C++ style guide

See also the C++ codebase README for C++ idioms in the Node.js codebase not related to stylistic issues.

Table of contents

Guides and references

The Node.js C++ codebase strives to be consistent in its use of language features and idioms, as well as have some specific guidelines for the use of runtime features.

Coding guidelines are based on the following guides (highest priority first):

1. This document.
2. The Google C++ Style Guide.
3. The ISO C++ Core Guidelines.

In general, code should follow the C++ Core Guidelines, unless overridden by the Google C++ Style Guide or this document. At the moment these guidelines are checked manually by reviewers with the goal to validate this with automatic tools.

Formatting

Unfortunately, the C++ linter (based on Google's `cpplint`), which can be run explicitly via `make lint-cpp`, does not currently catch a lot of rules that are specific to the Node.js C++ code base. This document explains the most common of these rules:

Left-leaning (C++ style) asterisks for pointer declarations

`char* buffer;` instead of `char *buffer;`

C++ style comments

Use C++ style comments (`//`) for both single-line and multi-line comments. Comments should also start with uppercase and finish with a dot.

Examples:

```
// A single-line comment.

// Multi-line comments
// should also use C++
// style comments.
```

The codebase may contain old C style comments (`/* */`) from before this was the preferred style. Feel free to update old comments to the preferred style when working on code in the immediate vicinity or when changing/improving those comments.

2 spaces of indentation for blocks or bodies of conditionals

```
if (foo)
  bar();
```

or

```
if (foo) {
  bar();
  baz();
}
```

Braces are optional if the statement body only has one line.

`namespace`s receive no indentation on their own.

4 spaces of indentation for statement continuations

```
VeryLongTypeName very_long_result = SomeValueWithAVeryLongName +
    SomeOtherValueWithAVeryLongName;
```

Operators are before the line break in these cases.

Align function arguments vertically

```
void FunctionWithAVeryLongName(int parameter_with_a_very_long_name,
                               double other_parameter_with_a_very_long_name,
                               ...);
```

If that doesn't work, break after the `(` and use 4 spaces of indentation:

```
void FunctionWithAReallyReallyReallyLongNameSeriouslyStopIt(
    int okay_there_is_no_space_left_in_the_previous_line,
    ...);
```

Initialization lists

Long initialization lists are formatted like this:

```
HandleWrap::HandleWrap(Environment* env,
                       Local<Object> object,
                       uv_handle_t* handle,
                       AsyncWrap::ProviderType provider)
    : AsyncWrap(env, object, provider),
      state_(kInitialized),
      handle_(handle) {
```

CamelCase for methods, functions, and classes

Exceptions are simple getters/setters, which are named `property_name()` and `set_property_name()`, respectively.

```
class FooBar {
 public:
  void DoSomething();
  static void DoSomethingButItsStaticInstead();

  void set_foo_flag(int flag_value);
  int foo_flag() const;  // Use const-correctness whenever possible.
};
```

`snake_case` for local variables and parameters

```
int FunctionThatDoesSomething(const char* important_string) {
  const char* pointer_into_string = important_string;
}
```

`snake_case_` for private class fields

```
class Foo {
 private:
```

```
  int counter_ = 0;
};
```

`snake_case` for C-like structs

For plain C-like structs snake_case can be used.

```
struct foo_bar {
  int name;
};
```

Space after `template`

```
template <typename T>
class FancyContainer {
 ...
};
```

Memory management

Memory allocation

- `Malloc()`, `Calloc()`, etc. from `util.h` abort in Out-of-Memory situations
- `UncheckedMalloc()`, etc. return `nullptr` in OOM situations

Use `nullptr` instead of `NULL` or `0`

Further reading in the C++ Core Guidelines.

Use explicit pointer comparisons

Use explicit comparisons to `nullptr` when testing pointers, i.e. `if (foo == nullptr)` instead of `if (foo)` and `foo != nullptr` instead of `!foo`.

Ownership and smart pointers

- R.20: Use `std::unique_ptr` or `std::shared_ptr` to represent ownership
- R.21: Prefer `unique_ptr` over `shared_ptr` unless you need to share ownership

Use `std::unique_ptr` to make ownership transfer explicit. For example:

```
std::unique_ptr<Foo> FooFactory();
void FooConsumer(std::unique_ptr<Foo> ptr);
```

Since `std::unique_ptr` has only move semantics, passing one by value transfers ownership to the callee and invalidates the caller's instance.

Don't use `std::auto_ptr`, it is deprecated (Reference).

Avoid non-const references

Using non-const references often obscures which values are changed by an assignment. Consider using a pointer instead, which requires more explicit syntax to indicate that modifications take place.

```
class ExampleClass {
 public:
  explicit ExampleClass(OtherClass* other_ptr) : pointer_to_other_(other_ptr) {}

  void SomeMethod(const std::string& input_param,
                  std::string* in_out_param);  // Pointer instead of reference

  const std::string& get_foo() const { return foo_string_; }
  void set_foo(const std::string& new_value) { foo_string_ = new_value; }

  void ReplaceCharacterInFoo(char from, char to) {
    // A non-const reference is okay here, because the method name already tells
    // users that this modifies 'foo_string_' -- if that is not the case,
    // it can still be better to use an indexed for loop, or leave appropriate
    // comments.
    for (char& character : foo_string_) {
      if (character == from)
        character = to;
    }
  }

 private:
  std::string foo_string_;
  // Pointer instead of reference. If this object 'owns' the other object,
  // this should be a `std::unique_ptr<OtherClass>`; a
  // `std::shared_ptr<OtherClass>` can also be a better choice.
  OtherClass* pointer_to_other_;
};
```

Use AliasedBuffers to manipulate TypedArrays

When working with typed arrays that involve direct data modification from C++, use an `AliasedBuffer` when possible. The API abstraction and the usage scope of `AliasedBuffer` are documented in aliased_buffer.h.

```
// Create an AliasedBuffer.
AliasedBuffer<uint32_t, v8::Uint32Array> data;
...

// Modify the data through natural operator semantics.
data[0] = 12345;
```

Others

Type casting

- Use `static_cast<T>` if casting is required, and it is valid.
- Use `reinterpret_cast` only when it is necessary.

- Avoid C-style casts (`(type)value`).
- `dynamic_cast` does not work because Node.js is built without Run Time Type Information.

Further reading:

- ES.48: Avoid casts
- ES.49: If you must use a cast, use a named cast

Using `auto`

Being explicit about types is usually preferred over using `auto` .

Use `auto` to avoid type names that are noisy, obvious, or unimportant. When doing so, keep in mind that explicit types often help with readability and verifying the correctness of code.

```
for (const auto& item : some_map) {
  const KeyType& key = item.first;
  const ValType& value = item.second;
  // The rest of the loop can now just refer to key and value,
  // a reader can see the types in question, and we've avoided
  // the too-common case of extra copies in this iteration.
}
```

Do not include `*.h` if `*-inl.h` has already been included

Do:

```
#include "util-inl.h"  // already includes util.h
```

Instead of:

```
#include "util.h"
#include "util-inl.h"
```

Avoid throwing JavaScript errors in C++

When there is a need to throw errors from a C++ binding method, try to return the data necessary for constructing the errors to JavaScript, then construct and throw the errors using `lib/internal/errors.js` .

In general, type-checks on arguments should be done in JavaScript before the arguments are passed into C++. Then in the C++ binding, simply using `CHECK` assertions to guard against invalid arguments should be enough.

If the return value of the binding cannot be used to signal failures or return the necessary data for constructing errors in JavaScript, pass a context object to the binding and put the necessary data inside in C++. For example:

```
void Foo(const FunctionCallbackInfo<Value>& args) {
  Environment* env = Environment::GetCurrent(args);
  // Let the JavaScript handle the actual type-checking,
  // only assertions are placed in C++
  CHECK_EQ(args.Length(), 2);
  CHECK(args[0]->IsString());
  CHECK(args[1]->IsObject());
```

```
  int err = DoSomethingWith(args[0].As<String>());
  if (err) {
    // Put the data inside the error context
    Local<Object> ctx = args[1].As<Object>();
    Local<String> key = FIXED_ONE_BYTE_STRING(env->isolate(), "code");
    ctx->Set(env->context(), key, err).FromJust();
  } else {
    args.GetReturnValue().Set(something_to_return);
  }
}

// In the initialize function
env->SetMethod(target, "foo", Foo);
```

```
exports.foo = function(str) {
  // Prefer doing the type-checks in JavaScript
  if (typeof str !== 'string') {
    throw new errors.codes.ERR_INVALID_ARG_TYPE('str', 'string');
  }

  const ctx = {};
  const result = binding.foo(str, ctx);
  if (ctx.code !== undefined) {
    throw new errors.codes.ERR_ERROR_NAME(ctx.code);
  }
  return result;
};
```

Avoid throwing JavaScript errors in nested C++ methods

When you need to throw a JavaScript exception from C++ (i.e. `isolate()->ThrowException()`), do it as close to the return to JavaScript as possible, and not inside of nested C++ calls. Since this changes the JavaScript execution state, doing it closest to where it is consumed reduces the chances of side effects.

Node.js is built without C++ exception handling, so code using `throw` or even `try` and `catch` **will** break.