Contents

1 C Programming

```
//import standard IO tools (eg printf, scanf...)
#include <stdio.h>

//begin main program - (int is optional)
int main ()
{
  //print hello world , then make new line
  printf("%s\n", "Hello World")
}
```

1.1 For/If/While statements:

```
for ( i=0 , i<5 , ++1 ) {printf("%d", i)}
for ( A, B, C) { D }

do A -> check B -> IF TRUE do D -> do C -> check B ...
                -> IF FALSE continue

if ( condition ) {group;}
else {}                        //optional
```

1.2 Condition checks:

```
<   Less than
<=  Less than or equal to
==  Equal to
>=  Greater than or equal to
>   Greater than
!=  inequality

Grouped using: (&&  And) (||  Or) (!   Not)
```

1.3 Different Data Types/Variables:

Data is stored in terms of 1s and 0s (binary). Bits are stored in groups of 8 for convenience (bytes).

Powers of 2: ($2^0 + 2^1 + ... + 2^7$)

Hexadecimal: (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10 ...)

Decimal: (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ...)

Booleans: Literally just ints in C. zero is False , non-zero is True. Checking statements give true as 1. (trick for efficiency in early programming). Example of use: count odd numbers in an array of 100 ints:

```
int sum = 0, i;
for (i = 0; i<100 ; ++i)
{sum = sum + (a[i]%2 == 1}
```

Names: Any string of mixed alphabetical, digit, underscore, NOT beginning with a digit. Letters are case sensitive.

```
int i,j, hello, Hello, _hello;
double x;
char c;
```

float - 32bit 'single precision' used when memory is scarce (eg satellite or GPU). Format is represent as $2^e 1. < ---32bits--- >$

1.4 IEEE standard floating point representations:

float : [sign 1b][exponent 8b][significant figures 23b]

double: [sign 1b][exponent 11b][significant figures 52b]

In single precision, the true exponent is e (which can be negative) is added to 127 for the stored biased exponent. sign: 1 negative, 0 nonnegative. Not 2s compliment. a number x is (except for zero) converted to $2^e \times 1.x_1x_2x_3$ ... where the $1.x_1x_2x_3...$ is less than 2 and greater than or equal to 1.

stored: sign , 127+e , 1.x$_1 x_2 ... x_{23}$

Finally, 'Little Endian'

EG: Convert 80/9

sign 0 (non-negative)

devide by 8 -¿ 10/9 (exponent 3 - 8=2^3)

biased: 127+3 = 130 = 10000010

Now we calculate digits:

$\frac{10}{9} = 1. - - - - - - - - - - - -=> \frac{1}{9}$

$\frac{2}{9} = 1.0 - - - - - - - - -$

$\frac{4}{9} = 1.00 - - - - - - - -$

$\frac{8}{9} = 1.000 - - - - - - -$

$\frac{16}{9} = 1.0001 - - - - -- ==> \frac{7}{9}$

$\frac{14}{9} = 1.00011 - - - - ==> \frac{5}{9}$

$\frac{10}{9} = 1.000111 - -- ==> \frac{1}{9}$ RECURSSION

10/9 = 1.000111 recurring so $7/64 + 7/62^2 + 7/64^3 etc...whichequals 1/9$

So in conclusion we get the following: $2^3 \times 1.000111000111000111000111(1)$... but last (24th) digit is 1 so we round up the last digit to 23rd but that is also 1 so we round up to 22nd digit, so what we get in conclusion is:

float 0 10000010 00011100011100011100100

Now for a double, the process is similar: sign = 0, exponent 3+1023, mantissa 1 + 1/9,

double 0 10000000010 000111 000111 000111 000111 000111 000111 000111 000111 000111 0001(1)

double 0100 0000 0010 0001 1100 0111 0001 1100 0111 0001 1100 0111 0001 1100 0111 0010

in hex: 4 0 2 1 C 7 1 C 7 1 C 7 1 C 7 1 2

IEE standard: Guarantees that addition prouces correctly rounded results

1.5 Arithmetic Assignments

```
+    Addition
-    Subtraction
*    Multiplication
/    Division (depends on type)
     Gives integer places for int, float precison or float.
%    Remainder (int only, divisor not zero)
     m % n formula is m - n*(m/n). (If m is - so is Ans)

     Often one wants (m-1) % n to to equal (n - 1).
     If m = 0 mod n then use (m + n - 1) % n
```

1.5.1 Calculate day of the week: yy mm dd

```
00 <= yy <= 99, 01 <= mm <= 12, 1 <= dd <= 31

Jan: 0%7 = 0   Feb: 31%7 = 3   Mar: 59%7 = 3   Apr: 90%7 = 6
monthOffset = [0, 3, 3, 6, 1, 4, 6, 2, 5, 0, 3, 5]
correctionFactor == 6 except for Jan, Feb in leap year (5)

Formula: (yy + yy/4 + [mm-1] + dd + correctionFactor)%7
```

4-digit year: refined Gregorian Calendar (1582) Conditions:

```
(yy%400 == 0) || ( (yy%4 == 0)&&(yy%100 != 0) )
```

1.6 Arrays: Blocks of data

```
//a is an array of 100 ints (400 bytes), a[0] to a[99]
int a[100];

//clear array
for (i=0; i<=99, ++i) { a[i] = 0}
```

Technicalities of Arrays:

(i) The 'value' of a is the address of a[0]. An array has a 'value' which is it's address.

(ii) The address of a[i] is address of a[0] + 4*i

(iii) In a C program, reference to a[i] is never checked. 4 bytes for correct range a[-1] or a[1000]

Character strings are character arrays: char hello[6] = "hello"; Remember that there is a Null character to end the string "\ø".

char hello[6] = { "h","e","l","l","o","\ø"}

int monthOffset[12] = {0,3,3,6,1,4,6,2,5,0,3,5 }

The ASCII code maps various printable control characters to 8-bit values. EG "hello" in hex is: 68 65 6C 6C 6F 00. A good example of use of array: Read data (eg doubles) and compute average (doesn't need array) standard deviation (needs array)

1.7 GCD Algorithm

```
#include <stdio.h>
main () {
  int x=165, y=39;
  while (y>0) {int z = x%y; x=y; y=z}
  printf("gcd is %d \n", x
}
```

Can also be done with ARRAYS, with the array xx showing the parts of the computation and the way the algorithm works:

```
#include <stdio.h>
main () {
  int x=165, y=39, n=0;
  int xx[100]; xx[0] = x; xx[1] = y;
  while (xx[n+1] != 0)
    {int z = xx[n]%xx[n+1]; xx[n+2] = z; ++n;}
  printf("gcd is %d \n", x
}
Result: xx[0]=165, xx[1]=39, xx[2]=9. xx[3]=3, xx[4]=0;
```

There is an improved version of the Euclidean algorith which calculates with gcd of m and n, integers s.t. gcd m,n = sm + tn;

1.8 2D Arrays

In C the start address of the data is stored in the array variable C: Arrays, addresses regarded as similar (which overcame the most serious limitation of Pascal, which is the superior language). In Pascal arrays had to be specified exactly to size. C regards a 2-D Array as an array of 1D Arrays

$int\ a[3][7]$

a is an array of 3 arrays of ints, all stored together in a single block of storage size 3x7 units or 84 bytes, indexing from a[0][0] to a[2][6]. stored in row-mayor form:

i.e: $a[0][0], ..., a[0][6], a[1][0], .., a[1][6], a[2][0], ..., a[2][6]$

so if address of a[0][0] is 1234, then a[1] is 1234 + (7*4) = 1262, giving us a general formula:

$startAddress + i * lengthOfRow + j * sizeOfEntry$

(sizes: char 1; short 2; int 4; float 4; double 8;)

For finding the address of an array after another array, we assume b starts where a ends. Exmaple:

int a[3][4]; double b[100];

array element a[3][4] corresponds to which b[j]?

a starts 1234 and ends (1234+(3x4x4)) = 1318 (where b starts)

but 1234 + 28(3) + 4(4) = 1334 = 1318 + 16

looking at b[j] = 1318 + 8j so j=2 so it corresponds to b[2]

1.9 Command-line Arguments

a.out -row 14

The "command-line arguments" are accecible to the program — -.c if:

$int\ main(intargc, char * argv[])$

argv is an array of character strings argv[0] is the name of the program (eg a.out) argc is the number of args (¿= 1 – argv[0]) a.out a65 39 command-line arguments int main (int argc, char * argv[]) argv [0] is the name of the executable program EG: a.out in this example argv[1] = "165" argv[2] = "39"

char*argv[][] means that argv is an array of character strings

you can use the same style to create arrays of character strings

```
char hello[6] = "hello"
char*weekday[7] =
{"Mon","Tue","Wed","Thu","Fri","Sat","Sun"}
```

The strings can be different lengths argc is the number of commandline arguements (including a.out)

EXAMPLE gcd calculator include ¡stdio.h¿ include ¡stdlib.h¿ //-standard library, want atoi int main (argc, char*argv[]) //there should be some error checking eg argc==3 printf(gcd of return 0; //shows ran with no errors aturingaturing

1.10 Routines Functions

printf is routine. atoi is a function ("returns a value"). scanf is also a function.

int main() is the main routine - but is actually a function (returns an integer value).One can write "subprograms" to do different tasks. Style is same as for main program. Routines do something and do not return any meaningful value. Functions do something and do return a value.

void name $(listOfArguments)$

int/char/etc. name $(listOfArguments)$

```
int gcd (int m, int n) {
int x=m, y=n;
while (y>0) {
    int z = x%y;
    x=y; y=z;
    }
    return x;
```

```
}
int main (int argc, char*argv[] ){
int m = atoi(argv[1]);
int n = atoi(argv[2]);
printf("gcd %d %d is %d\n",m,n,gcd(m,n);
}

void clearMatrix (int m, int n, double a[10][10]){
for(int i=0; i<m; ++i){
    for(int j=0; j<n; ++j){
        a[i][j]=0;
}}}
```

An interesting example is recursion. A function which calls itself.

```
int factorial(int n){ //assume n>0
if (n==0) {return 1;}
else {return n*factorial(n-1)}
}
```

1.11 Determinant

$$det(A) = \begin{cases} \text{a[0][0] n = 1.} \\ \sum_{j=0}^{n-1} (-1)^j a[0][j] det(minorMatrix(0, j, A)). \end{cases} \quad (1)$$

$$\det \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} = [\textbf{1.12} \quad LocalvsGlobalvariables$$

```
include ---

int version_no 2;
voide xxx(---)
{
    for (i=0, i>14, --i) {}
    main () {
```

```
        for (i=0, i<10, ++i) {xxx(---)}
    }
}
```

Local variablse = good remote variables = bad, liable to corruption
Otherwise, variables are local routines — or even more restricted

```
for (i=0, i<10, ++i) {
    int j; //keeping j as close as possible
    for (j=0, j<10, ++j) {
        a[i][j] = 0}
}
OR EVEN (gcc -c 99)
for (int j=0, j<10, ++j) {}
```

Where several varieables have the same name, the closest applies: Simulating a code framgent, it takes the closest variable

```
int i, j = 25;
int sum = 0;
for (i=0; i<3; ++i) {
    int j,
    for (j = 0; j<i, ++j) {
        sum = sum + j;
    }
    printf("1%d j%d sum%d\n",i,j,sum);
}
```

$$\sum_{i=0}^{4-1}(\sum_{j=0}^{i-1} j)$$

output: i3 j25 sum1

Simulation:

i	$i < 3$	sum	j_1	j_2	$j_2 < j_1$
		0	25		
0	1			0	0
1	1			0	1
		0			1
				1	
					0
2	1			0	1
		0		1	1
		1		2	0
3	0				

'Runtime stack' allows recursive programming (routines call themselves). Recursion was not allowed in Fortran: still populat for scientific programming. More streamlined, simpler, faster.

Global Variables.

Variable local to a routine.

Routine arguments: initalaised local variables.

The local variables are kept on a stack frame:

```
main [frame]
 calls A
A        [frame] pushed
A calls B
B              [frame] pushed
B terminates
B              [frame] popped
A resumes
A        [frame] resumed
A terminates
A        [frame] popped
main [frame] resumes
```

EXAMPLE

```
double topow ( double x, int n) {
```

```
    double y;
    if (n == 0) {return 1;}
    else {
        y = topow(x, n/2);
        if (n%2 == 0) {return y*y}
        else {return y*y*x}
}}

int main ()
{   printf("%f\n", topow(3,5));}
```

main calls topow (3, 5) topow (3,5)

x	n	y	nmod2	y*y	y*y*x	return
3	5					
3	2					
3	1					
3	0					1
3	1	1	1		3	3
3	2	3	0	9		9
3	5	9	1		273	273

1.13 Automatic Stack Variables

An automatic variable is stored in a stack frame - it is local and last until the end of a routine call. If initialised, it is uninitialised every time the routine is called.

Global Variables last to the end of the program. They are initialised just once.

Routines can include static variables (kept elsewhere other than the stack frame) which are initialised once, and last until the end of the program, and keep their value between calls.

```
#include <stdio.h>
int counttime ();
{
    static int n = 0; //only if it hasn't been yet called
```

```
    ++n;
    return n;
}

main ()
{
    int i;
    for (i=0, i<3, ++i);
    printf("counttime is called for"
           "the %d-th time\n", counttime() );
}
counttime is called for the 1-th time
counttime is called for the 2-th time
counttime is called for the 3-th time
```

1.14 Pointers in C

Asterisk in C can mean something besides multiplication. This indicates an address or "pointer" type.

$$chara[200], *x, y, **z;$$

Value of a is an address address. x is the address of a character or block of characters. In C, pointer types can be indexed.

```
int a[3], *x
printf("%d\n",a[2]);
printf("%d\n",x[2]);
```

Neither are initialised. First will print some delicious garbage. Second will either print garbage of lead to a segmentation fault.

More about pointers and cast. Note: using pointers, call by reference can be simulated.

```
void increment (int *x)
```

```
{ *x = *x + 1 }
main()
{ int z = 15;
printf("%d\n", z) //15
increment(&z);
printf("%d\n", z) //16
}
```

1.15 Conversions

var = expression

The types should usually match, but where numerical values are involved, there can be a conversion between int, char,double (short, long, float) if variable is double; expression an int, then converted to double.

If var is int, expression double, then the double is rounded to int. Is it rounded up or down? Neither and both - it is rounded towards zero (negative rounded up, positive rounded down).

Floats are uninteresting. C always converts float to doubles when evaluating an expression (float only saves space. Handy on things where memory is limited, like a spaceship or graphics card)

and short to int.

Subroutines inf f (double x) return x;

double y = f(1) - compiler expects double - the int 1 is converted to a double 1.0 - f converts x of 1.0 to an int 1 and returns that - y is a double - 1 is converted to 1.0 and sent to address of y

The important info about f can be given in a function: int f(double x);

1.15.1 Conversions

In expressions of mixed arithmetic type, int gets promoted to doubles when combined with ints (Promotion = converting int to a double):

$1.0/2 + 3 = 0.5 + 3 = 0.5 + 3.0 = 3.5$

$1/2 + 3.0 = 0 + 3.0 = 3.0$

C regard char as an 8bit integer. You can assign an int to a char: (char x = 1234); x gets the low order byte due to the little endian property (on intel machines anyway)

int x = 'a'; x gets (big endian) 00 00 00 61, but if character is something funny $\geq$ 128, then x picks up a negative value. For example, if char is (f3) in hex. Then x becomes ff ff ff f3. Chars as ints very useful for looking up tables, but sign extension ruins this.

ONE CAN DEFINE unsigned char a; x = a; Then $0 \leq x \leq 255$.

1.16 DATA TYPES

unsigned - char short int long not for float double pinter

unsigned int 0 to $2^32 - 1 signedint - 2^31 to 2^31 - 1$

characters treated as 8-bit integers: $2^8 comp$

can assign a char to an int

int a = b;

if b is a character in the range 128-255 then beware sign extension.

In mixed arithmetics expressions double, int, int is "promoted"

so double mean = sum/nj is correct

```
1.0 + 1/2 is 1.0
1 + 1.0/2 = 1.5
```

a CAST explicitly converts an expression to a given type.

```
double x = (int) 2.3 => x == 2.0
double x = (double) 1/2
```

Surprisingly, x gets the value 0.5! Casts have a higer precedence than devision (will go through this later).

The interesting use of casts is with pointer types.

A cast on a pointer doesn't change the value on a pointer, instead it changes associated things.

EXAMPLE

```
int a[2] = {1,2};
```

```
int *b = a;
        now b "tracks" a
b[0] = a[0]
char *c = (char *) a;

c and a share the same space;
a[0] = 01 00 00 00 (Little Endian)
a[1] = 02 00 00 00

so c[1] = 00 (the second box)

EXAMPLE car1.c;
#include <stdio.h>
main ()
{
        int x[2] = {60000, 40000};
        char *y = (char*)x;
        printf("%d &d \n", x[0], y[0]);
        printf("%d %d \n", x[1], y[1]);
}
OUTPUT:
60000  96
70000 -22

60000 = 60 EA 00 00 in little endian
70000 = 70 11 01 00 in little endian
96 = 60 in hex
-22 = EA in hex
```

Pointers can be used in complex interlinked structures. We'll focus mainly on the connection with arrays.

1.17 DYNAMIC ALLOCATION OF MEMORY.

strlib.h allocation functions NOT malloc or free use calloc

calloc(n, size) finds an unused portion of memory, (length n x size bytes) and returns its starting address, having initialised to 0;

```
EXAMPLE
need:
strlen string string.h;
snprintf stdio.h
fgets stdio.h

char buffer[200];
fgets (buffer, 200, stdin);
char *copy = (char*)calloc(1+strlen(buffer))
snprintf(copy, srelen(buffer), "%s", buffer)

LAST ASSIGNMENT:
Convert/print pieces of data in x.

Thu 14th Mar
COPYING STRINGS
int strlen( char s[] ) //finds length of a string
{       int n = 0;
        while( s[n] != '\0') { ++n }
        return n;
}

fgets ( buffer, maxLength, inputFile (stdin) )
-       "buffer" is a character array of lenght,
        200 characters lets say: char buffer[200]

gets ( buffer, stdin) - UNCONTROLLED
fgets ( buffer, 200, stdin)
```

reads next line from keyboard or redirected file, up to next new line if any. NEVER more than 199 characters (could leave you vulnerable to virus etc..) IF it is greater than 200, it reads to next new line and truncates. It ALWAYS adds a null character to the end.

PRINT FUNCTIONS

- printf - prints to terminal.
- fprintf - prints to file.
- sprintf - prints to string (takes data, formats it, puts result in a string). this is also DANGEROUS so NEVER use it.
- snprintf - length-controlled printf. Similar to how fegts is length-controlled read to string.
- strcpy DO NOT USE
- strncpy which can be donw with snprintf

To copy a string x to a string y (for y ¿= x):

```
int size = strlen(x) + 1;
snprintf(y, size, "%s", x);
```

CALLOC - for allocation
Allocates an unused block in memory

```
(void *) calloc (int n, int size)
```

a pointer to no particular type
Calloc Finds an unused clock of ($n \times sizeOfBytes$), clears them to 0, and returns start address
For example: calloc(10, 8) might be for an array of 10 doubles.

USING CALLOC

```
double *a = (double *) calloc(10,8)

sizeof() pseudo-function; argument is not a variable but a t
double *b = (double *) calloc(n, sizeof(double) )

sizeof(char) = 1;          char*
```

```
sizeof(short) = 2;        short*
sizeof(int) = 4;          int*
sizeof(float) = 4;        float*
sizeof(double) = 8;       double*
sizeof(long) = 8;         long*

address size depends on the machine
(32bit = 4 Bytes or 64bit = 8 Bytes)
```

Text Processing - The very last line may or may not end with newline. It is best to remove newlines, and add them back later if necessary.

-decr DElete Carriage Return.

```
void decr ( char *x)
{       while(*x != '\0')
        {
                if (*x == '\n')
                {
                        *x = '\0';
                }
                else
                {
                        ++x;
                }
        }
}

void decr (char x[])
{
        int n = 0;
        while (x[n] != '\0')
        {       if (x[n] != '\0')
                {       x[n]  = '\0';}
                else
```

```
                {++n}

        }
}
```

COPY STRING FUNCTION

```
char *copy-string (char *str)
{
        int size = strlen(str) + 1;
        char *copy;
        copy = (char*) callac(1, size);
        snprintf(copy, size, "%s", str);
        return copy;
}
```

Remember, scanf is a funtion. So is fgets. fgets returns a char* pointer, then FLAGS end-of-data with a fixed value NULL, which cannot be the address of anything.

```
int main()
{
  char *line[1000];
  int i;
  char buffer[200];
  int count = 0;
  while (count < 1000 && fgets(buffer,200,stdin) != NULL)
  {
    decr (buffer);
    line[count] = copy-string[buffer];
    ++count;
  }
  // print in reverse order
}
```

1.18 DYNAMIC 2-DIMENSIONAL ARRAYS

```
double **c;
c[0][2]; garbage as nothing is initialised
```

if c is to funtion as a 2-Dimensional array, then for each i, c[i] must be a 1-Dimensional array of doubles. So c must be initialised as an array of pointers, say m, n given.

```
C: m rows ot array of doubles.
[]
[]
[]
```

EXAMPLE

```
double a[2][3];
double **c;
a = { {1,2,3},[4,5,6} }

a is stored as:
[1][2][3][4][5][6]
but looks to us as:
[1][2][3]
[4][5][6]

c(0) [] --> [1][2][3]
c(1) [] --> [4][5][6]

double ** create_matrix (int m, int n)
{       //an array of rows
        double ** mat = (double **) calloc(m, sizeof(double
        int i;
        for ( i=0, i<m , ++i )
        {
                mat[i] = (double *) calloc(n, sizeof(double
        }
        return mat;
}
```

what is returned behaves like a 2-Dimensional Array when it comes to indexing.

```
void print-matrix (char header[], int m, int n, double **a)
{

}
3 3
2 7 1 8 2 8 1 8 2

main()
{
//mxk  ,  kxn
int m,k,kk,n
//read the first matrix
scanf("%d %d", &m, &k);
double **mat1 = create_matrix (m,k);
int i;
for (i=0, i<m, ++i)
{
        for(int j=0; j<k ; ++j)
        {
                scanf("%lf", &mat1[i][j]);
        }
}

//Similarly for matrix 2
scanf("%d %d", &m, &k);
double **mat2 = create_matrix (kk,n);
int i;
for (i=0, i<m, ++i)
{
        for(int j=0; j<k ; ++j)
        {
```

```
                scanf("%lf", &mat1[i][j]);
        }
}

//now we multiply them if k = kk
double **mat3
if (kk == k)
{
  double **mat3 = create_matrix;
  int i,j;
  for (int i = 0; i < m; ++i)
  {
    for (int j = 0; j < n; ++j)
        {
          int mm;
          for (mm = 0; mm<k ; ++mm)
                {
              mat3[i][j] += mat1[i][mm]*mat2[mm][j]
                }
        }
  }
}
PRINT RESULTS
}
```

1.19 Typedef

char, short, ..., arrays, pointers are the standard types. Thee are 2 ways to create new 'types': Union (bad) and Struct (good)

```
EXAMPLE
typedef struct {double re,im;} COMPLEX; // complex numbers

suppose complex a = {1,2};
re;im are called fields.
```

how do you find them? answer: a.re and a.im . When structured types occur, one usually creates them by calloc:

```
calloc:     complex *a = make_complex(1,2)
```

Hou do you get at the fields then?

```
(*a).re          (*a).im
but there is a preferred notation:
a->re            a->im

COMPLEX * make_complex (double re, double im)
{
    COMPLEX * res = (COMPLEX *) calloc(1,sizeof(COMPLEZ));
    res -> re = re;
    res -> im = im;
    return res;
}

COMPLEX * sum (COMPLEX * a; COMPLEX * b)
{
    return make_complex(a->re + b->re, a->im + be-> im);
    //or return make_complex(*a.re + *b.re, *a.im + *b.im);
}
```

Similarly, one can make functions for the product, inverse, modulus, square root etc... Struct allows one to keep related date close together — in one block of memory

```
EXAMPLE
Matrix + dimensions in one neat package.

typedef struct {int m,n; double ** entry;} MATRIX,

MATRIX * zeroes (int m, int n)
{
matrix * mat = (MATRIX*) calloc (1, sizeof(MATRIX));
mat->m = m;
```

```
mat->n = n;

double ** entry = (double**) calloc (m, sizeof(double*));

for (int i = 0; i<m; ++i)
{
    entry[i] = (double *) calloc(n, sizeof(double) );
    mat->entry = entry;
    return mat;
    //NB: 0 in double is 64 0-bits
    //calloc sets all initialised values to zero
}
}

MATRIX * add (MATRIX * a; MATRIX * b)
{Etc...}
```

This example of adding however is very wasteful of memory if blocks of memory have been allocated and then are forgotten. Called "memory leak".

Next we review initialising arrays:

```
char *weekday [7] = {"Sun","Mon","Tue", &c};
```

This is an array of character strings. Also allowed is:

```
char *weekday [] = {...}; //not preferable

double a[2] = {1,2};
char ** weekday = ??; maybe works
int a[2][3] = {{1,2,3},{4,5,6}};

char good[] = "good";
char also[] = {'a','l','s','o','\0'};
char bad[3] = {'b','a','d'}; // not printable;
```

1.20 Further Typedef

```
typedef struct {int capacity; char *contents;} STRING;

//can also just individually make an item of the
//'type' without defining it
struct {int capacity; char *contents;} a;

a.capacity //garbage
a.contents //nice error
```

SAMPLE PROGRAM

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

typedef struct {int capacity, char* contents} STRING;

void decr (char**) // delete new line
{----}

STRING * make-string() {
    STRING *str = (STRING*) calloc(1, sizeof(string));
    char* contents = (char*) calloc(100,1);

    str-> capacity = 100;
    str-> contents = contents;
    return str;
}

void append-word (STRING *str, char *word)
// or void append-word (STRING *str, char word[])
//string "we know", word "that" -> "we know that"
{
    //complication
```

```
//only add space if not empty
int newlen;
if (strlen(str->contents == 0) ) //or equals '/0'
{
    newlen = strlen(word);
}
else
{
    newlen = strlen(str->contents)+1+strlen(word);
}

if (newlen >= str->capacity)
{
  char *newcontents = (char*) calloc(newlen+100,1);
  if (str->contents[0] == "\0")
  {
    snprintf(newcontents, newlen+1, "%s", word);
  }
  else
  {
    snprintf(newcontents, newlen+1,
   "%s %s", str->contents,word);
  }
  free (str->contents) //avoids memory leaks
  str->contents = newcontents;
  str->capacity = newlen+1;
}
else
{
  int len = strlen(str->contents);
  if len == 0
  {
    snprintf(str->contents, newlen+1, "%s", word);
  }
  else
```

```
      {//copy the word to the end of the string w/ space
        strncpy( &(str->contents[len]); //risky
                 strlen(word)+2," %s",word);
      }
    }
}
```

1.21 'Protected' String Structure

A 'protected' string structure

```
typedef struct {int capacity; char * contents;} STRING;
decr
make-string()
append-word(STRING *str, char *word)

int main (int argc, char * argv[])
{
if (argc != 2) {
fprintf(stderr,
        "%s requires one argument line length, abort\n"),
        argv[0]
        return -1;
}
else
{
  int line-length = atoi(argv[1]);
  STRING *str[1000];
  char buffer [200];
  int maxIndex = 0;
  str[0] = make_string();
  while ( fgets(buffer,200,stdin) != NULL )
  {
    int buflen = strlen(buffer);
    decr(buffer); //remove newline
    int first_in_word = 0;
```

```
    while (first_in_word < buffer)
    {
      //pass blanks
      while (buffer[first_in_word] == ' ')
      {++first_in_word}
      if (first_in_word < buflen)
      {
        char word[200];
        int i = first_in_word,
        while( buffer[i] != ' ' && buffer[i] != '\0')
        {
          word[i - first-in-word] = buffer[i];
          ++i;
        }
        //end of word
        word[i - first_in_word] = '\0';

        if ( strlen(strlen[maxIndex]->contents + strlen(word
                    >= line-length) )
        {
          ++maxIndex; //new string
          str[maxindex] = make_string()
        }
        append-word(strlen[maxindex], word);
        first_in_word = i;
      }
    }
  }
  int i;
  for (i=0, i<=maxIndex, ++i)
  {
    printf("%s\n", str[i]->contents);
    return 0;
  }
}
```

}

1.22 Operation order

Arithmetic

DIRECTION (HIGHEST)

	x	n
1	LR	[] . -> postfix++,–
2	RL	! prefix ++, –, casts,
3	LR	* / %
4	LR	+ -
5	LR	¡ ¡= ¿= ¿
6	LR	== !=
7	LR	boolean and
8	LR	—— or
9	RL	assignment operators = += -=

* 'diference' *x where x is, say, pointer to int (*x is what is stored in address) & address of

Assignment has an effect on a value. The value assigned x=y=z=0 sets all of x,y,z to zero.

x++, x–, ++x, –x also have effects on values. x++ adds 1 to x; value is the value before increment; SO x = 2 y = x++; gives an end of x:3, y:2

```
(i)   while (*x++ != '\0')
      while ( (*(x++)) != '\0') legal
(ii)  a = b = c == 0
      a =(b =(c == 0)) legal
(iii) a = b == c = 0
      a = ((b==c) = 0) //illegal
              ^not an lval
    cannot be assigned to so illegal
(iv)  a=b=c==d && a||f||g
      a= ( b= ( ((((c==d)&&e)||f)||g) ))
```

```
(v)   *x[3] -> y[4]
    *( x[3]->y[4] ) //probably illegal
```

1.23 Pointers

IF p is of type int * then p+1 = p[1], which is 4 bytes beyond p. This is an unnecessary addition to c. Also, x++, if x is a pointer sets x to x+1 (in the sense just introduces).

```
void decr (char *x)
{
    while (*x != '\0')
    {
        if (*x == '\n')
        {*x = '\0'}
        else
        {++x}
    }
}
```

1.24 Random Number Generators

I use drand48()

this has double precision, range is half-open: [0,1) in stdlib.h To be useful, the result of reciprocated calls: $x_0, x_1, x_2, ...$ should always be the same, but the step from x_n to x_{n+1} is hard to guess. This gives us pseudo-random numbers.

Until recently, linear congruential method was used:

$$x_{n+1} = ax_n + c(modm)$$

Typically, $m = 2^{31}$ and numbers,integers in the range 0 to $2^{31} - 1$ which are converted to doubles somehow. We often want random numbers in the range (integers) 0,...,k-1. The obvious way for this (for integer rand()) is:

$$rand()\%k$$

BAD if k = 2 (and m is a power of 2)

$$x_{n+1} = ax_n + c(modm)$$

x mod 2	a mod 2	c mod 2	$x_n mod2$
0	0	0	0
1	0	0	1
0	0	1	0,1,0,1
1	0	1	1,0,1,0
0	1	0	0
1	1	0	1
0	1	1	0,1,0,1
1	1	1	1,0,1,0

The reccommended way is (int) (drand48()*k). This phenomenom seems to have dissapeared =¿ linear copying method is either not used or is modified.

```
PROGRAM TO MEASURE PI - RANDOM
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

double randval ()
{
static int first = 1;
if (first) {
    struct timeval tv;
    gettimeofday (&tv, NULL);
    srand48(tv.tv_usec);
    first = 0;
}
//timevalue tv  sec gives seconds since 1970
//timevalue tc usec gives microseconds;
```

```
return drand48();
}
```

Resetting each time ruins the randomness because it will be called several times within the same microsecond, so drand48() returns groups of equal numbers.

MONTE CARLO PROGRAM

```
int main (int argc, char * argv[])
{
int n = atoi(argv[1]);
//number of repititions
int i;
double sum = 0;
for (i = 0; i < n; ++i)
{
     double x = randval();
     double y = randval();
     if ((x*x + y*y) <= 1)
          {sum += 1;}
}
printf("%d trials, pi is %f\n", n, 4*sum);
}
```

drand48() is uniform on [0,1). How about random N(0,1)? There are ways of doing this (Wikipedia normal dist)

MARSAGLIA METHOD

1) Get random u,v in interval (-1,1) discard pairs outside disc.
2) Random pairs (u,v) in (open) disc (-1,1)

$$(x, y) = (u, v).\sqrt{\frac{ln(-S)}{S}}$$

where $s = u^2 + v^2$, x, y are independant variables.

```
void two_normally (double *x, double *y)
{
```

```
double v1, v2, s, mul;
//sorry v1:u, v2: v;
int found = 0;
while (!found)
    {
    v1 = 2*drand48() - 1;
    v2 = 2*drand48() - 1;
    s = v1*v1 + v2*v2;
    found = (s<1);
    }
mul = sqrt( -2 * log s / s)
*x = v1 * mul;
*y = v2 * mul;
}

void main (int argc, char * argv[])
{
// set seed in main program
    struct timeval tv;
    gettimeofday (&tv, NULL);
    srand48(tv.tv_usec);
    first = 0;

    n = atoi( argv[1] );
    // number of repititions

    int i;
    for (i=0 ; i<n ; i+= 2){
        two_normally (x,y);
        sum += x;
        sumsquares += x*x;
        printf("%f\n", x);

        if(i+1<n) //in case n odd
        {
```

```
                sum += y;
                sumsquares += y*y;
                printf("%\n", y);
            }

            //print avg & stdev
        }
}
```

BOX-MULLER METHOD u, v random (0,1)

$$x = \sqrt{\frac{-2lnu}{u}} * cos(2\pi v)$$

$$y = \sqrt{\frac{-2lnu}{u}} * sin(2\pi v)$$

These are essentially the same method. Wikipedia says it workd because $x^2 + y^2$ had PDF $\frac{1}{2}e^{-v}$, which is the same as x^2