

**School of Electronic
and Communications
Engineering**

Arrays

arrays

- So far, we have been using only scalar variables
 - “scalar” meaning a variable with a single value
- But many things require a set of related values
 - coordinates or vectors require 3 (or 2, or 4, or more) values
 - text requires a long list of characters in a specified order
 - spreadsheet data is often a long list of numbers

what is an array?

- The spec says: “...a continuously allocated nonempty set of objects with a particular member object type”
- In real language: a bunch of things of the same type
 - each value is like a little variable
 - but all you really need to know is
 - where the array starts (a pointer, usually)
 - what type of data is in the array
 - all elements of an array are always the same data type
 - how many elements are in the array

scores example

- So let's keep things simple: imagine there are only 5 students in a class
 - they get the following scores on an exam: 57, 85, 97, 16, 82
 - this is what I'd like my program to do:

```
Scores: a program to compute grade statistics.  
Enter all scores, finish with -1:
```

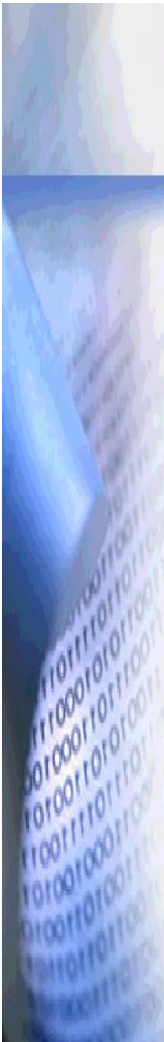
```
57 85 97 16 82 -1
```

```
Average: 61.4
```

```
Median: 82
```

```
Standard Deviation: 32.2
```

how can we do that?

- 
- A vertical decorative bar on the left side of the slide, featuring a blue and white abstract pattern that resembles binary code or a stylized 'C' shape.
- Since we don't know how many scores will be entered, we can't know how many variables to create
 - this will actually be a problem anyway, so we'll have to build in a maximum class size
 - let's say there will never be more than 250 students in a class
 - but still, the program will be awkward at best, since we have to read in an arbitrary number of values, and then add them up

so look at this example

```
#include <stdio.h>

int InputScores (double *);
void PrintStats (double *, int);

int main (void)
{
    double scores[250];
    int numScores;

    printf ("Scores: a program to compute grade statistics.\n");
    printf ("Enter all scores, finish with -1:\n\n");

    numScores = InputScores (scores);
    PrintStats (scores, numScores);
}
```

array declaration

```
#include <stdio.h>

int InputScores (double *);
void PrintStats (double *, int);

int main (void)
{
    double scores[250];
    int numScores;

    printf ("Scores: a program to compute grade statistics.\n");
```

- This declares a variable named “scores”
 - it is an array of 250 doubles
 - thus, it reserves enough memory for 250 contiguous doubles (1000 bytes)

array declarations

- As with other declarations, array variable declarations include:
 - a data type
 - a variable name
 - ends with a semicolon
 - But in addition, an array declaration has:
 - a length, enclosed in square brackets

type name[length] ;

 - we say that *name* has the type “array *length* of *type*”
- so in our previous example, scores is “array 250 of double”

initialization

- As with other declarations, array declarations can include an optional initialization
 - scalar variables are initialized with a single value
 - arrays are initialized with a list of values
- the list is enclosed in curly braces

```
int array [8] = {2, 4, 6, 8, 10, 12, 14, 16};
```

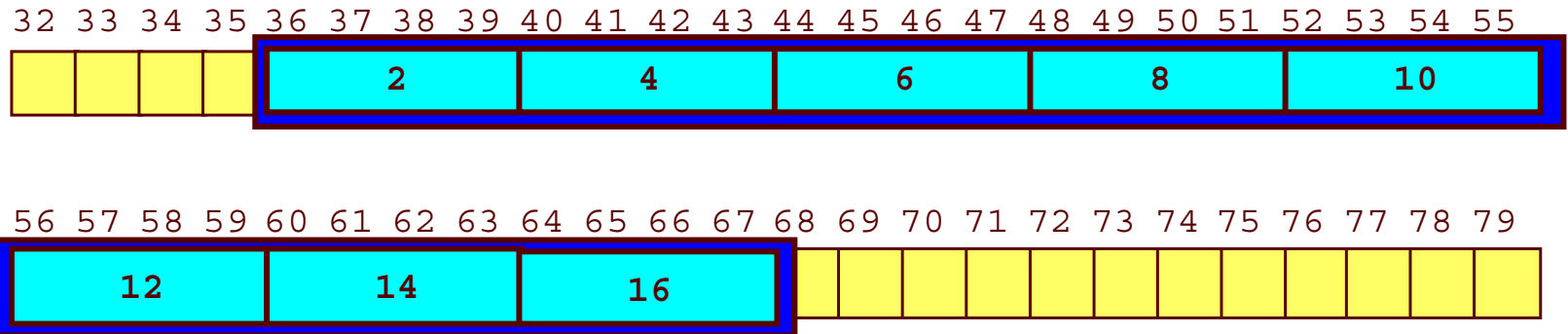
more about initialization

- The number of initializers cannot be more than the number of elements in the array
 - but it can be less
 - in which case, the remaining elements are initialized to 0
- If you like, the array size can be inferred from the number of initializers
 - by leaving the square brackets empty
 - so these are identical declarations

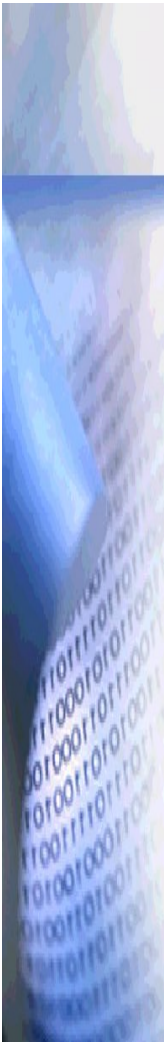
```
int array1 [8] = {2, 4, 6, 8, 10, 12, 14, 16};  
int array2 [] = {2, 4, 6, 8, 10, 12, 14, 16};
```

but what does this do?

```
int array [8] = {2, 4, 6, 8, 10, 12, 14, 16};
```



what can we do with arrays?

- 
- A vertical decorative bar on the left side of the slide, featuring a blue and white abstract pattern with binary code (0s and 1s) visible at the bottom.
- All well and good, but how do we use arrays?
 - There are two main things you can do with an array:
 - you can refer to an element in the array
 - you can get the address of the first element in the array

array elements

- When we refer to an array element, we use a subscript expression:
 - the operator is the square bracket pair []
 - the operands are the array variable, and the index expression

```
int array [8] = {2, 4, 6, 8, 10, 12, 14, 16};  
int i;  
  
for (i = 0; i < 8; ++i)  
    printf ("%d\n", array[i]);
```

subscript expression

- Some details about the subscript expression
 - for an array N of T, the data type of the subscript expression is T
- that is, the data type of **array[i]** is **int**
 - the expression inside the square brackets must have integral type
- that is, it can be char, short, int, or long
- but not double, float, long double, a pointer, an array, ...

address of first element

- When we use the array name in any value context, it is converted into a pointer to the first element of the array
 - what's a value context?
 - any use where the expression simply retrieves the value, and doesn't try to modify it
 - the type of the pointer is “pointer to T”
 - when the array type is “array N of T”

The Rule

In a value context, the name of an array of type “array N of T” becomes a “pointer to T”

- Remembering this rule will make arrays and pointers seem much more sensible
- Forgetting this rule will make them confusing

example

```
#include <stdio.h>

int main (void)
{
    int array [8] = {2, 4, 6, 8, 10, 12, 14, 16};
    int *p1, *p2;

    p1 = array;
    p2 = &array[0];

    printf ("%d  %d\n", *p1, *p2);

    return 0;
}
```

array recap: array declarations

- We have to declare an array variable just like any other variable

type name[size] ;

type name[size] = { initializer list } ;

type name[] = { initializer list } ;

- common elements:
 - type: the data type of each element of the array
 - name: the name of the variable we are declaring
 - size: either explicit or implicit, it's the number of elements

array recap: array definitions

- As with other declarations, a defining declaration will reserve storage
 - this will allocate an adjacent block of memory large enough to store all elements of the array
 - all of the declarations we've seen (and probably will see) have been definitions

array declaration examples

- Some examples of array declarations

double coordinate[3];

- this defines an array named coordinate
 - each element is a double
 - it reserves space for 3 elements

int frequency [26];

- this defines an array named frequency
 - each element is an int
 - it reserves space for 26 elements

int *pointer [6];

- this defines an array named pointer
 - each element is a pointer to an int
 - it reserves space for 6 elements

Multi-dimensional arrays

- Multi-dimensional arrays can be also implemented in C, although their use is rare.
- Higher order arrays are treated as arrays of array objects
- In this manner

```
char calendar [12] [31]
```

can be considered as 12 (month) objects of 31 days holding char objects or

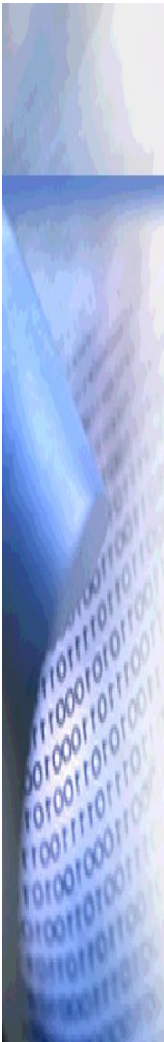
```
(calendar [12]) [31]
```

Multi-dimensional arrays initialization

Exercises

1. `char x [3] [2] = {1,2},{3,4},{5,6}`
2. `char y [2] [3] = {1,2,3},{4,5,6}`
3. `char z [3] [2] = {1,2},{3,4}`
4. `char a [2] [3] = {1,2},{3,4}`
5. `char b [3] [3] = {1,2,3},{4,5,6}`

array recap: using arrays

- 
- A vertical decorative graphic on the left side of the slide, showing a blue and white abstract pattern that resembles a close-up of a computer screen or a digital interface.
- We can only do two things with arrays:
 - refer to an element of the array
 - we can then do lots of different things with that element
 - get a pointer to the first element of the array

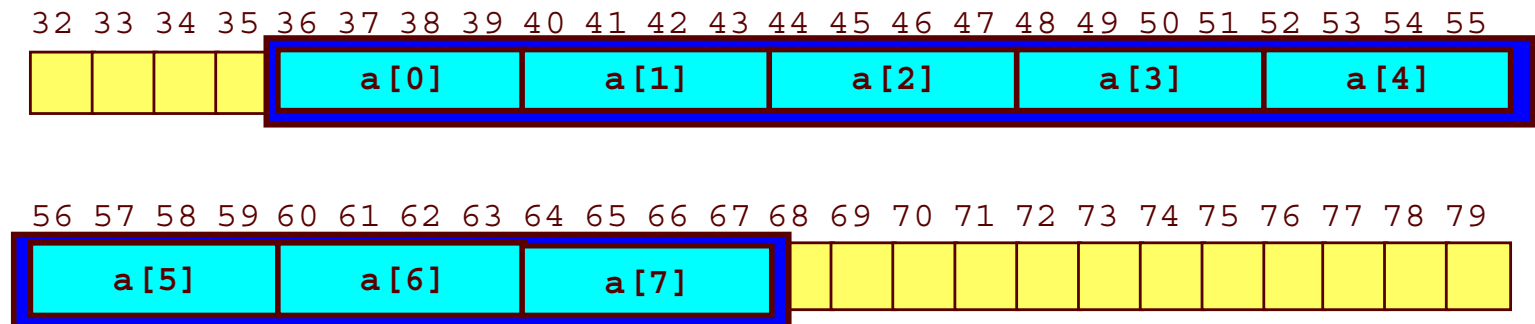
array recap: subscript expression

- We refer to an array element using a subscript expression
 - `array_name[index]`
 - details
- the index is any expression with integral value
- the index value selects a specific element of the array
 - **the first element is index 0 (!!!)**
- a subscript expression is usually an lvalue
 - which means we can assign a value to an array element

array recap: subscript expression

- Given the following declaration:

```
int a[8];
```



- it's not illegal to use an index off the end of the array
 - but it may have undefined results, since it will mean you're referring to memory that isn't part of the array
 - we'll see why this is legal a little later...

subscript expression examples

- So remember our example declarations:

```
double coor[3];
```

```
int frequency [26];
```

```
int *pointer [6];
```

- the following code is legal:

```
printf ("%f, %f, %f)\n", coor[0], coor[1],  
coor[2]);
```

```
for (i=0; i<26; ++i)
```

```
    frequency[i] = 0;
```

```
pointer[3] = &frequency[12];
```

array recap: array name as pointer

- In many contexts, using the array name by itself gets you a pointer to the first element

```
int frequency [26];
```

```
int *pointer [6];
```

```
pointer[3] = &frequency[12];
```

```
pointer[2] = frequency;
```

back to our example

```
#include <stdio.h>

int InputScores (double *);
void PrintStats (double *, int);

int main (void)
{
    double scores[250];
    int numScores;

    printf ("Scores: a program to compute grade statistics.\n");
    printf ("Enter all scores, finish with -1:\n\n");

    numScores = InputScores (scores);
    PrintStats (scores, numScores);
}
```

Note that we call the two functions with the argument **scores**

passing arrays

- It is perfectly legal to put an array name into a list of arguments to a function
 - this is a value context
 - therefore, the array name expression becomes a pointer to the first element of the array
 - note that the prototype for InputScores looks like this:

```
int InputScores (double *);
```

- So when we pass an array into a function, we're really just passing a pointer to the beginning of the array

passing arrays

- Note the “difference” between passing a scalar variable and an array
 - up to now, when we put a variable name in an argument list, its value was passed to the function
 - with an array, what gets passed is a pointer
- There’s no discrepancy here, really...
 - remember The Array Rule:
 - In a value context, the name of an array of type “array N of T” becomes a “pointer to T”
 - so in a manner of speaking, the “value” of the variable is the pointer to the first array element.

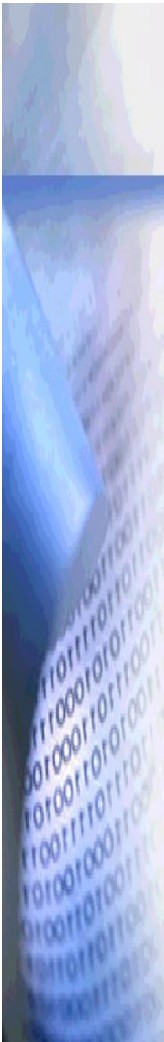
functions with array arguments

- We've talked about passing arrays into functions, but what does it look like from the function side?
- The function gets a pointer as an argument.
 - but we only know how to dereference a pointer to a single value
 - how does the function access the whole array?

arrays and pointers

- Here's where we start getting into the real overlap between arrays and pointers
 - remember The Rule:
 - **In a value context, the name of an array of type “array N of T” becomes a “pointer to T”**
 - and remember the syntax of the subscript expression
array_name [index]
 - in fact, there's nothing special about this use of the array name
 - it's simply a value context too
 - A subscript expression is really a pointer expression, not an array expression
 - and to understand this, we have to delve into pointer arithmetic

pointers again

- 
- A vertical decorative bar on the left side of the slide, featuring a blue and white abstract pattern with binary code (0s and 1s) visible at the bottom.
- So far, we have discussed three operators that deal with pointers:
 - the “address-of” operator: &
 - the dereference operator: *
 - the assignment operator
 - There are other pointer operators you can use, including:
 - addition: +
 - subtraction: -

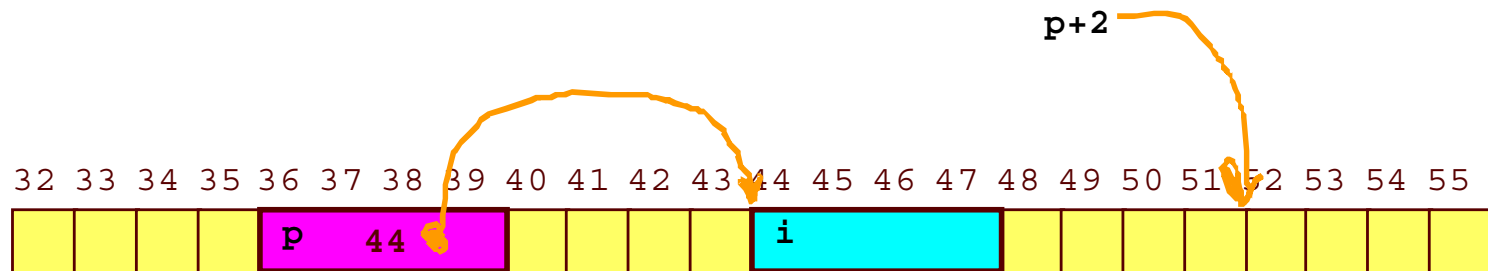
pointer arithmetic

- The familiar arithmetic operators $+$ and $-$ can also be used with pointers
 - but there are rules...
 - The computer addresses are just numbers, but pointer arithmetic isn't quite as simple as that...

pointer addition

- One of the operands of $+$ can be a pointer
 - but then the other operand must be an integer
 - and it doesn't just add the integer to the address
- instead, it adds the integer times the size of the thing the pointer points to
 - so given the declarations:

```
int i;
int *p = &i;
```



the logic of pointer addition

- Why do things work this way?
 - remember, things are different sizes on different computers
 - so we can't build into our programs an assumption that, say, ints are always 4 bytes long
 - plus, there's array indexing...
 - you can step through an array by simply incrementing a pointer
 - adding 1 to a pointer gets you to the next “thing” in the array
 - whether it's actually pointing to an array or not

pointer addition and array indexing

- So given an array declared like this:
 - `int a[8];`
 - remember that using the array name by itself just gets a pointer to the first element
 - that means:
 - `a` is equivalent to `&a[0]`
 - `*a` is equivalent to `a[0]`
 - now if we mix in pointer addition:
 - `a+1` is equivalent to `&a[1]`
 - `*(a+1)` is equivalent to `a[1]`

pointer addition and array indexing

- In fact, the C language defines the subscript expression in terms of pointer addition
 - that is, the expression $a[b]$ is identical to $*(a+b)$
- So back to the original reason we brought up pointer addition:
 - you can use an array name or a pointer in a subscript expression, because to C it's the same thing
 - it all becomes pointer arithmetic

some more details of pointer addition

- If one operand of $+$ is a pointer, the other operand must be an integer type
 - you cannot add two pointers
 - the data type of the expression is the same type as the pointer
- It doesn't matter what order you add the operands
 - $p+i$ is the same as $i+p$

pointer subtraction

- Subtraction with pointers follows the same logic as addition, but the rules are necessarily different
 - instead of moving forward in the array, as addition does, we're moving backwards

- so given the declarations:

```
int a[8];
```

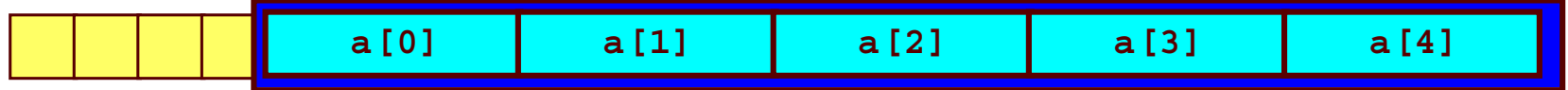
```
int *p = a+4;
```

- **p** now points to element **a[4]**, the fifth element of **a**
 - the expression **p-2** evaluates to a pointer to **a[2]**, the third element of **a**.

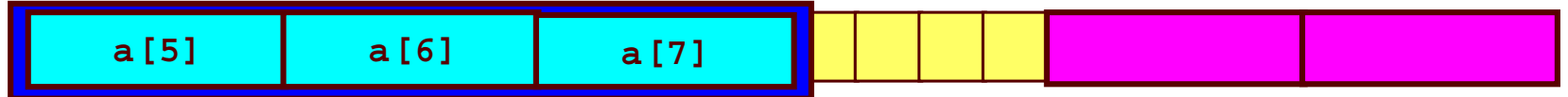
example

```
int a[8];
int *p, *q;
p = a+4;
q = p-2;
```

32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55

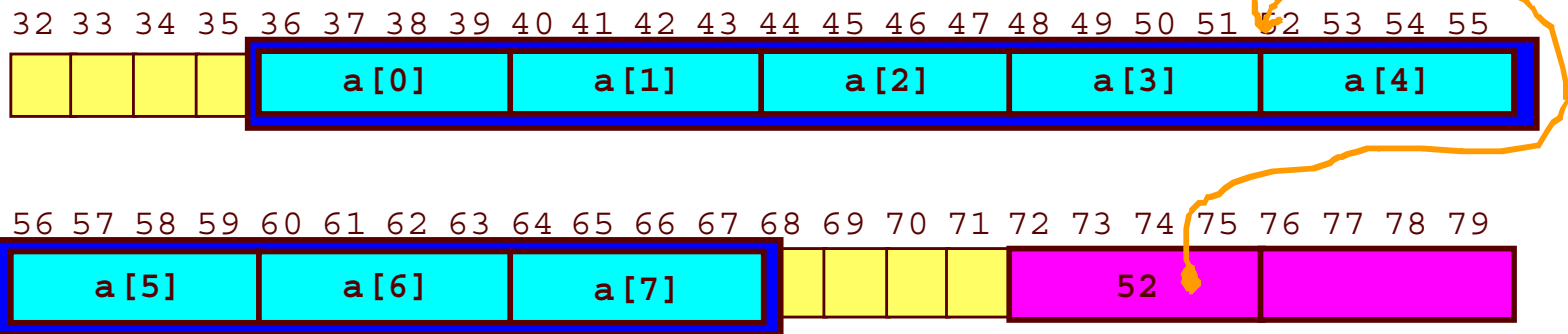


56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79



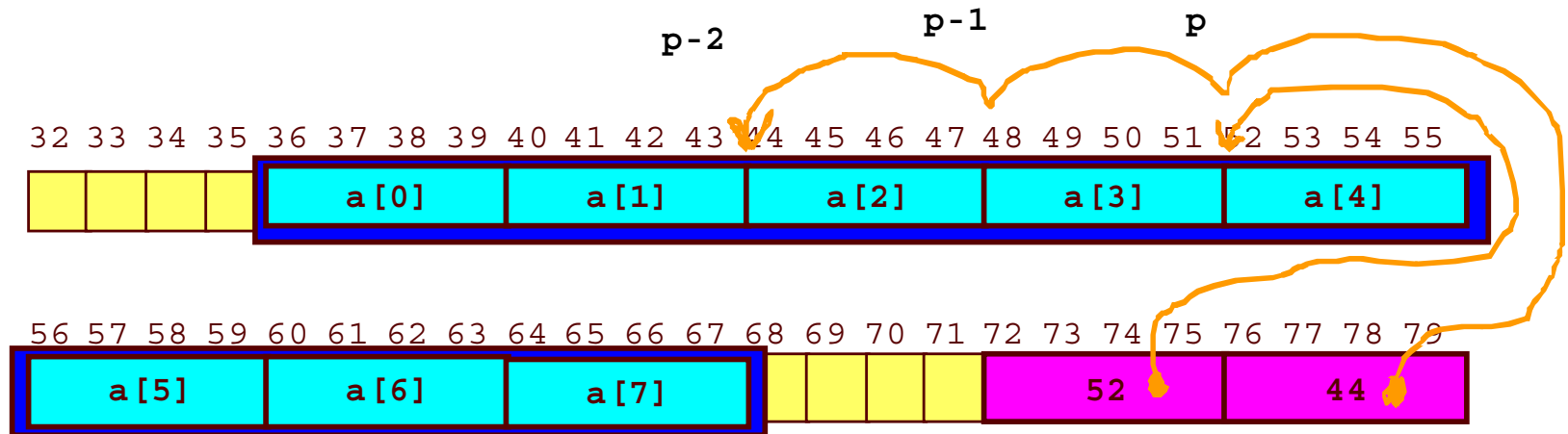
example

```
int a[8];
int *p, *q;
p = a+4;
q = p-2;
```



example

```
int a[8];
int *p, *q;
p = a+4;
q = p-2;
```



pointer subtraction

- So just like with addition, the `-` operator can take a pointer and an integer as operands
 - but you can only subtract an integer from a pointer, not vice versa
- You can go 2 lockers to the left of locker #583
- You can't go locker #583 to the left of 2
- But unlike addition, you can subtract one pointer from another
 - as long as the two pointers point to exactly the same type

pointer-pointer

- You can think of this algebraically:
 - given the following declarations:

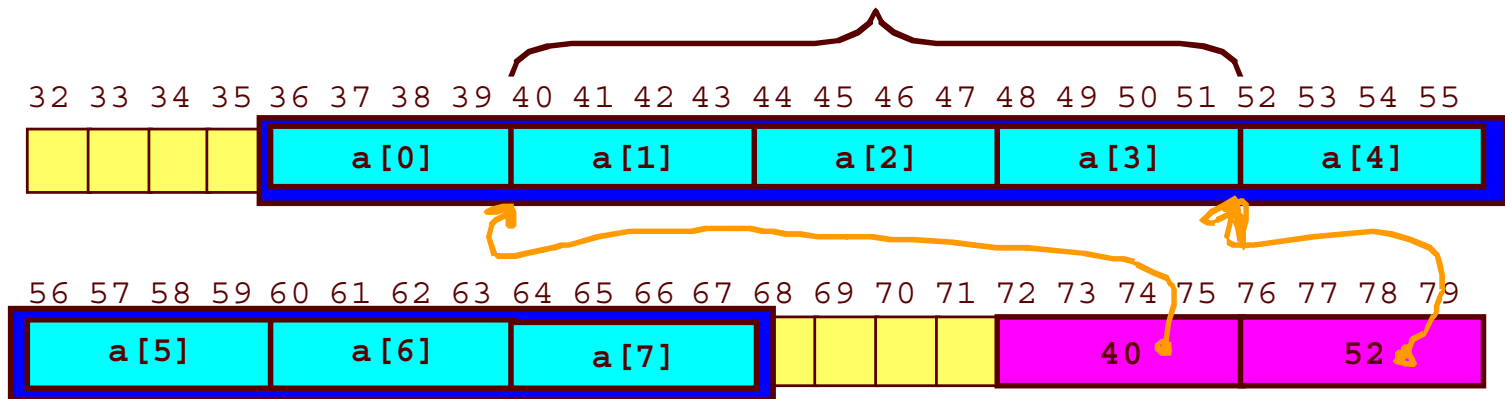
```
int a[8];  
int *p1 = a+2;  
int *p2;  
    if  
p2 = p1 + 3  
    then  
p1 == p2 - 3  
    and  
p2 - p1 == 3
```

so pointer subtraction...

- ... gives us the number of “things” between the two pointers.

```
int a[8];
int *p1 = &a[1];
int *p2 = &a[4];
```

p1-p2 is 3



so back to our example...

```
#include <stdio.h>

int InputScores (double *);
void PrintStats (double *, int);

int main (void)
{
    double scores[250];
    int numScores;

    printf ("Scores: a program to compute grade statistics.\n");
    printf ("Enter all scores, finish with -1:\n\n");

    numScores = InputScores (scores);
    PrintStats (scores, numScores);
}
```

- Now we need to write the two functions
 - InputScores
 - PrintStats

what's this supposed to do again?

- So let's keep things simple: imagine there are only 5 students in the class
 - they get the following scores on an exam: 57, 85, 97, 16, 82
 - this is what I'd like my program to do:

```
Scores: a program to compute grade statistics.  
Enter all scores, finish with -1:
```

```
57 85 97 16 82 -1
```

```
Average: 61.4
```

```
Median: 82
```

```
Standard Deviation: 32.2
```

PrintStats()

- So the PrintStats function should:
 - take the array and the number of scores as arguments
 - remember that our array can hold up to 250 scores
 - but that doesn't mean that all 250 were entered
 - it should compute the average, median, and standard deviation, and print them out
 - the function declaration is this:

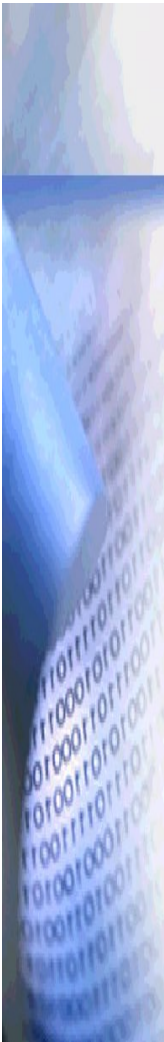
```
void PrintStats (double *, int);
```

so let's write the function

- First of all, we need to give names to our parameters
 - the declaration doesn't need names; all it needs are the data types
 - but the actual function definition is declaring local variables to use within the function; these variables need names
- We also can put in the braces that will surround the code for the function

```
void PrintStats (double *scores, int numScores){  
    ...  
}
```

so what's the function going to do?

- 
- A vertical decorative bar on the left side of the slide, featuring a blue and white abstract pattern with binary code (0s and 1s) visible at the bottom.
- It must:
 - compute the average
 - compute the median
 - compute the standard deviation
 - print the statistics
 - Let's punt on the math for now...
 - after all, this is a computer class, not a statistics class
 - we'll just use three more functions, which we can write later

our function so far

```
double ComputeAverage (double *, int);
double ComputeMedian (double *, int);
double ComputeStandardDev (double *, int);

void PrintStats (double *scores, int num)
{
    double average;
    double median;
    double stdDev;

    average = ComputeAverage (scores, num);
    median = ComputeMedian (scores, num);
    stdDev = ComputeStandardDev (scores, num);

    printf ("Average: %5.1f\n", average);
    printf ("Median: %5.1f\n", median);
    printf ("Standard deviation: %5.1f\n", stdDev);
}
```


how would we do those functions?

```
/*  
 * ComputeAverage: calculate and return the average  
 *   of the numbers in an array  
 */  
double ComputeAverage (double *array, int size)  
{  
    int i;  
    double sum = 0.0;  
  
    for (i = 0; i < size; ++i)  
        sum += array[i];  
    return sum/size;  
}
```

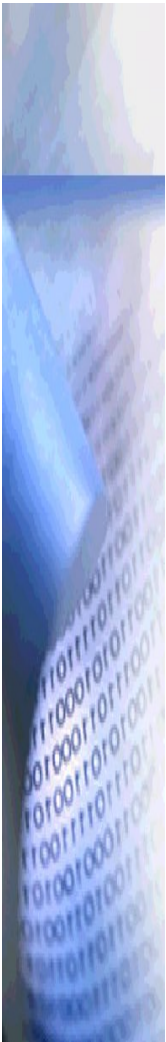
next...

- Now we have to implement InputScores

```
int InputScores (double *);
```

- What this will do:
 - keep reading numbers from the keyboard until the user enters a negative number
 - store all the entered numbers in the array
 - return the number of entries
- So as before, let's set things up for the code

```
int InputScores (double *scores){  
    ...  
}
```

- 
- A vertical decorative bar on the left side of the slide, featuring a blue and white abstract pattern that resembles binary code or a stylized 'C' shape.
- Now what?
 - we want to do the following:
 - read a number
 - if it's negative, return
 - otherwise, store it in the array
 - increment a running count of the number of entries
 - and go back to read another one
 - doesn't this sound a lot like a loop?
 - we're repeating the same steps over and over

choosing a loop

- We have 3 main kinds of loops at our disposal
 - for
 - while
 - do while
- How do we choose?
 - in reality, we could use any of the three

InputScores using while

- Here's one way to do it:

```
int InputScores (double *scores)
{
    int count = 0;
    double score;

    scanf ("%lf", &score);
    while (score >= 0.0) {
        scores[count] = score;
        count += 1;
        scanf ("%lf", &score);
    }

    return count;
}
```

InputScores using while

- Or slightly more compactly:

```
int InputScores (double *scores)
{
    int count = 0;
    double score;

    scanf ("%lf", &score);
    while (score >= 0.0) {
        scores[count++] = score;
        scanf ("%lf", &score);
    }

    return count;
}
```

another InputScores

```
int InputScores (double *scores)
{
    int count = 0;
    double score;

    scanf ("%lf", &score);
    while (score >= 0.0) {
        scores[count++] = score;
        scanf ("%lf", &score);
    }

    return count;
}
```