# Contributing to JsonCpp

## Building

Both CMake and Meson tools are capable of generating a variety of build environments for you preferred development environment.

Using cmake or meson you can generate an XCode, Visual Studio, Unix Makefile, Ninja, or other environment that fits your needs.

An example of a common Meson/Ninja environment is described next.

## Building and testing with Meson/Ninja

Thanks to David Seifert (@SoapGentoo), we (the maintainers) now use

[meson](http://mesonbuild.com/) and [ninja](https://ninja-build.org/) to build

for debugging, as well as for continuous integration (see

[`./.travis\_scripts/meson\_builder.sh`](./.travis\_scripts/meson\_builder.sh) ). Other systems may work, but minor

things like version strings might break.

First, install both meson (which requires Python3) and ninja.

If you wish to install to a directory other than /usr/local, set an environment variable called DESTDIR with the desired path:

DESTDIR=/path/to/install/dir

Then,

cd jsoncpp/

BUILD\_TYPE=debug

#BUILD\_TYPE=release

LIB\_TYPE=shared

#LIB\_TYPE=static

meson --buildtype ${BUILD\_TYPE} --default-library ${LIB\_TYPE} . build-${LIB\_TYPE}

ninja -v -C build-${LIB\_TYPE}

ninja -C build-static/ test

# Or

#cd build-${LIB\_TYPE}

#meson test --no-rebuild --print-errorlogs

sudo ninja install

## Building and testing with other build systems

See https://github.com/open-source-parsers/jsoncpp/wiki/Building

## Running the tests manually

You need to run tests manually only if you are troubleshooting an issue.

In the instructions below, replace `path/to/jsontest` with the path of the

`jsontest` executable that was compiled on your platform.

cd test

# This will run the Reader/Writer tests

python runjsontests.py path/to/jsontest

# This will run the Reader/Writer tests, using JSONChecker test suite

# (http://www.json.org/JSON\_checker/).

# Notes: not all tests pass: JsonCpp is too lenient (for example,

# it allows an integer to start with '0'). The goal is to improve

# strict mode parsing to get all tests to pass.

python runjsontests.py --with-json-checker path/to/jsontest

# This will run the unit tests (mostly Value)

python rununittests.py path/to/test\_lib\_json

# You can run the tests using valgrind:

python rununittests.py --valgrind path/to/test\_lib\_json

## Building the documentation

Run the Python script `doxybuild.py` from the top directory:

python doxybuild.py --doxygen=$(which doxygen) --open --with-dot

See `doxybuild.py --help` for options.

## Adding a reader/writer test

To add a test, you need to create two files in test/data:

\* a `TESTNAME.json` file, that contains the input document in JSON format.

\* a `TESTNAME.expected` file, that contains a flatened representation of the

input document.

The `TESTNAME.expected` file format is as follows:

\* Each line represents a JSON element of the element tree represented by the

input document.

\* Each line has two parts: the path to access the element separated from the

element value by `=`. Array and object values are always empty (i.e.

represented by either `[]` or `{}`).

\* Element path `.` represents the root element, and is used to separate object

members. `[N]` is used to specify the value of an array element at index `N`.

See the examples `test\_complex\_01.json` and `test\_complex\_01.expected` to better understand element paths.

## Understanding reader/writer test output

When a test is run, output files are generated beside the input test files. Below is a short description of the content of each file:

\* `test\_complex\_01.json`: input JSON document.

\* `test\_complex\_01.expected`: flattened JSON element tree used to check if

parsing was corrected.

\* `test\_complex\_01.actual`: flattened JSON element tree produced by `jsontest`

from reading `test\_complex\_01.json`.

\* `test\_complex\_01.rewrite`: JSON document written by `jsontest` using the

`Json::Value` parsed from `test\_complex\_01.json` and serialized using

`Json::StyledWritter`.

\* `test\_complex\_01.actual-rewrite`: flattened JSON element tree produced by

`jsontest` from reading `test\_complex\_01.rewrite`.

\* `test\_complex\_01.process-output`: `jsontest` output, typically useful for

understanding parsing errors.

## Versioning rules

Consumers of this library require a strict approach to incrementing versioning of the JsonCpp library. Currently, we follow the below set of rules:

\* Any new public symbols require a minor version bump.

\* Any alteration or removal of public symbols requires a major version bump, including changing the size of a class. This is necessary for

consumers to do dependency injection properly.

## Preparing code for submission

Generally, JsonCpp's style guide has been pretty relaxed, with the following common themes:

\* Variables and function names use lower camel case (E.g. parseValue or collectComments).

\* Class use camel case (e.g. OurReader)

\* Member variables have a trailing underscore

\* Prefer `nullptr` over `NULL`.

\* Passing by non-const reference is allowed.

\* Single statement if blocks may omit brackets.

\* Generally prefer less space over more space.

For an example:

```c++

bool Reader::decodeNumber(Token& token) {

Value decoded;

if (!decodeNumber(token, decoded))

return false;

currentValue().swapPayload(decoded);

currentValue().setOffsetStart(token.start\_ - begin\_);

currentValue().setOffsetLimit(token.end\_ - begin\_);

return true;

}

```

Before submitting your code, ensure that you meet the versioning requirements above, follow the style guide of the file you are modifying (or the above rules for new files), and run clang format. Meson exposes clang format with the following command:

```

ninja -v -C build-${LIB\_TYPE}/ clang-format

```