# Cpp Unit Testing

The CppUnit framework is used for unittesting C/C++ source code for all sipXecs related projects. The CppUnit Cookbook is the general guide to creating unittests but specific guidelines and information about extensions made to the CppUnit framework are described in this document.

For any given source file named `src/foo/MyClass.cpp`, place unittests in a file named `src/test/foo/MyClassTest.cpp`.

Use the source file TestTemplate.cpp as a template for creating all new unittests. (hint: Copy `TestTemplate.cpp` to a new file called `MyClassTest.cpp` then search and replace `TestTemplate` with `MyClassTest`.)

Add your unittest to the testsuite that is run as part of the build by listing it in the `src/test/Makefile.am` under the `testsuite_SOURCES` make variable. (TIP: To temporarily isolate just your unittest, remove all other source file listings from make variable except the files in the `sipxunit` directory. Alternatively you can create a separate target program with just your unittest and source files from the `sipxunit` directory, just be sure NOT to check it in that way.)

Logging is encouraged in unittests. All log levels are set to DEBUG during execution of the unittest, and the output of each test can be found in $BUILD /<project>/src/test with the appropriate test name.

If any one unittest fails, it will stop the build from succeeding. In scenarios where you want to check in a failed unittest to communicate a bug beyond your control to fix, you can use the macros `KNOWN_BUG` and `KNOWN_FATAL_BUG` to signify such issues. This will not break the build, but will report errors to stdout. See TestUtilities.h header file for more information.

libtool is used to build the testsuite, consequently you must use libtool to invoke a debugger on the testsuite. Example:

```
libtool --mode=execute gdb .../testsuite
```

where the last argument is the path to the `testsuite` file in the directory containing the `testsuite-....o` files generated from the tests. Since the tests are built into shared libraries, you might not be able to set breakpoints in them at the moment `gdb` starts, since the shared libraries may not have yet been loaded. If so, use the usual trick of `break main` then `run` to force the executable preamble code to link in all the shared libraries.

You can also execute a subset of the tests by using the "sandbox" feature. To use it, edit `Makefile.am` in the directory containing the `.o` files. The `sandbox_SOURCE` variable lists the tests that will be included in the subset. Rebuild the directory (including the `autoconf` and `configure` steps), then run them under the debugger by doing:

```
libtool --mode=execute gdb .../sandbox
```

It is recommended each unittest be run through tools like Valgrind, Electric Fence and gcov to test for memory management and threading issues and to determine unittest code coverage. These tools will be used by automated, stress-testing machines so it's best if you use these tools ahead of time.

## Valgrind

Valgrind can detect many memory management and threading bugs. See Valgrind HOWTO for more information. To run unittests under valgrind:

```
libtool --mode=execute valgrind --leak-check=yes -v --show-reachable=yes src/test/testsuite
```

Deciphering output printed to console will take some practice. You may find section "4.2.4 How to Suppress Errors" of Valgrind HOWTO helpful. Please email suggestions to dev list for ways to integrate expected leaks and errors into project.

## Electric Fence

Running the unittests under Electric Fence is still experimental, but it has been known to find some issues. Install
Electric Fence according to documentation and simply link to memory management library like so:

```
./configure LDFLAGS=-lefence
```

Best documentation is man page `man efence`. If you see Electric Fence's initialization header printed to console when running unittests then Electric Fence is compiled in and working. Errors will surface as segmentation faults closer to points of bad memory management. Run unittests thru `gdb` or similar debugger to further diagnose issues.

# gcov

It's important to analyze the code coverage of your unittests on the targeted source code. To compile a unittest with code coverage analysis built in, configure the project like so:

```
./configure CXXFLAGS="-fprofile-arcs -ftest-coverage"
make clean all check
```

whenever the unittests are run, coverage will be captured. To inspect coverage:

```
cd src
gcov {target}-MyClassName
cat MyClass.cpp.gcov
```

Where `{target}` is your library or binary name. (hint: run `ls` in `src` directory.) Files ending in `*.gcov` will be annotated listing of the source code with the number of times each line has been executed or ##### for lines that have not been executed.