Test Builds: Using A Toolchain and Build Manager for Fun and Profit

Audience: This is a high-level overview of compilers, linkers, and build management tools for those unfamiliar with the core details of such things. Executing and automating your C unit tests relies heavily on these tools in ways a bit different than in non-testing workflows. So it may be a good idea to read this even if you have a decent working knowledge of the functions of compilers, linkers, and build managers.

Some Context / A Disclaimer: This document is a good intro to the concepts it delves into… but it will not make you an expert on such things. Despite its length, it only summarizes much of the material it strives to communicate. But, it should be more than enough to acquaint you with the ideas needed to really rock out this course. If you read things here that seem very foreign to the main course presentation, that's because this document touches on several topics not yet covered in the course itself. As the units of the course presentation develop and expand, we will present these ideas below in richer detail (especially when we discuss Interaction Testing and porting to new targets). As a result, we will very likely refactor this document as we go. Confused on anything? That's what the forums on ThrowTheSwitch.org are for!

I Write Code and then Compile It. Boom. Done. Right?

Write some C. Click the button. Presto. You have a binary blob of machine code ready to run. Modern Integrated Development Environments (IDE's) have simplified software development workflows to a literal pushbutton interface. It's easy to forget that there's a lot going on behind the scenes.

The techniques in this class for unit testing C projects take advantage of the plumbing involved in transforming source code into a binary blob of machine code. So it's worthwhile understanding the tools involved in building C code into executable machine code as these tools are integral to automating your testing.

Compiling and Linking: Or, How Your Baby C Code Grows Up Into a Big Strong Executable

Greatly simplified, the process of transforming C source code into executable machine code involves two basic steps: compiling and linking. In compilation, a compiler takes as input a source code file and processes its named elements and instructions into an object file of symbols and relocatable machine code. A linker then smooshes multiple object files and any

libraries together to yield an executable binary. The linking step resolves the symbolic references among the object files and libraries and also translates all object file machine code and memory references into a single executable memory arrangement. Optionally throw in some extra meta-information during the build process, and from all this a debugger can get its hooks in the executable code to allow you to trace through execution.

Let's Talk About Your IDE and Just How Much It Loves You

When you add source files to your project in an IDE you're doing a lot more than just organizing your files in a graphical list for editing access. You're actually giving important information to all the tools the IDE coordinates to build your code into an executable or to provide debugging access.

Pulling files into your IDE project tells the compiler which source files to compile, header files to parse for names, and all of the paths to those files. Your project file also tells the linker which object files (and libraries) resulting from compilation to then link into an executable. Throw in various settings including debug options for the build in your project configuration, and the IDE is actually doing some big favors for you in managing options and directory paths for all the tools it calls on your behalf. For even a handful of source files, this process can be tedious to do yourself with direct access to the compiler and linker and debugger. Long before the advent of IDE's, the process of building code into a binary blob of executable machine code was quickly recognized to be a real pain in the butt. And so command line tools like `make` were developed specifically to address this pain. We'll talk more about `make` and its relatives in a later section.

Linking Is the Magic That Allows Unit Testing in C to Happen

The best way to unit test is to segregate your test code from your source code. These two should live separately but in parallel. This separation seems to magically lead to good design in your source code and greatly simplifies maintaining your tests. Further, this sort of separation motivates conventions that allow executing unit tests in an automated fashion. The linker is at the heart of much of this.

Testing As Remixing Modules and Interfaces

Once you realize that each C source code file becomes an object file upon compilation and that it can be remixed via linking with any other code aware of its interface (e.g. header file), then you can do some crazy useful things in testing. A single source module separated from its source file kin can be linked with a test module aware of its interface and a test framework module. Boom. There's an executable unit test. Now you can run assertions against the inputs and return values of a source function without inserting any test code in your source code.

Remember that unlike in a release build where all your source code becomes a single final executable, in the style of unit testing we're teaching, each source file together with supporting test code becomes its own individual test executable. If you have fifty source files, you get one release executable and at least fifty test executables.

A Tiny Introduction to Mocking

Now let's get even wackier. What if we consider a source module that relies on several other source modules. That seems like a nasty web of dependencies. Wouldn't it be nice if we could extract a single source module and fake out the interactions with the other source modules with which it interacts? Then we could isolate each source module and ensure it interacts with other source modules the way we intend. Boom. That's called mocking, and linking lets us do it as well.

Mocking is too big a topic to explain here in detail (we have a significant part of an entire course coming to address it!). However, in short, here's how it works. Each source module publishes an interface in the form of a header file. As far as a linker is concerned, it could care less which object files it links together so long as the symbols and references all match up. If `ModuleBar` makes a call to a function `foo()` outside of itself, the linker just needs to see that there's a function `foo()` somewhere in the object files it's been instructed to link. It has no way of knowing whether the code attached to the signature of `foo()` is "real" or "fake" code. Using the header files of modules we wish to fake, we can automatically generate code by convention from the functions listed. That generated code tracks whether a function has been called, captures the parameters its been passed, and prepares a return value to be sent back to the real source code under test. With some calls to the generated mock from inside our test code, we can set up expectations and verify if a source module interacted with the interfaces it relies upon as we intended. This too is unit testing.

Don't Worry If You Feel a Little Lost on All This

We promise it will all make sense in due time as the course progresses. The key ideas are simply that testing relies heavily on convention and that the linker is our friend in cleverly rearranging our compiled source and test and mock code object files into test executables.

How About a Tool? Because All This Compiling and Linking Sounds Like a Serious Pain.

You ain't kiddin', brother or sister (as the case may be). Keeping track of all these source and test and mock files and their resulting compiled object files and which object files to link together is a big hassle. This is where build management tools come in.

Keeping All the Rules

For all the help an IDE provides, it kinda cheats. It tricks you into telling it exactly what files to include in the build. Since the process of tracking those files and paths is directly connected to your editing needs, you don't mind setting this up at all (or even notice). But there is another way. Build management tools have been around for a long time. The non-IDE command line variants rely on a system of pattern specifiers, directory path searching, and dependency rules to control the building process. The patterns, paths, and dependency rules are all contained in a configuration file fed to the build management tool.

Dependency Rules and Patterns: That Thinking Is Plain Backwards

To understand these pattern and dependency rule based build tools, you just need to think backwards. The build configuration file will contain dependency rules that associate several different targets with a means to create each of those targets. In this hypothetical build configuration file, a binary executable will be dependent on all its object files, and each object file will be dependent on its corresponding source file. Then there will be a dependency rule that invokes the linker to link all the object files into the named executable target. Another dependency rule will invoke the compiler to produce an object file from a source file.

Seek and Ye Shall Find

When you request the build management tool to create the named executable target, you are instructing the tool to resolve all the dependency rules necessary for that target. First it will look for the object files the target is dependent upon. For each object files that doesn't exist, the build tool will then call the compiler to generate the needed object file from the source file it is dependent upon. Once all the object files exist, then the build manager will tell the linker to go ahead and link the final executable.

Dependency rules are not only associations between an input file and a transformed output file. Dependency rules also include the notion of precedence by way of file timestamps. If an object

file already exists and is newer than the source file it is dependent upon, the build tool knows it can skip compiling it again. If the final executable is newer than all its constituent object file dependencies, the build tool will skip linking (and compiling) anything at all. If there are source files newer than their object file dependencies, then the build tool knows it must compile them and then relink the executable as well (which, of course, will be older than the source files up the chain of dependency rules).

Patterns Are the Glue for All the Rules

Patterns specified in the configuration file tell the build tool how to find files and can also dynamically generate build targets. So, for instance, you can specify a pattern for the build tool to search certain directories for source files and then transform their .c source file extensions into .o object file target names. Presto. Any .c file you add to your project directories is automatically transformed into an .o object file target. Since you've already specified a means for how to call the compiler with a source file and spit out an object file for an object file target, you're all set.

So What About Building Unit Tests Into Test Executables?

We can use the same patterns and dependency rules to build our test executables. We add rules and patterns to our build tool configuration file to automatically build test executables. The configuration causes the tool to search test directories for test files. Those test file names are converted into test executable target names. The dependencies of those test executables are configured to cause compilation and linking of the corresponding test file, source file, and test framework. With a little extra script or outside tool help, we can also discover which mocks must be generated and build those into the test executable as well. And so that's how we do. If you look inside the build files we provide for the class examples you will see this very approach.

Build Tools You Want to Know

From everything in the preceding, it's probably fairly clear that your IDE's approach to building source into an executable just is not capable of handling the style of testing we're presenting. But command line pattern-and-rule build tools are entirely capable at doing this.

Most developers run their test builds with the sorts of command line tools that follow. They will continue to rely on their IDE to verify source code through compilation and especially to load binaries onto target embedded hardware and perform debugging. However, once you have some sort of automation in place, you may choose to replicate your IDE release build configuration in a command line tool setup as well. In this way your build system can

automatically push a release binary into a system testing rig or automatically push a tagged release binary to a repository.

Below we discuss just three build tools plus the general topic of Continuous Integration. There are certainly many more build tools than just these three. We use two of these tools in this course (we first use Make early on and Ceedling later on). Make is the longstanding standard that is available for essentially any platform. Ceedling is a tool specific to test builds using the techniques and tools we discuss in this class. Since Ceedling is based on Rake, we talk about Rake to better provide context for Ceedling.

Make

Make is pretty much the granddaddy of build tools. It was one of the first tools created back in the 1970s to formalize a means to execute a build configuration. Several variations of make have appeared over the years for different platforms and/or to extend or simplify the core features of the original. Microsoft even includes a version of make called nmake within its Visual Studio suite. Make is available for essentially any platform you may be using.

Sample Makefile Content

```
#Tool Definitions
CC=gcc
CFLAGS=-I. -I$(PATHU) -DTEST

#Path Definitions
PATHU = ../unity/
PATHS =
PATHT =
PATHB = build/

#Files We Are To Work With
SRC = testUnit1.c unity.c
OBJ = $(patsubst %.c,$(PATHB)%.o,$(SRC))
DEP = $(PATHU)unity.h $(PATHU)unity_internals.h
TGT = $(PATHB)test$(TARGET_EXTENSION)

test: $(PATHB) $(TGT)
    ./$(TGT)

$(PATHB)%.o:: $(PATHS)%.c $(DEP)
    $(CC) -c $(CFLAGS) $< -o $@
```

```
$(PATHB)%.o:: $(PATHU)%.c $(DEP)
    $(CC) -c $(CFLAGS) $< -o $@

$(TGT): $(OBJ)
    gcc -o $@ $^
```

Make Resources

- GNU Make
- Make by Example
- Writing and Debugging a Makefile
- Make Manual

Rake

Rake works very much like Make. It uses the same concepts and approach and can even run Makefiles. What distinguishes Rake from Make is that Rake was implemented in the Ruby scripting environment. Consequently, you can easily mix a powerful scripting programming language with Rake rules. As such, Rake has you covered if you have special regular expression needs or must generate some oddball file for a tool in your build environment or need to bundle up post-build report generation. Make can do these sorts of things as well, but as Make is a monolithic tool, it requires calling out to scripts and a hodgepodge of external tools. In many cases, a Rakefile may be a better approach. Rake is available on any platform where Ruby runs. Ruby runs on most any platform.

Sample Rakefile Content

```
require 'rake'
require 'rake/clean'

CC         = "clang"
PKGS       = "alsa libspotify ncurses"
CFLAGS     = "-std=gnu99 -ggdb -Wall"
LDFLAGS    = `pkg-config --libs #{PKGS}`.strip << " -lpthread"

TARGET      = "spoticli"
SOURCE_DIR  = "src"
OBJECT_DIR  = "build"
```

```
SOURCE_FILES = FileList.new("#{SOURCE_DIR}/**/*.c")

directory OBJECT_DIR

task :default => "build:target"

namespace :build do
      task :objects do
      SOURCE_FILES.each do |source|
            # replace source dir with object dir
            object = source.gsub(/^#{SOURCE_DIR}/, "#{OBJECT_DIR}")
            object = object.sub(/\.c$/, '.o')

            # create directory
            mkdir_p object.pathmap("%d").strip

            # compile source
            sh "#{CC} #{CFLAGS} -I./#{SOURCE_DIR} -c -o #{object}
#{source}"
      end
      end
      CLEAN.include('**/*.o', 'build')

      task :target => :objects do
      # find all object files in build
      object_files = FileList["#{OBJECT_DIR}/**/*.o"].join(' ')

      # link
      sh "#{CC} #{object_files} #{LDFLAGS} -o #{TARGET}"
      end
      CLOBBER.include("#{TARGET}")
end
```

Rake Resources

- Rake Ruby gem
- Rake Tutorial
- Using the Rake Build Language by Martin Fowler
- Rake documentation

Ceedling

Ceedling is a highly specialized build tool specific to working with C source and test files. It understands and comes prepackaged with Unity and CMock, and its default configuration assumes gcc as the preprocessor, compiler, and linker tool chain. Ceedling is built on top of Rake (see previous section) and relies on the conventions of testing we teach in this class. Ceedling is not nearly as polished and widely used as the other tools, but it's quite handy for its highly specific purpose. It works on any platform Ruby runs.

Sample Ceedling YAML Configuration File Content

```
# Using default gcc setup baked into Ceedling
:project:
  :build_root: project/build/
  :release_build: TRUE

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

Ceedling Resources

- Ceedling Ruby Gem
- Ceedling Introduction
- Ceedling Documentation
- Ceedling Forum

Continuous Integration

Continuous Integration (CI) is a tool-enabled practice that is, in turn, enabled by source code management tools (see document *Introduction to Revision Control and Source Code Management Plus A Super Tiny Git Reference Guide*) and test-driven development techniques and tools. The idea is simple. If a team is building a product by submitting small commits of test and source code to a shared code repository, each commit (or a timer) triggers a build and execution of the project's codebase and test suite. Ideally this build takes place on a dedicated machine (or at least a machine not used for development) and occurs frequently (even multiple times an hour in some cases). This approach identifies any broken source or test code soon

after it is committed; ensures the release artifact has access to all needed libraries and is built with proper compile flags and environment variables; and prevents “magic” environment setups on developer machines that diverge from other developers or the final production environment. Numerous free and commercial CI tools are available.