

## Lecture 4 – Introduction to C (pt 2)



**2010-01-27**

**C review update: Tonight  
7:30-8:30pm @ 306 Soda**

**Lecturer SOE Dan Garcia**

**[www.cs.berkeley.edu/~ddgarcia](http://www.cs.berkeley.edu/~ddgarcia)**

**All eyes on Apple ⇒  
Today, Apple will make a  
big announcement; many have  
speculated it's a "tablet" (much like the  
iPhone) which will change the industry.**



**[www.nytimes.com/2010/01/26/technology/26apple.html](http://www.nytimes.com/2010/01/26/technology/26apple.html)**

# Review

---

- All declarations go at the beginning of each function **except if you use C99**.
- Only 0 and NULL evaluate to FALSE.
- All data is in memory. Each memory location has an address to use to refer to it and a value stored in it.
- A **pointer** is a C version of the address.
  - \* “follows” a pointer to its value
  - & gets the address of a value



# More C Pointer Dangers

---

- Declaring a pointer just allocates space to hold the pointer – it does not allocate something 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;  
}
```



# 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-elt integer array.

- **Accessing elements:**

```
ar[num]
```

 returns the  $\text{num}^{\text{th}}$  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
```



## 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...)**





# Pointer Arithmetic (1/2)

---

- 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 elt.
- $*p++$  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 (2/2)

---

- **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);
}
```



# Pointers in C

---

- **Why use pointers?**
  - If we want to pass a huge struct or array, it's easier / faster / etc 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)



# 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;  
}
```



# Pointer Arithmetic Peer Instruction Q

---

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



# Pointer Arithmetic Peer Instruction Ans

- How many of the following are **invalid**?
  - pointer + integer  $\text{ptr} + 1$
  - integer + pointer  $1 + \text{ptr}$
  - pointer + pointer  **$\text{ptr} + \text{ptr}$**
  - pointer – integer  $\text{ptr} - 1$
  - integer – pointer  **$1 - \text{ptr}$**
  - pointer – pointer  $\text{ptr} - \text{ptr}$
  - compare pointer to pointer  $\text{ptr1} == \text{ptr2}$
  - compare pointer to integer  **$\text{ptr} == 1$**
  - compare pointer to 0  $\text{ptr} == \text{NULL}$
  - compare pointer to NULL  $\text{ptr} == \text{NULL}$

#invalid

a) 1

b) 2

**c) 3**

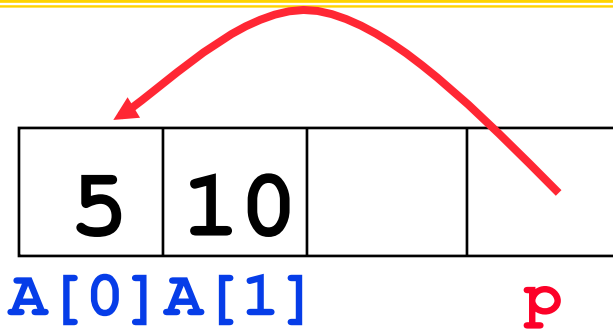
d) 4

e) 5



# Peer Instruction

```
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]);
}
```

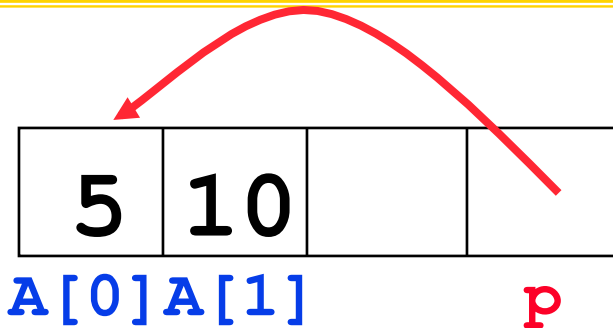
If the first printf outputs 100 5 5 10, what will the other two printf output?

- 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



# Peer Instruction Answer

```
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





## “And in Conclusion...”

---

- Pointers and arrays are **virtually 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!”**



# Reference slides

---

**You ARE responsible for the material on these slides (they're just taken from the reading anyway) ; we've moved them to the end and off-stage to give more breathing room to lecture!**



# Administrivia

---

- Read K&R 6 by the next lecture
- There is a language called D!
  - [www.digitalmars.com/d/](http://www.digitalmars.com/d/)
- Homework expectations
  - Readers don't have time to fix your programs which have to run on lab machines.
  - Code that doesn't compile or fails all of the autograder tests  $\Rightarrow$  0



# Administrivia

---

- **Slip days**

- You get 3 “slip days” per year to use for any homework assignment or project
- They are used at 1-day increments. Thus 1 minute late = 1 slip day used.
- They’re recorded automatically (by checking submission time) so you don’t need to tell us when you’re using them
- Once you’ve used all of your slip days, when a project/hw is late, it’s ... 0 points.
- If you submit twice, we ALWAYS grade the latter, and deduct slip days appropriately
- You no longer need to tell anyone how your dog ate your computer.
- You should really save for a rainy day ... we all get sick and/or have family emergencies!



# Pointers & Allocation (1/2)

---

- After declaring a pointer:

```
int *ptr;
```

`ptr` doesn't actually point to anything yet (*it actually points somewhere - but 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... (next time)



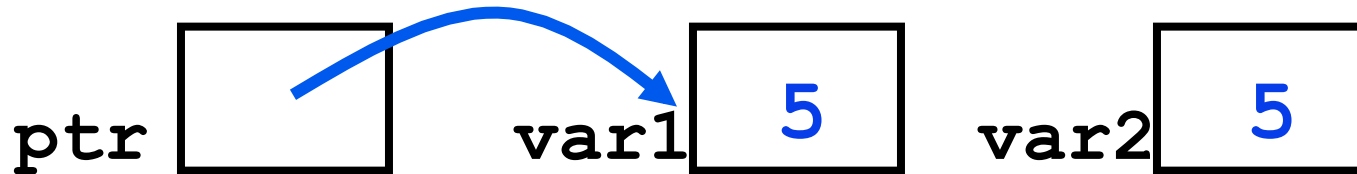
# Pointers & Allocation (2/2)

---

- Pointing to something that already exists:

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

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



# Arrays (one elt 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



# Pointer Arithmetic

---

- **So what's 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



# Arrays vs. Pointers

---

- An array name is a read-only pointer to the 0<sup>th</sup> 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 strlen(char *s)
{
    int n = 0;           {
    while (s[n] != 0)    int n = 0;
        n++;             while (s[n] != 0)
    return n;            n++;
}                       return n;
}
```

Could be written:  
`while (s[n])`



# Pointer Arithmetic Summary

- $x = *(p+1) ?$

$\Rightarrow x = *(p+1) ;$

- $x = *p+1 ?$

$\Rightarrow x = (*p) + 1 ;$

- $x = (*p)++ ?$

$\Rightarrow x = *p ; *p = *p + 1 ;$

- $x = *p++ ? (*p++) ? *(p)++ ? *(p++) ?$

$\Rightarrow x = *p ; p = p + 1 ;$

- $x = *++p ?$

$\Rightarrow p = p + 1 ; x = *p ;$

- Lesson?



• Using anything but the standard  $*p++$ ,  $(*p)++$  causes more problems than it solves!

# Segmentation Fault vs Bus Error?

---

- <http://www.hyperdictionary.com/>
- **Bus Error**
  - A fatal failure in the execution of a machine language instruction resulting from the processor detecting an anomalous condition on its bus. 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.
- **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.



# 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.
- The second declaration is valid C but is almost certainly wrong



• Is it ever correct?

# 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 insure 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!**



# Common C Error

---

- There is a difference between assignment and equality

`a = b` is assignment

`a == b` is an equality test

- This is one of the most common errors for beginning C programmers!
  - One solution (when comparing with constant) is to put the var on the right!  
If you happen to use =, it won't compile.

```
if (3 == a) { ...
```



# 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.

