



Published on **The O'Reilly Network** (<http://www.oreillynet.com/>)  
[http://www.oreillynet.com/pub/a/oreilly/opensource/news/cvs\\_0900.html](http://www.oreillynet.com/pub/a/oreilly/opensource/news/cvs_0900.html)  
[See this](#) if you're having trouble printing code examples

## Top 10 CVS Tips

by [Gregor N. Purdy](#)

09/11/2000

These are my favorite tips for using CVS. They cover topics ranging from repository setup to day-to-day usage to code distribution.

- 1. Get a quick view of the status of files.** The command `cvs -n -q upd` runs `cvs` in *quiet* mode (via the `-q` option), so notices are not displayed as it traverses the directory structure. Also, because the `-n` option is used, no files will actually be changed.

In the listing that results, you can see added files (A); files with conflicts (C); modified files (M); removed files (R); updated files (U or P); and files that have not been checked into the repository and are not being ignored (?).

The variant `cvs -n -q upd -AdP` makes the check against the head of the trunk of development (via `-A`), processes new directories from the repository (via `-d`), and prunes, or ignores empty directories (via `-P`).

- 2. Fix binary files accidentally checked in as text files.** The default keyword substitution mode is `kv` for keyword-value mode. This mode is useful because it allows certain keywords to expand with information about the file when the file is checked out. Keyword substitution is normally appropriate only for text files such as source code, because if a binary file happens to have data that matches one of the special keyword strings, CVS would perform keyword substitution, most likely corrupting the file.

---

In addition to writing the [CVS Pocket Reference](#), Gregor Purdy wrote the CVS material in chapter 14, "CVS and RCS," of [Linux in a Nutshell, 3rd Edition](#).

---

If you do a bulk import of a directory, some of the files may be binary files that may not be set up in the repository's `cvswrappers` file to default to `-kb` (binary) mode. If this is the case, then you'll have to go back and manually fix the "sticky options" for those files with the `cvs admin -kb` command, overlay the file with a fresh copy (in case the file was already corrupted), and commit it.

Here is an example showing the bulk import of a project followed by fixes for some binary files:

```
user@localhost$ cd ~/work/project
user@localhost$ cvs import project vendor start
user@localhost$ cd ..
```

```
user@localhost$ mv project project.bak
user@localhost$ cvs checkout project
user@localhost$ cd project
user@localhost$ cvs admin -kb *.png
user@localhost$ cp ../project.bak/*.png .
user@localhost$ cvs commit *.png
```

- 3. Hack the repository to remove files that should not have been imported.** Presuming you are the administrator of a repository, if someone does a bulk import of some code and it later turns out that many files were imported that should not be in the repository (such as derived files produced during a build process), you can remove the corresponding RCS files from the repository itself, instead of using the `cvs remove` command. If you only use `cvs remove`, then the repository will still contain the files and directories, even though they won't appear in sandboxes (so that the repository can fall back to the state of the initial import). You should not do this for a module that is in active use (one with sandboxes checked out).
- 4. Set up email notification of changes committed to the repository.** It can be helpful to send email notifications each time someone commits a file to the repository. Developers can monitor this stream of notices to determine when they should pull the latest development code into their private sandboxes. For example, consider a developer doing some preparatory work in his sandbox while he awaits stabilization and addition of another developer's new library. As soon as the new library is added and committed, email notification goes out, and the waiting developer sees the code is ready to use. So he runs `cvs upd -d` in the appropriate directory to pull in the new library code, and then sets about integrating it with his work.

It is simple to set up this kind of notification. Just add a line like this to the `CVSROOT/logininfo` file:

```
DEFAULT mail -s %s developers@company.com
```

Often, the email address is a mailing list, which has all the interested parties (developers or otherwise) on the distribution list. If you want to send messages to multiple email addresses, you can write a script to do that and have that script called via this file. Alternatively, you can use the `log.pl` program that comes as part of the CVS source distribution (located at `/usr/local/src/cvs-1.10.8/contrib/log.pl`, assuming CVS was unpacked into `/usr/local/src`). Instructions for its use are provided as comments in the file.

- 5. Import code snapshots for a project that has seen the release of a few versions before CVS comes into play.** If you maintain project history archives manually by taking periodic snapshots of the code, you can import the first snapshot, tag it with the date or version number, and then successively overlay the updated files from later archives. Each set can then be committed and tagged in order to bootstrap a repository that maintains the prior history. To do this, first unpack the distributions: (This assumes they unpack to directories containing the version numbers.)

```
user@localhost$ tar xvzf foo-1.0.tar.gz
user@localhost$ tar xvzf foo-1.1.tar.gz
user@localhost$ tar xvzf foo-2.0.tar.gz
```

Next, make a copy of the first version, import it into the CVS repository, check it out to make a sandbox (since importing doesn't convert the source directory into a sandbox), and use `cvs tag` to give it a symbolic name reflecting the project version:

```
user@localhost$ mkdir foo
user@localhost$ cp -R -p foo-1.0/* foo
```

```
user@localhost$ cd foo
user@localhost$ cvs import -m 'Imported version 1.0' foo vendor start
user@localhost$ cd ..
user@localhost$ mv foo foo.bak
user@localhost$ cvs checkout foo
user@localhost$ cd foo
user@localhost$ cvs tag foo-1_0
user@localhost$ cd ..
```

Next, apply the differences between Version 1.0 and 1.1 to the sandbox, commit the changes, and create a tag:

```
user@localhost$ diff -Naur foo-1.0 foo-1.1 | (cd foo; patch -Np1)
user@localhost$ cd foo
user@localhost$ cvs commit -m 'Imported version 1.1'
user@localhost$ cvs tag foo-1_1
user@localhost$ cd ..
```

You may need to handle the addition and deletion of files with the `cvs remove` and `cvs add` commands before doing the commit. You can use the `cvs -n -q upd` command to see which files were added, deleted, and modified by patch.

In the same way, apply the differences between Version 1.1 and 2.0 to the sandbox, commit the changes, and create a tag:

```
user@localhost$ diff -Naur foo-1.1 foo-2.0 | (cd foo; patch -Np1)
user@localhost$ cd foo
user@localhost$ cvs commit -m 'Imported version 2.0'
user@localhost$ cvs tag foo-2_0
```

Now you can use the log to view the history of the files, browse past versions of the files, and continue development under version control.

- 6. Use shell aliases to access multiple repositories with different remote shell (`$CVS_RSH`) settings.** Since there is only one way to specify the remote shell program, using `$CVS_RSH`, and this setting is global, users who commonly access multiple repositories need to pay close attention to which repository they are using. If one repository requires one setting of this variable and another requires a different setting, this variable needs to be changed between accesses to these repositories. This aspect of the repository access method is not stored in the `CVS/Root` file in the sandbox. For example, if you access some repositories via `rsh` and some via `ssh`, you can create these two utility aliases (bash syntax):

```
user@localhost$ alias cvs="export CVS_RSH=ssh; cvs"
user@localhost$ alias cvr="export CVS_RSH=rsh; cvs"
```

- 7. Ignore whitespace-only changes when doing `cvs diff`.** For many files, whitespace-only changes aren't significant except in special cases (such as when spacing changes are made to report generation code). If developers are using different text editors or the same editor with different settings, it is easy to end up with differences between file revisions that are a combination of substantive alterations and unimportant whitespace changes. If you are having trouble deciphering the output of `cvs diff` due to this situation, you can use the `-b` (ignore-space-change) option so the command will not report whitespace-only differences. The `-w` (ignore-all-space) and `-B` (ignore-blank-lines) options are also useful.

- 8. Use `cvs export` to create source code distributions.** If you use `tar` to package up a sandbox, files will be included that should probably be omitted, such as derived files and CVS directories and their contents. To avoid this, do the following (assuming a symbolic tag `'foo-1_0'` has been created):

```
user@localhost$ cvs export -r foo-1_0 -d foo-1.0 foo
user@localhost$ tar czf foo-1.0.tar.gz foo-1.0
```

- 9. Don't use periods in symbolic tags.** If you are using the `cvs tag` command, the symbolic tag must start with a letter and consist entirely of letters, numbers, dashes (-), and underscores (\_). Therefore, while you might want to tag your *hello project* with 1.0 when you release Version 1.0, you should tag it with something like `hello-1_0`, instead.

- 10. Use an interim shared sandbox (cautiously and only if necessary) to bring a project under version control.** Over time, projects may develop unintended environmental dependencies, especially when there is no pressure for the code to be relocatable. A project developed outside version control may even be initially developed in-place (at its intended installation location). While these practices are not recommended, they do occur in the real world. CVS can help to improve the situation by encouraging relocatability from the beginning of a project.

The default mode of operation for CVS is multiple independent sandboxes, all coordinated with a central shared repository. Code that runs in this environment is necessarily (at least partially) relocatable. So, using CVS from the beginning of a project helps ensure flexibility.

However, if a project is already underway, an interim approach can be used. For example, you can convert the development area to a single shared sandbox by importing the code into CVS and checking it back out again:

```
user@localhost$ cd /usr/local/bar
user@localhost$ cvs import bar vendor start
user@localhost$ cd ..
user@localhost$ mv bar bar.bak
user@localhost$ cvs checkout bar
```

It is a good idea to have version control in place before making flexibility enhancements, because it makes it easier to find (and possibly reverse) changes that cause trouble.

The repository locator is specified via the `-d` option or the `$CVSROOT` environment variable. It is stored in the various sandbox CVS/root files. If you use the Password Server (pserver), the user ID of the person checking out the sandbox is retained. If more than one person is working with a particular sandbox, they have to share an account for CVS access.

One way to do this is to have a neutral user account with a password known by everyone who has CVS access. Anyone can then issue the `cvs login` command with the same user ID and password, and access the repository. Once you stop using a shared sandbox, you won't need this workaround. However, if you are using a shared sandbox, it's important that any developers type their real user IDs into their log messages, otherwise all the changes will appear to be made by the common user.