Ceedling

[Fine Print]

All code is copyright © 2010 Ceedling Project
by Mike Karlesky, Mark VanderVoord, and Greg Williams

This Documentation Is Released Under a
Creative Commons 3.0 Attribution Share-Alike License

What the What?

Assembling build environments for C projects – especially with automated unit tests – is a pain. Whether it's Make or Rake or Premake or what-have-you, set up with an all-purpose build environment tool is tedious, and maintenance can be frustrating. Ceedling allows you to assemble an entire test and build environment for a C project from a single YAML configuration file. Ceedling is written in Ruby and works with the Rake build tool (plus other goodness like testing frameworks).

For a build project including unit tests and using the default toolchain gcc, the configuration file could be as simple as this:

```
:project:
  :build_root: build/
  :release_build: TRUE

:paths:
  :test:
    - tests/**
  :source:
    - source/**
```

From the command line, to build the release version of your project, you would simply run rake release. To run all your unit tests, you would run `rake test:all`. That's it!

Of course, many more advanced options allow you to configure your project with a variety of features to meet a variety of needs. Ceedling can work with practically any command line toolchain and directory structure – all by way of the configuration file. Further, because Ceedling piggy backs on Rake, you can add your own Rake tasks to accomplish project tasks outside of testing and release build. A facility for plugins also allows you to extend Ceedling's capabilities for needs such as custom code metrics reporting and coverage testing.

What's with this Name?

Glad you asked. Ceedling is tailored for C projects with unit tests and is built upon/around Rake (Rake is a Make replacement implemented in the Ruby scripting language). So, we've got C, Rake, and the fertile soil of a capable build environment in which to grow and tend your project and its unit tests. Ta da – *Ceedling*.

What Do You Mean "tailored for C projects with unit tests"?

Well, we like to write unit tests for our C code to make it lean and mean (that whole Test-Driven Development thing). Along the way, this style of writing C code spawned two tools to make the job easier: a unit test framework for C called *Unity* and a mocking library called *CMock*. And, though it's not directly related to testing, a C framework for exception handling called *CException* also came along.

These tools and frameworks are great, but they require quite a bit of environment support to pull them all together in a usable fashion. We started off with Rakefiles to assemble everything. These ended up being quite complicated and had to be edited or created anew for each new project. Ceedling replaces all that tedium and rework with a simple configuration file that ties everything together.

Back up. Ruby? Rake? YAML? Unity? CMock? CException?

Seem overwhelming? It's not bad at all, and for the benefits tests bring us, it's all worth it.

Ruby is a handy scripting language like Perl or Python. It's a modern, full featured language that happens to be quite handy for accomplishing tasks like code generation or automating one's workflow while developing in a compiled language such as C.

Rake is a utility written in Ruby for accomplishing dependency tracking and task automation common to building software. It's a modern, more flexible replacement for Make. Rakefiles are Ruby files, but they contain build targets and tasks similar to Makefiles.

YAML is a “human friendly data serialization standard for all programming languages.” It's kinda like a markup language. With a YAML library, you can serialize data structures to and from the file system in a textual, human readable form.

Unity is a unit test framework for C. It provides facilities for test assertions; executing tests; and collecting and reporting test results. It consists of a single C source file and two C header files.

CMock is a tool written in Ruby able to generate entire mock functions in C from a given C header file. Mock functions are invaluable in interaction-based unit testing.

CException is a C source and header file that provide a simple exception mechanism for C by way of wrapping up the setjmp / longjmp standard library calls. Exceptions are a much cleaner and preferable alternative to managing and passing error codes up your return call trace.

Notes:

- YAML support is included with Ruby.
- Unity, CMock, and CException are bundled with Ceedling.

Installation & Setup: What Exactly Do I Need to Get Started?

Installation and setup requires a handful of steps. From scratch:

1. Download and install Ruby
2. Use Ruby's command line gem package manager to install Rake: `gem install rake`
3. Grab the Ceedling package and place it in your file system
   (it already contains Unity, CMock, and CException)
4. Create an empty build directory for your project
   (Ceedling will fill out the directory structure below the build root upon first use)
5. Create a simple Rakefile (`rakefile.rb`) that contains only a load call to Ceedling on your file system:
   `load '<path>/ceedling/lib/rakefile.rb'`
6. Create your project YAML file (more on this later in this document) –
   `project.yml` is the default file name Ceedling recognizes

Notes:

- Steps 1-3 are a one time affair. Once steps 1-3 are completed once, only steps 4-6 are needed for each new project.
- See the sample starter project for a working setup. When steps 1-3 are complete and assuming you have gcc in your path (Ceedling's default toolchain), you will only need to edit the path within the sample Rakefile (see step 5 above) to yield a working, albeit simple, project.
- Certain advanced features of Ceedling rely on gcc and cpp as preprocessing tools. In most *nix systems, these tools are already available. For Windows environments, we recommend the mingw project (Minimalist GNU for Windows). This represents an optional, additional setup / installation step to complement the list above.

Now What? How Do I Make It GO?

We're getting a little ahead of ourselves here, but it's good context on how to drive this bus. Everything is done via the command line. We'll cover conventions and how to configure your project in later sections.

To run tests, build your release artifact, etc., you will be interacting with Rake on the command line. Ceedling works with Rake to present you with build & test tasks and to coordinate the file generation needed to accomplish something useful. You can also add your own independent Rake tasks or create plugins to extend Ceedling (more on this later).

Rake Command	**What It Does**
`rake -T`	List all available rake tasks with descriptions (rake tasks without descriptions are not listed)
`rake test:all`	Run all unit tests (rebuilding anything that's changed along the way)
`rake test:delta`	Run only those unit tests for which the source or test files have changed
`rake test:foo.c`	Run the test for the specified source file (will fail if no test file accompanies the named source)
`rake test:test_foo.c`	Run the specified test file
`rake release`	Build all source into a release artifact (if the release build option is configured)
`rake logging <tasks...>`	Enable logging to `<build path>/logs` – only meaningful in conjunction with release or test tasks (must come before said tasks to log their steps and output)
`rake verbosity[x] <tasks...>`	Change the default verbosity level. [x] ranges from 0 (quiet) to 4 (obnoxious). Level [3] is the default. Only meaningful in conjunction with release or test tasks (must come before tasks to log their steps and output)
`rake clean`	Deletes all toolchain binary artifacts (object files, executables), test results, and any temporary files
`rake clobber`	Extends clean task's behavior to also remove generated files: test runners, mocks, preprocessor output
`rake <tasks...> --trace`	For advanced users attempting to troubleshoot a confusing error, debug Ceedling or a plugin

Tasks for individual test files are not listed in `-T` output; so many tests may be available that it's unwieldy to list them all.

Multiple rake tasks can be executed at the command line (order is executed as provided). For example, `rake clobber test:all release` will removed all generated files; build and run all tests; and then build all source in that order. If any Rake task fails, execution halts before the next task.

Important Conventions

Directory Structure, Filenames & Extensions

Much of Ceedling's functionality is driven by collecting files matching certain patterns inside the directories it's configured to search.

At present test files and source files must be segregated by directories. Tests can be held in subdirectories within source directories, or tests and source directories can be separated at the top of the directory tree.

Source Files & Binary Release Artifacts

Your binary release artifact results from the compilation and linking of all source files Ceedling finds in the specified source directories. At present only source files with a single (configurable) extension are recognized. That is, *.c and *.cc files will not both be recognized – only one or the other.

Test Files & Executable Test Fixtures

Ceedling builds each individual test file into a corresponding monolithic test fixture executable. Test files are recognized by a naming convention: a (configurable) prefix such as "`test_`" in the file name with the same file extension as used by the source files.

Ceedling knows what files to compile and link into each individual test executable by way of the #include list contained in each test file. Any *.c files in the configured search directories that correspond to the *.h files included in a test file will be compiled and linked into the resulting test fixture executable. From this same #include list, Ceedling knows which files to mock and compile and link into the test executable (if you use mocks in your tests). Further, by naming your test functions according to convention, Ceedling will extract and collect into a runner all your test case functions. In this generated runner lives the `main()` entry point for the resulting test executable.

A sample test file with explanation follows on the next page.

```
// test_foo.c -----------------------------------------------

#include “unity.h”     // compile/link in Unity test framework

#include “types.h”     // header file with no *.c file -- no compilation/linking

#include “foo.h”       // source file foo.c under test

#include “mock_bar.h”  // bar.h will be found and mocked as mock_bar.c + compiled/linked in;

                       // foo.c includes bar.h and uses functions declared in it

#include “mock_baz.h”  // baz.h will be found and mocked as mock_baz.c + compiled/linked in

                       // foo.c includes baz.h and uses functions declared in it



void setUp(void) {}    // every test file requires this function;

                       // setUp() is called by the generated runner before each test case function


void tearDown(void) {} // every test file requires this function;

                       // tearDown() is called by the generated runner before each test case function


// a test case function

void test_Foo_Function1_should_Call_Bar_AndGrill(void)

{

    Bar_AndGrill_Expect();                     // setup function from mock_bar.c that instructs our

                                               // framework to expect Bar_AndGrill() to be called once

    TEST_ASSERT_EQUAL(0xFF, Foo_Function1()); // assertion provided by Unity

                                               // Foo_Function1() calls Bar_AndGrill() & returns a byte

}


// another test case function

void test_Foo_Function2_should_Call_Baz_Tec(void)

{

    Baz_Tec_ExpectAnd_Return(1);       // setup function provided by mock_baz.c that instructs our

                                       // framework to expect Baz_Tec() to be called once and return 1

    TEST_ASSERT_TRUE(Foo_Function2()); // assertion provided by Unity

}


// end of test_foo.c ----------------------------------------
```

From the above test file specified above Ceedling will generate `test_foo_runner.c`; this runner file will contain `main()` and call both of the example test case functions.

The final test executable will be `test_foo.exe` (for Windows machines or test_foo.out for *nix systems). Based on the #include list above, the test executable will be the output of the linker having processed `unity.o`, `foo.o`, `mock_bar.o`, `mock_baz.o`, `test_foo.o`, and `test_foo_runner.o`. Ceedling finds the files, generates mocks, generates a runner, compiles all the files, and links everything into the test executable. Ceedling will then run the test executable and collect test results from it to be reported to the developer at the command line.

For more on the assertions and mocks shown, consult the documentation for Unity and CMock.

The Almighty Project Configuration File (in Glorious YAML)

Please consult YAML documentation for the finer points of format and to understand details of our YAML-based configuration file. We recommend Wikipedia's entry on YAML for this. A few highlights from that reference page:

- YAML streams are encoded using the set of printable Unicode characters, either in UTF-8 or UTF-16
- Whitespace indentation is used to denote structure; however tab characters are never allowed as indentation
- Comments begin with the number sign (#), can start anywhere on a line, and continue until the end of the line
- List members are denoted by a leading hyphen (-) with one member per line, or enclosed in square brackets ([ ]) and separated by comma space (,   )
- Hashes are represented using the colon space (:   ) in the form key: value, either one per line or enclosed in curly braces ({   }) and separated by comma space (,   )
- Strings (scalars) are ordinarily unquoted, but may be enclosed in double-quotes ("), or single-quotes (')
- YAML requires that colons and commas used as list separators be followed by a space so that scalar values containing embedded punctuation can generally be represented without needing to be enclosed in quotes
- Repeated nodes are initially denoted by an ampersand (&) and thereafter referenced with an asterisk (*)

Notes on what follows:

- Each of the following sections represent top-level entries in the YAML configuration file.
- Unless explicitly specified in the configuration file, default values are used by Ceedling.
- These three settings, at minimum, must be specified:
  - [:project][:build_root]
  - [:paths][:source]
  - [:paths][:test]
- As much as is possible, Ceedling validates your settings in properly formed YAML.
- Improperly formed YAML will cause a Ruby error when the YAML is parsed. This is usually accompanied by a complaint with line and column number pointing into the project file.
- Certain advanced features rely on gcc and cpp as preprocessing tools. In most *nix systems, these tools are already available. For Windows environments, we recommend the mingw project (Minimalist GNU for Windows).
- Ceedling is primarily meant as a build tool to support automated unit testing. All the heavy lifting is involved there. Creating a release build artifact is quite trivial in comparison. Consequently, most default options and the construction of Ceedling itself is skewed towards supporting testing.

project:

Global settings live here.

Setting	**Description**	**Default**
build_root	The top level directory into which all generated directory structure and files are placed. *Note: this is one of a handful of values that must be set.*	<none>
logging	If enabled, a log of all executed steps and tool output will be written to `<build path>/logs`. This functionality can also be controlled at the command line.	FALSE
use_exceptions	Configures the build environment to make use of CException. Note that if you do not use exceptions, there's no harm in leaving this as its default value.	TRUE
use_mocks	Configures the build environment to make use of CMock. Note that if you do not use mocks, there's no harm in leaving this setting as its default value.	TRUE
use_test_preprocessor	This option allows Ceedling to work with test files that contain conditional compilation statements (e.g. #ifdef) and header files you wish to mock that contain conditional preprocessor statements and/or macros. Ceedling and CMock are advanced tools with sophisticated parsers. However, they do not include entire C language preprocessors. Consequently, with this option enabled, Ceedling will use gcc's preprocessing mode and the cpp preprocessor tool to strip down / expand test files and headers to their applicable content which can then be processed by Ceedling and CMock. With this option enabled, the gcc & cpp tools must exist in an accessible system search path.	FALSE
use_auxiliary_dependencies	The base rules and tasks that Ceedling creates using Rake capture most of the dependencies within a standard project (e.g. when the source file accompanying a test file changes, the corresponding test fixture executable will be rebuilt when tests are re-run). However, not all relationships can be captured this way. If a typedef or macro changes in a header file three levels of #include statements deep, this option allows the appropriate build actions to occur. This is accomplished by using the dependencies discovery mode of gcc. With this option enabled, gcc must exist in an accessible system search path.	FALSE
test_file_prefix	Ceedling collects test files by convention from within the test file search paths. The convention includes a unique name prefix and a file extension matching that of source files. Why not simply recognize all files in test directories as test files? By using the given convention, we have greater flexibility in what we do with C files in the test directories.	"test_"
verbosity	Verbosity ranges from 0 (quiet) to 4 (obnoxious). This functionality can also be controlled at the command line.	3
options_path	Just as you may have various build configurations for your source codebase, you may have build variations for your test codebase. By specifying an options path, Ceedling will search for other project YAML files, make command line tasks available (`rake options:variation` for a `variation.yml` file), and merge the project configuration of these option files in with the main project file at runtime. Note these Rake tasks, like verbosity or logging control, at the command line must come before the test or release task they are meant to modify.	<none>
release_build	When enabled, a `release` Rake task is exposed. This configuration option requires a corresponding release compiler and linker to be defined (gcc is used as the default). More release configuration options are available in the `release_build` section.	FALSE

release_build:

Options controlling the build of a binary release artifact kick back in their crib here.

Setting	**Description**	**Default**
output		<none>
use_assembly		FALSE

paths:

Options controlling search paths have their own place here.

Setting	**Description**	**Default**
test		[] (empty)
source		[] (empty)
support		[] (empty)
include		[] (empty)
test_toolchain_include		[] (empty)
release_toolchain_include		[] (empty)