Handling mixed character and numeric data.

Formal definition of polynomial for ease of programming.

{<sign><coeff><exponent>}*

<sign> ==> + | -

<coeff> ==> FLOAT

<exponent> ==> x | x^<degree> | []

<degree> ==> INTEGER

Use fscanf (fid, "%lf", coeff);

Use fscanf (fid, "%d", exponent); or fgetc(fid)

Use fgetc (fid) and ungetc (char, fid) to examine characters.

Probably best to copy input stream into an array, removing blanks as you go, and build a pointer to the array. One can then read from the array just like from a file.

Most sophisticated.

Review of Pointers and Addresses (see Lecture 6)

C has a simple memory model. Blocks of memory are organized as a sequence of bytes which can be manipulated individually or in contiguous groups.
Each byte of memory has an address.

An addresses is stored in a pointer.

Each datatype requires one or more bytes to store it. Typically a character requires one byte, an integer requires between 2 and 8 bytes, a double usually take 8 bytes, and so on. It follows that not every byte address is a legitimate address of a data object.

Consider how we exchange two values in memory: for the quantities a and b. We would simply write

```
float a, b;
float  t;

t = a;
a = b;
b = t;
```

However, if we want to do the same thing inside a procedure we would have to pass the addresses of a and b as actual parameters, as follows

```
swap( &a, &b );
```

We know that a pointer p that points to an float (decimal number) data object is declared as follows:

```
    float*  p;
```

Thus the formal parameters must be:

```
void swap ( float* p, float* q ) {
    float  tt;
        tt  =  *p;
        *p  =  *q;
        *q  =  tt;
}
```

To access the data object referenced by a pointer, use the dereferencing operation *, as in

```
    *p = *p + 1;          adds 1 to the contents of p, as does
    p[0] = p[0] + 1;
while
    *p = *(p + 1);        is the same as
    p[0] = p[1];
    *p = *(p++);          p is incr. by the size of the object
```

It is important to always remember that a pointer is an address of an object of <u>a certain type</u>. You must keep track of the type_size when you are manipulating addresses of objects, and when manipulating the objects themselves. Keep in mind what you want to do with the pointer.

- Do you want to **assign it** to another pointer, or to
    - **pass it** as a pointer argument?
- Do you want to dereference it (follow it) and work with the object that it points to?

Discuss the examples pointers-1.c and its associated output pointers-1.log, and pointers-2.c and its associated output pointers-2.log. See class handout and online notes.

Arrays and Pointers (continued)

C really does not have arrays. It has contiguous regions of identical objects with a base pointer. Array notation is simply a form of pointer shorthand.

Given the two declarations:

```
float  a[10];
float*  pa;
```

The following pairs of notation are equivalent

```
a        &a[0]        use the latter as argument
a[i]     *(a+i)
pa[i]    *(pa+i)
```

The only difference between arrays and pointers is that declaring **a** as an array means that the identifier **a** is read-only. That is, it cannot appear as an **Lvalue**.

```
pa  = &a[3]     is ok
a   = pa + 1    is not
```

Here we are incrementing pa by **one unit of size float**. This is an address, but if we store it in a, then we would be saying that &a[0] is now &a[1]. Even a computer would be confused!! Therefore not allowed, because of original array declaration: float a[10];

See King Chapter 17 or K&R P. 167 for discussion of **dynamic memory allocation**.

By these means that we can implement dynamically allocated arrays by providing a mechanism for obtaining a pointer to a block of new memory. This is accomplished with the routines:

```
#include <stdio.h>

void* malloc( size_t size );

void* calloc( size_t NumMembers, size_t size );

void free( void* p );
```

Look at the examples pointers-3.c and its associated output pointers-3.log