Programming on Unix

Many people contributed to this handout as a long-term project over the years. In particular, I got help from Mike Cleron and Peter Chang.

Introduction

In writing C programs to run under Unix, there are several concepts and tools that turn out to be quite useful. The most obvious difference, if you are coming from a PC or Macintosh programming background, is that the tools are separate entities, not components in a tightly coupled environment like Think C or Borland C. The appendix at the end of the handout gives a summary of UNIX and EMACS commands.

The most important tools in this domain are the editor, the compiler, the linker, the `make` utility, and the debugger. There are a variety of choices as far as "which compiler" or "which editor", but the choice is usually one of personal preference. The choice of editor, however, is almost a religious issue. `Emacs` integrates well with the other tools, has a nice graphical interface, and is almost an operating system unto itself, so we will encourage its use.

Caveat

This handout is not meant to give a comprehensive discussion of any of the tools discussed, rather it should be used as an introduction to getting started with the tools. If you find that you need more information, all of the programs have extensive `man` pages and `xinfo` entries, and `gdb` has some on-line help for all of its commands. These man pages list a lot of bits of information, some of which will appear to be quite attractive, but if you do not know or understand what an option does, especially for the compiler, please do not use it just because it sounds nice.

Also, O'Reilly & Associates publishes a pretty good set of references for these basic tools, the titles of which resemble "UNIX in a Nutshell". These are not required or endorsed (for the record), but may be useful if you get lost. Also, the L&IR people hold classes in Sweet Hall covering "how to use UNIX machines", etc. See their page at http://consult.stanford.edu.

The Compilation Process

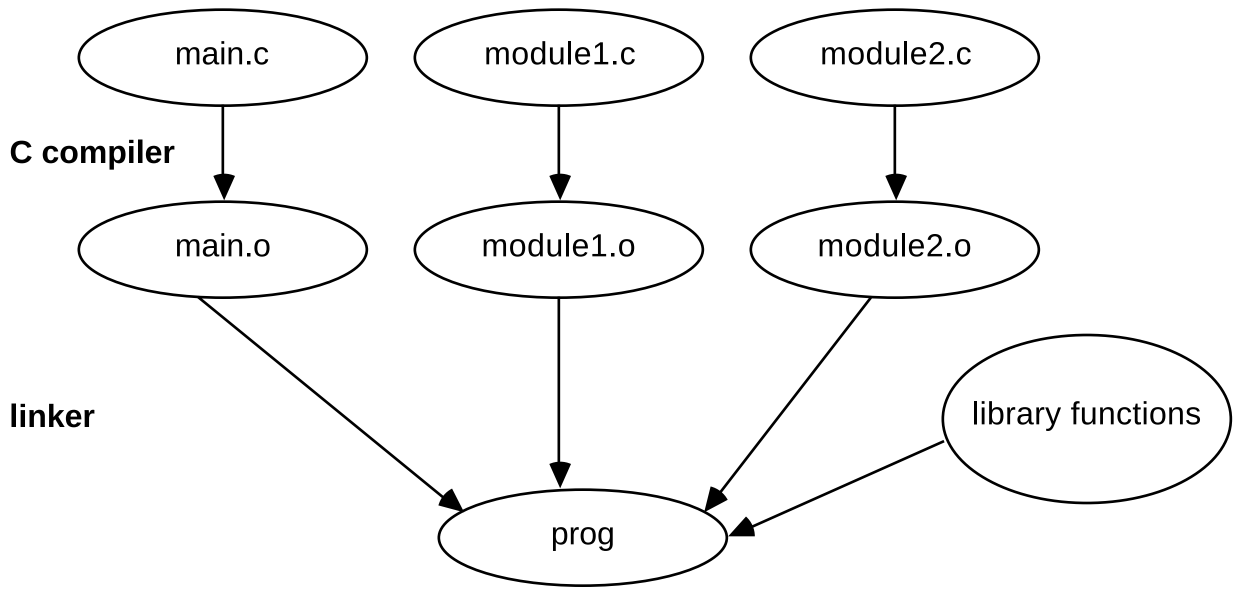
Before going into detail about the actual tools themselves, it is useful to review what happens during the construction of an executable program. There are actually two phases in the process, *compilation* and *linking*. The individual source files must first be *compiled* into object modules. These object modules contain a system dependent, relocatable representation of the program as described in the source file. The individual object modules are then *linked* together to produce a single executable file which the system loader can use when the program is actually invoked. (This process is illustrated by the diagram on the next page.) These phases are often combined with the `gcc` command, but it is quite useful to separate them when using `make`.

For example, the command:

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

can be broken down into the four steps shown below:

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



Compiler/Linker

Although there are a lot of different compilers out there we “guarantee” that all of the problems can be solved using the GNU C Compiler, `gcc` to its friends. Using `gcc` has several advantages, it is pretty much ANSI compliant, available on a variety of different platforms and, more importantly for you, it works pretty reliably. The current version of `gcc` installed on the L&IR machines is 2.6, and directly compiles C, C++, and Objective-C.

Running gcc

Even though it is called a compiler, `gcc` is used as both a compiler and linker. The general form for invoking `gcc` is :

```
gcc <option flags> <file list>
```

where `<option flags>` is a list of command flags that control how the compiler works, and `<file list>` is a list of files, source or object, that `gcc` is being directed to process. It is not, however, commonly invoked directly from the command line, that is what makefiles are for. If `gcc` is unable to process the files correctly it will print error messages on standard error. Some of these error messages, however, will be caused by `gcc` trying to recover from a previous error so it is best to try to tackle the errors in order.

Command-line options

Like almost all UNIX programs `gcc` has a myriad of options that control almost every aspect of its actions. However, most of these options deal with system dependent features and do not concern us. The most useful of these option flags for us are: -c, -o, -g, -Wall, -I, -L, and -l.

-c	Requests that `gcc` compile the specific source file directly into an object file without going through the linking stage. This is important for compiling only those files that have changed rather than the whole project.
-o *file*	The output from the compiler will be named *file*. If this option is not specified, the default is to create a file '`a.out`' if linking an executable or an object file with the name of the original source file with the suffix, .c for C files, replaced with .o. This is most useful for creating an application with a specific name, rather than changing the names of object files.
-g	Directs the compiler to produce debugging information. We recommend that you always compile your source with this option set. Note – The debugging information generated is for `gdb`, and could possibly cause problems with `dbx`. This is because there is typically more information stored for `gdb` that `dbx` will barf on. Additionally, on some systems, some `MIPS` based machines for example, this information cannot encode full symbol information and some debugger features may be unavailable.
-Wall	Give warnings about a lot of syntactically correct but dubious constructs. Think of this option as being a way to do a simple form of style checking. Again, we highly recommend that you compile your code with this option set. Most of the time the constructs that are flagged are actually incorrect usages, but there are occasionally instances where they are what you really want. Instead of simply ignoring these warnings there are simple workarounds for almost all of the warnings if you insist on doing things this way.

This sort of contrived snippet is a commonly used construct in C to set and test a variable in as few lines as possible :

```
bool flag;
int x = 13;

if (flag = IsPrime(x)) {
  ...
} else {
  < Do Error Stuff >
}
```

The compiler will give a warning about a possibly unintended assignment. This is because it is more common to have a boolean test in the `if` clause using the equality operator `==` rather than to take advantage of the return value of the assignment operator. This snippet could better be written as :

```
if ((flag = IsPrime(x)) == 0) {
  < Do Error Stuff >
} else {
  ...
}
```

so that the test for the `0` value is made explicit. The code generated will be the same, and it will make us and the compiler happy at the same time.

-I*dir* Adds the directory *dir* to the list of directories searched for include files. This will be important for any additional files that we give you. There are a variety of standard directories that will be searched by the compiler by default, for standard library and system header files, but since we do not have root access we cannot just add our files to these locations.

There is no space between the option flag and the directory name.

-l*lib* Search the library named *lib* for unresolved names when linking. The actual name of the file will be lib*lib*.a, and must be found in either the default locations for libraries or in a directory added with the '-L' flag.

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 names. 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.

-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.

make

As many of you probably already know, typing the entire command line to compile a program turns out to be a somewhat complicated and tedious affair. What the `make` utility does is to allow the programmer to write out a specification of all of the modules that go into creating an application, and how these modules need to be assembled to create the program. The `make` facility manages the execution of the necessary build commands (compiling, linking, loading etc.). In doing so, it also recognizes that only those files which have been changed need be rebuilt. Thus a properly constructed `makefile` can save a great deal of compilation time. Some people are "afraid" of `make` and its corresponding `makefile`, but in actuality creating a `makefile` is a pretty simple affair.

Running make

Invoking the `make` program is really simple, just type '`make`' at the shell prompt, or if you are an emacs aficionado 'M-x compile' will do basically the same thing[1]. Either of these commands will cause `make` to look in the current directory for a file called '`Makefile`' or '`makefile`' for the build instructions. If there is a problem building one of the targets along the way the error messages will appear on standard error or the emacs '`compilation`' buffer if you invoked `make` from within emacs.

Makefile-craft

A `makefile` consists of a series of `make` 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 compiler's 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. A variable is "declared" when it is set to a value. 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 capital letters.

While it is possible to define your own variables there are some that are considered 'standard,' and using them along with the default rules makes writing a `makefile` much easier. For the purposes of this class the important variables are: CC, CFLAGS, and LDFLAGS.

CC	The name of the C compiler, this will default to `cc` in most versions of `make`. Please make sure that you set this to be `gcc` since `cc` is not ANSI compliant on the LaIR SparcStations, and we will only be using ANSI C in this class.
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 or build debugging versions, the -I and -g compiler flags.

1 'M-x' means hold the 'meta' key down while hitting the 'x' key. If your keyboard does not have a 'meta' key then the 'ESC' will do the same thing. Hit the 'ESC' key and then the 'x' key. Do not hold down the 'ESC' key or else it will put you into eval mode.

LDFLAGS	A list of options to pass on to the linker. This is most commonly used to set the library search path to non-standard directories and to include application specific library files, the -L and -l compiler flags.

Referencing the value of a variable is done by having a '$' followed by the name of the variable within parenthesis or curly braces. For example :

```
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 the value of the variable CC as the name of the compiler to use passing to it the compiler options set in the previous line. If you use a variable that has not been previously set in the makefile, `make` will use the empty definition, an empty string.

The second major component of makefiles are dependency/build rules. A rule tells how to make a target based on changes to a list of certain files. The ordering of the rules in the `makefile` does not make any difference, except that the first rule is considered to be the default rule. The default rule is the rule that will be invoked when `make` is called without arguments, the usual way. If, however, you know eaxctly which rule you want to invoke you can name it directly with an argument to `make`. For example, if my makefile had a rule for '`clean`,' the command line '`make clean`' would invoke the actions listed after the `clean` label, more on actions later.

A rule generally consists of two lines, a dependency list and a command list. Here is an example rule :

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

The first line says that the object file `binky.o` must be rebuilt whenever `binky.c`, `binky.h,` or `akbar.h` are changed. 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 second line[2] lists the commands that must be taken in order to rebuild `binky.o`, invoking the C compiler with whatever compiler options have been previously set. These lines must be indented with a `<tab>` character, just using spaces will not work. This is a problem when using copy/paste from some terminal programs. For "standard" compilations[3], the second line can be omitted, and `make` will use the default build rule for the source file based on its extension, .c for C files. The default build rule that `make` uses for C files looks like this :

```
$(CC) $(CFLAGS) -c <source-file>
```

---

2 The second line can actually be more than one line if multiple commands need to be done for a single target. In this class, however, we will not be doing anything that requires multiple commands per target.

3 Most versions of make handle at least FORTRAN, C, and C++.

Here is a “complete” makefile for your reading pleasure.

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

PROG = example
HDRS = binky.h akbar.h defs.h
SRCS = main.c binky.c akbar.c
OBJS = main.o binky.o akbar.o

$(PROG) : $(OBJECTS)
  $(CC) -o $(PROG) $(LDFLAGS) $(OBJS)

clean :
  rm -f core $(PROG) $(OBJS)

TAGS : $(SRCS) $(HDRS)
  etags -t $(SRCS) $(HDRS)

main.o : binky.h akbar.h defs.h
binky.o : binky.h
akbar.o : akbar.h defs.h
```

This makefile includes two extra targets, in addition to building the executable: `clean` and `TAGS`. These are commonly included in makefiles to make your life as a programmer a little bit easier. The `clean` target is used to remove all of the object files and the executable so that you can start the build process from scratch[4], you will need to do this if you move to a system with a different architecture from where your object libraries were originally compiled. The `TAGS` rule creates a tag file that most Unix editors can use to search for symbol definitions[5].

Compiling in Emacs

The Emacs editor provides 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 ctrl-c ctrl-c. This will jump the cursor to the line in your code where the error occurred.

4 It also removes any ‘core’ files that you might have lying around, not that there should be any.

5 Use ‘M-x find-tag’ or ‘M-.’ in emacs to search for a symbol within emacs. Use tags, they make your life a lot easier.

The Debugger (gdb)

During the course of the quarter you may run into a bug or two in your programs[6]. There are a variety of different techniques for finding these “anomalies,” but a good debugger can make the job a lot easier and faster. We are recommending the GNU debugger, since it basically stomps on dbx in every possible way and works nicely with the `gcc` compiler we recommend. Other nice debugging environments include `ups` and `CodeCenter`, but these are not as universally availible 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 she could possibly want and then some.

This section does not come anywhere close to describing all of the features of `gdb`, but will, rather, 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.

A debugger is invaluable to a programmer because it eases the process of discovering and repairing bugs at run-time. In most programs of any significant size, it is not possible to determine all of the bugs in a program at compile-time because of oversights and misconceptions about the problem that the application is designed to solve.

The way debuggers allow you to find bugs is by allowing you to run your program on a line-by-line basis, pausing the program at times or conditions that you specify and allowing you to examine variables, registers, the run time stack and other facets of program state while paused.

Sometimes these 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 a severe bug that causes a program crash, the debugger will "catch" the signal sent by the processor that indicates the error it found, and allow you to further examine the program. This information can be quite valuable when trying to reason about what caused your program to die, all segmentation faults sort of look the same.

Starting the debugger

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

```
gdb <Target Name>
```

where `<Target Name>` is the name of the 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 ‘`M-x gdb`’ which will then prompt you for the name of the target file. You cannot start an inferior `gdb` session from within `emacs` without specifying a target. The `emacs` window will then split between the `gdb` ‘window’ and a buffer containing the current source line.

6 We recommend that you have fewer.

Running the debugger

Once started, the debugger will load your application and its symbol table (which contains useful information about varaible names, source code files, etc.). This symbol table is the map that the debugger reads as it is running your program.

*Warning:*If you forget to specify the '-g' flag (debugging info.) when compiling your source files, this symbol table will be missing from your program and `gdb` (and you) will be "in the dark" as your program runs.

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 to 'run' 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.

As you run your program, it will always be executing some line of code in some source file. When you pause the program (using a "breakpoint"), 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.

When a target application 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.

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 a variety of different ways; file name and line number or file name and function name[7]. If the file name argument is not specified

[7] It is a good idea to specify lines that are really code, comments and whitespace will not do the right thing.

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` *`linenumber`*	
`break` *`function`*	
`break` *`method`*	
`break` *`filename:function`*	
`break` *`filename:linenumber`*	Place a breakpoint at the specified line within the specified source file. You can also specify an *if* clause with any of above.
`break` *`function`* `if` *`expression`*	Stop at the breakpoint, only if *expression* is true. Expression is any valid C or Objective-C expression, evaluated within current stack frame.
`tbreak` *`arguments`*	Place a one-time 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. This can be useful to fix code ™on-the-fly∫ in the debugger without re-compiling.
`cont`	Continue a program that has been stopped.

For example, the commands :

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

set a breakpoint on line 120 of the file binky.c and another on the first line of the function `DoGoofyStuff` which is found in the file akbar.c. The second command could alternatively be specified as :

```
break DoGoofyStuff
```

if there are no other instances of DoGoofyStuff in the program. If it is ambiguous what name is to be used then the debugger will prompt for more information.

`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 "where are we in the program?" and "what are the values of the variables around us?".

Examining the stack

To answer the question of "where are we in the program?", we use the '`where`' 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 RT stack frame is pushed). When a return from a function occurs, the corresponding RT stack frame is popped and discarded. These stack frames contain valuable information about where the function was called in the source code (line # and file name), what the parameters for the call were, etc.

`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.
`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:

`view`	Message the Edit application to show the current source lines. Commands like list, step, next, will highlight the code in Edit.
`unview`	Turns off viewing in Edit.
`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

It is also useful to answer the question, "what are the values of the variables around us?" In order to do so, we use the following commands to examine variables:

`print` *`expression`*	Print value of *expression.* Expression is any valid C or Objective-C expression, 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 convenience variables in 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.
`browse` *`object`*	Browse an object. This messages AppInspector to view the specified object. AppInspector allows you to see the values of instance variables, follow pointers, see the chain of inheritance, etc.

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 variables value on every step for as long as the variable is 'live.' 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. For example, 'c' unambiguously refers to 'continue' but 'w' could refer to 'whatis', 'where', or 'watch' so more letters would be needed.

Additionally, most commands can be repeated by just hitting the <return key> again. This is really useful for single stepping for a range while watching variables change. There are, however, a few commands that won't repeat this way. For example, it would not make sense to repeatedly set a breakpoint on the same line if you accidentally hit the return key.

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 Strategy

If your program has been crashing spectacularly, you can just run the program by using the '`run`' command[8] right after you start the debugger. The debugger will catch the signal and allow you to examine the program (and hopefully find the cause and remedy).

More often the bug will be something more subtle. In these cases the "best" strategy is often to try to isolate the source of the bug, using breakpoints and checking the values of the program's variables before setting the program in motion using "run", "step", or "continue". A common technique for isolating bugs is to set a breakpoint at some point before the offending code and slowly continuing toward the crash site examining the state of the program along the way.

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
```

Where we want to print 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.

8 If your application takes command line arguments include these on the same line as the 'run' command.

Appendix A: UNIX[9]/Emacs[10] Survival Guide

This handout summarizes many of the commands helpful for getting around on the Unix operating system and the Emacs editor. AIR provides nice thick readers on both of these topics at the LAIR and on the 2nd floor of Sweet Hall if you need more information.

Basic UNIX

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
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's previous contents are lost.
cp *old new*	Creates a file named new containing the same thing as old. If there was already a file named new, it's previous contents are lost.
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
~<user>	Home Directory of *user*
*	Any number of characters (not '.') Ex: *.c is all files ending in '.c'
?	Any single character (not '.')

[9] UNIX* is a registered trademark of AT&T
*UNIX** is a registered trademark of AT&T
**UNIX† is a registered trademark of AT&T
†...

[10] GNU EMACS is provided for free by the FREE SOFTWARE FOUNDATION, which write software and gives it away because they think that's how software should be.

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 currently active processes
jobs	Show jobs that have been suspended
*process***&**	Runs a process in the background
%	Continue last job suspended (suspend a process with ^Z)
% *number*	Continue a particular job
kill *process-id*	Kill a process
kill -9 *process*	Kill a process with extreme prejudice
grep *exp files*	Look 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
webster *word*	Looks up the given word in the dictionary. Works from most AIR machines, but is not standard UNIX

Getting Help

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

History

history	Show the most recent commands executed
!!	Re-execute the last command
!*number*	Re-execute a particular command
!*string*	Re-execute the last command beginning with string
^*wrong*^*right*^	Re-execute the last command, substituting right for wrong
^P	Scroll backwards through previous commands

Pipes

a > *b*	Redirect a's standard output to the file b
a >> *b*	Redirect a's standard output to append to the file b
a >**&** *b*	Redirect a's error output to the 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

GNUEMACS

For the following "^z" means hit the "z" key while holding down the "ctrl" key. "M-z" means hit the "z" key while hitting the "META" or after hitting the "ESC" key.

Running Emacs

leland>emacs <filename>	run emacs (on a particular file). Make sure you don't already have an emacs job running which you can just revive. Adding a '**&**' after the above command will run emacs in the background, freeing up your shell)
^z	suspend emacs— revive with % command above
^x^c	kill emacs for good
^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

```
                    ^p previous line
        ^b backward              ^f forward
                    ^n next line
```

(Note: the standard arrow keys also usually work.)

^a	go to beginning of line
^e	go to end of line
^v	scroll down a page
M-v	scroll up a page
M-<	go to beginning of document
^x-[	go to beginning of page
M->	go to end of document
^x-]	go 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 word
^r	search for a word backwards from the cursor (both of these terminate with ^f)
M-%	search-and-replace

Deletion

^d	deletes letter under the cursor
^k	kill from the cursor all the way to the right
^y	yanks back all the last kills

using the ^k ^y combination you can get a 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 ^<spc> (control-spacebar). The point is at the cursor position.

M-w	copy the region
^w	kill the region

using ^y will also yank back the last region killed or copied. This is what we used for "paste" back in the bad old mainframe days before there was "paste".

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 <#>	goes to the given line number
^x-u	undo
M-x shell	start a shell within emacs

Compiling

M-x compile	compile code in active window. Easiest if you have a makefile set up.
ctrl-c ctrl-c	do this with the cursor in the compile window, scrolls to the next compiler error. Yay!

Getting Help

^h	emacs help
^h t	run the emacs tutorial

Emacs does command completion for you. Typing M-x <spc> will give you a list of emacs commands. There is also a man page on emacs. Type 'man emacs' in a shell.