C++ addons

Addons are dynamically-linked shared objects written in C++. The `require()` function can load addons as ordinary Node.js modules. Addons provide an interface between JavaScript and C/C++ libraries.

There are three options for implementing addons: Node-API, nan, or direct use of internal V8, libuv and Node.js libraries. Unless there is a need for direct access to functionality which is not exposed by Node-API, use Node-API. Refer to C/C++ addons with Node-API for more information on Node-API.

When not using Node-API, implementing addons is complicated, involving knowledge of several components and APIs:

- V8: the C++ library Node.js uses to provide the JavaScript implementation. V8 provides the mechanisms for creating objects, calling functions, etc. V8's API is documented mostly in the `v8.h` header file (`deps/v8/include/v8.h` in the Node.js source tree), which is also available online.
- libuv: The C library that implements the Node.js event loop, its worker threads and all of the asynchronous behaviors of the platform. It also serves as a cross-platform abstraction library, giving easy, POSIX-like access across all major operating systems to many common system tasks, such as interacting with the filesystem, sockets, timers, and system events. libuv also provides a threading abstraction similar to POSIX threads for more sophisticated asynchronous addons that need to move beyond the standard event loop. Addon authors should avoid blocking the event loop with I/O or other time-intensive tasks by offloading work via libuv to non-blocking system operations, worker threads, or a custom use of libuv threads.
- Internal Node.js libraries. Node.js itself exports C++ APIs that addons can use, the most important of which is the `node::ObjectWrap` class.
- Node.js includes other statically linked libraries including OpenSSL. These other libraries are located in the `deps/` directory in the Node.js source tree. Only the libuv, OpenSSL, V8 and zlib symbols are purposefully re-exported by Node.js and may be used to various extents by addons. See Linking to libraries included with Node.js for additional information.

All of the following examples are available for download and may be used as the starting-point for an addon.

Hello world

This "Hello world" example is a simple addon, written in C++, that is the equivalent of the following JavaScript code:

```
module.exports.hello = () => 'world';
```

First, create the file `hello.cc`:

```
// hello.cc
#include <node.h>

namespace demo {

using v8::FunctionCallbackInfo;
using v8::Isolate;
using v8::Local;
using v8::Object;
```

```
using v8::String;
using v8::Value;

void Method(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();
  args.GetReturnValue().Set(String::NewFromUtf8(
      isolate, "world").ToLocalChecked());
}

void Initialize(Local<Object> exports) {
  NODE_SET_METHOD(exports, "hello", Method);
}

NODE_MODULE(NODE_GYP_MODULE_NAME, Initialize)

}  // namespace demo
```

All Node.js addons must export an initialization function following the pattern:

```
void Initialize(Local<Object> exports);
NODE_MODULE(NODE_GYP_MODULE_NAME, Initialize)
```

There is no semi-colon after `NODE_MODULE` as it's not a function (see `node.h`).

The `module_name` must match the filename of the final binary (excluding the `.node` suffix).

In the `hello.cc` example, then, the initialization function is `Initialize` and the addon module name is `addon`.

When building addons with `node-gyp`, using the macro `NODE_GYP_MODULE_NAME` as the first parameter of `NODE_MODULE()` will ensure that the name of the final binary will be passed to `NODE_MODULE()`.

Context-aware addons

There are environments in which Node.js addons may need to be loaded multiple times in multiple contexts. For example, the Electron runtime runs multiple instances of Node.js in a single process. Each instance will have its own `require()` cache, and thus each instance will need a native addon to behave correctly when loaded via `require()`. This means that the addon must support multiple initializations.

A context-aware addon can be constructed by using the macro `NODE_MODULE_INITIALIZER`, which expands to the name of a function which Node.js will expect to find when it loads an addon. An addon can thus be initialized as in the following example:

```
using namespace v8;

extern "C" NODE_MODULE_EXPORT void
NODE_MODULE_INITIALIZER(Local<Object> exports,
                        Local<Value> module,
                        Local<Context> context) {
  /* Perform addon initialization steps here. */
}
```

Another option is to use the macro `NODE_MODULE_INIT()`, which will also construct a context-aware addon. Unlike `NODE_MODULE()`, which is used to construct an addon around a given addon initializer function, `NODE_MODULE_INIT()` serves as the declaration of such an initializer to be followed by a function body.

The following three variables may be used inside the function body following an invocation of `NODE_MODULE_INIT()`:

- `Local<Object> exports`,
- `Local<Value> module`, and
- `Local<Context> context`

The choice to build a context-aware addon carries with it the responsibility of carefully managing global static data. Since the addon may be loaded multiple times, potentially even from different threads, any global static data stored in the addon must be properly protected, and must not contain any persistent references to JavaScript objects. The reason for this is that JavaScript objects are only valid in one context, and will likely cause a crash when accessed from the wrong context or from a different thread than the one on which they were created.

The context-aware addon can be structured to avoid global static data by performing the following steps:

- Define a class which will hold per-addon-instance data and which has a static member of the form

  ```
  static void DeleteInstance(void* data) {
    // Cast `data` to an instance of the class and delete it.
  }
  ```

- Heap-allocate an instance of this class in the addon initializer. This can be accomplished using the `new` keyword.
- Call `node::AddEnvironmentCleanupHook()`, passing it the above-created instance and a pointer to `DeleteInstance()`. This will ensure the instance is deleted when the environment is torn down.
- Store the instance of the class in a `v8::External`, and
- Pass the `v8::External` to all methods exposed to JavaScript by passing it to `v8::FunctionTemplate::New()` or `v8::Function::New()` which creates the native-backed JavaScript functions. The third parameter of `v8::FunctionTemplate::New()` or `v8::Function::New()` accepts the `v8::External` and makes it available in the native callback using the `v8::FunctionCallbackInfo::Data()` method.

This will ensure that the per-addon-instance data reaches each binding that can be called from JavaScript. The per-addon-instance data must also be passed into any asynchronous callbacks the addon may create.

The following example illustrates the implementation of a context-aware addon:

```
#include <node.h>

using namespace v8;

class AddonData {
 public:
  explicit AddonData(Isolate* isolate):
      call_count(0) {
    // Ensure this per-addon-instance data is deleted at environment cleanup.
    node::AddEnvironmentCleanupHook(isolate, DeleteInstance, this);
  }
```

```
  // Per-addon data.
  int call_count;

  static void DeleteInstance(void* data) {
    delete static_cast<AddonData*>(data);
  }
};

static void Method(const v8::FunctionCallbackInfo<v8::Value>& info) {
  // Retrieve the per-addon-instance data.
  AddonData* data =
      reinterpret_cast<AddonData*>(info.Data().As<External>()->Value());
  data->call_count++;
  info.GetReturnValue().Set((double)data->call_count);
}

// Initialize this addon to be context-aware.
NODE_MODULE_INIT(/* exports, module, context */) {
  Isolate* isolate = context->GetIsolate();

  // Create a new instance of `AddonData` for this instance of the addon and
  // tie its life cycle to that of the Node.js environment.
  AddonData* data = new AddonData(isolate);

  // Wrap the data in a `v8::External` so we can pass it to the method we
  // expose.
  Local<External> external = External::New(isolate, data);

  // Expose the method `Method` to JavaScript, and make sure it receives the
  // per-addon-instance data we created above by passing `external` as the
  // third parameter to the `FunctionTemplate` constructor.
  exports->Set(context,
               String::NewFromUtf8(isolate, "method").ToLocalChecked(),
               FunctionTemplate::New(isolate, Method, external)
                  ->GetFunction(context).ToLocalChecked()).FromJust();
}
```

Worker support

In order to be loaded from multiple Node.js environments, such as a main thread and a Worker thread, an add-on needs to either:

- Be an Node-API addon, or
- Be declared as context-aware using `NODE_MODULE_INIT()` as described above

In order to support `Worker` threads, addons need to clean up any resources they may have allocated when such a thread exists. This can be achieved through the usage of the `AddEnvironmentCleanupHook()` function:

```
void AddEnvironmentCleanupHook(v8::Isolate* isolate,
                               void (*fun)(void* arg),
                               void* arg);
```

This function adds a hook that will run before a given Node.js instance shuts down. If necessary, such hooks can be removed before they are run using `RemoveEnvironmentCleanupHook()`, which has the same signature. Callbacks are run in last-in first-out order.

If necessary, there is an additional pair of `AddEnvironmentCleanupHook()` and `RemoveEnvironmentCleanupHook()` overloads, where the cleanup hook takes a callback function. This can be used for shutting down asynchronous resources, such as any libuv handles registered by the addon.

The following `addon.cc` uses `AddEnvironmentCleanupHook`:

```
// addon.cc
#include <node.h>
#include <assert.h>
#include <stdlib.h>

using node::AddEnvironmentCleanupHook;
using v8::HandleScope;
using v8::Isolate;
using v8::Local;
using v8::Object;

// Note: In a real-world application, do not rely on static/global data.
static char cookie[] = "yum yum";
static int cleanup_cb1_called = 0;
static int cleanup_cb2_called = 0;

static void cleanup_cb1(void* arg) {
  Isolate* isolate = static_cast<Isolate*>(arg);
  HandleScope scope(isolate);
  Local<Object> obj = Object::New(isolate);
  assert(!obj.IsEmpty());  // assert VM is still alive
  assert(obj->IsObject());
  cleanup_cb1_called++;
}

static void cleanup_cb2(void* arg) {
  assert(arg == static_cast<void*>(cookie));
  cleanup_cb2_called++;
}

static void sanity_check(void*) {
  assert(cleanup_cb1_called == 1);
  assert(cleanup_cb2_called == 1);
}

// Initialize this addon to be context-aware.
NODE_MODULE_INIT(/* exports, module, context */) {
  Isolate* isolate = context->GetIsolate();

  AddEnvironmentCleanupHook(isolate, sanity_check, nullptr);
  AddEnvironmentCleanupHook(isolate, cleanup_cb2, cookie);
```

```
  AddEnvironmentCleanupHook(isolate, cleanup_cb1, isolate);
}
```

Test in JavaScript by running:

```
// test.js
require('./build/Release/addon');
```

Building

Once the source code has been written, it must be compiled into the binary `addon.node` file. To do so, create a file called `binding.gyp` in the top-level of the project describing the build configuration of the module using a JSON-like format. This file is used by node-gyp, a tool written specifically to compile Node.js addons.

```
{
  "targets": [
    {
      "target_name": "addon",
      "sources": [ "hello.cc" ]
    }
  ]
}
```

A version of the `node-gyp` utility is bundled and distributed with Node.js as part of `npm`. This version is not made directly available for developers to use and is intended only to support the ability to use the `npm install` command to compile and install addons. Developers who wish to use `node-gyp` directly can install it using the command `npm install -g node-gyp`. See the `node-gyp` installation instructions for more information, including platform-specific requirements.

Once the `binding.gyp` file has been created, use `node-gyp configure` to generate the appropriate project build files for the current platform. This will generate either a `Makefile` (on Unix platforms) or a `vcxproj` file (on Windows) in the `build/` directory.

Next, invoke the `node-gyp build` command to generate the compiled `addon.node` file. This will be put into the `build/Release/` directory.

When using `npm install` to install a Node.js addon, npm uses its own bundled version of `node-gyp` to perform this same set of actions, generating a compiled version of the addon for the user's platform on demand.

Once built, the binary addon can be used from within Node.js by pointing `require()` to the built `addon.node` module:

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

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

Because the exact path to the compiled addon binary can vary depending on how it is compiled (i.e. sometimes it may be in `./build/Debug/`), addons can use the bindings package to load the compiled module.

While the `bindings` package implementation is more sophisticated in how it locates addon modules, it is essentially using a `try…catch` pattern similar to:

```
try {
  return require('./build/Release/addon.node');
} catch (err) {
  return require('./build/Debug/addon.node');
}
```

Linking to libraries included with Node.js

Node.js uses statically linked libraries such as V8, libuv and OpenSSL. All addons are required to link to V8 and may link to any of the other dependencies as well. Typically, this is as simple as including the appropriate `#include <...>` statements (e.g. `#include <v8.h>`) and `node-gyp` will locate the appropriate headers automatically. However, there are a few caveats to be aware of:

- When `node-gyp` runs, it will detect the specific release version of Node.js and download either the full source tarball or just the headers. If the full source is downloaded, addons will have complete access to the full set of Node.js dependencies. However, if only the Node.js headers are downloaded, then only the symbols exported by Node.js will be available.
- `node-gyp` can be run using the `--nodedir` flag pointing at a local Node.js source image. Using this option, the addon will have access to the full set of dependencies.

Loading addons using `require()`

The filename extension of the compiled addon binary is `.node` (as opposed to `.dll` or `.so`). The `require()` function is written to look for files with the `.node` file extension and initialize those as dynamically-linked libraries.

When calling `require()`, the `.node` extension can usually be omitted and Node.js will still find and initialize the addon. One caveat, however, is that Node.js will first attempt to locate and load modules or JavaScript files that happen to share the same base name. For instance, if there is a file `addon.js` in the same directory as the binary `addon.node`, then `require('addon')` will give precedence to the `addon.js` file and load it instead.

Native abstractions for Node.js

Each of the examples illustrated in this document directly use the Node.js and V8 APIs for implementing addons. The V8 API can, and has, changed dramatically from one V8 release to the next (and one major Node.js release to the next). With each change, addons may need to be updated and recompiled in order to continue functioning. The Node.js release schedule is designed to minimize the frequency and impact of such changes but there is little that Node.js can do to ensure stability of the V8 APIs.

The Native Abstractions for Node.js (or `nan`) provide a set of tools that addon developers are recommended to use to keep compatibility between past and future releases of V8 and Node.js. See the `nan` examples for an illustration of how it can be used.

Node-API

> *Stability: 2 - Stable*

Node-API is an API for building native addons. It is independent from the underlying JavaScript runtime (e.g. V8) and is maintained as part of Node.js itself. This API will be Application Binary Interface (ABI) stable across versions of Node.js. It is intended to insulate addons from changes in the underlying JavaScript engine and allow modules compiled for one version to run on later versions of Node.js without recompilation. Addons are built/packaged with the same approach/tools outlined in this document (node-gyp, etc.). The only difference is the set of APIs that are used by the native code. Instead of using the V8 or Native Abstractions for Node.js APIs, the functions available in the Node-API are used.

Creating and maintaining an addon that benefits from the ABI stability provided by Node-API carries with it certain implementation considerations.

To use Node-API in the above "Hello world" example, replace the content of `hello.cc` with the following. All other instructions remain the same.

```
// hello.cc using Node-API
#include <node_api.h>

namespace demo {

napi_value Method(napi_env env, napi_callback_info args) {
  napi_value greeting;
  napi_status status;

  status = napi_create_string_utf8(env, "world", NAPI_AUTO_LENGTH, &greeting);
  if (status != napi_ok) return nullptr;
  return greeting;
}

napi_value init(napi_env env, napi_value exports) {
  napi_status status;
  napi_value fn;

  status = napi_create_function(env, nullptr, 0, Method, nullptr, &fn);
  if (status != napi_ok) return nullptr;

  status = napi_set_named_property(env, exports, "hello", fn);
  if (status != napi_ok) return nullptr;
  return exports;
}

NAPI_MODULE(NODE_GYP_MODULE_NAME, init)

}  // namespace demo
```

The functions available and how to use them are documented in C/C++ addons with Node-API.

Addon examples

Following are some example addons intended to help developers get started. The examples use the V8 APIs. Refer to the online V8 reference for help with the various V8 calls, and V8's Embedder's Guide for an explanation of several concepts used such as handles, scopes, function templates, etc.

Each of these examples using the following `binding.gyp` file:

```
{
  "targets": [
    {
      "target_name": "addon",
      "sources": [ "addon.cc" ]
    }
  ]
}
```

In cases where there is more than one `.cc` file, simply add the additional filename to the `sources` array:

```
"sources": ["addon.cc", "myexample.cc"]
```

Once the `binding.gyp` file is ready, the example addons can be configured and built using `node-gyp`:

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

Function arguments

Addons will typically expose objects and functions that can be accessed from JavaScript running within Node.js. When functions are invoked from JavaScript, the input arguments and return value must be mapped to and from the C/C++ code.

The following example illustrates how to read function arguments passed from JavaScript and how to return a result:

```
// addon.cc
#include <node.h>

namespace demo {

using v8::Exception;
using v8::FunctionCallbackInfo;
using v8::Isolate;
using v8::Local;
using v8::Number;
using v8::Object;
using v8::String;
using v8::Value;

// This is the implementation of the "add" method
// Input arguments are passed using the
// const FunctionCallbackInfo<Value>& args struct
void Add(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();

  // Check the number of arguments passed.
  if (args.Length() < 2) {
    // Throw an Error that is passed back to JavaScript
```

```
    isolate->ThrowException(Exception::TypeError(
        String::NewFromUtf8(isolate,
                            "Wrong number of arguments").ToLocalChecked()));
    return;
  }

  // Check the argument types
  if (!args[0]->IsNumber() || !args[1]->IsNumber()) {
    isolate->ThrowException(Exception::TypeError(
        String::NewFromUtf8(isolate,
                            "Wrong arguments").ToLocalChecked()));
    return;
  }

  // Perform the operation
  double value =
      args[0].As<Number>()->Value() + args[1].As<Number>()->Value();
  Local<Number> num = Number::New(isolate, value);

  // Set the return value (using the passed in
  // FunctionCallbackInfo<Value>&)
  args.GetReturnValue().Set(num);
}

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

NODE_MODULE(NODE_GYP_MODULE_NAME, Init)

}  // namespace demo
```

Once compiled, the example addon can be required and used from within Node.js:

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

console.log('This should be eight:', addon.add(3, 5));
```

Callbacks

It is common practice within addons to pass JavaScript functions to a C++ function and execute them from there. The following example illustrates how to invoke such callbacks:

```
// addon.cc
#include <node.h>

namespace demo {

using v8::Context;
using v8::Function;
```

```
using v8::FunctionCallbackInfo;
using v8::Isolate;
using v8::Local;
using v8::Null;
using v8::Object;
using v8::String;
using v8::Value;

void RunCallback(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();
  Local<Context> context = isolate->GetCurrentContext();
  Local<Function> cb = Local<Function>::Cast(args[0]);
  const unsigned argc = 1;
  Local<Value> argv[argc] = {
      String::NewFromUtf8(isolate,
                          "hello world").ToLocalChecked() };
  cb->Call(context, Null(isolate), argc, argv).ToLocalChecked();
}

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

NODE_MODULE(NODE_GYP_MODULE_NAME, Init)

}  // namespace demo
```

This example uses a two-argument form of `Init()` that receives the full `module` object as the second argument. This allows the addon to completely overwrite `exports` with a single function instead of adding the function as a property of `exports`.

To test it, run the following JavaScript:

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

addon((msg) => {
  console.log(msg);
// Prints: 'hello world'
});
```

In this example, the callback function is invoked synchronously.

Object factory

Addons can create and return new objects from within a C++ function as illustrated in the following example. An object is created and returned with a property `msg` that echoes the string passed to `createObject()`:

```
// addon.cc
#include <node.h>
```

```
namespace demo {

using v8::Context;
using v8::FunctionCallbackInfo;
using v8::Isolate;
using v8::Local;
using v8::Object;
using v8::String;
using v8::Value;

void CreateObject(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();
  Local<Context> context = isolate->GetCurrentContext();

  Local<Object> obj = Object::New(isolate);
  obj->Set(context,
           String::NewFromUtf8(isolate,
                               "msg").ToLocalChecked(),
                               args[0]->ToString(context).ToLocalChecked())
           .FromJust();

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

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

NODE_MODULE(NODE_GYP_MODULE_NAME, Init)

}  // namespace demo
```

To test it in JavaScript:

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

const obj1 = addon('hello');
const obj2 = addon('world');
console.log(obj1.msg, obj2.msg);
// Prints: 'hello world'
```

Function factory

Another common scenario is creating JavaScript functions that wrap C++ functions and returning those back to JavaScript:

```
// addon.cc
#include <node.h>

namespace demo {
```

```
using v8::Context;
using v8::Function;
using v8::FunctionCallbackInfo;
using v8::FunctionTemplate;
using v8::Isolate;
using v8::Local;
using v8::Object;
using v8::String;
using v8::Value;

void MyFunction(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();
  args.GetReturnValue().Set(String::NewFromUtf8(
      isolate, "hello world").ToLocalChecked());
}

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

  Local<Context> context = isolate->GetCurrentContext();
  Local<FunctionTemplate> tpl = FunctionTemplate::New(isolate, MyFunction);
  Local<Function> fn = tpl->GetFunction(context).ToLocalChecked();

  // omit this to make it anonymous
  fn->SetName(String::NewFromUtf8(
      isolate, "theFunction").ToLocalChecked());

  args.GetReturnValue().Set(fn);
}

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

NODE_MODULE(NODE_GYP_MODULE_NAME, Init)

}  // namespace demo
```

To test:

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

const fn = addon();
console.log(fn());
// Prints: 'hello world'
```

Wrapping C++ objects

It is also possible to wrap C++ objects/classes in a way that allows new instances to be created using the JavaScript `new` operator:

```
// addon.cc
#include <node.h>
#include "myobject.h"

namespace demo {

using v8::Local;
using v8::Object;

void InitAll(Local<Object> exports) {
  MyObject::Init(exports);
}

NODE_MODULE(NODE_GYP_MODULE_NAME, InitAll)

}  // namespace demo
```

Then, in `myobject.h`, the wrapper class inherits from `node::ObjectWrap`:

```
// myobject.h
#ifndef MYOBJECT_H
#define MYOBJECT_H

#include <node.h>
#include <node_object_wrap.h>

namespace demo {

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

 private:
  explicit MyObject(double value = 0);
  ~MyObject();

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

  double value_;
};

}  // namespace demo

#endif
```

In `myobject.cc`, implement the various methods that are to be exposed. Below, the method `plusOne()` is exposed by adding it to the constructor's prototype:

```
// myobject.cc
#include "myobject.h"

namespace demo {

using v8::Context;
using v8::Function;
using v8::FunctionCallbackInfo;
using v8::FunctionTemplate;
using v8::Isolate;
using v8::Local;
using v8::Number;
using v8::Object;
using v8::ObjectTemplate;
using v8::String;
using v8::Value;

MyObject::MyObject(double value) : value_(value) {
}

MyObject::~MyObject() {
}

void MyObject::Init(Local<Object> exports) {
  Isolate* isolate = exports->GetIsolate();
  Local<Context> context = isolate->GetCurrentContext();

  Local<ObjectTemplate> addon_data_tpl = ObjectTemplate::New(isolate);
  addon_data_tpl->SetInternalFieldCount(1);  // 1 field for the MyObject::New()
  Local<Object> addon_data =
      addon_data_tpl->NewInstance(context).ToLocalChecked();

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

  // Prototype
  NODE_SET_PROTOTYPE_METHOD(tpl, "plusOne", PlusOne);

  Local<Function> constructor = tpl->GetFunction(context).ToLocalChecked();
  addon_data->SetInternalField(0, constructor);
  exports->Set(context, String::NewFromUtf8(
      isolate, "MyObject").ToLocalChecked(),
      constructor).FromJust();
}

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

```
  Isolate* isolate = args.GetIsolate();
  Local<Context> context = isolate->GetCurrentContext();

  if (args.IsConstructCall()) {
    // Invoked as constructor: `new MyObject(...)`
    double value = args[0]->IsUndefined() ?
        0 : args[0]->NumberValue(context).FromMaybe(0);
    MyObject* obj = new MyObject(value);
    obj->Wrap(args.This());
    args.GetReturnValue().Set(args.This());
  } else {
    // Invoked as plain function `MyObject(...)`, turn into construct call.
    const int argc = 1;
    Local<Value> argv[argc] = { args[0] };
    Local<Function> cons =
        args.Data().As<Object>()->GetInternalField(0).As<Function>();
    Local<Object> result =
        cons->NewInstance(context, argc, argv).ToLocalChecked();
    args.GetReturnValue().Set(result);
  }
}

void MyObject::PlusOne(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();

  MyObject* obj = ObjectWrap::Unwrap<MyObject>(args.Holder());
  obj->value_ += 1;

  args.GetReturnValue().Set(Number::New(isolate, obj->value_));
}

}  // namespace demo
```

To build this example, the `myobject.cc` file must be added to the `binding.gyp`:

```
{
  "targets": [
    {
      "target_name": "addon",
      "sources": [
        "addon.cc",
        "myobject.cc"
      ]
    }
  ]
}
```

Test it with:

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

```
const obj = new addon.MyObject(10);
console.log(obj.plusOne());
// Prints: 11
console.log(obj.plusOne());
// Prints: 12
console.log(obj.plusOne());
// Prints: 13
```

The destructor for a wrapper object will run when the object is garbage-collected. For destructor testing, there are command-line flags that can be used to make it possible to force garbage collection. These flags are provided by the underlying V8 JavaScript engine. They are subject to change or removal at any time. They are not documented by Node.js or V8, and they should never be used outside of testing.

During shutdown of the process or worker threads destructors are not called by the JS engine. Therefore it's the responsibility of the user to track these objects and ensure proper destruction to avoid resource leaks.

Factory of wrapped objects

Alternatively, it is possible to use a factory pattern to avoid explicitly creating object instances using the JavaScript `new` operator:

```
const obj = addon.createObject();
// instead of:
// const obj = new addon.Object();
```

First, the `createObject()` method is implemented in `addon.cc`:

```
// addon.cc
#include <node.h>
#include "myobject.h"

namespace demo {

using v8::FunctionCallbackInfo;
using v8::Isolate;
using v8::Local;
using v8::Object;
using v8::String;
using v8::Value;

void CreateObject(const FunctionCallbackInfo<Value>& args) {
  MyObject::NewInstance(args);
}

void InitAll(Local<Object> exports, Local<Object> module) {
  MyObject::Init(exports->GetIsolate());

  NODE_SET_METHOD(module, "exports", CreateObject);
}
```

```
NODE_MODULE(NODE_GYP_MODULE_NAME, InitAll)

}  // namespace demo
```

In `myobject.h`, the static method `NewInstance()` is added to handle instantiating the object. This method takes the place of using `new` in JavaScript:

```
// myobject.h
#ifndef MYOBJECT_H
#define MYOBJECT_H

#include <node.h>
#include <node_object_wrap.h>

namespace demo {

class MyObject : public node::ObjectWrap {
 public:
  static void Init(v8::Isolate* isolate);
  static void NewInstance(const v8::FunctionCallbackInfo<v8::Value>& args);

 private:
  explicit MyObject(double value = 0);
  ~MyObject();

  static void New(const v8::FunctionCallbackInfo<v8::Value>& args);
  static void PlusOne(const v8::FunctionCallbackInfo<v8::Value>& args);
  static v8::Global<v8::Function> constructor;
  double value_;
};

}  // namespace demo

#endif
```

The implementation in `myobject.cc` is similar to the previous example:

```
// myobject.cc
#include <node.h>
#include "myobject.h"

namespace demo {

using node::AddEnvironmentCleanupHook;
using v8::Context;
using v8::Function;
using v8::FunctionCallbackInfo;
using v8::FunctionTemplate;
using v8::Global;
using v8::Isolate;
```

```
using v8::Local;
using v8::Number;
using v8::Object;
using v8::String;
using v8::Value;

// Warning! This is not thread-safe, this addon cannot be used for worker
// threads.
Global<Function> MyObject::constructor;

MyObject::MyObject(double value) : value_(value) {
}

MyObject::~MyObject() {
}

void MyObject::Init(Isolate* isolate) {
  // Prepare constructor template
  Local<FunctionTemplate> tpl = FunctionTemplate::New(isolate, New);
  tpl->SetClassName(String::NewFromUtf8(isolate, "MyObject").ToLocalChecked());
  tpl->InstanceTemplate()->SetInternalFieldCount(1);

  // Prototype
  NODE_SET_PROTOTYPE_METHOD(tpl, "plusOne", PlusOne);

  Local<Context> context = isolate->GetCurrentContext();
  constructor.Reset(isolate, tpl->GetFunction(context).ToLocalChecked());

  AddEnvironmentCleanupHook(isolate, [](void*) {
    constructor.Reset();
  }, nullptr);
}

void MyObject::New(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();
  Local<Context> context = isolate->GetCurrentContext();

  if (args.IsConstructCall()) {
    // Invoked as constructor: `new MyObject(...)`
    double value = args[0]->IsUndefined() ?
        0 : args[0]->NumberValue(context).FromMaybe(0);
    MyObject* obj = new MyObject(value);
    obj->Wrap(args.This());
    args.GetReturnValue().Set(args.This());
  } else {
    // Invoked as plain function `MyObject(...)`, turn into construct call.
    const int argc = 1;
    Local<Value> argv[argc] = { args[0] };
    Local<Function> cons = Local<Function>::New(isolate, constructor);
    Local<Object> instance =
        cons->NewInstance(context, argc, argv).ToLocalChecked();
    args.GetReturnValue().Set(instance);
```

```
  }
}

void MyObject::NewInstance(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();

  const unsigned argc = 1;
  Local<Value> argv[argc] = { args[0] };
  Local<Function> cons = Local<Function>::New(isolate, constructor);
  Local<Context> context = isolate->GetCurrentContext();
  Local<Object> instance =
      cons->NewInstance(context, argc, argv).ToLocalChecked();

  args.GetReturnValue().Set(instance);
}

void MyObject::PlusOne(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();

  MyObject* obj = ObjectWrap::Unwrap<MyObject>(args.Holder());
  obj->value_ += 1;

  args.GetReturnValue().Set(Number::New(isolate, obj->value_));
}

}  // namespace demo
```

Once again, to build this example, the `myobject.cc` file must be added to the `binding.gyp`:

```
{
  "targets": [
    {
      "target_name": "addon",
      "sources": [
        "addon.cc",
        "myobject.cc"
      ]
    }
  ]
}
```

Test it with:

```
// test.js
const createObject = require('./build/Release/addon');

const obj = createObject(10);
console.log(obj.plusOne());
// Prints: 11
console.log(obj.plusOne());
// Prints: 12
```

```
console.log(obj.plusOne());
// Prints: 13

const obj2 = createObject(20);
console.log(obj2.plusOne());
// Prints: 21
console.log(obj2.plusOne());
// Prints: 22
console.log(obj2.plusOne());
// Prints: 23
```

Passing wrapped objects around

In addition to wrapping and returning C++ objects, it is possible to pass wrapped objects around by unwrapping them with the Node.js helper function `node::ObjectWrap::Unwrap`. The following examples shows a function `add()` that can take two `MyObject` objects as input arguments:

```
// addon.cc
#include <node.h>
#include <node_object_wrap.h>
#include "myobject.h"

namespace demo {

using v8::Context;
using v8::FunctionCallbackInfo;
using v8::Isolate;
using v8::Local;
using v8::Number;
using v8::Object;
using v8::String;
using v8::Value;

void CreateObject(const FunctionCallbackInfo<Value>& args) {
  MyObject::NewInstance(args);
}

void Add(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();
  Local<Context> context = isolate->GetCurrentContext();

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

  double sum = obj1->value() + obj2->value();
  args.GetReturnValue().Set(Number::New(isolate, sum));
}

void InitAll(Local<Object> exports) {
```

```
  MyObject::Init(exports->GetIsolate());

  NODE_SET_METHOD(exports, "createObject", CreateObject);
  NODE_SET_METHOD(exports, "add", Add);
}

NODE_MODULE(NODE_GYP_MODULE_NAME, InitAll)

}  // namespace demo
```

In `myobject.h`, a new public method is added to allow access to private values after unwrapping the object.

```
// myobject.h
#ifndef MYOBJECT_H
#define MYOBJECT_H

#include <node.h>
#include <node_object_wrap.h>

namespace demo {

class MyObject : public node::ObjectWrap {
 public:
  static void Init(v8::Isolate* isolate);
  static void NewInstance(const v8::FunctionCallbackInfo<v8::Value>& args);
  inline double value() const { return value_; }

 private:
  explicit MyObject(double value = 0);
  ~MyObject();

  static void New(const v8::FunctionCallbackInfo<v8::Value>& args);
  static v8::Global<v8::Function> constructor;
  double value_;
};

}  // namespace demo

#endif
```

The implementation of `myobject.cc` is similar to before:

```
// myobject.cc
#include <node.h>
#include "myobject.h"

namespace demo {

using node::AddEnvironmentCleanupHook;
using v8::Context;
```

```cpp
using v8::Function;
using v8::FunctionCallbackInfo;
using v8::FunctionTemplate;
using v8::Global;
using v8::Isolate;
using v8::Local;
using v8::Object;
using v8::String;
using v8::Value;

// Warning! This is not thread-safe, this addon cannot be used for worker
// threads.
Global<Function> MyObject::constructor;

MyObject::MyObject(double value) : value_(value) {
}

MyObject::~MyObject() {
}

void MyObject::Init(Isolate* isolate) {
  // Prepare constructor template
  Local<FunctionTemplate> tpl = FunctionTemplate::New(isolate, New);
  tpl->SetClassName(String::NewFromUtf8(isolate, "MyObject").ToLocalChecked());
  tpl->InstanceTemplate()->SetInternalFieldCount(1);

  Local<Context> context = isolate->GetCurrentContext();
  constructor.Reset(isolate, tpl->GetFunction(context).ToLocalChecked());

  AddEnvironmentCleanupHook(isolate, [](void*) {
    constructor.Reset();
  }, nullptr);
}

void MyObject::New(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();
  Local<Context> context = isolate->GetCurrentContext();

  if (args.IsConstructCall()) {
    // Invoked as constructor: `new MyObject(...)`
    double value = args[0]->IsUndefined() ?
        0 : args[0]->NumberValue(context).FromMaybe(0);
    MyObject* obj = new MyObject(value);
    obj->Wrap(args.This());
    args.GetReturnValue().Set(args.This());
  } else {
    // Invoked as plain function `MyObject(...)`, turn into construct call.
    const int argc = 1;
    Local<Value> argv[argc] = { args[0] };
    Local<Function> cons = Local<Function>::New(isolate, constructor);
    Local<Object> instance =
        cons->NewInstance(context, argc, argv).ToLocalChecked();
```

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

void MyObject::NewInstance(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();

  const unsigned argc = 1;
  Local<Value> argv[argc] = { args[0] };
  Local<Function> cons = Local<Function>::New(isolate, constructor);
  Local<Context> context = isolate->GetCurrentContext();
  Local<Object> instance =
      cons->NewInstance(context, argc, argv).ToLocalChecked();

  args.GetReturnValue().Set(instance);
}

}  // namespace demo
```

Test it with:

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

const obj1 = addon.createObject(10);
const obj2 = addon.createObject(20);
const result = addon.add(obj1, obj2);

console.log(result);
// Prints: 30
```