Chapter 4

Programming in the Unix Environment

4.1 Text Editors

When writing programs you have many editor options. The most popular Unix text editors are vi(1), nano(1), and emacs(1).

4.1.1 vi(1)

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

```
vi primes.c
```

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.

While 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
- `I` 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 beginning 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 $\frac{1}{2}$ a screen-full
- `^D` scroll the screen down $\frac{1}{2}$ 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 '(', ')', '{', '}', 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 extremely 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 beginning 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 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 separate 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 consistent 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 accidentally 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 pasting into vi(1).

4.1.2 nano

Nano is an open source clone of the popular pico editor (get the joke?) that is included with the University of Washington's mail program, pine.

You run nano(1) by preceding the file name with nano:

```
nano myfile.c
```

If you want to start editing at a particular line number (very common when debugging a program) use the option of a plus sign followed immediately by the line number when launching nano. For example, in order to edit `myfile.c` starting at line 245, you would type

```
nano +245 myfile.c
```

Unlike vi(1), nano(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
- `^_` (that is a control-underscore) marks a section
- `^U` undelete
- `^X` saves the file
- `^W` search for a specific word

4.1.3 Emacs

Emacs is the Swiss army chainsaw of text mode editors. It tries to have something for everyone. The joke goes that emacs is not so much an editor as it is a lifestyle. Emacs is extremely powerful, and if it doesn't do what you want exactly the way you want it, it can be reprogrammed through configuration files. The learning curve for emacs can be very steep, but with just a few commands it is possible to be very productive with a flexibility that other Unix editors can't match.

Terminology

Emacs commands fall into two basic categories, those that are given by relatively simple keystrokes and those where the full command name must be given. The first category of commands are usually associated with the control key while the full commands are tied to what is called the *meta* key.

In the emacs universe, the prefix `C-` is used to denote a control character rather than the caret character `^`. So the control x character would be listed as `C-x` rather than `^x`.

The prefix `M-` is used to denote a meta character. So a meta x character would be listed as `M-x`. Actually using the meta key depends highly on the terminal you are using and how your version of emacs has been configured. If you are fortunate, the meta key will be tied to the Alt key on your keyboard. So in order to type `M-x` you would hold down the Alt key and then press x. If your editor has not been configured this way, or your keyboard does not have an alt key it is still possible to input the meta character commands. In such a situation, the ESC key followed by the desired character will be translated into the proper meta key. For example, `ESC x` would be translated by the editor into `M-x` the same as if you had used the Alt key. (It is important to note however, that when using the ESC key, it should *not* be held down while the other key is being typed, unlike the control or alt keys.)

A nice feature of emacs is that when you type in command sequences, they will be shown in a small, one line window at the bottom of the screen.

Basic Movement and Commands

This section gives some of the basic commands needed to get around emacs.

- `C-x C-c` Quit emacs.
- `C-g` Cancel a command sequence. This is one of the most important commands. If you find that you have inadvertently hit a command sequence or something has messed up, use `C-g`. This is not the same as an undo command.

Moving around your text file is usually done with the cursor keys. The page up and page down keys usually work as well. But emacs has some alternative command bindings if for some reason these don't work.

- `C-f` Move the cursor forward to the right by one character.
- `C-b` Move the cursor back to the left by one character.
- `C-p` Move the cursor up to the previous line.
- `C-n` Move the cursor down to the next line.
- `C-v` Move down one page in the text.
- `M-v` Move up one page in the text.
- `C-a` Move the cursor to the beginning of the line.
- `C-e` Move the cursor to the end of the line.
- `M-<` Move the cursor to the beginning of the document.
- `M->` Move the cursor to the end of the document.

Files, Buffers, and Frames

Emacs has the ability to work with multiple files at the same time. Files are loaded from disk into memory regions called buffers. Buffers are displayed into windows. Most of the time these can be thought of as the same thing, but there are occasional differences.

- `C-x C-f` Load a file into a buffer. After typing in this sequence, you will be asked for the name of the file to load at the bottom of the screen. If you type in just a few characters of the name and hit the TAB key, the editor will attempt to auto-complete the name for you if the file exists.

The file will be loaded and displayed. If the file does not exist, an empty file and buffer with that name will be created.

- `C-x C-s` Save the buffer into a file. It will be saved into the name associated with the buffer.
- `C-x C-w` Write the buffer into a different file. You will be asked for a new file name. This is the same as the "save as" feature found on many programs.
- `C-x k` Kill the buffer. This removes the file from the editor. It does not delete the file from the disk.
- `C-x b` Move the cursor to another buffer. This is one way to move between different files in the editor. You will be asked to give the name of the buffer to move to. If you hit enter without giving a buffer name, the last buffer to be displayed will be used. Don't forget the use of the TAB key to help provide name completion here.
- `C-x C-b` List all of the buffers being used by the editor.

Emacs has the ability to display multiple text windows at the same time.

- `C-x 2` Split the window where the cursor is into two pieces.
- `C-x o` Move the cursor to the other window. Don't forget that the one line command window at the bottom of the screen counts as a window. It may be necessary to execute this command several times to get the cursor into the proper window.
- `C-x 1` Just one window, please.

Editing

Emacs has the ability to cut, copy, paste, search, and replace.

- `C-k` Delete a line. Actually, it only deletes from the current cursor position to the end of the line. If the line has text and the cursor is at the beginning, one `C-k` is needed to delete the text on the line and one more `C-k` to remove the blank line that results.

  When `C-k` is used, it actually doesn't delete the line entirely. It transfers it to what is known as the cut buffer. Successive `C-k`s will place additional lines in the cut buffer.

- `C-y` Yank lines out of the cut buffer and place them back in the text. Lines are not removed from the cut buffer. This makes it possible to create quick duplicates of lines, cutting once, and then pasting multiple times with `C-y`.
- `C-SPC` Mark the beginning of a region. Another way to cut/copy and paste is by marking the beginning and ending of the region to cut/copy. The control key and the space bar marks the beginning of such a region, although it may actually be at the farther position in the text file. It is probably more technically accurate to think of it as the first mark.
- `C-w` Cut the region marked by the first mark (see `C-SPC` above) and the current cursor position into the cut buffer. Use `C-y` to paste the cut text.
- `M-w` Copy the region marked by the first mark and the current cursor position into the cut buffer. Use `C-y` to paste the copied text.
- `C-s` Search for a given piece of text, starting from the current position. Hitting `C-s` multiple times without specifying a different search string will repeat the last search, moving on to the next match. If the end of the file is reached, searching will wrap around to the beginning of the file.
- `C-r` Search in reverse, towards the beginning of the file. Wrap around to the end as needed.
- `M-x replace-string` Asks the user for the string to be searched for and the value to replace it with and then repeatedly makes the change until the end of the file.
- `M-%` or `M-x query-replace` Same as the `replace-string` command except that it asks the user for verification (yes, no, all, or quit) at every possible substitution.

Code Development

Emacs has the facility to integrate with many of the tools of a Unix code development environment.

- `M-x compile` If a Makefile exists in the current directory, this command will start the make process and display the results in a new compilation buffer.
- `M-x gdb` This will launch the debugger. Running code through the debugger in emacs will automatically show the location in the source code while stepping through the program.

- `M-x shell` Opens up a shell where UNIX commands can be run. One problem with this mode is that it does not hide characters that are normally not printed. For example, passwords entered in this shell session will be very visible.

4.1.4 Other Editors

It is possible to use DOS Edit or any other text editor to do the programs and then use ftp(1) to send the files to the Unix system at NIU. If you use Word or Word Perfect you must be sure to save the file as an *ASCII file*!

4.2 Compiling Your Programs

4.2.1 gcc, g++, and cc(1)

You will use the GNU compiler g++(1) in class. Note that there is another C compiler named cc(1). The g++(1) compiler generates more warnings which are *very* useful while learning how to program.

There are many different C and C++ compilers for UNIX environments. Most of these are specific for a particular architecture and operating system. The fine folks in the GNU project have created a C/C++ compiler that has been ported to many different architectures. The name of the compiler is `gcc` or `g++` for the C or C++ compiler respectively. Because of the similarities between C and C++, the two commands actually call the same compiler with different default options. Because of the similarity, the rest of this document will talk exclusively about `g++`. The discussion will automatically apply to `gcc` unless stated otherwise.

Note that there is another C compiler named cc(1) which will *not* be used.

Creating an executable program is a two step process. First the source code is *compiled* to object code. The object code may or may not be stored as a separate file, depending on the compiler used and the compiler options. Second, the object code is *linked* with other collections of object code (including system libraries) to form an executable file.

The GNU g++ compiler can be used to control both the compilation phase and the linking phase of creating executable files.

Like most UNIX commands, `g++` has numerous options. The most useful of these are

`-v` Prints out the version of the compiler. This option is usually used by itself.

`-c` Compiles source code files to object files and stops. The default behavior of the compiler is to compile the given source code files and link them directly into an executable file. The `-c` option is useful for efficiently making projects that have multiple source code files.

`-o` **filename** Specifies the name of the output file. Without this option, executable files have the default name of `a.out`. The default name of an object file is the same as a source code file, but with a `.o` extension replacing whatever source code extension existed.

`-O` Turns on optimization. Allows the compiler to modify the code as it's compiling and linking to produce smaller and/or faster files. The results are (hopefully) functionally equivalent to the program without optimization. In practice optimizer bugs do occur. Recognizing things that can be optimized is actually rather difficult and is not always done correctly. Because of this, optimizing should be the last thing done to a program. Once a program works without optimization, then the optimizer can be used to try to improve the program. If the program doesn't work, then the optimizer can be blamed.

(It's a computer science joke that compilers also contain a *pessimizer* that breaks your code and introduces bugs. Unfortunately, nobody has been able to find the flag that turns this feature off.)

`-g` Turns on code generating options for debugging. Detailed information is stored in the object files and executable files about which lines in the source code file are associated with the machine code instructions. This can make programs comparatively large and slow. However, this information is used by a debugger to allow stepping through a program line by line as it executes, which is very useful. The `-g` option is usually incompatible with the `-O` option. They should not be used together.

`-Wall` Turn on all the warning messages possible. Most useful during the compilation phase, but also works during the linking phase.

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

`-pedantic` considers some of the warnings generated by `-ansi` as errors

`-I` **directory_name** Use the given directory as a place to search for include files. This directory is used in addition to the standard system include directories. Multiple include directories are specified by using a separate `-I` option for each directory. There is no space between the `-I` option and the directory name.

The standard directories used by a compiler are usually built in to the compiler and may vary from machine to machine, but `/usr/include` is often one of the standard directories.

This option is useful only during the compilation phase.

`-l` **library_name** Link the given library into the program. A library is a collection of pre-compiled object code. This option is used only during the linking phase. This option comes at the end of the command line. Multiple libraries are included by using a separate `-l` option for each library.

The standard system libraries are usually found in `/lib` and `/usr/lib`. The names of the libraries take the form of `libname.a` or `libname.so`. The part of the library name after the `lib` and before the suffix is used as the name for linking. For example a common version of the math library is named `libm.a`. It is linked into a program using the `-lm` option.

`-L` **directory_name** Use the given directory as a place to search for library files. This directory is used in addition to the standard system library directories. Multiple library directories are specified by using a separate `-L` option for each directory. There is no space between the `-L` option and the directory name.

This option is useful only during the linking phase.

Here are some examples of using `g++` with many of the options given above.

For the compilation phase:

```
g++ -c -O -Wall -Imy_include_dir myprog.C
```

compiles myprog.C to an object code file with optimization turned on, showing all warnings, and looking in the directory `my_include_dir` for any additional include files.

For the linking phase:

```
g++ -g -Wall -o fun fun.o support.o graphics.o -L/usr/lib/X11 -lX11 -lm
```

links the object code files fun.o support.o and graphics.o into the executable file fun. All warnings are displayed and debugging information is preserved. In addition, the directory `/usr/lib/X11` is used to search for additional libraries and the libraries X11 and m (X11 graphics and math) are linked in as well.

4.2.2 od(1)

This will display a file in an octal dump format. Some useful options are:

`-x` output is in hex, rather than octal

`-v` verbose will show all data

`-c` output is in ASCII

Some examples:

```
        od core
        od -c prog1.c
        od -c output.key
```

Generally, you will probably be most interested in using `od -c` to see the data in most files for CSCI 241.

4.2.3 Core Files

A core file is automatically produced if your prog crashes. These can be quite large. They can be used by the gdb(1) and dbx(1) debuggers 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 .cshrc file:

```
        limit coresize 0
```

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

Another way of prohibiting core file generation is to create an unwriteable core file in the directory where the program is run. This is done by changing to the directory where your program resides and then running

```
         touch core
         chmod 000 core
```

This only needs to be done once for each program directory.

4.3 Debugging Your Program—gdb(1) and dbx(1)

When using the debuggers you should include the -g option when using cc(1) or g++(1). dbx goes with cc and gdb goes with g++ and gcc. See the man-pages for more information on dbx.

The GNU debugger (GDB) allows a programmer to examine the internal operation of a running program. Variables and expressions can be printed and values changed. The program can be executed a line at a time or be forced to pause at specific locations. For programs that step outside the bounds of properly allocated memory, GDB can run a program while constantly watching specified memory location to see when they change. GDB is also useful to perform a post-mortem analysis of a program that has crashed and left behind a core file (more below).

GDB is best used with programs that have been compiled using the `-g` option. This places information in the executable code that lets the debugger know which parts of the executable correspond to which lines in source code files, which memory locations correspond to which variable names, and so forth. Programs compiled with the `-g` option are usually larger and slower than they would be otherwise.

GDB is used by typing in commands at a prompt. To reduce the amount of typing, most commands have abbreviations, many as small a single letter. Because some commands are used successively, GDB has the feature that when an empty command line is entered (no characters, just the return key) it will repeat the previous command. This point is important to understand in the efficient use of GDB.

The GDB commands fall naturally into several categories. An explanation of how to start gdb and a description of some of the most common commands is given below. Where they exist, the command abbreviations are given as well.

4.3.1 Invocation

GDB is started by typing `gdb` on the command line along with the name of the executable file to be debugged. If the program takes command line arguments, they are not given here. A splash message is displayed and then the GDB prompt is given, `(gdb)`. This does not actually run the executable, but loads it in preparation for execution.

4.3.2 Running a Program

The commands for starting and stopping a program are given below.

`r` or `run` This command (re)starts running the program from the beginning. If the program takes command line arguments, these are placed after the `run` (or `r`) command. GDB is smart enough to remember the command line arguments, so subsequent restarts of the program do not need to explicitly list them. For example, in the following session

```
(gdb) r arg1 arg2 arg3

...  More debugging session here

(gdb) r
```

these two program restarts provide the exact same set of arguments to the program.

`q` or `quit` This command exits the debugger.

`n` or `next` Execute the next source code line of the program. If the next line to be executed involves a function call, then the entire function call is executed. This is similar to the "step over" feature found in other debuggers.

Every time the execution stops, the source code line of the line *about* to be executed is printed. Not the line that was just executed. This is one of the most common misunderstandings about any debugger.

This command takes an optional integer argument which specifies the number of steps to take.

`s` or `step` Execute the next source code line of the program. If the next line to be executed involves a function call, then step into that function. This is similar to the "step into" feature found in other debuggers.

Using a combination of `next` and `step` commands it is possible to begin executing in the `main()` function of a program and then step into functions of interest, quickly arriving at a function that may be buried deep within the executing program.

It must be noted that a program can not be started using the `next` and `step` commands. If single stepping through a program from the beginning is desired, a breakpoint must be set on the `main()` function.

This command also takes an optional integer argument specifying the number of steps to take.

`c` or `continue` Continue the execution of the program until the next breakpoint or the program ends.

`finish` Complete execution of the current function and stop after returning. This is very useful after stepping into a function by mistake, or after having verified that the function is executing correctly.

`return` Return immediately from the current function without executing the rest of the code for the function. This command takes an optional argument, which is the value to be returned. This command is useful for working around functions that are known to be defective, but must be used with caution.

`bt or backtrace` Print the execution stack. When a program stops because of a breakpoint or an error, chances are good that it is "executing" in multiple functions. Function A has called function B which in turn is calling function C which called function D when the program stopped. Did the program stop in function A, B, C or D? The answer is yes, it stopped in all of them.

When a function calls another, it is placed on a stack of executing functions. The information associated with any one function that is in the process of being executed is called a *frame*. The `backtrace` command prints out the stack frame of executing functions. This is useful in knowing exactly how the function of interest (the last one called) was called.

The execution stack has a direction. The `main()` function is considered to be at the top of the stack. The innermost function being executed is considered to be at the bottom.

`up` Move up the execution stack.

`down` Move down the execution stack. The `up` and `down` commands change the focus of the GDB commands. If a program has a serious error such as a segmentation fault, it will invariably break inside a library function, during input or output. The code at these levels can not be listed, and in any event, the problem is not likely the fault of the libraries. In order to examine the values of variables in the functions that called the library functions, it is necessary to move up the stack until the desired level has been found.

Moving up or down the stack does not execute any code in the innermost functions. In order to finish the execution of code in any given function, the `finish` command must be used.

4.3.3 Examining Data

`l` or `list` List part of the program source code. Only 10 lines are listed. If no arguments are given to this command the current line to be executed is shown in the middle of the 10 line listing. If no arguments are given and the last command was also a `list` command, then the 10 lines following the lines already shown are displayed. This makes it possible to list large sections of code by using the `list` command multiple times. (Or by using the `list` command once and hitting the enter key multiple times to perform last command repetition.)

The `list` command takes optional arguments. Some of these (shown as examples) are

`list 20` List starting at line 20.

`list my_function1` List starting at the function `my_function1()`

`list +30` List the next 30 lines. This gets around the 10 line limitation mentioned above. This default number of lines to be listed can be changed using the command `set listsize 30`. The current setting of the number of lines to be listed by default can be examined using the command `show listsize`.

`list 10,75` List lines 10 through 75.

`p` or `print` expression Print the given expression. It must be emphasized that this command is capable of printing *expressions*, which can be very complex. The expressions can contain multiple variables. Nearly anything that can be evaluated as a valid C or C++ expression is legal.

Only the variables that are availiable in the current function being executed can be used in expressions. This includes any global variables that are available. Variables in other functions are not available.

This command will print out as much as it intelligently can, given the type of the expression. If printing out the value of a `char *` it will print the value of the C style string pointed at. Arrays can be printed out as well, simply by giving the name of the array as the expression. However, this assumes that the size of the array is known, so only explicitly sized arrays can be printed out this way. Printing out a class instance will show all of the data members for that instance.

Because the `print` command can evaluate any C/C++ expression, it can also evaluate assignment expressions. This is how values are changed using the debugger. For example if the current function contained an integer variable `a` that had been assigned the value of 7, the following session fragment shows how the value of a is examined, set to have the value of 4 and re-examined:

```
(gdb) p a
$1 = 7
(gdb) p a = 4
$2 = 4
(gdb) p a
$3 = 4
(gdb)
```

4.3.4 Breakpoints

Breakpoints are places in the program being examined where the execution is forced to stop. Breakpoints can be inserted and deleted during a debugging session. No changes are made to the source code. Temporary breakpoints can also be created, which exist only until they are encountered the first time.

`b` **or** `break` This command sets a breakpoint. If it is not given an argument it places the breakpoint at the current location in the code. This command can take different kinds of options, some of which are shown below as examples.

`break 30` Set a breakpoint at line 30

`break my_function2` Set a breakpoint at the first executable statement in the function `my_function2()`

`break +10` Set a breakpoint 10 lines after the current position.

`break 40 if *p == 'c'` Set a breakpoint at line 40 that will break only if the expression `*p == 'c'` evaluates to true at that point in the program. This is a very powerful form of setting breakpoints. The breakpoint position can be set using any of the forms given previously: by line number, by function name, or by relative position. The condition can be any valid C/C++ expression that evaluates to a non-zero/zero value, which is treated as true or false.

Each breakpoint is given an integer number when it is assigned. All references to a particular breakpoint are given with that number. The number is printed out when the breakpoint is assigned, but can be printed out as explained below.

`tb` **or** `tbreak` Set a temporary breakpoint. A temporary breakpoint is one that is automatically deleted when it is encoutered. This is useful for one-time situations, getting out of long loops, and the like.

Temporary breakpoints take all of the arguments for setting them that regular breakpoints take.

`info break` Display all of the breakpoints set in the program. The information printed includes the breakpoint number, the location of the breakpoint, and the number of times that the breakpoint has been examined.

`d` **or** `delete` If this command is given no arguments, then all breakpoints are deleted. Otherwise, this command takes a list of integer arguments, the breakpoints to be deleted.

`clear` Delete any breakpoints on the next line to be executed. This command can also take arguments, which are illustrated below:

`clear my_function3` Clear any breakpoints at the beginning of `my_function3`

`clear 25` Clear any breakpoints found on line 25.

4.3.5 Watchpoints

Watchpoints are used to examine the value of an expression at every step of the program, stopping whenever the value of the expression changes. Watchpoints are invaluable in finding out what function is trashing a particular section of memory that should not be changing. Because the watchpoint is examined with every step of the program, watchpoints make a program hundreds or thousands of times slower. But sometimes they are the best tool for the job.

`watch` **expression** Set a watchpoint for the expression. The program will break whenever the value of the expression changes from its current value.

Watchpoints are listed using the `info break` command described above.

4.4 `Make` and `Makefiles`

In any programming project, even a small one, there is a considerable amount of typing involved in building a final executable file from a collection of source code files: Each source code file must be compiled into an object file. The object files must be linked with system libraries into an executable file. The compilation commands may have a considerable number of options. The linking command may contain several system libraries. And if changes are made to any source code file, which files need to recompiled? Or should all of them be recompiled and linked? The opportunities for mistakes are great, especially during peak programming hours (10:00 p.m. - 5:00 a.m.)

The `make` program was designed to build and maintain programming projects.

4.4.1 Options

Like most UNIX commands, `make` has numerous options. The most useful of these are

`-f` **filename** Uses the filename specified as the makefile containing the rules to be used. By default, `make` searches for files of rules named `makefile` or `Makefile`.

`-k` Keep going. By default, `make` stops building a project as soon as an error occurs. The use of this option forces `make` to attempt building other pieces of the project as well. This enables the person building the project to collect a large list of errors for debugging purposes, rather than being limited to only one error at a time.

target By giving on the command line the name of a target in the makefile, only the rules necessary to make that target and its dependencies are executed. This is useful for project maintenance. Pseudo-targets (see below) that represent common sequences of actions can be created. For example, by creating a pseudo-target named `clean` which removes all object files and executables, the project can be quickly cleaned up by typing `make clean` at the command prompt.

4.4.2 Makefile Rules

A makefile consists largely of rules, which specify how various intermediate files in the projects are constructed. Rules are separated by blank lines. Makefile rules consist of three parts — a target, a dependency list and a command which are laid out in the following manner:

```
target:  dependency list
        command
```

Targets

The target in a makefile rule is usually the name of a file that is to be made as part of the project. This is most commonly an executable file or an object code file. But it doesn't have to be a file (see Pseudo-Targets, below). The target must be separated from the dependency list with a colon. The target name should start in the first column position on the line.

Dependency Lists

Dependency lists are lists of files which must all exist and be up to date in order to create the target. The files in the dependency list must be separated by spaces and placed on one line. If the line becomes too long for the editor (a very real possibility for the dependency lists in a large project), then the line may be logically extended by typing a \ *immediately* followed by a new line. Anything on the following line is considered to be logically part of the line extended with the \.

A typical dependency list is a list of object code files if the target is an executable file. For example

```
raytrace:  trace.o light.o input.o sphere.o polygon.o ray.o \
               bound.o triangle.o quad.o
       (a command goes here to make the executable)
```

Other common dependency lists are made of source code and header files if the target is an object code file.

```
trace.o:  trace.cc trace.h rt_attrib.h light.h jitter.h
       (a command goes here to make the object file)
```

Commands

The command in a makefile rule is simply a command that would normally be typed in on the command line to create the target file from the files in the dependency list:

```
pxform: pxform.o similar.o affine.o
        g++ -o pxform pxform.o similar.o affine.o -lm
```

An extremely important point about commands in makefile rules is that they must be indented. But they must *NOT* be indented with spaces. They must be indented with the tab character (`\ t`). To use spaces is an error.

This is a severe design flaw in the make program. By inspection it is impossible to tell if spaces or tabs are being used. The only way to tell is by using an editor and moving the cursor around. If the cursor jumps a distance larger than one space, then tabs are being used.

A command in a makefile rule is run only if the target is out of date with respect to the dependencies. This is determined by examining the timestamps on the target and the dependencies. If any of the dependencies are newer than the target then the target is regenerated (hopefully) by executing the command.

As part of checking the dependencies of any given rule, the make program will verify that the dependency is not the target of some other rule. If it is, then the other rule is evaluated first. Only when all of the dependencies are known to be up to date is the comparison made with the target of the current rule.

Rules about Rules

Some other important points about makefile rules:

1. The rules are not steps in a program. They are not run from top to bottom. They are a collection and may be run in any order according to the dependencies that need to be satisfied. The entire collection of rules is analyzed along with the timestamps of the files involved to determine the specific order to run the commands in.
2. If no target is specified when `make` is invoked, then the target of the first rule listed in the file is assumed. This usually means that the rule that makes the project executable(s) from the object code files is listed first. If the project produces only a single

executable then the rule that creates the executable is used. If the project creates multiple executables, then a pseudo-target that depends on all of the executables is used.

3. Just because it's a common source of errors for beginning makefile users, it is repeated here: Always use tabs to indent makefile rule commands instead of spaces.

Pseudo-Targets

The targets of makefile rules need not correspond to actual files on the system. These are called ¡dfn¿pseudo targets¡/dfn¿ and can be very useful for maintaining the project. One use of rules with pseudo targets is to create complex commands that can be easily invoked. These rules have empty dependency lists and are invoked by giving the target name to `make` when it is called.

For example, a common and useful rule in any project has the pseudo target of "`clean`".

```
clean:
       -rm *.o my_executable_files_here
```

This rule, when invoked, will remove all of the object files and any executable files that are listed as part of the command. This rule is invoked by typing `make clean` at the command line prompt.

By convention, rules of this form are placed towards the end of the makefile, after the other rules that actually build the project.

Another thing to note in the above example is the hyphen in front of the `rm` command. When placed immediately in front of the command to be executed, a hyphen will cause the make program to ignore any errors associated with the command. For example,

```
% make clean
rm *.o

....  A file editing session takes place here


% make clean
```

```
rm: cannot remove *.o : No such file or directory
make: [clean] Error 1 (ignored)
%
```

and if you think that this will not happen to you, that you will never execute a command unless it is needed, you are wrong.

Another use of a pseudo target is to create a rule for projects with multiple executables. This rule has a target and a dependency list, but no executables:

```
all:  program1  prog2  mystuff3


program1: program1.o class2.o otherthings.o
     g++ -o program1 program1.o class2.o otherthings.o -lm

... and so on
```

In this case, the pseudo target `all` depends on multiple executable files. Typing `make all` or placing this rule at the top of the rule list for default application will cause `make` to create all of the executables.

4.4.3 Comments

No computer language is really worth anything until you can write comments in it. (And that's only halfway meant as a joke.) In makefiles, anything following a `#` character is a comment up to the end of the line. Comment characters typically occur at the first column position of a line but do not need to.

4.4.4 Variables

Consider the following situation. In a large project, a bug has come up. (Imagine that.) It is not known what the source of the bug is. You have been assigned to debug the problem. This requires going to the makefile and adding the debugging option (`-g`) to all of the compiling commands. And after the problem is solved, you must go back and restore those options to their original form.

To help this situation, makefiles allow the use of variables. A makefile variable is assigned a string value in a similar to an assignment in a programming language:

```
CC = g++
```

The symbol CC is the variable and its value is `g++`. Makefile variable names are case sensitive. By convention, they are all capital letters. Variables are typically set once, at the beginning of the makefile.

There are some standard variable names that are used for common purposes. The name `CC` is used to hold the name of the compiler. The variable `CCFLAGS` is often used to hold common C++ compiler options.

Unlike most programming languages, using the value of makefile variable does *NOT* consist of simply giving the variable name. To use a makefile variable it is necessary to put the name in parantheses and place a dollar sign on the left. For example, the variable `CC` is given a value as described above. To use the variable, it is necessary to write the expression `$(CC)`. To use the variable `CCFLAGS`, the expression `$(CCFLAGS)` must be used:

```
CC = g++
CCFLAGS = -O -Wall

...


twister:  twister.o  rotate.o
      $(CC) $(CCFLAGS) -o twister  twister.o rotate.o -lm
```

4.4.5 Example

Here is a detailed example of a complete makefile for a relatively small project consisting of several pieces.

```
# Some variables needed to access private libraries for this project
HOME_DIR = /home/z123456
INCLUDES = -I$(HOME_DIR)/include
```

```
LIB_DIR = $(HOME_DIR)/lib/

# Standard compiler variables
CC = g++
CCFLAGS = -Wall -g


# Rules start here

pxform: pxform.o similar.o translation.o perspective.o incremental.o \
        panoramic.o
        $(CC) -o pxform $(CCFLAGS) pxform.o \
        panoramic.o incremental.o perspective.o similar.o \
                translation.o \
                -L$(LIB_DIR) -lrender -lmatrix -lm

incremental.o: incremental.cc incremental.h pxform.h
        $(CC) -c $(CCFLAGS) $(INCLUDES) incremental.cc

panoramic.o: panoramic.cc panoramic.h pxform.h
        $(CC) -c $(CCFLAGS) $(INCLUDES) panoramic.cc

perspective.o: perspective.cc perspective.h pxform.h
        $(CC) -c $(CCFLAGS) $(INCLUDES) perspective.cc

pxform.o: pxform.cc panoramic.h incremental.h perspective.h similar.h \
                translation.h pxform.h
        $(CC) -c $(CCFLAGS) $(INCLUDES) pxform.cc

similar.o: similar.cc similar.h pxform.h
        $(CC) -c $(CCFLAGS) $(INCLUDES) similar.cc

translation.o: translation.cc translation.h pxform.h
        $(CC) -c $(CCFLAGS) $(INCLUDES) translation.cc

clean:
        -rm *.o pxform
```