# Contributing

## Building Mapnik for development

Since tests, especially visual tests, are sensitive to versions of dependencies and their configuration, all tests are made against fixed versions of dependencies in prepared environment. There is [bootstrap](https://github.com/mapnik/mapnik/blob/master/bootstrap.sh) script to prepare the environment. Use it by sourcing the script, so all needed variables can be set in your shell session:

```sh

source bootstrap.sh

```

When the bootstrap script exits successfully, you can continue with usual `./configure && make` combination.

The bootstrap script generates `config.py` in the root of the repository with options for the `configure` script. Be aware that setting custom options to the configure script can override options from `config.py`. This can possibly lead to a failed build. For example, `CUSTOM\_CXXFLAGS = '-D\_GLIBCXX\_USE\_CXX11\_ABI=0'` is being set in `config.py`, thus you must not forget to include `-D\_GLIBCXX\_USE\_CXX11\_ABI=0` if you override `CUSTOM\_CXXFLAGS` option of the configure script, otherwise the build will fail. This doesn't mean custom configure options in bootstrapped environment should be avoided. It should be perfectly fine to use custom compiler, for example:

```sh

./configure CXX=g++-6.4.0 CC=gcc-6.4.0

```

Once the build is successful, run tests to ensure all is prepared to start work on Mapnik.

## Testing

Developers adding new features or fixing bugs should always write tests alongside.

Mapnik has both C++ unit tests in `./test/unit` and visual tests in `./test/visual`.

Test data for the unit and visual tests is pulled in from standalone repos via git submodules.

After building Mapnik (see INSTALL.md), the submodules can be loaded and the tests can be run like:

make test

Developers with commit access can update test data like:

cd test/data

git remote set-url origin git@github.com:mapnik/test-data

git commit -a -m "update test data"

git push origin HEAD:master

And the visual test data can up updated like:

cd test/data-visual

git remote set-url origin git@github.com:mapnik/test-data-visual

git add styles/\* images/\*

git commit -a -m "add more visual tests"

git push origin HEAD:master

After updating the test data you can then do:

# then go back to mapnik core

cd ../../

# the submodule should be marked dirty

git status

# now commit the changes to the submodule

git commit test/data test/data-visual -m "update visual tests and data"

### Visual tests

Visual tests are very helpful when developing a software with visual output such as Mapnik. Mapnik has quite sofisticated tool for running visual tests. You can run it by `./test/visual/run` which runs all visual tests by default. There are many options you can set to the visual test runner:

```console

$ ./test/visual/run -h

visual test runner:

-h [ --help ] produce usage message

-v [ --verbose ] verbose output

-o [ --overwrite ] overwrite reference image

-d [ --duration ] output rendering duration

-i [ --iterations ] arg (=1) number of iterations for benchmarking

-j [ --jobs ] arg (=1) number of parallel threads

-l [ --limit ] arg (=0) limit number of failures

--styles-dir arg (=test/data-visual/styles)

directory with styles

--images-dir arg (=test/data-visual/images)

directory with reference images

--output-dir arg (=/tmp/mapnik-visual-images)

directory for output files

-u [ --unique-subdir ] write output files to subdirectory with

unique name

--styles arg selected styles to test

--fonts arg (=fonts) font search path

--plugins arg (=plugins/input) input plugins search path

-s [ --scale-factor ] arg (=1.0, 2.0) scale factor

--agg render with AGG renderer

--cairo render with Cairo renderer

--cairo-svg render with Cairo SVG renderer

--cairo-ps render with Cairo PS renderer

--cairo-pdf render with Cairo PDF renderer

--svg render with SVG renderer

--grid render with Grid renderer

```

Most commonly needed during development is to run particular visual test or a set of visual tests. This can be done by referencing test by its name or its path.

This command will run test `tiff-reprojection-1` by looking for it in the default path with visual tests `test/data-visual/styles`.

```console

$ ./test/visual/run tiff-reprojection-1

........

Visual rendering: 0 failed / 8 passed / 0 overwritten / 0 errors

```

With `8 passed` it might seem like more visual tests were running. In fact the test was running with different renderers and scale factors. You can see more by adding verbose option:

```console

$ ./test/visual/run tiff-reprojection-1 -v

"tiff-reprojection-1-250-250-1.0" with agg... OK

"tiff-reprojection-1-250-250-1.0" with cairo... OK

"tiff-reprojection-1-250-250-1.0" with svg... OK

"tiff-reprojection-1-250-250-1.0" with grid... OK

"tiff-reprojection-1-250-250-2.0" with agg... OK

"tiff-reprojection-1-250-250-2.0" with cairo... OK

"tiff-reprojection-1-250-250-2.0" with svg... OK

"tiff-reprojection-1-250-250-2.0" with grid... OK

Visual rendering: 0 failed / 8 passed / 0 overwritten / 0 errors

```

It is possible to limit testing to particular renderer and scale factor:

```console

$ ./test/visual/run tiff-reprojection-1 -v --agg -s 1

"tiff-reprojection-1-250-250-1.0" with agg... OK

Visual rendering: 0 failed / 1 passed / 0 overwritten / 0 errors

```

When a visual test fails, HTML report is generated:

```console

$ ./test/visual/run tiff-reprojection-1 -v --agg -s 1

"tiff-reprojection-1-250-250-1.0" with agg... FAILED (20427 different pixels)

Visual rendering: 1 failed / 0 passed / 0 overwritten / 0 errors

tiff-reprojection-1 FAILED (20427 different pixels)

"/tmp/mapnik-visual-images/tiff-reprojection-1-250-250-1.0-agg.png" (actual)

"test/data-visual/images/tiff-reprojection-1-250-250-1.0-agg-reference.png" (reference)

View failure report at "/tmp/mapnik-visual-images/visual-test-results/index.html"

```

When you think the failed test is actually correct, you can overwrite the reference image by `-o` option.

If the visual test name ends with `.xml`, it is interpreted as a path to the visual test style. You can use visual test runner to render your style then.

```console

$ ./test/visual/run test/data-visual/styles/tiff-reprojection-1.xml

........

Visual rendering: 0 failed / 8 passed / 0 overwritten / 0 errors

```

#### Visual test style format

Styles used for visual tests can have additional information describing the test.

```xml

<Map>

<Parameters>

<!-- Size of output image. Multiple sizes are separated by semicolon. -->

<Parameter name="sizes">256,256</Parameter>

<!-- Map extent. -->

<Parameter name="bbox">-1, -1, 1, 1</Parameter>

<!-- Tiled rendering. Multiple tile layouts are separated by semicolon. -->

<!-- In this case the test renders one complete image and one image stitched out of 4x4 tiles. -->

<Parameter name="tiles">1,1;4,4</Parameter>

<!-- The test can be disabled. -->

<Parameter name="status">on</Parameter>

<!-- Test can be disabled for particular renderer. -->

<Parameter name="cairo">off</Parameter>

</Parameters>

...

```

## Community

Mapnik is a creative community focused on making beautiful maps with beautiful software.

We host our code on github.com/mapnik and encourage anyone interested to fork the repository and provide pull requests or patches for things they want to see added or fixed.

If you just have a question about the code, or a feature you want to discuss then feel free to create a new issue at github.com/mapnik-support.

## Plugins

Mapnik has a plugin interface for reading various geodata formats. Plugins are both viable inside of Mapnik core and also able to be used outside of Mapnik.

Plugins should be developed outside of core except in rare cases when most of the following are met:

- The plugin has no external dependencies or dependencies are easily installed

- The plugin has excellent unit tests

- The plugin has a maintainer willing to support the plugin over time

- Setup and testing of the plugin is easy on travis.ci (see .travis.yml)

Therefore plugins that depend on proprietary, unmaintained, or difficult to test third-party dependencies are not viable for Mapnik core. However they are still likely very valuable for the Mapnik community, so get in touch via https://github.com/mapnik/mapnik-support if we can help you develop your plugin outside of core.

## Code Philosophy

Look through the code to get an idea, and do not hesitate to ask questions.

Also read the design philosophy page for the motivations that lead to code decisions.

Templates are good, within reason. We seek to use templates where possible for flexible code, but not in cases where functional patterns would be just as concise and clear.

Since version 3.0 we use C++11 which brings many advantages and makes the code easier to write and to read.

In general we use Boost, it makes more possible in C++. It is a big build time dependency (as in time to compile against and # of headers) but ultimately compiles to small object code and is very fast (particularly spirit). It also has no dependencies itself (it's really an extension to the C++ language) so requiring it is much easier than requiring a hard dependency that itself has other dependencies. This is a big reason that we prefer AGG to Cairo as our primary renderer. Also AGG produces the best visual output and strikes an excellent balance between speed and thread safety by using very lightweight objects. Cairo not so much.

You will also notice that we don't use many of the standard geo libraries when we could. For instance we don't use GDAL, OGR, or GEOS anywhere in core, and only leverage them in optional plugins. We feel we can often write code that is faster and more thread safe than these libraries but that still does the job. If this ever changes we can adapt and start using these libraries or others as dependencies - nothing is nicer than standing on the shoulders of giants when it makes sense.

## Code commits best practices.

#### Big changes - awesome as pull requests

We love big, aggressive refactoring - but ideally in branches. Even if the changes should go directly into the mainline code and are stable, very big changes are useful to see as a group and branches are cheap. So, branch and commit then create a pull request against master so that other developers can take a quick look. This is a great way for informal code review when a full issue is not warrented.

#### Commits that fix issues should note the issue #

git commit plugins/input/ogr/ -m "implemented sql query in OGR plugin (closes #472)"

#### Commits that relate to issues should reference them:

git commit tests/python\_tests/sqlite\_test.py -m "more robust sqlite tests - refs #928"

#### Commits that add a new feature or fix should be added to the CHANGELOG

Ideally the CHANGELOG can be a very concise place to look for the most important recent development and should not read like a full commit log. So, some developers may prefer to weekly or monthly look back over their commits and summarize all at once with additions to the CHANGELOG. Other developers may prefer to add as they go.

## License

Mapnik is licensed LGPL, which means that you are a free to use the code in any of your applications whether they be open source or not. It also means that if you contribute code to Mapnik that others are free to continue using Mapnik in the same way, even with your new additions. If you choose to redistribute an application using Mapnik just make sure to provide any source code modifications you make back to the community. For the actual details see the full LGPL license in the COPYING doc.

## Copyright

Mapnik is an open source project and will always be. Your contributions to Mapnik should be motivated by your desire to contribute to a community effort and by the knowledge that your open code will stay that way.

Artem, as the founder and leader of the Mapnik project, is the primary copyright holder and therefore also the primary contact for any current or future license questions around Mapnik. It is important that the copyright holder is respected, trusted, and known to the community so maintaining copyright with Artem is key to maintaining the project as open source.

Therefore all files created by any core developers or patch authors should have a copyright declaration like:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* This file is part of Mapnik (c++ mapping toolkit)

\*

\* Copyright (C) {YEAR} Artem Pavlenko

\*

\* This library is free software; you can redistribute it and/or

\* modify it under the terms of the GNU Lesser General Public

\* License as published by the Free Software Foundation; either

\* version 2.1 of the License, or (at your option) any later version.

\*

\* This library is distributed in the hope that it will be useful,

\* but WITHOUT ANY WARRANTY; without even the implied warranty of

\* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU

\* Lesser General Public License for more details.

\*

\* You should have received a copy of the GNU Lesser General Public

\* License along with this library; if not, write to the Free Software

\* Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

## Coding Conventions

Mapnik is written in C++, and we try to follow general coding guidelines.

If you see bits of code around that do not follow these please don't hesitate to flag the issue or correct it yourself.

#### Prefix cmath functions with std::

The avoids ambiguity and potential bugs of using old C library math directly.

So always do `std::abs()` instead of `abs()`. Here is a script to fix your code in one fell swoop:

```sh

DIR=./bindings

for i in {abs,fabs,tan,sin,cos,floor,ceil,atan2,acos,asin}; do

find $DIR -type f -name '\*.cpp' -or -name '\*.h' -or -name '\*.hpp' | xargs perl -i -p -e "s/ $i\(/ std::$i\(/g;"

find $DIR -type f -name '\*.cpp' -or -name '\*.h' -or -name '\*.hpp' | xargs perl -i -p -e "s/\($i\(/\(std::$i\(/g;"

done

```

#### Avoid boost::lexical\_cast

It's slow both to compile and at runtime.

#### Avoid sstream objects if possible

They should never be used in performance critical code because they trigger std::locale usage

which triggers locks

#### Spaces not tabs, and avoid trailing whitespace

#### Indentation is four spaces

#### Use C++ style casts

static\_cast<int>(value); // yes

(int)value; // no

#### Use const keyword after the type

std::string const& variable\_name // preferred, for consistency

const std::string & variable\_name // no

#### Pass built-in types by value, all others by const&

void my\_function(int double val); // if int, char, double, etc pass by value

void my\_function(std::string const& val); // if std::string or user type, pass by const&

#### Use unique\_ptr instead of new/delete

#### Use std::copy instead of memcpy

#### When to use shared\_ptr and unique\_ptr

Sparingly, always prefer passing objects as const& except where using share\_ptr or unique\_ptr express more clearly your intent. See http://herbsutter.com/2013/06/05/gotw-91-solution-smart-pointer-parameters/ for more details.

#### Shared pointers should be created with [boost::make\_shared](http://www.boost.org/doc/libs/1\_47\_0/libs/smart\_ptr/make\_shared.html) where possible

Since Mapnik 3.0 use std::make\_shared.

#### Use assignment operator for zero initialized numbers

double num = 0; // please

double num(0); // no

#### Function definitions should not be separated from their arguments:

void foo(int a) // please

void foo (int a) // no

#### Separate arguments by a single space:

void foo(int a, float b) // please

void foo(int a,float b) // no

#### Space between operators:

if (a == b) // please

if(a==b) // no

#### Braces should always be used:

if (!file)

{

throw mapnik::datasource\_exception("not found"); // please

}

if (!file)

throw mapnik::datasource\_exception("not found"); // no

#### Braces should be on a separate line:

if (a == b)

{

int z = 5;

// more...

}

#### Prefer `empty()` over `size() == 0` if container supports it

This avoids implicit conversions to bool and reduces compiler warnings.

if (container.empty()) // please

if (container.size() == 0) // no

### Other C++ style resources

Many also follow the useful [Google style guide](http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml) which mostly fits our style. However, Google obviously has to maintain a lot of aging codebases. Mapnik can move faster, so we don't follow all of those style recommendations.

### Emacs helper

To auto-convert to the above syntax you can put this in an .emacs file:

;; mapnik c++

(setq c-default-style "bsd")

;; no tabs please

(setq indent-tabs-mode nil)

;; ident by four spaces

(setq c-basic-offset 4)

;; don't ident inside namespace decl

(c-set-offset 'innamespace 0)

;;

(c-set-offset 'template-args-cont 'c-lineup-template-args)

### Generating \*.gcov files with LLVM/clang toolchain to check tests coverage

\* Ensure `llvm-cov` is available

\* On OS X do `xcrun -f llvm-cov` to see the location of the binary

\* Build and link mapnik with `--coverage` and ensure optimization is off and profile flags are present. For example pass following options to Scons: `CUSTOM\_CXXFLAGS='--coverage -g -O0' LDFLAGS='--coverage'`

\* Run test(s) e.g `make test`

\* To generate \*.gcov file(s) run `llvm-cov gcov <path-to-cpp-file>`

\* \*.gcov files can be viewed in text editor