Tip 60: Package your code with Autotools

Ben Klemens

30 January 2012

level: your code is good enough to share
purpose: love your users

The Autotools are what make it possible for you to download a library or program, and run

```
./configure
make
sudo make install
```

(and nothing else) to set it up. Please recognize what a miracle of modern science this is: the developer has no idea what the name of your compiler is, what sort of computer you have, where you keep your programs and libraries (`/usr/bin`? `/sw`? `/cygdrive/c/bin`?), and whatever quirks your machine demonstrates, and yet `configure` sorted everything out so that `make` could run seamlessly. And so, Autotools is the center of how anything gets distributd in the modern day. If you want anybody who is not your personal friend to use your code (or if you want a Linux distro to include your program in their package manager), then you need to have Autotools generate the build for you.

You will quickly realize how complicated the Autotools can get, but the basics are darn simple. By the end of this, we will have written six lines of packaging text and run four commands, and will have a complete (but rudimentary) package ready for distribution.

Here's how I imagine it all happening. [The actual history is different from the sequence here. These are distinct packages, and there is a reason to run any of them without the other. But all that is irrelevant, and the purpose of this little dialogue is to help you think of the several tools as a unified whole working toward a unified goal.]

P1: I love `make`. It's so nice that I can write down all the little steps to building my project in one place.

P2: Yes, automation is great. Everything should be automated, all the time.

P1: Yeah, I started adding lots of steps to my makefile, so users can type `make` to just produce the program, `make install` to install, `make check` to run tests, and so on. It's a lot of work to write all those makefile targets, but so smooth when it's all assembled.

P2: OK, I shall write a system—it will be called Automake—that will automatically generate makefiles from a very short pre-Makefile.

P1: That's great. Producing shared libraries is especially annoying, because every system has a different procedure.

P2: It is annoying. Given the system information, I shall write a program for generating the scripts needed to produce shared libraries from source code, and then put those into Automade makefiles.

P1: Wow, so all I have to do is tell you my operating system, and whether my compiler is named `cc` or `clang` or `gcc` or whatever, and you'll drop in the right code for the system I'm on?

P2: That's too much work. I will write a system called Autoconf that will be aware of every system out there and that will produce a report of everything Automake and your program needs to know about the system. Then Automake can use the list of environment variables in my report to produce a makefile.

P1: I am flabbergasted—you've automated the process of autogenerating Makefiles. But it sounds like we've just changed the work I have to do from inspecting the various platforms to writing configuration files for Autoconf and makefile templates for Automake.

P2: You're right. I shall write a tool, Autoscan, that will scan the `Makefile.am` you wrote for Automake, and autogenerate Autoconf's `configure.ac` for you.

P1: Now all you have to do is autogenerate `Makefile.am`.

P2: Yeah, whatever. RTFM and do it yourself.[1]

Each step in the story adds a little more automation to the step that came before it: Automake uses a simple script to generate makefiles (which already go pretty far in automating compilation over manual command-typing); Autoconf tests the environment and uses that information to run Automake; Autoscan checks your code for what you need to write to make Autoconf run. Libtool works in the background to assist Automake.

If you are doing something reasonably common (and compiling straight-up C code is the #1 most common task for the Autotools), then the system will do the right thing without your needing to go off the beaten path. The trouble shows up when you need to modify the defaults, at which point you're going to need to know what macro-generating macros to modify where.

Here's a demo, in which we get Autotools to take care of Hello, World. As per the count below, the script writes nine lines of text, and it produces a full package ready for distribution to the world.

This is a shell script you can copy/paste onto your command line (as long as you make sure there are no spaces after the backslashes). Of course, it won't run until you ask your package manager to install Autotools, Autoconf, Automake, and Libtool.

- The first few lines create a directory and write `hello.c` to it.
- Then we need to hand-write `Makefile.in`, which is two lines long, and four files that are required by the GNU coding standards (so GNU Autotools won't proceed without them).
- Getting Autotools up and running after this takes us three steps:

[1] *RTFM* is an acronym meaning *Read The Manual*.

- Run `autoscan`, which produces `configure.scan`.
- Edit the file to give the specs of your project (name, version, contact email), and add the line `AM_INIT_AUTOMAKE` to initialize Automake. [Yes, this is annoying, especially given that Autoscan used Automake's `Makefile.in` to gather info, so it is well aware that we want to use Automake.] You could do this by hand; I used `sed` to directly stream the corrected version to `configure.ac`.
- Run `autoreconf` to use `configure.ac` to generate the files to ship out.

```
    #Make a directory; write a hello world program to it.
mkdir -p autodemo
cd autodemo
cat > hello.c <<\
--------------
#include <stdio.h>

int main(){ printf("Hi.\n"); }
--------------

    #Autoscan needs a Makefile.am.
cat > Makefile.am <<\
--------------
bin_PROGRAMS=hello
hello_SOURCES=hello.c
--------------

    #GNU coding standards require these; a human has to write them:
echo 'No news' > NEWS
echo 'Just run it.' > README
echo 'Kernighan & Ritchie' > AUTHORS
echo 'None yet' > ChangeLog


    #Autoscan creates configure.scan, then we edit a few
    #things to make configure.ac.
    #Notably, add AM_INIT_AUTOMAKE
autoscan
sed -e 's/FULL-PACKAGE-NAME/hello/' \
    -e 's/VERSION/1/'   \
    -e 's|BUG-REPORT-ADDRESS|/dev/null|' \
    -e '10iAM_INIT_AUTOMAKE' \
       <configure.scan > configure.ac

    #Given configure.ac, run autoreconf to produce everything.
autoreconf -i -vv
```

So how much do all these macros do? The `hello.c` program itself is a leisurely three lines, `Makefile.am` is two lines, and we wrote four one-line files, for nine lines of user-written text. Your results may differ a little, but when I run `wc -l *` in the post-script directory, I find 8,920 lines of text, including a 4,700-line `configure` script. It's so bloated because it's so portable: thie script doesn't depend on Autotools, and can be run on any system with basic POSIX-compliance.

Run `./configure` at the command prompt, and now you have the 600-line Makefile. Thanks to `zsh`'s autocomplete, I can tell you there are 216 targets in this makefile. The default target, when you just type `make` on the command line, produces the executable, and `sudo make install` would install this program if you are so interested; run `sudo make uninstall` to clear it out.

As the author of the package, you will also be interested in `make dist`, which generates a tar file with everything a user would need to unpack and run the usual `./configure; make; sudo make install` (without the aid of the Autotools system that you have on your development box). Also, `make check` verifies the tar file; don't forget that you can list multiple targets to `make`, like `make dist check` to produce the output package and then check it.

So, having run `autoscan` and `autoreconf`, two more commands and we've got a distributable package in one tar file:

```
./configure
make dist check
```

In the next few episodes, some notes on reading and producing the `Makefile.am` and `config.ac` files for when you need more than the defaults.