

Software Management

The basic unit of software in Debian — and, indeed, in almost every major operating system today — is the *package*. A lot of effort goes into making a package that's easy to install and works on the first try. Each package needs to declare its relationship to other packages; perhaps it needs a library written by another author, or a separate program to round out functionality. Packages also need to monitor and work with their configuration files to ease upgrades and preserve changes.

A large part of administering a system — whether by a home user or a paid administrator taking care of thousands of machines — is maintaining the software installed on the computers. This chapter provides some basic background on packaging systems in general, as well as hands-on examples for maintaining software on a Debian system.

The Anatomy of a Software Package

This section reviews the properties common to almost all packaging systems, as well as some traits unique to Debian packages.

Common package properties

Everything about a packaging system, whether source or binary, is designed around standardization and ease of use. With the source code available, each user is perfectly capable of configuring and compiling (*building*, which is not to be confused with writing) his own software, but the end result is a very unique machine that can be difficult to administer.

4

C H A P T E R



In This Chapter

- Package anatomy
- Debian packages
- Finding and examining packages
- Installing packages
- Removing packages
- Configuring packages
- Integrity-checking
- Package repositories
- apt pinning



Source packages versus binary packages

A *source package* is a package that is made up completely of source code. Source code is useful for programmers but must be built or compiled before it can actually be run. Building packages can take long periods of time even on fast machines, and it takes even longer on older and slower machines. *Binary packages* are already compiled and are distributed in a ready-to-use manner. A perennial debate in the GNU/Linux world is whether to distribute source packages or binary packages to end-users. The difference is simple: If *source* packages are distributed, the user builds and installs the software on his machine; if *binary* packages are distributed, only the installation part is necessary. In each case, the package system will configure the source, compile it, install it, and throughout the process run any scripts that are required.

Compiling a package doesn't necessarily need to be complex. There are several source-based packaging systems in use, and they all make the job of configuring the source package, compiling it, and installing it relatively pain-free. In fact, this is a basic precept of any packaging system: Do as much of the work as possible for the user.

In the case of source packages, there are typically numerous default source-configuration values that users can, should they choose to, override. This allows for a degree of build-time customization that can vary from package to package. However, most of these systems allow for one notable customization—the ability to specify compiler settings. Customizing compiler settings allows a greater degree of control over the compilation process, which can include performance-related optimization.



Caution

There have been many studies conducted on the effect of changing default compiler settings, and in almost every common case it proved that the resulting software was slower and more crash prone than software compiled with conservative settings—the settings that Debian package maintainers choose by default. If you wish to investigate changing compiler settings and rebuilding packages, avail yourself of the compiler documentation and take some time to research studies that have been done in the field.

Though source-based packaging systems exist and are used by some, binary-based packaging systems are far more prevalent. In these systems, the package maintainer configures the source and compiles the software, and then packages the result into a simple file, which is distributed to the end-user. Compared to compiling a source package, installing a binary package is very fast. A large source package can take hours to compile on high-end hardware, whereas a binary package created from that source code can be installed in seconds. Binary packages also result in a far more consistent end-user installation because each end-user installs the same binary package. Last but certainly not least, a binary package maintainer will typically be very familiar with the package she is producing and will have in-depth knowledge of the ideal compiler and source settings to use.

Binary packages are not without their drawbacks, however. Notably, they tend to require more ancillary software to be installed because disabling the use of this software must be done with the source code. In the interests of making the package applicable to the widest audience, most of the knobs that control the use of external software are turned all the way up.

Debian is a binary-based distribution. It was decided long ago that the tradeoffs for using a binary packaging format far outweighed the added flexibility gained by requiring users to compile each and every bit of software on their system. Debian does distribute source packages, though: Each binary package in Debian that the user installs has an associated source package from which the binary package is built. So while Debian's primary means of providing software for installation is via binary packages, users who really need to change compile-time settings can do so without *too* much trouble.

Dependencies

Whether dealing with source packages or binary packages, another common theme is *dependencies*, or more generally, *package relationships*. Over the years, more and more common tasks have been moved into shared libraries — code that each developer can use instead of writing their own each time. Every time this happens, a dependency is created.

Package relationships in their most common form allow a package to declare a dependency on another package. A calculator might declare a dependency on a library containing common math functions. Graphical applications would declare a dependency on a common graphical user interface toolkit library. Though *dependencies*, as they're called, are the most widely understood and implemented package relationships, they're not the only kind. In advanced packaging systems like Debian's, packages can also declare a number of other relationships (see the list of possible relationships in the "Package relationships" section of this chapter).

Installation scripts

Aside from containing the software itself, as well as declaring any dependencies or other package relationships, packages typically include *installation scripts*. These are small programs that act on the user's machine in a fairly intelligent manner. These scripts might migrate configuration files to new locations or upgrade configuration files to new versions. They might print a warning telling the user to be careful with a particularly dangerous piece of software. Because these are programs in their own right — typically small, but not necessarily — they're the active portion of a given package.

Debian's packages

As I hinted in the previous subsection, Debian's package management system is quite advanced. It's been evolving for a decade and has been continually refined, though that's not to say it's the be-all and end-all of package management. However,

most users find the most compelling aspect of Debian to be its package management system. This subsection discusses its particular properties in detail.

Package relationships

The quality of a package in Debian has a lot to do with how well it declares its relationships to other packages, and there certainly can be a lot of relationships to declare. The full list of package relationships available for a Debian package to declare includes *Depends*, *Pre-Depends*, *Recommends*, *Suggests*, *Conflicts*, *Provides*, and *Replaces*. Table 4-1 describes how each of these fields is used.

Table 4-1
Package Relationships

<i>Relationship Field</i>	<i>Description and Usage</i>
Depends	If package A requires package B for use by the user, or if package B is required for package A to be configured, package A must declare that it Depends on package B.
Pre-Depends	If package A's installation procedure requires package B, package A must declare that it Pre-Depends upon package B.
Recommends	When package A's functionality is enhanced with an additional package, B, then package A must declare that it Recommends the installation of package B.
Suggests	The Suggests relationship is a weaker form of the Recommends relationship. It's usually used to indicate that some related software can be used alongside the package declaring the relationship.
Conflicts	When package A absolutely can <i>not</i> exist on the user's machine at the same time as package B, package A must declare that it Conflicts with package B.
Provides	When package A provides identical functionality to package B, it may declare that it Provides package B. Therefore, when a package Depends on package B, its dependency may be satisfied by package A or B.
Replaces	Occasionally, as when a package maintainer is reorganizing a set of closely related packages, package A will contain the same files as another package, B. When this is the case, package A declares that it Replaces package B, and any files from package A will overwrite files in package B. Without this relationship being declared, if two packages claim the same file, the package management system will produce an error and abort the operation.

Maintaining package relationships is a large part of a Debian maintainer's job because these relatively simple concepts make software management easy and painless for the user. However, choosing which relationships to declare with which packages is something of an art.

Maintainer scripts

Debian packages have *maintainer scripts*, which are scripts the package maintainer writes to help ease the installation, removal, or upgrade of his packages by the user. There isn't a single installation script. Rather, there are five scripts that the maintainer can use, although not every package uses every script: the *pre-installation* script, the *post-installation* script, the *pre-removal* script, the *post-removal* script, and the *debconf* script. These scripts are all run automatically during different portions of package management operations.

The names should be self-explanatory, except for perhaps the *debconf* script. During the installation of your Debian system, you were asked a number of questions. These were done using Debian's configuration framework, the self-titled *debconf*. *debconf* scripts primarily serve to collect information from the user to provide a sensible default configuration for a package. However, *debconf* scripts are also occasionally used to print messages and warnings to the user.

Each of the maintainer scripts is provided with a great deal of information about the user's system before they're run, including whether the package is being upgraded, what version of the package was last installed, any errors that might have occurred, and so on.

Configuration files

In addition to the regular files that make up the software, the declarations of package relationships, and the maintainer scripts that ease installation and upgrades, a major component of a Debian package is its configuration files. Unlike regular files, which overwrite old versions or preexisting versions without any prompting on the part of the user, configuration files are treated specially. A large part of Debian's appeal to system administrators is the care taken to preserve any changes made to a configuration file.

Preserving changes made to a configuration file isn't easy. There are many different configuration file formats, and sometimes a single configuration file will have multiple formats—depending on which version of the package you have installed. As such, there are two standard methods of dealing with configuration files in a Debian package: *dpkg*'s *conffile* handling and a *debconf*-based handler called *ucf* (which stands for “update configuration file”).



Dealing with configuration files is presented in detail in the “Installing Packages” section of this chapter.

Package repositories

Package repositories are large archives of packages that are stored, usually on the Internet, for users to pull from when they want to install or upgrade software on their system. Though obviously not a part of a package, *repositories* are nevertheless very important to the implementation of package management systems. Though some package management systems require users to operate directly on package files themselves, advanced systems keep databases describing the packages contained in a repository and can pull from them automatically. Not only does this allow the user to retrieve information with only the package name to work with, but it also avoids the tedious step of first finding a package file and then downloading it.

Debian's package repository contains approximately 15,000 binary packages for each architecture it supports. The database that describes the packages in this repository is copied to the user's machine on a regular basis by the user (in a process described later in this chapter). With the database stored locally, package searches and information retrieval are fast and convenient. The form that a Debian repository takes varies — for instance, installation CDs are partial repositories. However, most users prefer using Debian's Internet servers, which allow the user to download only those packages they're interested in (as opposed to either downloading all 13 CDs' worth of packages or purchasing and carrying around a small library of CDs).

Configuring which Debian repositories to use is as simple as editing `/etc/apt/sources.list` (type `man sources.list` for more information on how to do this) or removing `/etc/apt/sources.list` and then running `apt-setup`.

Package Management Tools

Getting into the meat of package management, this section simply provides an overview of the various package management tools in Debian. In addition to graphical front ends and package management systems, there are three main tools that can be used for different purposes at different times:

- ♦ **The apt family** — Probably the most famous part of Debian, `apt-get` is a command-line tool to install and remove packages. `apt-get` is part of the apt family of tools, where apt stands for “advanced package tool.” `apt-get`'s simplicity is unmatched — to install a package, you issue a command such as `apt-get install packagename`. Though it doesn't provide pretty buttons and lots of colorful icons, `apt-get` is easy to learn and simple to use. Its command-line nature also allows it to be used remotely over very low-bandwidth or unreliable connections.

`apt-get`, however, is just one part of the `apt` family. `apt-cache`, for example, is the `apt` family's command-line search and display tool. `apt-cache` shares the same advantages as `apt-get`, namely simplicity and ease of use. `apt-setup`, which was mentioned above, is yet another in the family of `apt` tools.

- ♦ `aptitude` — `aptitude` is a fullscreen console-based package management front end browser. It's based on the `apt` libraries, meaning that the functionality provided by the `apt` family of command-line tools is available within `aptitude`. `aptitude`'s strengths lie in its presentation of the data. With so many components in the distribution, it can be a bit difficult to find exactly what you want. Even examining your current system can be a chore. With `aptitude`, you can “drill down” through the various levels of the system, quickly collecting whatever data you need.

A special feature specific to `aptitude` — and the reason why I recommend its use — is its ability to track packages that were installed automatically. As an example, say you install package `foo`. During `foo`'s installation, the library `bar` was installed automatically to satisfy `foo`'s dependencies. When you remove package `foo`, library `bar` will be removed as well, so long as no other package still depends on it.



Tip

`aptitude` can actually be used on the command-line as well, by providing the same arguments you would give `apt-get` and `apt-cache`. So not only does `aptitude` provide a fullscreen package browser that can often aid in your searching and system examination, but it can also be used with the same simplicity as `apt-get` and `apt-cache` — with the added benefit of the tracking of automatically installed packages.

- ♦ `dpkg` — At the bottom of the pile is `dpkg`, Debian's basic package management tool. Whenever `apt-get` or `aptitude` install, remove, or otherwise operate on a package, they're calling `dpkg` to do the heavy lifting. `dpkg` provides a lower-level interface to the packaging system, allowing you to override particular behaviors. These low-level package operation interfaces should be used only when you know exactly what you need to do, however, because making a mistake with `dpkg` might require some painful hand-editing of package databases, which requires expert knowledge.

Finding and Examining Packages

One of the harder parts of dealing with a Debian system is finding the package you want from among the many available. Though finding packages can be difficult, it isn't the only sort of information-gathering you'll need to use routinely. You'll also need to list packages, list packaging details, determine which files belong to which packages, and even occasionally view information about packages available in a different Debian tree than the one you use. This section covers these sorts of tasks.

Listing installed packages

Listing installed packages is a pretty easy task, and you can use either `dpkg` or `aptitude` to do it.

Using `dpkg` to list packages

`dpkg` has a listing mode that doesn't affect the packaging system databases, so it's safe to use on a regular basis. It can be the most convenient way to quickly see whether a package is installed, as you can see with the following shell session:

```
user@hostname:~$ dpkg -l dpkg
Desired=Unknown/Install/Remove/Purge/Hold
| Status=Not/Installed/Config-files/Unpacked/Failed-config/Half-installed
|/ Err?=(none)/Hold/Reinst-required/X=both(Status,Err:uppercase=bad)
||/ Name          Version          Description
+++-----+-----+-----+
ii dpkg            1.9.21           Package maintenance system for Debian
user@hostname:~$
```

`dpkg`'s `-l` option turns on listing mode, and you can supply it with a list of arguments or none at all. If you provide no arguments, `dpkg` will list all the packages that have records in the database. If you provide it with a list of arguments, `dpkg` will interpret it as a list of package names. In our example, we asked `dpkg` to list its own information. `dpkg` also supports the use of wildcards, as discussed in Chapter 3, to allow you to easily list a set of related packages, as in this example:

```
user@hostname:~$ dpkg -l "base*"
Desired=Unknown/Install/Remove/Purge/Hold
| Status=Not/Installed/Config-files/Unpacked/Failed-config/Half-installed
|/ Err?=(none)/Hold/Reinst-required/X=both(Status,Err: uppercase=bad)
||/ Name          Version          Description
+++-----+-----+-----+
un base           <none>           (no description available)
ii base-config    1.33.18          Debian base configuration package
ii base-files     3.0.2            Debian base system miscellaneous files
ii base-passwd    3.4.1            Debian Base System Password/Group Files
user@hostname:~$
```

Note the double-quotes surrounding `base*`. Without the quotes, your shell will interpret the wildcards itself, which might produce unexpected results. The last three fields in the output, Name, Version, and Description, should be self-explanatory. The first field, however, could use a closer examination.

Those two letters in the first field indicate the state the package should be in, and the state the package is actually in. Even though Debian goes to great lengths to ensure easy package management, bugs do occasionally creep in and interrupt a package installation or removal halfway through. Likewise, if your computer were

to lose power during a package maintenance session, the database would be left in an inconsistent state. There's actually a third space there for another letter, which is the error indicator. See Table 4-2 for the meanings of the three status letters.

**Table 4-2
Status Indicators**

<i>Letter</i>	<i>Description</i>
First Character – Desired Status	
u	Unknown: For package names that <code>dpkg</code> has never had to install or remove, this character is displayed.
i	Install: The package is supposed to be installed.
r	Remove: The package is supposed to be removed.
p	Purge: The package is supposed to be purged (removed, and with all configuration files deleted as well).
h	Hold: Any packages that the administrator has put on hold won't be automatically upgraded or removed.
Second Character – Actual Status	
n	Not Installed: The package is not installed.
i	Installed: The package is installed.
c	Config-files: The package's configuration files remain.
u	Unpacked: The package has been unpacked, and its files have overwritten old ones, but the package's post-installation script has not been run.
f	Failed-config: The package's post-installation script ran, but it failed for some reason.
h	Half-installed: The package's installation was interrupted.
Third Character – Error Status	
	A single space in this position means there are no errors with the package.
H	Held: A package can also be put on hold by the packaging system; for instance, if its dependencies are broken (this often happens when the administrator forces a package's installation, ignoring the package's dependencies).
R	Reinst-required: If a package is in a particularly poor state, its reinstallation may be required, and this letter indicates that.
X	Both problems: When a package is both put on hold by the system <i>and</i> needs to be reinstalled, X is the character displayed.

`dpkg` can also display which files are part of a given package and search for which package owns a particular file. These actions are done using the `-L` and `-S` options, respectively. The following example illustrates the use of these options:

```
user@hostname:~$ dpkg -L base-passwd
/.
/usr
/usr/sbin
/usr/sbin/update-passwd
/usr/share
/usr/share/doc
/usr/share/doc/base-passwd
/usr/share/doc/base-passwd/README
/usr/share/doc/base-passwd/copyright
/usr/share/doc/base-passwd/changelog.gz
/usr/share/man
/usr/share/man/man8
/usr/share/man/man8/update-passwd.8.gz
/usr/share/base-passwd
/usr/share/base-passwd/group.master
/usr/share/base-passwd/passwd.master
user@hostname:~$ dpkg -S /usr/share/base-passwd
base-passwd: /usr/share/base-passwd
user@hostname:~$
```



`dpkg -S` and `dpkg -L` function only for packages that you have installed. See the subsection “Searching using the Debian package Web site” for information on how to look at packages in this manner when you don’t have them installed.

Using `aptitude` to list packages

As mentioned previously, one of `aptitude`’s great strengths is its ability to flexibly display information about your package database. The following instructions walk you step-by-step through an example description of using `aptitude`:

1. Open `aptitude` with this command:

```
user@hostname:~$ aptitude
```

Your screen should look similar to the one shown in Figure 4-1.

As you can see, `aptitude`’s main display is nicely organized.

2. By default, the bottom half of the screen is used to display the long descriptions of packages. Toggle that off by pressing `Shift+D`, giving you more real estate to work with in the top half.

```

Actions Undo Options Views Help
f10: Menu ?: Help q: Quit u: Update g: Download/Install/Remove Pkgs
aptitude 0.2.14
--- Installed Packages
--- Not Installed Packages
--- Virtual Packages
--- Tasks

These packages are currently installed on your computer.

```

Figure 4-1: aptitude's main display.

- Aside from the categories shown in Figure 4-1, at times others will appear, such as `New Packages` and `Upgradeable Packages`. We're primarily interested now in the packages that are currently installed, so use the up and down arrow keys to highlight that category, and press the Enter key. The category will expand into a number of subcategories. Each package in Debian is assigned a particular category, and this listing shows how the results are often displayed.
- Now highlight `Editors` and press Enter. The next (lower) category level is displayed; this is the license level, and it allows you to restrict your listing to a particular style of license. Packages in the `main` archive on this level are totally free for use, modification, and redistribution. Though you likely don't see any on your screen, packages in the `contrib` and `non-free` archives have other licensing or use restrictions that may require a careful license examination on your part.
- Press Enter to expand the `main` category. You should now see three packages: `ed`, `nano`, and `nvi`. These are the only text editors installed on the system by default. Examine an example line:

```
i      ed                0.20-20      0.20-20
```

The first few characters in the line are equivalent to `dpkg`'s listing mode, as is the second field (the package name). The third field indicates the currently installed version of the package, and the fourth field indicates the most recent available version of the package. (If an upgrade is available, for instance, the new version of the package is displayed here.)

- Highlight the package's line, and the bottom line of the screen will display the package's short description. You can press `Shift+D` to turn the extended description area back on and view the package's long description.

Don't quit `aptitude` yet because we'll examine the `ed` package in more detail later in this chapter.

Showing package details

Of course, performing a package operation — whether install, remove, purge, or what have you — may require that you understand what the package is and how it works within the system as a whole. You can use `aptitude` or `apt-cache` to see the package details.

Using `aptitude` to show package details

Examining the package system in detail is where `aptitude` can really shine. Any place where a package name is displayed by `aptitude`, you can press Enter to view that package's details. Using the example from the previous section, highlight `ed` and press Enter. You're now presented with a new screen detailing all sorts of information about the package. Wherever you can highlight a line that starts with three dashes, you can press Enter to expand the tree and get more information. Go ahead and expand the `Packages which depend on ed tree`. Use the down-arrow key to scroll down, and you can see all the packages that use `ed` in some way or another, as well as their status (again, status is indicated in the first field). At the bottom of this detailed package display, you can see all versions of the package that are available; sometimes the package has an upgrade available. You can press Enter on any of those to view that version's particular details.

Press the Q key to leave the package detail screen and return to the main screen. If you decided to explore a bit and view other packages' details, just keep pressing Q until you're back at the main screen. Then press Q one last time to quit, and confirm that you wish to quit.

Using `apt-cache` to show package details

Though I really love `aptitude`, it isn't suitable for every purpose. For instance, the fullscreen nature of the detail view can make it more difficult to use. It's also darned near impossible to use `aptitude`'s fullscreen display in a script to automatically retrieve information. A good alternative utility is `apt-cache`. Take a look at how `apt-cache` displays the `ed` package's details, using the `apt-cache show` command followed by the package name:

```
user@hostname:~$ apt-cache show ed
Package: ed
Priority: important
Section: editors
Installed-Size: 144
Maintainer: James Troup <james@nocrew.org>
Architecture: i386
Version: 0.2-20
```

```

Depends: libc6 (>= 2.3.1-1)
Filename: pool/main/e/ed/ed_0.2-20_i386.deb
Size: 44718
MD5sum: 0c466ce6a160c62fa558fbbb46a4ea45
Description: The classic unix line editor
ed is a line-oriented text editor. It is used to
create, display, modify and otherwise manipulate text
files.
.
red is a restricted ed: it can only edit files in the
current directory and cannot execute shell commands.

user@hostname:~$

```

The information `apt-cache` displays is much the same as what `aptitude` presented, but it allows for easier scripting.

Searching with `apt-cache`

Finding packages to install can be a tough job, but there are a few good options. Though `aptitude` is by far the most flexible, allowing for all sorts of search queries and narrowing-down of the results, `apt-cache` is definitely the easiest to use, especially for short queries. For searching, try `apt-cache`'s search option, as shown in the following example. The format is `apt-cache search one or more search terms`. This command will search the package names, short description, and package long descriptions and return a list of package names and short one-line descriptions for packages that contain the search term or terms. If you get too many results, try adding words to the search to narrow it down:

```

user@hostname:~$ apt-cache search debian goodies
debian-goodies - Small toolbox-style utilities for Debian systems
emacs-goodies-el - Miscellaneous add-ons for Emacs
python - An interactive high-level object-oriented language (default
version)
user@hostname:~$

```

When you find a package that looks interesting, you can use `apt-cache show` (as explained in the previous subsection) to display the details about a listed package, including its long description.

Searching with `aptitude`

`aptitude` has great browsing facilities and really flexible searching facilities. However, the cool stuff is a bit complex, so I'll discuss the simplest search method in `aptitude`. Go ahead and run it so that you're at the main screen. Type in a single forward slash followed by a single term, and then press Enter. Pressing the forward

slash key tells `aptitude` that you will be typing in a term to search for. This term will be searched for in all packages' names. `aptitude` will show your search's first hit. To repeat the search again and find subsequent hits, press the N key until you find what you're looking for. Here's that same process broken down into steps:

1. At the prompt, type **aptitude** to start the program:

```
user@hostname:~$ aptitude
```

The main `aptitude` window should now be displayed.

2. Once you are in `aptitude`, type a single forward slash (/) to bring up the search prompt. Now type the term you want to search for. Press Enter to start the search.
3. `aptitude` will jump to the first package it can find that matches or includes the word you have searched for. If this is not the package you are looking for, press N to go to the next match. Keep repeating until you find a satisfactory package.



Tip

For more information regarding `aptitude`'s searching and filtering capabilities, see `/usr/share/doc/aptitude/README`.

Searching using the Debian package Web site

All the tools described above work on the package lists stored locally on your computer. This means that they describe only packages that are available on your particular architecture for the Debian tree you're using (stable, testing, or sid). This is a lot of packages, but it will never be all of them. If you are using Debian stable, there may be new software that is uploaded only into testing or even sid. Occasionally you'd like to see what's available anywhere in Debian. <http://packages.debian.org/> is the Debian package Web site, and it's a great resource for these sorts of situations. The most useful sections of the site are the "Search package directories" and "Search the contents of packages" windows located toward the bottom of the page. Using these tools, you can enter a search term and search the complete list of packages in the Debian archive through the archive's Web interface.

"Search package directories" functions like `apt-cache search`. "Search the contents of packages" functions like `dpkg -S` and `dpkg -L`, but you don't need to have the package installed on your machine to get results. Additionally, after clicking on a package's name in the results page, you can see other information like the package's copyright license and its change log because this information is all provided at the bottom of the page.



Caution

You can — but shouldn't — download package files from packages.debian.org and install them manually with `dpkg`. This will not always work, though, and it can possibly break your system. `apt-get` and `aptitude` do a lot of consistency-checking behind the scenes. Use packages.debian.org to find the name of the package you want, and then install it with either `apt-get` or `aptitude`.

Installing Packages

Now that you know how to find packages, you likely want to install some. I'll use several example packages, one for each tool available. First, if you aren't logged in as `root` (and you shouldn't be), do so now. You can either switch to a different console with `Ctrl+Alt+F n` , where n is a number from 1 (the first console) through 6 (`Ctrl+Alt+F7` is reserved for X Windows). Alternatively, you can use the `su` command, as in the following:

```
user@hostname:~$ su
Password:
hostname:~#
```

`su` means “super user,” and it allows you to switch to the `root` account from a regular user account. The password you type is the `root` password, which you initially configured during the installation. Note the prompt change; the terminator is now a pound sign (`#`) instead of a dollar sign, indicating that you're logged in as `root`. Since you know you're logged in as `root` from the pound sign, the prompt doesn't include the username at the beginning.

Installing packages using apt-get

The simplest is usually the best way to start, so let's start with `apt-get`. While I recommend `aptitude` for regular use (due mainly to its ability to track automatically installed packages), you can't beat `apt-get` as a learning tool. First and foremost, before you install any new packages or upgrade old packages with `apt-get`, you need to run `apt-get update`, as shown in the following code example. This downloads the latest list of packages from Debian servers, and it ensures that you get the latest versions of packages available for the Debian tree you have installed:

```
hostname:~# apt-get update
Hit http://ftp.us.debian.org stable/main Packages
Hit http://ftp.us.debian.org stable/main Release
Hit http://ftp.us.debian.org stable/main Sources
Hit http://ftp.us.debian.org stable/main Release
Get:1 http://security.debian.org stable/updates/main Packages
[183kB]
Hit http://non-us.debian.org stable/non-US/main Packages
Hit http://non-us.debian.org stable/non-US/main Release
Hit http://non-us.debian.org stable/non-US/main Sources
Hit http://non-us.debian.org stable/non-US/main Release
Get:2 http://security.debian.org stable/updates/main Release
[110B]
Fetched 183kB in 2s (85.7kB/s)
Reading Package Lists... Done
Building Dependency Tree... Done
user@hostname:~$
```

**Tip**

The output you get won't be identical to the output here, but it should be close. If you get any errors, chances are your `/etc/apt/sources.list` is broken. If this is the case, go ahead and remove that file with `rm /etc/apt/sources.list`, and run `apt-setup` to generate a new one.

Having updated the package list, go ahead and install `vim`. `vim` is an editor, a `Vi` clone, but it has far more features:

```
hostname:~# apt-get install vim
Reading Package Lists... Done
Building Dependency Tree... Done
The following extra packages will be installed:
  libgpm1
The following NEW packages will be installed:
  libgpm1 vim
0 packages upgraded, 2 newly installed, 0 to remove and 0 not upgraded.
Need to get 0B/3796kB of archives. After unpacking 12.3MB will be used.
Do you want to continue? [Y/n] y
Get:1 http://ftp.us.debian.org stable/main libgpm1 1.19.6-12 [45.2kB]
Get:2 http://ftp.us.debian.org stable/main vim 6.1.018-1 [3751kB]
Selecting previously deselected package libgpm1.
(Reading database ... 6633 files and directories currently installed.)
Unpacking libgpm1 (from .../libgpm1_1.19.6-12_i386.deb) ...
Selecting previously deselected package vim.
Unpacking vim (from .../vim_6.1.018-1_i386.deb) ...
Setting up libgpm1 (1.19.6-12) ...

Setting up vim (6.1.018-1) ...

hostname:~#
```

Here you see Debian's dependency-resolution at play. `vim` depends on `libgpm1`, so that package was marked for installation automatically. `apt-get` wanted to be sure that the extra installation was acceptable, so it asked for confirmation. Then it went ahead and downloaded and installed the packages. Pretty simple, and that's it.

To reinstall a package, use `apt-get --reinstall install packagename`.

Installing packages using aptitude

Before using `aptitude`, you should run `aptitude update`, which has a similar purpose and effect as running `apt-get update`. Alternatively, you can press the `U` key when in `aptitude`'s normal interactive fullscreen mode.

`aptitude` can be used in the same way as `apt-get`, right on the command-line. In this example, you'll install `mutt`, a common console-based e-mail client. First, look at `mutt`'s package relationships more closely, using the `apt-cache show` command:

```
hostname:~# apt-cache show mutt | grep -E '^(Depends|Recommends)'
```

```
Depends: libc6 (>= 2.3.2.ds1-4), libidn11, libncursesw5
(>= 5.3.20030510-1), libsasl2 (>= 2.1.15), exim | mail-transport-agent
Recommends: locales, mime-support
hostname:~#
```

As you can see, `mutt` recommends both the `locales` and `mime-support` packages. `locales` is installed as part of the basic installation, but `mime-support` is not. Whereas `apt-get` installs only Depends packages, by default `aptitude` will also install recommended packages in the field “Recommends,” as shown in the following code sample:

```
hostname:~# aptitude install mutt
Reading Package Lists... Done
Building Dependency Tree
Reading extended state information... Done
The following NEW packages will be automatically installed:
  libidn11 libncursesw5 libsasl2 libsasl2-modules mime-support
The following NEW packages will be installed:
  libidn11 libncursesw5 libsasl2 libsasl2-modules mime-support mutt
0 packages upgraded, 6 newly installed, 0 to remove and 62 not upgraded.
Need to get 2185kB of archives. After unpacking 5788kB will be used.
Do you want to continue? [Y/n/?] y
Writing extended state information... Done
Get:1 http://http.us.debian.org sid/main libsasl2 2.1.18-2 [255kB]
Get:2 http://http.us.debian.org sid/main libidn11 0.4.1-1 [90.2kB]
Get:3 http://http.us.debian.org sid/main libncursesw5 5.4-3 [287kB]
Get:4 http://http.us.debian.org sid/main mime-support 3.26-1 [28.6kB]
Get:5 http://http.us.debian.org sid/main mutt 1.5.5.1-20040112+1 [1375kB]
Get:6 http://http.us.debian.org sid/main libsasl2-modules 2.1.18-2 [150kB]
Fetched 2185kB in 9s (230kB/s)
Selecting previously deselected package libsasl2.
(Reading database ... 17338 files and directories currently installed.)
Unpacking libsasl2 (from ../libsasl2_2.1.18-2_i386.deb) ...
Selecting previously deselected package libidn11.
Unpacking libidn11 (from ../libidn11_0.4.1-1_i386.deb) ...
Selecting previously deselected package libncursesw5.
Unpacking libncursesw5 (from ../libncursesw5_5.4-3_i386.deb) ...
Selecting previously deselected package mime-support.
Unpacking mime-support (from ../mime-support_3.26-1_all.deb) ...
Selecting previously deselected package mutt.
Unpacking mutt (from ../mutt_1.5.5.1-20040112+1_i386.deb) ...
Selecting previously deselected package libsasl2-modules.
Unpacking libsasl2-modules (from ../libsasl2-modules_2.1.18-2_i386.deb)
...
Setting up libsasl2 (2.1.18-2) ...

Setting up libidn11 (0.4.1-1) ...

Setting up libncursesw5 (5.4-3) ...
```

```

Setting up mime-support (3.26-1) ...
Setting up mutt (1.5.5.1-20040112+1) ...

Setting up libsasl2-modules (2.1.18-2) ...
Reading Package Lists... Done
Building Dependency Tree
Reading extended state information... Done
hostname:~#

```

Now `mutt` is installed. `aptitude` installed everything `mutt` really absolutely required for functioning (the Depends), and an additional package that `mutt`'s maintainer feels adds a good deal of value (the Recommends). The other recommended package, `locales`, and the package on which `mutt` depended, `libc6`, were already installed. As a result, neither of them was installed. Both of these packages were already installed because they are in the core Debian distribution and are always included in new installs. The output from the command should look very much like of `apt-get`'s output. Not only do `apt-get` and `aptitude` share a lot of code, but they also both end up calling `dpkg` to do the messy work of installation. `apt-get` and `aptitude` are responsible for selecting and downloading the packages to install (to satisfy the various package relationships), while `dpkg` does the heavy lifting.

Now go ahead and run `aptitude` without any arguments on the command-line to open up a fullscreen session. Press `/` to begin a search, type `elinks`, and press Enter. You should now have the `elinks` package highlighted. To mark the package for installation, type `+`. The color of the line will change to green, and in the far left column an `i` will appear; both of these are intended to indicate that the package will be installed when you initiate operations.

Speaking of initiating operations, here's a quick note about `aptitude` in fullscreen mode: You first tell it what you want to do, and then you press the `G` key, for "go." `aptitude` displays a staging screen, showing you what operations it will perform. Look over the list of changes and then press `G` at this staging screen to confirm its actions. The screen changes, and you can see that `aptitude` is going to install two extra packages—`liblua50` and `liblua50`—to satisfy `elinks`' dependencies. Press `G` a third and final time to begin the installation process.

When the downloads have completed, you'll be asked to confirm for the last time that you want to continue, at which point `aptitude` will call upon `dpkg` to perform the package installations.

To reinstall a package with `aptitude`, use either `aptitude reinstall packagename` or browse to the package in the fullscreen interface and press `L` to mark the package for reinstallation. It will be reinstalled when you next initiate packaging operations.

Removing and Purging Packages

In Debian, great pains are taken to ensure that configuration data is kept intact. When a package is simply removed, its configuration files and data are left in place in case you ever want to reinstall the package. To tell the packaging system to remove the configuration files too (so that the next time you install the package you get the default configuration), you need to purge the package.

Removing and purging packages using apt-get

Once again, `apt-get` is a quite simple way to solve this problem. Simply run `apt-get remove packagename`, and it will remove the package. Remove `vim`, as seen in the following session:

```
hostname:~# apt-get remove vim
Reading Package Lists... Done
Building Dependency Tree... Done
The following packages will be REMOVED:
  vim
0 packages upgraded, 0 newly installed, 1 to remove and 0 not upgraded.
Need to get 0B of archives. After unpacking 12.2MB will be freed.
Do you want to continue? [Y/n] y
(Reading database ... 7567 files and directories currently installed.)
Removing vim ...
dpkg - warning: while removing vim, directory `/etc/vim' not empty so not
removed.
hostname:~#
```

`apt-get` will always ask you before it removes a package. You'll note that `dpkg` issued a warning saying that it wasn't able to remove a directory—`/etc/vim/`—because it wasn't empty. That's not surprising, since you just removed the package instead of purging it—thus `vim`'s configuration files remain. To purge a package, pass `apt-get` the `--purge` option:

```
hostname:~# ls /etc/vim/
vimrc
hostname:~# apt-get --purge remove vim
Reading Package Lists... Done
Building Dependency Tree... Done
The following packages will be REMOVED:
  vim*
0 upgraded, 0 newly installed, 1 to remove and 61 not upgraded.
Need to get 0B of archives.
After unpacking 1700kB disk space will be freed.
Do you want to continue? [Y/n] y
(Reading database ... 17629 files and directories currently installed.)
Removing vim ...
```

```
Purging configuration files for vim ...
hostname:~# ls /etc/vim/
ls: /etc/vim/: No such file or directory
hostname:~#
```

`apt-get` again asked us for permission to perform this operation; it indicated it was purging the package by appending an asterisk to the package's name. The final `ls` command verifies that that directory is in fact now gone. Now that `vim` is purged, if you reinstall it, its default configuration file will be installed freshly again.

Note, however, that `apt-get` didn't remove `libgpgm1`, which was installed alongside `vim` to satisfy a dependency. If packages are installed as dependencies of another package, `apt-get` will not look to see whether they are still needed when you uninstall the original package and will simply leave them there. Now take a look at how `aptitude` deals with that scenario.

Removing and purging packages using `aptitude`

Let's get rid of `mutt` first, which we installed earlier via `aptitude`:

```
hostname:~# aptitude remove mutt
Reading Package Lists... Done
Building Dependency Tree
Reading extended state information... Done
The following packages are unused and will be REMOVED:
  libidn11 libncursesw5 libsasl2 libsasl2-modules mime-support
The following packages will be REMOVED:
  mutt
0 packages upgraded, 0 newly installed, 6 to remove and 61 not upgraded.
Need to get 0B of archives. After unpacking 5788kB will be freed.
Do you want to continue? [Y/n/?] y
Writing extended state information... Done
(Reading database ... 17519 files and directories currently installed.)
Removing mutt ...
Removing libidn11 ...
Removing libncursesw5 ...
Removing libsasl2-modules ...
Removing libsasl2 ...
Removing mime-support ...
Reading Package Lists... Done
Building Dependency Tree
Reading extended state information... Done
hostname:~#
```


Note

You can purge a package on the command-line with `aptitude` by running the command `aptitude purge packagename`. Note the difference between this and `apt-get`, which accepts `purge` as a double-dash option to remove.

As you can see, the packages that were installed alongside `mutt` were also removed when you asked for `mutt`'s removal. This is because you installed it with `aptitude`, removed it with `aptitude`, and `mutt` was the only installed package that declared dependencies on these packages. If another package had required, say, `mime-support`, the `mime-support` package would have been left intact. Unfortunately, `aptitude` can only know whether other packages need a given package. You may have, after installing a package automatically to satisfy a dependency, started using the package directly, so be careful and read `aptitude`'s output before telling it to go ahead with the operation. If you want to keep some of those automatically installed packages, one way to do it is to use `aptitude`'s fullscreen interactive interface.

So, as usual, fire up `aptitude`. Press `/` to start a search, type `e links`, and press `Enter`. `e links` should be highlighted. To mark a highlighted package for removal, press the minus/dash key (`-`). To purge it, type in an underscore (`_`). For this example, you want to purge `e links`, so type an underscore. You'll note that the line once again changes color, this time to purple. Press the `G` key to go to the summary screen, and you'll see that both `liblua50` and `liblua50` are also going to be uninstalled. Suppose you'd like to keep `liblua50` around. Highlight that package and press `+`, the same key you would use to mark a package for installation. The color of the line changes back to normal, and now `liblua50` won't be removed. If you want to mark packages as automatically or manually installed, highlight the packages and press `M` or `m`, respectively. Press the minus key to mark `liblua50` for removal again, and then press `G` to start the operation. When you're done, quit `aptitude`.

Configuring and Configuration Files

There are two forms of configuration on a Debian system: *application-specific* configuration and *debconf-based* configuration. Application-specific configuration typically means that the configuration for programs is stored in configuration files. Changing or customizing this type of configuration usually means editing a configuration file by hand and changing or adding variables and values. Debian attempts to make package installation painless, and most package maintainers avoid using interactive `debconf` prompts during a package install, but sometimes it's unavoidable. If a package can't provide a sensible default configuration file, it will instead ask some `debconf` questions. The advantage of `debconf` is that the user can decide what sorts of questions he'd like to see, and how to see them.

Configuring packages using `debconf`

`debconf` is the standard Debian configuration system. Many packages use `debconf` to provide a single interface to configuration that eliminates the need to have users edit files by instead providing sane defaults and a way to prompt users with questions. `debconf` questions are generally asked as part of the package's installation,

but you can also invoke them later on to reconfigure the package. This is done by running `dpkg-reconfigure packagename`:

```
hostname:~# dpkg-reconfigure debconf
```

Reconfiguring `debconf` allows you to set a variety of parameters used to display and process questions asked via packaged `debconf` scripts. The first question `debconf`'s own configuration script asks is “What interface should be used for configuring packages?” You can safely leave it at the default. After you've selected which interface you'd like to use, press Enter. You're then asked what priority of questions you'd like to see. Low-priority questions aren't displayed by default; they're usually only of interest to obsessive-compulsive knob-twiddlers like myself.

Note

I selected medium-priority questions, and you should as well. Unlike changing the `debconf` interface, changing which questions are asked will result in subtly different configuration files, which may break the instructions given in this book. When you're done with this book, or when you're familiar with the system as a whole, feel free to reconfigure `debconf` and choose whichever priority level you'd like.

Configuration file handling

Configuration files are always handled specially in Debian because they're far more important than most other software files. Configuration files are meant to be edited by the administrator, and as such may have hours of work put into them. Given that, and because your installation can break badly if a package blindly overwrites your carefully crafted configuration file, extra prompts are displayed whenever something damaging might happen.

Prompts are displayed whenever:

1. A new package is being installed for the first time, and there is a preexisting configuration file — perhaps copied from an older installation.
2. A package is being upgraded, but the administrator has manually changed a configuration file so that it's no longer the same one that was contained in the package.

Prompts are *not* displayed when:

1. The administrator has removed a configuration file; if you want to get a configuration file back from a package, you need to purge and reinstall it.
2. A new version of a configuration file exists in the new version of the package, and the administrator hasn't changed the current version.

Let's take a working example, and reinstall vim. If you haven't purged it already, do so now with `aptitude purge vim`. This will remove all its configuration files and tell the packaging system that when the package is installed later, it should restore the packaged configuration files. With vim no longer installed on the system in any way, create a fake `/etc/vim/vimrc` and see what happens when you reinstall it:

```
hostname:~# mkdir /etc/vim
hostname:~# echo Testing > /etc/vim/vimrc
hostname:~# cat /etc/vim/vimrc
Testing
hostname:~# aptitude install vim
Reading Package Lists... Done
Building Dependency Tree
Reading extended state information... Done
The following NEW packages will be installed:
  vim
0 packages upgraded, 1 newly installed, 0 to remove and 61 not upgraded.
Need to get 774kB of archives. After unpacking 1704kB will be used.
Writing extended state information... Done
Get:1 http://http.us.debian.org sid/main vim 1:6.2-426+1 [774kB]
Fetched 774kB in 11s (69.6kB/s)
Selecting previously deselected package vim.
(Reading database ... 17306 files and directories currently installed.)
Unpacking vim (from .../vim_1%3a6.2-426+1_i386.deb) ...
Setting up vim (6.2-426+1) ...

Configuration file `/etc/vim/vimrc'
==> File on system created by you or by a script.
==> File also in package provided by package maintainer.
  What would you like to do about it ? Your options are:
    Y or I  : install the package maintainer's version
    N or O  : keep your currently-installed version
    D       : show the differences between the versions
    Z       : background this process to examine the situation
  The default action is to keep your current version.
*** vimrc (Y/I/N/O/D/Z) [default=N] ?
```

As you can see, it's a somewhat ugly-looking prompt. It's simple and gets the job done, however. It tells us that there's a configuration file of the same name in the same directory in the package we're installing, and that tells us which configuration file it is talking about.

Your choices at this prompt are simple: Press the Y key and press Enter to have the packaged configuration file overwrite your own. Press the N key if you know that the file you have is good. You can press D to see the differences between the current file and the new file; you'll be returned to the prompt after you've finished reading the differences. Lastly, if you choose the Z option, you're dropped into a shell where you can manually poke about and perhaps merge the two configuration files by hand in a text editor. When you exit the shell, you're returned to the prompt.

Configuration File Handling with ucf

The information on configuration file handling in this section refers to regular configuration files, the kind that are included in the package normally and aren't generated at install time. In that simple case, `dpkg` itself is displaying the prompt and handling the configuration file. However, pretty much any package that uses `debconf` scripts also creates a default configuration file on the fly during the package's installation. As such, the handling is slightly different. `ucf` emulates the standard `dpkg` prompt in many ways, with the exception that it uses `debconf` for its user interface. The options will be the same, and the reasons are the same, although the display will look different.

Since the configuration file you created is fake, press `Y` and press `Enter` to use the packaged version. The installation will complete normally after that.

Upgrading Packages

As mentioned in the introduction of this book, `sarge` users won't be getting updates very often; it's a stable release and will only occasionally get a security update. However, when `sarge` does get an update, it's more than likely a very important fix. As such, it's important to go through the upgrade process at least once daily to check for upgrades. `sid` users will likely go through this upgrading process daily anyway, since `sid` receives updates on a continual basis.



If you use `sid`, see <http://people.debian.org/~dbharris/tracking-sid/> for some helpful tips on upgrading that are specific to `sid`. Whether you use `sid` or `sarge`, visit <http://people.debian.org/~dbharris/check-updates/> for details on getting your Debian system to automatically check for updates and e-mail you when any are available.

Upgrading using apt-get

As usual, before attempting to install or upgrade a package, run `apt-get update` to download the latest package lists from the Debian servers. After having done so, upgrading is as simple as running `apt-get upgrade`. `apt-get upgrade` won't, by default, allow any packages to be removed. If there's a case where a package needs to be removed, some packages will be *held back*. This is often the case where a package has been renamed; the procedure in that case is for `apt-get` to remove the old package and install the new one. This is typically a trouble-free process, performed automatically, but it's always safer to act conservatively. If you see a package being held back, you can instead use `apt-get dist-upgrade`, which will then allow `apt-get` to satisfy package relationships by removing packages. Carefully examine the output, however, to make sure that nothing you care about is being removed — or at least that nothing you care about is being removed without being replaced.

If there are a number of upgrades available but you want to upgrade only one or more specific packages, you can instead use `apt-get install` and supply it with a package or list of packages on the command-line. This will upgrade only those packages that you've specified, as well as any other upgrades that absolutely must take place to install the newer versions of the packages you've specified.

Upgrading using aptitude

In terms of upgrading, `aptitude` is used in the exact same way on the command-line as `apt-get`. However, as usual, `aptitude` has a fullscreen interface available as well. In the case of upgrades, this is particularly useful because it allows you to fine-tune which packages get upgraded quickly and easily. By default, `aptitude` will mark every package that has a newer available version than the one you've got installed, so simply running `aptitude` and then pressing the G key to go to the operation summary screen will show you what packages will be upgraded, added, or removed. If you wish to put a package on hold to prevent it from being upgraded, press the = (equal sign) key. This package will be displayed as being held back every time you go to the operation summary screen, and you can decide to upgrade it later; just highlight the package and press the + (plus sign) key.

Integrity-Checking Packages

To test the validity of your system, you may want to verify that the files on your machine match those that were contained in the original package. Almost every package in Debian includes cryptographic signatures for all the files it contains. Using the `debsums` tool, you can compare the files on your system against these cryptographic signatures.

Once you have `debsums` installed, run `debsums` to see whether any of the files on your system have been modified from the packaged version. Since you just did an installation, it's very unlikely. Typically, a file will have changed if you tried to install, from source, some of the same software you already had installed. Ignore any output that warns that some packages don't contain `md5sums`. It's just a warning, not an error.



The other reason for wanting to check your files' consistency is if you believe your machine's security has been compromised. There is no absolute way to ensure that your system hasn't been subverted. At best you can confirm that it *has* been broken into. Since the cryptographic signatures contained in Debian packages are stored on your hard drive after the package has been installed, an attacker could just as easily modify those signatures to match any modifications they might have made to the system files. In the event of a security breach, the *only* option is reinstalling from scratch and copying only nonexecutable files from the old installation. There are many reasons to do this, but I'll leave that to the next chapter, "Basic System Administration." Suffice it to say that if you care about such things, you'll keep your system up to date with respect to any security updates available and closely read all the chapters in this book relating to security.

Package Repositories and `/etc/apt/sources.list`

Although Debian's package repositories are very large and complete, they don't quite contain everything. The two primary reasons for using unofficial package repositories are to get patent-encumbered software, or to retrieve newer versions of packages than are available in `sarge`, without using the `sid` development tree.

Though specifics differ from repository to repository, the general idea is the same: add one or more lines to `/etc/apt/sources.list`. Since `/etc/apt/sources.list` is the file `apt-get update` and `aptitude update` read to determine which package lists to download, it's worth discussing the makeup of that file.

Each line describes a single repository and is made up of three parts: the binary/source keyword, the repository's URL, and the components within that repository. Standard lines look like the following:

```
deb http://http.us.debian.org/debian stable main
deb-src http://http.us.debian.org/debian stable main
```

The first word in the line is the binary/source indicator. `deb` means the repository contains binary packages you can install, and `deb-src` is for source packages. Most archives have both binary and source, so lines are often added in pairs. The second part of the line is simply the URL to the repository. Everything after the URL reflects the components of the repository. Though the components you want to add are repository specific, there needs to be *some* text there. The examples here refer to a standard Debian package repository for the stable distribution — `sarge`, in this case — and the main portion of that repository.

**Note**

Whenever you change anything in `/etc/apt/sources.list`, run `aptitude update` or `apt-get update`.

apt Pinning

The ability to add multiple sources to `/etc/apt/sources.list` makes it possible to seamlessly integrate unofficial repositories of Debian packages. These repositories may contain packages not found in Debian or newer or enhanced versions of official packages.

What happens if the same package exists in two or more repositories? (As far as the packaging system is concerned, it is the same package if it has the same name, regardless of the contents.) By default, `apt` will use the package with the higher version number. If two packages have the same version number, `apt` will select the one that best fits your current release (that is, `sarge`), which in most cases will be the official Debian package. If the packages have the same version number and belong to the same release, `apt` will pick the first one it comes across in its database, or in other words, essentially pick one at random.

Sometimes the default is not good enough. `apt`'s “pinning” feature lets you change it. Here are a few scenarios where you might want to change the way `apt` prioritizes upgrades.

Using selected packages from another release

Sometimes you need a newer version of a package on your stable system, but you don't want to move entirely to testing or unstable. Debian also has an experimental distribution for packages whose state is too raw even for unstable. You will want to be very careful about installing and upgrading such packages, too careful to trust the computer to do the right thing. `apt` pinning will help in both these and other situations.

`/etc/apt/sources.list`

Start by adding the `apt` sources for each release to `/etc/apt/sources.list`. A file that includes stable, testing, unstable, and experimental will look something like this:

```
# Debian sarge (stable)
deb http://security.debian.org sarge/updates main contrib non-free
deb http://http.us.debian.org/debian/ sarge main contrib non-free
# Debian etch (testing)
deb http://http.us.debian.org/debian/ etch main contrib non-free
# Debian sid (unstable)
deb http://http.us.debian.org/debian/ sid main contrib non-free
# Debian experimental
deb http://http.us.debian.org/debian/ experimental main contrib non-free
```

After adding these lines, run

```
# apt-get update
```

so `apt` can add all the packages in these sources to its database. You can now install packages from any of these distributions. But which will `apt-get` pick?

`apt-cache policy`

The command

```
$ apt-cache policy <packagename>
```

will tell you what it is going to do. For instance, as I write this, I get the following report for the KDE word processor `kword` on a system running `sarge` with the same `sources.list` as above:

```
$ apt-cache policy kword
kword:
  Installed: 1:1.3.2-1.sarge.1
```

```

Candidate: 1:1.3.4-1
Version Table:
 1:1.3.4-1 0
   500 http://http.us.debian.org sid/main Packages
   500 http://http.us.debian.org etch/main Packages
*** 1:1.3.2-1.sarge.1 0
   100 /var/lib/dpkg/status
 1:1.3.2-1.sarge.1 0
   500 http://http.us.debian.org sarge/main Packages

```

(The version numbers may be different by the time you read this, but the same principles apply.) What the output is telling you is that I currently have version 1:1.3.2-1 of `kword` installed. (`/var/lib/dpkg/status`, which is `dpkg`'s database, is treated as the source of the package.) `apt` has assigned this version a score of 100. Sarge currently has the same version, which has a score of 500, as does 1:1.3.4-1 in `etch/sid`. `kword` doesn't exist in `experimental`. As both the versions in `sarge` and `etch/sid` have the same score, the tie is broken by the higher version. The number after the version is the priority of the package, which is set only if you pin by version—see the “Installing (or keeping) a specific version” section—so it is 0 here.

`/etc/apt/preferences`

Were I to do an upgrade of `kword` now, the version from `testing/unstable` would be the candidate for installation. Because `kword 1:1.3.4-1` depends on other packages from `testing/unstable`, installing it could inadvertently result in a large part of my system being upgraded to `unstable`. So be very careful in using pinning with common library packages such as `libc6`, which large numbers of packages depend on. This can be prevented by creating a file called `/etc/apt/preferences` with contents like this:

```

Explanation: Sarge
Package: *
Pin: release a=stable
Pin-Priority: 999

```

```

Explanation: Etch
Package: *
Pin: release a=testing
Pin-Priority: 90

```

```

Explanation: Sid
Package: *
Pin: release a=unstable
Pin-Priority: 80

```

```

Package: *
Pin: release a=experimental
Pin-Priority: 10

```

Each stanza of this file consists of several fields, each on their own line. (Within a stanza, fields don't need to be in any particular order, however.) The first field, `Explanation`, can be used for comments, a short description of what you are trying to do with the pin. `Package` describes which packages are affected by this pin. The special value `*` means all packages, or you can give one or more package names here. The next field contains the criterion you want to use to pick packages to pin. In this case, the keyword `release` in the first half of the line means you want to sort packages based on the information in the `release` file that should accompany each package repository. (For more information on the subject, see Chapter 27.) The second half of the line specifies which field of the release file is to be used; `a` means archive. Unfortunately, `apt` doesn't support release code names, so you have to use the archive name (stable, testing, unstable) instead of `sarge`, `etch`, and `sid`. You can also use `c` for component (such as `main`, `contrib`, or `nonfree`), `v` for release version (such as 3.0 for Debian's woody release,) `o` for origin, or `l` for label. Other keywords you can use in the first half are `origin`, which is confusingly not the same as the `origin` field in the release file, but the Internet address of the package repository (such as `ftp.debian.org`), and `version`, which also is unconnected to the similarly named field in the release file and instead refers to a package version. This will be explained in greater detail further on. The last field is the score you want `apt` to give to this pin. By default, an installed package gets a score of 100, and any other sources get 500, unless a default release has been specified in `apt`'s configuration, in which case packages belonging to that release are assigned 990.

In the example preferences file, packages belonging to the stable release get the highest priority. Testing, unstable, and experimental are both less than 100, so assuming you have a package installed, you will not be prompted to upgrade it even if a newer version is available than what is in stable. If you run `apt-cache policy` again, you can see the difference pinning makes:

```
$ apt-cache policy kword
kword:
  Installed: 1:1.3.2-1.sarge.1
  Candidate: 1:1.3.2-1.sarge.1
  Version Table:
     1:1.3.4-1 0
        80 http://http.us.debian.org sid/main Packages
        90 http://http.us.debian.org etch/main Packages
*** 1:1.3.2-1.sarge.1 0
    100 /var/lib/dpkg/status
    1:1.3.2-1.sarge.1 0
    999 http://http.us.debian.org sarge/main Packages
```

Installing from multiple distributions

With pinning set up as shown previously, you are protected from accidental upgrades to a different release. But what if you want to install a different version of

keyword? You can temporarily override your pinning preferences with `apt-get's --target-release` option (synonyms are `-t` and `--default-release`):

```
# apt-get install -t testing kword
```

Another method, which is handy if you are installing packages from multiple distributions in the same install session, looks like this:

```
# apt-get install kdelibs4 kword/testing kdm
```

In this case, `kdelibs4` and `kdm` will come from stable, while only `kword` will come from testing.

These methods can be used to remove packages too.

Installing from a specific site

The `origin` variant of the `pin` field in `/etc/apt/preferences` can be used to ensure packages have a higher priority if they come from a particular place. Use a stanza like this:

```
Explanation: packages from my own site
Package: *
Pin: origin "src.braincells.com"
Pin-Priority: 999
```

Installing (or keeping) a specific version

Sometimes you might want to ensure that a package *doesn't* get upgraded. Maybe you have a local package that you've tweaked and don't want overwritten with a generic version. Or maybe you are relying on the functionality of one particular version of the package. Again, pinning can help. This time the `version` variant of the `pin` field is used. A specific package name is also used:

```
Explanation: bash 2.* only
Package: bash
Pin: version 2.*
Pin-Priority: 999
```

This allows upgrades of any new `2.*` versions of the `bash` shell, but it keeps you from any nasty surprises when `bash 3` comes out. (This is just an example. Usually in Debian, packages that are significantly different from their earlier versions are given different names, such as `bash 3`.) Note that the comparison of version numbers is done alphabetically, not numerically.

You can also override version preferences on the `apt-get` command line like this:

```
# apt-get install kword=1:1.3.4-1
```

And once again, this works for package removals too.



`apt` pinning is a very powerful tool that gives you a lot of control over the shape of your system, but it can be dangerous, too. The consensus among savvy Debian users is that it should be used very sparingly and certainly not for important system libraries and other core packages. If you absolutely must have a newer version of one of those and you just can't upgrade your entire system, you should try rebuilding the packages instead.

Summary

I covered a lot of material in this chapter, running the gamut from packaging concepts to using unofficial Debian package repositories. Having read this chapter, you should be perfectly capable of handling software management on your Debian installation, including such tasks as finding, examining, installing, removing, purging, configuring, and upgrading packages. You should also be familiar with the concept of pinning.



