Learn X in Y minutes (/)

Where X=bash

Get the code: LearnBash.sh (/docs/files/LearnBash.sh)

Bash is a name of the unix shell, which was also distributed as the shell for the GNU operating system and as default shell on Linux and Mac OS X. Nearly all examples below can be a part of a shell script or executed directly in the shell.

Read more here. (http://www.gnu.org/software/bash/manual/bashref.html)

```
#!/bin/bash
# First line of the script is shebang which tells the system how
to execute
# the script: http://en.wikipedia.org/wiki/Shebang_(Unix)
# As you already figured, comments start with #. Shebang is also
a comment.

# Simple hello world example:
echo Hello world!

# Each command starts on a new line, or after semicolon:
echo 'This is the first line'; echo 'This is the second line'

# Declaring a variable looks like this:
Variable="Some string"

# But not like this:
Variable = "Some string"
# Bash will decide that Variable is a command it must execute and
give an error
# because it can't be found.

# Or like this:
Variable= 'Some string'
# Bash will decide that 'Some string' is a command it must
execute and give an
# error because it can't be found. (In this case the 'Variable='
part is seen
# as a variable assignment valid only for the scope of the 'Some
string'
# command.)

# Using the variable:
echo $Variable
echo "$Variable"
echo '$Variable'
# When you use the variable itself — assign it, export it, or
else — you write
# its name without $. If you want to use the variable's value,
you should use $.
# Note that ' (single quote) won't expand the variables!

# Parameter expansion ${ }:
```

```bash
echo ${Variable}
# This is a simple usage of parameter expansion
# Parameter Expansion gets a value from a variable.  It "expands"
or prints the value
# During the expansion time the value or parameter are able to be
modified
# Below are other modifications that add onto this expansion

# String substitution in variables
echo ${Variable/Some/A}
# This will substitute the first occurrence of "Some" with "A"

# Substring from a variable
Length=7
echo ${Variable:0:Length}
# This will return only the first 7 characters of the value

# Default value for variable
echo ${Foo:-"DefaultValueIfFooIsMissingOrEmpty"}
# This works for null (Foo=) and empty string (Foo=""); zero
(Foo=0) returns 0.
# Note that it only returns default value and doesn't change
variable value.

# Brace Expansion { }
# Used to generate arbitrary strings
echo {1..10}
echo {a..z}
# This will output the range from the start value to the end
value

# Builtin variables:
# There are some useful builtin variables, like
echo "Last program's return value: $?"
echo "Script's PID: $$"
echo "Number of arguments passed to script: $#"
echo "All arguments passed to script: $@"
echo "Script's arguments separated into different variables: $1
$2..."

# Now that we know how to echo and use variables,
# let's learn some of the other basics of bash!

# Our current directory is available through the command `pwd`.
# `pwd` stands for "print working directory".
# We can also use the builtin variable `$PWD`.
# Observe that the following are equivalent:
echo "I'm in $(pwd)" # execs `pwd` and interpolates output
echo "I'm in $PWD" # interpolates the variable

# If you get too much output in your terminal, or from a script,
the command
# `clear` clears your screen
clear
# Ctrl-L also works for clearing output

# Reading a value from input:
echo "What's your name?"
read Name # Note that we didn't need to declare a new variable
echo Hello, $Name!

# We have the usual if structure:
# use 'man test' for more info about conditionals
if [ $Name != $USER ]
then
    echo "Your name isn't your username"
else
    echo "Your name is your username"
fi

# NOTE: if $Name is empty, bash sees the above condition as:
if [ != $USER ]
# which is invalid syntax
# so the "safe" way to use potentially empty variables in bash
is:
if [ "$Name" != $USER ] ...
# which, when $Name is empty, is seen by bash as:
if [ "" != $USER ] ...
# which works as expected

# There is also conditional execution
echo "Always executed" || echo "Only executed if first command
fails"
echo "Always executed" && echo "Only executed if first command
does NOT fail"

# To use && and || with if statements, you need multiple pairs of
square brackets:
if [ "$Name" == "Steve" ] && [ "$Age" -eq 15 ]
then
```

```
    echo "This will run if $Name is Steve AND $Age is 15."
fi

if [ "$Name" == "Daniya" ] || [ "$Name" == "Zach" ]
then
    echo "This will run if $Name is Daniya OR Zach."
fi

# Expressions are denoted with the following format:
echo $(( 10 + 5 ))

# Unlike other programming languages, bash is a shell so it works
in the context
# of a current directory. You can list files and directories in
the current
# directory with the ls command:
ls

# These commands have options that control their execution:
ls -l # Lists every file and directory on a separate line
ls -t # Sorts the directory contents by last-modified date
(descending)
ls -R # Recursively `ls` this directory and all of its
subdirectories

# Results of the previous command can be passed to the next
command as input.
# grep command filters the input with provided patterns. That's
how we can list
# .txt files in the current directory:
ls -l | grep "\.txt"

# Use `cat` to print files to stdout:
cat file.txt

# We can also read the file using `cat`:
Contents=$(cat file.txt)
echo "START OF FILE\n$Contents\nEND OF FILE"

# Use `cp` to copy files or directories from one place to
another.
# `cp` creates NEW versions of the sources,
# so editing the copy won't affect the original (and vice versa).
# Note that it will overwrite the destination if it already
exists.
```

```
cp srcFile.txt clone.txt
cp -r srcDirectory/ dst/ # recursively copy

# Look into `scp` or `sftp` if you plan on exchanging files
between computers.
# `scp` behaves very similarly to `cp`.
# `sftp` is more interactive.

# Use `mv` to move files or directories from one place to
another.
# `mv` is similar to `cp`, but it deletes the source.
# `mv` is also useful for renaming files!
mv s0urc3.txt dst.txt # sorry, l33t hackers...

# Since bash works in the context of a current directory, you
might want to
# run your command in some other directory. We have cd for
changing location:
cd ~    # change to home directory
cd ..   # go up one directory
        # (^^say, from /home/username/Downloads to
/home/username)
cd /home/username/Documents   # change to specified directory
cd ~/Documents/..    # still in home directory..isn't it??

# Use subshells to work across directories
(echo "First, I'm here: $PWD") && (cd someDir; echo "Then, I'm
here: $PWD")
pwd # still in first directory

# Use `mkdir` to create new directories.
mkdir myNewDir
# The `-p` flag causes new intermediate directories to be created
as necessary.
mkdir -p myNewDir/with/intermediate/directories

# You can redirect command input and output (stdin, stdout, and
stderr).
# Read from stdin until ^EOF$ and overwrite hello.py with the
lines
# between "EOF":
cat > hello.py << EOF
#!/usr/bin/env python
from __future__ import print_function
import sys
```

```
print("#stdout", file=sys.stdout)
print("#stderr", file=sys.stderr)
for line in sys.stdin:
    print(line, file=sys.stdout)
EOF

# Run hello.py with various stdin, stdout, and stderr
redirections:
python hello.py < "input.in"
python hello.py > "output.out"
python hello.py 2> "error.err"
python hello.py > "output-and-error.log" 2>&1
python hello.py > /dev/null 2>&1
# The output error will overwrite the file if it exists,
# if you want to append instead, use ">>":
python hello.py >> "output.out" 2>> "error.err"

# Overwrite output.out, append to error.err, and count lines:
info bash 'Basic Shell Features' 'Redirections' > output.out 2>>
error.err
wc -l output.out error.err

# Run a command and print its file descriptor (e.g. /dev/fd/123)
# see: man fd
echo <(echo "#helloworld")

# Overwrite output.out with "#helloworld":
cat > output.out <(echo "#helloworld")
echo "#helloworld" > output.out
echo "#helloworld" | cat > output.out
echo "#helloworld" | tee output.out >/dev/null

# Cleanup temporary files verbosely (add '-i' for interactive)
# WARNING: `rm` commands cannot be undone
rm -v output.out error.err output-and-error.log
rm -r tempDir/ # recursively delete

# Commands can be substituted within other commands using $( ):
# The following command displays the number of files and
directories in the
# current directory.
echo "There are $(ls | wc -l) items here."

# The same can be done using backticks `` but they can't be
nested - the preferred way
```

```
# is to use $( ).
echo "There are `ls | wc -l` items here."

# Bash uses a case statement that works similarly to switch in
Java and C++:
case "$Variable" in
    #List patterns for the conditions you want to meet
    0) echo "There is a zero.";;
    1) echo "There is a one.";;
    *) echo "It is not null.";;
esac

# for loops iterate for as many arguments given:
# The contents of $Variable is printed three times.
for Variable in {1..3}
do
    echo "$Variable"
done

# Or write it the "traditional for loop" way:
for ((a=1; a <= 3; a++))
do
    echo $a
done

# They can also be used to act on files..
# This will run the command 'cat' on file1 and file2
for Variable in file1 file2
do
    cat "$Variable"
done

# ..or the output from a command
# This will cat the output from ls.
for Output in $(ls)
do
    cat "$Output"
done

# while loop:
while [ true ]
do
    echo "loop body here..."
    break
done
```

```bash
# You can also define functions
# Definition:
function foo ()
{
    echo "Arguments work just like script arguments: $@"
    echo "And: $1 $2..."
    echo "This is a function"
    return 0
}

# or simply
bar ()
{
    echo "Another way to declare functions!"
    return 0
}

# Calling your function
foo "My name is" $Name

# There are a lot of useful commands you should learn:
# prints last 10 lines of file.txt
tail -n 10 file.txt
# prints first 10 lines of file.txt
head -n 10 file.txt
# sort file.txt's lines
sort file.txt
# report or omit repeated lines, with -d it reports them
uniq -d file.txt
# prints only the first column before the ',' character
cut -d ',' -f 1 file.txt
# replaces every occurrence of 'okay' with 'great' in file.txt,
(regex compatible)
sed -i 's/okay/great/g' file.txt
# print to stdout all lines of file.txt which match some regex
# The example prints lines which begin with "foo" and end in
"bar"
grep "^foo.*bar$" file.txt
# pass the option "-c" to instead print the number of lines
matching the regex
grep -c "^foo.*bar$" file.txt
# Other useful options are:
grep -r "^foo.*bar$" someDir/ # recursively `grep`
grep -n "^foo.*bar$" file.txt # give line numbers
grep -rI "^foo.*bar$" someDir/ # recursively `grep`, but ignore
binary files
# perform the same initial search, but filter out the lines
containing "baz"
grep "^foo.*bar$" file.txt | grep -v "baz"

# if you literally want to search for the string,
# and not the regex, use fgrep (or grep -F)
fgrep "foobar" file.txt

# trap command allows you to execute a command when a signal is
received by your script.
# Here trap command will execute rm if any one of the three
listed signals is received.
trap "rm $TEMP_FILE; exit" SIGHUP SIGINT SIGTERM

# `sudo` is used to perform commands as the superuser
NAME1=$(whoami)
NAME2=$(sudo whoami)
echo "Was $NAME1, then became more powerful $NAME2"

# Read Bash shell builtins documentation with the bash 'help'
builtin:
help
help help
help for
help return
help source
help .

# Read Bash manpage documentation with man
apropos bash
man 1 bash
man bash

# Read info documentation with info (? for help)
apropos info | grep '^info.*('
man info
info info
info 5 info

# Read bash info documentation:
info bash
info bash 'Bash Features'
info bash 6
info --apropos bash
```

Simple RegEx Tutorial

Regular Expression can be used in Content Filter conditions.

Regular Expressions can be extremely complex but they are very flexible and powerful and can be used to perform comparisons that cannot be done using the other checks available.

There follows some very basic examples of regular expression usage. For a complete description please visit www.regular-expressions.info.

^' and '$'

First of all, let's take a look at two special symbols: '^' and '$'. These symbols indicate the start and the end of a string, respectively:

"^The"	matches any string that starts with "The".
"of despair$"	matches a string that ends in with "of despair".
"^abc$"	a string that starts and ends with "abc" - effectively an exact match comparison.
"notice"	a string that has the text "notice" in it.

You can see that if you don't use either of these two characters, you're saying that the pattern may occur anywhere inside the string -- you're not "hooking" it to any of the edges.

'*', '+', and '?'

In addition, the symbols '*', '+', and '?', denote the number of times a character or a sequence of characters may occur. What they mean is: "zero or more", "one or more", and "zero or one." Here are some examples:

"ab*"	matches a string that has an a followed by zero or more b's ("ac", "abc", "abbc", etc.)
"ab+"	same, but there's at least one b ("abc", "abbc", etc., but not "ac")
"ab?"	there might be a single b or not ("ac", "abc" but not "abbc").
"a?b+$"	a possible 'a' followed by one or more 'b's at the end of the string: Matches any string ending with "ab", "abb", "abbb" etc. or "b", "bb" etc. but not "aab", "aabb" etc.

Braces { }

You can also use bounds, which appear inside braces and indicate ranges in the number of occurrences:

"ab{2}"	matches a string that has an a followed by exactly two b's ("abb")
"ab{2,}"	there are at least two b's ("abb", "abbbb", etc.)
"ab{3,5}"	from three to five b's ("abbb", "abbbb", or "abbbbb")

Note that you must always specify the first number of a range (i.e., "{0,2}", not "{,2}"). Also, as you might have noticed, the symbols '*', '+', and '?' have the same effect as using the bounds "{0,}", "{1,}", and "{0,1}", respectively.

Now, to quantify a sequence of characters, put them inside parentheses:

"a(bc)*"	matches a string that has an a followed by zero or more copies of the sequence "bc"
"a(bc){1,5}"	one through five copies of "bc."

'|' OR operator

There's also the '|' symbol, which works as an OR operator:

"hi|hello"	matches a string that has either "hi" or "hello" in it
"(b|cd)ef"	a string that has either "bef" or "cdef"
"(a|b)*c"	a string that has a sequence of alternating a's and b's ending in a c

('.')

A period ('.') stands for any single character:

"a.[0-9]"	matches a string that has an a followed by one character and a digit
"^.{3}$"	a string with exactly 3 characters

Bracket expressions

specify which characters are allowed in a single position of a string:

"[ab]"	matches a string that has either an a or a b (that's the same as "a|b")
"[a-d]"	a string that has lowercase letters 'a' through 'd' (that's equal to "a|b|c|d" and even "[abcd]")
"^[a-zA-Z]"	a string that starts with a letter
"[0-9]%"	a string that has a single digit before a percent sign
",[a-zA-Z0- 9]$"	a string that ends in a comma followed by an alphanumeric character

You can also list which characters you DON'T want -- just use a '^' as the first symbol in a bracket expression (i.e., "%[^a- zA-Z]%" matches a string with a character that is not a letter between two percent signs).

In order to be taken literally, you must escape the characters "^.[$()|*+?{\" with a backslash ('\'), as they have special meaning. On top of that, you must escape the backslash character itself in PHP3 strings, so, for instance, the regular expression "(\$|A)[0-9]+" would have the function call: ereg("(\\$|A)[0-9]+", $str) (what string does that validate?)

Just don't forget that bracket expressions are an exception to that rule--inside them, all special characters, including the backslash ('\'), lose their special powers (i.e., "[*\+?{}.]" matches exactly any of the characters inside the brackets). And, as the regex manual pages tell us: "To include a literal ']' in the list, make it the first character (following a possible '^'). To include a literal '-', make it the first or last character, or the second endpoint of a range."

See Also

Shared

Database Settings

Access Mode

Schedule

Domain Admin Rights

Select Accounts

Account Options

GDB QUICK REFERENCE GDB Version 4

Essential Commands

`gdb` *program* [*core*]	debug *program* [using coredump *core*]
`b` [*file*`:`]*function*	set breakpoint at *function* [in *file*]
`run` [*arglist*]	start your program [with *arglist*]
`bt`	backtrace: display program stack
`p` *expr*	display the value of an expression
`c`	continue running your program
`n`	next line, stepping over function calls
`s`	next line, stepping into function calls

Starting GDB

`gdb`	start GDB, with no debugging files
`gdb` *program*	begin debugging *program*
`gdb` *program core*	debug coredump *core* produced by *program*
`gdb --help`	describe command line options

Stopping GDB

`quit`	exit GDB; also `q` or `EOF` (eg `C-d`)
`INTERRUPT`	(eg `C-c`) terminate current command, or send to running process

Getting Help

`help`	list classes of commands
`help` *class*	one-line descriptions for commands in *class*
`help` *command*	describe *command*

Executing your Program

`run` *arglist*	start your program with *arglist*
`run`	start your program with current argument list
`run ... <`*inf* `>`*outf*	start your program with input, output redirected
`kill`	kill running program
`tty` *dev*	use *dev* as stdin and stdout for next `run`
`set args` *arglist*	specify *arglist* for next `run`
`set args`	specify empty argument list
`show args`	display argument list
`show env`	show all environment variables
`show env` *var*	show value of environment variable *var*
`set env` *var string*	set environment variable *var*
`unset env` *var*	remove *var* from environment

Shell Commands

`cd` *dir*	change working directory to *dir*
`pwd`	Print working directory
`make ...`	call "`make`"
`shell` *cmd*	execute arbitrary shell command string

[ ] surround optional arguments `...` show one or more arguments

Breakpoints and Watchpoints

`break` [*file*`:`]*line* `b` [*file*`:`]*line*	set breakpoint at *line* number [in *file*] eg: `break main.c:37`
`break` [*file*`:`]*func*	set breakpoint at *func* [in *file*]
`break +`*offset* `break -`*offset*	set break at *offset* lines from current stop
`break *`*addr*	set breakpoint at address *addr*
`break`	set breakpoint at next instruction
`break ... if` *expr*	break conditionally on nonzero *expr*
`cond` *n* [*expr*]	new conditional expression on breakpoint *n*; make unconditional if no *expr*
`tbreak ...`	temporary break; disable when reached
`rbreak` *regex*	break on all functions matching *regex*
`watch` *expr*	set a watchpoint for expression *expr*
`catch` *event*	break at *event*, which may be `catch`, `throw`, `exec`, `fork`, `vfork`, `load`, or `unload`.
`info break`	show defined breakpoints
`info watch`	show defined watchpoints
`clear`	delete breakpoints at next instruction
`clear` [*file*`:`]*fun*	delete breakpoints at entry to *fun*()
`clear` [*file*`:`]*line*	delete breakpoints on source line
`delete` [*n*]	delete breakpoints [or breakpoint *n*]
`disable` [*n*]	disable breakpoints [or breakpoint *n*]
`enable` [*n*]	enable breakpoints [or breakpoint *n*]
`enable once` [*n*]	enable breakpoints [or breakpoint *n*]; disable again when reached
`enable del` [*n*]	enable breakpoints [or breakpoint *n*]; delete when reached
`ignore` *n count*	ignore breakpoint *n*, *count* times
`commands` *n* [`silent`] *command-list*	execute GDB *command-list* every time breakpoint *n* is reached. [`silent` suppresses default display]
`end`	end of *command-list*

Program Stack

`backtrace` [*n*] `bt` [*n*]	print trace of all frames in stack; or of *n* frames—innermost if *n*`>0`, outermost if *n*`<0`
`frame` [*n*]	select frame number *n* or frame at address *n*; if no *n*, display current frame
`up` *n*	select frame *n* frames up
`down` *n*	select frame *n* frames down
`info frame` [*addr*]	describe selected frame, or frame at *addr*
`info args`	arguments of selected frame
`info locals`	local variables of selected frame
`info reg` [*rn*]`...` `info all-reg` [*rn*]	register values [for regs *rn*] in selected frame; `all-reg` includes floating point

Execution Control

`continue` [*count*] `c` [*count*]	continue running; if *count* specified, ignore this breakpoint next *count* times
`step` [*count*] `s` [*count*]	execute until another line reached; repeat *count* times if specified
`stepi` [*count*] `si` [*count*]	step by machine instructions rather than source lines
`next` [*count*] `n` [*count*]	execute next line, including any function calls
`nexti` [*count*] `ni` [*count*]	next machine instruction rather than source line
`until` [*location*]	run until next instruction (or *location*)
`finish`	run until selected stack frame returns
`return` [*expr*]	pop selected stack frame without executing [setting return value]
`signal` *num*	resume execution with signal *s* (none if `0`)
`jump` *line* `jump *`*address*	resume execution at specified *line* number or *address*
`set var=`*expr*	evaluate *expr* without displaying it; use for altering program variables

Display

`print` [`/`*f*] [*expr*] `p` [`/`*f*] [*expr*]	show value of *expr* [or last value `$`] according to format *f*:
`x`	hexadecimal
`d`	signed decimal
`u`	unsigned decimal
`o`	octal
`t`	binary
`a`	address, absolute and relative
`c`	character
`f`	floating point
`call` [`/`*f*] *expr*	like `print` but does not display `void`
`x` [`/`*Nuf*] *expr*	examine memory at address *expr*; optional format spec follows slash
N	count of how many units to display
u	unit size; one of `b` individual bytes `h` halfwords (two bytes) `w` words (four bytes) `g` giant words (eight bytes)
f	printing format. Any `print` format, or `s` null-terminated string `i` machine instructions
`disassem` [*addr*]	display memory as machine instructions

Automatic Display

`display` [`/`*f*] *expr*	show value of *expr* each time program stops [according to format *f*]
`display`	display all enabled expressions on list
`undisplay` *n*	remove number(s) *n* from list of automatically displayed expressions
`disable disp` *n*	disable display for expression(s) number *n*
`enable disp` *n*	enable display for expression(s) number *n*
`info display`	numbered list of display expressions

Expressions

expr	an expression in C, C++, or Modula-2 (including function calls), or:
addr`@`*len*	an array of *len* elements beginning at *addr*
file`::`*nm*	a variable or function *nm* defined in *file*
{*type*}*addr*	read memory at *addr* as specified *type*
`$`	most recent displayed value
`$`*n*	*n*th displayed value
`$$`	displayed value previous to $
`$$`*n*	*n*th displayed value back from $
`$_`	last address examined with `x`
`$__`	value at address $_
`$`*var*	convenience variable; assign any value
`show values` [*n*]	show last 10 values [or surrounding $*n*]
`show conv`	display all convenience variables

Symbol Table

`info address` *s*	show where symbol *s* is stored
`info func` [*regex*]	show names, types of defined functions (all, or matching *regex*)
`info var` [*regex*]	show names, types of global variables (all, or matching *regex*)
`whatis` [*expr*] `ptype` [*expr*]	show data type of *expr* [or `$`] without evaluating; `ptype` gives more detail
`ptype` *type*	describe type, struct, union, or enum

GDB Scripts

`source` *script*	read, execute GDB commands from file *script*
`define` *cmd* *command-list*	create new GDB command *cmd*; execute script defined by *command-list*
`end`	end of *command-list*
`document` *cmd* *help-text*	create online documentation for new GDB command *cmd*
`end`	end of *help-text*

Signals

`handle` *signal act*	specify GDB actions for *signal*:
`print`	announce signal
`noprint`	be silent for signal
`stop`	halt execution on signal
`nostop`	do not halt execution
`pass`	allow your program to handle signal
`nopass`	do not allow your program to see signal
`info signals`	show table of signals, GDB action for each

Debugging Targets

`target` *type param*	connect to target machine, process, or file
`help target`	display available targets
`attach` *param*	connect to another process
`detach`	release target from GDB control

Controlling GDB

`set` *param value*	set one of GDB's internal parameters
`show` *param*	display current setting of parameter

Parameters understood by `set` and `show`:

`complaint` *limit*	number of messages on unusual symbols
`confirm` *on/off*	enable or disable cautionary queries
`editing` *on/off*	control `readline` command-line editing
`height` *lpp*	number of lines before pause in display
`language` *lang*	Language for GDB expressions (`auto`, `c` or `modula-2`)
`listsize` *n*	number of lines shown by `list`
`prompt` *str*	use *str* as GDB prompt
`radix` *base*	octal, decimal, or hex number representation
`verbose` *on/off*	control messages when loading symbols
`width` *cpl*	number of characters before line folded
`write` *on/off*	Allow or forbid patching binary, core files (when reopened with `exec` or `core`)

`history ...`
`h ...`
	groups with the following options:

`h exp` *off/on*	disable/enable `readline` history expansion
`h file` *filename*	file for recording GDB command history
`h size` *size*	number of commands kept in history list
`h save` *off/on*	control use of external file for command history

`print ...`
`p ...`
	groups with the following options:

`p address` *on/off*	print memory addresses in stacks, values
`p array` *off/on*	compact or attractive format for arrays
`p demangl` *on/off*	source (demangled) or internal form for C++ symbols
`p asm-dem` *on/off*	demangle C++ symbols in machine-instruction output
`p elements` *limit*	number of array elements to display
`p object` *on/off*	print C++ derived types for objects
`p pretty` *off/on*	struct display: compact or indented
`p union` *on/off*	display of union members
`p vtbl` *off/on*	display of C++ virtual function tables

`show commands`	show last 10 commands
`show commands` *n*	show 10 commands around number *n*
`show commands +`	show next 10 commands

Working Files

`file` [*file*]	use *file* for both symbols and executable; with no arg, discard both
`core` [*file*]	read *file* as coredump; or discard
`exec` [*file*]	use *file* as executable only; or discard
`symbol` [*file*]	use symbol table from *file*; or discard
`load` *file*	dynamically link *file* and add its symbols
`add-sym` *file addr*	read additional symbols from *file*, dynamically loaded at *addr*
`info files`	display working files and targets in use
`path` *dirs*	add *dirs* to front of path searched for executable and symbol files
`show path`	display executable and symbol file path
`info share`	list names of shared libraries currently loaded

Source Files

`dir` *names*	add directory *names* to front of source path
`dir`	clear source path
`show dir`	show current source path
`list`	show next ten lines of source
`list -`	show previous ten lines
`list` *lines*	display source surrounding *lines*, specified as:
[*file*`:`]*num*	line number [in named file]
[*file*`:`]*function*	beginning of function [in named file]
`+`*off*	*off* lines after last printed
`-`*off*	*off* lines previous to last printed
`*`*address*	line containing *address*
`list` *f*`,`*l*	from line *f* to line *l*
`info line` *num*	show starting, ending addresses of compiled code for source line *num*
`info source`	show name of current source file
`info sources`	list all source files in use
`forw` *regex*	search following source lines for *regex*
`rev` *regex*	search preceding source lines for *regex*

GDB under GNU Emacs

`M-x gdb`	run GDB under Emacs
`C-h m`	describe GDB mode
`M-s`	step one line (`step`)
`M-n`	next line (`next`)
`M-i`	step one instruction (`stepi`)
`C-c C-f`	finish current stack frame (`finish`)
`M-c`	continue (`cont`)
`M-u`	up *arg* frames (`up`)
`M-d`	down *arg* frames (`down`)
`C-x &`	copy number from point, insert at end
`C-x SPC`	(in source file) set break at point

GDB License

`show copying`	Display GNU General Public License
`show warranty`	There is NO WARRANTY for GDB. Display full no-warranty statement.

GitHub

GIT CHEAT SHEET

V 1.1.1

Git is the open source distributed version control system that facilitates GitHub activities on your laptop or desktop. This cheat sheet summarizes commonly used Git command line instructions for quick reference.

INSTALL GIT

GitHub provides desktop clients that include a graphical user interface for the most common repository actions and an automatically updating command line edition of Git for advanced scenarios.

GitHub for Windows
https://windows.github.com

GitHub for Mac
https://mac.github.com

Git distributions for Linux and POSIX systems are available on the official Git SCM web site.

Git for All Platforms
http://git-scm.com

CONFIGURE TOOLING

Configure user information for all local repositories

```
$ git config --global user.name "[name]"
```

Sets the name you want attached to your commit transactions

```
$ git config --global user.email "[email address]"
```

Sets the email you want attached to your commit transactions

```
$ git config --global color.ui auto
```

Enables helpful colorization of command line output

CREATE REPOSITORIES

Start a new repository or obtain one from an existing URL

```
$ git init [project-name]
```

Creates a new local repository with the specified name

```
$ git clone [url]
```

Downloads a project and its entire version history

MAKE CHANGES

Review edits and craft a commit transaction

```
$ git status
```

Lists all new or modified files to be committed

```
$ git diff
```

Shows file differences not yet staged

```
$ git add [file]
```

Snapshots the file in preparation for versioning

```
$ git diff --staged
```

Shows file differences between staging and the last file version

```
$ git reset [file]
```

Unstages the file, but preserve its contents

```
$ git commit -m "[descriptive message]"
```

Records file snapshots permanently in version history

GROUP CHANGES

Name a series of commits and combine completed efforts

```
$ git branch
```

Lists all local branches in the current repository

```
$ git branch [branch-name]
```

Creates a new branch

```
$ git checkout [branch-name]
```

Switches to the specified branch and updates the working directory

```
$ git merge [branch]
```

Combines the specified branch's history into the current branch

```
$ git branch -d [branch-name]
```

Deletes the specified branch

GIT CHEAT SHEET

REFACTOR FILENAMES

Relocate and remove versioned files

```
$ git rm [file]
```

Deletes the file from the working directory and stages the deletion

```
$ git rm --cached [file]
```

Removes the file from version control but preserves the file locally

```
$ git mv [file-original] [file-renamed]
```

Changes the file name and prepares it for commit

SUPPRESS TRACKING

Exclude temporary files and paths

```
*.log
build/
temp-*
```

A text file named `.gitignore` suppresses accidental versioning of files and paths matching the specified patterns

```
$ git ls-files --other --ignored --exclude-standard
```

Lists all ignored files in this project

SAVE FRAGMENTS

Shelve and restore incomplete changes

```
$ git stash
```

Temporarily stores all modified tracked files

```
$ git stash pop
```

Restores the most recently stashed files

```
$ git stash list
```

Lists all stashed changesets

```
$ git stash drop
```

Discards the most recently stashed changeset

REVIEW HISTORY

Browse and inspect the evolution of project files

```
$ git log
```

Lists version history for the current branch

```
$ git log --follow [file]
```

Lists version history for a file, including renames

```
$ git diff [first-branch]...[second-branch]
```

Shows content differences between two branches

```
$ git show [commit]
```

Outputs metadata and content changes of the specified commit

REDO COMMITS

Erase mistakes and craft replacement history

```
$ git reset [commit]
```

Undoes all commits after `[commit]`, preserving changes locally

```
$ git reset --hard [commit]
```

Discards all history and changes back to the specified commit

SYNCHRONIZE CHANGES

Register a repository bookmark and exchange version history

```
$ git fetch [bookmark]
```

Downloads all history from the repository bookmark

```
$ git merge [bookmark]/[branch]
```

Combines bookmark's branch into current local branch

```
$ git push [alias] [branch]
```

Uploads all local branch commits to GitHub

```
$ git pull
```

Downloads bookmark history and incorporates changes