

## Lecture 4 – C Pointers

2005-01-26



Lecturer PSOE Dan Garcia

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

Machine learns games! ⇒

A computer has learned to play “rock, paper, scissors” by observing and mimicking humans! This is one step toward the game theory holy grail - show it the rules and it plays well!



[www.comp.leeds.ac.uk/vision/cogvis/games.html](http://www.comp.leeds.ac.uk/vision/cogvis/games.html)

CS61C L04 C Pointers (1)

Garcia, Spring 2005 © UCB

## Pointers & Allocation (1/2)

- After declaring a pointer:

```
int *ptr;
```

`ptr` doesn't actually point to anything yet. 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)



CS61C L04 C Pointers (2)

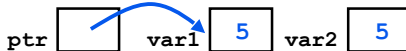
Garcia, Spring 2005 © UCB

## 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.



CS61C L04 C Pointers (3)

Garcia, Spring 2005 © UCB

## 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;  
}
```



CS61C L04 C Pointers (4)

Garcia, Spring 2005 © UCB

## Arrays (1/6)

- Declaration:

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

declares a 2-element integer array.

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

declares and fills a 2-elt integer array.

- Accessing elements:

```
ar[num];
```

returns the `num`<sup>th</sup> element.



CS61C L04 C Pointers (5)

Garcia, Spring 2005 © UCB

## Arrays (2/6)

- 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.



CS61C L04 C Pointers (6)

Garcia, Spring 2005 © UCB

### Arrays (3/6)

#### • 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
```



CSE1C L04 C Pointers (7)

Garcia, Spring 2005 © UCB

### Arrays (4/6)

- 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 a bus error or address error



CSE1C L04 C Pointers (8)

Garcia, Spring 2005 © UCB

### Arrays (5/6)

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

#### • Wrong

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

#### • Right

```
#define 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



CSE1C L04 C Pointers (9)

Garcia, Spring 2005 © UCB

### Arrays (6/6)

- 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...)



CSE1C L04 C Pointers (10)

Garcia, Spring 2005 © UCB

### Segmentation Fault vs Bus Error?

- <http://www.hyperdictionary.com/>

#### • Segmentation Fault

- 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.

#### • Bus Error

- 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.



CSE1C L04 C Pointers (11)

Garcia, Spring 2005 © UCB

### 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 elt.

- **`(*p)+1` vs `*p++` vs `*(p+1)` 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**.



CSE1C L04 C Pointers (12)

Garcia, Spring 2005 © UCB

### Pointer Arithmetic (2/3)

- 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



CS61C L04 C Pointers (13)

Garcia, Spring 2005 © UCB

### Pointer Arithmetic (3/3)

- C knows the size of the thing a pointer points to – every addition or subtraction moves that many bytes.
- So the following are equivalent:

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



CS61C L04 C Pointers (14)

Garcia, Spring 2005 © UCB

### Pointers in C

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



CS61C L04 C Pointers (15)

Garcia, Spring 2005 © UCB

### 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?



CS61C L04 C Pointers (16)

Garcia, Spring 2005 © UCB

### Administrivia

- Read K&R 6 for Friday
- There is a language called D!
  - [www.digitalmars.com/d/](http://www.digitalmars.com/d/)
- Answers to the reading quizzes?
  - Ask your TA in discussion
- 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 ⇒ 0**



CS61C L04 C Pointers (17)

Garcia, Spring 2005 © UCB

### 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!



CS61C L04 C Pointers (18)

Garcia, Spring 2005 © UCB

## 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;
}
```



CS61C L04 C Pointers (19)

Garcia, Spring 2005 © UCB

## 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!



CS61C L04 C Pointers (20)

Garcia, Spring 2005 © UCB

## Common C Errors

- 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!



CS61C L04 C Pointers (21)

Garcia, Spring 2005 © UCB

## Pointer Arithmetic Peer Instruction Q

How many of the following are **invalid**?

- |                                  | #invalid |
|----------------------------------|----------|
| I. pointer + integer             | 1        |
| II. integer + pointer            | 2        |
| III. pointer + pointer           | 3        |
| IV. pointer - integer            | 4        |
| V. integer - pointer             | 5        |
| VI. pointer - pointer            | 6        |
| VII. compare pointer to pointer  | 7        |
| VIII. compare pointer to integer | 8        |
| IX. compare pointer to 0         | 9        |
| X. compare pointer to NULL       | (1) 0    |

	#invalid
I. pointer + integer	1
II. integer + pointer	2
III. pointer + pointer	3
IV. pointer - integer	4
V. integer - pointer	5
VI. pointer - pointer	6
VII. compare pointer to pointer	7
VIII. compare pointer to integer	8
IX. compare pointer to 0	9
X. compare pointer to NULL	(1) 0



CS61C L04 C Pointers (22)

Garcia, Spring 2005 © UCB

## “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!”



CS61C L04 C Pointers (24)

Garcia, Spring 2005 © UCB

## Kilo, Mega, Giga, Tera, Peta, Exa, Zetta, Yotta

1. Kim's melodious giddiness terrifies people, excepting zealous yodelers
2. Kirby Messed Gigglypuff Terribly, (then) Perfectly Exterminated Zelda and Yoshi
3. Killed meat gives teeth peace except zebra yogurt
4. Kind Men Give Tense People Extra Zeal (for) Yoga
5. Killing melee gives terror; peace exhibits Zen yoga
6. Killing messengers gives terrible people exactly zero, yo
7. Kindergarten means giving teachers perfect examples (of) zeal (&) youth
8. Kissing mediocre girls teaches people (to) expect zero (from) you
9. Kinky Mean Girls Teach Penis-Extending Zen Yoga
10. Kissing Mel Gibson, Teddy Pendergrass exclaimed, “Zesty, yo!”



CS61C L04 C Pointers (28)

Garcia, Spring 2005 © UCB