A tutorial for porting to autoconf & automake

l.u. 21/11/2005

A first disclaimer is that I don't really like autoconf and automake. This is not the place for longer dissertations, so I won't spend more words on this. However, it is a matter of facts that many users just like to fetch your application, and issue the usual `./configure && make && make install` right ahead.
So, this is a synthetic tutorial for moving a Makefile-based program to an autohell- (this is a popular way to refer to { autoconf, automake, libtool } that I will encourage) enabled package.

A somewhat complete, likely example is given here. If your PRE is not exactly like the one proposed, you just probably won't need to perform the corresponding following steps.

Definition of the problem:

PRE: you have a tree with

- sources in `src/`
- documentation in `doc/`
- man pages in `man/`
- some scripts in `scripts/` (in general, stuff to be installed but not compiled)
- examples in `examples/`

POST: you want to

- check for the availability of the needed headers/libraries
- possibly adjust some things (say some path in scripts, or in docimentation) at compile-time
- install everything in its adequate place

So, this is what to do for moving with the very minimum effort:

1. Cleaning up
   Move away every possible Makefile you have in the package (rename it for now)
2. Generating `configure.ac`
   Run autoscan:

   ```
   $ autoscan
   ```

   autoscan tries to produce a suitable `configure.ac` file (autoconf's driver) by performing simple analyses on the files in the package. This is enough for the moment (many people are just happy with it as permanent). Autoscan actually produces a `configure.scan` file, so let it have the name autoconf will look for:

   ```
   $ mv configure.scan configure.ac
   ```

   -- note: `configure.in` was the name used for autoconf files, now deprecated.
3. Adjusting things
   Adjust the few things left to you by autoscan: open `configure.ac` with your favourite editor

   ```
   $ vim configure.ac
   ```

   look in the very first lines for the following:

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

and replace with your stuff, e.g.:

```
AC_INIT(pippo, 2.6, paperino@staff.pippo.org)
```

4. Generating a first `configure` script
   At this point, you're ready to make autoconf produce the configure script:

```
$ autoconf
```

   This produces two files: `autom4te.cache` and `configure`. The first one is a directory used for speeding up the job of autohell tools, and may be removed when releasing the package. The latter is the shell script called by final users.
   In this status, what the configure script does is just checking for requirements as suggested by autoscan, so nothing very conclusive yet.
5. Generating suitable `Makefiles`
   We have the system checking part. We now want the building and installing part. This is given by a cooperation of automake and autoconf. Automake generates some "templates" that autoconf-generated scripts will traduce into actual Makefile. A first, "main" automake file is needed in the root of the package:

```
$ vim Makefile.am
```

   list the subdirectories where work is needed:

```
AUTOMAKE_OPTIONS = foreign
SUBDIRS = src doc examples man scripts
```

   the first line sets the mode automake will behave like. "foreign" means not GNU, and is common for avoiding boring messages about files organized differently from what gnu expects.
   The second line shows a list of subdirectories to descend for further work. The first one has stuff to compile, while the rest just needs installing, but we don't care in this file. We now prepare the Makefile.am file for each of these directories. Automake will step into each of them and produce the corresponding `Makefile.in` file. Those .in files will be used by autoconf scripts to produce the final Makefiles.
   Edit `src/Makefile.am`:

```
$ vim src/Makefile.am
```

   and insert:

```
# what flags you want to pass to the C compiler & linker
CFLAGS = --pedantic -Wall -std=c99 -O2
LDFLAGS =

# this lists the binaries to produce, the (non-PHONY, binary) targets in
# the previous manual Makefile
bin_PROGRAMS = targetbinary1 targetbinary2 [...] targetbinaryN
targetbinary1_SOURCES = targetbinary1.c myheader.h [...]
targetbinary2_SOURCES = targetbinary2.c
.
.
targetbinaryN_SOURCES = targetbinaryN.c
```

   This was the most difficult one. In general, the uppercase, suffix part like "_PROGRAMS" is called *primary* and tells partially what to perform on the argument; the lowecase, prefix (it's not given a name) tells the directory where to install.
   E.g.:

```
bin_PROGRAMS
```

   installs binaries in $(PREFIX)/bin , and

```
sbin_PROGRAMS
```

   installs in $(PREFIX)/sbin . More primaries will appear in the following, and here is a complete list of primaries. Not all can be prefixed to such primaries (see later for how to work around this problem).

   Let us now move to mans:

```
$ vim man/Makefile.am
```

insert the following in it:

```
man_MANS = firstman.1 secondman.8 thirdman.3 [...]
```

yes, automake will deduce by itself what's needed for installing from this. Now edit for scripts:

```
$ vim scripts/Makefile.am
```

insert:

```
bin_SCRIPTS = script1.sh script2.sh [...]
```

The primary "SCRIPTS" instruct makefiles to just install the arguments, without compiling of course.

So far so good. Two jobs remain to define: installing examples and installing plain docs. This is the nasty part, as automake doesn't handle primaries for installing in the usual $(PREFIX)/share/doc/pippo . The workaround is to specify a further variable and using it as prefix:

```
$ vim doc/Makefile.am
```

```
docdir = $(datadir)/doc/@PACKAGE@
doc_DATA = README DONTS
```

if "abc" is wanted for prefix, "abc*dir*" is to be specified. E.g. the code above expands to /usr/local/share/doc/pippo ("@PACKAGE@" will be expanded by autoconf when producing the final Makefile, see below). `$(datadir)` is known by all configure scripts it generates. You may look for the list of directory variables.

Similarly for examples, but we want to install in $(PREFIX)/share/examples/pippo , so:

```
$ vim examples/Makefile.am
```

```
exampledir = $(datarootdir)/doc/@PACKAGE@
example_DATA = sample1.dat sample2.dat [...]
```

All these Makefile.am files now exist, but autoconf has now to be told about them.

6. Integrating the checking (autoconf) part and the building (automake) part
   We insert now some macros in `configure.ac` for telling autoconf that the final Makefiles have to be produced after ./configure :

```
$ vim configure.ac
```

right after AC_INIT(), let initialize automake:

```
AM_INIT_AUTOMAKE(pippo, 2.6)
```

then, let autoconf generate a configure script that will output Makefiles for all of the above directories:

```
AC_OUTPUT(Makefile src/Makefile doc/Makefile examples/Makefile man/Makefile scripts/Makefile)
```

7. Making tools output the `configure` script and Makefile templates
   we have now complete instructions for generating the famous configure script run by the users when installing, that both checks for building/running requirements and generates Makefiles for actually building and installing everything in place. Let now actually make tools generate such script:

```
$ aclocal
```

This generates a file `aclocal.m4` that contains macros for automake things, e.g. AM_INIT_AUTOMAKE.

```
$ automake --add-missing
```

Automake now reads `configure.ac` and the top-level `Makefile.am`, interprets them (e.g. see further work has to be done in some subdirectories) and, for each Makefile.am produces a `Makefile.in`. The argument `--add-missing` tells automake to provide default scripts for reporting errors, installing etc, so it can be omitted in the next runs.
Finally, let autoconf build the configure script:

```
$ autoconf
```

This produces the final, full-featured `configure` shell script.

8. Further customizations
   if you need to perform custom checks, or actions in configure, just write the (shell) code somewhere in `configure.ac` (before OUTPUT commands), then run `autoconf` again. For some checks, autoconf may already provide some macro: look in the list of autoconf macros before writing useless code.

How do things work from now on

The user first runs:

```
$ ./configure
```

The shell script just generated will:

1. scan for dependencies on the basis of the AC_* macros instructed in `configure.ac`. If there's something wrong/missing in the system, an opportune error message will be dumped.
2. for each Makefile requested in AC_OUTPUT(), translate the `Makefile.in` template for generating the final `Makefile`. The main makefile will provide the most common targets like `install`, `clean`, `distclean`, `uninstall` et al.

if configure succeeds, all the `Makefile` files are available. The user then issues:

```
$ make
```

The target `all` from the main Makefile will be worked. This target expands into all the hidden targets to first build what you requested. Then, by mean of

```
# make install
```

everything is installed.

An actual example

Hellofuck is a little and stupid application for you to have a working example of something arranged with the autoconf and automake tools.
Hellofuck has 2 binary programs: `hello` and `fuck`, with a man page for each. One script "hellofuck.sh" is also shipped, and documented by a man page. A plain text file outlines the features of hellofuck, and is installed in the system's doc folder as shared data.
Download hellofuck and browse the folder for a first-hand on autohell.
