

command line

purpose	command	flags	example(s)
view file(s)	ls [-l] [path...]	-l → verbose	ls *.c
change directory	cd [directory]		cd ps1
make directory	mkdir [-m [permissions]] [directory]	-m → set permissions	mkdir tempdir
remove directory	rmdir [directory]		rmdir tempdir
delete (remove) files	rm [-R] [-f] [path...]	-r → recursive	rm mytester
copy files	cp [-R] [-f] [from...] [to]	-f → force (remove or overwrite) without asking	cp -R * ../ps1 backup 5
move or rename files	mv [from...] [to]		mv
view processes	ps [uxw]	uxw → detailed output	ps auxw
hex dump	xxd [-g [# of bytes]]	-g → group by [# of bytes]	
edit file	vim [-p] [path...]	-p → open files in tabs	vim -p *.c *.h
compile	gcc [-o [executable]] [path...]	-o → output executable	gcc -o ps1 ps1.c
get starter files	264get [asg]	[asg] is the short name of the assignment (e.g., "hw01")	264get hw01
pre-test submission	264test [asg]		264test hw01
submit	264submit [asg] [path...]	[path...] is the file(s) or "*" for all	264submit hw01 *

vim

	h	l	0	\$	^	w	e	b
motion within line	←	→	to beginning of line	to end of line	to first non-blank in line	to beginning of next word	to end of this word	to beginning of this or last word
motion between lines	k ↑	j ↓	gg to beginning of file	G to end of file	line# G line number	% to matching ({ <)	m[a-z] mark position	'[a-z] go to mark
motion search	* find word, forward	# find word, backward	/[pattern] find pattern, forward	. [any char] any char	[pattern] alphanumeric	n to next match	N to previous match	:noh clear search highlighting
action current line	dd delete line (cut)	cc change line	yy yank line (copy)	>> indent line	<< dedent line	== indent code line	gugu lowercase line	gUgU uppercase line
action by motion	d[motion] delete (cut)	c[motion] change	y[motion] yank (copy)	>[motion] indent	<[motion] dedent	= [motion] indent code	gu[motion] lowercase	gU[motion] uppercase
action add text	i insert before this character	I Insert before line beginning	a append after this character	A Append after line end	o open line below	O Open line above	p put (paste) text here/below	P Put (paste) text before/above
other visual, undo, ...	v visual select	V visual select line	u undo last action	^R redo last undone action	. repeat last action	q[a-z] record quick macro	q stop recording quick macro	@[a-z] play quick macro
commands "ex" mode	:w write (save) file	:e [file] edit (open) file	:tabe [file] tab: edit file	:split split window	:s/[pattern]/[text]/gc replace [pattern] with [text]	:h [topic/cmd] help	:q quit Vim	

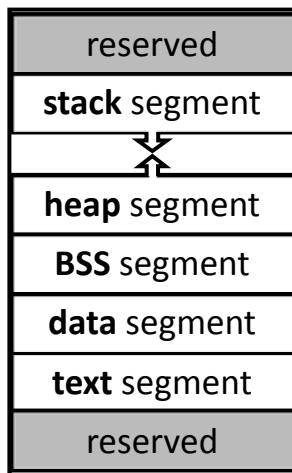
Press **Esc** to return to Normal mode. | Most normal mode commands can be repeated by preceding with a number (e.g., **3dd** to delete 3 lines).

gdb

Start	Automatic display	Controlling execution	View variables and memory
In bash: gdb [--tui] [file]	info display	continue	print [/format] [expression]
quit	display [expression]	finish	• [format] ∈ d (decimal), x (hex), s (string), f (float), c (char.), u (unsigned decimal), o (octal), t (binary), z (zero-padded hex)
set args [arglist...]	undisplay [expression#]	jump [file]:function [file]:line#	• [expression]: a C expression
Breakpoints	Explore the stack frame	next	x / [# of units] [unit] [format] [address]
break [file]:function [file]:line#	backtrace [full] [n]	return [expr]	• [unit] ∈ b (1 byte), h (2 bytes), w (4 bytes), g (8 bytes)
clear [file]:function [file]:line#	down # toward current frame	run [arguments...]	• [# of units]: how many units
delete [breakpoint#]	Frame [frame#]	set variable var = expr	• [format]: same as for p/
info breakpoints	info args	step	
Watchpoints	info frame	until [line#]	
watch [variable]	info locals	Reverse debugging	
awatch [variable]	list [function line#, line#]	record	
rwatch [variable]	up # toward main()	reverse-next	
info watchpoints	whatis [variable]	reverse-step # and so on...	

Underlined letters indicate shortcuts (e.g., n for next, rn for reverse-next, etc.) | Brackets denote parameters that are optional.

memory



<code>void oat(char pie) {</code>	Your code, compiled binary	text segment
<code>char ham;</code>	parameters	stack segment
<code>char bun[4];</code>	local variable	stack segment
<code>char* ice =</code>	statically-allocated array	stack segment
<code>"pop";</code>	local variable (even an address)	stack segment
<code>char* yam =</code>	string literals	data segment, read-only
<code>malloc(sizeof(*yam));</code>	local variable (even an address)	stack segment
<code>static char egg = 1;</code>	dynamic allocation block	heap segment
<code>static char nut;</code>	static variable, initialized	data segment, read-write
<code>free(yam);</code>	static variable, uninitialized	BSS segment
<code>}</code>		
<code>char _g_jam = 2;</code>	global variable, initialized	data segment, read-write
<code>char _g_tea;</code>	global variable, uninitialized	BSS segment

addresses (pointers)

```
int a = 10; // "a gets 10"
int* b; // "b is an address of an int"
b = &a; // "b gets the address of a"
int c = *b; // "c gets the value at b"
int* d = malloc(sizeof(*d));
// "d gets the address of a new allocation block
// sufficient for 1 int"
*d = 10; // "store 10 at address d"
All (a, *b, c, *d) equal 10.
char (*a_f)(int, int) = f;
// "a_f is the address of function f(...) taking 2
// arguments (int, int) and returning char."
```

arrays

```
int a1[2];
a1[0] = 7;
a1[1] = 8;
int a2[] = {7, 8};
int a3[2] = {7, 8};
int* a4 = {7, 8};
int* a5 = malloc(
    sizeof(*a5) * 2);
a5[0] = 7;
a5[1] = 8;
All (a1...a5) contain {7, 8}.
```

strings

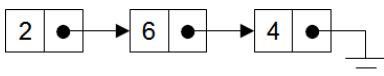
```
char s1[3];
s1[0] = 'H'; // 'H' == 72
s1[1] = 'i'; // 'i' == 105
s1[2] = '\0'; // '\0' == 0
char s2[] = {'H', 'i', '\0'};
char s3[] = "Hi";
char* s4 = "Hi";
char s5[] = {72, 105, 0};
char s6[] = {0x48, 0x69, 0x00};
char s7[] = "\x48\x69";
char* s8 = malloc(sizeof(*s8)*3);
strcpy(s8, "Hi");
char* s9 = strdup("Hi"); // non-std
All (s1...s9) contain the string "Hi".
```

structs

	Basic syntax	Basic syntax + typedef alias	Concise syntax (popular)
Define struct type	<pre>struct Point { int x, y; };</pre>	<pre>struct _P { int x, y; }; typedef struct _P Point;</pre>	<pre>typedef struct { int x, y; } Point;</pre>
Declare + initialize	<pre>struct Point p = { .x = 10, .y = 20 };</pre>	<pre>Point p = { .x = 10, .y = 20 };</pre>	
Declare object	<pre>struct Point p;</pre>		<pre>Point p;</pre>
Initialize fields	<pre>p.x = 10; p.y = 20;</pre>		
Access fields	<pre>int w = p.x; // p.x is the same as (&p) -> x</pre>		
Address (pointer)	<pre>struct Point* a_p = &p;</pre>		<pre>Point* p = &p;</pre>
Access via address	<pre>int w = a_p -> x; // a_p -> x is the same as (*a_p).x</pre>		

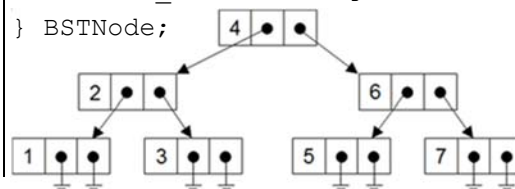
linked lists

```
typedef struct _Node {
    int value;
    struct _Node* next;
} Node;
```



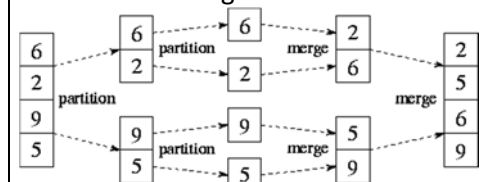
binary search tree (BST)

```
typedef struct _BSTNode {
    int value;
    struct _BSTNode* left;
    struct _BSTNode* right;
} BSTNode;
```



merge sort

Step 1: Partition the list in half.
 Step 2: Merge sort each half.
 Step 3: Merge the two sorted halves into a single sorted list.



Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
32	0x20	' '	44	0x2c	,	56	0x38	8	68	0x44	D	80	0x50	P	92	0x5c	\	104	0x68	h	116	0x74	t
33	0x21	!	45	0x2d	-	57	0x39	9	69	0x45	E	81	0x51	Q	93	0x5d	]	105	0x69	i	117	0x75	u
34	0x22	"	46	0x2e	.	58	0x3a	:	70	0x46	F	82	0x52	R	94	0x5e	^	106	0x6a	j	118	0x76	v
35	0x23	#	47	0x2f	/	59	0x3b	;	71	0x47	G	83	0x53	S	95	0x5f	_	107	0x6b	k	119	0x77	w
36	0x24	\$	48	0x30	0	60	0x3c	<	72	0x48	H	84	0x54	T	96	0x60	`	108	0x6c	l	120	0x78	x
37	0x25	%	49	0x31	1	61	0x3d	=	73	0x49	I	85	0x55	U	97	0x61	a	109	0x6d	m	121	0x79	y
38	0x26	&	50	0x32	2	62	0x3e	>	74	0x4a	J	86	0x56	V	98	0x62	b	110	0x6e	n	122	0x7a	z
39	0x27	'	51	0x33	3	63	0x3f	?	75	0x4b	K	87	0x57	W	99	0x63	c	111	0x6f	o	123	0x7b	{
40	0x28	(	52	0x34	4	64	0x40	@	76	0x4c	L	88	0x58	X	100	0x64	d	112	0x70	p	124	0x7c	
41	0x29	)	53	0x35	5	65	0x41	A	77	0x4d	M	89	0x59	Y	101	0x65	e	113	0x71	q	125	0x7d	}
42	0x2a	*	54	0x36	6	66	0x42	B	78	0x4e	N	90	0x5a	Z	102	0x66	f	114	0x72	r	126	0x7e	~
43	0x2b	+	55	0x37	7	67	0x43	C	79	0x4f	O	91	0x5b	[	103	0x67	g	115	0x73	s	127	0x7f	DEL

#define	#elif	#if	#pragma	__DATE__	__FILE__	__TIME__	
#include	#end	#ifdef	pack(1)	__FILE__	__LINE__	#	<i>macro (stringify)</i>

FILE*	fopen (const char* filename,	int	feof (FILE *stream)
	const char* mode)	int	ferror (FILE* stream)
int	fputc (int c, FILE* stream)	int	fclose (FILE* stream)
int	fprintf (FILE* stream,	size_t	fread (void* dest, size_t size,
	const char* fmt, ...)		size_t count, FILE* stream)
int	fseek (FILE* stream, long offset,	size_t	fwrite (const void* src, size_t size,
	int whence)		size_t count, FILE* stream)
long	ftell (FILE* stream)	FILE*	stderr
int	fgetc (FILE* stream)	FILE*	stdout
char*	fgets (char* buf, int n, FILE* stream)	FILE*	stdin

%d	decimal	65	decimal		bitwise or	0b1001 0b0011 == 0b1011	"address of v "	& v
%x	hex	0x41	hex	&	bitwise and	0b1001 & 0b0011 == 0b0001	"value at a "	* a
%c	character	0101	octal	^	bitwise xor	0b1001 ^ 0b0011 == 0b1010	"write v at a "	* a = v
%p	address	'A'	character	~	bitwise not	~ 0b00001111 == 0b11110000	other operators	
%s	string	'\0'	null terminator	>>	bitshift right	0b00001111 >> 2 == 0b00000011	? : ternary	3 > 4 ? 1 : 2 == 2
%zd	size_t	NULL	null address	<<	bitshift left	0b00001111 << 2 == 0b00111100	sizeof	sizeof(v) == 4

*a	a[i]	o.x	a -> x
$\Updownarrow$	$\Updownarrow$	$\Updownarrow$	$\Updownarrow$
a[0]	*(a+i)	(&o) -> x	(*a).x

Example: If a is an int ... then &a is an int*
 If b is an int* ... then &b is an int**
 If c is an int** ... then &c is an int***

The diagram illustrates the sequence of C operators from left to right, categorized into groups:

- unary operators:** `()`, `[]`, `->`, `.`, `+expr`, `++expr`, `expr++`, `-expr`, `--expr`, `expr--`, `! ~ *addr &expr`, `(type)`, `sizeof(expr)`.
- arithmetic:** `*`, `/`, `%`, `+`, `-`.
- comparison:** `<<`, `>>`, `<`, `>`, `<=`, `>=`, `==`, `!=`.
- bitwise:** `&`, `^`, `|`.
- logical:** `&&`, `||`.
- ternary:** `?:`.
- assignment:** `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `^=`, `|=`, `<<=`, `>>=`.

The sequence ends with a comma operator `,`.

how to write bug-free code

- DRY – Don't Repeat Yourself
- Take time to learn your tools.
- Fix "broken windows" (e.g., warnings)
- Plan before you begin coding.
- Get enough sleep and nutrition.
- Crash early, e.g., with assert(...).
- Use assert() to validate *your* code only.
- Free() where you malloc(), when possible.
- Design with contracts.

how to debug

- Use scientific method.
- Prove your assumptions.
- Re-read library documentation.
- Follow leads.
- The compiler isn't broken.
- Use the right debugging tool(s).
- Take notes to stop going in circles.
- Write test tools.
- Take a nap / walk / break.

memory faults / Valgrind error messages

To start Valgrind, run:
valgrind ./myprog

"Invalid write"

Buffer overflow – heap
`int* a = malloc(4 * sizeof(*a));`
`a[10] = 20; // !!!`

Write dangling pointer – heap
`int* a = malloc(...);`
`free(a);`
`a[0] = 1;`

"Invalid read"

Buffer overread - heap
`int* a = malloc(4 * sizeof(*a));`
`int b = a[10]; // !!!`

Read dangling pointer – heap
`int* a = malloc(4 * sizeof(*a));`
`free(a);`
`int b = a[0]; // !!!`

Not detected by Valgrind

Buffer overread - stack
`int a[4];`
`int b = a[10]; // !!!`

Buffer overflow – stack
`int a[4];`
`a[10] = 1; // !!!`

Segmentation fault – crash

Writing at NULL with *
`int* a = NULL;`
`*a = 10;`

Writing at NULL with ->
`Node* a = NULL;`
`a -> value = 10;`

Writing at NULL with [...]
`int* array = NULL;`
`array[0] = 1;`

Reading from NULL with *
`int* a = NULL;`
`int b = *a;`

Reading from NULL with ->
`Node* p = NULL;`
`int b = p -> value;`

Reading from NULL with [...]
`int* array = NULL;`
`int b = array[0];`

Not detecting malloc() failure
`int* a = malloc(1000000000000000000);`
`*a = 1; // a is NULL`

Stack overflow
`void foo() {`
 `foo(); // !!!`
`}`

Writing to read-only memory
`char* s = "abc";`
`s[0] = 'A';`

Calling va_arg too many times
`while(a == 0) {`
 `b = va_arg(...);`
`}`

"Conditional jump or move depends on uninitialised value(s)"

If with uninitialized condition
`int a; // garbage!!!`
`if(a == 0) {`
 `// ...`
`}`

Loop with uninitialized condition
`int a; // garbage!!!`
`while(a == 0) {`
 `// ...`
`}`

Switch with uninitialized condition
`int a; // garbage!!!`
`switch(a) {`
 `// ...`
`}`

Printing unterminated string
`char s[2];`
`s[0] = 'A'; // no '\0'`
`printf("%s", s);`

"Use of uninitialized value"

Passing uninitialized value to fn
`int a;`
`printf("%d", a);`

"Syscall param ... uninitialised byte(s)"

Return uninitialized value from fn
`void foo() {`
 `int a;`
 `return a;`
`}`

Write uninitialized value to file
`char c;`
`fwrite(&c, 1, 3, stdout);`

"Definitely lost" – leak

Lose address of block
`void foo() {`
 `int* a = malloc(...);`
`} // !!!`

"Indirectly lost" – leak

Lose address of address of block
`void foo() {`
 `void** a =`
 `malloc(...);`
 `*a = malloc(4);`
`} // !!!`

"Still reachable" – leak

Address of block still in memory
`int main() {`
 `static void* a;`
 `a = malloc(...);`
 `return EXIT_SUCCESS;`
`}`

"Invalid free()"

"glibc ... free"

Double free
`int* a = malloc(...);`
`free(a);`
`free(a); // !!!`

Free something not malloc'd
`int a = 0;`
`free(&a); // !!!`

Free wrong part
`int* a = malloc(...);`
`free(a + 3); // !!!`

"silly arg (...) to malloc()"

Negative size to malloc(...)
`void* a = malloc(-3);`
`free(a);`