Unit tests

The sources in this directory are unit test cases. Boost includes a unit testing framework, and since Bitcoin Core already uses Boost, it makes sense to simply use this framework rather than require developers to configure some other framework (we want as few impediments to creating unit tests as possible).

The build system is set up to compile an executable called `test_bitcoin` that runs all of the unit tests. The main source file for the test library is found in `util/setup_common.cpp`.

Compiling/running unit tests

Unit tests will be automatically compiled if dependencies were met in `./configure` and tests weren't explicitly disabled.

After configuring, they can be run with `make check`.

To run the unit tests manually, launch `src/test/test_bitcoin`. To recompile after a test file was modified, run `make` and then run the test again. If you modify a non-test file, use `make -C src/test` to recompile only what's needed to run the unit tests.

To add more unit tests, add `BOOST_AUTO_TEST_CASE` functions to the existing .cpp files in the `test/` directory or add new .cpp files that implement new `BOOST_AUTO_TEST_SUITE` sections.

To run the GUI unit tests manually, launch `src/qt/test/test_bitcoin-qt`

To add more GUI unit tests, add them to the `src/qt/test/` directory and the `src/qt/test/test_main.cpp` file.

Running individual tests

`test_bitcoin` accepts the command line arguments from the boost framework. For example, to run just the `getarg_tests` suite of tests:

```
test_bitcoin --log_level=all --run_test=getarg_tests
```

`log_level` controls the verbosity of the test framework, which logs when a test case is entered, for example. `test_bitcoin` also accepts the command line arguments accepted by `bitcoind`. Use `--` to separate both types of arguments:

```
test_bitcoin --log_level=all --run_test=getarg_tests -- -printtoconsole=1
```

The `-printtoconsole=1` after the two dashes redirects the debug log, which would normally go to a file in the test datadir (`BasicTestingSetup::m_path_root`), to the standard terminal output.

... or to run just the doubledash test:

```
test_bitcoin --run_test=getarg_tests/doubledash
```

Run `test_bitcoin --help` for the full list.

Adding test cases

To add a new unit test file to our test suite you need to add the file to `src/Makefile.test.include`. The pattern is to create one test file for each class or source file for which you want to create unit tests. The file naming convention is `<source_filename>_tests.cpp` and such files should wrap their tests in a test suite called `<source_filename>_tests`. For an example of this pattern, see `uint256_tests.cpp`.

Logging and debugging in unit tests

`make check` will write to a log file `foo_tests.cpp.log` and display this file on failure. For running individual tests verbosely, refer to the section above.

To write to logs from unit tests you need to use specific message methods provided by Boost. The simplest is `BOOST_TEST_MESSAGE`.

For debugging you can launch the `test_bitcoin` executable with `gdb` or `lldb` and start debugging, just like you would with any other program:

```
gdb src/test/test_bitcoin
```

Segmentation faults

If you hit a segmentation fault during a test run, you can diagnose where the fault is happening by running `gdb ./src/test/test_bitcoin` and then using the `bt` command within gdb.

Another tool that can be used to resolve segmentation faults is valgrind.

If for whatever reason you want to produce a core dump file for this fault, you can do that as well. By default, the boost test runner will intercept system errors and not produce a core file. To bypass this, add `--catch_system_errors=no` to the `test_bitcoin` arguments and ensure that your ulimits are set properly (e.g. `ulimit -c unlimited`).

Running the tests and hitting a segmentation fault should now produce a file called `core` (on Linux platforms, the file name will likely depend on the contents of `/proc/sys/kernel/core_pattern`).

You can then explore the core dump using

```
gdb src/test/test_bitcoin core

(gbd) bt  # produce a backtrace for where a segfault occurred
```