

About CVS

CVS stands for Concurrent Version Control. It's free, open-source software used by multiple developers to share code, keep track of changes, and keep different versions of a project. It can be (and is) used for large, wide-scale projects, such as the Gnome interface. You'll be using it in your projects to ease the problem of many people working on the same codebase.

1 Version Control - what is it? why is it useful?

Allows you to keep a 'master' source tree and have each user check out their own private version of the tree. This allows for:

1. Parallel development.
2. A way to 'roll back' changes.
3. A way to integrate changes from different developers.
4. A way to keep several versions of a codebase on hand.
5. Automated annotations of all changes.

I've placed a directory on nexus in

```
~/brooks/public_html/examples/cvs-code
```

that you can use throughout this tutorial. You should treat this as an existing codebase that you're going to move into CVS, check out into a shadow tree, modify, and check back in.

2 Step 1: set up a repository

Create a directory where the 'master' version of your code will reside. This will need to be readable and writable by everyone on the project. If you're using UNIX and have the appropriate privileges, you might consider making a separate group for access.

You'll need to set an environment variable to tell CVS where your master directory - you can do this with:

```
setenv CVSROOT /path/to/my/master/directory (for tcsh users)
```

or

```
CVSROOT=/path/to/my/master/directory
export CVSROOT
```

 (for bash users)

Then do:

```
cvs init
```

to set up the appropriate subdirectories under \$CVSROOT.

You'll probably have a hierarchy of files somewhere that you want to add in to the hierarchy. They could be either third-party software (e.g. java libraries) or code you've developed yourself. To add these to your repository, do:

```
cvs import <directory> <branchname> <version-name>
```

For example, `cvs import ./c++code myCode start` takes the code in the directory `./c++code` and adds it to the repository.

If you look at the working directory that's been created, you'll see a directory called CVS. This is where CVS does its bookkeeping - keeping track of the CVSROOT variable, what's been checked out, and so on. You shouldn't need to make any changes to these.

3 Step 2: Check out the code you're interested in.

The top directory in the code hierarchy is called 'c++code'. To initially create a shadow tree in your own home directory, `cd` to wherever you want the shadow tree to be and type:

```
cvs checkout c++code
```

This creates the directory 'myProject' and populates it with all the source files. If you just wanted to check out a subset of the tree, you could do something like:

```
cvs checkout 'c++code/c++code/schedules'
```

Now, go ahead and make some changes to the code in your local version of the tree.

4 Step 3: Commit changes.

Once you've made your changes, tested them, and you think they're stable enough to save permanently and share with your group, you want to check them back in so that other people can get them. To do this, if you edited `Linear.C` in `multi-producer/schedules`, you would type:

```
cvs commit Linear.C
```

(to commit everything in your current directory and its subtree, do `cvs commit`)

at this point, an editor opens (which one depends on the value of `$CVSEDITOR` or `$EDITOR`) and you can add a comment describing the changes/additions that you made. Note: this is not optional! I know commenting is tedious, but it's the best way to inform your team members about changes and figure out why something's broken a week later when you're digging through past revisions.

5 Step 4: add new code to the repository

If you have a new file that you want to add to the repository, you can do this by editing it in your working directory, and then adding it to the repository with:

```
cvs add <filename>
```

This schedules the new file for adding. It won't be officially entered into the repository until you commit it.

6 Step 5: Get the latest version of the code.

As you're working, you'll want to get the changes that your partners have committed to the repository. You can do this by typing `cvs update` in your working directory. This will replace all files in the working directory and its subtree with their latest versions (assuming they've changed.)

Note: Once you've created a working directory, be sure you use 'update' and not 'checkout' to get the latest versions of the files. 'checkout' will wipe out all the changes you haven't committed yet!

7 Step 6: Remove obsolete files from the repository.

If you have files in the repository that you don't need anymore, you can delete them from the repository by: 1) removing them from your working directory:
`rm foo.c`

2) removing them from CVS: `cvs remove foo.C`

3) committing the change: `cvs commit`.

As always, nothing happens to the repository until you commit.

If you want to completely get rid of a working directory, you can do: `cvs release [module]`. This will remove your working directory and note in the log that you've released it.

8 Step 7: Comparing changes

You may want to know how your version of a file compares to the one in the repository. (What did I change in here, anyway?) cvs has a diff tool, just like regular UNIX:

`cvs diff Linear.C` will compare the version of `Linear.C` in your working directory to the one in the code repository.

To see who made what changes, do: `cvs log Linear.C` This will print out a summary of the change log (which everyone commented in. Right? Right!)

To see the status of a particular file (the version, whether your working version is up-to-date, when it was last checked in, etc.) you can do: `cvs status filename.i`. This is nice if you just want to see whether you're up-to-date without having a merge move everything around on you. (see below)

9 Step 8: Merging

Suppose you're editing a file, and you go to update, and the version in the repository has changed. (Your partner has also edited the file.) If the two can't be merged, CVS will give you a warning, save your original file as

```
.#filename.C.versionnumber (e.g. .#Linear.C.1.4)
```

and mark the changes in the new file, pointing out both your version and the repository version with

```
<<<</>>>>.
```

CVS will not let you check this file back in (commit it) until you've resolved these discrepancies. (by hand - it's not *that* smart.)

There are lots of other things you can do with cvs, but these are the basics. For example, you can tag certain versions (alpha/beta releases) you can place the repository on the network, you can read and write-lock files, you can have CVS notify users when a file is checked in, and lots of other stuff. Most of the time, you'll use the commands below.

- checkout (co)
- update
- add
- remove
- commit
- diff
- export

- release
- status

There's plenty of information about CVS out there. Two good sources are the man page and the following URL:

<http://www.cvshome.org/docs/manual>

<http://cvsbook.red-bean.com/>

is an online book about CVS and opensource development. This has some nice examples (an essential with CVS!)

The GNU manual for CVS is at:

http://www.gnu.org/manual/cvs/html_chapter/cvs_toc.html

Emacs also has tools called PCL-CVS and VC that serve as a frontend for CVS, if you're an emacs fan.