<a id="top"></a>

# Contributing to Catch2

\*\*Contents\*\*<br>

[Using Git(Hub)](#using-github)<br>

[Testing your changes](#testing-your-changes)<br>

[Writing documentation](#writing-documentation)<br>

[Writing code](#writing-code)<br>

[CoC](#coc)<br>

So you want to contribute something to Catch2? That's great! Whether it's

a bug fix, a new feature, support for additional compilers - or just

a fix to the documentation - all contributions are very welcome and very

much appreciated. Of course so are bug reports, other comments, and

questions, but generally it is a better idea to ask questions in our

[Discord](https://discord.gg/4CWS9zD), than in the issue tracker.

This page covers some guidelines and helpful tips for contributing

to the codebase itself.

## Using Git(Hub)

Ongoing development happens in the `master` branch for Catch2 v2, and in

`dev-v3` for the next major version, v3.

Commits should be small and atomic. A commit is atomic when, after it is

applied, the codebase, tests and all, still works as expected. Small

commits are also prefered, as they make later operations with git history,

whether it is bisecting, reverting, or something else, easier.

\_When submitting a pull request please do not include changes to the

single include. This means do not include them in your git commits!\_

When addressing review comments in a MR, please do not rebase/squash the

commits immediately. Doing so makes it harder to review the new changes,

slowing down the process of merging a MR. Instead, when addressing review

comments, you should append new commits to the branch and only squash

them into other commits when the MR is ready to be merged. We recommend

creating new commits with `git commit --fixup` (or `--squash`) and then

later squashing them with `git rebase --autosquash` to make things easier.

## Testing your changes

\_Note: Running Catch2's tests requires Python3\_

Catch2 has multiple layers of tests that are then run as part of our CI.

The most obvious one are the unit tests compiled into the `SelfTest`

binary. These are then used in "Approval tests", which run (almost) all

tests from `SelfTest` through a specific reporter and then compare the

generated output with a known good output ("Baseline"). By default, new

tests should be placed here.

However, not all tests can be written as plain unit tests. For example,

checking that Catch2 orders tests randomly when asked to, and that this

random ordering is subset-invariant, is better done as an integration

test using an external check script. Catch2 integration tests are written

using CTest, either as a direct command invocation + pass/fail regex,

or by delegating the check to a Python script.

There are also two more kinds of tests, examples and "ExtraTests".

Examples serve as a compilation test on the single-header distribution,

and present a small and self-contained snippets of using Catch2 for

writing tests. ExtraTests then are tests that either take a long time

to run, or require separate compilation, e.g. because of testing compile

time configuration options, and take a long time because of that.

Both of these are compiled against the single-header distribution of

Catch2, and thus might require you to regenerate it manually. This is

done by calling the `generateSingleHeader.py` script in `scripts`.

Examples and ExtraTests are not compiled by default. To compile them,

add `-DCATCH\_BUILD\_EXAMPLES=ON` and `-DCATCH\_BUILD\_EXTRA\_TESTS=ON` to

the invocation of CMake configuration step.

Bringing this all together, the steps below should configure, build,

and run all tests in the `Debug` compilation.

1. Regenerate the single header distribution

```

$ cd Catch2

$ ./scripts/generateSingleHeader.py

```

2. Configure the full test build

```

$ cmake -Bdebug-build -H. -DCMAKE\_BUILD\_TYPE=Debug -DCATCH\_BUILD\_EXAMPLES=ON -DCATCH\_BUILD\_EXTRA\_TESTS=ON

```

3. Run the actual build

```

$ cmake --build debug-build

```

4. Run the tests using CTest

```

$ cd debug-build

$ ctest -j 4 --output-on-failure -C Debug

```

## Writing documentation

If you have added new feature to Catch2, it needs documentation, so that

other people can use it as well. This section collects some technical

information that you will need for updating Catch2's documentation, and

possibly some generic advise as well.

### Technicalities

First, the technicalities:

\* If you have introduced a new document, there is a simple template you

should use. It provides you with the top anchor mentioned to link to

(more below), and also with a backlink to the top of the documentation:

```markdown

<a id="top"></a>

# Cool feature

Text that explains how to use the cool feature.

---

[Home](Readme.md#top)

```

\* Crosslinks to different pages should target the `top` anchor, like this

`[link to contributing](contributing.md#top)`.

\* We introduced version tags to the documentation, which show users in

which version a specific feature was introduced. This means that newly

written documentation should be tagged with a placeholder, that will

be replaced with the actual version upon release. There are 2 styles

of placeholders used through the documentation, you should pick one that

fits your text better (if in doubt, take a look at the existing version

tags for other features).

\* `> [Introduced](link-to-issue-or-PR) in Catch X.Y.Z` - this

placeholder is usually used after a section heading

\* `> X (Y and Z) was [introduced](link-to-issue-or-PR) in Catch X.Y.Z`

- this placeholder is used when you need to tag a subpart of something,

e.g. a list

\* For pages with more than 4 subheadings, we provide a table of contents

(ToC) at the top of the page. Because GitHub markdown does not support

automatic generation of ToC, it has to be handled semi-manually. Thus,

if you've added a new subheading to some page, you should add it to the

ToC. This can be done either manually, or by running the

`updateDocumentToC.py` script in the `scripts/` folder.

### Contents

Now, for some content tips:

\* Usage examples are good. However, having large code snippets inline

can make the documentation less readable, and so the inline snippets

should be kept reasonably short. To provide more complex compilable

examples, consider adding new .cpp file to `examples/`.

\* Don't be afraid to introduce new pages. The current documentation

tends towards long pages, but a lot of that is caused by legacy, and

we know that some of the pages are overly big and unfocused.

\* When adding information to an existing page, please try to keep your

formatting, style and changes consistent with the rest of the page.

\* Any documentation has multiple different audiences, that desire

different information from the text. The 3 basic user-types to try and

cover are:

\* A beginner to Catch2, who requires closer guidance for the usage of Catch2.

\* Advanced user of Catch2, who want to customize their usage.

\* Experts, looking for full reference of Catch2's capabilities.

## Writing code

If want to contribute code, this section contains some simple rules

and tips on things like code formatting, code constructions to avoid,

and so on.

### Formatting

To make code formatting simpler for the contributors, Catch2 provides

its own config for `clang-format`. However, because it is currently

impossible to replicate existing Catch2's formatting in clang-format,

using it to reformat a whole file would cause massive diffs. To keep

the size of your diffs reasonable, you should only use clang-format

on the newly changed code.

### Code constructs to watch out for

This section is a (sadly incomplete) listing of various constructs that

are problematic and are not always caught by our CI infrastructure.

#### Naked exceptions and exceptions-related function

If you are throwing an exception, it should be done via `CATCH\_ERROR`

or `CATCH\_RUNTIME\_ERROR` in `catch\_enforce.h`. These macros will handle

the differences between compilation with or without exceptions for you.

However, some platforms (IAR) also have problems with exceptions-related

functions, such as `std::current\_exceptions`. We do not have IAR in our

CI, but luckily there should not be too many reasons to use these.

However, if you do, they should be kept behind a

`CATCH\_CONFIG\_DISABLE\_EXCEPTIONS` macro.

#### Unqualified usage of functions from C's stdlib

If you are using a function from C's stdlib, please include the header

as `<cfoo>` and call the function qualified. The common knowledge that

there is no difference is wrong, QNX and VxWorks won't compile if you

include the header as `<cfoo>` and call the function unqualified.

## CoC

This project has a [CoC](../CODE\_OF\_CONDUCT.md). Please adhere to it

while contributing to Catch2.

-----------

\_This documentation will always be in-progress as new information comes

up, but we are trying to keep it as up to date as possible.\_

---

[Home](Readme.md#top)