CS 61C: Great Ideas in Computer Architecture

C Arrays, Strings, More Pointers

Instructor: Justin Hsia
Review of Last Lecture

• C Basics
  – Variables, Functions, Flow Control, Types, and Structs
  – Only 0 and NULL evaluate to FALSE

• Pointers hold addresses
  – Address vs. Value
  – Allow for efficient code, but prone to errors

• C functions “pass by value”
  – Passing pointers circumvents this
Struct Clarification

• Structure definition:
  – Does NOT declare a variable
  – Variable type is “struct name”
    ```
    struct name name1, *pn, name_ar[3];
    ```

• Joint struct definition and typedef
  – Don’t need to name struct in this case
    ```
    typedef struct {
        /* fields */
    } name;
    typedef struct nm name;
    ```
Question: What is the result from executing the following code?

```c
#include <stdio.h>
int main() {
    int *p;
    *p = 5;
    printf("%d\n", *p);
}
```

(A) Prints 5  
(B) Prints garbage  
(C) Always crashes  
(D) Almost always crashes
Great Idea #1: Levels of Representation/Interpretation

- **Higher-Level Language Program (e.g. C)**
  - Compiler
  - **Assembly Language Program (e.g. MIPS)**
    - Assembler
    - **Machine Language Program (MIPS)**
      - Machine Interpretation
      - **Hardware Architecture Description (e.g. block diagrams)**
      - **Architecture Implementation**
      - **Logic Circuit Description (Circuit Schematic Diagrams)**

- **temp = v[k];**
  - **v[k] = v[k+1];**
  - **v[k+1] = temp;**

- **lw $t0, 0($2)**
- **lw $t1, 4($2)**
- **sw $t1, 0($2)**
- **sw $t0, 4($2)**

- We are here

- Logic Circuit Description (Circuit Schematic Diagrams)

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

- Machine Interpretation

- Architecture Implementation

- **Register File**
  - ALU

- 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
Agenda

- Miscellaneous C Syntax
- Arrays
- Administrivia
- Strings
- More Pointers
  - Pointer Arithmetic
  - Pointer Misc
Assignment and Equality

• One of the most common errors for beginning C programmers
  
  \[ a = b \quad \text{is assignment} \]
  
  \[ a == b \quad \text{is equality test} \]

• When comparing with a constant, can avoid this by putting the variable on the right!
  
  \[ \begin{array}{l}
  \text{– if (3 == a)} \{ \ldots \} \quad \text{Correct} \\
  \text{– if (3 = a)} \quad \{ \ldots \} \quad \text{Won’t compile}
  \end{array} \]

• Comparisons use assigned value
  
  \[ \text{– if (a=b) is true if } a \neq 0 \text{ after assignment (} b \neq 0) \]
## Operator Precedence

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>() ][ -&gt; .</td>
<td>left to right</td>
</tr>
<tr>
<td>! ~ ++ -- + - * (type) sizeof</td>
<td>right to left</td>
</tr>
<tr>
<td>* / &amp;</td>
<td>left to right</td>
</tr>
<tr>
<td>+ -</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt;</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>left to right</td>
</tr>
<tr>
<td>== !=</td>
<td>left to right</td>
</tr>
<tr>
<td>&amp;</td>
<td>left to right</td>
</tr>
<tr>
<td>^</td>
<td>left to right</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>left to right</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>? :</td>
<td>right to left</td>
</tr>
<tr>
<td>= += -= *= /= %= &amp;= ^=</td>
<td>right to left</td>
</tr>
<tr>
<td>,</td>
<td>left to right</td>
</tr>
</tbody>
</table>
Operator Precedence

For precedence/order of execution, see Table 2-1 on p. 53 of K&R

• Use parentheses to manipulate

• Equality test (==) binds more tightly than logic (&, |, &&, ||)
  
  – \( x \& 1 == 0 \) means \( x \& (1 == 0) \) instead of \( (x \& 1) == 0 \)

• Pre-increment (++p) takes effect immediately

• Post-increment (p++) takes effect last
Increment and Dereference

- Dereference operator (*) and in/decrement operators are the same level of precedence and are applied from **right to left**

  *\texttt{p++} returns \texttt{*p}, then increments \texttt{p}  
  - ++ binds to \texttt{p} before *, but takes effect last

  *\texttt{--p} decrements \texttt{p}, returns val at that addr  
  - -- binds to \texttt{p} before * and takes effect first

  \texttt{++*p} increments \texttt{*p} and returns that val  
  - * binds first (get val), then increment immediately

  (*\texttt{p})-- returns \texttt{*p}, then decrements in mem  
  - Post-decrement happens last
Question: What is the output of the following code?

```c
char blocks[] = {'6', '1', 'C'};
char *ptr = blocks, temp;
temp = *++ptr;
printf("1:  %c\n", temp);
temp = *ptr++;
printf("2:  %c\n", temp);
```

(A) 7 8
(B) 7 1
(C) 1 1
(D) 1 C
Agenda

• Miscellaneous C Syntax

• Arrays

• Administrivia

• Strings

• More Pointers
  – Pointer Arithmetic
  – Pointer Misc
Array Basics

• **Declaration:**

```c
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:**

```c
ar[num] returns the num\textsuperscript{th} element
```

– Zero-indexed
Arrays Basics

• **Pitfall:** An array in C does not know its own length, and its bounds are not checked!
  – We can accidentally access off the end of an array
  – We must pass the array **and its size** to any procedure that is going to manipulate it

• Mistakes with array bounds cause *segmentation faults* and *bus errors*
  – Be careful! These are VERY difficult to find
  (You’ll learn how to debug these in lab)
Accessing an Array

- Array size $n$: access entries 0 to $n-1$
- Use separate variable for declaration & bound

Bad Pattern

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

Better Pattern

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

Single source of truth!
Arrays and Pointers

• Arrays are (almost) identical to pointers
  – char *string and char string[] are nearly identical declarations
  – Differ in subtle ways: initialization, sizeof(), etc.

• Key Concept: An array variable looks like a pointer to the first (0th) element
  – ar[0] same as *ar; ar[2] same as *(ar+2)
  – We can use pointer arithmetic to conveniently access arrays

• An array variable is read-only (no assignment)
  (i.e. cannot use “ar = <anything>”)

6/26/2013 Summer 2013 -- Lecture #3
Array and Pointer Example

- $ar[i]$ is treated as $(ar+i)$
- To zero an array, the following three ways are equivalent:
  1) for ($i=0; i<\text{SIZE}; i++$) $ar[i] = 0$
  2) for ($i=0; i<\text{SIZE}; i++$) $(ar+i) = 0$
  3) for ($p=ar; p<ar+\text{SIZE}; p++$) $*p = 0$
- These use *pointer arithmetic*, which we will get to shortly
Arrays Stored Differently Than Pointers

```c
void foo() {
    int *p, a[4], x;
    p = &x;

    *p = 1; // or p[0]
    printf("*p:%u, p:%u, &p:%u\n", *p, p, &p);
    *a = 2; // or a[0]
    printf("*a:%u, a:%u, &a:%u\n", *a, a, &a);
}
```

K&R: “An array name is not a variable”
Arrays and Functions

• Declared arrays only allocated while the scope is valid:

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

• An array is passed to a function as a pointer:

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

Must explicitly pass the size!
Arrays and Functions

• Array size gets lost when passed to a function
• What prints in the following code:

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

- `sizeof(int *)` is printed 10 times
- `10 * sizeof(int)` is printed once
Agenda

• Miscellaneous C Syntax
• Arrays
• Administrivia
• Strings
• More Pointers
  – Pointer Arithmetic
  – Pointer Misc
Administrivia

• Lab 2 tomorrow
• HW1 due Sunday night
• Lab 3 (big lab) next Tue (July 2)
• HW2 released Fri, due next Wed (July 3)
• Suggested plan of attack:
  — Finish HW1 by Sat night
  — Do 1\textsuperscript{st} half of Lab 3 Sun, start HW2
  — Do 2\textsuperscript{nd} half of Lab 3 Tue, finish HW2 by Wed
Agenda

• Miscellaneous C Syntax
• Arrays
• Administrivia
• Strings
• More Pointers
  – Pointer Arithmetic
  – Pointer Misc
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 (\0) (a.k.a. “null terminator”)

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

    Array size here is 4

    This means you need an extra space in your array!!!
C String Standard Functions

• Accessible with `#include <string.h>`
• `int strlen(char *string);`
  – Returns the length of string (not including null term)
• `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 pointer (the address)
#include <stdio.h>
#include <string.h>
int main () {
    char s1[10], s2[10], s3[]="hello", *s4="hola";
    strcpy(s1,"hi");  strcpy(s2,"hi");
}

Value of the following expressions?

`sizeof(s1)`  `strcmp(s1,s2)`

`strlen(s1)`  `strcmp(s1,s3)`  `(s1 > s3)`

`s1==s2`  `strcmp(s1,s4)`  `(s1 < s4)`
Question: What does this function do when called?

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

(A) Always throws an error
(B) Changes characters in string \( t \) to the next character in the string \( s \)
(C) Copies a string at address \( t \) to the string at address \( s \)
(D) Appends the string at address \( t \) to the end of the string at address \( s \)
Agenda

• Miscellaneous C Syntax
• Arrays
• Administrivia
• Strings
• More Pointers
  – Pointer Arithmetic
  – Pointer Misc
Pointer Arithmetic

- *pointer ± number*
  - e.g. `pointer + 1` adds 1 something to the address

- **Compare what happens:** (assume `a` at address 100)

```c
char *p; char a;
int *p; int a;
p = &a;
printf(“%u %u\n”, p, p+1);
```

- **Pointer arithmetic should be used cautiously**
Pointer Arithmetic

• A pointer is just a memory address, so we can add to/subtract from it to move through an array

• \( p + 1 \) correctly increments \( p \) by \( \text{sizeof}(*p) \)
  – i.e. moves pointer to the next array element

• What about an array of large structs (objects)?
  – Struct declaration tells C the size to use, so handled like 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:

```c
void copy(int *from, int *to, int n)
{
    int i;
    for (i=0; i<n; i++) {
        *to++ = *from++;
    }
}
```

• We have to pass the size \(n\) to `copy`
Question: The first `printf` outputs 100 5 5 10. What will the next two `printf` output?

```c
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]);
}
```

(A) 101 10 5 10 then 101 11 5 11
(B) 104 10 5 10 then 104 11 5 11
(C) 100 6 6 10 then 101 6 6 10
(D) 100 6 6 10 then 104 6 6 10
Get To Know Your Staff

- Category: Cal
Agenda

- Miscellaneous C Syntax
- Arrays
- Administrivia
- Strings
- **More Pointers**
  - Pointer Arithmetic
  - Pointer Misc
Pointers and Allocation

• When you declare a pointer (e.g. `int *ptr;`), it doesn’t actually point to anything yet
  – It points somewhere (garbage; don’t know where)
  – Dereferencing will usually cause an error

• **Option 1:** Point to something that already exists
  – `int *ptr, var;  var = 5;  ptr = &var1;
  – `var` has space implicitly allocated for it (declaration)

• **Option 2:** Allocate room in memory for new thing to point to (next lecture)
Pointers and Structures

Variable declarations:
struct Point {
    int x;
    int y;
    struct Point *p;
};

struct Point pt1;
struct Point pt2;
struct Point *ptaddr;

Valid operations:
/* dot notation */
int h = pt1.x;
pt2.y = pt1.y;

/* arrow notation */
int h = ptaddr->x;
int h = (*ptaddr).x;

Cannot contain an instance of itself, but can point to one

/* This works too */
pt1 = pt2;

Copies contents
Pointers to Pointers

• **Pointer to a pointer**, declared as **h**

• Example:

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

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

```
50 60 70

A q q

*q = 60
```
Question: Struct and Pointer Practice

Assuming everything is properly initialized, what do the following expressions evaluate to?

```c
struct node {
    char *name;
    struct node *next;
};
struct node *ar[5];
struct node **p = ar;
... /* fill ar with initialized structs */

1) &p
2) p->name
3) p[7]->next
4) *((p + 2))
5) *(p[0]->next)
6) (*p)->next->name
```
Answers: Struct and Pointer Practice

1) \&p  
   address (ptr to ptr to ptr)  
   “address of” operator returns an address

2) p->name  
   invalid  
   Attempt to access field of a pointer

3) p[7]->next  
   invalid  
   Increment p into unknown memory, then dereference

4) **(p + 2)**  
   data (struct node)  
   Move along array, access pointer, then access struct

5) *(p[0]->next)  
   data (struct node)  
   This is tricky. p[0] = *(p + 0) is valid and accesses the array of pointers, where -> operator correctly accesses field of struct, and dereference leaves us at another struct.

6) (*p)->next->name  
   address (char array)  
   next field points to struct, access name field, which is, itself, a pointer (string)
Summary

• Pointers and array variables are very similar
  – Can use pointer or array syntax to index into arrays
• Strings are null-terminated arrays of characters
• Pointer arithmetic 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