Computer Science 62

Terminal Windows, Emacs, Subversion and Make

or, "Out of Eclipse and into the blinding glare of the command line . . . "

This reference guide gives you a brief and pragmatic introduction to a few widely-used programs and utilities. The descriptions here are merely to get you started; they are by no means complete. Long books have been written about each of these programs. See the course resources page for pointers.

Terminal window. The terminal window permits keystroke-based communication with the computer. You already have a bit of experience with it. Here is a quick summary of the most common ideas and commands.

Directory structure. The directory (or folder) structure is hierarchical. The top-level, called the root, is `/`. Your home directory is something like `/home/csagehen` and is abbreviated with the symbol `~`. Your directory for this class is probably `~/cs062`.

Working directory. Each terminal window has a default directory, called the working directory. When you type a file name like `MyClass.java`, you are referring to the file with that name in the current working directory. A single period is an abbreviation for the current directory, and two consecutive periods denotes the parent of the current directory.

File specifications. As just mentioned, files in the current working directory may be specified by name. Other files may be specified by giving the full path, a path relative to the home directory, or a path relative to the current directory:

```
/common/cs/cs062/dropbox/
~/cs062/workspace/Assignment5/Calculator.java
../bin/Calculator.class
```

One can specify multiple files with the wildcard character `*`. For example,

```
~/cs062/workspace/*/src/*.java
```

corresponds to a list of all the Java files in your Eclipse workspace.

Common commands. Each command consists of a program, followed possibly by some options, and then a list of files on which to operate. Here are a few.

`pwd`	Print the current working directory.
`cd` ⟨new-directory⟩	Change the working directory to the one specified.
`ls` ⟨files⟩	List the files specified.
`ls` ⟨directory⟩	List the files in the named directory. An empty directory specification means the current directory.
`cp` ⟨source⟩ ⟨destination⟩	Copy one file to another.
`cp` ⟨files⟩ ⟨destination-directory⟩	Copy one or more files to another directory.
`mv` ⟨old-name⟩ ⟨new-name⟩	Rename and/or move a file.
`mv` ⟨files⟩ ⟨destination-directory⟩	Move files to another directory, preserving the names.
`mkdir` ⟨directory⟩	Create a directory with the specified name.
`rm` ⟨files⟩	Delete the specified files.
`rm` -rf ⟨directory⟩	Delete the specified directory, its contents, and (recursively) all the subdirectories.
`cat` ⟨file⟩	Print the contents of a file to the terminal window. (Only useful for short files.)
`javac` ⟨files⟩	Compile the specified Java files.
`java` ⟨class⟩	Run the `main` method in the specified class.
`./programname`	Run a program that appears in the current directory.

The difference between "copy" and "move" is that copying preserves the original file. Be as careful with `cp` and `mv` as you would be with `rm`: If the target of a copy or a move is an existing file, that file will be replaced—and there is no getting it back!

The Java commands are inconsistent. With `javac`, you *must* specify the extension `.java`. With `java`, you cannot specify the extension `.class`.

A very handy feature of most terminal windows, including the ones we use, is command-line editing. You can recall previously-typed commands with the up-arrow key, and you can edit such lines with the left- and right-arrow keys, the backspace key, and the delete key.

Emacs. For some, Emacs is a complete computing environment, a paradise from which one need never emerge. For others, Emacs is an incomprehensible morass from which one *can* never emerge. For us, Emacs is simply a text editor.

The version of Emacs on the Macintosh computers in the laboratory is Aquamacs. You can start it by clicking on the gnu icon in the toolbar. There are buttons at the top of the window for opening and saving files and for exiting the program.

From 2008, here are Charles Zhou's ideas of fun: Figure out how to change Emacs's color schemes for different file types. Learn how to run programs from inside of Emacs. Explore the many useless features like ELIZA.

On other systems, you may have to invoke Emacs from the command line by typing `emacs` at the command line. Depending on the configuration, you will either get a new window or else the editor will open in the terminal window. You may not have all the buttons, and you may have to use keystroke commands. A few appear below; there are more on the quick reference card linked from the course resources page.

ctrl-X s	save a file
ctrl-X c	exit Emacs
ctrl-X f	open a file
ctrl-minus	undo
ctrl-S	search forward
ctrl-R	search backwards
esc %	replace text
ctrl-K	delete to the end of a line
ctrl-space	begin a selection
ctrl-W	complete the selection and delete it
ctrl-Y	insert the most recently deleted text

Subversion. Subversion is a version control system, also known as a software configuration management system. The idea is to maintain a collection of "snapshots" of a project. The basic cycle is to check out a copy of the project, make changes, and then post (commit) the project files back as a new version. The value to you as a solo programmer is that you can recover from a disastrous mistake by reverting back to an earlier version of the project. Subversion is even more valuable when there is a team of programmers because it keeps one programmer from interfering with the work of another. (Imagine your state of mind if you and another programmer both check out a project. You work for hours and check it back in. Minutes later, your partner, who has done almost nothing, checks in the other copy and overwrites all your hard work. Subversion prevents that kind of loss.)

Erik Kuefler's comment from 2008: "On the few occasions when I thought a project would be easy and chose not to use Subversion, I ended up regretting the decision."

Let us begin with some terminology. You have a *project directory* in which you work. Separately, Subversion maintains a record of this directory, called the *project repository.* Normally, the Subversion repository is located in shared space where all the programmers have access to it. It might even be on another computer far away. A repository is named with URL; in our case it will be a file specification with a full path, like this:

```
file:///home/dkauchak/cs062/repos/projectA
```

When you type the name of your repository, be sure that you substitute your own userid and the actual name of your project directory. (If we had a class project, the URL might be something like `svn+ssh://vpn.cs.pomona.edu/usr/local/svn/cs062` or `https://svn.cs.pomona.edu/cs062`. These more sophisticated uses of Subversion allows us to share repositories among programmers across the network.)

First, create the repository. For our class, each of you will have one directory in which you store the repositories. It is probably best to put it in your `cs062` directory. Change to your `cs062` directory and create a new directory `repos` there.

```
cd cs062
mkdir repos
```

The directory you just created, `repos` can contain several different projects. Make an entry in the `repos` directory with the following command. (Do not type the angle brackets; they are used to indicate places where you are to substitute a name in the context in which you are working.)

```
svnadmin create repos/<your project's name>
```

This command must be given once (and only once) for each project that you want Subversion to track. Do not worry about what files are in the `repos` directory; Subversion will manage them. Your concern is with the files in the project directory.

Second, create the working project directory. It makes sense to give the project directory the same name as the Subversion project. Create your working directory, if it does not already exist, and navigate to it. In the working directory, give the command below. (The period at the end is not punctuation; it stands for the current directory.)

```
svn checkout <path to repos directory>/repos/<your project's name> .
```

As mentioned earlier the path must be a full URL, for example,

```
svn checkout file:///home/rbull/cs062/repos/projectA .
```

Third, add files to the svn repository. You now have a version-controlled project, but no files are being tracked. You must *add* files to a svn repository to tell it to keep track of them. Just because their in your project directory does not mean that they are in the svn repository. If you already have files in the project directory, ask Subversion to track them:

```
svn add <filenames>
```

In many cases, you can use the wildcard `*` to signify all the files in the directory. Similarly, whenever you create a new file (like a new .java file) add it to the repository. It's generally good practice to only keep track of files that you actually create and edit (e.g. .java files or .cpp files) and not compiled files (like .class or .o files).

The basic cycle for Subversion is (1) to "check out" the project and put the files into a clean directory, (2) to work on it, and (3) to "commit" your work by having Subversion record the next version. The cycle assumes that there are many programmers, and each time you start work, you will want new, up-to-date copies of the files. If you are the only programmer, and you are sure that the files are as you last left them, there is no need to check them out again. In fact, you will see an error message if you try to check out a project and some of the files already exist.

Finally, use Subversion to your advantage. Subversion will track the files that you request. If you create a new file and want Subversion to track it, just invoke Subversion with the `add` command.

```
svn add <newfile>
```

Do not move, rename, or delete files directly. Instead, use the Subversion equivalents, like `svn mv`, `svn rename`, and `svn delete`. Type `svn help` to get a list of all the commands.

The status command is often helpful and always reassuring. (At least it is reassuring when you remember that no response means that Subversion has nothing to complain about.) Just type

```
svn status
```

In case of emergency, you can discard your work on one file. The `revert` command will restore a file to its state when it was last committed. See the Subversion documentation if you need to go back further or replace several files.

```
svn revert filename
```

Similarly, you can use the `revert` command to recover a file that you accidently deleted.

Whenever you finished work, check the files back in using the `commit` command. Do not forget this step!

```
svn commit -m "Tired ... going to sleep"
```

(a message is always required when you commit changes and the -m flag is the easiest way to add your message)

Make. Make is a command-line program that automates the process of compiling files and building programs. You may not think you need it now, but it will become indispensable as your work becomes more complex.

Typing `make <target>` at the command line causes Make to create or update the file named as `<target>`. There are many options that you can give, but we will not worry about them now.

The actions of Make are controlled by a file named `Makefile`. When we get started with C++, we will give you a `Makefile`. Later, you will write your own. The simple `Makefile` below tells how to build `myprogram` from `main.o`, `part1.o`, and `part2.o`. It then goes on to tell how to build each of the three components. An invocation of Make will build the components recursively if they do not already exist. Also, Make will re-build a file if any of its components have changed since the last time it was built.

```
# A simple Makefile
#
# Rett Bull
# March 19, 2008

.PHONY: clean

myprogram: main.o part1.o part2.o
      g++ -o myprogram main.o part1.o part2.o

part1.o: part1.cpp part1.h header.h
      g++ -O -c part1.cpp

part2.o: part2.cpp header.h
      g++ -O -c part2.cpp

main.o: main.cpp header.h
      g++ -O -c main.cpp

clean:
      rm -f myprogram main.o part1.o part2.o
```

The character `#` is the comment character; a comment lasts to the end of the line. Like all programming files, a `Makefile` always has a comment at the top that includes the purpose, author, and date.

The heart of a `Makefile` is of a sequence of *rules* of the following form.

```
target: prerequisites
        instructions
```

An important (and invisible) detail: Each instruction in the list *must* be preceded by a `tab` character, not spaces!

Usually, targets are files, but a few are just words that act as placeholders—like `clean` in our sample. The `.PHONY` directive tells Make not to look for a file named `clean`.

You can specify a target on the command line by typing something like this:

```
make main.o
```

If you type simply `make`, the program will build the first target in the Makefile. For that reason, programmers often define a target `all` or `default` at the beginning of a `Makefile`.

A variant of our sample `Makefile`, which illustrates the use of variables, appears below.

```
# An almost-simple Makefile
#
# Rett Bull
# March 19, 2008

CXX = g++
CXXFLAGS = -O

OBJS = main.o part1.o part2.o

.PHONY: clean default

default: myprogram

myprogram: $(OBJS)
      $(CXX) -o $@ $(CXXFLAGS) $(OBJS)

part1.o: part1.cpp part1.h header.h
      $(CXX) $(CXXFLAGS) -c $<

part2.o: part2.cpp header.h
      $(CXX) $(CXXFLAGS) -c $<

main.o: main.cpp header.h
      $(CXX) $(CXXFLAGS) -c $<

clean:
      rm -f myprogram $(OBJS)
```

The variables like `CXX` and `OBJS` are simply abbreviations for the longer expressions to the right of the equals sign; they are used just as constants are in programs. The symbol `$@` is an abbreviation for the current target, and `$<` stands for the first prerequisite.

Make tries very hard to “do the right thing.” Sometimes it appears to be reading our minds, and at other times it can be mysterious or downright stubborn. One simple example occurs when we ask it to make a file `secondary.o`. If no rule exists in the `Makefile`, Make will look for a file `secondary.c` and run the C compiler on it. Failing that, it will look for a file `secondary.cpp` and run the C++ compiler. If even that fails, it will look for files in FORTRAN or other languages. There is no real magic here; Make is simply using an extensive set of built-in rules.

As you can see, the `Makefile` is complicated and rule specifications can easily get out of hand. The best advice for now is to keep things simple and use only the basic features of Make.