C Crib Sheet:Version 7 March 2012

Abreviations:

le => Logical expression evaluating to true or false.
st => One or more C statements.
ie => Integer expression, must evaluate to an integer
args => Arguments or argument list
v => variable, *iv* => integer variable,
dv => double variable, *fv* => Float variable
Lv => long double variable
[st] => Optional C statements

Basic C Program;

```
#include <stdio.h> /*I/O Header files */
int main(void) /* Program starts here */
{ printf("\n Hello World!");
  return(0);}       /* final statement */
```

Data Types:

short, int, long, long long Integers
float, double, long double Floating pt.
void no value, missing or not there

Variable declaration and initialization

type *name[=value[],name=[value]]* **;** forgotten
static *type* *name[=value[]]* ; remembered

Statements:

v = expression ; Assign the value of expression to v

Arithmetic operators:

+, -, *, / % (Add, Subtract, Multiply, Divide, Integer divide remainder)

Logical Comparison Operators(LCO) : x LCO y
(Must compare like with like!!)

<,<= (x less than y, x less than or equal to y)
==, != (x equal y, x not equal y)
>, >= (x greater than y, x greater than or equal to y)

le1 || *le2* (*le1* or *le2*)
le1 && *le2* (*le1* and *le2*)
!*le* (not *le*)

<math.h> type double built in Math functions:

cos(*dv***), sin(***dv***), tan(***dv***), tan2(***dv1,dv2***),**
acos(*dv***), asin(***dv***), atan(***dv***), sqrt(***dv***)**
cosh(*dv***), sinh(***dv***), tanh(***dv***), pow(***dv1,dv2***),**

Changing Data types (casting)

(double)*iv* or *fv* or *Lv*	creates a double
(long double) *iv* or *fv* or *dv*	creates a long double
(int) *dv* or *fv* or *Lv*	creates an integer
(float) *iv* or *dv* or *Lv*	creates a float

Flow control:

while (*le***) {** *st* **};** test and do
do{ *st* **} while (***le***)** do and test
for (*ie1***;** *le***;** *ie3* **) {** *st* **}** for (start; stop; change)
if(*le***){***st***}** if (true) {do}
if(*le***){***st***}** if (true) {do} else etc..
[else [if(*le***){***st***}] {***st***}]**
switch(*iv***){** on value of *iv*
 case *iv1* **:***st* **; break;** do *st* if *iv* matches *iv1*
 case *iv2* **: case** *iv3* **:** *st* **; break;**
 default : *st* **; break;}** do *st* if no case matches.
break; leave the current loop or section.
continue; go to the beginning of the loop.
Do test for **do** and **for**

Preprocessor Commands:

#include <*fn*> **copy *fn* from *include* subdirectory**
#include "*fn*" **copy *fn* from *current* subdirectory**
#define NAME *value* **replace every occurrence of** NAME by *value* which can be a C statement

```
  #define MAXSIZE 1024;
  #define REAL float;
```

Explicit arrays:

***type name*[SIZE]** SIZE must be an integer value cannot be an integer variable
Array addresses run from ***name*[0]** to ***name*[SIZE-1]**

Function Declarations:

argument declarations=> ,…***type name,type name***
return-type function-name*(arg declarations***)**
{*st***}**
return-type function-name*(arg declarations***);**
Prototype declaration (Obligatory!)

Scope:

Storage Class	Keyword	Life-time	Where defined	Scope	Value Retention
Automatic		temp	inside	Local	lost
Static	static	temp	inside	Local	retained
External		perm	outside	global	retained
External	static	perm	outside	Global in modules	retained

Addressing and pointers

&A => Address of variable A. Useful for functions
int *A Defines A to hold the address of integers
scanf("%d",&A) Puts address of variable A into function scanf so scanf can load a value into that address

Main function: **quad(A,B,C,&r1,&r2)** Load values for variables A, B and C and addresses for r1 & r2 into argument list for function:
In function:

```
quad(double A,double B,double C,
  double *r1,double *r2)
. . .
*r1=(-B+sqrt( etc. . .  );
*r2=(-B- . . .
```

Variables declared to be arrays are passed by address to a function. In the main function:

```
double A[3],r[2];
quad(A,r)
```

In the quad function:

```
quad(double* A,double* r,)
. . .
Disc=A[1]*A[1]-Four*A[0]*A[2];
. . .
r[0]=(-A[1]-sqrt(Disc))/(two*A[0]);
r[1]=. . .  ;
```

Allowed pointer operations

Declaration: **float *pA, *pB;**
Assignment: **pA=&var;**
Increment: **pA=pA+1; (**written **pA++;)**
Decrement: **pA=pA-1; (**written **pA--;)**
Difference: **gap=pA-pB;**
Comparison: **pA==pB+gap;**
De-referencing: ***pA=var;**

How to borrow memory

The C **m**emory **alloc**ation function is called: `malloc()`
This function takes a type `int` number as its only argument and it returns a memory pointer of type `void`
It is good practice to recast this void pointer to the specific type pointer needed. e.g.

```
double *dv1;
int N;
dv1=(double*)malloc(N*sizeof(double));
```

If something goes wrong `malloc` returns the memory pointer `NULL` ***{ (0) (FALSE) ! }***

How to return memory

`free(dv1);` and returns a `void;`
The argument to `free` must be a pointer that was previously returned by a memory allocation function.

`#include <stdlib.h>`
`malloc()` and `free()` have their prototypes defined in the header file `stdlib.h` along with other memory allocation functions. The two other standard memory allocation functions are:

1. `realloc()` which changes the size of a `malloc()` or `calloc()` memory block.
2. `calloc()` which allocates memory and sets it to zero. `calloc()` takes two arguments, the number of memory elements and the size of each element, both integers

How to allocate a Matrix: pointers to pointers

```
double **make_matrix(int NR, int NC)
{double **M; int n;
/*allocate matrix M[1..NR][1..NC]*/
 M=  (double **)malloc(
            (NR+1)*sizeof(double *));
 M[0]=(double *)malloc(
           (NR*NC+1)*sizeof(double));
M[1]=M[0];
 for(n=2; n<=NR; n++) M[n]=M[n-1]+NC;
return(M);}
```

Matrix addition example:

```
double **matrix_addition
  (double **B, double **C, int L, int M)
{double **A;
 int i,j;
 A=make_matrix(L,M);
 for(i=1;i<=L;i++)
   {for(j=1;j<=M;j++)
       {A[i][j]=B[i][j]+C[i][j];}}
 return(A);}
```

How to free a Matrix

```
void free_matrix(double **M)
  {free(M[0]); free(M);}
```

The first `malloc` call allocates NR+1 pointers to the start of each row of the matrix. The second `malloc` call allocates NR*NC+1 memory locations of type double. The `for` loop makes each pointer in `M[i]` point to the start of row `i` in the memory block holding the numbers.

Bitwise Logic: C can do bitwise logic on unsigned integers. Declaration: `unsigned int U1,U2;`
C can change all of the 0's to 1's and vice versa in a number.
`~one = 1111111111111110` in bits.
C can shift all of the bits in a number a fixed number of places to the left or the right. Zeros are propagated in to the vacated places. Bits disappear when shifted out of the number!
The C operator to do the shifting is `>>` (right) or (left) `<<`
The C convention is that the lowest order bit is rightmost.
So if `one = 1;` then `(one << 2) = 4;`
Shifts are cheap integer multiply or divide by powers of two.

Bitwise Logical Operators:
`&` (and), `|` (or), `^` (exclusive or)
Truth tables:

```
And: &  U1: 0 0 1 1    Or: |  U1: 0 0 1 1
        U2: 0 1 0 1           U2: 0 1 0 1
 U1 & U2:0 0 0 1    U1 | U2:0 1 1 1

Exclusive or: ^          U1: 0 0 1 1
                         U2: 0 1 0 1
                    U1 ^ U2:  0 1 1 0
```

All binary logic is possible with the 4 operators `& ! ^ ~`

`char` data type

A character is a byte (8 bits). It can have 256 values from 0 to 255. In ASCII the values 0 – 31 are non printing characters such as tab, bell, line feed, page feed, etc.,
Values 32-127 are defined standard characters and values 128-255 are extended characters that vary from font to font.

Character specific I/O routines.
And character specific format descriptors for `printf` and `scanf`.
`char c; c = getchar();`
will read whatever character is typed at the keyboard (*sysin*):`char c; FILE *f;  c = getc(*f);`
will read whatever is the next character in the file pointed to by the file name pointer `*f`
`putchar(c);` Puts the character `c` on to the screen.
`putc(c,*f);` Puts the character `c` in to the file pointed to by the file name pointer `*f`
A series of characters as a single variable called a string. The string literal is set off by the double quote character. All strings in C are assumed to be terminated by `NULL` sometimes written `\0.` This means that the size of all strings is one byte larger than the number of characters.

C Keywords (reserved for the Compiler)

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	while

Obsolete words are underlined!

Compiler Directive Keywords:

#include	#define	#undef
#if	#ifdef	#ifndef
#elif	#else	#endif
#error	#line	#pragma
#		

GNU Scientific Library in C:
www.gnu.org/software/gsl/
This is an updated C-Language version of SLATEC and is managed by scientists at Los Alamos National Laboratory. It is installed in the ICT Cygwin Shell on Windows PCs.

General Numerical Software Repository
www.netlib.org Most of the code is in FORTRAN, but:

f2c Converts Fortran 77 Code into legal C code. See:
http://www.netlib.org/f2c/