APS105 Lecture 15
6.3, Array Parameters

Dan Zingaro

March 1, 2010

Feedback from Reading Quiz

Considerable confusion over the comparison of pointers (i.e. comparing pointers with < or >). More of this today ...

Addresses of Array Elements

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

- Let's say the address of `a[0]` is 1280
  - e.g. `&a[0] == 1280`
- We are guaranteed that elements of the same array are in order in memory
- So, the address of `a[1]` here is 1284, the address of `a[2]` is 1288, etc.

Comparing Pointers

- We can compare pointers using the relational operators (<, <=, ==, !=, >, >=), but only if they are pointers to elements in the same array
- Pointer `p` is < pointer `q` if `p` points to an array element that is to the left of the element pointed to by `q`
- Comparing pointers that are not pointing to elements of the same array is undefined
- We also have pointer arithmetic: if `p` points to an element of an array, `p+j` points `j` positions to the right, and `p-j` points `j` positions to the left

ConcepTest

```
int a[2] = {4, 2};
int *p = &a[1];
```

Which of the following is true after the code runs?

- A. `p < &a[0]`
- B. `p == &a[0]`
- C. `p > &a[0]`

Array Access: Pointer Syntax

- Using pointer arithmetic and comparison, we can access the elements of an array
- e.g. to sum the elements of an array:

```
int a[10], sum, *p;
...
sum = 0;
for (p = &a[0]; p <= &a[9]; p++)
  sum += *p;
```

Array Names

- So far, we have always followed an array name by an index
- If we use an array name a by itself, C interprets it as `&a[0]`
- So, the name of an array is the address of its first element
- We can use this to slightly simplify the array sum code

```
int a[10], sum, *p;
...
sum = 0;
for (p = a; p <= a + 9; p++)
  sum += *p;
```

Array Arguments

- When we pass an array to a function using its name, we now know that we are passing a pointer to the array's first element
- It's a pointer already; no &!
- We usually pass the length of the array to functions that operate on arrays
- There's no way for a function to find out the length of an array otherwise

```
int findLargest (int n, int a[n])
{ ...}

...
int b[] = {5, 2, 4, 1, 9, 3};
findLargest (6, b);
```

Array Arguments...

- Since we pass a pointer to the start of the array, not a copy of the array, functions can change arrays

```
void storeZeros (int n, int a[n]) {
  int i;
  for (i = 0; i < n; i++)
    a[i] = 0;
}
```

ConcepTest

What does the following code fragment do?

```
void storeZeros (int n, int a[n]) {
  int i;
  for (i = 0; i < n; i++)
    a[i] = 0;
}

...

int a[10];
storeZeros (5, &a[5]);
```

- A. Stores zeros in `a[0]`, `a[1]`, ..., `a[4]`
- B. Stores zeros in `a[0]`, `a[1]`, ..., `a[5]`
- C. Stores zeros in `a[5]`, `a[6]`, ..., `a[9]`
- D. Erroneously tries to store zeros in `a[5]`, `a[6]`, ..., `a[10]`
- E. Causes a compiler error

Suggested Exercise: Transpose

A matrix is a rectangular array of values. The transpose of a matrix is the matrix obtained by interchanging the rows and columns of the original matrix. As an example, the transpose of

3	1	4
2	0	7

is

3	2
1	0
4	7

Write a C function that has two parameters — both two-dimensional arrays of `int` values. The first array, called `old`, is of size `ROWS*COLS` while the second array, called `new`, is of size `COLS*ROWS`. The function should return the value `true` if and only if `new` is the transpose of `old`.

Suggested Exercise: Magic Square

Write a function that returns true if and only if its parameter is a magic square. A magic square of order `n` is a square array containing the integers $1, 2, \ldots, n^2$, each one used exactly once. The values in each row, column, and diagonal must sum to the same value. As an example, the following array is a magic square of order 3.

4	3	8
9	5	1
2	7	6

```
bool isMagic (int n, int square[n][n])
```