A USER'S GUIDE TO
Tools for C Programming
IN THE MATH AND COMPUTER SCIENCE DEPARTMENT
Version 0.9T

Paul English
Patrick E. O'Neil
Elizabeth J. O'Neil
Elizabeth Schwartz
Tsutomu Ishikawa
and others

March 21, 2008

1 Table of Contents

Contents

1 Table of Contents **2**

2 Read me first **3**

3 Editors **3**
- 3.1 The vi Editor . . . 3
  - 3.1.1 Command Mode . . . 4
  - 3.1.2 Insert Mode . . . 4
  - 3.1.3 Last Line Mode . . . 5
- 3.2 The vi Editor - more advanced commands . . . 5
- 3.3 Emacs . . . 6

4 Mail **7**
- 4.1 BSD-Style Mail . . . 7
- 4.2 Advanced Mail . . . 8
- 4.3 Getting mail from the POP server . . . 9

5 The C Programming Environment **10**
- 5.1 The gcc Command: Compiling C Programs . . . 10
- 5.2 The gdb Command: Debugging C Programs . . . 11
- 5.3 The gdb Command - more advanced commands . . . 14

6 The script command **17**

7 Unix file protection **18**

2 Read me first

This GUIDE is for Unix systems users at the Computer Science Lab of the Department of Math and Computer Science at UMASS/Boston. Computing Services Unix systems (used for CS110 and CS210) are **not** covered. This GUIDE was originally written for students in CS240, but we hope it will be useful to everyone.

This GUIDE TO TOOLS is designed to accompany the CSGUIDE. If you do not have a copy of the CSGUIDE see the operators in room 157.

3 Editors

There are two important editors in use on the Unix System. These are `vi` and `emacs`. There is also a simple editor called `pico`. This editor is not powerful enough for class use but it might help for the first few weeks. Finally, the `ed` editor is a very simple line editor, which does not require full-screen capability. This was once useful for printing paper terminals, but the only reason to know it now is that `vi` is based on `ed` and some manual pages refer to it.

The `emacs` editor is the most powerful one, but rather complex to master, and not available at many sites. Some instructors recommend `vi` for new users, and others require `emacs`.

3.1 The vi Editor

The `vi` editor has a tremendous number of commands (too many) This article explains only the basic commands. There's also a list of less frequently used commands, but this is still only a small fraction of the total that exist. Don't worry though; most people who use vi never learn most of the commands and never miss them.

You can create and edit a file with the name "fname" by typing:

```
eris% vi fname
```

If the file "fname" does not already exist, an empty file with that name will be created. The `vi` editor has three major modes of operation: **command mode**, **insert mode** and **last line command mode**. You enter the `vi` editor in command mode, and your screen will show a window on the first page of existing text in the file.

3.1.1 Command Mode

In command mode, you can move the cursor around using the ”arrow” keys, or perform any of the following actions by typing the character string commands WITHOUT TYPING A RETURN (you will not see the characters, only the action):

```
command          action

[n]x             delete the [next n] character[s] under the cursor
[n]dd            delete the [next n] line[s] under the cursor
[n]G or :n         move cursor to line number n
G                move cursor to last line of the file
i                enter insert mode, before character under the cursor
                 (need "i" to insert at beginning of line)
a                enter insert mode, after character under the cursor
                 (need "a" to insert at the end of line
o                enter insert mode on NEW LINE created below current
                 (need "o" to insert new last line in file)
O                enter insert mode on NEW LINE created above current
                 (need "O" for the first line in file)
:                enter last line command mode
ZZ               save file and return to UNIX
```

Some of the commands may be preceded by an integer, symbolized by ”[n]” in the above, which repeats the command action “[n]” number of times. Thus, typing `x` three times deletes the next three characters or type `3x` to do the same thing with a single command. The command `3dd` will delete three lines, starting at the current line.

3.1.2 Insert Mode

In insert mode, everything you type will be placed in the text. To leave insert mode and return to command mode, use the `ESC` key.

3.1.3 Last Line Mode

The colon character, "**:**", will bring you from command mode to last line command mode. The colon will appear at the beginning of the last line along with successive character you type, and you should terminate these commands with a RETURN. Historically, last line mode is equivalent to the old `ed` editor, and you may see these called "`ed`" commands:

```
command          action

:w filename      write contents of file to file "filename"
:w               write to default filename the was opened by vi
:r filename      read file "filename" into current buffer
:q (or :q!)      quit back to UNIX without saving updates to file
```

The RETURN at the end of line returns you to the normal command mode.

3.2 The vi Editor - more advanced commands

This is still only a subset of the `vi` editor's commands.

```
command          action

/string          search forward in text to find "string"
?string          search backward in text to find "string"
J                Join the next line with the current one
                 --i.e. delete intervening newline char
[n]yy            yank (cut out copy) of next n lines
p                paste yanked lines into text on new lines
                 following current cursor
D                Delete characters to end of current line
```

There is also an "`ed`" command to change text, which can be used by `vi`, like all `ed` commands, by prefacing a colon to enter last-line command mode:

```
:m,n s/text1/text2/g
```

This will search from line m to line n and change all occurrences of text1 strings to text2.

3.3 Emacs

Emacs is more than an editor; it is a flexible, extensible command environment. Some users spend all of their time online in Emacs. Emacs has extensions to compile and debug programs, edit directories and manage files, send mail, read Usenet news, and even act as a psychoanalyst. Emacs comes with its own LISP language.

Emacs also has a tutorial, and the best way to begin using Emacs is to start Emacs and run the tutorial. Enter `emacs` and then type `CTRL-h-t` (Hold down the Control key and type CTRL-h, then release the control key and type t.)

An Emacs reference card is appended to the CSGUIDE.

4 Mail

The default mail program is the "BSD" or "ucb" mail program. This is a simple, command-line program. You will find this mail program on every Unix system, but it is rapidly being replaced by other, more user-friendly mail handling programs. Two that we support here are `pine` and `elm`. `emacs` users may also want to explore `RMAIL` but this is not recommended for beginners.

The programs `pine` and `elm` are very similar to each other, and both have on-line help. `elm` has a more sophisticated ability to sort and filter messages; `pine` handles more complex (MIME) types of mail with ease.

Emacs has a very complex mail program called RMAIL . If you invoke RMAIL , it will convert your new mail, and any mail in your "mbox" file, into an RMAIL file. To unconvert your mail back to Unix mailbox format, the command is `Meta-x unrmail`

Emacs also has a program called VM. It has the greater functionality, approximately a superset of the keystrokes, and uses standard unix mailbox format. In particular, the latter means that its mailboxes can be read with Mail, as well as Pine, Netscape, MSIE, or other pop3 or imap4 compliant mail readers. VM will move all of your mail into a file named `INBOX` in your home directory

You can also use to read your mail. Netscape will move all of your mail into a file named `nsmail/Inbox` in your home directory.

4.1 BSD-Style Mail

It is a good idea to get familiar with the regular Unix `mail` program because many systems do not have other mail programs installed.

To read your mail using the default BSD mail program, give the Unix command

```
eris%  mail
```

and you will be given a numbered list of mail messages you have received, followed by an "`&`" prompt in the mail environment, or a message that no new messages exist followed by the Unix prompt. This is confusing sometimes, as the mail commands from the "`&`" prompt are not the same as the Unix command line mail commands. You can start reading your messages from the top by typing RETURN, and then a return after each message. The most basic mail commands are:

```
help             get list of commands
t <m-l>          type message-list m-l, e.g. type 13  7-9, a sublist
                    numbered message you are told about on entering
CR               CARRIAGE RETURN - type next message in sequence
h                print out list of numbered messages (headers) again
d <m-l>          delete message-list
s <m-l> fname    save message-list to file fname
R <m-l>          reply to current message sender(s) and all recipient
r <m-l>          reply to current message sender only
m <u-l>          send mail to the user-list u-l -e.g.: "m poneil ram"
x                abort-quit (leave mail) - messages remain as they we
                   you entered mail
q                normal quit (leave mail) - messages deleted or  save
                    disappear, other messages get appended to mbox fi
```

To send mail to frequent set of recipients, you can also create an ”alias” abbreviation for a list of names in file named .mailrc in your login directory. Each ”alias” in this file should begin on a new line and have the form:

```
eris% alias twoprofs poneil ram
```

You can then give the `m` command with a user-list including the name ”twoprofs”. A long list of names should still be on a single line (which will wrap on the screen to new lines).

More information on BSD `mail` is available online in `man mail`, and there is a long section on it in the Strang book “Learning the Unix Operating System.”

4.2 Advanced Mail

Internet mail as originally designed had a major limitation: it could only transmit 7-bit ASCII text. This meant that people could not mail binary files, or even files using foreign alphabets that included ASCII characters greater than 127.

The uuencode program was written to take 8-bit material and encode it as mailable 7-bit ASCII; the uudecode program converts it to the original form. More recently, the MIME standard has been proposed to allow mailing of all sorts of material. If you receive a mail message that has several lines of header material,

followed by gibberish, it is probably either a uuencoded message or a MIME-compliant message.

See `man MIME`, `man metamail`, or `man uudecode` for more information.

4.3 Getting mail from the POP server

If you have a SLIP or PPP connection to the Internet, you can retrieve your mail from our mail machine using Netscape, Eudora, or any other PC mail client. See your software for instructions. Our pop server is at `pop.cs.umb.edu` and the smtp server is `smtp.cs.umb.edu`. Please use these aliases as they are guaranteed to work.

5 The C Programming Environment

The first course at UMass/Boston dealing with the Unix System and C Programming is CS240. The basic C texts are **The C Programming Language** by Kernighan and Ritchie ("**K&R**"), and **C, A Reference Manual** by Harbison and Steele. There are a number of procedures you will need to master. You will use the `script` command to prepare a turn-in that demonstrates the workings of your program assignments. You will use the `gcc` command to compile your program, and the `gdb` debugger to run it under interactive control to find errors in logic (debug the program).

5.1 The gcc Command: Compiling C Programs

`gcc` is the version of the `cc` command that we use in all programming classes. `gcc` is "Gnu cc", part of the "Gnu" software from the Free Software Foundation; similarly `gdb` is the Gnu debugger. To compile a C program, give it a filename ending in ".c", for example "prog.c" then compile it by typing:

```
eris% gcc prog.c
```

This creates an executable file known as `a.out`. You can see this in your directory using the ls command, but don't try to edit "`a.out`", since the format is not one that can be viewed. If you type `a.out` alone on a line, followed by a carriage return, you will execute the program. If you want to give a specific name other than `a.out` to your executable file, say the name "runfile", you can compile with the -o flag:

```
eris% gcc prog.c -o runfile
```

and now typing `runfile` alone on a line will execute the result. To be able to debug your program (as explained in the `\index{gdb}` article below), you would use the flag `-g`.

```
eris% gcc -g prog.c -o prog
```

HINT: Do **NOT** name your program "test.c" or make an executable program called "test!" The Unix shell contains a "built-in" command called `test` and this will run instead of your program!

As you will learn in Chapter 4 of **K&R**, it is possible to have different routines of a single program in different files. The different files might be named as in the example of Chapter 4: main.c getop.c, stack.c and getch.c. Then you can compile all of these files at once into an executable file named calc, allowing for debugging use, by giving the command:

```
eris% gcc -g main.c getop.c stack.c getch.c -o calc
```

This command will create "object" files in your directory, main.o, getop.o, stack.o and getch.o. These are files for which most of the compilation work has been done, but the final linked load into the executable is yet to be performed. Now if you need to modify a single one of these files to fix a bug, say stack.c, you can save a lot of time in recompiling by giving the command:

```
eris% gcc -g main.o getop.o stack.c getch.o -o calc
```

By specifying the ".o" extension (suffix) on three out of the four modules named, the compilation step is skipped, only one module is compiled, and the result is link loaded together with the existing object files into calc.

In cs240 you will learn about the `make` command, which is a program that manages compiler options and program building.

NOTE: It is good practice to ask `gcc` for more warnings than it gives by default.

```
eris% gcc -g -Wall prog.c -o prog
```

The `Wall` flag will give all (or almost all) warnings as well as errors. The make-files for cs310 give more examples of options.

5.2 The gdb Command: Debugging C Programs

The `gdb` permits you to run a program in such a way as to interactively control its progress and watch the results of its operations; in this way you should be able to detect how bugs arise. The first step in debugging a program is to compile it with the `-g` flag, as explained in the `gcc` command article above, e.g.,

```
eris% gcc -g prog.c -o prog
```

Following this, you can give the command `gdb prog`

The program is now loaded into the `gdb` environment. However, it is not yet running. Before starting it running, you will probably first wish to set up a "breakpoint", to make it stop when it reaches a particular source line. As an example, you can set up a breakpoint to make the program stop after it enters main() (a number of things can happen in running a program before the main() function you wrote is entered), and then start it running, taking its standard input from some file named infile:

```
(gdb) break main
(gdb) run < infile
```

To reduce confusion you probably want to provide interactive input to the program through an infile, so that all interaction during debugging is with the `gdb` environment. After the breakpoint is reached, `gdb` will print out the C language statement reached. You can now make the program progress through single steps, executing consecutive source lines of program logic in the normal flow of control, by giving the command:

```
(gdb) s           -- s is for "step" -- enter and step through stater
                     of function when a function call is encountered
(gdb) n           -- n is for "next" - step, but skip over function c
(gdb) r           -- run program
(gdb) r small     -- run program, and give it ``small'' as an argumer
(gdb) r < infile -- run program with input from file ``infile''.
```

As each step is executed, the corresponding source line will be printed out. You can type CR on successive lines after the first command to perform successive steps. At any time where your input is expected, you can print out the value of any variable in the scope of the current logic or an expression involving such variables (such as 3*i), by typing:

```
(gdb) p 3*i        -- p is for "print"
                   -- i is a variable in scope of current position
```

The type printed will be the type derived from the variables in the expression; if special output formats are needed, you can use the ”p/format” form for print:

```
(gdb) p/x 3*i    --  x for hexadecimal, o for octal, d for decimal, f
                      float, c for char, s for string
```

You can also get the (default type) values of all variables in your scope by typing:

```
(gdb) i lo        -- i is for "info" -- gives values of local variabl
                     current stack level
(gdb) i var       -- values of global and static variables
```

You can get help (relatively useful help messages) while in the by typing:

```
(gdb) h          -- h is for help -- list of help topics
(gdb) h topic    -- help on named topic
(gdb) h p        -- help on print command (or any other command)
```

At any time, you can type:

```
(gdb) q          - q is for "quit", i.e. Leave the debugger.
```

5.3 The gdb Command - more advanced commands

So far we have only told you how to set a breakpoint at the entry to the main program, and how to perform single steps in running the program. This is enough control only for very short programs. More generally, you need to be able to set breakpoints at arbitrary positions and continue running the program after one breakpoint until you get to another breakpoint.

```
(gdb) b 36              - break at line number 36 of current source
(gdb) b 36 if i==3      - break at line 36 of current source file if
                          variable i is equal to 3
                          (A condition such as if "i==3"  can be add
                          any breakpoint definition )
(gdb) b fn.c:22         - break at line number 22 of source file fn.
                          is compiled into this executable
(gdb) b func            - break at function "func" entry point (or i
                          source  file  fn.c:  b fn.c:func)
(gdb) i b               - info breakpoints. Lists all breakpoints no
(gdb) d                 - delete all breakpoints (d 1 for just #1).
```

An executable file (`a.out` if the `-o` option is not used) can be compiled from a number of source files in C, and the will know about these source files and the lines they contain. To find source line numbers so you can set breakpoints, use the following commands:

```
(gdb) l 23              - list to terminal screen 10 lines of curren
                          centered at line 23
(gdb) l fn.c:22         - print 10 lines from source file in fn.c ce
                          at line 22
(gdb) l func            - print 10 lines around function "func" entr
                          (or l fn.c:func)
(gdb) l                 - print 10 or more lines after lines last pr
```

```
(gdb) l fn.c:1          - print all lines starting from line 1 in so
                          file fn.fc
```

Some other commands for examining memory and program information are:

```
(gdb) i sources         - info on sources - print the names of all s
(gdb) i lo              - info locals: values of all local vars, cur
                          function
(gdb) i var             - info variables: values of global/static va
(gdb) i s               - info stack:calls made to get to this execu
                          point.
(gdb) bt                - same as i s
(gdb) where               same as i s
(gdb) up                - go up one stack level (to caller)
(gdb) down              - go down one stack level (to called functio
(gdb) x 0x20034   - examine memory address 0x20034
(gdb) x/s 0x20034 - examine memory, addr 0x20034, as a string  (also
(gdb) display x   - prints x each time program stops
(gdb) whatis  x   - prints type information for x
```

bf NOTE: While running, `CTRL-C` brings you back to the `(gdb)` prompt to examine the program state. This is useful for programs that are in an infinite loop or hung and for debugging performance problems

To begin running a program again after a break (including the first break at main) type:

```
(gdb) c                 - continue running program from stopped poin
(gdb) c 22              - continue, don't stop at this breakpoint ag
                          until it is encountered 22 times
```

In setting a breakpoint we were allowed to give a condition (`b 36 if i==3`). In a counted continue (`c 22`), the condition is not checked until the breakpoint has been encountered 22 times. To find out what breakpoints exist and delete them:

```
(gdb) i b               - info on breakpoints - returns list of brea
(gdb) d 3               - delete break number 3
(gdb) d                 - delete all breakpoints
```

You can create an array starting at any position to print out values all with the same type:

```
(gdb) p array[3]@12     - will print out values starting at array po
                          3 for 12 positions
```

You can also call your own function from within the debugger:

```
(gdb)  p my_function ()                - calls a function
(gdb)  call my_function ()             - same as p
(gdb)  p my_function (10, ``Boston'') - call function with values
(gdb)  p my_function (x, p->name)      - call  using variable values
```

Other `gdb` capabilities exist which are useful for heavy-duty debugging. There are some photocopied GDB Manuals (stapled, marked: "PLEASE DO NOT REMOVE FROM Unix ROOM") either in the CS Department Terminal Room or in the CS Library in room 157.

6 The script command

The `script` command is used to make a record of an interactive session. If you type:

```
eris% script     (begins recording in file "typescript")
...              (any sequence of commands)
eris% exit       (ends script command, closes "typescript" file)
```

you will create a file named "**typescript**" which contains all the keystrokes you type on the keyboard and all the characters which appear on your terminal screen. Your instructor may ask you to create a script file to demonstrate your work. Commonly, you will be asked to list the contents of your program (use the `cat` command), perform compilation, and exercise the program to demonstrate that it works properly. The typescript file can be left in your directory or can be printed out to be turned in at class.

Be careful that if you have a typescript file you don't wish to lose that you rename it (`mv` command) before giving the script command again, since the new script command will wipe out the contents of the old typescript file.

Also **DO NOT** try to print out binary files (a.out or *.o files!) The printer cannot handle binary files and you will create a huge mess.

7 Unix file protection

When you are given a new account, you will start off with settings that make any file you create readable by everyone else on the system. This is good because it lets you exchange information easily with other users, and learn from them. However, your instructor will want to place restrictions on sharing assignment work, so new users will find they have a subdirectory for each course they have applied for, `cs240` for CS240, with special access control. Any file you create in that directory will be protected from access by normal users ("others" in the file protection nomenclature), although it will still be accessible to your instructor and grader (people in your grading group).

The file protection settings used in Unix are explained in any Unix reference manual. The command `ls -lg` shows file protection and group ownership. The `chmod` command allows you to set protection on individual files or directories. There are three categories of readers: `u` for "user", the original creator of the file, `g` for a member of a "group" with the user, and `o` for "other". Your instructor and grader are members of your group, everyone else is other. You will be able, using the `chmod` command to "add" (+) and "subtract" (-) capabilities for members of these user categories to have various kinds of access to your file. the types of access are: `r` for "read", `w` for "write" (or "update" as in an editor) and `x` for "execute". To disallow other students from reading and executing your file, fname, type:

```
eris% chmod o-rx fname
```

while to allow such access use "`chmod o`rx fname+". Notice that the standard meanings for these types of access change when they are applied to directories; consider giving the command which denies "x" access for the user category "o" on your directory, fname:

```
eris% chmod o-x fname
```

The effect will be that other users are not allowed to execute a listing of your directory structure, and therefore all files existing in that directory are protected from access (even if the files are readable by "other", such users can't access them since they can't interpret the directory they sit in). This is the type of protection you start with in your "cs240" directory.

In your **course directory** the protections have been set for you. It is very important that you **do not touch** the protection on any files in the course directory.

For complex reasons, if you change the protection you will not be able to correct it, and your grader will be unable to access your files.

(The reason, for the record, is that your course directories have a group ownership of your grader's group, to allow the grader to read your files, and the directory has the "setgid bit" set. The setgid bit makes new files and directories inherit the grader's group instead of *your* group. You are *not* a member of the grader group, so that only the grader and the professor can read everyone's files. If you touch the directory protection in any way, Unix will not let you write this setgid bit since you are not in the group. The setgid bit will not be reset, and new files will NOT be readable by the grader, which means that your homework will not be readable by the grader.

If you understand this, you are not a beginner!