

CSE 31

Computer Organization

Lecture 2 – C Programming (2)

C Pointers



Announcement

- ▶ Lab #1 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 of K&R (C book) to review on C/C++ programming

Address vs. Value

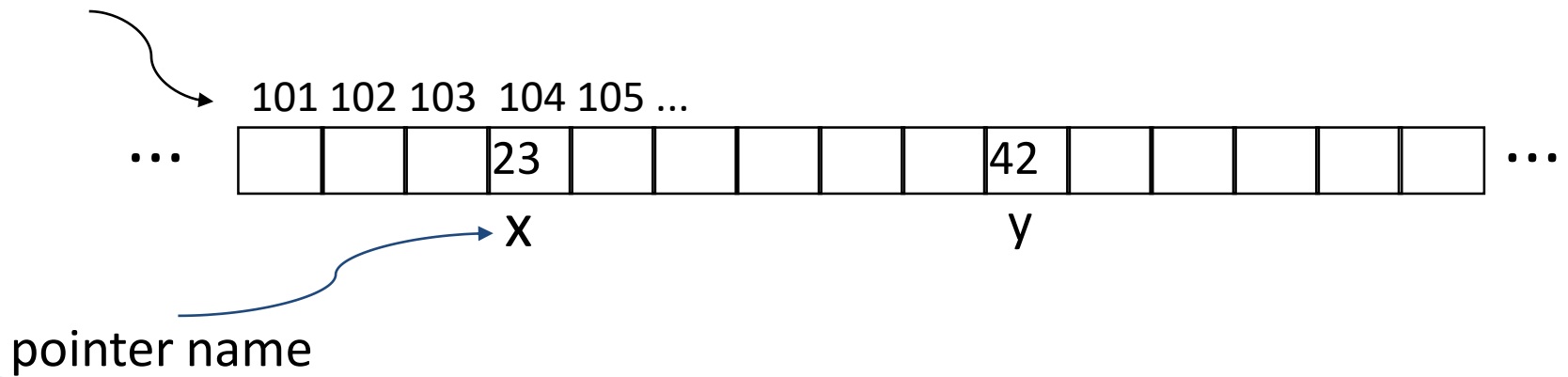
- ▶ Consider memory to be a single huge array:
 - Each cell of the array has an address associated with it.
 - Each cell also stores some value.
 - Do you think addresses use signed or unsigned numbers?
 - Negative address?!
- ▶ Don't confuse the **address** referring to a memory location with the **value** stored in that location.



Pointers

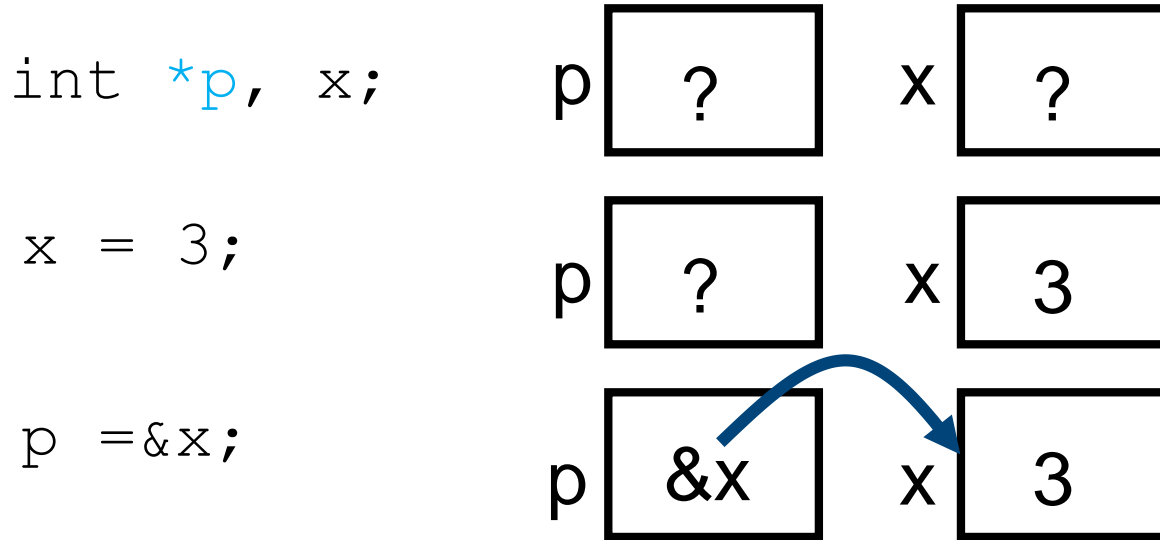
- ▶ An address refers to a particular memory location. In other words, it points to a memory location.
- ▶ **Pointer**: A variable that contains the address of a variable.

Location (address)



Pointers

- ▶ How to create a pointer:
 & **operator**: get address of a variable



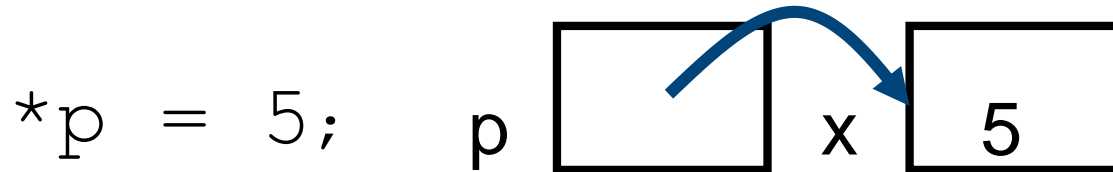
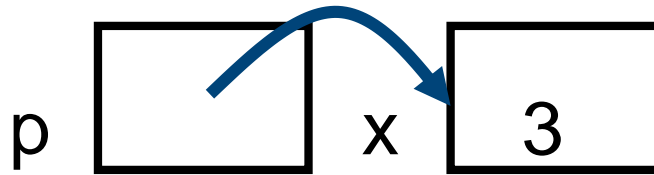
Note the “*” gets used 2 different ways in this example. In the declaration to indicate that `p` is going to be a pointer, and in the `printf` to get the value pointed to by `p`.

- How to get a value pointed to?
 - * “dereference operator”: get value pointed to

```
printf("p points to %d\n", *p);
```

Pointers

- ▶ How to change a variable pointed to?
 - Use dereference $*$ operator on left of =



Pointers and Parameter Passing

- ▶ Java and C pass parameters “by value”
 - procedure/function/method gets a copy of the parameter, so changing the copy cannot change the original

```
void addOne (int x) {  
    x = x + 1;  
}  
  
int y = 3;  
addOne(y);
```

y is still = 3 after the program ends!

Pointers and Parameter Passing

- ▶ How to get a function to change a value?

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

```
int y = 3;
```

```
addOne (&y) ;
```

Passing the reference of y



y is now = 4 after the program ends.

Pointers

- ▶ Pointers are used to point to **any** data type (`int`, `char`, `a struct`, etc.).
- ▶ Normally a pointer can only point to one type (`int`, `char`, `a struct`, etc.).
 - `void *` is a type that can point to anything (generic pointer)
 - Use sparingly to help avoid program bugs... and security issues... and a lot of other bad things!

Pointers & Allocation (1/2)

- ▶ After declaring a pointer:

```
int *ptr;
```

`ptr` doesn't actually point to anything yet (it actually points somewhere - but we don't know where!).

- ▶ We can either:
 - make it point to something that already exists, or
 - allocate room in memory for something new that it will point to... (we will talk about it later)

C Pointer Dangers

- ▶ Unlike Java, C lets you **cast** a value of any type to any other type without performing any checking.

```
int x = 1000;  
int *p = x;           /* invalid */  
int *q = (int *) x;  /* valid */
```

- ▶ The first pointer declaration is invalid since the types do not match. (unsigned vs signed)
- ▶ The second declaration is valid in C but is almost certainly wrong
 - Is it ever correct?

More C Pointer Dangers

- ▶ Declaring a pointer just allocates space to hold the pointer – it does not allocate anything to be pointed to!
- ▶ Local variables in C are not initialized, they may contain anything.
- ▶ What does the following code do?

```
void f() {  
    int *ptr;  
    *ptr = 5;  
}
```

Where does it store the “5”?

Pointers & Allocation (2/2)

- ▶ Pointing to something that already exists:

```
int *ptr, var1, var2;
```

- ▶ `var1` and `var2` have room implicitly allocated for them.

ptr ?

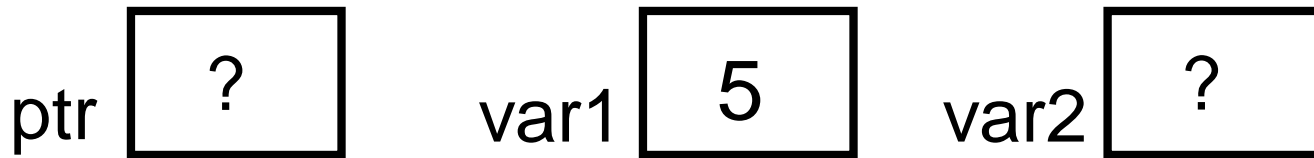
var1 ?

var2 ?

Pointers & Allocation (2/2)

- ▶ Pointing to something that already exists:

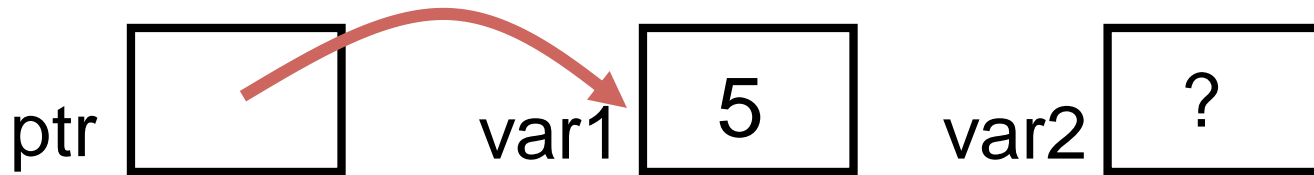
```
int *ptr, var1, var2;  
var1 = 5;
```



Pointers & Allocation (2/2)

- ▶ Pointing to something that already exists:

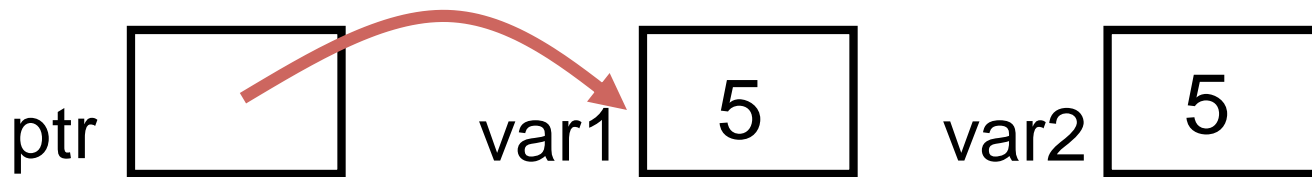
```
int *ptr, var1, var2;  
var1 = 5;  
ptr = &var1;
```



Pointers & Allocation (2/2)

- ▶ Pointing to something that already exists:

```
int *ptr, var1, var2;  
var1 = 5;  
ptr = &var1;  
var2 = *ptr;
```



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)

- ▶ Array size n ; want to access from 0 to $n-1$, so you should use counter AND utilize a variable for declaration & incr
 - Wrong

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

```
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** and **bus errors**:
 - These are VERY difficult to find;
be careful! (You'll learn how to debug these in lab...)

Segmentation Fault vs Bus Error?

▶ Segmentation Fault

- An error in which a running Unix program attempts to **access memory not allocated** to it and terminates with a segmentation violation error and usually a core dump.

▶ Bus Error

- A fatal failure in the execution of a machine language instruction resulting from the processor detecting an anomalous condition on its bus (communicating connection between hardware) . Such conditions include **invalid address alignment** (accessing a multi-byte number at an odd address), accessing a physical address that does not correspond to any device, or some other device-specific hardware error. A bus error triggers a processor-level exception which Unix translates into a “SIGBUS” signal which, if not caught, will terminate the current process.

Arrays (one element past array must be valid)

- ▶ Array size n ; 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++;
```

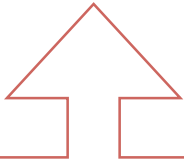
- Is this legal?
- ▶ C defines that one element past end of array **must be a valid address**, i.e., not cause an bus error or address error

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 mem address, we can add to it to traverse an array.
- ▶ $p+1$ returns a ptr 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 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

Pointer Arithmetic Summary

- ▶ $x = *(p+1)$?
 - $x = *(p+1)$;
- ▶ $x = *p+1$?
 - $x = (*p) + 1$;
- ▶ $x = (*p)++$?
 - $x = *p$; $*p = *p + 1$;
- ▶ $x = *p++$? $(*p++)$? $*(p)++$? $*(p++)$?
 - $x = *p$; $p = p + 1$;
- ▶ $x = *++p$?
 - $p = p + 1$; $x = *p$;
- ▶ Lesson?
 - Using nothing but the standard $*p++$, $(*p)++$ causes more problems than it solves!

Pointers (1/4)

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

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

y = 5

```
int y = 5;
AddOne(y);
printf("y = %d\n", y);
```

Pointers (2/4)

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

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

y = 6

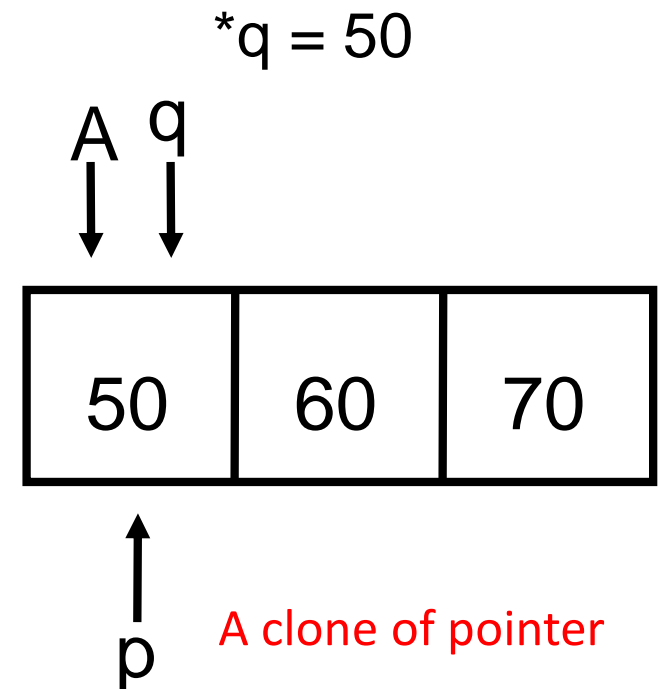
```
int y = 5;
AddOne(&y);
printf("y = %d\n", y);
```

Pointers (3/4)

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

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

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

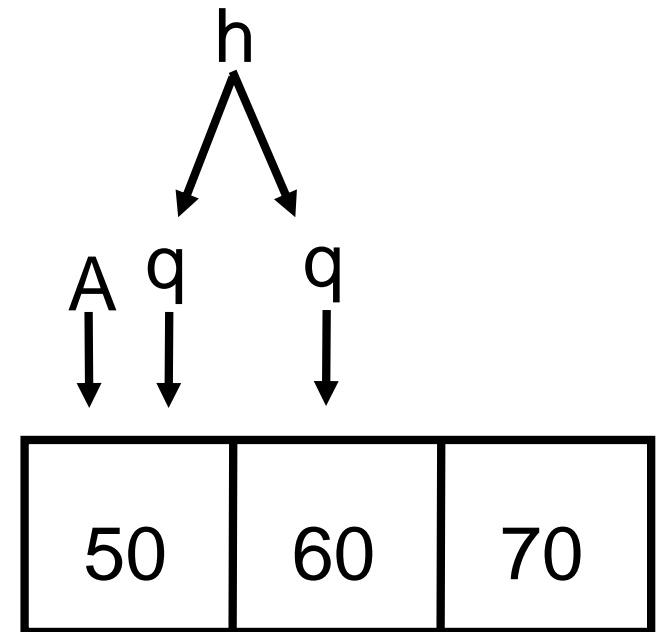


Pointers (4/4)

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

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

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



`*q = 60`

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

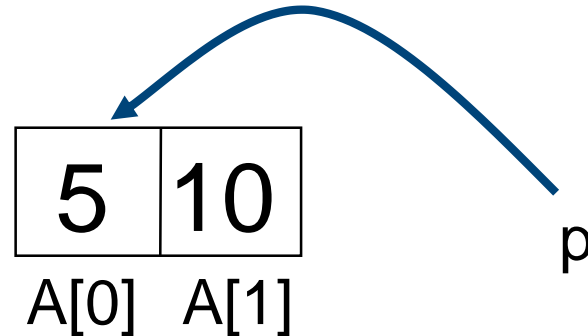
e) 5

Quiz:

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

```
  printf("%u %d %d %d\n",p,*p,A[0],A[1]);  
  p = p + 1;  
  printf("%u %d %d %d\n",p,*p,A[0],A[1]);  
  *p = *p + 1;  
  printf("%u %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 causes an ERROR

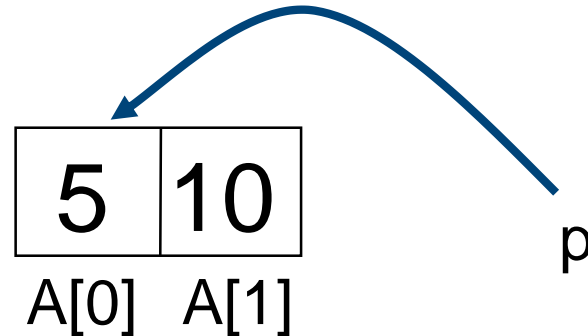


Quiz:

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

```
  printf("%u %d %d %d\n",p,*p,A[0],A[1]);  
  p = p + 1;  
  printf("%u %d %d %d\n",p,*p,A[0],A[1]);  
  *p = *p + 1;  
  printf("%u %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 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!

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 gives you a lot of extra rope but be careful not to hang yourself with it!”