# How to contribute

?? First off, thanks for taking the time to contribute! ??

Check out the [Stellar Contribution

Guide](https://github.com/stellar/.github/blob/master/CONTRIBUTING.md) for details on contributing

to stellar-core and Stellar's other repositories, especially with regard to our code of conduct and

contributor license agreement.

## Submitting Changes

Submit a pull request rebased on top of master

\* Include a descriptive commit message.

\* Changes contributed via pull request should focus on a single issue at a time.

At this point you're waiting on us. We like to at least comment on pull requests within one week

(and, typically, three business days). We may suggest some changes or improvements or alternatives.

## Stellar Core Contribution Specifics

### General

\* Try to separate logically distinct changes into separate commits and thematically distinct

commits into separate pull requests.

\* Please ensure that all tests pass before submitting changes. The local testsuite can be run as

`make check` or `src/stellar-core test`, see [README](./README.md) for details on running tests.

### Keeping our commit history clean

We're striving to keep master's history with minimal merge bubbles. To achieve this, we're asking

PRs to be submitted rebased on top of master.

To keep your local repository in a "rebased" state, simply run:

\* `git config branch.autosetuprebase always` \_changes the default for all future branches\_

\* `git config branch.master.rebase true` \_changes the setting for branch master\_

Note: you may still have to run manual "rebase" commands on your branches, to rebase on top of

master as you pull changes from upstream.

### Testing

Please ensure that all tests pass before submitting changes. The local testsuite can be run as

`make check` or `src/stellar-core --test`, see [README](./README.md) for details on running tests.

### Code Style

Code formatting wise, we have a `.clang-format` config file that you should use on modified files.

- Names of structs and classes are camel case with the first letter capital, eg. `class MyClassName`

- Names of functions and local variables are camel case with the first letter lowercase, eg. `void

myFunctionName()` or `int myLocalInt`

- Names of member variables are camel case with the first letter "m", eg. `int mMyMemberInt`

- Names of compile-time constants are macro case, eg. `int const MY\_CONST\_INT`

- Braces appear on their own line

- Braces are not omitted for a control statement with only a single statement in its scope

- const appears to the right of the type, eg. `int const` or `int const&`

### Performance Enhancements

When submitting changes that may impact performance, you need to also provide some evidence of the

improvement (which also implies no regression). See the [performance

evaluation](../performance-eval/performance-eval.md) document for more details.

### Special configure flags for improved diagnostics

When building with `configure`, a few options are available to enable better diagnostics when

running tests.

It is strongly recommended to enable C++ runtime checks and the address sanitizer during

development by passing the following flags to configure (in addition to other flags that you may

need):

./configure --enable-extrachecks --enable-asan

### enable-extrachecks

This enables additional debug checks such as passed the end iterators.

More information can be found:

\* [libstdc++ debug mode](https://gcc.gnu.org/onlinedocs/libstdc++/manual/debug\_mode.html)

\* [libc++ debug mode](https://libcxx.llvm.org/docs/DesignDocs/DebugMode.html#using-debug-mode)

## Sanitizers

Sanitizers are mutually exclusive.

### enable-asan

Build with asan (address-sanitizer) instrumentation, which detects invalid address utilization.

See https://clang.llvm.org/docs/AddressSanitizer.html for more information.

### enable-undefinedcheck

build with undefinedcheck (undefined-behavior-sanitizer) instrumentation.

See https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html for more information.

### enable-memcheck

build with memcheck (memory-sanitizer) instrumentation.

See https://clang.llvm.org/docs/MemorySanitizer.html for more information.

`memcheck` only works with clang and `libc++`.

For memcheck to work, you will need to compile your own version of `libc++` (see below), and pass

the path to your libraries to the configure script using the `LIBCXX\_PATH` variable, something

like:

./configure --disable-postgres --enable-memcheck LIBCXX\_PATH=/home/user/src/llvm/libcxx\_msan/lib

If you do not have an instrumented version of `libpq` (postgres client library), you may get false

positives as well (disabling postgres is a good workaround).

What the configure script does under the cover is use the custom version of your library at link

time, but still uses the system headers - so make sure that the two don't conflict!

#### Building a custom `libc++`

The steps for building an instrumented version of libc++ can be found on the [memory sanitizer

how-to](https://github.com/google/sanitizers/wiki/MemorySanitizerLibcxxHowTo#instrumented-libc).

When done, note the path to `libc++.so` and `libc++abi.so` (that will be passed with `LIBCXX\_PATH`

to the configure script).

# Running tests

## Running tests basics

run tests with:

`src/stellar-core test`

run one test with:

`src/stellar-core test testName`

run one test category with:

`src/stellar-core test '[categoryName]'`

Categories (or tags) can be combined: AND-ed (by juxtaposition) or OR-ed (by comma-listing).

Tests tagged as [.] or [hide] are not part of the default test.

Tests tagged as [acceptance] are not part of `make check` test runs.

supported test options can be seen with

`src/stellar-core test --help`

display tests timing information:

`src/stellar-core test -d yes '[categoryName]'`

xml test output (includes nested section information):

`src/stellar-core test -r xml '[categoryName]'`

## Running tests against postgreSQL

There are two options. The easiest is to have the test suite just

create a temporary postgreSQL database cluster in /tmp and delete it

after the test. That will happen by default if you run `make check`.

You can also use an existing database cluster so long as it has

databases named `test0`, `test1`, ..., `test9`, and `test`. To set

this up, make sure your `PGHOST` and `PGUSER` environment variables

are appropriately set, then run the following from bash:

for i in $(seq 0 9) ''; do

psql -c "create database test$i;"

done

You will need to set the `TEMP\_POSTGRES` environment variable to 0

in order to use an existing database cluster.

## Running tests in parallel

The `make check` command also supports parallelization. This functionality is

enabled with the following environment variables:

\* `TEST\_SPEC`: Used to run just a subset of the tests (default: "~[.]")

\* `NUM\_PARTITIONS`: Partitions the test suite (after applying `TEST\_SPEC`) into

`$NUM\_PARTITIONS` disjoint sets (default: 1)

\* `RUN\_PARTITIONS`: Run only a subset of the partitions, indexed from 0

(default: "$(seq 0 $((NUM\_PARTITIONS-1)))")

\* `TEMP\_POSTGRES`: Automatically generates temporary database clusters instead

of using an existing cluster (default: 1)

For example,

`env TEST\_SPEC="[history]" NUM\_PARTITIONS=4 RUN\_PARTITIONS="0 1 3" make check`

will partition the history tests into 4 parts then run parts 0, 1, and 3.

## Running stress tests

There are a few special stress tests included in the test suite. Those are \*subsystem level\* tests,

not to be confused with more advanced tests that would be done as part of [performance

evaluation](../performance-eval/performance-eval.md).

We adopt the convention of tagging a stress-test for subsystem foo as [foo-stress][stress][hide].

Then, running:

\* `stellar-core test [stress]` will run all the stress tests,

\* `stellar-core test [foo-stress]` will run the stress tests for subsystem foo alone, and

\* neither `stellar-core test` nor `stellar-core test [foo]` will run stress tests.