C Cheatsheet

The single best book on C is The C Programming Language by Kernighan and Richie.

CODE:
Source Code that does work goes into files with ".c" suffix.
Shared declarations's (included using #include "mylib.h") in "header" files, end in ".h"

COMMENTS:
Characters to the right of // are not interpreted; they're a comment.
Text between /* and */ (possibly across lines) is commented out.

DATA TYPES: (32 bit mode)

name	size	description
char	1 byte	an ASCII value: e.g. 'a' (see: man ascii)
short	2 bytes	a signed integer up to 16,567
int/long	4 bytes	a signed integer up to 2,147,483,648
long long	8 bytes	a signed integer up to 9,223,372,036,854,775,807
float	4 bytes	a floating point (approx. to real) number
double	8 bytes	a higher precision floating point number

char, int, and double are most frequently and easily used in small programs
sizeof(double) computes the size of a double in addressable units (bytes)
Zero values represent logical false, nonzero values are logical true.
Math library (#include <math.h>, compile with -lm) prefers double.

CASTING:
Preceding a primitive expression with an alternate parenthesized type converts or "casts" value to a new value equivalent in new type:

```
int a - (int) 3.131; // Assigns a=3 without complaint
```

C++: `int a = static_cast<int>(3.131);`
Preceding any other expression with a cast forces new type for unchanged value.

```
  double b = 3.131;
  int a = *(int*)&b; //interprets the double b as an integer (not necessarily 3)
```

C++: `int a = reinterpret_cast<int>(3.131);`

OPERATIONS:

+ - * / %	Arithmetic ops. /truncates on integers, % is remainder.
++i --i	Add or subtract 1 from i, assign result to i, return new val
i++ i--	Remember i, inc or decrement i, return remembered value
&& || !	Logical ops. Left side is not executed if right side is enough.
& | ^ ~	bit logical ops: and, or, exclusive-or, invert
>> <<	Shift right and left: int n=10; n<<2 computes 40
=	Assignment. Result is the value assigned.
+= -= *= etc	Short for a = a + <*any*>; etc.
== != < > <= >=	Comparison operators C++: you write these for your types.
?:	expression version of if: (x%2==0)?"even":"odd"
,	ell evaluated, value is last: a = (b,c,d); exec's b,c,d then a=d

STATEMENTS
Angle brackets identify syntactic elements and don't appear in real statements

```
<type> <name> = <expression>; // declares a variable and gives it a value.
<name> = <expression>; // changes the value of a variable.
<expression> ;          //semicolon indicates end of a simple statement
break;                  //quits the tightest for,while or switch
continue;               //jumps to next loop test, skipping rest of loop body
return x;               //quits this function, returns x as value
{ <stmt><stmt>... }     //curly-brace groups statements into 1 compound (no ; after)
if (<condition>) <stmt>         //statement evaluated if condition is non-zero.
if (<condition>) <stmt> else <stmt>  // else part evaluated if condition is false.
while (<condition>) <stmt>    // stmt repeatedly evaluated if condition is non-zero.
do <stmt> while (<condition>);        // note semicolon. stmt eval'ed at least once.
for (<init>; <condition>; <step>) <stmt>
switch (<expression>) {
case <value>: <stmt>  // evaled if <expr> equals value;
  break;              // and break out of the switch statement.
case <value2>: <stmt2>        // falls through to next case.
case <value3>: <stmt3> break;
default: <stmt4>      // if nothing else matches. Can be anywhere in switch
  break;
}
```

FUNCTIONS
A function is a pointer to some code, parameterized by formal parameters, that may be executed by providing actual parameters. Functions must be declared before they are used, but code may be provided later. A sqrt function for positive n might be declared as:

```
  double sqrt(double n) {
     double guess;
     for (guess = n/2.0; 0.001<abs(n-guess*guess);
        guess = (n/guess+guess)/2);
     return guess;
  }
```

This function has type double (s*sqrt)(double).

```
  printf("%g\n", sqrt(7.0)); //calls sqrt; actuals are always passed by value
```

C:Functions parameters are always passed by value. Functions must return a value.
The return value need not be used. Function names with parameters returns the function pointer. Thus, an alias for sqrt may be declared:

```
  double (*root)(double) = sqrt;
  printf("%g\n", root(7.0));
```

Procedures or valueless functions return 'void'.
There must always be a main function that returns an int.

```
  int main(int argc, char **argv)  OR  int main(int argc, char *argv[])
```

Program arguments may be accessed as strings through main's array argv with argc elements. First is the program name. Function declarations are never nested.

KEY WORDS

unsigned	makes unsigned integer types.
extern	in a .h file, says data is defined elsewhwere
static	in a .c file: value can't be seen outside. Initialized once.
typedef	before declaration defines a new type name.
enum {	gives integers meaningful names.
red,	
green,	
orange	
}	

ARRAYS and POINTERS and ADDRESS COMPUTATION
Arrays indicated by right associative brackets ([]) in the type declaration

int a[10];	//a is a 10 int array. a[0] is the first element. a[9] is the last
char b[];	//in a function header, b is an array of chars with unknown length
int c[2][3];	//c is an array of 2 arrays of three ints. a[1][0] follows a[0][2]

Array variables (e.g. a,b,c above) cannot be made to point to other arrays
Strings are represented as character arrays terminated by ASCII zero.
Pointers are indicated by left associative asterisk (*) in the type declarations:

int *a;	//a is a pointer to an integer
char *b;	//b is a pointer to a character
int *c[2];	//c is an array of two pointers to ints (same as int *(c[2]);
int (*d)[2];	//d is a pointer to an array of 2 integers.

Pointers are simply addresses. Pointer variables may be assigned.
Adding 1 computes pointer to the next value (by adding sizeof(X) for type X)
adding integers to a pointer (even 0 or negative values) behave in the same way

The ampersand (&) operator gives you the address.
An array without an index or a struct without field gives you the address:

int a[10], b[20];	// two arrays
int *p = a;	// p points to first int of array a
p = b;	// p now points to the first int of array b

An array or pointer with an index n in square brackets returns the nth value:

int i = a[0];	// i gets the first value of a
i = *a;	//pointer dereference: i gets the first value of a
p = a;	//because a is an array: same as p=&a[0]
p++;	//same as p=p+1; same as p=&a[1]; same as p=a+1

Bounds are not checked. You are responsible. Don't access outside the array.

COMPILING:

```
gcc prog.c # compiles prog.c into a.out run result with ./a.out
gcc -o prog prog.c #  compiles prog.c into prog; run result with ./prog
```