4

Using GNU Autotools to Manage a "Minimal Project"

This chapter describes how to manage a minimal project using the GNU Autotools. For this discussion, a minimal project refers to the smallest possible project that can still illustrate a sufficient number of principles related to using the tools. Studying a smaller project makes it easier to understand the more complex interactions between these tools when larger projects require advanced features.

The example project used throughout this chapter is a fictitious command interpreter called `'foonly'`. `'foonly'` is written in C, but like many interpreters, uses a lexical analyzer and a parser expressed using the `lex` and `yacc` tools. The package is developed to adhere to the GNU `'Makefile'` standard, which is the default behavior for Automake.

This project does not use many features of the GNU Autotools. The most noteworthy one is libraries; this package does not produce any libraries of its own, so Libtool does not feature them in this chapter. The more complex projects presented in Chapter 7, "A Small GNU Autotools Project" and Chapter 11, "A Large GNU Autotools Project" illustrate how Libtool participates in the build system. The essence of this chapter is to provide a high-level overview of the user-written files and how they interact.

4.1 User-Provided Input Files

The smallest project requires the user to provide only two files. The GNU Autotools generate the remainder of the files needed to build the package are in Section 4.2, "Generated Output Files." The files are as follows:

`'Makefile.am'`	An input to `automake`
`'configure.in'`	An input to `autoconf`

I like to think of 'Makefile.am' as a high-level, bare-bones specification of the project's build requirements: what needs to be built, and where does it go when it is installed? This is probably Automake's greatest strength—the description is about as simple as it could possibly be, yet the final product is a 'Makefile' with an array of convenient make targets.

The 'configure.in' is a template of macro invocations and shell code fragments used by autoconf to produce a 'configure' script (see Appendix C, "Generated File Dependencies"). autoconf copies the contents of 'configure.in' to 'configure', expanding macros as they occur in the input. Other text is copied verbatim.

Let's take a look at the contents of the user-provided input files relevant to this minimal project. Here is the 'Makefile.am':

```
bin_PROGRAMS = foonly
foonly_SOURCES = main.c foo.c foo.h nly.c scanner.l parser.y
foonly_LDADD = @LEXLIB@
```

This 'Makefile.am' specifies that we want a program called 'foonly' to be built and installed in the 'bin' directory when make install is run. The source files used to build 'foonly' are the C source files 'main.c', 'foo.c', and 'foo.h'; the lex program 'scanner.l'; and a yacc grammar in 'parser.y'. This points out a particularly nice aspect about Automake: Because lex and yacc both generate intermediate C programs from their input files, Automake knows how to build such intermediate files and link them into the final executable. Finally, we must remember to link a suitable lex library, if 'configure' concludes that one is needed.

And here is the 'configure.in':

```
dnl Process this file with autoconf to produce a configure script.
AC_INIT(main.c)
AM_INIT_AUTOMAKE(foonly, 1.0)
AC_PROG_CC
AM_PROG_LEX
AC_PROG_YACC
AC_OUTPUT(Makefile)
```

This 'configure.in' invokes some mandatory Autoconf and Automake initialization macros, and then calls on some Autoconf macros from the AC_PROG family to find suitable C compiler, lex, and yacc programs. Finally, the AC_OUTPUT macro is used to cause the generated 'configure' script to output a 'Makefile'—but from what? It is processed from 'Makefile.in', which Automake produces for you based on your 'Makefile.am' (see Appendix C).

4.2 Generated Output Files

By studying the diagram in Appendix C, you can see which commands must be run to generate the required output files from the input files shown in the preceding section.

First, we generate `'configure'`:

```
$ aclocal
$ autoconf
```

Because `'configure.in'` contains macro invocations that are not known to Autoconf itself—`AM_INIT_AUTOMAKE` being a case in point—it is necessary to collect all the macro definitions for Autoconf to use when generating `'configure'`. This is done using the `aclocal` program, so called because it generates `'aclocal.m4'` (see Appendix C). If you were to examine the contents of `'aclocal.m4'`, you would find the definition of the `AM_INIT_AUTOMAKE` macro contained within.

After running `autoconf`, you will find a `'configure'` script in the current directory. It is important to run `aclocal` first, because `automake` relies on the contents of `'configure.in'` and `'aclocal.m4'`. Now on to `automake`:

```
$ automake --add-missing
automake: configure.in: installing ./install-sh
automake: configure.in: installing ./mkinstalldirs
automake: configure.in: installing ./missing
automake: Makefile.am: installing ./INSTALL
automake: Makefile.am: required file ./NEWS not found
automake: Makefile.am: required file ./README not found
automake: Makefile.am: installing ./COPYING
automake: Makefile.am: required file ./AUTHORS not found
automake: Makefile.am: required file ./ChangeLog not found
```

The `'--add-missing'` option copies some boilerplate files from your Automake installation into the current directory. Files such as `'COPYING'`, which contain the GNU General Public License, change infrequently, and so can be generated without user intervention. A number of utility scripts are also installed—these are used by the generated `'Makefile'`s, particularly by the `install` target. Notice that some required files are still missing. These are as follows:

`'NEWS'`	The `'NEWS'` file is a record of user-visible changes to a package. The format is not strict, but the changes to the most recent version should appear at the top of the file.
`'README'`	A `'README'` file is the first place a user will look to get an overview for the purpose of a package, and perhaps special installation instructions.
`'AUTHORS'`	The `'AUTHORS'` file lists the names, and usually mail addresses, of individuals who worked on the package.

continues ▶

▶ *continued*

`'ChangeLog'`	The `'ChangeLog'` is an important file—it records the changes made to a package. The format of this file is quite strict (see Section 4.5, "Documentation and ChangeLogs").

For now, we will do enough to placate Automake:

```
$ touch NEWS README AUTHORS ChangeLog
$ automake --add-missing
```

Automake has now produced a `'Makefile.in'`. At this point, you may want to take a snapshot of this directory before we really let loose with automatically generated files.

By now, the contents of the directory are looking fairly complete and reminiscent of the top-level directory of a GNU package you may have installed in the past:

```
AUTHORS    INSTALL      NEWS        install-sh    mkinstalldirs
COPYING    Makefile.am  README      configure     missing
ChangeLog  Makefile.in  aclocal.m4  configure.in
```

It should now be possible to package up your tree in a `tar` file and give it to other users for them to install on their own systems. One of the `make` targets that Automake generates in `'Makefile.in'` makes it easy to generate distributions (see Chapter 12, "Rolling Distribution Tarballs"). A user would merely have to unpack the `tar` file, run `configure` (see Chapter 2, "How to Run Configure, and The Most Useful Standard Makefile Targets"), and finally type `make all`:

```
$ ./configure
creating cache ./config.cache
checking for a BSD compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking whether make sets $@{MAKE@}... yes
checking for working aclocal... found
checking for working autoconf... found
checking for working automake... found
checking for working autoheader... found
checking for working makeinfo... found
checking for gcc... gcc
checking whether the C compiler (gcc  ) works... yes
checking whether the C compiler (gcc  ) is a cross-compiler... no
checking whether we are using GNU C... yes
checking whether gcc accepts -g... yes
checking how to run the C preprocessor... gcc -E
checking for flex... flex
checking for flex... (cached) flex
checking for yywrap in -lfl... yes
```

```
checking lex output file root... lex.yy
checking whether yytext is a pointer... yes
checking for bison... bison -y
updating cache ./config.cache
creating ./config.status
creating Makefile

$ make all
gcc -DPACKAGE=\"foonly\" -DVERSION=\"1.0\" -DYYTEXT_POINTER=1  -I. -I. \
  -g -O2 -c main.c
gcc -DPACKAGE=\"foonly\" -DVERSION=\"1.0\" -DYYTEXT_POINTER=1  -I. -I. \
  -g -O2 -c foo.c
flex   scanner.l && mv lex.yy.c scanner.c
gcc -DPACKAGE=\"foonly\" -DVERSION=\"1.0\" -DYYTEXT_POINTER=1  -I. -I. \
  -g -O2 -c scanner.c
bison -y   parser.y && mv y.tab.c parser.c
if test -f y.tab.h; then \
  if cmp -s y.tab.h parser.h; then rm -f y.tab.h; \
  else mv y.tab.h parser.h; fi; \
else :; fi
gcc -DPACKAGE=\"foonly\" -DVERSION=\"1.0\" -DYYTEXT_POINTER=1  -I. -I. \
  -g -O2 -c parser.c
gcc  -g -O2  -o foonly  main.o foo.o scanner.o parser.o -lfl
```

4.3 Maintaining Input Files

If you edit any of the GNU Autotools input files in your package, you must regenerate the machine-generated files for these changes to take effect. If you add a new source file to the `foonly_SOURCES` variable in `'Makefile.am'`, for instance, you must regenerate the derived file `'Makefile.in'`. If you are building your package, you need to rerun `configure` to regenerate the site-specific `'Makefile'`, and then rerun `make` to compile the new source file and link it into `'foonly'`.

It is possible to regenerate these files by running the required tools, one at a time. As you can see from the preceding discussion, it can be difficult to compute the dependencies—does a particular change require `aclocal` to be run? Does a particular change require `autoconf` to be run? There are two solutions to this problem.

The first solution is to use the `autoreconf` command. This tool regenerates all derived files by rerunning all the necessary tools in the correct order. It is somewhat of a brute-force solution, but it works very well, particularly if you are not trying to accommodate other maintainers, or regular maintenance that would render this command bothersome.

The alternative is Automake's `'maintainer mode'`. By invoking the `AM_MAINTAINER_MODE` macro from `'configure.in'`, Automake activates an `'--enable-maintainer-mode'` option in `'configure'`. Chapter 8, "Bootstrapping," explains this at length.

4.4 Packaging Generated Files

What to do with generated files is keenly contested on the relevant Internet mailing lists. There are two points of view, and this section presents both of them so that you can try to determine the best policy for your project.

One argument is that generated files should not be included with a package, but rather only the "preferred form" of the source code should be included. By this definition, `'configure'` is a derived file, just like an object file, and it should not be included in the package. Therefore, the user should use the GNU Autotools to bootstrap themselves prior to building the package. I believe there is some merit to this purist approach, because it discourages the practice of packaging derived files.

The other argument is that the advantages of providing these files can far outweigh the violation of good software-engineering practice already mentioned. By including the generated files, users have the convenience of not needing to be concerned with keeping up-to-date with all the different versions of the tools in active use. This is especially true for Autoconf, because `'configure'` scripts are often generated by maintainers using locally modified versions of `autoconf` and locally installed macros. If the user were to regenerate `'configure'`, the result could be different to that intended. Of course, this is poor practice, but it does happen in real life.

I think the answer is to include generated files in the package when the package is going to be distributed to a wide user community (that is, the general public). For in-house packages, the former argument might make more sense, because the tools may also be held under version control.

4.5 Documentation and ChangeLogs

As with any software project, it is important to maintain documentation as the project evolves. The documentation must reflect the current state of the software, but it must also accurately record the changes that have been made in the past. The GNU coding standard rigorously enforces the maintenance of documentation. Automake, in fact, implements some of the standard by checking for the presence of a `'ChangeLog'` file when `automake` is run!

A number of files exist, with standardized filenames, for storing documentation in GNU packages. The complete GNU coding standard, which offers some useful insights, can be found at `http://www.gnu.org/prep/standards.html`.

Other projects, including in-house projects, can use these same tried-and-true techniques. The purpose of most of the standard documentation files was outlined earlier (see Section 4.2, "Generated Output Files"), but the `'ChangeLog'` deserves additional treatment.

When recording changes in a `'ChangeLog'`, one entry is made per person, per day. Logical changes are grouped together, whereas logically distinct changes (that is, "change sets") are separated by a single blank line. Here is an example from Automake's own `'ChangeLog'`:

```
1999-11-21  Tom Tromey  <tromey@cygnus.com>

        * automake.in (finish_languages): Only generate suffix rule
        when not doing dependency tracking.

        * m4/init.m4 (AM_INIT_AUTOMAKE): Use AM_MISSING_INSTALL_SH.
        * m4/missing.m4 (AM_MISSING_INSTALL_SH): New macro.

        * depend2.am: Use @SOURCE@, @OBJ@, @LTOBJ@, @OBJOBJ@,
        and @BASE@.  Always use -o.
```

Another important point to make about `'ChangeLog'` entries is that they should be brief. It is not necessary for an entry to explain in detail *why* a change was made but rather *what* the change was. If a change is not straightforward, the explanation of *why* belongs in the source code itself. The GNU coding standard offers the complete set of guidelines for keeping `'ChangeLog's`. Although any text editor can be used to create ChangeLog entries, Emacs provides a major mode to help you write them.