Pointers! They aren’t that hard!

Goals
1. Learn what a pointer is;
2. Solidify the connection between pointers and array types;
3. Learn how to declare, initialize and reassign pointers;
4. Learn how to “dereference” a pointer to obtain values;

Goals (continued)
5. Learn how to do pointer “arithmetic”;
6. Learn how to allocate and release dynamic memory using pointers;
Pointer Variables
As we have seen from the text, a pointer is an address of some memory location. A pointer variable, therefore, is a variable, whose content is an address. Note the distinction here that the content is not a value of some standard type or class as with a standard variable. Instead, it is the address of a location whose value is some standard type or class. As a variable, the address can change, unless it has been declared as a constant pointer (address).

Pointers and types
When a pointer variable is declared, it is assigned a type or class, and that is the type (or class object) to which it must point. (This doesn’t represent the whole truth, but it suffices for what we will be doing with pointers.)

Defining a pointer variable
Pointer variables, like regular variables, must be declared. This is done with type *. Following are some pointer declarations:
• int * Aptr; //Aptr is a pointer to an integer

(continued on next slide)
Defining a pointer variable

- `double * Bptr; // Bptr is a pointer to a double.
- `char * chptr; // chptr is a pointer to a character.
- `Fraction * Fptr; // Fptr is a pointer to a Fraction, assuming the Fraction class has been defined.

Assigning an address to a pointer.

Once a pointer variable has been declared, it can be assigned addresses. Even though an address is an integer value designating a memory location, the only integer you can assign to a pointer is 0. Any other value must be an address.

```
int B = 5;
int * Bptr = &B; // read the & as "address of".

Now the value of B is 5 and the value of Bptr is the address of where the 5 is stored; i.e., the memory location assigned to B.

Note:
int * Bptr = B; would not be allowed as this is attempting to assign an integer value to the pointer and not an address.
```
int A[ ] = {1,2,3,4,5};
int *Aptr = A;
In this case we don’t put the & in front of A, because the value of A is already an address, namely the address of A[0]. We could have done it the following way, but it is not as neat as the method above.

int A[ ] = {1,2,3,4,5};
int *Aptr = &A[0];
A[0] is the value of the first location, so &A[0] is the address of that location. It is therefore the same as the value of A, which is a pointer to the first location.

Caution!
As we have seen, type *, is used to declare a pointer to whatever type is specified. But the statement
int * A, B, C, D;
doesn’t do what you might think. This declares A as a pointer to an integer; however, B, C, and D are integers, not pointers. So to declare all as pointers write:
In other words, a * is needed in front of each variable to make it a pointer.
Obtaining values from pointers

When declaring a variable the * makes it a pointer. After that, placing the * in front of a pointer "dereferences" the pointer, and gives the value of what it is pointing to. What is the output of the following?

```cpp
int a = 200;
int *aPtr = &a;
cout << a << endl;
cout << *aPtr << endl;
```

Expand previous example

```cpp
int a = 200;
int *aPtr = &a;
cout << a << endl;
cout << *aPtr << endl;
*aPtr = 300;
cout << a << endl;
cout << *aPtr << endl;
```

Swapping using pointers

```cpp
void swap(int * A, int * B){
    int temp = *A;
    *A = *B;
    *B = temp;
    return;
}
```

Note where dereferencing is used and where it isn't used.
Using the swap function

Let the following statements be in main( ) or some driver function.

```cpp
int m = 100;
int n = 200;
cout << m << endl << n << endl;
swap(&m, &n); //Why have the & ?
cout << m << endl << n << endl;
```

Make the distinction

Suppose p1 and p2 are pointers to some type. What is the difference in the following two statements?

```cpp
p1 = p2;
*p1 = * p2;
```

We will draw some pictures to illustrate this.

Dynamically allocated arrays and new

Up to now, our arrays were declared in the program at compile time. Once done, the array size remains constant throughout the program, possibly wasting space if the whole array is not filled during execution of the program. The next few slides will indicate how you can compute a size during execution and then allocate an array of that size “on the fly”.
new (cont.)

Suppose at some point in a program, the integer variable N has been assigned a positive value. What happens when the following is executed?

```c
int * intPtr;
intPtr = new int[N];
```

The new command will cause an array of integers of size N to be allocated in memory, and it will return the address of the first location if the allocation is successful.

Accessing dynamic arrays

Once the array is allocated and the address stored in intPtr, the pointer variable intPtr can be used to access the array elements just as we have done with regular arrays:

Example:

```c
for(int i=0; i<N; i++)
    intPtr[i]=i;
```

What is this for loop doing? Are you beginning to see similarities in array names and pointers?

Deleting dynamic arrays

For reasons we will discuss, dynamic arrays should be returned to the memory pool when they are no longer needed. This is done in the following way:

```c
delete [] intPtr;
```

Note: This does not destroy the pointer variable, intPtr, rather it destroys or returns to the memory pool, the memory pointed to by intPtr. The pointer intPtr can now be used for other tasks, such as dynamically allocating another array if desired.
In the blanks show what is printed.

int *p = A;
int m;
m = *p +8;
cout << m;
m = *(p+5);
cout << m;
m=*A;
cout << m;
m=A[6+A[2]];
cout << m;

Problem

Suppose a data file named “Sample.dat” contains information as follows. The first record is an integer telling how many records follow. The remaining records each contain one double number. Write the code that will open the file; read the first record; dynamically allocate an array of that length; and read the remaining values into that array.