Unix Programming Tools

By Parlante, Zelenski, and many others

Introduction

This article explains the overall edit-compile-link-debug programming cycle and introduces several common Unix programming tools -- gcc, make, gdb, emacs, and the Unix shell. The goal is to describe the major features and typcial uses of the tools and show how they fit together with enough detail for simple projects. We've used a version of this article at Stanford to help students get started with Unix.

Contents

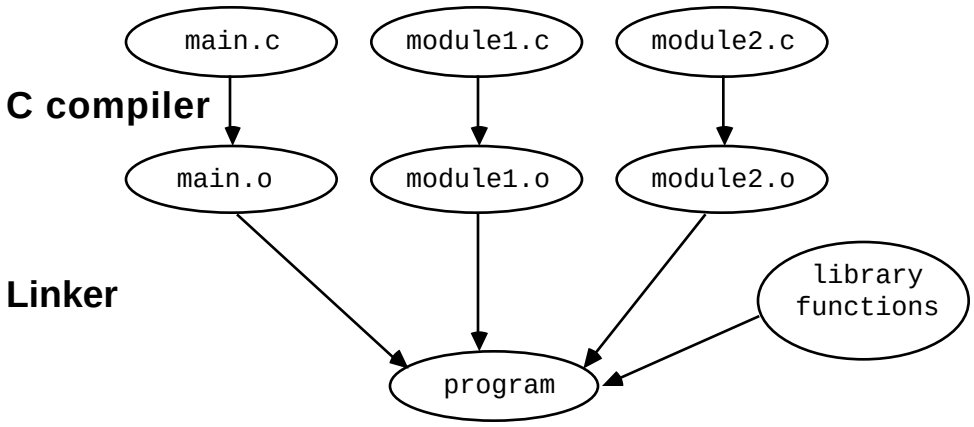
Other Resources

This article is an introduction — for more detailed information about a particular tool, see the tool's `man` pages and `xinfo` entries. Also, O'Reilly & Associates publishes a pretty good set of references for many Unix related tools (the books with animal pictures on the cover). For basic coverage of the C programming language, see CS Education Library #101, (http://cslibrary.stanford.edu/101/).

The Compile Process

Before going into detail about the individual tools themselves, it is useful to review the overall process that goes into building an executable program. After the source text files have been edited, there are two steps in the build process: *compiling* and *linking*. Each source file (foo.c) is compiled into an object file (foo.o). Each object file contain a system dependent, compiled representation of the program as described in its source file. Typically the file name of an object module is the same as the source file that produced it, but with a ".o" file extension — "main.c" is compiled to produce "main.o". The .o file will include references, known as symbols, to functions, variables, etc. that the code needs. The individual object files are then linked together to produce a single executable file which the system loader can use when the program is actually run. The link step will

also bring in library object files that contain the definitions of library functions like `printf()` and `malloc()`. The overall process looks like this...



Section 1 — gcc

The following discussion is about the gcc compiler, a product of the open-source GNU project (www.gnu.org). Using gcc has several advantages— it tends to be pretty up-to-date and reliable, it's available on a variety of platforms, and of course it's free and open-source. Gcc can compile C, C++, and objective-C. Gcc is actually both a compiler and a linker. For simple problems, a single call to gcc will perform the entire compile-link operation. For example, for small projects you might use a command like the following which compiles and links together three .c files to create an executable named "program".

```
gcc main.c module1.c module2.c -o program
```

The above line equivalently could be re-written to separate out the three compilation steps of the .c files followed by one link step to build the program.

```
gcc -c main.c                                ## Each of these compiles a .c
gcc -c module1.c
gcc -c module2.c
gcc main.o module1.o module2.o -o program       ## This line links the .o's
                                                ## to build the program
```

The general form for invoking gcc is...

```
gcc options files
```

where *`options`* is a list of command flags that control how the compiler works, and *`files`* is a list of files that gcc reads or writes depending on the options

Command-line options

Like most Unix programs, gcc supports many command-line options to control its operation. They are all documented in its man page. We can safely ignore most of these options, and concentrate on the most commonly used ones: `-c, -o, -g, -Wall, -I, -L, and -l`.

`-c files` Direct gcc to compile the source files into an object files without going through the linking stage. Makefiles (below) use this option to compile files one at a time.

`-o file` Specifies that gcc's output should be named *file*. If this option is not specified, then the default name used depends on the context...(a) if compiling a source .c file, the output object file will be named with the same name but with a .o extension. Alternately, (b) if linking to create an executable, the output file will be named `a.out`. Most often, the -o option is used to specify the output filename when linking an executable, while for compiling, people just let the default .c/.o naming take over.

It's a memorable error if your -o option gets switched around in the command line so it accidentally comes before a source file like "`...-o foo.c program`" -- this can overwrite your source file -- bye bye source file!

`-g` Directs the compiler to include extra debugging information in its output. We recommend that you always compile your source with this option set, since we encourage you to gain proficiency using the debugger such as gdb (below).

Note -- the debugging information generated is for gdb, and could possibly cause problems with other debuggers such as dbx.

`-Wall` Give warnings about possible errors in the source code. The issues noticed by -Wall are not errors exactly, they are constructs that the compiler believes may be errors. We highly recommend that you compile your code with -Wall. Finding bugs at compile time is soooo much easier than run time. the -Wall option can feel like a nag, but it's worth it. If a student comes to me with an assignment that does not work, and it produces -Wall warnings, then maybe 30% of the time, the warnings were a clue towards the problem. 30% may not sound like that much, but you have to appreciate that it's **free** debugging.

Sometimes -Wall warnings are not actually problems. The code is ok, and the compiler just needs to be convinced. Don't ignore the warning. Fix up the source code so the warning goes away. Getting used to compiles that produce "a few warnings" is a very bad habit.

Here's an example bit of code you could use to assign and test a flag variable in one step...

```
int flag;

if (flag = IsPrime(13)) {
  ...
}
```

The compiler will give a warning about a possibly unintended assignment, although in this case the assignment is correct. This warning would catch the common bug where you meant to type == but typed = instead. To get rid of the warning, re-write the code to make the test explicit...

```
int flag;

if ((flag = IsPrime(13)) != 0) {
  ...
}
```

This gets rid of the warning, and the generated code will be the same as before. Alternately, you can enclose the entire test in another set of parentheses to indicate your intentions. This is a small price to pay to get -Wall to find some of your bugs for you.

`-I`*`dir`* Adds the directory *dir* to the list of directories searched for #include files. The compiler will search several standard directories automatically. Use this option to add a directory for the compiler to search. There is no space between the "-I" and the directory name. If the compile fails because it cannot find a #include file, you need a -I to fix it.

Extra: Here's how to use the unix "find" command to find your #include file. This example searches the /usr/include directory for all the include files with the pattern "inet" in them...

```
nick% find /usr/include -name '*inet*'
/usr/include/arpa/inet.h
/usr/include/netinet
/usr/include/netinet6
```

`-l`*`mylib`* (lower case 'L') Search the library named *mylib* for unresolved symbols (functions, global variables) when linking. The actual name of the file will be lib*mylib*.a, and must be found in either the default locations for libraries or in a directory added with the `-L` flag (below).

The position of the `-l` flag in the option list is important because the linker will not go back to previously examined libraries to look for unresolved symbols. For example, if you are using a library that requires the math library it must appear before the math library on the command line otherwise a link error will be reported. Again, there is no space between the option flag and the library file name, and that's a lower case 'L', not the digit '1'. If your link step fails because a symbol cannot be found, you need a -l to add the appropriate library, or somehow you are compiling with the wrong name for the function or global variable.

`-L`*`dir`* Adds the directory *dir* to the list of directories searched for library files specified by the `-l` flag. Here too, there is no space between the option flag and the library directory name. If the link step fails because a library file cannot be found, you need a -L, or the library file name is wrong.

Section 2 — make

Typing out the gcc commands for a project gets less appealing as the project gets bigger. The "make" utility automates the process of compiling and linking. With make, the programmer specifies what the files are in the project and how they fit together, and then make takes care of the appropriate compile and link steps. Make can speed up your compiles since it is smart enough to know that if you have 10 .c files but you have only changed one, then only that one file needs to be compiled before the link step. Make has some complex features, but using it for simple things is pretty easy.

Running make

Go to your project directory and run make right from the shell with no arguments, or in emacs (below) `[esc]-x compile` will do basically the same thing. In any case, make looks in the current directory for a file called `Makefile` or `makefile` for its build instructions. If there is a problem building one of the targets, the error messages are written to standard error or the emacs compilation buffer.

Makefiles

A makefile consists of a series of variable definitions and dependency rules. A variable in a makefile is a name defined to represent some string of text. This works much like macro replacement in the C pre-processor. Variables are most often used to represent a list of directories to search, options for the compiler, and names of programs to run. Variables are not pre-declared, you just set them with '='. For example, the line :

```
CC = gcc
```

will create a variable named `CC`, and set its value to be `gcc`. The name of the variable is case sensitive, and traditionally make variable names are in all upper case letters.

While it is possible to make up your own variable names, there are a few names that are considered standard, and using them along with the default rules makes writing a makefile much easier. The most important variables are: CC, CFLAGS, and LDFLAGS.

`CC`	The name of the C compiler, this will default to cc or gcc in most versions of make.
`CFLAGS`	A list of options to pass on to the C compiler for all of your source files. This is commonly used to set the include path to include non-standard directories (-I) or build debugging versions (-g).
`LDFLAGS`	A list of options to pass on to the linker. This is most commonly used to include application specific library files (-l) and set the library search path (-L).

To refer to the value of a variable, put a dollar sign ($) followed by the name in parenthesis or curly braces...

```
CFLAGS = -g -I/usr/class/cs107/include
$(CC) $(CFLAGS) -c binky.c
```

The first line sets the value of the variable `CFLAGS` to turn on debugging information and add the directory `/usr/class/cs107/include` to the include file search path. The second line uses CC variable to get the name of the compiler and the CFLAGS variable

to get the options for the compiler. A variable that has not been given a value has the empty-string value.

The second major component of a makefile is the dependency/build rule. A rule tells how to make a target based on changes to a list of certain files. The ordering of the rules does not make any difference, except that the first rule is considered to be the default rule -- the rule that will be invoked when make is called without arguments (the most common way).

A rule generally consists of two lines: a dependency line followed by a command line. Here is an example rule :

```
binky.o : binky.c binky.h akbar.h
tab$(CC) $(CFLAGS) -c binky.c
```

This dependency line says that the object file `binky.o` must be rebuilt whenever any of `binky.c`, `binky.h,` or `akbar.h` change. The target `binky.o` is said to depend on these three files. Basically, an object file depends on its source file and any non-system files that it includes. The programmer is responsible for expressing the dependencies between the source files in the makefile. In the above example, apparently the source code in `binky.c` #includes both `binky.h` and `akbar.h`-- if either of those two .h files change, then `binky.c` must be re-compiled. (The `make depend` facility tries to automate the authoring of the makefile, but it's beyond the scope of this document.)

The command line lists the commands that build `binky.o` -- invoking the C compiler with whatever compiler options have been previously set (actually there can be multiple command lines). Essentially, the dependency line is a trigger which says when to do something. The command line specifies what to do.

The command lines must be indented with a `tab` characte -- just using spaces will not work, even though the spaces will sortof look right in your editor. (This design is a result of a famous moment in the early days of make when they realized that the tab format was a terrible design, but they decided to keep it to remain backward compatible with their user base -- on the order of 10 users at the time. There's a reason the word "backward" is in the phrase "backward compatible". Best to not think about it.)

Because of the tab vs. space problem, make sure you are not using an editor or tool which might substitute space characters for an actual tab. This can be a problem when using copy/paste from some terminal programs. To check whether you have a tab character on that line, move to the beginning of that line and try to move one character to the right. If the cursor skips 8 positions to the right, you have a tab. If it moves space by space, then you need to delete the spaces and retype a tab character.

For standard compilations, the command line can be omitted, and make will use a default build rule for the source file based on its file extension, .c for C files, .f for Fortran files, and so on. The default build rule for C files looks like...

```
$(CC) $(CFLAGS) -c source-file.c
```

It's very common to rely on the above default build rule -- most adjustments can be made by changing the CFLAGS variable. Below is a simple but typical looking makefile. It compiles the C source contained in the files `main.c, binky.c`, `binky.h`, `akbar.c`, `akbar.h`, and `defs.h`. These files will produce intermediate files `main.o`, `binky.o`, and `akbar.o`. Those files will be linked together to produce the executable file `program`. Blank lines are ignored in a makefile, and the comment character is '#'.

```
## A simple makefile

CC = gcc
CFLAGS = -g -I/usr/class/cs107/include
LDFLAGS = -L/usr/class/cs107/lib -lgraph


PROG = program
HDRS = binky.h akbar.h defs.h
SRCS = main.c binky.c akbar.c

## This incantation says that the object files
## have the same name as the .c files, but with .o
OBJS = $(SRCS:.c=.o)

## This is the first rule (the default)
## Build the program from the three .o's
$(PROG) : $(OBJS)
tab$(CC) $(LDFLAGS) $(OBJS) -o $(PROG)


## Rules for the source files -- these do not have
## second build rule lines, so they will use the
## default build rule to compile X.c to make X.o
main.o : main.c binky.h akbar.h defs.h

binky.o : binky.c binky.h

akbar.o : akbar.c akbar.h defs.h


## Remove all the compilation and debugging files
clean :
tabrm -f core $(PROG) $(OBJS)

## Build tags for these sources
TAGS : $(SRCS) $(HDRS)
tabetags -t $(SRCS) $(HDRS)
```

The first (default) target builds the program from the three `.o`'s. The next three targets such as "`main.o : main.c binky.h akbar.h defs.h`" identify the `.o`'s that need to be built and which source files they depend on. These rules identify what needs to be built, but they omit the command line. Therefore they will use the default rule which knows how to build one `.o` from one `.c` with the same name. Finally, `make` automatically knows that a `X.o` always depends on its source `X.c`, so `X.c` can be omitted from the rule. So the first rule could b ewritten without `main.c` -- "`main.o : binky.h akbar.h defs.h`".

The later targets, `clean` and `TAGS`, perform other convenient operations. The `clean` target is used to remove all of the object files, the executable, and a core file if you've been debugging, so that you can perform the build process from scratch . You can `make clean` if you want to recover space by removing all the compilation and debugging output files. You also may need to `make clean` if you move to a system with a different architecture from where your object libraries were originally compiled, and so

you need to recompile from scratch. The `TAGS` rule creates a tag file that most Unix editors can use to search for symbol definitions.

Compiling in Emacs

Emacs has built-in support for the compile process. To compile your code from emacs, type `M-x compile`. You will be prompted for a compile command. If you have a makefile, just type `make` and hit return. The makefile will be read and the appropriate commands executed. The emacs buffer will split at this point, and compile errors will be brought up in the newly created buffer. In order to go to the line where a compile error occurred, place the cursor on the line which contains the error message and hit `^c-^c`. This will jump the cursor to the line in your code where the error occurred ("cc" is the historical name for the C compiler).

Section 3 — gdb

You may run into a bug or two in your programs. There are many techniques for finding bugs, but a good debugger can make the job a lot easier. In most programs of any significant size, it is not possible to track down all of the bugs in a program just by staring at the source — you need to see clues in the runtime behavior of the program to find the bug. It's worth investing time to learn to use debuggers well.

GDB

We recommend the GNU debugger gdb, since it basically stomps on dbx in every possible area and works nicely with the gcc compiler. Other nice debugging environments include ups and CodeCenter, but these are not as universally available as gdb, and in the case of CodeCenter not as cheaply. While gdb does not have a flashy graphical interface as do the others, it is a powerful tool that provides the knowledgeable programmer with all of the information they could possibly want and then some.

This section does not come anywhere close to describing all of the features of gdb, but will hit on the high points. There is on-line help for gdb which can be seen by using the `help` command from within gdb. If you want more information try `xinfo` if you are logged onto the console of a machine with an X display or use the info-browser mode from within emacs.

Starting the debugger

As with make there are two different ways of invoking gdb. To start the debugger from the shell just type...

`gdb` *`program`*

where *`program`* is the name of the target executable that you want to debug. If you do not specify a target then gdb will start without a target and you will need to specify one later before you can do anything useful.

As an alternative, from within emacs you can use the command `[Esc]-x gdb` which will then prompt you for the name of the executable file. You cannot start an inferior gdb session from within emacs without specifying a target. The emacs window will then split between the gdb buffer and a separate buffer showing the current source line.

Running the debugger

Once started, the debugger will load your application and its symbol table (which contains useful information about variable names, source code files, etc.). This symbol

table is the map produced by the -g compiler option that the debugger reads as it is running your program.

The debugger is an interactive program. Once started, it will prompt you for commands. The most common commands in the debugger are: setting breakpoints, single stepping, continuing after a breakpoint, and examining the values of variables.

Running the Program

`run`	Reset the program, run (or rerun) from the beginning. You can supply command-line arguments the same way you can supply command-line arguments to your executable from the shell.
`step`	Run next line of source and return to debugger. If a subroutine call is encountered, follow into that subroutine.
`step` *`count`*	Run *count* lines of source.
`next`	Similar to step, but doesn't step into subroutines.
`finish`	Run until the current function/method returns.
`return`	Make selected stack frame return to its caller.
`jump` *`address`*	Continue program at specified line or address.

When a target executable is first selected (usually on startup) the current source file is set to the file with the main function in it, and the current source line is the first executable line of the this function.

As you run your program, it will always be executing some line of code in some source file. When you pause the program (when the flow of control hits a “breakpoint” of by typing Control-C to interrupt), the “current target file” is the source code file in which the program was executing when you paused it. Likewise, the “current source line” is the line of code in which the program was executing when you paused it.

Breakpoints

You can use breakpoints to pause your program at a certain point. Each breakpoint is assigned an identifying number when you create it, and so that you can later refer to that breakpoint should you need to manipulate it.

A breakpoint is set by using the command `break` specifying the location of the code where you want the program to be stopped. This location can be specified in several ways, such as with the file name and either a line number or a function name within that file (a line needs to be a line of actual source code — comments and whitespace don't count). If the file name is not specified the file is assumed to be the current target file, and if no arguments are passed to `break` then the current source line will be the breakpoint. `gdb` provides the following commands to manipulate breakpoints:

`info break`	Prints a list of all breakpoints with numbers and status.

`break` *`function`*	Place a breakpoint at start of the specified function
`break` *`linenumber`*	Prints a breakpoint at line, relative to current source file.
`break` *`filename:linenumber`*	Place a breakpoint at the specified line within the specified source file.

You can also specify an *if* clause to create a conditional breakpoint:

`break` *`fn`* `if` *`expression`*	Stop at the breakpoint, only if *expression* evaluates to true. Expression is any valid C expression, evaluated within current stack frame when hitting the breakpoint.
`disable` *`breaknum`* `enable` *`breaknum`*	Disable/enable breakpoint identified by *breaknum*
`delete` *`breaknum`*	Delete the breakpoint identified by *breaknum*
`commands` *`breaknum`*	Specify commands to be executed when *breaknum* is reached. The commands can be any list of C statements or gdb commands. This can be useful to fix code on-the-fly in the debugger without re-compiling (Woo Hoo!).
`cont`	Continue a program that has been stopped.

For example, the commands...

```
break binky.c:120
break DoGoofyStuff
```

set a breakpoint on line 120 of the file binky.c and another on the first line of the function `DoGoofyStuff`. When control reaches these locations, the program will stop and give you a chance to look around in the debugger.

Gdb (and most other debuggers) provides mechanisms to determine the current state of the program and how it got there. The things that we are usually interested in are (a) where are we in the program? and (b) what are the values of the variables around us?

Examining the stack

To answer question (a) use the `backtrace` command to examine the run-time stack. The run-time stack is like a trail of breadcrumbs in a program; each time a function call is made, a crumb is dropped (an run-time stack frame is pushed). When a return from a function occurs, the corresponding stack frame is popped and discarded. These stack frames contain valuable information about the sequence of callers which brought us to the current line, and what the parameters were for each call.

Gdb assigns numbers to stack frames counting from zero for the innermost (currently executing) frame. At any time gdb identifies one frame as the “selected” frame. Variable lookups are done with respect to the selected frame. When the program being debugged stops (at a breakpoint), gdb selects the innermost frame. The commands below can be used to select other frames by number or address.

`backtrace`	Show stack frames, useful to find the calling sequence that produced a crash.
`frame` *framenumber*	Start examining the frame with *framenumber*. This does not change the execution context, but allows to examine variables for a different frame.
`down`	Select and print stack frame called by this one. (The metaphor here is that the stack grows down with each function call.)
`up`	Select and print stack frame that called this one.
`info args`	Show the argument variables of current stack frame.
`info locals`	Show the local variables of current stack frame.

Examining source files

Another way to find our current location in the program and other useful information is to examine the relevant source files. `gdb` provides the following commands:

`list` *linenum*	Print ten lines centered around *linenum* in current source file.
`list` *function*	Print ten lines centered around beginning of *function (*or *method).*
`list`	Print ten more lines.

The `list` command will show the source lines with the current source line centered in the range. (Using `gdb` from within `emacs` makes these command obsolete since it does all of the current source stuff for you.)

Examining data

To answeer the question (b) “what are the values of the variables around us?” use the following commands...

`print` *expression*	Print value of *expression.* Expression is any valid C expression, can include function calls and arithmetic expressions, all evaluated within current stack frame.
`set` *variable = expression*	Assign value of *variable* to *expression*. You can set any variable in the current scope. Variables which begin with $ can be used as temporary variables local to gdb.
`display` *expression*	Print value of *expression* each time the program stops. This can be useful to watch the change in a variable as you step through code.
`undisplay`	Cancels previous display requests.

In gdb, there are two different ways of displaying the value of a variable: a snapshot of the variable's current value and a persistent display for the entire life of the variable. The `print` command will print the current value of a variable, and the `display` command will make the debugger print the variable's value on every step for as long as the variable exists. The desired variable is specified by using C syntax. For example...

```
print x.y[3]
```

will print the value of the fourth element of the array field named `y` of a structure variable named `x`. The variables that are accessible are those of the currently selected function's activation frame, plus all those whose scope is global or static to the current target file. Both the `print` and `display` functions can be used to evaluate arbitrarily complicated expressions, even those containing, function calls, but be warned that if a function has side-effects a variety of unpleasant and unexpected situations can arise.

Shortcuts

Finally, there are some things that make using gdb a bit simpler. All of the commands have short-cuts so that you don't have to type the whole command name every time you want to do something simple. A command short-cut is specified by typing just enough of the command name so that it unambiguously refers to a command, or for the special commands `break`, `delete`, `run`, `continue`, `step`, `next` and `print` you need only use the first letter. Additionally, the last command you entered can be repeated by just hitting the return key again. This is really useful for single stepping for a range while watching variables change.

Miscellaneous

`editmode` *`mode`*	Set editmode for gdb command line. Supported values for *mode* are emacs, vi, dumb.
`shell` *`command`*	Execute the rest of the line as a shell command.
`history`	Print command history.

Debugging Strategies

Some people avoid using debuggers because they don't want to learn another tool. This is a mistake. Invest the time to learn to use a debugger and all its features — it will make you much more productive in tracking down problems.

Sometimes bugs result in program crashes (a.k.a. "core dumps", "register dumps", etc.) that bring your program to a halt with a message like "Segmentation Violation" or the like. If your program has such a crash, the debugger will intercept the signal sent by the processor that indicates the error it found, and allow you to examine the state program. Thus with almost no extra effort, the debugger can show you the state of the program at the moment of the crash.

Often, a bug does not crash explicitly, but instead produces symptoms of internal problems. In such a case, one technique is to put a breakpoint where the program is misbehaving, and then look up the call stack to get some insight about the data and control flow path that led to the bad state. Another technique is to set a breakpoint at some point before the problems start and step forward towards the problems, examining the state of the program along the way.

Section 4 — emacs

The following is a quick introduction to the “emacs” text editor which is a free program produced by GNU (www.gnu.org). It's a fine editor, and it happens to integrate with many other Unix tools nicely. There's a fabulous history of various editor adherents having long and entertaining arguments about why their editor is best, but we're just going to avoid that subject entirely.

To start editing a new or existing file using emacs, simply type the following to the UNIX prompt...

`emacs` *filename*

where *filename* is the file to be edited. The X-Windows version of emacs is called `xemacs`, and if you're using it... well just look in the menus. The commands are all the same, but you don't have to remember the funny key-combinations.

All the fancy editing commands, such as find-and-replace, are invoked through typing special key sequences. Two important key sequences to remember are: `^x` (holding down the “ctrl” key while typing “x”) and `[esc]-x` (simply pressing the “esc” key followed by typing “x”), both of which are used to start many command sequences. Note that for historical reasons in most user manuals for emacs, the “esc” key is actually referred to as the “Meta” or “M-” key. Therefore, you may see the `[esc]-x` written as equivalently as `M-x.`

To save the file being edited the sequence is `^x^s`. To exit (and be prompted to save) emacs, the sequence is `^x^c`. To open another file within emacs, the sequence is `^x^f`. This sequence can be used to open an existing file as well as a new file. If you have multiple files open, emacs stores them in different “buffers”. To switch from one buffer to another (very handy when you are editing a `.c` source file and need to refer to the prototypes and definitions in the `.h` header file), you use the key sequence `^x-b` (note the “b” is typed plain). You can then enter the name of the file to switch to the corresponding buffer (a default name is provided for fast switching). The arrow keys usually work as the cursor movement keys, but there are other nagivation key combinations listed below.

Running emacs

`emacs <filename>`	Run `emacs` (on a particular file). Make sure you don't already have an `emacs` job running which you can just revive with `fg`. Adding a '**&**' after the above command will run emacs in the background, freeing up your shell)
`^z`	Suspend `emacs`— revive with % command above, or the `fg` command
`^x^c`	Quit `emacs`
`^x^f`	Load a new file into emacs
`^x^v`	Load a new file into emacs and unload previous file
`^x^s`	Save the file
`^x-k`	Kill a buffer

Moving About

`^f`	Move forward one character
`^b`	Move backward one character
`^n`	Move to next line
`^p`	Move to previous line

^a	Move to beginning of line
^e	Move to end of line
^v	Scroll down a page
M-v	Scroll up a page
M-<	Move to beginning of document
^x-[	Move to beginning of page
M->	Move to end of document
^x-]	Move to end of page
^l	Redraw screen centered at line under the cursor
^x-o	Move to other screen
^x-b	Switch to another buffer

Searching

^s	Search for a string
^r	Search for a string backwards from the cursor (quit both of these with ^f)
M-%	Search-and-replace

Deleting

^d	Deletes letter under the cursor
^k	Kill from the cursor all the way to the end of the line
^y	Yanks back all the last kills. Using the ^k ^y combination you can get a primitive cut-paste effect to move text around

Regions

emacs defines a region as the space between the mark and the point. A mark is set with ^-space (control-spacebar). The point is at the cursor position.

M-w	Copy the region
^w	Delete the region. Using ^y will also yank back the last region killed or copied — this is the way to get a cut/copy/paste effect with regions.

Screen Splitting

^x-2	Split screen horizontally
^x-3	Split screen vertically
^x-1	Make active window the only screen
^x-0	Make other window the only screen

Miscellaneous

M-$	Check spelling of word at the cursor
^g	In most contexts, cancel, stop, go back to normal command
M-x goto-line *num*	Goes to the given line number
^x-u	Undo
M-x shell	Start a shell within emacs
M-q	Re-flow the current line-breaks to make a single paragraph of text

Compiling

`M-x compile`	Compile code in active window. Easiest if you have a makefile set up.
`^c ^c`	Do this with the cursor in the compile window, scrolls to the next compiler error. Cool!

Getting Help

`^h`	emacs help
`^h t`	Run the emacs tutorial

emacs does command completion for you. Typing `M-x space` will give you a list of emacs commands. There is also a man page on emacs. Type `man emacs` in a shell.

Printing Your Source Files

There's a really neat way to print out hardcopies of your source files. Use a command called “enscript”. Commonly, it's used at the Unix command line as follows:

```
enscript -2GrPsweet5 binky.c lassie.c *.h
```

This example shoes printing the two source files `binky.c` and `lassie.c`, as well as all of the header files to printer sweet5. You can change these parameters to fit your needs.

Section 5 — Unix Shell

This section summarizes many of the commands used in the Unix shell.

Directory Commands

`cd` *`directory`*	Change directory. If *directory* is not specified, goes to home directory.
`pwd`	Show current directory (*p*rint *w*orking *d*irectory)
`ls`	Show the contents of a directory. **ls -a** will also show files whose name begins with a dot. **ls -l** shows lots of miscellaneous info about each file. **ls -t** sorts the most recently changed to the top.
`rm` *`file`*	Delete a file
`mv` *`old new`*	Rename a file from *old* to *new* (also works for moving things between directories). If there was already a file named *new*, it gets overwritten.
`cp` *`old new`*	Creates a file named *new* containing the same thing as *old*. If there was already a file named new, it is overwritten.
`mkdir` *`name`*	Create a directory
`rmdir` *`name`*	Delete a directory. The directory must be empty.

Shorthand Notations & Wildcards

`.`	Current directory
`..`	Parent directory
`~`	Your home directory
~/cs107	The cs107 directory in your home directory
`~`*`user`*	Home directory of *user*
`*`	Any number of characters (not '.') Ex: *.c is all files ending in '.c'
`?`	Any single character (not '.')

Miscellaneous Commands

`cat` *`file`*	Print the contents of *file* to standard output
`more` *`file`*	Same as `cat`, but only a page at a time (useful for displaying)
`less` *`file`*	Same as `more`, but with navigability (`less` is `more`)
`w`	Find out who is on the system and what they are doing
`ps`	List all your processes (use the process id's in `kill` below)
`jobs`	Show jobs that have been suspended (use with fg)
`program``&`	Runs *program* in the background
`ctrl-z`	Suspend the current program
`%`	Continue last job suspended, or use `fg` (foreground)
`%`*`number`*	Continue a particular job (the number comes from the `jobs` list)
`kill` *`process-id`*	Kill a process
`kill -9` *`process`*	Kill a process with extreme prejudice
`grep` *`exp files`*	Search for an expression in a set of files
`wc` *`file`*	Count words, lines, and characters in a file
`script`	Start saving everything that happens in a file. type **exit** when done
`lpr` *`file`*	Print *file* to the default printer
`lpr -Pinky` *`file`*	Print *file* to the printer named *inky*
`diff` *`file1 file2`*	Show the differences between two files
`telnet` *`hostname`*	Log on to another machine
`source` *`file`*	Execute the lines in the given file as if they were typed to the shell

Getting Help

`man` *`subject`*	Read the manual entry on a particular subject
`man -k keyword`	Show all the manual pages for a particular keyword

History

`history`	Show the most recent commands executed
`!!`	Re-execute the last command (or type up-arrow with modern shells)
`!`*`number`*	Re-execute a particular command by number
`!`*`string`*	Re-execute the last command beginning with string
`^`*`wrong`*`^`*`right`*`^`	Re-execute the last command, substituting right for wrong
`ctrl-P`	Scroll backwards through previous commands

Pipes

`a` `>` *`b`*	Redirect a's standard output to overwrite file b
`a` `>>` *`b`*	Redirect a's standard output to append to the file b
`a` `>&` *`b`*	Redirect a's error output to overwrite file b
`a` `<` *`b`*	Redirect a's standard input to read from the file b
`a` `|` *`b`*	Redirect a's standard output to b's standard input