  //42: biYPelsPerMeter
  bmp.push_back(0); bmp.push_back(0);
  bmp.push_back(0); bmp.push_back(0);  //46: biClrUsed
  bmp.push_back(0); bmp.push_back(0);
  bmp.push_back(0); bmp.push_back(0);  //50: biClrImportant
```

```
  int imagerowbytes = outputChannels * w;
  //must be multiple of 4
  imagerowbytes = imagerowbytes % 4 == 0 ? imagerowbytes :
            imagerowbytes + (4 - imagerowbytes % 4);

  for(int y = h - 1; y >= 0; y--)
  {
    int c = 0;
    for(int x = 0; x < imagerowbytes; x++)
    {
      if(x < w * outputChannels)
      {
        int inc = c;
        //Convert RGB(A) into BGR(A)
        if(c == 0) inc = 2;
        else if(c == 2) inc = 0;
        bmp.push_back(image[inputChannels
            * (w * y + x / outputChannels) + inc]);
      }
      else bmp.push_back(0);
      c++;
      if(c >= outputChannels) c = 0;
    }
  }

  // Fill in the size
  bmp[2] = bmp.size() % 256;
  bmp[3] = (bmp.size() / 256) % 256;
  bmp[4] = (bmp.size() / 65536) % 256;
  bmp[5] = bmp.size() / 16777216;
}


bool do_convert(
  std::vector<unsigned char> & input_data,
  std::vector<unsigned char> & bmp)
{
  std::vector<unsigned char> image; //the raw pixels
  unsigned width, height;
  unsigned error = lodepng::decode(image, width,
    height, input_data, LCT_RGB, 8);
  if(error) {
    std::cout << "error " << error << ": "
              << lodepng_error_text(error)
              << std::endl;
```

```
    return false;
  }
  encodeBMP(bmp, &image[0], width, height);
  return true;
}
```

Sorry... that listing was long, but it's important to see what's actually going on! Let's get to work bridging all this code to JavaScript.

Passing buffers to an addon

Our first task is to create the `saveBMP` addon function, which accepts a filename (destination BMP) along with the png data. The png image data is actually read when we are in JavaScript, so it's passed in as a Node.js `Buffer`. The first rule to recognize is that `Buffer` is unknown to V8, it's a Node.js construct. We'll use NAN to access the buffer itself (and later create new `Buffer` objects). Whenever `Buffer` objects are passed to C++ addons, it is necessary to specify it's length as an added parameter, as it's difficult to ascertain the actual data length of a `Buffer` from C++.

Let's set up the first function call in NAN:

```
NAN_METHOD(SaveBMP) {
    v8::String::Utf8Value val(info[0]->ToString());
    std::string outfile (*val);

    ....
}

NAN_MODULE_INIT(Init) {
   Nan::Set(target,
    New<String>("saveBMP").ToLocalChecked(),
    GetFunction(New<FunctionTemplate>(SaveBMP))
      .ToLocalChecked());
}

NODE_MODULE(basic_nan, Init)
```

In `SaveBMP`, the first step we do is a simple extraction of the output filename. Next, we must extract the binary data (which will be represented by `unsigned char` data).

```
unsigned char* buffer =
  (unsigned char*) node::Buffer::Data(info[1]->ToObject());
```

```
unsigned int size = info[2]->Uint32Value();
```

Notice just how easy this is... Node.js provides a static `Data` method on the `Buffer` class that accepts a standard `v8::Object` handle and returns an `unsigned char` pointer to the underlying data. This pointer **does NOT** point to data managed by V8 recall - it's on the normal C++ heap and can be worked with as such. We also extract the size from the third argument to the addon function.

Often in C++ we prefer to deal with STL containers rather than raw memory arrays, so we can easily create a `vector` from this pointer - which we need to do in order to call `do_convert`. Below is the full code listing - which converts the buffer's data pointer to a vector, calls `do_convert` which works it's magic to fill in bmp data into the vector we give it, and finally saves it to the desired output file (using a lodepng utility call - `save_file`).

```
NAN_METHOD(SaveBMP) {
  v8::String::Utf8Value val(info[0]->ToString());
  std::string outfile (*val);

  // Convert the Node.js Buffer into a C++ Vector
  unsigned char*buffer =
    (unsigned char*) node::Buffer::Data(info[1]->ToObject());

  unsigned int size = info[2]->Uint32Value();
  std::vector<unsigned char> png_data(buffer, buffer + size);

  // Convert to bmp, stored in another vector.
  std::vector<unsigned char> bmp;
  if ( do_convert(png_data, bmp)) {
      info.GetReturnValue().Set(Nan::New(false));
  }
  else {
      lodepng::save_file(bmp, outfile);
      info.GetReturnValue().Set(Nan::New(true));
  }
}
```

Run this program by doing an `npm install` and then an `npm start` and you'll see a `sample.bmp` generated that looks eerily similar to `sample.png`, just a whole lot bigger (bmp compression is far less efficient than png).

Returning buffers from addon

This addon would be a lot more flexible if we simply returned the bitmap image data, rather than needing to save it to a file while within C++. To do this, we must learn how to return `Buffer` objects. This concept, on the surface, seems easy enough - you can look at examples on NAN's website to see new `Buffer`s being created in C++ and returned to JavaScript. Upon closer look though, there are some issues we must be careful with, which we'll tackle here.

Let's create a new addon entry point - `getBMP` - which would be called from JavaScript like so:

```
...
var png_buffer = fs.readFileSync(png_file);

bmp_buffer = png2bmp.getBMP(png_buffer, png_buffer.length);
fs.writeFileSync(bmp_file, bmp_buffer);
```

In the original C++ function, we called `do_convert` which put the bitmap data into a `vector<unsigned int>` which we wrote to a file. Now we must *return* that data, by constructing a new `Buffer` object. NAN's `NewBuffer` call aptly does the trick here - let's look at a first draft of the addon function:

```
void buffer_delete_callback(char* data, void* hint) {
    free(data);
}

NAN_METHOD(GetBMP) {
  unsigned char*buffer =
    (unsigned char*) node::Buffer::Data(info[0]->ToObject());

  unsigned int size = info[1]->Uint32Value();

  std::vector<unsigned char> png_data(buffer, buffer + size);
  std::vector<unsigned char> bmp = vector<unsigned char>();

  if ( do_convert(png_data, bmp)) {
      info.GetReturnValue().Set(
          NewBuffer((char *)bmp.data(),
            bmp.size(), buffer_delete_callback, 0)
            .ToLocalChecked());
    }
}
```

The code example above follows what most tutorials online advocate. We call `NewBuffer` with a char * (which we grab from the `bmp` vector using the `data` method), the size of the amount of memory we are creating the buffer out of, and then 2 additional parameters that might raise your curiosity. The 3rd parameter to `NewBuffer` is a callback - which ends up being called when the `Buffer` you are creating gets garbage collected by V8. Recall, `Buffer`s are JavaScript objects, whose data is stored outside V8 - but the object itself is under V8's control. From this perspective, it should make sense that a callback would be handy - when V8 destroys the buffer, we need some way of freeing up the data we have created - which is passed into the callback as it's first parameter. The signature of the callback is defined by NAN - `Nan::FreeCallback()`. The seconds parameter is a hint to aid in deallocation, we can use it however we want. It will be helpful soon, but for now we just pass null (0).

So - *here is the problem* with this code: The data contained in the buffer we return is likely deleted before our JavaScript gets to use it. Why? If you understand C++ well, you likely already see the problem: the `bmp` vector is going to go out of scope as our `GetBMP` function returns. C++ vector semantics hold that when the vector goes out of scope, the vector's destructor will delete all data within the vector - in this case, our bmp data! This is a huge problem, since the `Buffer` we send back to JavaScript will have it's data deleted out from under it. You might get away with this (race conditions are fun right?), but it wil eventually cause your program to crash.

How do we get around this? One method is to create a `Buffer` containing a *copy* of the `bmp` vector's data. We could do this like so:

```
if ( do_convert(png_data, bmp)) {
        info.GetReturnValue().Set(
            CopyBuffer(
              (char *)bmp.data(),
              bmp.size()).ToLocalChecked());
}
```

This indeed is safe, but it involves creating a copy of the data - slow and memory wasting... One way to avoid this whole mess is not to use a vector, and store the bitmap data in a dynamically allocated `char *` array - however that makes the bitmap conversion code a lot more cumbersome. Thankfully, the answer to this problem, which allows us to still use vectors, is suggested by the the `Nan::FreeCallback` call signature - namely the `hint` parameter. Since our problem is that the vector containing bitmap data goes out of scope, we can instead *dynamically* allocate the vector itself, and pass it into the free callback, where it can be properly deleted when the `Buffer` has been garbage collected. Below is the completed solution - take careful note now that we are utilizing the `hint` parameter in our callback, and that we are using a dynamically allocated (heap) vector instead of a stack variable.

```
void buffer_delete_callback(char* data, void* the_vector) {
  delete reinterpret_cast<vector<unsigned char> *> (the_vector);
}

NAN_METHOD(GetBMP) {

  unsigned char*buffer =
    (unsigned char*) node::Buffer::Data(info[0]->ToObject());
  unsigned int size = info[1]->Uint32Value();

  std::vector<unsigned char> png_data(buffer, buffer + size);

  // allocate the vector on the heap because we
  // are building a buffer out of it's data to
  // return to Node - and don't want to allow
  // it to go out of scope until the buffer
  // does (see buffer_delete_callback).

  std::vector<unsigned char> * bmp = new vector<unsigned char>();

  if ( do_convert(png_data, *bmp)) {
      info.GetReturnValue().Set(
          NewBuffer((char *)bmp->data(),
            bmp->size(), buffer_delete_callback, bmp)
            .ToLocalChecked());
  }
}
```

When you run the program, JavaScript will now safely be able to operate on the returned `Buffer` without needing to worry about `vector` deleting the memory.

Buffers as a solution to the worker thread copying problem

Reading the segment above, you might recall a discussion in Chapter 4 regarding V8 memory and worker threads. We had a significant issue when using asynchronous addons, in that C++ threads created to do the asynchronous work could *never* access V8 data directly. There was no real solution to this problem, other than creating a copy of the data in C++ heap space. For lots of addons, this if fine - however as was suggested at the time, when moving large amounts of data between JavaScript and C++ this is a real issue. Now we have a glimpse at a possible solution - allocating data as `Buffer` objects!

Let's develop an asynchronous version of the png to bitmap converter. We'll perform the actual conversion in a C++ worker thread, using `Nan::AsyncWorker`. Through the use of `Buffer` objects however, we will be no need to create a copy

of the png data - we will only need to hold a pointer to the underlying data such that our worker thread can access it. Likewise, the data produced by the worker thread (the `bmp` vector can be used to create a new `Buffer` without copying data, as shown above. Since we've worked with `AsyncWorker` a lot already in this book, I'll simply show you the code below - it's pretty straightforward:

```
 class PngToBmpWorker : public AsyncWorker {
    public:
    PngToBmpWorker(Callback * callback,
        v8::Local<v8::Object> &pngBuffer, int size)
        : AsyncWorker(callback) {
        unsigned char*buffer =
          (unsigned char*) node::Buffer::Data(pngBuffer);

        std::vector<unsigned char> tmp(
          buffer,
          buffer +  (unsigned int) size);

        png_data = tmp;
    }
    void Execute() {
       bmp = new vector<unsigned char>();
       do_convert(png_data, *bmp);
    }
    void HandleOKCallback () {
        Local<Object> bmpData =
               NewBuffer((char *)bmp->data(),
               bmp->size(), buffer_delete_callback,
               bmp).ToLocalChecked();
        Local<Value> argv[] = { bmpData };
        callback->Call(1, argv);
    }

    private:
        vector<unsigned char> png_data;
        std::vector<unsigned char> * bmp;
};

NAN_METHOD(GetBMPAsync) {
    int size = To<int>(info[1]).FromJust();
    v8::Local<v8::Object> pngBuffer =
      info[0]->ToObject();

    Callback *callback =
      new Callback(info[2].As<Function>());
```

```
    AsyncQueueWorker(
      new PngToBmpWorker(callback, pngBuffer , size));
}
```

Now we've got an asynchronous function to get the bitmap encoded data too - without any copying of data unnecessarily.

```
png2bmp.getBMPAsync(png_buffer,
  png_buffer.length,
  function(bmp_buffer) {
    fs.writeFileSync(bmp_file, bmp_buffer);
});
```

Closing

I really hope you've learned some useful new things from this book. I've enjoyed writing it, and I'm excited to see how Node and C++ addons will evolve in the future.

The V8 runtime is evolving, and there are many important features that I haven't even touched upon in this book, such as support for exceptions, maybe types, promises, typed arrays, etc.). Node.js doesn't necessarily need to be bound to V8 though - while its still a work in progress, the Node-Chakra is allowing Node.js to run on top of Microsoft Edge's JavaScript engine as well. Using different engines will place an even greater importance on projects like NAN, as it will become nearly impossible to natively target V8 if you hope for wide-spread adoption.

The JavaScript and C++ space isn't limited to just addons - there are really exciting projects like Emscripten that allow C and C++ binaries to be transpiled to JavaScript so you can run C++ programs in the browser. Pushing this further, nbind allows developers to simultaneously target Node.js and browsers with C/C++ code using compiler directives and headers - which could completely change the way C++ is integrated with JavaScript.

If you've liked this book, please visit my blog - blog.scottfrees.com to access more content about this topic, and maybe even leave a nice review!

Thanks -