CS 61C: Great Ideas in Computer Architecture (Machine Structures)
Introduction to C (Part II)

Instructors:
Randy H. Katz
David A. Patterson

http://inst.eecs.Berkeley.edu/~cs61c/sp11

Levels of Representation/Interpretation

High Level Language Program (e.g., C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

We are here!

Compiler

Assembly Language Program (e.g., MIPS)

```
lw   $t0, 0($2)
lw   $t1, 4($2)
sw   $t1, 0($2)
sw   $t0, 4($2)
```

Assembler

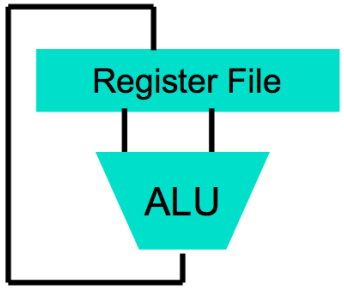
Machine Language Program (MIPS)

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

Anything can be represented as a *number*, i.e., data or instructions

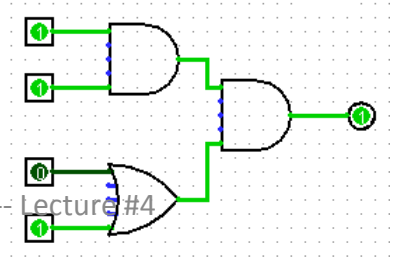
Machine Interpretation

Hardware Architecture Description (e.g., block diagrams)



Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)



Agenda

- C and Pointer review
- Arrays
- Administrivia
- Pointer Arithmetic
- Break
- Pointer Misc
- Pointer Problems
- Summary

Peer Instruction Question 1

```
int *p;
*p = 5;
printf("%d\n",*p);
```

What is the result from executing this code?

a) Prints 5
b) Prints garbage
c) Always crashes.
d) Almost always crashes.

Peer Instruction Answer 1

```
int *p;
*p = 5;
printf("%d\n",*p);
```

What is the result from executing this code?

a) Prints 5
b) Prints garbage
c) Always crashes.
d) Almost always crashes.

- Uninitialized variables contain garbage, p may point to an invalid memory address.
- Trying to write to an invalid address will crash program (segmentation fault).
- There's a (very) small chance p may just happen to contain a valid address.

Peer Instruction Question 2

```
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)}
```

How many syntax + logic errors in this C code?

#Errors
a: 1
b: 2
c: 3
d: 4
e: >4

Peer Instruction Answer 2

```
#insert <stdio.h>
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);}
```

How many syntax + logic errors in this C code?

#Errors
a: 1
b: 2
c: 3
d: 4
e: >4 (6!)

Pointers

- Pointers are used to point to any kind of data (`int`, `char`, a `struct`, etc.)
- Normally a pointer only points 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 other bad things!

Pointers and Structures

```
typedef struct {
    int x;
    int y;
} Point;

Point p1;
Point p2;
Point* paddr;
```

```
/* dot notation */
int h = p1.x;
p2.y = p1.y;

/* arrow notation
   these lines are equivalent */
int h = (*paddr).x;
int h = paddr->x;
```

Pointers Summary

- Why use pointers?
  - If we want to pass a large 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 C, so be careful anytime you deal with them
    - Most problematic with dynamic memory management—which you will to know by the end of the semester, but not for the projects (there will be a lab later in the semester)
    - *Dangling references* and *memory leaks*

Agenda

- C and Pointer review
- Arrays
- Administrivia
- Pointer Arithmetic
- Break
- Pointer Misc
- Pointer Problems
- Summary

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 numth 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
- *Key Concept*: Array variable is a "pointer" to the first (0th) element

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
- An array variable is read-only
  - Cannot say "`ar = [anything]`"
- 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

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)

Array Summary

- 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;`

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;
}
```

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

Agenda

- C and Pointer review
- Arrays
- Administrivia
- Pointer Arithmetic
- Break
- Pointer Misc
- Pointer Problems
- Summary

Administrivia

- Give your consent forms to your TA in discussion or lab.
- HW 1 has been posted – get started if you haven't already.
- Lab 2 has been posted.

Agenda

- C and Pointer review
- Arrays
- Administrivia
- Pointer Arithmetic
- Break
- Pointer Misc
- Pointer Problems
- Summary

Arrays as Arguments

- An array is passed to a function as a pointer
  - The array size is lost!

Passing arrays:

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) …
}
```

Arrays as Arguments

```
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**

Pointer Arithmetic

pointer + number, pointer – number

E.g., *pointer* + `1` adds 1 <u>something</u> to a pointer

(in both examples, pretend variable **a** sits at address 100)

```
char   *p;
char    a;

p = &a;
```

```
int   *p;
int    a;

p = &a;
```

What does the following line yield?

```
printf("%u %u\n",p,p+1);
```

100 101

Adds `1*sizeof(char)` to the memory address

100 104

Adds `1*sizeof(int)` to the memory address

Pointer arithmetic should be used <u>cautiously</u>

Pointer Arithmetic

- 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)`
- What if we have an array of large structs (objects)?
  - C takes care of it in the same way it handles basic types.

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

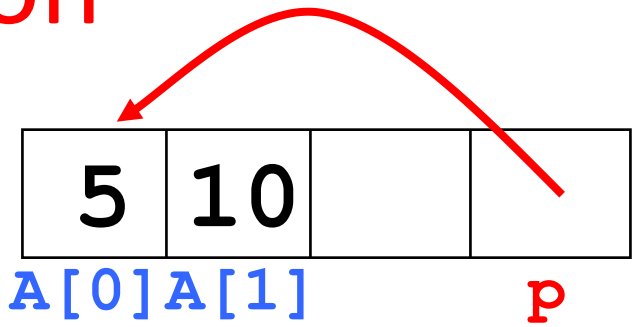
Pointer Arithmetic to Copy Memory

- 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`

Peer Instruction



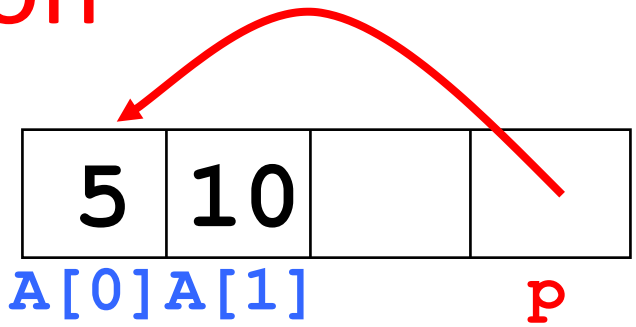

```
int main(void){
    int A[] = {5,10};
    int *p = A;

    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 next 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 printfs causes an ERROR

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 next 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 printfs causes an ERROR

Pointer Arithmetic: Peer Instruction Question

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 Answer

How many of the following are invalid?

I.	pointer + integer	ptr + 1
II.	integer + pointer	1 + ptr
III.	pointer + pointer	ptr + ptr
IV.	pointer – integer	ptr - 1
V.	integer – pointer	1 - ptr
VI.	pointer – pointer	ptr - ptr
VII.	compare pointer to pointer	ptr1 == ptr2
VIII.	compare pointer to integer	ptr == 1
IX.	compare pointer to 0	ptr == NULL
X.	compare pointer to NULL	ptr == NULL

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

Agenda

- C and Pointer review
- Arrays
- Administrivia
- Pointer Arithmetic
- Break
- Pointer Misc
- Pointer Problems
- Summary

Agenda

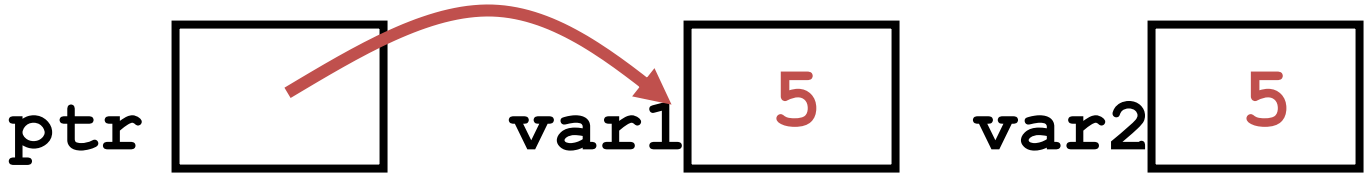
- C and Pointer review
- Arrays
- Administrivia
- Pointer Arithmetic
- Break
- Pointer Misc
- Pointer Problems
- Summary

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 …

Pointers & Allocation (2/2)

- Option 1: 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



- Option 2: Allocate room in memory for new thing to point to. See next lecture.

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 an bus error or address error

Pointers and Functions (1/4)

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

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

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

y = 5

Pointers and Functions (2/4)

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

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

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

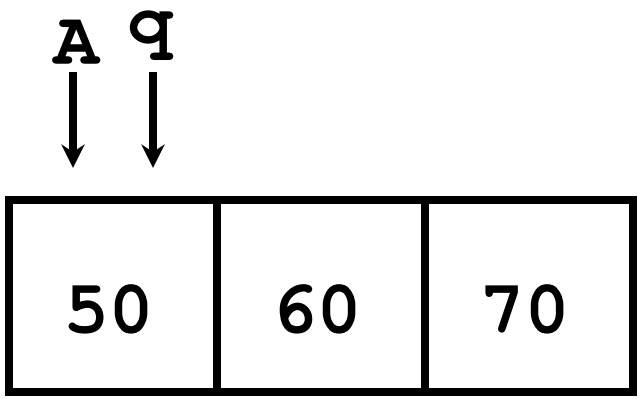
y = 6

Pointers and Functions (3/4)

- But 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);
```

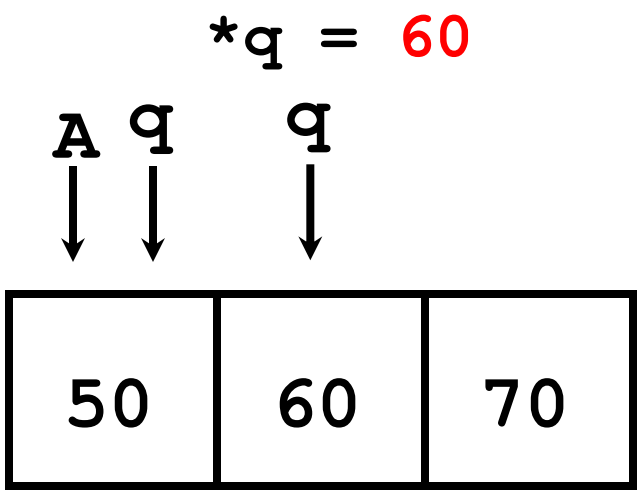


Pointers and Functions (4/4)

- 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);
```



Agenda

- C and Pointer review
- Arrays
- Administrivia
- Pointer Arithmetic
- Break
- Pointer Misc
- Pointer Problems
- Summary

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 */
```

- First pointer declaration is invalid since the types do not match
- Second declaration is valid C but is almost certainly wrong
  - Is it ever correct?

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!

More Common C Errors

- 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) { ...`

And in Conclusion, …

- Pointers and array variables are very similar.
  - Can use pointer or array syntax to index into arrays.
- Strings are (null terminated) arrays of characters
- Adding 1 to a pointer moves the pointer by the size of the thing it's pointing to.
- Pointers are the source of many bugs in C, so handle with care