Tip 61: Get to know Makefile.am

Ben Klemens

1 February 2012

level: Autotools user
purpose: learn the conventions and let Automake do the rest

Last time (Entry #110), you met Autotools, and saw how friendly it can be for a simple project. Except your project is more than one `.c` file, which is why you need Autotools to begin with. The hard part is writing `Makefile.am`, which gives a somewhat encrypted summary of your project. Once you learn the language, though, it's not so bad.

First, if you include a target and its associated actions in `Makefile.am`, then Automake will copy it into the final makefile verbatim.

If you add a variable, that too gets added verbatim. This will especially be useful in conjunction with Autoconf, because if `Makefile.am` has variable assignments like

```
TEMP=@autotemp@
HUMIDITY=@autohum@
```

and your `configure.ac` has

```
#configure is a plain shell script; these are plain shell vars
autotemp=40
autohum=.8

AC_SUBST(autotemp)
AC_SUBST(autohum)
```

then the final makefile will have text reading

```
TEMP=40
HUMIDITY=.8
```

So you have an easy conduit from the shell script that Autoconf spits out to the final makefile.

The rest of `Makefile.am` will largely consist of two types of entry, neither of which look anything like the final makefile. They are *product list variables* and *product source/option variables*, but in an effort to avoid like-sounding jargon I will refer to them as form variables and content variables, respectively.

Form variables The example of this from last time was this line:

```
bin_PROGRAMS=hello
```

If the install location is `il` and the type of compilation `TYPE`, these all have the form `il_TYPE`. The most important examples:

```
bin_PROGRAMS        #programs
include_HEADERS     #headers to install in system-wide includedir.
pkginclude_HEADERS #same, but install in includedir/yourprogram subdir.
lib_LTLIBRARIES     #dynamic libraries, via libtool
EXTRA_DIST          #distribute with pkg, but don't install
```

There are many others; `python_PYTHON`, for example. The location/TYPE combo provides a bit of false generality, because it makes no sense to install programs in the include directory, for example, even if the system would let you do it. However, the location half can usefully be `noinst`, meaning that something gets produced but not installed, and you can put `pkg` in front of several locations to produce `pkgbin`, `pkglib`, et cetera.

`nodist_EXTRAS`: files that have to be in the package for the thing to compile, but which won't be installed in the system. I could never work out whether this is the right place for 'em, but this is where I put Apophenia's test data, needed for the tests but not worth installing.

The TYPE half tells the system what form of make target to generate. It has built-in rules for generating a program from source and built-in rules for generating a library via Libtool, and you are telling it which template to use.

Put as many items on each line as you'd like, e.g.:

```
pkginclude_HEADERS = firstpart.h secondpart.h
EXTRA_DIST = sample1.csv sample2.csv \
             sample99.csv sample100.csv
```

Content variables Items under `EXTRA_DIST` just get copied over, and the process for dealing with header files is basically just to copy them to the right place. So those are basically settled.

For the compilation steps like `..._PROGRAMS` and `..._LDLIBRARIES`, Automake needs to know more details about how the compilation works. At the very least, it needs to know what source files are being compiled. Thus, for every item on the right-hand side of an equals sign of a form variable about compilation, we need a variable specifying the sources:

```
bin_PROGRAMS= weather wxpredict
weather_SOURCES= temp.c barometer.c
wxpredict_SOURCES=rng.c tarotdeck.c
```

This may be all you need for a basic package.

Notice that the content variables have the same `lower_UPPER` look as the form variables above, but they are formed from entirely different parts and serve entirely different purposes.

Back to traditional makefiles for a second: if you don't specify a rule for compiling (but not linking) from source to object, `make` will apply a POSIX-standard implicit rule that it has memorized:

```
$(CC) $(CPPFLAGS) $(CFLAGS) -c
```

To link together object files, the implicit rule is:

```
$(CC) $(LDFLAGS) obj1.o obj2.o $(LOADLIBES) $(LDLIBS)
```

Let's just look the other way from the variable `LOADLIBES` [sic], and Automake prefers `LDADD` for the second half of the link line anyway (i.e., always use `LDLIBS` with `make`; always use `LDADD` with Automake).

That little caveat noted, you can set all of these variables on a per-program or per-library basis, like `weather_CFLAGS=-O1`. Or, use `AM_` to set a variable for all compilations or linkings. I consider this line to be essential, giving debugger symbols and all warnings for every compilation/link:

```
AM_CFLAGS=-g -Wall -O3
```

If you've been following me for very long, then you know that I always use `-std=gnu99` to get GCC to use a less obsolete standard. However, this is a very compiler-specific flag. If I put

```
AC_PROG_CC_C99
```

in `configure.ac`, then Autoconf will set the `CC` variable to `gcc -std=gnu99` for me. Autoscan isn't (yet) smart enough to put this into the `configure.scan` that it generates for you, so you will probably have to put it into `configure.ac` yourself.

Specific rules override `AM_`-based rules, so here's how we'd keep the general rules and add on an override for one flag:

```
AM_CFLAGS=-g -Wall -O3
hello_CFLAGS = $(AM_CFLAGS) -O0
```

To give a fuller example, say that several programs all depend on common source files. Then perhaps you could generate a no-install static library (without Libtool) and link everything to that library. Notice how `hello.a` turns into `hello_a` for the purposes of the content variable naming scheme, as all of the characters that aren't alphanumeric get converted to underscores.

```
noinst_LIBRARIES = hello.a
hello_a_SOURCES = guts1.c guts2.c
AM_CFLAGS=-g -Wall -O3
```

```
bin_PROGRAMS = hello hi

hello_SOURCES= hello.c
hello_LDADD=hello.a

hi_SOURCES= hi.c
hi_LDADD=hello.a
```

OK, those are all the parts of `Makefile.am`: `make` variables as usual and `make` target/rules as usual are copied verbatim (after Autoconf does variable substitutions); form variables specify which files are to be handled how and where to put them; and content variables specify the details of how compilation happens for each output file.