

Lecture 12 - Pointers and Arrays

Meng-Hsun Tsai
CSIE, NCKU

Introduction

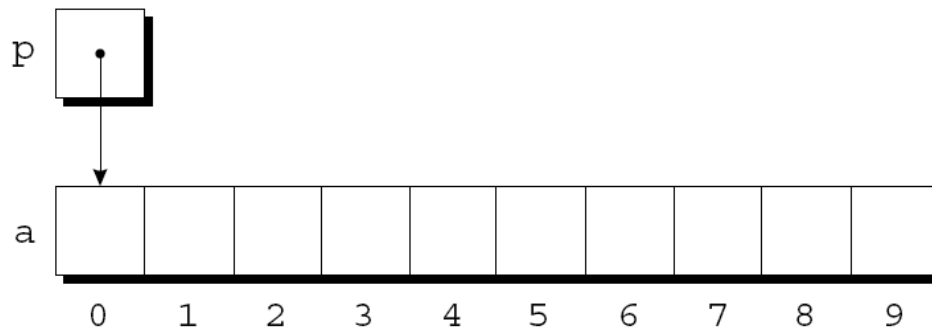
- C allows us to **perform arithmetic**—addition and subtraction—**on pointers to array elements**.
- This **leads to an alternative way of processing arrays** in which **pointers take the place of array subscripts**.
- The **relationship** between **pointers and arrays** in C is a close one.
- Understanding this relationship is critical for mastering C.

Pointer Arithmetic

- Lecture 11 showed that **pointers can point to array elements**:

```
int a[10], *p;  
p = &a[0];
```

- A graphical representation:

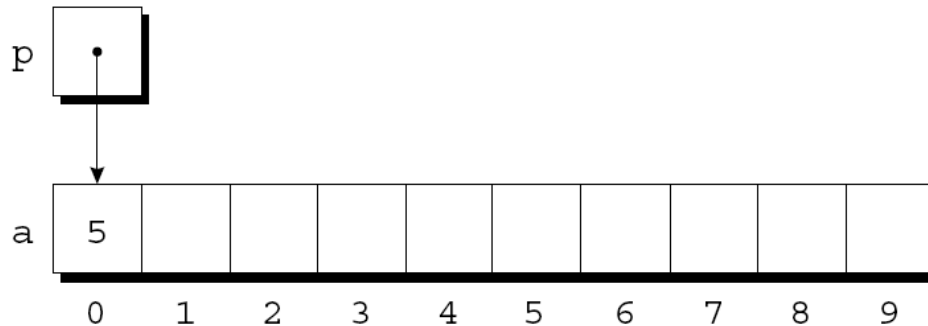


Pointer Arithmetic (cont.)

- We can now **access** `a[0]` **through** `p`; for example, we can **store the value 5** in `a[0]` by writing

```
*p = 5;
```

- An updated picture:



Pointer Arithmetic (cont.)

- If p points to an element of an array a , the **other elements** of a **can be accessed by performing *pointer arithmetic*** (or ***address arithmetic***) on p .
- C supports **three (and only three) forms** of pointer arithmetic:
 - **Adding an integer** to a pointer
 - **Subtracting an integer** from a pointer
 - **Subtracting one pointer from another**

Adding an Integer to a Pointer

- Adding an integer j to a pointer p yields a pointer to the element j places after the one that p points to.
- More precisely, if p points to the array element $a[i]$, then $p + j$ points to $a[i+j]$.

Adding an Integer to a Pointer (cont.)

- Assume that the following declarations are in effect:

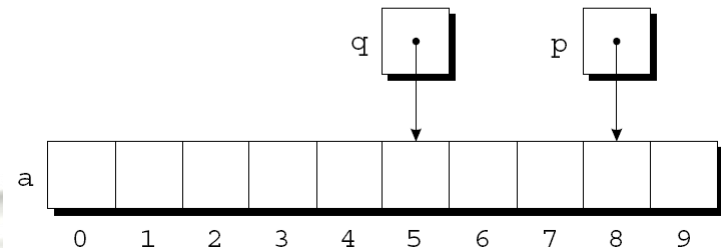
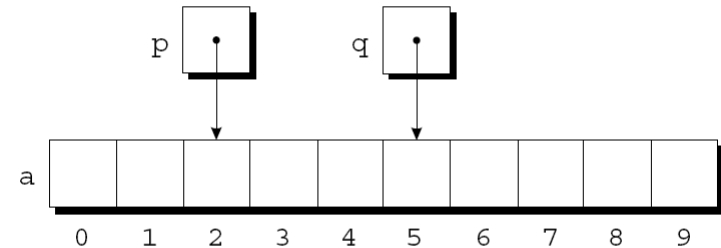
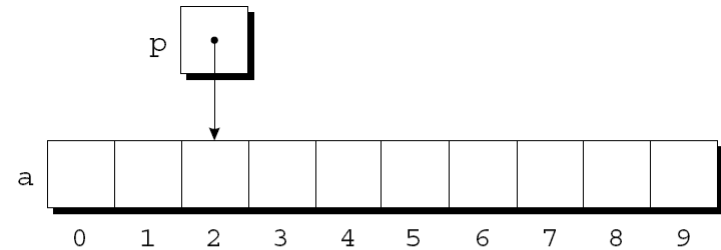
```
int a[10], *p, *q, i;
```

- Example of pointer addition:

```
p = &a[2];
```

```
q = p + 3;
```

```
p += 6;
```

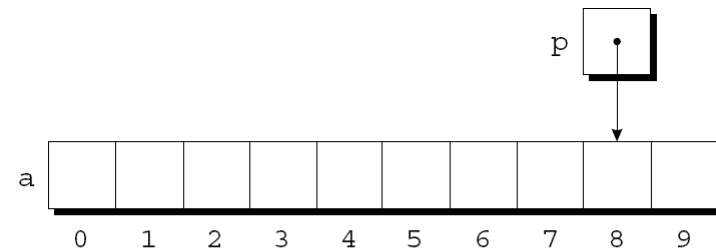


Subtracting an Integer from a Pointer

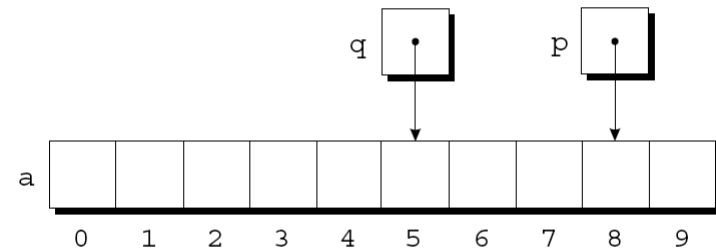
- If p points to $a[i]$, then $p - j$ points to $a[i-j]$.

- Example:

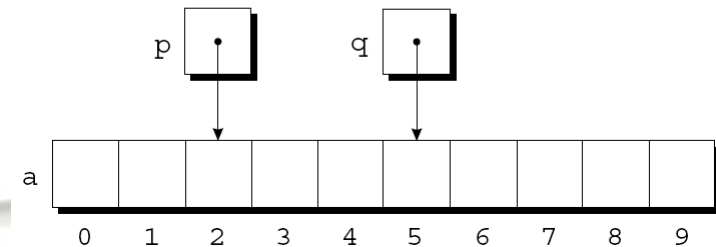
```
p = &a[8];
```



```
q = p - 3;
```



```
p -= 6;
```



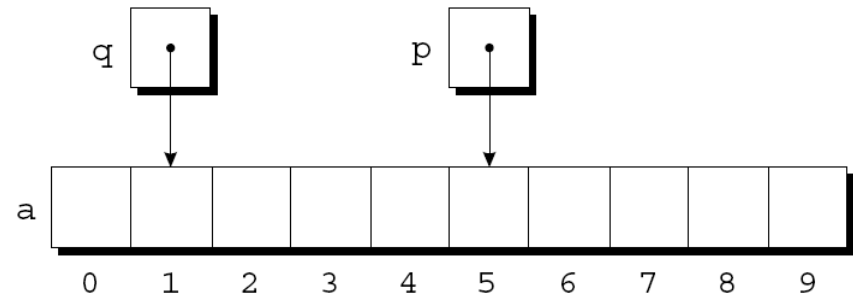
Subtracting One Pointer from Another

- When one pointer is subtracted from another, the result is the distance (measured in array elements) between the pointers.
- If p points to $a[i]$ and q points to $a[j]$, then $p - q$ is equal to $i - j$.

- Example:

```
p = &a[5];
```

```
q = &a[1];
```



```
i = p - q;    /* i is 4 */
```

```
i = q - p;    /* i is -4 */
```

Subtracting One Pointer from Another (cont.)

- Operations that **cause undefined behavior**:
 - **Performing arithmetic on a pointer that doesn't point to an array element**
 - **Subtracting pointers unless both point to elements of the same array**

Comparing Pointers

- Pointers **can be compared** using the **relational operators** ($<$, $<=$, $>$, $>=$) and the **equality operators** ($=$ and \neq).
- Using **relational operators is meaningful only** for pointers to elements of **the same array**.
- The **outcome of the comparison** depends on the **relative positions** of the two elements in the array.
- After the assignments

```
p = &a[5];  
q = &a[1];
```

the value of $p <= q$ is **0** and the value of $p >= q$ is **1**.

Using Pointers for Array Processing

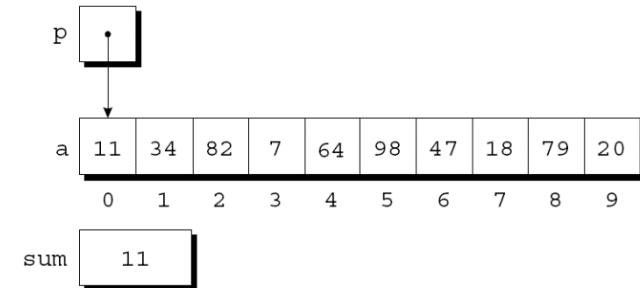
- **Pointer arithmetic allows** us to **visit the elements of an array** by **repeatedly incrementing a pointer variable**.
- A loop that sums the elements of an array `a`:

```
#define N 10  
...  
int a[N], sum, *p;  
...  
sum = 0;  
for (p = &a[0]; p < &a[N]; p++)  
    sum += *p;
```

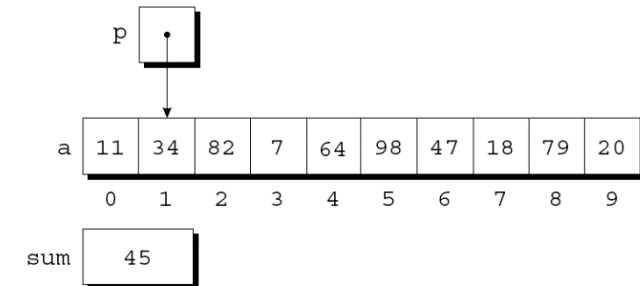
Using Pointers for Array Processing (cont.)

```
for (p = &a[0]; p < &a[N]; p++)  
    sum += *p;
```

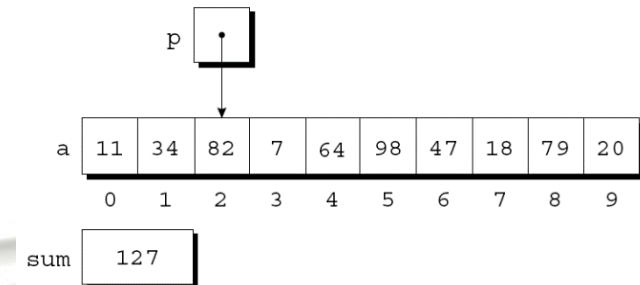
At the end of the first iteration:



At the end of the second iteration:



At the end of the third iteration:



Using Pointers for Array Processing (cont.)

- It's legal to apply the address operator to `a[N]`, even though this element doesn't exist.
- Pointer arithmetic may save execution time.

```
for (p = &a[0]; p < &a[N]; p++)  
    sum += *p;
```

Combining the * and ++ Operators

- C programmers often **combine** the * (**indirection**) and ++ **operators**.
- A statement that **modifies an array element** and **then advances to the next element**:

```
a[i++] = j;
```

- The corresponding pointer version:

```
*p++ = j;
```

- Because the postfix version of ++ takes precedence over *, the **compiler sees this as**

```
*(p++) = j;
```

Combining the * and ++ Operators (cont.)

- Possible combinations of * and ++:

<i>Expression</i>	<i>Meaning</i>
<code>*p++</code> or <code>* (p++)</code>	Value of expression is <code>*p</code> before increment; increment <code>p</code> later
<code>(*p) ++</code>	Value of expression is <code>*p</code> before increment; increment <code>*p</code> later
<code>*++p</code> or <code>* (++p)</code>	Increment <code>p</code> first; value of expression is <code>*p</code> after increment
<code>++*p</code> or <code>++ (*p)</code>	Increment <code>*p</code> first; value of expression is <code>*p</code> after increment

Combining the * and ++ Operators (cont.)

- The **most common** combination of * and ++ is `*p++`, which is **handy in loops**.

- Instead of writing

```
for (p = &a[0]; p < &a[N]; p++)  
    sum += *p;
```

to sum the elements of the array `a`, we could write

```
p = &a[0];  
while (p < &a[N])  
    sum += *p++;
```

Using an Array Name as a Pointer

- **Pointer arithmetic** is one way in which **arrays and pointers are related**.

- Another key relationship:

*The **name of an array** can be used as a pointer to the first element in the array.*

- This relationship **simplifies pointer arithmetic** and **makes both arrays and pointers more versatile**.

Using an Array Name as a Pointer (cont.)

- Suppose that `a` is declared as follows:

```
int a[10];
```

- Examples of **using `a` as a pointer**:

```
*a = 7;    /* stores 7 in a[0] */
```

```
*(a+1) = 12; /* stores 12 in a[1] */
```

- In general, `a + i` is **the same as** `&a[i]`.
 - Both represent **a pointer to element `i` of `a`**.
- Also, `*(a+i)` is **equivalent to** `a[i]`.
- Both represent **element `i` itself**.

Using an Array Name as a Pointer (cont.)

- The fact that an array name can serve as a pointer **makes it easier to write loops** that step through an array.

- Original loop:

```
for (p = &a[0]; p < &a[N]; p++)  
    sum += *p;
```

- Simplified version:

```
for (p = a; p < a + N; p++)  
    sum += *p;
```

Using an Array Name as a Pointer (cont.)

- Although an **array name** can be used as a pointer, it's **not possible to assign it a new value**.
- **Attempting to make it point elsewhere is an error:**

```
while (*a != 0)
    a++;                /* *** WRONG *** */
```

- This is no great loss; **we can always copy a into a pointer variable, then change the pointer variable:**

```
p = a;
while (*p != 0)
    p++;
```

Program: Reversing a Series of Numbers (Revisited)

- The `reverse.c` program of **Lecture 8** reads 10 numbers, then **writes the numbers in reverse order**.
- The original program stores the numbers in an array, **with subscripting** used to access elements of the array.
- `reverse3.c` is a new version of the program in which **subscripting has been replaced with pointer arithmetic**.

Program: Reversing a Series of Numbers (Revisited) (cont.)

reverse3.c

```
#include <stdio.h>
#define N 10

int main(void)
{
    int a[N], *p;

    printf("Enter %d numbers: ", N);
    for (p = a; p < a + N; p++)
        scanf("%d", p);

    printf("In reverse order:");
    for (p = a + N - 1; p >= a; p--)
        printf(" %d", *p);
    printf("\n");

    return 0;
}
```

```
for (i = 0; i < n; i++)
    scanf("%d", &a[i]);
```

```
for (i = n - 1; i >= 0; i--)
    printf(" %d", a[i]);
```

Array Arguments (Revisited)

- When passed to a function, an array name is treated as a pointer.

- Example:

```
int find_largest(int a[], int n)
{
    int i, max;

    max = a[0];
    for (i = 1; i < n; i++)
        if (a[i] > max)
            max = a[i];
    return max;
}
```

- A call of find_largest:

```
largest = find_largest(b, N);
```

This call causes a pointer to the first element of `b` to be assigned to `a`; the array itself isn't copied.

Array Arguments (Revisited) (cont.)

- The fact that an **array argument is treated as a pointer** has some important consequences.
- *Consequence 1:*
 - When an **ordinary variable** is passed to a function, its value is copied; **any changes to the corresponding parameter don't affect the variable.**
 - **In contrast, an array used as an argument isn't protected against change.**

Array Arguments (Revisited) (cont.)

- For example, the `store_zeros()` function **modifies an array** by storing zero into each of its elements.

```
void store_zeros(int a[], int n)
{
    int i;

    for (i = 0; i < n; i++)
        a[i] = 0;
}
```

- To indicate that an array parameter won't be changed, we can include the word `const` in its declaration:

```
int find_largest(const int a[], int n)
```

- If `const` is present, **the compiler will check that no assignment to an element of `a` appears** in the body of `find_largest`.

Array Arguments (Revisited) (cont.)

- *Consequence 2:*
 - The time required to pass an array to a function doesn't depend on the size of the array.
 - There's no penalty for passing a large array, since no copy of the array is made.

Array Arguments (Revisited) (cont.)

- *Consequence 3:*

- An **array parameter can be declared as a pointer** if desired.
- `find_largest` could be defined as follows:

```
int find_largest(int *a, int n)
```
- **Declaring `a` to be a pointer is equivalent to declaring it to be an array**; the compiler treats the declarations as though they were identical.

Array Arguments (Revisited) (cont.)

- Although declaring a *parameter* to be an array is the same as declaring it to be a pointer, **the same isn't true for a *variable*.**
- The following declaration causes the compiler to **set aside space for 10 integers:**

```
int a[10];
```

- The following declaration causes the compiler to **allocate space for a pointer variable:**

```
int *a;
```

Array Arguments (Revisited) (cont.)

- In the latter case, `a` is not an array; attempting to use it as an array can have disastrous results.
- For example, the assignment
`*a = 0; /** WRONG ***/`
will store 0 where `a` is pointing.

`int *a;`
- Since we don't know where `a` is pointing, the effect on the program is undefined.

Array Arguments (Revisited) (cont.)

- *Consequence 4:*

- A function with an array parameter **can be passed an array "slice"**—a sequence of consecutive elements.
- An example that applies `find_largest` to elements 5 through 14 of an array `b`:

```
largest = find_largest(&b[5], 10);
```

Using a Pointer as an Array Name

- **C allows us to subscript a pointer** as though it were an array name:

```
#define N 10
```

```
...
```

```
int a[N], i, sum = 0, *p = a;
```

```
...
```

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

```
    sum += p[i];
```

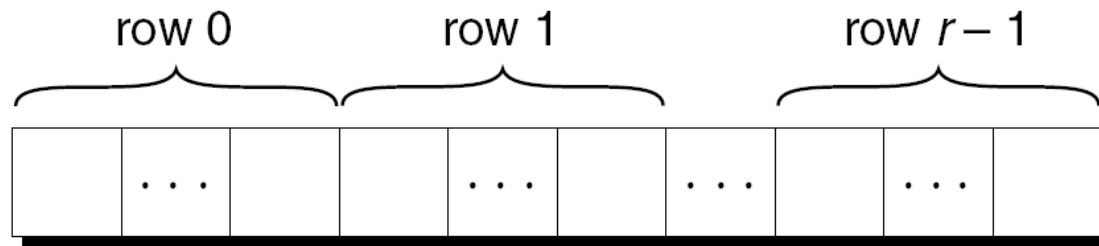
The compiler treats `p[i]` as `*(p+i)`.

Pointers and Multidimensional Arrays

- Just as **pointers can point to** elements of **one-dimensional arrays**, they can **also** point to elements of **multidimensional arrays**.
- This section explores common techniques for using pointers to process the elements of multidimensional arrays.

Processing the Elements of a Multidimensional Array

- Lecture 8 showed that **C stores two-dimensional arrays in row-major order**.
- Layout of an array with r rows:



- If p initially points to the element in **row 0, column 0**, we can visit every element in the array by **incrementing p repeatedly**.

Processing the Elements of a Multidimensional Array (cont.)

- Consider the problem of **initializing all elements** of the following array **to zero**:

```
int a[NUM_ROWS][NUM_COLS];
```

- The **obvious technique** would be to use **nested for loops**:

```
int row, col;  
for (row = 0; row < NUM_ROWS; row++)  
    for (col = 0; col < NUM_COLS; col++)  
        a[row][col] = 0;
```

Processing the Elements of a Multidimensional Array (cont.)

- If we view `a` as a one-dimensional array of integers, a single loop is sufficient:

```
int *p;  
for (p = &a[0][0]; p <= &a[NUM_ROWS-1][NUM_COLS-1]; p++)  
    *p = 0;
```

- Although treating a two-dimensional array as one-dimensional may seem like cheating, it works with most C compilers.
- Techniques like this one definitely hurt program readability, but—at least with some older compilers—produce a compensating increase in efficiency.
- With many modern compilers, though, there's often little or no speed advantage.

Processing the Rows of a Multidimensional Array (cont.)

- A pointer variable p can also be used for **processing** the elements in **just one row of a two-dimensional array**.
- To visit the elements of row i , we'd initialize p to point to element 0 in row i in the array a :

```
 $p = \&a[i][0];$ 
```

or we could simply write

```
 $p = a[i];$ 
```

Processing the Rows of a Multidimensional Array (cont.)

- For any two-dimensional array a , the expression $a[i]$ is a pointer to the first element in row i .
- To see why this works, recall that $a[i]$ is equivalent to $*(a + i)$.
- Thus, $\&a[i][0]$ is the same as $\&(* (a[i] + 0))$, which is equivalent to $\&*a[i]$.
- This is the same as $a[i]$, since the $\&$ and $*$ operators cancel.

Processing the Rows of a Multidimensional Array (cont.)

- A loop that **clears row i** of the array a :

```
int a[NUM_ROWS][NUM_COLS], *p, I;  
for (p = a[i]; p < a[i] + NUM_COLS; p++)  
    *p = 0;
```

- Since $a[i]$ is a pointer to row i of the array a , **we can pass $a[i]$ to a function that's expecting a one-dimensional array** as its argument.
- In other words, **a function that's designed to work with one-dimensional arrays will also work with a row belonging to a two-dimensional array.**

Processing the Rows of a Multidimensional Array (cont.)

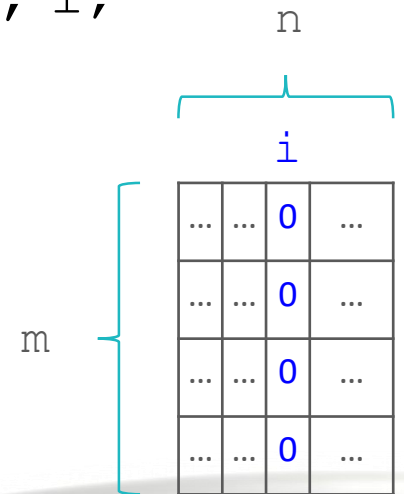
- Consider `find_largest`, which was originally designed to find the largest element of a one-dimensional array.
- We can just as easily use `find_largest` to determine the largest element in row `i` of the two-dimensional array `a`:

```
largest = find_largest(a[i], NUM_COLS);
```


Processing the Columns of a Multidimensional Array

- Processing the elements in a *column* of a two-dimensional array isn't as easy, because arrays are stored by row, not by column.
- A loop that **clears column i** of the array `a`:

```
int a[NUM_ROWS][NUM_COLS], (*p)[NUM_COLS], i;  
for (p = &a[0]; p < &a[NUM_ROWS]; p++)  
    (*p)[i] = 0;
```



Using the Name of a Multidimensional Array as a Pointer

- The **name of *any* array** can be used as a pointer, **regardless of how many dimensions it has, but some care is required.**
- Example:

```
int a[NUM_ROWS][NUM_COLS];
```

***a* is *not* a pointer to *a*[0][0]; instead, it's a pointer to *a*[0].**
- C regards ***a*** as a one-dimensional array whose elements are one-dimensional arrays.
- When used as a pointer, ***a*** has type `int (*)[NUM_COLS]` (pointer to an integer array of length `NUM_COLS`).

Using the Name of a Multidimensional Array as a Pointer (cont.)

- Knowing that `a` points to `a[0]` is useful for simplifying loops that process the elements of a two-dimensional array.

- Instead of writing

```
for (p = &a[0]; p < &a[NUM_ROWS]; p++)  
    (*p)[i] = 0;
```

to clear column `i` of the array `a`, we can write

```
for (p = a; p < a + NUM_ROWS; p++)  
    (*p)[i] = 0;
```

Using the Name of a Multidimensional Array as a Pointer (cont.)

- We can “trick” a function into thinking that a multidimensional array is really one-dimensional.
- A first attempt at using using `find_largest` to find the largest element in `a`:

```
largest = find_largest(a, NUM_ROWS * NUM_COLS); /* WRONG */
```

This an error, because the type of `a` is `int (*) [NUM_COLS]` but `find_largest` is expecting an argument of type `int *`.

- The correct call:

```
largest = find_largest(a[0], NUM_ROWS * NUM_COLS);
```

`a[0]` points to element 0 in row 0, and it has type `int *` (after conversion by the compiler).

A Quick Review to This Lecture

- Arithmetic on pointers can take place of array subscripts.
- Three supported pointer arithmetic:
 - Adding an integer to a pointer
 - Subtracting an integer from a pointer
 - Subtracting one pointer from another
- Pointers can be compared using $<$, $<=$, $>$, $>=$, $==$ and $!=$

```
int a[10], *p, *q, k;  
p = &a[0];  
q = &a[5];  
*p = 5;    //a[0]=5  
*p++ = 3;  //a[0]=3;  
*p-- = 4;  //a[1]=4;  
k = q - p; // k = 5;
```

A Quick Review to This Lecture (cont.)

- Name of an array can be used as **pointer to the first element.**, but it's **not possible to assign a new value.**
- Summing elements of an array

```
sum = 0;  
for(p = a; p < a + N; p++)  
    sum += *p;
```
- Array name as an argument is **treated as a pointer**

```
int fun(int a[], const int *b)  
{ *a = 3; }  
int main()  
{  
    int c[10];  
    fun(c, c);  
}
```

