CSE 31
Computer Organization

Lecture 5 – C Pointers (wrap up) and C strings

Announcements

- Labs
  - Lab 1 due this week (with 7 days 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 (with 7 days grace period after due date)
    - » You must demo your submission to your TA within 14 days from posting of lab
    - » Demo is REQUIRED to receive full credit
- Reading assignment
  - Chapter 4-6 of K&R (C book) to review C/C++ programming
  - Reading 01 (zyBooks 1.1 – 1.5) due 13-FEB
    - » 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
- Homework assignment
  - Homework 01 (zyBooks 1.1 – 1.5) due 20-FEB
    - » 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

Pointer Arithmetic (review)

- Since a pointer is just a memory address, we can add to it to traverse an array (when a pointer points to it)
  - 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.
- 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 Summary

- `x = *(p + 1)` ?
- `x = *p + 1` ?
- `x = (*p)++` ?
- `x = *p++` ? or `(*p++)` ? or `*(p)++` ? or `*(p++)` ?
- `x = *++p` ?
- `x = ++*p` ?

Pointer Arithmetic Summary

- `x = *(p + 1)` ?
  - `x = *(p + 1);`
- `x = *p + 1` ?
  - `x = (*p) + 1;`
- `x = (*p)++` ?
  - `x = *p; *p = *p + 1;`
- `x = *p++` ? or `(*p++)` ? or `*(p)++` ? or `*(p++)` ?
  - `x = *p; p =  p + 1;`
- `x = *++p` ?
  - `p = p + 1; x = *p;`
- `x = ++*p` ?
  - `*p = *p + 1; x = *p;`

- Lesson?
  - Using nothing but the standard `*p++` , `(*p)++` causes more problems than it solves!

Quiz:

How many of the following are invalid?

I. pointer + integer

II. integer + pointer

III. pointer + pointer

IV. pointer – integer

V. integer – pointer

VI. pointer – pointer

VII. compare pointer to pointer

VIII. compare pointer to integer

IX. compare pointer to 0

X. compare pointer to `NULL`

```
#invalid
  a) 1
  b) 2
  c) 3
  d) 4
  e) 5
```

Quiz:

How many of the following are invalid?

I. pointer + integer

II. integer + pointer

III. pointer + pointer

IV. pointer – integer

V. integer – pointer

VI. pointer – pointer

VII. compare pointer to pointer

VIII. compare pointer to integer

IX. compare pointer to 0

X. compare pointer to `NULL`

```
#invalid
   a) 1
   b) 2
   c) 3
   d) 4
   e) 5
```

Pointers (1/4)

- Sometimes you want to have a function to increment a variable
- What gets printed?

```
void AddOne(int x) {
    x = x + 1;
}

int main() {
    int y = 5;
    AddOne(y);
    printf("y = %d\n", y);
    return 0;
}
```

Output:

```
y = 5
```

Pointers (2/4)

- Solved by passing in a pointer to our subroutine.
- Now what gets printed?

```
void AddOne(int *p) {
     *p =  *p + 1;
}

int main() {
     int y = 5;
     AddOne(&y);
     printf("y = %d\n", y);
     return 0;
}
```

Output:

y = 6

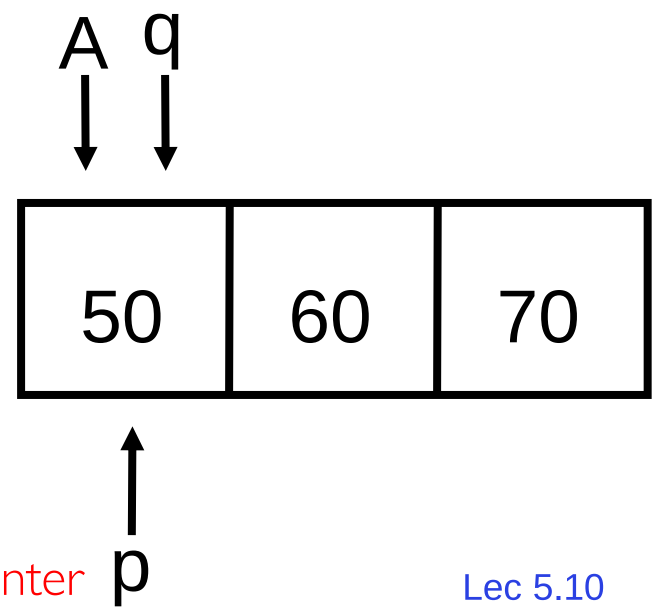
Pointers (3/4)

- But what if what you want changed is a pointer
- What gets printed?

```
void IncrementPtr(int *p) {
    p =  p + 1;
}

int main() {
    int A[3] = {50, 60, 70};
    int *q = A;
    IncrementPtr(q);
    printf("*q = %d\n", *q);
    return 0;
}
```

Output:

```
*q = 50
```



A q

50 60 70

p

A clone of pointer p

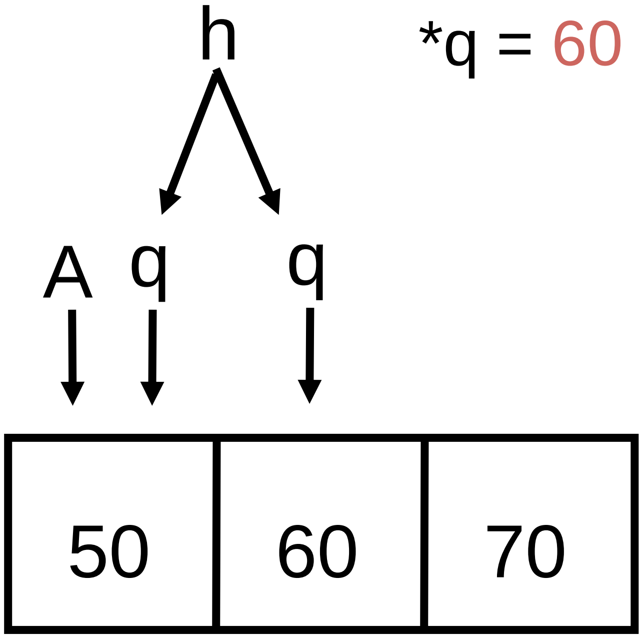
Pointers (4/4)

- Solution! Pass a pointer to a pointer, declared as `**h`
- Now what gets printed?

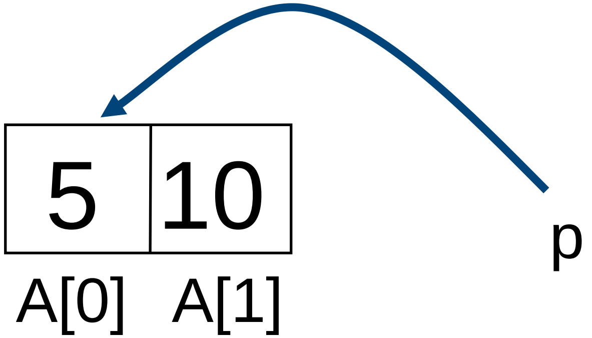
```
void IncrementPtr(int **h) {
    *h = *h + 1;
}

int main() {
    int A[3] = {50, 60, 70};
    int *q = A;
    IncrementPtr(&q);
    printf("*q = %d\n", *q);
    return 0;
}
```

Output:

*q = 60



Quiz:

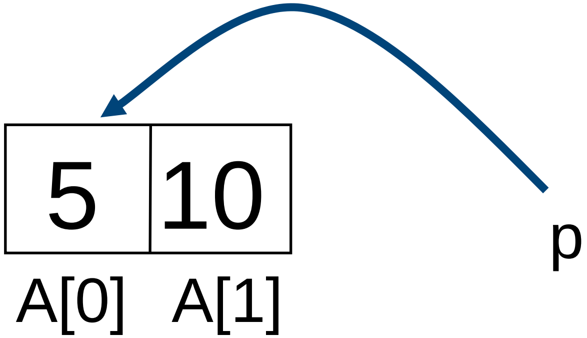


```
int main(void){
    int A[] = {5,10};
    int *p = A;

    printf(“%p %d %d %d\n”, p, *p, A[0], A[1]);
    p =  p + 1;
    printf(“%p %d %d %d\n”, p, *p, A[0], A[1]);
    *p = *p + 1;
    printf(“%p %d %d %d\n”, p, *p, A[0], A[1]);
}
```

- If the first `printf` outputs 100 5 5 10, what will the other two `printf` output?

a) 101 10 5 10 then 101 11 5 11
b) 104 10 5 10 then 104 11 5 11
c) 101 <other> 5 10 then 101 <3-others>
d) 104 <other> 5 10 then 104 <3-others>
e) One of the two `printf` statements causes an ERROR

Quiz:



```
int main(void){
    int A[] = {5,10};
    int *p = A;

    printf(“%p %d %d %d\n”, p, *p, A[0], A[1]);
    p =  p + 1;
    printf(“%p %d %d %d\n”, p, *p, A[0], A[1]);
    *p = *p + 1;
    printf(“%p %d %d %d\n”, p, *p, A[0], A[1]);
}
```

- If the first `printf` outputs 100 5 5 10, what will the other two `printf` output?

a) 101 10 5 10 then 101 11 5 11
b) 104 10 5 10 then 104 11 5 11
c) 101 <other> 5 10 then 101 <3-others>
d) 104 <other> 5 10 then 104 <3-others>
e) One of the two `printf` statements causes an ERROR

Pointers in C

- Why use pointers?
  - If we want to pass a huge struct or array, it's easier / faster to pass a pointer than the whole thing.
  - In general, pointers allow cleaner, more compact code.

- So, what are the drawbacks?
  - Pointers are probably the single largest source of bugs in software, so be careful anytime you deal with them.
    - » Dangling reference (premature free)
    - » Memory leaks (tardy free)

- Make sure you know what you are doing!

Pointers Summary

- Pointers and arrays are virtually the same

- C knows how to increment pointers

- C is an efficient language, with little protection
  - Array bounds not checked
  - Variables not automatically initialized

- (Beware) The cost of efficiency is more overhead for the programmer.

C Strings

- A string in C is just an array of characters.

```
char string[] = "abc";
```

- How do you tell how long a string is?
  - Last character is followed by a 0 byte (null terminator)

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

C Strings Headaches

- One common mistake is to forget to allocate an extra byte for the null terminator.

- More generally, C requires the programmer to manage memory manually (unlike Java or C++).
  - When creating a long string by concatenating several smaller strings, the programmer must ensure there is enough space to store the full string!
  - What if you don't know ahead of time how big your string will be?
    - » Buffer overrun security holes!

C String Standard Functions

- `int strlen(char *string);`
  - compute the length of `string`

- `int strcmp(char *str1, char *str2);`
  - return 0 if `str1` and `str2` are identical
  - how is this different from `str1 == str2`?

- `char *strcpy(char *dst, char *src);`
  - copy the contents of string `src` to the memory at `dst`. The caller must ensure that `dst` has enough memory to hold the data to be copied.