CS241 Class Notes
rev 2

John Winans & Eric Bieche

Spring 1999

Contents

Chapter 1

UNIX Command Summary

1.1 Overview

Just a few things that we all need to know before we start.

- Unix is case sensitive... `PROG1.C` is *not* the same as `prog1.c`
- . is the name of the current directory
- .. is that name of the current parent directory
- `$HOME` is a variable that contains the full name of your home directory. It contains something like `/usr2/z912345` which includes your login-ID (aka. your z9 number)
- files that begin with a '.' are hidden files (see ls(1))
- the default command-line prompt on `mp` is `%`

1.1.1 Additional Documentation

Some other sources that you'll want to check are:

- Unix in a Nutshell, Daniel Gilly an O'reily book.
- The online Unix reference manual (see below)

1.2 Online Unix Reference Manual

The online Unix reference manual is read by using the man(1) command. Most texts that reference Unix commands include the Unix manual page section as a reference. For example, the reference to man(1) in this sentence indicated that *man* is a command that is documented in section 1 of the online Unix reference manual. This on manual system displays what are commonly known as "man pages".

Like everything in life, *most* of the man-pages can be overwelming the first time you experience them. The best advice I can give you about them is that you skim them over when help is needed in order to try to figure out what you need. If they do not help, look elsewhere... but return to the man-page afterward in order to try to figure out how that information was presented in the online manual and how you could have determined the answer without the outside help. After a while, you will understand the philosophy of the man(1) system and how to use it for fast answers to questions.

Useful options to the man(1) command are `-s` *section* and `-k` *keyword*

- `-s` *section*

  Used to specify the section number of the desired man-page. The section number is optional, and often not necessary. But if a command-line program has the same name as a C function, or something else that may be in the on-line manual, you may get a man-page for the term you specify, but from the wrong section… and, thus, the wrong thing. For example, there is a write(1) and a write(2) in the online manual. If you leave out the section number, you will get the page for write(1) which is a command used to send instant-messages to other users on the system (write(2) is the doc for a C function that can be used to write data into a file).

  In order to prevent confusion about these things, the man-pages include a set of notes at the end that include references to other man-pages that may be more suitable to what you are looking for. Additionally, the online manual is sectioned based on the types of documents of interest to programmers. Commonly accessed sections are:

  – section 1 User Commands
  – section 2 system calls
  – section 3 C library functions
  – section 4 Devices and network interfaces
  – section 5 File formats

  The sections of interest to C programmers are section 2, 3 and 3c. Examples of how to use the online manual include:

  – `man ls`
  – `man -s1 ls`
  – `man -s3 strstr`
  – `man -s2 write`
  – `man -s3 printf`

- `-k` *keyword*

  This option is used to request a keyword search. It will result in giving you a list of pages including the specified *keyword* in in their name or in their synopsis. For example:

  – `man -k printf`
  – `man -k strncmp`

1.3 Compiling Your Programs

1.3.1 gcc(1) and cc(1)

You will use the gnu compiler gcc(1) in your class so I will stick with gcc for the examples. Note that there is another C compiler named cc(1). The gcc(1) compiler can be more noisy than cc(1) in that it generates more warnings… which are *very* useful while learning how to program.

Some useful options to gcc(1) are:

- `-ansi` generates error messages when you violate ANSI standards

- `-lm` link in the math library (use this if you put `#include<math.h>` in your program)
- `-o` the default executable program name is `a.out` the `-o` allows you to name your own prog name
- `-g` generate special code so that the debugger can present details about source code in the debugger output
- `-Wall` generates warnings about everyting that is even slightly suspicious about your source code
- `-pedantic` considers some of the warnings generated by `-ansi` as errors

It is recommended that students always use `-ansi -Wall -pedantic` when working on their programs. Any instructor-supplied `Makefile` will include these flags.

Some examples:

- `gcc -ansi -Wall -pedantic -lm -o prog1 prog1.c`
- `gcc -ansi -Wall -pedantic prog3.c`
- `gcc -o prog2 prog2.c`

1.3.2 hexdump(1)

This will display a file in a hexadecimal dump format. A useful option is:

- `-C` output both hex and ASCII

Some examples:

- `hexdump -C prog1.c`
- `hexdump -C output.datafile`

1.3.3 Core Files

A core file is automatically produced if your program crashes. These can be quite large, I've seen one that was 1.5 MB. They can be used by the gdb(1) debugger to tell you what went wrong. If you're not using a debugger, you'll want to rm(1) them as they will count toward your disk usage quota.

If you *never* want to use the debugger, consider putting the following in your `.bashrc` file:

```
ulimit -c 0
```

This limit command will prevent them from being automatically generated when programs crash.

1.4 Changing Your Unix Password

You may change your Unix password at any time by logging into `mp.cs.niu.edu` and using passwd(1). An example password change scenairo is shown below:

```
% passwd                             <- I type the command
passwd:  Changing password for winans <- the program comments
(current) UNIX password:             <- I enter my new password
New password:                        <- and again to verify
```

1.5 ps(1)

To see what is happening on a Unix machine, you may use ps(1) to get a listing of the programs (called processes) that are currently executing. To get a complete listing, consider running `ps ax`

1.6 quota(1)

At NIU, each student may use only a limited amount of disk space. In order to check your current usage and limit, use quota(1). Most of the time you will want to use: `quota -v`

1.7 File Management Commands

1.7.1 ls(1)

To see what files are in a given directory, use ls(1). You can used several options to give you different types of information.

Some common options include:

- `-a` wil list all the hidden files as well as the normal ones
- `-l` gives a longer listing with date, time, and permissions
- `-R` will recursively list all subdirectories and their files
- `-d` will list only the directories
- `-F` will put a special character behind each filename to indicate the type of the file. These are:
  - `/` for a dir
  - `*` for executable files
  - `@` for symbolic links

1.7.2 mkdir(1)

To create a new directory, use mkdir(1). A common option is `-p` which will create intervening parent directories if they don't already exist.

Some examples:

- `mkdir Assn1`
- `mkdir progs/assn1`
- `mkdir -p this/is/a/test`

1.7.3 cd(1)

Changes the current working directory.

Some examples:

- `cd` Changes the current working directory to the user's home directory.
- `cd ..` Changes the current working directory to the parent directory *note the space between the d and the first period.*
- `cd 241` Changes the current working directory to a subdirectory named 241 *note that the 241 subdirectory must already exist for this to work.*

1.7.4 rm(1)

You delete unwanted (and if you're not careful... *wanted*) files with rm(1). **Warning:** once you have deleted a file it is **gone** for good! There is no "undelete" like in DOS.

A common option for beginners is:

- `-i` rm(1) will ask you to answer 'y' or 'n' to make sure you want to delete the file

An example:

```
% rm -i Primes.ps
rm: remove Primes.ps (yes/no)?
```

The `-i` option is mostl useful when you do things like `rm -i *c` which would delete all files ending with the letter 'c'.

1.7.5 rmdir(1)

Without special options, the rm(1) command can only delete files. To removes directories use rmdir(1). You can only use rmdir(1) on empty directories.

An example:

```
rmdir -p $HOME/241/assignments/a4/prog1.c
```

1.7.6 mv(1)

In Unix, you do not "rename" a file or directory, you "move" it. Use mv(1) to move file. You can move a file or directoy to a new name in the same directory or into a different directory.

Some examples:

- To rename a file in the same directory

  ```
  mv prog1.c program1.c
  ```

- To move a file into another directory

  ```
  mv prog1.c ../anotherdir/
  mv prog1.c ../diffdir/diffname.c
  ```

- To move a directory into another directory

  ```
  mv subdir ..
  mv subdir ../diffdir/
  ```

1.7.7 cp(1)

To make a copy of a file, use cp(1).

Useful options:

- `-r` will recursively copy a directory, its files, and its subdirectories to a destination directory, duplicating the tree structure.
- `-i` will prompt for confirmation before overwriting an existing file

Some examples:

- To copy two files to their parent directory

  ```
  cp prog1.c prog1.h ..
  ```

- To copy a file to the testprogs directory in your home directory with a confirmation

  ```
  cp -i prog4.c $HOME/testprogs
  ```

1.7.8 cat(1)

To list a file's contents on the screen (normally called standard-out or `stdout`) use cat(1)—the concatenate command. This may be used to list one or more files.

Useful options:

- `-n` will print the file with line numbers
- `-v` Displays any control characters using carat (^) notation
- `-e` prints a $ at the end of a line (useful to detect unwanted spaces at the ends of lines)
- `-t` will print `^I` for each tab (useful to see what the instructor wants when you must generate *exact* output)

Some examples:

- To display a file on the screen with all the above options turned on

  ```
  cat -nvet prog4.key
  ```

- To combine two (or more) files into a big one

  ```
  cat part1file part2file > bigfile
  ```

- To append prog10.c to the end of an existing file named programs

  ```
  cat prog10.c >> programs
  ```

1.7.9 less(1) is more(1)

less(1) and more(1) do the same thing. They list a file on standard-out while pausing at each screenfull of text. less(1) is a nicer lister than more(1). These commands are often used with a *pipe* "`|`". When the command runs, you may:

- hit the enter key to advance one line
- hit the space bar to advance a page
- hit `q` to exit
- when using less(1), press the `j` or `k` keys to scroll up and down one line at a time and `b` to scroll up a page at a time

Some examples:

- `cat -nvet prog7.c | more`
- `ls -alFR | less`
- `ps -e | more`
- `more prog3.c`

1.7.10 pwd(1)

To print the name of the current working directory use pwd(1)

1.7.11 chmod(1)

To change the access mode of the specified file(s) use chmod(1). This is a *very* important topic!! If you're not careful, anyone can look at or even modify your work or email. If a classmate takes advantage of your lack of action in this area, and it is caught with a copy of your work, you will *both* be in a lot of trouble since it is impossible to tell who cheated on whom. So pay special attention to this section. If you were to `ls -l` in your assignment 1 directory you will get the following output:

```
% cd cs241/a1
% ls -l
total 14
-rw-r--r--   1 z912345   csci         154 Jan  6 14:55 Makefile
-rw-------   1 z912345   csci        5116 Dec 18 13:56 Primes.tex
-rw-------   1 z912345   csci         921 Dec 18 13:56 primes.c
```

Lets look at the first ten characters.

Position(s)	Meaning
1	Either a regular file (-) or a directory (d)
2-4	The permissions for the owner (r) read (w) write (x) execute
5-7	The permissions for the group rwx is the same
8-10	The permissions for other [or world] rwx is the same

Now for a little binary:

000 = 0	001 = 1	010 = 2	011 = 3
100 = 4	101 = 5	110 = 6	111 = 7

Think of the 1's as **on** and the 0's as **off**. Now break up the characters 2-10 into three groups of three of (r)ead, (w)rite and e(x)ecute permissions for your files or directories. For example, if you want to have read, write, and execute permissions for yourself and only execute permissions for group and others you would want the following:

```
-rwx--x--x
 111001001
```

If we break the bits into groups of three, that gives us 111, 001, and 001 (in binary) and 7, 1, and 1 in octal.

So, to tie it all up:

- `chmod 711 prog1.c` will perform the above example
- `chmod 700 prog2.c` will only allow access to you, the owner
- `chmod 644 public.letter` will give you read and write and the group and others will get only read access

Note that you need not worry about *any* of this if you do a `chmod 700` on your home directory. This is because setting the permissions on your home directory such that only you can access and minipulate it will prevent anyone else from looking at it, listing the files in it or anything else to it or its contents.

1.7.12 diff(1)

Use diff(1) to see if two files are identical and to summarize any differences.

You might be asked to have output that *exactly* matches the instructor's output. They might refer to this as a "clean diff" because, if the two files are exactly alike, there will be no differences listed and prompt will immediately reappear. If they are not identical (even characters you can't see such as spaces and tabs count) then you will get a message saying as such (see cat(1) to help you tell). The lines that are different will appear with a `<` with the text from the first file and a `>` with text that differs in the second file.

For example, if you had the following data saved in file `fn1`:

```
xxx
yyy
zzz
The cat  and the dog slept.
a
 b
  c
```

and the following saved in file `fn2`:

```
xxx
yyy
zzz
the Cat and the dog slept.
a
 b
  zdfgsdfgsdfg
```

The command `diff fn1 fn2` would result in the following output:

```
4c4
< The cat  and the dog slept.
---
> the Cat and the dog slept.
7c7
<   c
---
>   zdfgsdfgsdfg
```

As you can see, there were two lines that differed in the files and they are on lines 4 and 7.

1.7.13 I/O Redirection

I do not doubt that you will be doing some redirection for some of your programs. Redirection can be from standard input, standard output and standard error. In this part we will also look at appending to files. Note that the redirection of I/O is documented in csh(1) because it is performed by the Unix command line shell.

Sometimes you will be given a data file to test your program. You use it by running your program (with its command line options) and then using the less-than symbol '`<`' followed by a filename that contains the data to send to the running program. This prevents you from having to type it on the keyboard each time you run your program.

To redirect the data in the file named `data1` into `prog1`

```
prog1 < data1
```

Sometimes, rather than printing on the screen, you will want to save your output in a file for later use. To do this, you redirect standard output to a file using the greater-than symbol '`>`'.

The operating system will automatically create the file for you if it does not already exist. Caution: if the specified output file already exists doing this command will erase the contents of the file and fill it with the new output.

To save the output of `prog2` into a file called `out1`

```
prog2 > out1
```

To *append* data to the end of a file you would do the following:

```
prog2 >> $HOME/241/out
```

You may also redirect the standard output *and* the standard error to a file. (You may be asking what is standard error? It's a seperate output stream that is used, by convention, by programs when they are printing error messages). One very handy use is to redirect the errors you get from the c compiler to a file. Then you can print it out. This will prevent you from wasteing the time of writing it down in the event that you wish to take it with you to study elsewhere.

```
cc prog1.c 2> prog1.err
```

If you want to *append* to the end of the output file, you would again use a double-greater-than symbol like this:

```
cc prog1.c &> prog1.err
```

Rather than redirecting the output of a command into a file, you might want to redirect into the input of another program. To do this, you use the pipe symbol. For example, if you want to sort the output from the command `xyzzy` you would pipe it to sort(1) like this:

```
xyzzy -l | sort
```

Note that the standard input for the `xyzzy -l` is not altered and will be the keyboard. The standard output of the `sort` is also not altered and will print to the screen.

You can also mix redirection. For example, if you want your prog to take data from a file and put the output into another file you do the following:

```
prog6 < datafile > output
```

Or you might even want to do something like this:

```
prog6 < datafile | sort > output
```

1.8 Text Editors

When you write your programs you have many editor options. The most popular Unix text editors are vi(1), pico(1), and emacs(1). If you are interested in using emacs(1) I sugest purchasing an O'Reilly book and budgeting some time to learn it. It is very powerful, but it requires a significant understanding of its workings in order to use. vi(1) and pico(1) are more suitable for new Unix users.

1.8.1 vi(1)

This editor is midway between pico(1) and emacs(1). Once you get the hang of it you can edit more code per-keystroke than you can with pico. If you commit 2-4 hours to just learning how to use it, you will probably save yourself twice that time in just typing over the course of the semester. Of course, if you try to use it *without* taking the time to learn it, you will probably *loose* 50+ hours work over the course of the semester.

To use vi(1) to edit a file named `primes.c`, you run it at the command line like this:

```
vi primes.c
```

And vi(1) will create the file if it does not exist and then allow you to edit its contents.

vi(1) uses a dual-mode operation. At any point in time, you are either in *command* mode or *insert* mode. When you first start up, you are in command mode.

Whilst in command mode, you may edit or save your file. To save your file, you type `:w` and press return. And you may exit the editor with `:q`. To edit your file, you need to be able to move the cursor about and insert and delete text.

To move the cursor around, you may use the arrow keys, or the following keys when in command mode:

- `j` move down one line
- `k` move up one line

- `h` move to the left one character
- `l` move to the right one character
- `b` move back one word delimited by punctuation
- `B` move back one word delimited by whitespace
- `w` move to the right one word delimited by punctuation
- `W` move to the right one word delimited by whitespace
- `H` move to the top of the screen
- `L` move to the bottom of the screen
- `M` move to the middle of the screen
- `G` move to the end of the file
- `1G` move to the first line in the file
- `123G` move to line 123 of the file
- `0` (the number zero) move to the begining of the current line
- `$` move to the end of the current line
- `^Y` leave the cursor where it is, but scroll the screen up one line
- `^E` leave the cursor where it is, but scroll the screen down one line
- `^U` scroll the screen up 1/2 a screen-full
- `^D` scroll the screen down 1/2 a screen-full
- `^B` scroll the screen up/back a whole screen-full
- `^F` scroll the screen down/forward a whole screen-full
- `^G` tells you where you are and about the file you are editing
- `%` if the cursor is on a '`(`', '`)`', '`{`', or '`}`', the cursor will be moved to the matching element (includes smart matching so that you can properly navigate things such as `{{{(sadf)(((sadf)sdf)asdf)}{()(())}}}` This is *extreemly* useful when fixing the nasty "mismatched '`{`' " compiler errors.

Once you get to where you are going, you may enter insert mode and start typing new text by pressing the `i` key. When in insert mode, anything you type will go into the file. To get out of insert mode, press the `esc` key. The following list shows all the ways to get yourself into insert mode:

- `a` move the cursor to the right one character and enter insert move
- `A` move the cursor to the end of the line and enter insert move
- `o` open a new line below the current position and enter insert mode
- `O` open a new line above the current position and enter insert mode
- `I` move the cursor to the begining of the current line and enter insert mode
- `cw` delete the text from the current cursor position to the end of the word and enter insert mode (change word)

- `c$` delete the text from the current cursor position to the end of the line and enter insert mode (change change to end)

If you did something you do not like, you may enter command mode and press `u` to undo it. If you do not like what you have undone, press `u` again and undo your undo. Note that there is only one level of undo! You can not undo more than one thing. If you find that you have gotten into vi(1) and done more damage than good to your file, you may abort your editing session by getting into command mode and typing `:q!` to discard all unsaved changes to your file.

To delete things from your file, you may use the backspace when in insert mode. When in command mode, you may use the `x` command to delete the character under the cursor, or you may use the `d` command along with a cursor movement command to indicate what you want to delete. For example:

- `dd` deletes the physical line under the cursor
- `2dd` deletes the physical line under the cursor as well as the next one (two total)
- `11dd` deletes the physical line under the cursor as well as the next ten lines (eleven total)
- `dw` deletes the word starting at the cursor and continuing to the next word delimited by punctuation
- `dW` deletes the word starting at the cursor and continuing to the next word delimited by space
- `d%` deletes the `{()}` under the cursor and everything up the matching `{()}`
- `d$` deletes the character under the cursor and everything to the end of the line
- `x` deletes the character under the cursor
- `X` deletes the character under the cursor and backspaces

To take two seperate lines and make them into one big one, place the cursor on the first of the two lines and press `J` to "join" them.

To move text from one place in your file to another, you may delete it or yank it and then paste it. Use above delete commands if you want to remove the text and relocate it, or you may leave the text in place and make a copy of it by yanking it. To yank text, you use the `y` command the same way you use the `d` command. For example, `yy` will yank one line and `5yy` will yank five lines… Then position the cursor where you want to paste it and press `p`.

To search for things in your file, you enter command mode and then you type `/` followed by what is called a regular expression that describes what you are looking for. A regular expression includes the identity set for anything spelled with letters and numbers, so you can easily type something like `/main(` to jump to the start of the main function in your program… provided that your open-parenthesis is pressed against the functions name. If you use a consistant style in your coding, this mechanism allows you to fly around your source and edit it with great ease.

The biggest problem with vi(1) is that if you are using it on a system supporting a mouse and copy/paste operations (which is recommended), you may accidently paste some text that you want to insert into your file when you are in command mode. As you can see by the above command description (and there are many more commands) there are several ways to delete things from your file. Odds are good that you will destroy your file by pasting stuff into command mode. Consider the consequences of pasting a `d1000` into command mode followed by a `d1`. You can use the undo to undo the `d1`, but that is it... the `d1` is the last command! The moral of this??? Be way careful about pasteing into vi(1).

I strongly suggest that if you want to use this editor that you check out the man page and practice before you are in a hurry. Consult your TA, instructor, or the course web site for additional information about vi(1).

1.8.2 pico

This editor is simple to learn and operate, but it takes more keystrokes per-edit than vi(1) or emacs(1) to use. This simplicity, however, makes it safer than the more advanced editors because you can't do so much damage per-keystroke when you don't know what you are doing. You run pico(1) like vi(1):

```
pico myfile.c
```

Unlike vi(1), pico(1) does *not* operate in a dual-mode style and thus may be more suitable for beginners. Control characters are used to move the cursor about and to enter commands.

- `^Y` moves up one screen
- `^V` moves down one screen
- `^A` moves cursor to the beginning of the line
- `^E` moves cursor to the end of the line
- `^D` deletes the current cursor position
- `^K` deletes the current line
- `^U` undelete
- `^X` saves the file
- `^W` search for a specific word

To move blocks of text around use the `^K` for each sucessive line you want to copy. Then move the cursor to the location that you want the text to go and use `^U` to put it there. For those that don't want to invest time in learning (which would cause me to ask the question, "why are you in school?") about better editors that can save you time in the long run, this editor is the one for you.

1.8.3 Other Editors

For those of you who want to be able to do your programs off campus and don't want to waste phone time just typing code, you can use any editor you want. I used DOS Edit to do all of my 440 progs then I used ftp(1) to send the files to my Unix account at NIU. You can also use word or wordperfect only you must be sure to save the file as an *ascii* file! When you are typing your program on another editor just use the basic stuff no bolding, special fonts, watermarks. . .

1.9 Termination and Output

I have no doubt that everyone has run into an endless loop at one time or another. Here is what you do about it in unix.

1.9.1 Job Control

Job control is used to run more than one command at a time. In order to run more than one command at a time, you place an ampersand '`&`' at the end of your Unix command. This can be helpful especially if the command will take a long time to run.

```
% prog2 &
```

One handy thing to do is to hit `^Z` while in an application like pico(1) or vi(1). This will put the current application in the background and give you the prompt to do other things.

To find out what jobs/processes you have going, type `jobs -l` at the prompt and it will list the jobs and their statuses:

```
[1] + 12345 Running     loop
[2] - 54321 stopped     loop2
[3]   32145 stopped     pico prog1.c
```

The number inside the `[]` is just the job number, the next number is the process identification number of the job, next column tells about the state of the job, and the last column is the name of the process. The `+` stands for the current job and the `-` stands for the previous job.

As you can see 2 and 3 are in the background and 1, the loop, is running in the forground.

1. To bring job 3 into the forground type `fg %3`
2. To put job 1 into the background type `bg %1`
3. To kill job 1 just type `kill -KILL %1`
4. (or) To kill job 1 just type `kill -KILL 12345`
5. To kill the current job just type `kill -KILL %`
6. Of course you can also use `^C` to kill forground jobs.

1.9.2 kill(1)

Use kill(1) to send a signal to a process and let it know that it should be stopped. To make sure that the process will die use `kill -KILL` *(processid or job number)*. This is usefull to kill an unwanted endless loop. Depending on what you have gotten yourself into, you may have to login on another terminal to do this. **Note that if the system administrator has kill your programs it tends to make them cranky... to put it nicely**. `:-)`

Chapter 2

Designing Debugging Into Your Programs

As the semester progresses, you will learn that when writing complex programs, you have the design debugging directly into your program from the begining. The following function is used just like a printf(3c), but by changing the value in `debugLevel`, you can tell the debug function if it should print or not. This way, you can leave your debugging statements in your final program and simply tell it not to print any of the debugging output.

Why would you want to do this? Well, few programs are ever perfect. It is likely that you will debug, run, debug, run, debug, run... and will not want to have to add, delete, add, delete... statements that print debugging output all the time.

The way you would use this function is copy it into your assignment and then use `debug` instead of `printf` any time you would want to add a print statement to display some debugging output. To turn debugging on and off, you will want to change the value assigned to `debugLevel`.

```
#include <stdarg.h>
static void debug(int level, char *format, ...)
#ifdef __GNUC__
    __attribute__((format(printf, 2, 3)))
#endif
;

int debugLevel = 0;

/**********************************************************
*
* A custom version of printf() that only prints
* stuff if debugFlag is at or above the given
* message level.
*
**********************************************************/
static void debug(int level, char *format, ...)
{
    va_list ap;
    if (debugLevel >= level)
    {
        va_start(ap, format);
        (void) vfprintf(stderr, format, ap);
        va_end(ap);
    }
    return;
}
```

Some examples of how to use the above `debug` function:

```
debug(90, "I am in the ***** function\n");
debug(50, "The answer before the while is: %d\n", ans);
```

The first example will only print its output if the current value of `debugLevel` is greater 89. The second will only print if `debugLevel` is greater than 49. Using a numbering convention will allow you to increase or decrease the volume of debugging output detail... and if you only use positive numbers in your `debug` statements, setting `debugLevel` to zero will turn them *all* off.

What I normally do is define a command-line parameter `-d` that can be used to specify the value for `debugLevel` so that I can change it on any run of the program like this:

```
myprog -d50 other parms go here
myprog other parms go here
myprog -d1000 other parms go here
```

Chapter 3

Using the gcc(1) Debugger gdb(1)

You can use this debugger to help you debug your programs. This is not an exaustive list of what you can do with it but it is a good place to start.

When compiling your code you must include `-ggdb` option. This will put the debugging info in. Your command might look something like:

```
gcc -lm -Wall -ansi -pendantic -ggdb -o prog1 prog1.c
```

To start the debugger you type `gdb prog1`. This will give you the prompt `(gdb)`. You must reload for every time you recompile your program.

1. General Stuff
   - file executable file – Loads debugging info.
   - `quit` – Will quit the debugger
   - `help` – Will give the help options
   - `help topic` – will give the help on that specific topic
   - Running (help running)
     - `run` – Runs until break point, abends, or normal termination
     - `run < prog.dat > output` – Will let you use redirection
     - `continue N` – Continue execution after the breakpoint. N is the number of times to ignore this breakpoint, and N is optional.
     - `step N` – Execute next line of code. This will trace into your function calls. N is optional and is the number of lines to be executed.
     - `next N` – execute next line. If it is a function call execute function, but do not trace into the function.
     - `kill` – to stop the current program
     - `set args arg1 ‘‘arg2’’` – sets the arguments. Strings must be in quotes. Arguments stay set until you type set args by itself.
   - Examining Data (help data)
     - `whatis Expression` – Will tell you what it is. Expression can be a variable name or `p -> member`.
     - `print Expression` – Prints the current value of the expression. Note that `NULL = 0x0`
     - `printf "format string", arg1, arg2` – Gives formated print
     - `display expression(s)` – Print expression every time the prog pauses execution display also assigns a code number for each expression starting at 1.

  - `undisplay code \#'s` – Delete display items. If no code number is given the default is all are deleted. The code #'s are separated by a space.
  - `enable display code \#'s` – Switch them on.
  - `disable display code \#'s` – Switch them off.
  - `info display` – Will show you the list of expressions with their corresponding code #'s.
- Setting Break Points You have to set at least one break pt to be of any use
  - `break line \# -or- function name` – Will clear all break points at that place
  - `delete code \#'s` – Defaults to all if no code number is given
  - `enable breakpoints code \#'s` – Self explanitory
  - `disable breakpoints code \#'s` – Self explanitory
  - `info breakpoints` – Shows the list of breakpts
  - `enable once` – Enable them once then disable
  - `enable delete` – Enable once then delete the breakpoints
  - `code \#` (condition) – If true then enable else disable. See help condition
- Listing Program Code
  - `list function name` – Frst 10 or so lines are printed. List will default to where you are now
  - `watch` – Runs program until the value changes slows things down.