

CSE 31

Computer Organization

Lecture 4 – C Pointers (cont.),
Arrays

Announcement

▶ Labs

- Lab 1 due this week (**no grace period** after due date)
 - Demo is REQUIRED to receive full credit
- Lab 2 out this week
 - Due at 11:59pm on the same day of your next lab
 - You must demo your submission to your TA within 14 days

▶ Reading assignment

- Chapter 4-6, 8.7 of K&R (C book)
- Reading 01 (zyBooks 1.1 – 1.5) due 20-SEP
 - Complete Participation Activities in each section to receive grade towards Participation
 - IMPORTANT: Make sure to submit score to CatCourses by using the link provided on CatCourses

Announcement

▶ Homework assignment

- Homework 01 (zyBooks 1.1 – 1.5) due 27-SEP
 - Complete *Challenge Activities* in each section to receive grade towards Homework
 - IMPORTANT: Make sure to submit score to CatCourses by using the link provided on CatCourses

Arrays (1/5)

- ▶ Declaration:

```
int ar[2];
```

declares a 2-element integer array. An array is really just a block of memory.

```
int ar[] = {795, 635};
```

declares and fills a 2-element integer array.

- ▶ Accessing elements:

```
ar[num]
```

returns the $(\text{num}+1)^{\text{th}}$ element.

Arrays (2/5)

- ▶ Arrays are (almost) identical to pointers
 - `char *string` and `char string[]` are nearly identical declarations
 - They differ in very subtle ways: incrementing, declaration of filled arrays
- ▶ **Key Concept:** An array variable is a “pointer” to the first element.

Arrays (3/5)

► Consequences:

- `ar` is an array variable but looks like a pointer in many respects (though not all)
- `ar[0]` is the same as `*ar`
- `ar[2]` is the same as `*(ar+2)`
- We can use pointer arithmetic to access arrays more conveniently.

► Declared arrays are only allocated while the scope is valid

```
char *foo() {  
    char string[32];  
    ...;  
    return string;  
} is incorrect!
```

What's wrong?

Arrays (4/5)

- ▶ Good practice: You should use a counter AND utilize a variable for declaration & checking for bounds

- Not as good:

```
int i, ar[10];  
for(i = 0; i < 10; i++) { ... }
```

- Better:

```
int ARRAY_SIZE = 10  
int i, a[ARRAY_SIZE];  
for(i = 0; i < ARRAY_SIZE; i++) { ... }
```

- ▶ Why? SINGLE SOURCE OF TRUTH

- You're utilizing **indirection** and avoiding maintaining two copies of the number 10

Arrays (5/5)

- ▶ Pitfall: An array in C does not know its own length, & bounds not checked!
 - Consequence: We can accidentally access off the end of an array.
 - Consequence: We must pass the array and its size to a procedure which is going to traverse it.
- ▶ **Segmentation faults:**
 - These are VERY difficult to find;
be careful! (You'll learn how to debug these in lab...)

Arrays (one element past array must be valid)

- ▶ With array size n , we want to access from 0 to $n-1$. But test for exit by comparing to address one element past the array

```
int ar[10], *p, *q, sum = 0;
```

```
...
```

```
p = &ar[0]; q = &ar[10];
```

```
while (p != q)
```

```
    /* sum = sum + *p; p = p + 1; */
```

```
    sum += *p++;
```

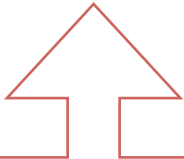
- Is this legal?
- ▶ C defines that one element past end of array **must be a valid address**, i.e., not cause a bus or address error

Arrays vs. Pointers

- ▶ An array name is a *read-only* pointer to the 1st element of the array.
- ▶ An array parameter can be declared as an array or a pointer; an array argument can be passed as a pointer.

```
int strlen(char s[])  
{  
    int n = 0;  
    while (s[n] != 0)  
        n++;  
    return n;  
}
```

```
int strlen(char *s)  
{  
    int n = 0;  
    while (s[n] != 0)  
        n++;  
    return n;  
}
```



Could be written:
while (s[n])

Pointer Arithmetic (1/3)

- ▶ Since a pointer is just a memory address, we can add to it to traverse an array.
- ▶ $p+1$ returns a pointer to the next array element
- ▶ $*p++$ vs $(*p)++$?
 - $x = *p++ \rightarrow x = *p ; p = p + 1 ;$
 - $x = (*p)++ \rightarrow x = *p ; *p = *p + 1 ;$
- ▶ What if we have an array of large structs (objects)?
 - C takes care of it: In reality, $p+1$ doesn't add 1 to the memory address, it adds the size of the array element.

Pointer Arithmetic (2/3)

- ▶ C knows the size of the thing a pointer points to – every addition or subtraction moves that many bytes.
 - 1 byte for a `char`, 4 bytes for an `int`, etc.
- ▶ So the following are equivalent:

```
int get(int array[], int n)
{
    return  (array[n]);
    // OR...
    return *(array + n);
}
```

Pointer Arithmetic (3/3)

- ▶ What is valid pointer arithmetic?
 - Add an integer to a pointer.
 - Subtract integer from pointer.
 - Subtract 2 pointers (in the same array).
 - Compare pointers ($<$, $<=$, $=$, $!=$, $>$, $>=$)
 - Compare pointer to `NULL` (indicates that the pointer points to nothing).
- ▶ Everything else is illegal since it makes no sense:
 - adding two pointers
 - multiplying pointers
 - subtract pointer from integer

Pointer Arithmetic to Copy Memory

- ▶ We can use pointer arithmetic to “walk” through memory:

```
void copy(int *from, int *to, int n) {  
    int i;  
    for (i=0; i<n; i++) {  
        *to++ = *from++;  
    }  
}
```

- Note we had to pass size (n) to copy