20 Dynamic allocation of memory: malloc and calloc

As noted in the last lecture, several new functions will be used in this section.

- `strlen` (string.h), the length of a string.
- `fgets(buffer, max length, input file (stdin))`
- `snprintf(char array, max length, format, item_1, ...)`
- `calloc(count, size)`
- `sizeof()` pseudo-function.

There are 3 functions (use `stlib.h`) we can use for getting areas of storage:

- `malloc (), free ()` not covered in these lectures.
- `calloc ( int n, int s )` reserves $n \times s$ bytes of memory, **initialises** them to 0, and returns the address where they start.

There is a built-in function (sort of function) **sizeof (type)** which returns the number of bytes of storage occupied by an object of the given type.

To **cast** an expression is to convert it to another type. This is done with pointers to make types match. A **cast** is a type description in parentheses.

For example, `(double) 22` converts the integer expression to a double. More about this later.

Suppose you want an array of n doubles. Here is a function which does this.

```
double * array ( int n )
{
  double * a;
  a = ( double * ) calloc ( n, sizeof ( double ) );
  return a;
}
```

- `calloc( int n, int s )` obtains a block of $n * s$ bytes of free storage,
- **initialised** to all zeroes,
- and returns the address of the block.
- The type returned by the `calloc()` function is the most general pointer type possible: `void *`.
- Hence, to satisfy the rules of C, it is necessary to 'cast' it to the correct array type.

20.1 Character strings

```
The type (char *) means pointer to character, equated
to array of characters or character string.


The header file
#include <string.h>
describes various functions of strings:


int strlen ( char * str ) ----------- length
void snprintf(<buffer>,<size>,<format>,<item>,...,<item>)
        --- formatted conversion to a single string.
int strcmp (char * a, char * b )
        --- compare.  VALUE is negative, zero, positive!

        mnemomic: returns a-b.  That is, negative if
        a before b, zero if equal, positive if a after b.

int strncmp ( char * a, char * b, char * len)
        --- limited compare
```

Here is a program which copies strings to an array and prints them in reverse order.

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

void decr ( char * str )
{
  int len = strlen ( str );
  if ( len > 0 && str[ len-1 ] == '\n')
    str [ len-1 ] = '\0';
}

        /*
         * copy_string allocates storage for a copy
         * of the string, uses snprintf() to copy
         * the string to that storage, and returns
         * the copy.
         */

char * copy_string ( char * x )
{
  int len = strlen ( x );
```

```
  char * copy;

  copy = (char *) calloc (1, len+1 );
  snprintf ( copy, len+1, "%s", x );
  return copy;
}

main()
{
  char * line[1000];
  int count,i;
  char buffer[200];

  count = 0;
  while ( count < 1000 && fgets ( buffer, 200, stdin ) != NULL )
  {
    decr ( buffer );
    line[count] = copy_string ( buffer );
    ++count;
  }

        /* print in reverse order */

  for ( i=count-1; i>=0; --i )
    printf ("%s\n", line[i] );
}
```

20.2 2-dimensional arrays

We have created 1-dimensional arrays with no difficulty. Two-dimensional arrays are a lot more tricky.

```
  double a[10][10];
```

is a typical 2-dimensional array definition. How can we use pointers to produce arrays of flexible sizes? First of all the type must describe an array of arrays of doubles. Translating 'array' into 'pointer' we see that the appropriate type is `double * *`

```
  double ** c;
```

Can we create `c` like this?

```
  c = ( double * * ) calloc ( m, n * sizeof ( double ) );
```

No. Suppose that $m = n = 10$ and c is allocated 100 doubles beginning at address 40, say. Then

```
  c = 40
  c[0] = 0
  c[0][0] = ???
```

What we must do is create 10 separate arrays of size 10, and make `c` into an array of 10 pointers, giving the start of each 1-dimensional array.

Here is one way to do it.

```
  c = ( double * * ) calloc ( 10, sizeof ( double * ) );
  c[0] = ( double * ) calloc ( 10, sizeof ( double * ) );
  c[1] = ( double * ) calloc ( 10, sizeof ( double * ) );
etcetera

  Then c is an array of 'rows,' and each row is an array
of 10 doubles.  So for 0 <= i,j < 10,
  c[i][j]
    is the j-th entry in the i-th row of c.
```

These ideas lead to a matrix creation function

```
double * * mat ( int m, int n )
{
  int i;
  double * * mt;

  mt = ( double * * ) calloc ( m, sizeof ( double * ) );
  for (i=0; i<m; ++i)
  {
    mt[i] = ( double * ) calloc (n, sizeof ( double ) );
  }
}
```

In practical terms, memory allocation is a bit expensive in time and in space. Here is another version of the same function, more efficient because it uses 2 `callocs()`, not $m + 1$.

```
double * * mat ( int m, int n )
{
  int i;
  double * * mt;
  double * pool;

  mt = ( double * * ) calloc ( m, sizeof ( double * ) );
  pool = ( double * ) calloc ( m*n, sizeof ( double ) );
  for (i=0; i<m; ++i)
  {
    mt[i] = & ( pool [ i * n ] );
  }
}
```

EXAMPLE

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

double ** create_matrix ( int m, int n )
{
  double ** row = (double **) calloc ( m, sizeof ( double * ) );
  int i;
  for ( i=0; i<m; ++i)
    row[i] = (double *) calloc ( n, sizeof (double) );

  return row;
}

void print_matrix ( char header[], int m, int n, double ** a )
{
  printf("\n");
  printf("%s\n", header);
  int i;
  for (i=0; i<m; ++i)
  {
    int j;
    for (j=0; j<n; ++j)
      printf(" %6g", a[i][j]);
    printf("\n");
  }
  printf("\n");
}


int main()
{
  int m,k,kk, n;
  scanf ( "%d %d", &m, &k );
  double ** mat1 = create_matrix ( m, k );

  int i;
  for (i=0; i<m; ++i)
  {
    int j;
    for (j=0; j<k; ++j)
     scanf ( "%lf", &( mat1[i][j] ) );
  }
```

```
  scanf ( "%d %d", &kk, &n);
  double **mat2 = create_matrix ( kk, n );

  for ( i = 0; i<kk; ++i)
  {
    int j;
    for ( j = 0; j<n; ++j )
      scanf("%lf", &(mat2[i][j]));
  }

  if ( k != kk )
    printf("incompatible matrices\n");
  else
  {
    double ** mat3 = create_matrix ( m, n );
    int i,j;
    for (i=0; i<m; ++i)
    for (j=0; j<n; ++j)
    {
      int mm;
      for ( mm=0; mm<k; ++mm )
        mat3[i][j] += mat1[i][mm]*mat2[mm][j];
    }

    print_matrix ("A", m, k, mat1 );
    print_matrix ("B", k, n, mat2 );
    print_matrix ("AxB", m, n, mat3 );
  }
  return 0;
}
```

Sample run

```
A
      2      7      1
      8      2      8
      1      8      2


B
      3      1      4
      1      5      9
      2      6      5
```

```
AxB
     15      43      76
     42      66      90
     15      53      86
```