Tip 62: Get to know configure.ac

Ben Klemens

3 February 2012

level: Autotools user
purpose: Write a bang-up setup script

You are going to write a script named `configure.ac`, and it is going to produce two outputs: a makefile, and a header file named `config.h`.

At this point, I expect that you've already opened one the sample `configure.ac` files produced so far, and noted that it looks nothing at all like a shell script. This is because it makes heavy use of a set of macros (in the m4 macro language) that are predefined by Autoconf. Rest assured that every one of them will blow up into familiar-looking (but probably illegible) lines of shell script. That is, `configure.ac` isn't a script to generate the `configure` shell script, it *is* `configure`, just compressed by some very impressive macros.

The m4 language doesn't have all that much syntax that we as users have to care about. Every macro is function-like, with parens after the macro name listing the comma-separated arguments (if any; else the parens are dropped). Where most languages write `'literal text'`, m4-via-autoconf writes `[literal text]`, and to prevent surprises where m4 macro-expands your input, all of your macro inputs should be wrapped in those square brackets.

The first line that Autoscan generated is a good example:

```
AC_INIT([FULL-PACKAGE-NAME], [VERSION], [BUG-REPORT-ADDRESS])
```

We know that this is going to generate a few hundred lines of shell code, and somewhere in there, the given elements will be set. Change the values in square brackets to whatever is relevant.

Also, you can often omit elements, so something like

```
AC_INIT([hello], [1.0])
```

is valid. At the extreme, one might give zero arguments to a macro like `AC_OUTPUT`, in which case you don't need to bother with the parentheses. By the way, the current custom in m4 documentation is to mark optional arguments with—I am not making this up—square brackets. So bear in mind that in m4 macros for Autoconf, square brackets mean literal not-for-expansion text, and in m4 macro documentation it means an optional argument.

What do we need for a functional Autoconf file? In order of appearance:

- `AC_INIT(...)`, as above.
- `AM_INIT_AUTOMAKE`, to have Automake generate the makefile.
- `LT_INIT` sets up Libtool, so you need this if and only if you are installing a shared library.
- `AC_CONFIG_FILES([Makefile subdir/Makefile])`, which tells Autoconf to go through those files listed and replace variables like `@cc@` with their appropriate value. If you have several makefiles (like subdirectories), then list them here.
- `AC_OUTPUT` to ship out.

So we have a functional build package for any POSIX system in four or five lines, three of which Autoscan probably wrote for you.

But the real art that takes `configure.ac` from functional to intelligent is in identifying what needs to be checked or set up—predicting problems some users might have—and finding the macro that handles it.

Getting back to the outputs, `config.h` is a standard C header consisting of a series of `#define` statements. For example, if Autoconf verified the presence of the GSL, you would find

```
#define HAVE_LIBGSL 1
```

in `config.h`. You can then put `#ifdef`s into your code to behave appropriately under appropriate circumstances.

Autoconf's check doesn't just find the library based on some naming scheme and hope that it actually works—it compiles a program using that library and any one function somewhere in the library. So autoscan can't autogenerate a check for the library, because it doesn't know what functions are to be found in it. The macro to check for a library is an easy one-liner, to which you provide the library name and a function that can be used for the check. E.g.:

```
AC_CHECK_LIB([glib-2.0],[g_free])
AC_CHECK_LIB([gsl],[gsl_blas_dgemm])
```

Add one line to `configure.ac` for every library you use that is not 100% guaranteed by the POSIX standard, and those one-liners will blossom into the appropriate shell script snippets in `configure`.

Further, you may recall my complaint about how package managers always split libraries into the binary shared object package and the header package. Users of your library may not remember (or even know) to install the header package, so check for it with, e.g.:

```
AC_CHECK_HEADER([gsl/gsl_matrix.h], , [AC_MSG_ERROR(
    [Couldn't find the GSL header files (I searched for <gsl/gsl_matrix.h> \
    on the include path). If you are using a package manager, don't forget \
    to install the libgsl-devel package, as well as libgsl itself.])])
```

Notice the two commas: the arguments are (header to check, action if found, action if not found), and we are leaving the second blank.

What else could go wrong in a compilation? After the basic makefile from last time, what else needs to be prepped? It's hard to become an authority on all the glitches of all the world's computers, given that we all have only a couple of machines at our disposal. The best reference I have seen—a veritable litany of close readings of the POSIX standard, implementation failures, and practical advice—is the Autoconf manual[1] itself. Some of it will just be nitpicking to the rest of us,[2], some of it is good advice for your code-writing, and some of the the descriptions of system quirks are followed by the name of an Autoconf macro to include in your project's `configure.ac` should it be relevant to your situation.

Then, check the GNU Autoconf macro archive[3] for additional macros that don't ship with Autoconf itself but which you can save to an `m4` subdirectory in your project directory, where Autoconf will be able to find and use them.

More bits of shell Because `configure.ac` is a compressed version of the `configure` script the user will run, you can throw in any arbitrary shell code you'd like. Before you do, double-check that what you want to do isn't yet handled by any macros—is your situation really so unique that it never happened to any Autotools users before?

A banner notifying the user that they've made it through the configure process might be nice, and there's no macro for it because all you need is `echo`.

You'll notice that I use several variables defined by Autoconf. There's documentation about what shell variables the system defines for you to use, but I find that the easiest way to find out what a macro did is to just run `autoreconf` to fully expand `configure` and look at the post-expansion details.

Anyway, here's a sample banner:

```
echo \
"-------------------------------------------------------------

 ${PACKAGE_NAME} version ${PACKAGE_VERSION}

 Installation directory prefix: '${prefix}'.
 Compilation command: '${CC} ${CFLAGS} ${CPPFLAGS}'

 Now type 'make; sudo make install' to generate the program
 and install it to your system.

-------------------------------------------------------------"
```

[1] `http://www.gnu.org/software/autoconf/manual/autoconf.html#Existing-Tests`

[2] E.g., "Solaris 10 `dtksh` and the UnixWare 7.1.1 Posix shell [...] mishandle braced variable expansion that crosses a 1024- or 4096-byte buffer boundary within a here-document."

[3] `http://www.gnu.org/software/autoconf-archive/`