Tip 59: Use a package manager

Ben Klemens

28 January 2012

level: Computer user
purpose: let a librarian organize your libraries

Oh man, if you are not using a package manager, you are missing out.

If you are already using one, then here's a simple tip: open a listing of available packages via whatever means you are used to, and skip down to the packages beginning with `lib`. There, you'll find a few hundred C libraries available for your use. Some will be attached to a specific program, but these are the portions of the program that the authors felt might be useful outside of the program itself, so some of those may be useful to you.

Different package managers have different customs, but they tend to split a single library into two or three packages, typically a base libsomething package for installation with executables that rely on that library; a libsomething-dev or libsomething-devel package for developers (that's you), including the headers; and a libsomething-doc package, in case you decide you want a local copy of the online documentation. Be sure to install at least the first two.

For those of you who aren't using a package manager, the rest of this section is a brief introduction, in two parts. The first is about language-specific package managers, explaining why we C users are going to be sticking with the system-wide package managers. The second part discusses the options for system-wide package managers.

CPAN, et al You may be familiar with the repository systems for Perl, Python, Ruby, and company. From within the interpreter, you can call a command that will pull a package from a central repository and install it into a local directory. They're pretty spiffy, and when they work they save a whole lot of trouble.

What would a package manager for C look like? For the most part, what other languages call a *package*, we call a *library*. If you give me the header files, a compiled object file, and some documentation, I can use the library for my own code. Or you can give me the source and I can compile the object files for myself.

Here in reality, there are complications. The library may depend on other libraries, so we will need to have a list of dependencies, and will then need to work out from where to get them. We may need the installation location. The compilation may depend on nonstandard tools. These are exactly the considerations that a package manager keeps track of.

Language-specific managers, like the CPAN (Comprehensive Perl Archive Network) or the Ruby Gem system solve many of the dependency problems by imposing some restrictions. The first is of course that the package must be written in the language of the associated system. If your R package depends on a Ruby gem, well, good luck with that. Of course, every scripting language package manager includes a mechanism for linking to C code, although some are more harrowing than others. None that I have seen provide an automated means of dealing with a C library that calls other C libraries, and as a result, we often find script-supporting C code that reimplements all the basics, just in case the build system at the central repository may not have a copy of Glib installed.

There are people whose livelihoods depend on advancing a given language, and those are the people who (hire people to) put in the labor to make the repositories work. For example, the R implementation of the S programming language, for example, is maintained by The R Foundation, a Viennese non-profit funded by pharmaceutical companies, academic statistics departments, et al. The R Foundation distributes the one and only R interpreter, which is of course built to talk to the CRAN.

There are no insurmountable technical reasons for why a C package manager couldn't happen, but there are social reasons preventing this. C is far bigger than any one foundation or compiler, and it is unlikely that we will ever see one effort to be a central repository for C code really take off (though there have been many efforts, including a CCAN). Further, most of what we need is already provided by the system-wide package managers.

The system-wide package manager These provide all of the tools one would need to get a full POSIX subsystem up and running. Of course, you can't have a POSIX subsystem without C tools and libraries, [this is trivially true, because the standard requires certain C libraries], so any POSIX-oriented package manager will have a system in place for handling libraries.

As above, there isn't all that much distance between a C library and a package as it is typically understood. To bridge the gap, we need an address where our package manager can find the files, a means of expressing what other packages have to be installed before this one, and perhaps some pre- and post-install scripts that establish the important environment variables and go through all that stuff about `./configure; make; sudo make install` for the user.

There isn't all that much serious divergence from those basics. Here's my sampling from Wikipedia's list[1]. They are variously maintained by a mix of operating system vendors and volunteers.

Windows The front-runner here is Cygwin[2], which is based on a Windows-native base library (DLL) that provides all the POSIX-type stuff missing from a basic Windows machine. In theory, a program compiled under Cygwin could run on boxes that don't have Cygwin installed if you were to copy over the Cygwin DLL with your compiled program; I've never tried it.

[1] `http://en.wikipedia.org/wiki/List_of_software_package_management_systems`

[2] `http://cygwin.com`

On top of this very basic concept, Cygwin has its own package management system, which is pretty together.

Mac OS X There are many. Deb-based Fink[3] seems to be the front-runner.

Debian/RPM RPM stands for Red Hat Package Manger, and Debian stands for Debra and Ian. Both lean toward precompiled binary packages, meaning that there's a different package and different subrepository for every type of OS. Between the two of them, it's hard to say much about the differences. I've found it a little easier to write packages for RPM.

Source-based Autotools already gets us most of the way toward standardized installation (I promise a discussion of Autotools soon), because its `configure` script works out all the platform-specific stuff. From there, there are several systems that will do the additional stuff about resolving dependencies and such.

In all cases, there's little technical difference between the different versions in each category. Generally, the older and biggest network will have the most packages available.

I sometimes lament the great diversity here, because each system is slightly incompatible with every other: ¿Should my package depend on libgsl-dev, libgsl-0-dev, or libgsl-devel? Also, I'm somewhat certain that five years from now the above list of recommendations will be very different.

But in the mean time, the recommendation is simple: if you are a Mac or Windows user, get a package manager immediately. It's already an immense streamlining of the software-obtention process, but if you're a C author, it's how you're going to get your libraries.

[3] `http://finkproject.org`