

CS 61C:
Great Ideas in Computer Architecture
Introduction to C, Part II

Instructor:

Randy H. Katz

<http://inst.eecs.Berkeley.edu/~cs61c/fa13>

9/10/13

Fall 2013 -- Lecture #4

1

Agenda

- Pointers and Arrays
- Administrivia
- Pointer arithmetic
- Arrays vs. pointers
- Technology Break
- Pointer Problems
- Criticisms of C
- And in Conclusion, ...

9/10/13

Fall 2013 -- Lecture #4

2

Agenda



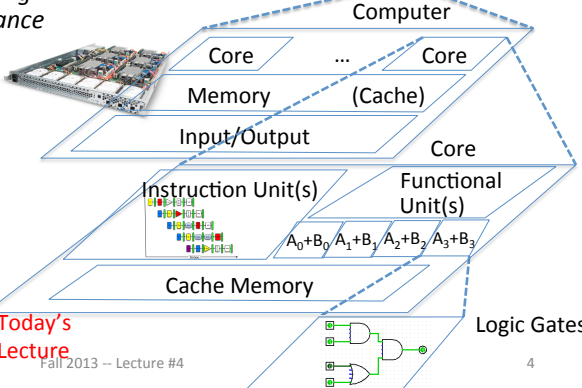
- Pointers and Arrays
- Administrivia
- Pointer arithmetic
- Arrays vs. pointers
- Technology Break
- Pointer Problems
- Criticisms of C
- And in Conclusion, ...

9/10/13

Fall 2013 -- Lecture #4

3

New-School Machine Structures (It's a bit more complicated!)

<ul style="list-style-type: none"> • Parallel Requests Assigned to computer e.g., Search "Katz" • Parallel Threads Assigned to core e.g., Lookup, Ads • Parallel Instructions >1 instruction @ one time e.g., 5 pipelined instructions • Parallel Data >1 data item @ one time e.g., Add of 4 pairs of words • Hardware descriptions All gates @ one time • Programming Languages 	<p style="font-size: 2em; color: blue;"> </p>	<p style="text-align: center;"><i>Software</i> <i>Hardware</i></p> <div style="display: flex; justify-content: space-between;"> <div style="width: 45%;"> <p>Warehouse Scale Computer</p>  </div> <div style="width: 10%; text-align: center;">  <p>Smart Phone</p> </div> </div> <div style="margin-top: 20px;"> <p style="text-align: center;"><i>Harness Parallelism & Achieve High Performance</i></p>  </div>
---	---	---

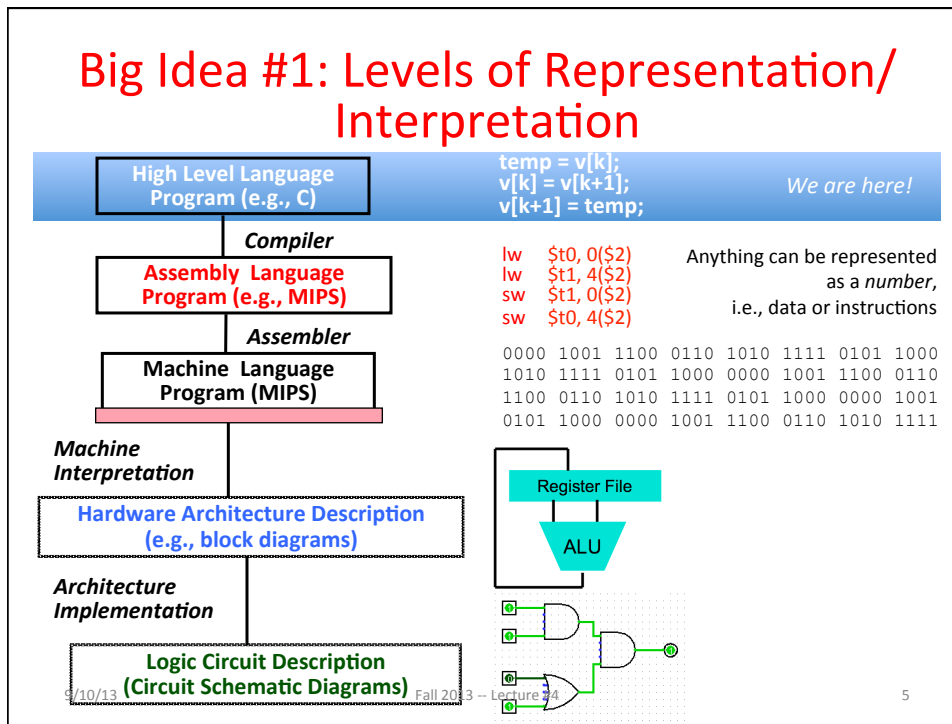
9/10/13

Today's Lecture

Fall 2013 -- Lecture #4

4

Big Idea #1: Levels of Representation/ Interpretation

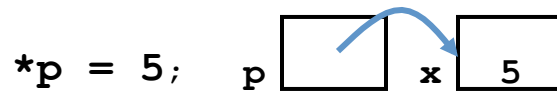
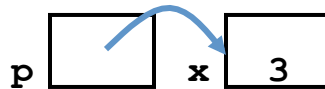


Pointer Review

- `int *x;`
 - Tells compiler that `variable x` is address of an `int`
- `x = &y;`
 - Tells compiler to assign `address of y` to `x`
 - `&` called the “address operator” in this context
- `z = *x;`
 - Tells compiler to assign `value at address in x` to `z`
 - `*` called the “dereference operator” in this context

Pointer Review

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



9/10/13

Fall 2013 -- Lecture #4

7

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 remains equal to 3
```

9/10/13

Fall 2013 -- Lecture #4

8

Pointers and Parameter Passing

- How can we get a function to change the value held in a variable?

```
void addOne (int *p) {
    *p = *p + 1;
}
int y = 3;
addOne(&y);
y is now equal to 4
```

9/10/13

Fall 2013 -- Lecture #4

9

C Pointer Dangers

- Declaring a pointer just allocates space to hold the pointer – it does not allocate the thing being pointed to!
- Local variables in C are not initialized, they may contain anything (aka “garbage”)
- What does the following code do?

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

9/10/13

Fall 2013 -- Lecture #4

10

Pointers and Structures

```

struct Point {          /* dot notation */
    int x;              int h = p1.x;
    int y;              p2.y = p1.y;
};

                          /* arrow notation */
Point p1;               int h = paddr->x;
Point p2;               int h = (*paddr).x;
Point *paddr;

                          /* This works too */
p1 = p2;

```

9/10/13

Fall 2013 -- Lecture #4

11

How many logic and syntax errors?



- ```

void main(); {
 int *p, x=5, y; // init
 y = *(p = &x) + 1;
 int z;
 flip-sign(p);
 printf("x=%d,y=%d,p=%d\n", x, y, p);
}
flip-sign(int *n){*n = -(*n)}

```
- 1
  - 2
  - 3
  - $\geq 4$

9/10/13

Fall 2013 -- Lecture #4

12

## Peer Instruction Answer

```
#insert <stdio.h>
void main(); { //int main(void) {
 int *p, x=5, y; // init
 y = *(p = &x) + 1;
 int z;
 flip_sign(p);
 printf("x=%d,y=%d,p=%d\n", x, y, *p);
}
flip_sign(int *n) { *n = -(*n); }
// return (0); }
```

*More than four syntax + logic errors in this C code*

What is output after correct errors?



x=5, y=6, p=-5

x=-5, y=6, p=-5

x=-5, y=4, p=-5

x=-5, y=-6, p=-5

```
void main(); {
 int *p, x=5, y; // init
 int z;
 y = *(p = &x) + 1;
 flip_sign(p);
 printf("x=%d,y=%d,p=%d\n",
 x, y, *p);
}
flip_sign(int *n)
 { *n = -(*n); }
```

What is output after correct errors?



$x=5, y=6, p=-5$

$x=-5, y=6, p=-5$

$x=-5, y=4, p=-5$

$x=-5, y=-6, p=-5$

```
void main(); {
 int *p, x=5, y; // init
 int z;
 y = *(p = &x) + 1;
 flip_sign(p);
 printf("x=%d,y=%d,p=%d\n",
 x, y, *p);
}
flip_sign(int *n)
{ *n = -(*n); }
```

## Arrays (1/5)

- Declaration:

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

declares a 2-element integer array: just a block of memory

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

declares and initializes a 2-element integer array

- Accessing elements:

```
ar[num]
```

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



## Arrays (2/5)

- Arrays are (almost) identical to pointers
  - `char *string` and `char string[]` are nearly identical declarations
  - Differ in subtle ways: incrementing, declaration of filled arrays
  - End of C string marking by 0 in last character
- *Key Concept*: Array variable is a “pointer” to the first (0<sup>th</sup>) element

9/10/13

Fall 2011 -- Lecture #4

17

## C Strings

- 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 (aka “null terminator”)

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

9/10/13

Fall 2013 -- Lecture #4

18

## Arrays (3/5)

- Consequences:
  - `ar` is an array variable, but looks like a pointer
  - `ar[0]` is the same as `*ar`
  - `ar[2]` is the same as `*(ar+2)`
  - We can use pointer arithmetic to conveniently access arrays
- Declared arrays are only allocated while the scope is valid

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

is incorrect *and very very bad*

## 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 & incrementation
  - Bad pattern
 

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

```
int ARRAY_SIZE = 10
int i, a[ARRAY_SIZE];
for(i = 0; i < ARRAY_SIZE; i++){ ... }
```
- SINGLE SOURCE OF TRUTH
  - You're utilizing indirection and avoiding maintaining two copies of the number 10
  - DRY: "Don't Repeat Yourself"

## Arrays (5/5)

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

9/10/13

Fall 2011 -- Lecture #4

21

## Array And in Conclusion ...

- Array indexing is syntactic sugar for pointers
- `a[i]` is treated as `*(a+i)`
- E.g., three equivalent ways to zero an array:
  - `for (i=0; i < size; i++) a[i] = 0;`
  - `for (i=0; i < size; i++) *(a+i) = 0;`
  - `for (p=a; p < a+size; p++) *p = 0;`

9/10/13

Fall 2011 -- Lecture #4

22

What is TRUE about this function?



```
void foo(char *s, char *t)
{ while (*s)
 s++;
 while (*s++ = *t++)
 ;
}
```

- It has syntax errors
- No syntax errors; it changes characters in string `t` to next character in the string `s`
- No syntax errors; it copies a string at address `t` to the string at address `s`
- No syntax errors; it appends the string at address `t` to the end of the string at address `s`

23

What is TRUE about this function?



```
void foo(char *s, char *t)
{ while (*s)
 s++;
 while (*s++ = *t++)
 ;
}
```

- It has syntax errors
- No syntax errors; it changes characters in string `t` to next character in the string `s`
- No syntax errors; it copies a string at address `t` to the string at address `s`
- No syntax errors; it appends the string at address `t` to the end of the string at address `s`

24

Question: Which statement is FALSE regarding C and Java?



- Arrays in C are just pointers to the 0-th element
- As Java was derived from C, it has the same control flow constructs
- Like Java, in C you can check the length of an array ( `a.length` gives no. elements in `a` )
- C has pointers but Java does not allow you to manipulate pointers or memory addresses of any kind

25

Question: Which statement is FALSE regarding C and Java?



- Arrays in C are just pointers to the 0-th element
- As Java was derived from C, it has the same control flow constructs
- Like Java, in C you can check the length of an array ( `a.length` gives no. elements in `a` )
- C has pointers but Java does not allow you to manipulate pointers or memory addresses of any kind

26

## Pointer Arithmetic

*pointer + number*      *pointer - number*

E.g., *pointer + 1* adds 1 something to a pointer

```
char *p;
char a;
char b;

p = &a;
p += 1;
```

```
int *p;
int a;
int b;

p = &a;
p += 1;
```

In each, p now points to b  
(Assuming compiler doesn't  
reorder variables in memory)

Adds  $1 * \text{sizeof}(\text{char})$   
to the memory address

Adds  $1 * \text{sizeof}(\text{int})$   
to the memory address

*Pointer arithmetic should be used cautiously*

9/10/13

Fall 2013 -- Lecture #4

27

## Arrays and Pointers

Passing arrays:

- Array  $\approx$  pointer to the initial (0th) array element

$a[i] \equiv *(a+i)$

- An array is passed to a function as a pointer
  - The array size is lost!
- Usually bad style to interchange arrays and pointers
  - Avoid pointer arithmetic!

*Really int \*array*      *Must explicitly pass the size*

```
int
foo(int array[],
 unsigned int size)
{
 ... array[size - 1] ...
}

int
main(void)
{
 int a[10], b[5];
 ... foo(a, 10) ... foo(b, 5) ...
}
```

9/10/13

Fall 2013 -- Lecture #4

28

## Arrays and Pointers

```

int
foo(int array[],
 unsigned int size)
{
 ...
 printf("%d\n", sizeof(array));
}

int
main(void)
{
 int a[10], b[5];
 ... foo(a, 10)... foo(b, 5) ...
 printf("%d\n", sizeof(a));
}

```

What does this print? **8**

... because **array** is really a pointer (and a pointer is architecture dependent, but likely to be 8 on modern machines!)

What does this print? **40**

## Arrays and Pointers

```

int i;
int array[10];

for (i = 0; i < 10; i++)
{
 array[i] = ...;
}

```

```

int *p;
int array[10];

for (p = array; p < &array[10]; p++)
{
 *p = ...;
}

```

These code sequences have the same effect!

## Agenda

- Arrays
- **Administrivia**
- Pointer arithmetic
- Arrays vs. pointers
- Technology Break
- Pointer Problems
- Criticisms of C
- And in Conclusion, ...

9/10/13

Fall 2013 -- Lecture #4

31

## Administrivia

- CS61c is relentless!
  - Lab #2, HW #2 posted
  - HW #2 due Sunday before midnight
- Midterm rooms determined!
  - 1 Pimental, 10 Evans, 155 Dwinelle

9/10/13

Fall 2013 -- Lecture #4

32



SEPTEMBER 9, 2013, 10:00 AM | 10 Comments

## The Cloud Era Begins for Enterprise Tech

By QUENTIN HARDY



Peter DaSilva/The New York Times; Joe Kiamar/Agence France-Presse — Getty Images; Justin Sullivan/Getty Images

**Chief Executives** Clockwise from top left: Marc Benioff of Salesforce, Aneel Bhusri of Workday, Jeffrey P. Bezos of Amazon and Steven A. Ballmer of Microsoft.

Let's say it: Last summer was the beginning of the end for the old guard in what is still the biggest part of technology – business spending on everything from servers to software. This fall begins a new competition for the hearts and minds of corporate customers.

9/10/13

FACEBOOK

TWITTER

GOOGLE+

33

## Agenda

- Pointers and Arrays
- Administrivia
- Pointer arithmetic
- Arrays vs. pointers
- Technology Break
- Pointer Problems
- Criticisms of C
- And in Conclusion ...

## Pointer Arithmetic (1/2)

- Since a pointer is just a memory address, we can add to it to step through an array
- `p+1` correctly computes a ptr to the next array element automatically depending on `sizeof(type)`
- `*p++` vs. `(*p)++`?
  - `x = *p++`  $\Rightarrow$  `x = *p; p = p + 1;`
  - `x = (*p)++`  $\Rightarrow$  `x = *p; *p = *p + 1;`
  - This is a C syntax/semantics thing*
- What if we have an array of large structs (objects)?
  - C takes care of it in the same way it handles arrays

9/10/13

Fall 2013 – Lecture #4

35

## Pointer Arithmetic (2/2)


- Every addition or subtraction to a pointer steps the number of bytes of thing it is declared to point to
  - This is why type-casting can get you into trouble
  - 1 byte for a char, 4 bytes for an int, etc.
- Following are equivalent:
 

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

9/10/13

Fall 2013 – Lecture #4

36



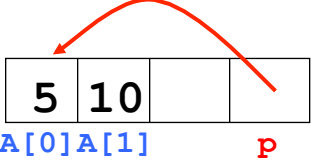
If the first printf outputs 100 5 5 10,  
what will the next two printf output?

```

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]);
}


```

- 101 10 5 10  
101 11 5 11
- 104 10 5 10  
104 11 5 11
- 101 <other> 5 10  
101 <3-others>
- Error message



A[0]A[1]      p

37



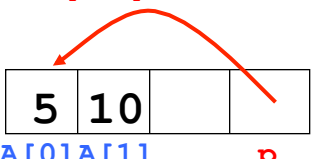
If the first printf outputs 100 5 5 10,  
what will the next two printf output?

```

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]);
}

```

- 101 10 5 10  
101 11 5 11
- 104 10 5 10  
104 11 5 11
- 101 <other> 5 10  
101 <3-others>
- Error message



A[0]A[1]      p

38

## Pointers & Allocation (1/2)

- After declaring a pointer:
  - `int *ptr;`
- `ptr` doesn't actually point to anything yet (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 ...

9/10/13

Fall 2013 -- Lecture #4

39

## Pointers & Allocation (2/2)

- Pointing to something that already exists:
  - `int *ptr, var1, var2; var1 = 5;`  
`ptr = &var1; var2 = *ptr;`
- `var1` and `var2` have space implicitly allocated for them



9/10/13

Fall 2013 -- Lecture #4

40

## 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., will not cause a bus error or address error

9/10/13

Fall 2013 -- Lecture #4

41

## Pointer Arithmetic

- 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

9/10/13

Fall 2013 -- Lecture #4

42

## 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

9/10/13

Fall 2013 -- Lecture #4

43

## Arrays vs. Pointers

- Array name is a read-only pointer to the 0th element of the array
- 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])

9/10/13

Fall 2013 -- Lecture #4

44

## Pointer Arithmetic And in Conclusion ...

- $x = *(p+1)$  ?  
 $\Rightarrow x = *(p+1);$
- $x = *p+1$  ?  
 $\Rightarrow x = (*p) + 1 ;$
- $x = (*p)++$  ?  
 $\Rightarrow x = *p ; *p = *p + 1;$
- $x = *p++$  ?  $(*p++)$  ?  $*(p)++$  ?  $*(p++)$  ?  
 $\Rightarrow x = *p ; p = p + 1;$
- $x = *++p$  ?  
 $\Rightarrow p = p + 1 ; x = *p ;$
- Lesson?
  - Using anything but the standard  $*p++$ ,  $(*p)++$  causes more problems than it solves!

9/10/13

Fall 2013 -- Lecture #4

45


Which one of the pointer arithmetic operations is INVALID?



- Pointer + pointer
- Pointer – integer
- Integer + pointer
- Pointer – pointer

46


Which one of the pointer arithmetic operations is INVALID?



- Pointer + pointer
- Pointer – integer
- Integer + pointer
- Pointer – pointer

47

Which one of the pointer comparisons is INVALID?



- Compare pointer to pointer
- Compare pointer to integer
- Compare pointer to 0
- Compare pointer to NULL

48



Which one of the pointer comparisons is INVALID?



- Compare pointer to pointer
- Compare pointer to integer
- Compare pointer to 0
- Compare pointer to NULL

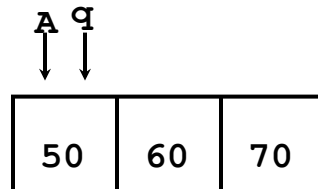
49

## Pointers and Functions (1/2)

- What if the thing you want changed is a *pointer*?
- What gets printed?

```
void IncrementPtr(int *p) *q = 50
{ p = p + 1; }

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



9/10/13

Fall 2013 – Lecture #4

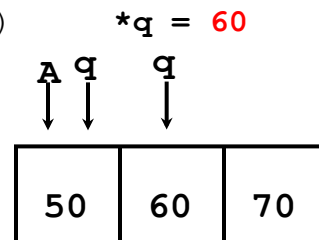
50

## Pointers and Functions (2/2)

- 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);
```



9/10/13

Fall 2013 -- Lecture #4

51

## C String Standard Functions

### #include <string.h>

- `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 contents of string `src` to the memory at `dst`. Caller must ensure that `dst` has enough memory to hold the data to be copied
  - Note: `dst = src` only copies pointers, not string itself

9/10/13

Fall 2013 -- Lecture #4

52

## Agenda

- Pointers and Arrays
- Pointer arithmetic
- Administrivia
- Arrays vs. pointers
- **Technology Break**
- Pointer Problems
- Criticisms of C
- And in Conclusion, ...

9/10/13

Fall 2013 -- Lecture #4

53

## Agenda

- Arrays
- Pointer arithmetic
- Administrivia
- Arrays vs. pointers
- **Technology Break**
- **Pointer Problems**
- Criticisms of C
- And in Conclusion, ...

9/10/13

Fall 2013 -- Lecture #4

54

## 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.

9/10/13

Fall 2013 -- Lecture #4

55

## C String Problems

- 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!

9/10/13

Fall 2013 -- Lecture #4

56

## Criticisms of C - Syntax

- K&R: *C, like any other language, has its blemishes. Some of the operators have the wrong precedence; some parts of the syntax could be better.*
- Precedence: `==` binds more tightly than `&`, |
  - `x & 1 == 0` means `x & (1 == 0)`  
vs. `(x & 1) == 0`
- 15 levels of precedence for 45 operators
  - K&R p. 53
  - Therefore use `()`

9/10/13

Fall 2013 – Lecture #4

57

## Criticisms of C - Syntax

- Difference between assignment and equality
  - `a = b` is assignment
  - `a == b` is an equality test
- One of the most common errors for beginning C programmers!
  - One pattern (when comparing with constant) is to put the var on the right!  
If you happen to use `=`, it won't compile!
    - `if (3 == a) { ...`

9/10/13

Fall 2013 – Lecture #4

58

## Criticisms of C - Syntax

- Syntax: confusion about = and ==
  - `if (a=b)` is true if `a ≠ 0` after assignment
- Syntax: `*p++` means get value at address pointed to by `p`, then increment `p` to point to next data item
- `*--p` means decrement `p` to point to the previous data item and that value

9/10/13

Fall 2013 -- Lecture #4

59

## Criticisms of C - Syntax

- Case statement (`switch`) requires proper placement of `break` to work properly
  - Will do all cases until sees a `break`

```
switch(ch) {
 case '+': ... /* does + and - */
 case '-': ... break;
 case '*': ... break;
 default: ...
}
```

9/10/13

Fall 2013 -- Lecture #4

60

## Criticisms of C – Type casting

- Type casting - pretend that a variable declared in one type is actually of another type

```
int x, y, *p; ...
```

```
y = *p; /* legal */
```

```
y = *x; /* illegal */
```

```
y = *((int *)x); /* legal! */
```

## Criticisms of C - Functionality

- No runtime checking of array bounds

## And in Conclusion, ...

- Pointers are aliases to variables
- Pointers can be used to index into arrays
- Strings are (null terminated) arrays of characters
- Pointers are the source of many bugs in C, so handle with care
- C, like all languages, has flaws but its small and useful language for some tasks