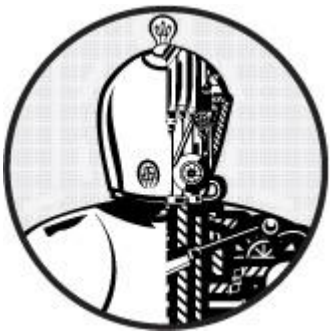
15.7 Looking Forward: Compiling Packages

The world of compilers and scripting languages is vast and constantly expanding. As of this writing, new compiled languages such as Go (golang) and Swift are gaining popularity.

The LLVM compiler infrastructure set (*http://llvm.org/*) has significantly eased compiler development. If you're interested in how to design and implement a compiler, two good books are *Compilers: Principles, Techniques, and Tools*, 2nd edition, by Alfred V. Aho et al. (Addison-Wesley, 2006) and *Modern Compiler Design*, 2nd edition, by Dick Grune et al. (Springer, 2012). For scripting language development, it's usually best to look for online resources, as the implementations vary widely.

Now that you know the basics of the programming tools on the system, you're ready to see what they can do. The next chapter is all about how you can build packages on Linux from source code.

Chapter 16. Introduction to Compiling Software From C Source Code



Most nonproprietary third-party Unix software packages come as source code that you can build and install. One reason for this is that Unix (and Linux itself) has so many different flavors and architectures that it would be difficult to distribute binary packages for all possible platform combinations. The other reason, which is at least as important, is that widespread source code distribution throughout the Unix community encourages users to contribute bug fixes and new features to software, giving meaning to the term *open source*.

You can get nearly everything you see on a Linux system as source code—from the kernel and C library to the web browsers. It's even possible to update and augment your entire system by (re-)installing parts of your system from the source code. However, you probably *shouldn't* update your machine by installing *everything* from source code, unless you really enjoy the process or have some other reason.

Linux distributions typically provide easier ways to update core parts of the system, such as the programs in */bin*, and one particularly important property of distributions is that they usually fix security problems very quickly. But don't expect your distribution to provide everything for you. Here are some reasons why you may want to install certain packages yourself:

- To control configuration options.
- To install the software anywhere you like. You can even install several different versions of the same package.
- To control the version that you install. Distributions don't always stay up-to-date with the latest versions of all packages, particularly add-ons to software packages (such as Python libraries).
- To better understand how a package works.

16.1 Software Build Systems

There are many programming environments on Linux, from traditional C to interpreted scripting languages such as Python. Each typically has at least one distinct system for building and installing packages in addition to the tools that a Linux distribution provides.

We're going to look at compiling and installing C source code in this chapter with only one of these build systems—the configuration scripts generated from the GNU autotools suite. This system is generally considered stable, and many of the basic Linux utilities use it. Because it's based on existing tools such as `make`, after you see it in action, you'll be able to transfer your knowledge to other build systems.

Installing a package from C source code usually involves the following steps:

1. Unpack the source code archive.
2. Configure the package.
3. Run `make` to build the programs.

4. Run `make install` or a distribution-specific install command to install the package.

NOTE

You should understand the basics in Chapter 15 before proceeding with this chapter.

16.2 Unpacking C Source Packages

A package's source code distribution usually comes as a *.tar.gz*, *.tar.bz2*, or *.tar.xz* file, and you should unpack the file as described in 2.18 Archiving and Compressing Files. Before you unpack, though, verify the contents of the archive with `tar tvf` or `tar ztvf`, because some packages don't create their own subdirectories in the directory where you extract the archive.

Output like this means that the package is probably okay to unpack:

```
package-1.23/Makefile.in
package-1.23/README
package-1.23/main.c
package-1.23/bar.c
--snip--
```

However, you may find that not all files are in a common directory (like *package-1.23* in the preceding example):

```
Makefile
README
main.c
--snip--
```

Extracting an archive like this one can leave a big mess in your current directory. To avoid that, create a new directory and `cd` there before extracting the contents of the archive.

Finally, beware of packages that contain files with absolute pathnames like this:

```
/etc/passwd
/etc/inetd.conf
```

You likely won't come across anything like this, but if you do, remove the archive from your system. It probably contains a Trojan horse or some other malicious code.

16.2.1 Where to Start

Once you've extracted the contents of a source archive and have a bunch of files in front of you, try to get a feel for the package. In particular, look for the files *README* and *INSTALL*. Always look at any *README* files first because they often contain a description of the package, a small manual, installation hints, and other useful information. Many packages also come with *INSTALL* files with instructions on how to compile and install the package. Pay particular attention to special compiler options and definitions.

In addition to *README* and *INSTALL* files, you will find other package files that roughly fall into three categories:

- Files relating to the `make` system, such as *Makefile*, *Makefile.in*, *configure*, and *CMakeLists.txt*. Some very old packages come with a Makefile that you may need to modify, but most use a configuration utility such

as GNU autoconf or CMake. They come with a script or configuration file (such as *configure* or *CMakeLists.txt*) to help generate a Makefile from *Makefile.in* based on your system settings and configuration options.

- Source code files ending in *.c*, *.h*, or *.cc*. C source code files may appear just about anywhere in a package directory. C++ source code files usually have *.cc*, *.C*, or *.cxx* suffixes.
- Object files ending in *.o* or binaries. Normally, there aren't any object files in source code distributions, but you might find some in rare cases when the package maintainer is not permitted to release certain source code and you need to do something special in order to use the object files. In most cases, object (or binary executable) files in a source distribution mean that the package wasn't put together well, and you should run `make clean` to make sure that you get a fresh compile.

16.3 GNU Autoconf

Even though C source code is usually fairly portable, differences on each platform make it impossible to compile most packages with a single Makefile. Early solutions to this problem were to provide individual Makefiles for every operating system or to provide a Makefile that was easy to modify. This approach evolved into scripts that generate Makefiles based on an analysis of the system used to build the package.

GNU autoconf is a popular system for automatic Makefile generation. Packages using this system come with files named *configure*, *Makefile.in*, and *config.h.in*. The *.in* files are templates; the idea is to run the `configure` script in order to discover the characteristics of your system, then make substitutions in the *.in* files to create the real build files. For the end user, it's easy; to generate a Makefile from *Makefile.in*, run `configure`:

```
$ ./configure
```

You should get a lot of diagnostic output as the script checks your system for prerequisites. If all goes well, `configure` creates one or more Makefiles and a *config.h* file, as well as a cache file (*config.cache*), so that it doesn't need to run certain tests again.

Now you can run `make` to compile the package. A successful `configure` step doesn't necessarily mean that the `make` step will work, but the chances are pretty good. (See 16.6 Troubleshooting Compiles and Installations for troubleshooting failed configures and compiles.)

Let's get some firsthand experience with the process.

NOTE

At this point, you must have all of the required build tools available on your system. For Debian and Ubuntu, the easiest way is to install the build-essential package; in Fedora-like systems, use the Chapter 15 groupinstall.

16.3.1 An Autoconf Example

Before discussing how you can change the behavior of autoconf, let's look at a simple example so that you know what to expect. You'll install the GNU coreutils package in your own home directory (to make sure that you don't mess up your system). Get the package from *http://ftp.gnu.org/gnu/coreutils/* (the latest version is usually the best), unpack it, change to its directory, and configure it like this:

```
$ ./configure --prefix=$HOME/mycoreutils
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
--snip--
```

```
config.status: executing po-directories commands
config.status: creating po/POTFILES
config.status: creating po/Makefile
```

Now run `make`:

```
$ make
  GEN       lib/alloca.h
  GEN       lib/c++defs.h
--snip--
make[2]: Leaving directory '/home/juser/coreutils-8.22/gnulib-tests'
make[1]: Leaving directory '/home/juser/coreutils-8.22'
```

Next, try to run one of the executables that you just created, such as *./src/ls*, and try running `make check` to run a series of tests on the package. (This might take a while, but it's interesting to see.)

Finally, you're ready to install the package. Do a dry run with `make -n` first to see what `make install` does without actually doing the install:

```
$ make -n install
```

Browse through the output, and if nothing seems strange (such as installing anywhere other than your *mycoreutils* directory), do the install for real:

```
$ make install
```

You should now have a subdirectory named *mycoreutils* in your home directory that contains *bin*, *share*, and other subdirectories. Check out some of the programs in *bin* (you just built many of the basic tools that you learned in Chapter 2). Finally, because you configured the *mycoreutils* directory to be independent of the rest of your system, you can remove it completely without worrying about causing damage.

16.3.2 Installing Using a Packaging Tool

On most distributions, it's possible to install new software as a package that you can maintain later with your distribution's packaging tools. Debian-based distributions such as Ubuntu are perhaps the easiest; rather than running a plain `make install`, you can do it with the `checkinstall` utility, as follows:

```
# checkinstall make install
```

Use the `--pkgname=`*`name`* option to give your new package a specific name.

Creating an RPM package is a little more involved, because you must first create a directory tree for your package(s). You can do this with the `rpmdev-setuptree` command; when complete, you can use the `rpmbuild` utility to work through the rest of the steps. It's best to follow an online tutorial for this process.

16.3.3 configure Script Options

You've just seen one of the most useful options for the `configure` script: using `--prefix` to specify the installation directory. By default, the `install` target from an autoconf-generated Makefile uses a *prefix* of */usr/local*—that is, binary programs go in */usr/local/bin*, libraries go in */usr/local/lib*, and so on. You will often want to change that prefix like this:

```
$ ./configure --prefix=new_prefix
```

Most versions of `configure` have a `--help` option that lists other configuration options. Unfortunately, the list is usually so long that it's sometimes hard to figure out what might be important, so here are some essential options:

- **`--bindir=`*`directory`*** Installs executables in *`directory`*.
- **`--sbindir=`*`directory`*** Installs system executables in *`directory`*.
- **`--libdir=`*`directory`*** Installs libraries in *`directory`*.
- **`--disable-shared`** Prevents the package from building shared libraries. Depending on the library, this can save hassles later on (see 15.1.4 Shared Libraries).
- **`--with-`*`package`*`=`*`directory`*** Tells `configure` that *`package`* is in *`directory`*. This is handy when a necessary library is in a nonstandard location. Unfortunately, not all `configure` scripts recognize this type of option, and it can be difficult to determine the exact syntax.

Using Separate Build Directories

You can create separate build directories if you want to experiment with some of these options. To do so, create a new directory anywhere on the system and, from that directory, run the `configure` script in the original package source code directory. You'll find that `configure` then makes a symbolic link farm in your new build directory, where all of the links point back to the source tree in the original package directory. (Some developers prefer that you build packages this way, because the original source tree is never modified. This is also useful if you want to build for more than one platform or configuration option set using the same source package.)

16.3.4 Environment Variables

You can influence `configure` with environment variables that the `configure` script puts into `make` variables. The most important ones are `CPPFLAGS`, `CFLAGS`, and `LDFLAGS`. But be aware that `configure` can be very picky about environment variables. For example, you should normally use `CPPFLAGS` instead of `CFLAGS` for header file directories, because `configure` often runs the preprocessor independently of the compiler.

In `bash`, the easiest way to send an environment variable to `configure` is by placing the variable assignment in front of `./configure` on the command line. For example, to define a `DEBUG` macro for the preprocessor, use this command:

```
$ CPPFLAGS=-DDEBUG ./configure
```

NOTE

You can also pass a variable as an option to `configure`; for example:

```
$ ./configure CPPFLAGS=-DDEBUG
```

Environment variables are especially handy when `configure` doesn't know where to look for third-party include files and libraries. For example, to make the preprocessor search in *`include_dir`*, run this command:

```
$ CPPFLAGS=-Iinclude_dir ./configure
```

As shown in 15.2.6 Standard Macros and Variables, to make the linker look in ***`lib_dir`***, use this command:

```
$ LDFLAGS=-Llib_dir ./configure
```

If ***`lib_dir`*** has shared libraries (see 15.1.4 Shared Libraries), the previous command probably won't set the runtime dynamic linker path. In that case, use the `-rpath` linker option in addition to `-L`:

```
$ LDFLAGS="-Llib_dir -Wl,-rpath=lib_dir" ./configure
```

Be careful when setting variables. A small slip can trip up the compiler and cause `configure` to fail. For example, say you forget the `-` in `-I`, as shown here:

```
$ CPPFLAGS=Iinclude_dir ./configure
```

This yields an error like this:

```
configure: error: C compiler cannot create executables
See 'config.log' for more details
```

Digging through the *config.log* generated from this failed attempt yields this:

```
configure:5037: checking whether the C compiler works
configure:5059: gcc Iinclude_dir conftest.c >&5
gcc: error: Iinclude_dir: No such file or directory
configure:5063: $? = 1
configure:5101: result: no
```

16.3.5 Autoconf Targets

Once you get `configure` working, you'll find that the Makefile that it generates has a number of other useful targets in addition to the standard `all` and `install`:

- **`make clean`** As described in Chapter 15, this removes all object files, executables, and libraries.
- **`make distclean`** This is similar to `make clean` except that it removes all automatically generated files, including Makefiles, *config.h*, *config.log*, and so on. The idea is that the source tree should look like a newly unpacked distribution after running `make distclean`.
- **`make check`** Some packages come with a battery of tests to verify that the compiled programs work properly; the command `make check` runs the tests.
- **`make install-strip`** This is like `make install` except that it strips the symbol table and other debugging information from executables and libraries when installing. Stripped binaries require much less space.

16.3.6 Autoconf Log Files

If something goes wrong during the configure process and the cause isn't obvious, you can examine *config.log* to find the problem. Unfortunately, *config.log* is often a gigantic file, which can make it difficult to locate the exact source of the problem.

The general approach to finding the problem is to go to the very end of *config.log* (for example, by pressing G in `less`) and then page back up until you see the problem. However, there is still a lot of stuff at the end because `configure` dumps its entire environment there, including output variables, cache variables, and other definitions. So rather than going to the end and paging up, go to the end and search backward for a string such as `for more details` or some other part near the end of the failed `configure` output. (Remember that you can initiate a reverse search in `less` with the `?` command.) There's a good chance that the error will be just above what your search finds.

16.3.7 pkg-config

There are so many third-party libraries that keeping all of them in a common location can be messy. However,

installing each with a separate prefix can lead to problems when building packages that require these third-party libraries. For example, if you want to compile OpenSSH, you need the OpenSSL library. How do you tell the OpenSSH configuration process the location of the OpenSSL libraries and which libraries are required?

Many libraries now use the `pkg-config` program not only to advertise the locations of their include files and libraries but also to specify the exact flags that you need to compile and link a program. The syntax is as follows:

```
$ pkg-config options package1 package2 ...
```

For example, to find the libraries required for OpenSSL, you can run this command:

```
$ pkg-config --libs openssl
```

The output should be something like this:

```
-lssl -lcrypto
```

To see all libraries that `pkg-config` knows about, run this command:

```
$ pkg-config --list-all
```

How pkg-config Works

If you look behind the scenes, you will find that `pkg-config` finds package information by reading configuration files that end with *.pc*. For example, here is *openssl.pc* for the OpenSSL socket library, as seen on an Ubuntu system (located in */usr/lib/i386-linux-gnu/pkgconfig*):

```
prefix=/usr
exec_prefix=${prefix}
libdir=${exec_prefix}/lib/i386-linux-gnu
includedir=${prefix}/include


Name: OpenSSL
Description: Secure Sockets Layer and cryptography libraries and tools
Version: 1.0.1
Requires:
Libs: -L${libdir} -lssl -lcrypto
Libs.private: -ldl -lz
Cflags: -I${includedir} exec_prefix=${prefix}
```

You can change this file, for example, by adding `-Wl,-rpath=${libdir}` to the library flags to set a runtime dynamic linker path. However, the bigger question is how `pkg-config` finds the *.pc* files in the first place. By default, `pkg-config` looks in the *lib/pkgconfig* directory of its installation prefix. For example, a `pkg-config` installed with a */usr/local* prefix looks in */usr/local/lib/ pkgconfig*.

Installing pkg-config Files in Nonstandard Locations

Unfortunately, by default, `pkg-config` does not read any *.pc* files outside its installation prefix. So a *.pc* file that's in a nonstandard location, such as */opt/ openssl/lib/pkgconfig/openssl.pc*, will be out of the reach of any stock `pkg-config` installation. There are two basic ways to make *.pc* files available outside of the `pkg-`

`config` installation prefix:

- Make symbolic links (or copies) from the actual *.pc* files to the central *pkgconfig* directory.
- Set your `PKG_CONFIG_PATH` environment variable to include any extra *pkgconfig* directories. This strategy does not work well on a system-wide basis.

16.4 Installation Practice

Knowing *how* to build and install software is good, but knowing *when* and *where* to install your own packages is even more useful. Linux distributions try to cram in as much software as possible at installation, and you should always check whether it would be best to install a package yourself instead. Here are the advantages of doing installs on your own:

- You can customize package defaults.
- When installing a package, you often get a clearer picture of how to use the package.
- You control the release that you run.
- It's easier to back up a custom package.
- It's easier to distribute self-installed packages across a network (as long as the architecture is consistent and the installation location is relatively isolated).

Here are the disadvantages:

- It takes time.
- Custom packages do not automatically upgrade themselves. Distributions keep most packages up-to-date without requiring much work. This is a particular concern for packages that interact with the network, because you want to ensure that you always have the latest security updates.
- If you don't actually use the package, you're wasting your time.
- There is a potential for misconfiguring packages.

There's not much point in installing packages such as the ones in the coreutils package that you built earlier in the chapter (`ls`, `cat`, and so on) unless you're building a very custom system. On the other hand, if you have a vital interest in network servers such as Apache, the best way to get complete control is to install the servers yourself.

16.4.1 Where to Install

The default prefix in GNU autoconf and many other packages is */usr/local*, the traditional directory for locally installed software. Operating system upgrades ignore */usr/local*, so you won't lose anything installed there during an operating system upgrade and for small local software installations, */usr/local* is fine. The only problem is that if you have a lot of custom software installed, this can turn into a terrible mess. Thousands of odd little files can make their way into the */usr/local* hierarchy, and you may have no idea where the files came from.

If things really start to get unruly, you should create your own packages as described in 16.3.2 Installing Using a Packaging Tool.

16.5 Applying a Patch

Most changes to software source code are available as branches of the developer's online version of the source code (such as a git repository). However, every now and then, you might get a *patch* that you need to apply against source code to fix bugs or add features. You may also see the term *diff* used as a synonym for patch,

because the `diff` program produces the patch.

The beginning of a patch looks something like this:

```
--- src/file.c.orig    2015-07-17 14:29:12.000000000 +0100
+++ src/file.c  2015-09-18 10:22:17.000000000 +0100
@@ -2,16 +2,12 @@
```

Patches usually contain alterations to more than one file. Search the patch for three dashes in a row (`---`) to see the files that have alterations and always look at the beginning of a patch to determine the required working directory. Notice that the preceding example refers to *src/file.c*. Therefore, you should change to the directory that contains *src* before applying the patch, *not* to the *src* directory itself.

To apply the patch, run the `patch` command:

```
$ patch -p0 < patch_file
```

If everything goes well, `patch` exits without a fuss, leaving you with an updated set of files. However, `patch` may ask you this question:

```
File to patch:
```

This usually means that you are not in the correct directory, but it could also indicate that your source code does not match the source code in the patch. In this case, you're probably out of luck: Even if you could identify some of the files to patch, others would not be properly updated, leaving you with source code that you could not compile.

In some cases, you might come across a patch that refers to a package version like this:

```
--- package-3.42/src/file.c.orig    2015-07-17 14:29:12.000000000 +0100
+++ package-3.42/src/file.c  2015-09-18 10:22:17.000000000 +0100
```

If you have a slightly different version number (or you just renamed the directory), you can tell `patch` to strip leading path components. For example, say you were in the directory that contains *src* (as before). To tell `patch` to ignore the *package-3.42/* part of the path (that is, strip one leading path component), use `-p1`:

```
$ patch -p1 < patch_file
```

16.6 Troubleshooting Compiles and Installations

If you understand the difference between compiler errors, compiler warnings, linker errors, and shared library problems as described in Chapter 15, you shouldn't have too much trouble fixing many of the glitches that arise when building software. This section covers some common problems. Although you're unlikely to run into any of these when building using autoconf, it never hurts to know what these kinds of problems look like.

Before covering specifics, make sure that you can read certain kinds of `make` output. It's important to know the difference between an error and an ignored error. The following is a real error that you need to investigate:

```
make: *** [target] Error 1
```

However, some Makefiles suspect that an error condition might occur but know that these errors are harmless. You can usually disregard any messages like this:

```
make: *** [target] Error 1 (ignored)
```

Furthermore, GNU `make` often calls itself many times in large packages, with each instance of `make` in the error message marked with `[N]`, where *N* is a number. You can often quickly find the error by looking at the

`make` error that comes *directly* after the compiler error message. For example:

```
[compiler error message involving file.c]
make[3]: *** [file.o] Error 1
make[3]: Leaving directory '/home/src/package-5.0/src'
make[2]: *** [all] Error 2
make[2]: Leaving directory '/home/src/package-5.0/src'
make[1]: *** [all-recursive] Error 1 make[1]: Leaving directory
'/home/src/package-5.0/'
make: *** [all] Error 2
```

The first three lines practically give it away: The trouble centers around *file.c* located in */home/src/package-5.0/src*. Unfortunately, there is so much extra output that it can be difficult to spot the important details. Learning how to filter out the subsequent `make` errors goes a long way toward digging out the real cause.

16.6.1 Specific Errors

Here are some common build errors that you might encounter.

Problem

Compiler error message:

```
src.c:22: conflicting types for 'item'
/usr/include/file.h:47: previous declaration of 'item'
```

Explanation and Fix

The programmer made an erroneous redeclaration of `item` on line 22 of *src.c*. You can usually fix this by removing the offending line (with a comment, an `#ifdef`, or whatever works).

Problem

Compiler error message:

```
src.c:37: 'time_t' undeclared (first use this function)
--snip--
src.c:37: parse error before '...'
```

Explanation and Fix

The programmer forgot a critical header file. The manual pages are the best way to find the missing header file. First, look at the offending line (in this case, line 37 in *src.c*). It's probably a variable declaration like the following:

```
time_t v1;
```

Search forward for `v1` in the program for its use around a function call. For example:

```
v1 = time(NULL);
```

Now run `man 2 time` or `man 3 time` to look for system and library calls named `time()`. In this case, the section 2 manual page has what you need:

```
SYNOPSIS
```

```
#include <time.h>


time_t time(time_t *t);
```

This means that `time()` requires *time.h*. Place `#include <time.h>` at the beginning of *src.c* and try again.

Problem

Compiler (preprocessor) error message:

```
src.c:4: pkg.h: No such file or directory
(long list of errors follows)
```

Explanation and Fix

The compiler ran the C preprocessor on *src.c* but could not find the *pkg.h* include file. The source code likely depends on a library that you need to install, or you may just need to provide the compiler with the nonstandard include path. Usually, you will just need to add a `-I` include path option to the C preprocessor flags (`CPPFLAGS`). (Keep in mind that you might also need a `-L` linker flag to go along with the include files.)

If it doesn't look as though you're missing a library, there's an outside chance that you're attempting a compile for an operating system that this source code does not support. Check the Makefile and *README* files for details about platforms.

If you're running a Debian-based distribution, try the `apt-file` command on the header filename:

```
$ apt-file search pkg.h
```

This might find the development package that you need. For distributions that provide `yum`, you can try this instead:

```
$ yum provides */pkg.h
```

Problem

`make` error message:

```
make: prog: Command not found
```

Explanation and Fix

To build the package, you need *prog* on your system. If *prog* is something like `cc`, `gcc`, or `ld`, you don't have the development utilities installed on your system. On the other hand, if you think *prog* is already installed on your system, try altering the Makefile to specify the full pathname of *prog*.

In rare cases, `make` builds *prog* and then uses *prog* immediately, assuming that the current directory (`.`) is in your command path. If your `$PATH` does not include the current directory, you can edit the Makefile and change *prog* to `./prog`. Alternatively, you could append `.` to your path temporarily.

16.7 Looking Forward

We've only touched on the basics of building software. Here are some more topics that you can explore after you get the hang of your own builds:

- **Understanding how to use build systems other than autoconf, such as CMake and SCons.**

- **Setting up builds for your own software.** If you're writing your own software, you want to choose a build system and learn to use it. For GNU autoconf packaging, *Autotools* by John Calcote (No Starch Press, 2010) can help you out.
- **Compiling the Linux kernel.** The kernel's build system is completely different from that of other tools. It has its own configuration system tailored to customizing your own kernel and modules. The procedure is straightforward, though, and if you understand how the boot loader works, you won't have any trouble with it. However, you should be careful when doing so; make sure that you always keep your old kernel handy in case you can't boot with a new one.
- **Distribution-specific source packages.** Linux distributions maintain their own versions of software source code as special source packages. Sometimes you can find useful patches that expand functionality or fix problems in otherwise unmaintained packages. The source package management systems include tools for automatic builds, such as Debian's `debuild` and the RPM-based `mock`.

Building software is often a stepping-stone to learning about programming and software development. The tools you've seen in the past two chapters take the mystery out of where your system software came from. It's not difficult to take the next steps of looking inside the source code, making changes, and creating your own software.