# Programming with Unix Reference

James Wadsley

## 1 Basic Unix

You can get help on any UNIX command or program in the online manual:

| | |
|---|---|
| **man** *text* | Get manual page for command, program, thing called *text* |
| **apropos** *text* | Looks for manual pages with *text* in the short description |

A good reference book:
*Linux in a nutshell*, Jessica P. Hekman, (O'Reilly)
This book has all the common commands including info about tcsh and editors vi and emacs. There are other "in a nutshell" books for variants of Unix but they are more or less equivalent.

A good longer web reference:
`http://x86.cs.duke.edu/csl/docs/unix_course/intro-1.html`

## 2 Editors

Several editors are available. You can cut and paste text to and from xterm terminals with the emacs, xemacs or gedit.

| | |
|---|---|
| **emacs, xemacs** | Pop-up X-Window GNU editor. Very powerful but a little quirky if you are only used to windows editors. Very popular on unix/linux systems for text editing and programming. Each version has buttons for common operations (e.g. save, load, cut, paste, etc...) which make it easier to use for someone new to emacs. |
| **gedit, mousepad** | Pop-up X-Window generic editor associated with GNOME window manager and Xfce desktop environment (also **kate** in KDE). Gedit is very easy to use. Mousepad has graphical buttons for standard operations (e.g. save, load, cut, paste) and are fairly intuitive to use. You can use C-x, C-c, C-v cut, copy paste like Windows. Note: In preferences you can make it show line numbers which is very helpful! |
| **gvim, vim, vi** | A keyboard based editor that is powerful but tricky to use. It has been a unix standard tool for a very long time. It has particularly powerful regular expressions. |

### 2.1 User and Machine Info

**ssh** *username@machinename* Secure login to remote machine

| | |
|---|---|
| **whoami** | What user am I logged on as? |
| **hostname** | What machine am I logged on to? |
| **who** | Who are the other users logged on? |
| **passwd** | Change current user password |
| **exit** | exits from a shell or terminal |
| **ping** *machinename* | Is the machine *machinename* connected to the internet? |
| **date** | Display current time (according to machine clock) |

## 2.2 Shells: Entering commands with TC shell (tcsh)

tcsh is the program that interprets things you type into a terminal. It's overall behaviour can be controlled by setting internal variables. Help for commands that are built into the shell rather than separate programs can be found by looking at the tcsh manual: **man tcsh**.

| | |
|---|---|
| **.cshrc** | Start-up script file run when a new shell (terminal) started |
| **.login** | Start-up script file run on login |
| **source** *filename* | Run commands in file *filename* |
| **set** | List shell internal variables (e.g. prompt, history. autologout) |
| **set** *varname*=*value* | Set shell variable *varname* to equal *value* |
| **set prompt='***text***'** | Set shell prompt |
| **set ignoreeof** | Don't logout with CTRL-D |
| **set noclobber** | Don't allow redirection to overwrite a file |
| **limit, unlimit** | List or change personal limits on files and programs |
| **env** | Print my environment variables ( e.g. PATH, SHELL ) |
| **setenv** *varname value* | Set environment variable *varname* equal to *value* |
| **alias** | List and set command aliases (shorthand for commands) |
| **unalias** | remove a command alias |
| **which** *command* | Show program file (or alias) associated with *command* |
| **whereis** *program* | List executable program location, source and manual page for *program* |

### 2.2.1 Command Line editing, history

tcsh allows emacs style editor commands on the command line.

| | |
|---|---|
| **CTRL-A** | Go to beginning of line |
| **CTRL-E** | Go to end of line |
| **CTRL-K** | Delete to end of line |
| **CTRL-D** | Delete character or list possible completions if at end-of-line |
| **CTRL-C** | Abort command input line and start new empty line |
| **TAB** | Complete command or file based on what I have typed so far |
| **history** | Show history of commands entered |
| **!1** | Rerun command number 1 from the history |
| **!***text* | Rerun last command starting with *text* |
| **UP** | Get previous command from history |
| **DOWN** | Get next command from history (or blank line if at the end) |

### 2.2.2 Running programs interactively

Commands may be built into the shell or a separate program (usage is the same).

| | |
|---|---|
| *progname* | Run a program or command called *progname* |
| *progname* **&** | Run a program or command called *progname* in the background |
| **CTRL-S** | Suspend output to/input from terminal |
| **CTRL-Q** | Resume output to/input from terminal |
| **CTRL-Z** | Suspend a program in the foreground (Gives you back the command prompt) |
| **CTRL-C** | Kill program in the foreground (Gives you back the command prompt) |
| **CTRL-D** | EOF (end of file), logs you out if there is nothing else on the line |

| | |
|---|---|
| **jobs** | List jobs running associated with this terminal with status |
| **kill %1** | Kill program listed as number 1 from job listing |
| **fg %1** | Bring the program to the foreground |
| **bg %1** | Run program in the background |
| **ps** | List my processes on the machine |
| **top** | Interactive tool to look at all running programs (q to quit) |

### 2.2.3 Redirecting input and output

Many commands expect you to type input into the terminal and print their output to the terminal window. You can modify this behaviour to take input from or send output to files or other programs instead.

| | |
|---|---|
| *prog1* **;** *prog2* | Run *prog1* then run *prog2* |
| *progname* < *infile* | Run *progname* and take input from file *infile* |
| *progname* > *outfile* | Run *progname* and put output in a file *outfile* instead of the screen |
| *progname* >> *outfile* | Run *progname* and append the output to the file *outfile* instead of the screen |
| *prog1* \| *prog2* | Run *prog1* and use output as input for *prog2* |

## 2.3 Files

These programs manipulate files and directories specifically. Any command that manipulates files can be given **wildcards** as arguments instead of a specific filename (matching is done by tcsh). The filename matching set is smaller and slightly different to the full regular expressions set:

| | |
|---|---|
| **?** | match any single character |
| ***** | match any zero or more characters |
| **[abc]** | match any of the characters enclosed |
| **[a-d]** | match any character in the enclosed range |
| **{abc,defg}** | match any substring in the list, e.g. abc or defg |

Directories:

| | |
|---|---|
| *dir1/file* | File *file* in subdirectory *dir1* |
| **~/** | My home directory |
| **./** | This directory |
| **../** | One directory up (**../../** is two directories up, etc...) |

File Manipulation Utilities:

| | |
|---|---|
| **ls** | List files in current directory (many useful options, e.g. **ls -alt**) |
| **ls** *files...* | List files if they exist (most useful with wildcard expressions) |
| **ls** *dirname* | List files in directory *dirname* |
| **pwd** | Current directory name |
| **cd** *dirname* | Change directory to directory *dirname* |
| **mkdir** *newdir* | Make a new directory called *newdir* |
| **rmdir** *deaddir* | Remove exisiting directory *deaddir* |
| **cp** *file1 file2* | Copy *file1* to *file2* |
| **cp** *file1 file2 ... dir1* | Copy the files *file1 file2 ...* into the directory *dir1* |

**scp** *file1 myusername*@**phys-ugrad.mcmaster.ca:**

| | |
|---|---|
| | Secure (remote) copy to phys-ugrad.mcmaster.ca(**man scp**) |
| **sftp** | Secure remote file transfer utility |
| **mv** *file1 file2* | Move *file1* to *file2* (like rename) |
| **rm** *file* | Delete file *file* |
| **find** | Find files matching a description |
| **locate** *text* | Find files with names containing *text* |
| **chown, chmod, chgrp** | Commands to modify file status/attributes |
| **df, du** | Show how much space is used on a disk (df) or in files (du) |
| **gzip, gunzip** | Compress or uncompress file utilities |
| **tar** | File archive utility |

## 2.4 Text file utilities

These tools often useful to deal at large amounts of output:
e.g. **program_bigoutput | less**

**Regular expressions** are quite important in unix: (e.g. grep)

| | |
|---|---|
| **.** | match any single character except <newline> |
| ***** | match zero or more instances of the single character (or meta-character) immediately preceding it |
| **[abc]** | match any of the characters enclosed |
| **[a-d]** | match any character in the enclosed range |
| **[^exp]** | match any character not in the following expression |
| **^abc** | the regular expression must start at the beginning of the line (Anchor) |
| **abc$** | the regular expression must end at the end of the line (Anchor) |
| \ | treat the next character literally. This is normally used to escape the meaning of special characters such as "." and "*". |
| **{n,m}** | match the regular expression preceding this a minimum number of n times and a maximum of m times (0 through 255 are allowed for n and m). The { and } sets should be thought of as single operators. In this case the preceding the bracket does not escape its special meaning, but rather turns on a new one. |
| **<abc>** | will match the enclosed regular expression as long as it is a separate word. Word boundaries are defined as beginning with a <newline> or anything except a letter, digit or underscore (_) or ending with the same or a end-of-line character. Again the < and > sets should be thought of as single operators. |
| **(abc)** | saves the enclosed pattern in a buffer. Up to nine patterns can be saved for each line. You can reference these latter with the \n character set. Again the ( and ) sets should be thought of as single operators. |
| **\n** | where n is between 1 and 9. This matches the nth expression previously saved for this line. Expressions are numbered starting from the left. The \n should be thought of as a single operator. |
| e.g. **[^a-zA-Z]** | any occurrence of a non-alphabetic character |

Text utilities:

| | |
|---|---|
| **cat** *file* | Show contents of *file* all at once |
| **more** *file* | Show contents of *file* one page at a time |
| **less** *file* | Show contents of *file* one page at a time, more features |
| **head** *file* | Show first 10 lines (**head -20** *file*, first 20 lines) |
| **tail** *file* | Show last 10 lines |
| **grep** *regexp files...* | Look for instances of **regular expression** *regexp* in *files* |
| **diff** *file1 file2* | Look for differences between *file1* and *file2* |
| **sort** *file* | Sort contents of *file* and output result to standard output (screen) |
| **wc** *file* | Count lines, words and characters in *file* |

Quick summary for using **less**:

**Enter** moves forward one line, **SPACE** or **CTRL-F** goes forward one page and **CTRL-B** back one page. **q** quits. You can search for *text* using */text* **RET** and hit **n** to go to the next match in the file.

# 3 Compiling C++ (and C) Programs

A C source file conventionally ends in **.c**. Included files end in **.h**. These are added to a **.c** program at compile time. C++ sources files usually end in **.cpp**. Unfortunately the convention is not universal – you also see **.C**, **.cxx** and **.cc** sometimes. Included files are **.h** (same as C) or **.hpp** (or **.hh** or **.H** which we do not recommend).

The default C++ Compiler which is run using **c++**. Our default is the GNU C++ compiler, accessed directly as **g++**. The GNU compilers can tell what language you are using by the ending of the file: **.cpp** for C++ and **.c** for C. However to correctly link to make a runnable program you should use **c++ or g++** for C++ and **cc or gcc** for C respectively.

Non-commerical versions of the Intel compilers are also available for free use for linux. Full versions of Cygwin also include compilers and there are free compilers available for Mac/OSX. Microsoft also makes its Visual Studio compilers freely available now.

Note that linking and compiling can be done in one command but it better practice to separate it into two steps and better still to use a **Makefile**. Note that many scientific programs need math functions (sqrt, sin) and for C/C++ that means you need to include the math library (libm). So to link and make an executable you use the -lm option. Modern versions of g++ often link it automatically.

| | |
|---|---|
| **Compiler:** | |
| **c++** *[options] file1 [file2 ...]* | General usage for GNU C++ Compiler |
| **c++** –**help** | List all options |
| **c++** *sourcefile.cpp* **-o** *progname* | Compiles the source code file *sourcefile.cpp* and produces an executable called *progname*. |
| **-c** | Compile to object file (e.g. sourcefile.o) rather than an executable program |
| **-O2 -O3** | More agressive optimization options |
| **-g** | Option to include debugging symbols |
| **-I***includepath* | Look in directory *includepath* for included files |
| **Linking:** | |
| **c++** *sourcefile.o* **-o** *progname* | Link object file *sourcefile.o* and produces an executable called *progname*. |
| **-l***libname* | Link in library called *libname* |
| **-L***libpath* | Look in directory *libpath* for libraries |
| **Debuggers:** | |
| **gdb** *progname* | Run GNU Debugger on program *progname* |

# 4 Make

The **make** command is a utility that builds a program by compiling and linking the necessary files into an executable. It relies on a Makefile to describe which files are needed to compile a program. The make utility determines which files have changed and recompiles only those files that are affected by the changed files. Makefiles organise a coding project that potentially consists of many files. To use make:

| | |
|---|---|
| **make** | Make the default target program |
| **make** *target* | Make a specific target |
| **make clean** | Clean away unnecessary files (Fancy version) |

You can google for help with make. A detailed reference is at:
`http://www.gnu.org/software/make/manual/make.html`

## 4.1 Makefile syntax

```
NAME=value

Target1: dependency1a dependency1b ...
<TAB> Action1a \
      Action1b

Target2: dependency2
      Action2
```

The space before actions is a TAB key. Spaces do not work. If a line must be continued it should be ended with a \ and then continued on the next line.

A Macro is like a variable where the NAME is a shorthand notation for a block of text used several times. It means you can change something in one place and have it used for the whole Makefile. To substitute in the value of a macro within a Makefile use `$(NAME)`.
e.g.

```
BINDIR = /home/wadsley/bin
...
       c++ -O2 prog1.cpp prog2.cpp -o $(BINDIR)/prog
```

**Target** is something to be made (e.g. compiled or linked). **Dependencies** are lists of files that are used to make the Target: if any of these are changed, the target should be recompiled. These include source files and modules for object files. **Actions** are the commands that create the Target.

Macros with special meanings:

| | |
|---|---|
| **$*** | Filename of target without suffix (e.g. source from source.cpp) |
| **$@** | Target name |
| # | Comment line: all text following the # is ignored |

## 4.2 Example Makefiles

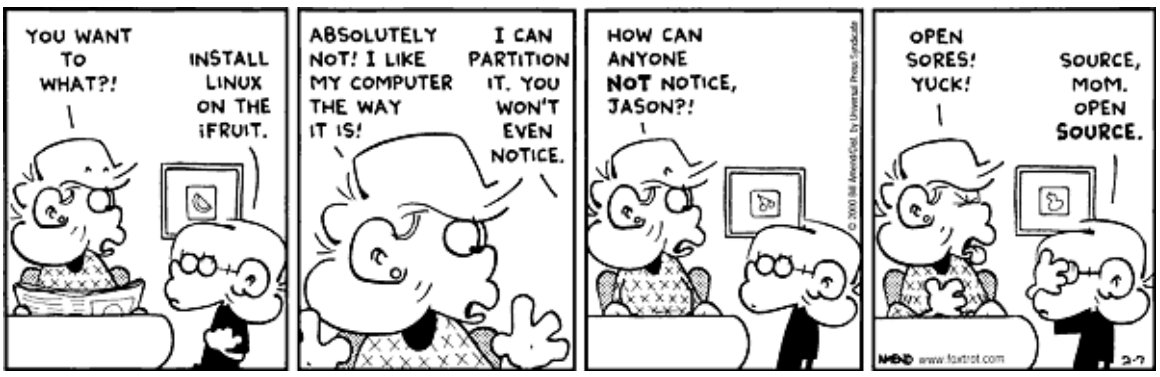
### 4.2.1 Basic Makefile

```
# Basic Makefile to make program ctest
ctest: code1.o code2.o
      c++ code1.o code2.o -o ctest
```

```
code1.o: code1.cpp
      c++ -c code1.cpp

code2.o: code2.cpp
      c++ -c code2.cpp
```

#### 4.2.2 Fancier Makefile for the same program

```
# Fancy Makefile to make program ctest
CPP=c++
LINKER=c++
CFLAGS=-O2
#lm to include math functions
LFLAGS=-lm
OBJECTFILES=code1.o code2.o

default: ctest

ctest: $(OBJECTFILES)
      $(LINKER) $(LFLAGS) $(OBJECTFILES) -o $@

$(OBJECTFILES):
      $(CPP) $(CFLAGS) $*.cpp -c

code1.o: code1.cpp code1.h general.h
code2.o: code2.cpp general.h

clean:
      rm *.o
```



# 5 Debugging

When a program has an error it often crashes without telling you much (e.g. SIGSEGV or OVERFLOW). You can run the program within a debugger program to find out more about why it crashed, including the exact place in the program it crashed, what functions have been called to get there and the value of any variable in the program. This is typically much more efficient than adding lots of *print* statements to your program and running again.

To use the debugger you should compile with **-g** (see compiling above). This includes information to allow the debugger to link variable names and line numbers in your source code (the **.cpp** files) to values and locations in the program at run-time.

To start the debugger (always in the foreground):
**gdb** *programname*

**Debugger commands:**

| | |
|---|---|
| **q** | Quit the debugger |
| **help** | Get help on topics, e.g. **help where** |
| *Stopping and starting the program* | |
| **r** or **run** | Start the program running. |
| **Control-C** | Stop the program. |
| **break** *file.cpp*:*line* | When the program is run,<br>stop at the code associated with this *line* number in the *file.cpp* |
| **break** *functionname* | When the program is run,<br>stop in the function called *functionname* |
| **cont** | Continue running the program. |
| **next** | Run the next line only.<br>For a call, go to functions and come-back. |
| **step** | Run the next line only.<br>For a call, go into the function and run the first line. |
| *Determining where you are* | |
| **where** | Print a list of functions called to get to where the program stopped. |
| **up** | Go back one call to the previous function. |
| **down** | Go forward one call. |
| *Looking at variables* | |
| **print** *var* | Print the value of variable with name *var*. |
| or **p** *var* | You can only print variables in the current function.<br>You can use up and down to look at earlier functions. |