Review of Pointers and Addresses

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

An address is stored in a pointer.

- Each datatype requires one or more bytes to store it. Typically a character requires one byte, an integer requires 2 or 4 bytes, a double usually take 8 bytes, and so on. It follows that not every byte address is a legitimate memory address for the start of a data object.

Consider how we exchange two values in memory. For the quantities a and b, we would simply write

```
double a, b;
double 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 to a double data object is declared as follows:

```
double* p;
```

Thus the formal definition of the swap function must be:

```
void swap ( double* p, double* q ) {
   double tt;
      tt = *p;    // p is a pointer, *p is the value
      *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[0] = p[1];` is the same as
`*p = *(p + 1);`
p is incremented by the **size of the object**.

But what about

```
*p = *(p++);    // avoid such ambiguous constructs.
```

- It is important to remember that a pointer is an address of an object of a certain type. 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
- **pass it** as a pointer argument?
- dereference it (**follow it**) and work with the object that it points to?

Review 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. **C has contiguous regions of identical objects with a base pointer.** Array notation is simply a form of pointer shorthand.

Given the two declarations:

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

The following pairs of notation are equivalent

`&a[0]`	`a`	the former is preferred
`a[i]`	`*(a+i)`	the former is preferred
`*(pa+i)`	`pa[i]`	the former is preferred

- 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]   // OK to use pointer as L-value
```

but not

```
a  = pa + 1  // cannot use array name here
```

Here we are incrementing pa by **one unit of size double**. 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 the original array declaration was

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

See King Chapter 17 or K&R P. 167

Dynamic Memory Allocation.

By using the three routines below we can implement dynamically allocated arrays by providing a mechanism for obtaining a pointer to a block of new memory.

```
#include <stdlib.h>

void* malloc( size_t size );

void* calloc( size_t NumMembers, size_t size );

void* realloc( void* ptr, size_t size );

void free( void* ptr );
```

Notes:
Malloc allocates N-bytes of space (not initialized)
Calloc allocates N-units of space, each of the same specified size. Each unit is set to "zero"
Realloc, is like malloc, creates a new space of M-bytes and copies over the contents of the data pointed to by "ptr".

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

Handling mixed character and numeric data.

Formal definition of polynomial for ease of programming.

```
<polynomial> ==> {<sign><coeff><exponent>}*

<sign> ==> + | -

<coeff> ==> DOUBLE

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

<degree> ==> INTEGER
```

Draw a state-transition diagram that describes the "processing" of a polynomial.

Note: ax^2 + bx + c = c + bx + ax^2 = c + x(b + x(a + ...)))
So a polynomial can be evaluated without the use of pow().

Typically want to read in a data stream, remove blanks or whitespace, store into memory (e.g. an array) and then use fgetc(), fscanf() to re-read the data now in a more regular form, but from an array instead of from a file.

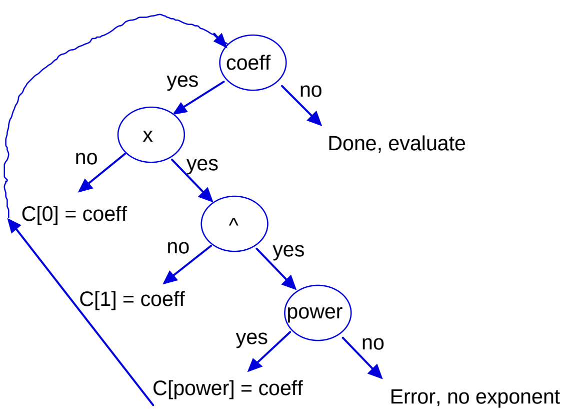
Use `fscanf ( fid, "%lf", coeff );`

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

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

Flow Diagram:

Recognition of polynomial terms



polynomial = sum (C[i] x^i)
i = 0 to N

int i; double C[100];

Example:

```
char  str  [100];                    // char* str = malloc(100);
int ch, i = 0;
while ((ch = getchar()) != NL)       // or perhaps EOF
    if (ch != BLANK)
        str[i++] = ch;               // or *(str+i++) = ch;
str[i] = EOS;                        // EOS is '\0'
```

we can now re-read str[ ] as if it were a file. In the case of polynomial evaluation one might do:

```
int rcode, power;
char x = 'x';
double coeff;

    rcode = sscanf (str, "%lf%c^%d", &coeff, &x, &power);
    // seeking a term

    switch (rcode) {
        case 1:     power = 0; break;
        case 2:     power = 1; break;
        case 3:     break;
        default:    exit (-printf("End of polynomial.\n"));
    }

    printf ("%c^%d has coefficient %f\n", x, power, coeff);
    return 0;
}
```