Great Ideas in Computer Architecture

# *C Arrays, Strings, More Pointers*

**Instructor:** Steven Ho

# 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 foo"

```
struct foo {
    /* fields */
};
```

```
struct foo name1, *pn, name_ar[3];
```

- Joint struct definition and typedef
  - Don't need to name struct in this case

```
struct foo {
    /* fields */
};
typedef struct foo bar;
bar name1;
```

→

```
typedef struct foo {
    /* fields */
} bar;
bar name1;
```

# sizeof()

- Integer and pointer sizes are machine dependent—how do we tell?
- Use `sizeof()` function
  - Returns size <u>in bytes</u> of variable or data type name

    Examples:

```
int x, *y;
sizeof(x);        // 4  (32-bit int)
sizeof(int);      // 4  (32-bit int)
sizeof(y);        // 4  (32-bit addr)
sizeof(char);     // 1  (always)
```

# sizeof()

- Acts differently with arrays and structs (to be explained later)
  - Arrays: returns size of whole array
  - Structs: returns size of one instance of struct (sum of sizes of all struct variables + padding)
- eg:

```
typedef struct example {
    char *name;
    int   year;
} Song;
Song song1;
sizeof(song1);   // 4 + 4 = 8
```

**Question:** What is the result from executing the following code?

```
#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)**

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

We are here

↓ *Compiler*

**Assembly Language Program (e.g. RISCV)**

```
lw    x5, 0(x2)
lw    x6, 4(x2)
sw    x6, 0(x2)
sw    x5, 4(x2)
```

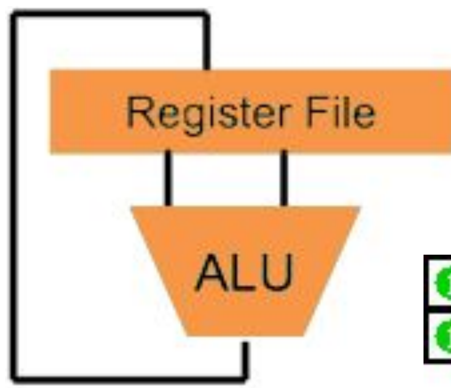
↓ *Assembler*

**Machine Language Program (RISCV)**

```
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
```

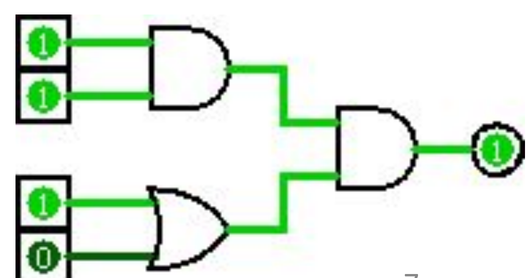
↓ *Machine Interpretation*

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




↓ *Architecture Implementation*

**Logic Circuit Description (Circuit Schematic Diagrams)**



# Agenda

- C Operators
- Arrays
- Administrivia
- Strings
- More Pointers
  - Pointer Arithmetic
  - Pointer Misc

# Operator Precedence

| Precedence | Operator | Description | Associativity |
|---|---|---|---|
| **1** | ++ -- | Suffix/postfix increment and decrement | Left-to-right |
| | () | Function call | |
| | [] | Array subscripting | |
| | . | Structure and union member access | |
| | -> | Structure and union member access through pointer | |
| | (*type*){*list*} | Compound literal(C99) | |
| **2** | ++ -- | Prefix increment and decrement | Right-to-left |
| | + - | Unary plus and minus | |
| | ! ~ | Logical NOT and bitwise NOT | |
| | (*type*) | Type cast | |
| | * | Indirection (dereference) | |
| | & | Address-of | |
| | sizeof | Size-of | |
| | _Alignof | Alignment requirement(C11) | |

# Assignment and Equality

- One of the most common errors for beginning C programmers

  `a = b` is *assignment*

  `a == b` is *equality test*

- Comparisons use assigned value
  - `if (a=b)` is true if $a \neq 0$ after assignment ($b \neq 0$)

# 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`

# Operator Precedence

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

- **Prefix** (++p) takes effect *immediately*
- **Postfix/Suffix** (p++) takes effect *last*

```
int main () {
   int x = 1;
   int y = ++x;  // y = 2, x = 2
   x--;
   int z = x++;  // z = 1, x = 2
   return 0;
}
```

# Agenda

- C Operators
- Arrays
- Administrivia
- Strings
- More Pointers
  - Pointer Arithmetic
  - Pointer Misc

# Array Basics

- **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$^{th}$ 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 array declaration & array bound to be reused (eg: no hard-coding)

Bad Pattern

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

Better Pattern

```
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 ($0^{th}$) 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>`")

# 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<SIZE; i++)   ar[i] = 0;`

2) `for(i=0; i<SIZE; i++)   *(ar+i) = 0;`

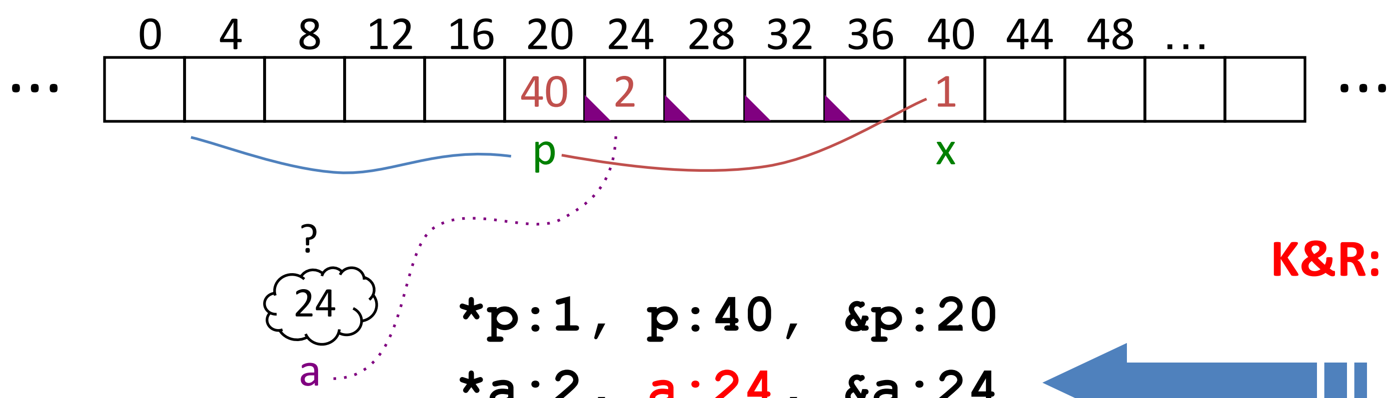
3) `for(p=ar; p<ar+SIZE; p++)   *p = 0;`

- These use *pointer arithmetic*, which we will get to shortly

# Arrays Stored Differently Than Pointers

```
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);
}
```




**`*p:1, p:40, &p:20`**

**`*a:2, a:24, &a:24`**

**K&R: "An array name is not a variable"**

# Arrays and Functions

- Declared arrays only allocated while the scope is valid:

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

**BAD**

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

*Really* `int *ar`

```
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:

```
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 *)**

**10*sizeof(int)**

# Agenda

- C Operators
- Arrays
- Administrivia
- Strings
- More Pointers
  - Pointer Arithmetic
  - Pointer Misc

# Administrivia

- Disc1 today, Lab1 tomorrow
  - To rent iClicker from us, bring check to lab!
- HW0 and mini-bio due Monday
- HW1 will be released Thursday, so be prepared!
- Proj1 (Chatroom) will be released tonight, due June 29

# Agenda

- C Operators
- Arrays
- Administrivia
- Strings
- More Pointers
  - Pointer Arithmetic
  - Pointer Misc

# C Strings

- String in C is just an array of characters

```
char string[] = "abc";
```

Array size here is **4**

  - Last character is followed by a 0 byte (`'\0'`) (a.k.a. "null terminator")

This means you need an extra space in your array!!!

- How do you tell how long a string is?

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

# 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)

# String Examples

```
#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==s2              strcmp(s1,s4)
```

Point to different locations!

(s1 > s3)
e, f, g, h, i

(s1 < s4)
i, j, k, l ,
m, n, o

# Agenda

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

# Pointer Arithmetic

- *pointer* ± *number*
  - e.g. *pointer* + 1 adds 1 <u>something</u> to the address
- Compare what happens: (assume `a` at address 100)

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

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

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

**`100 101`**

**`100 104`**

Adds `1*sizeof(char)`

Adds `1*sizeof(int)`

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

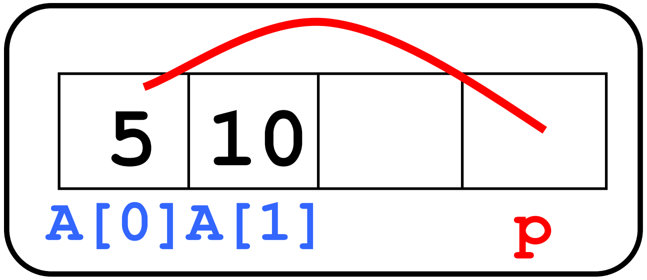
# Pointer Arithmetic

- 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 `sizeof(*p)`
  - i.e. moves pointer to the next array element
- What about an array of structs?
  - 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

**Question:** 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]);
}
```

**(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

# (REVIEW) Operator Precedence

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

- **Prefix** (++p) takes effect *immediately*
- **Postfix/Suffix** (p++) takes effect *last*

```
int main () {
    int x = 1;
    int y = ++x;  // y = 2, x = 2
    x--;
    int z = x++;  // z = 1, x = 2
    return 0;
}
```

# Increment and Dereference

- When multiple prefixal operators are present, they are applied from *right to left*

`*--p` decrements `p`, returns val at that addr

- `--` binds to `p` before `*` and takes effect first

`++*p` increments `*p` and returns that val

- `*` binds first (get val), then increment immediately

# Increment and Dereference

- *Postfixal* in/decrement operators have precedence over prefixal operators (e.g. `*`)
  - BUT the in/decrementation takes effect last because it is a postfix. The “front” of expression is returned.
- `*p++` returns `*p`, then increments `p`
  - `++` binds to `p` before `*`, but takes effect last

Equivalent C code:

```
char *p = "hi"; // assume p has value 40
char c = *p++;  // c = 'h', p = 41
c      = *p;    // c = 'i'
```

# Increment and Dereference

- *Postfixal* in/decrement operators have precedence over prefixal operators (e.g. `*`)
  - BUT the in/decrementation takes effect last because it is a postfix. The "front" of expression is returned.
- `(*p)++` returns `*p`, then increments in mem
  - Post-increment happens last

Equivalent C code:

```
char *p = "bye";   // assume p has value 40
char c  = (*p)++;  // c = 'b', p = 40
c       = *p;      // c = 'c'  because 'b'+1 = 'c'
```

**Question:** What does this function do when called?

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

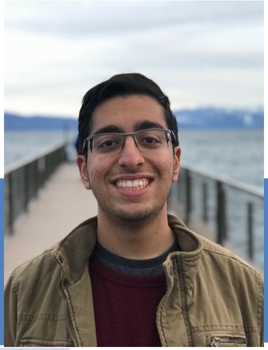
**(A)** Always throws an error

**(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`**

# Get To Know Your Staff

| |  Sukrit |  Suvansh |
|---|---|---|
| **Bribe Food** | Twix | Chocolate without nuts |
| **Biggest Fear** | Darkness | Spiders |
| **Unpopular Opinion** | EDM music is bad | Didn't like Breaking Bad |
| **Café Getaway** | TeaOne Cory | Strada |

# Agenda

- C Operators
- 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;
```

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

Valid operations:

```
/* dot notation */
int h = pt1.x;
pt2.y = pt1.y;

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

/* This works too */
pt1 = pt2;
```

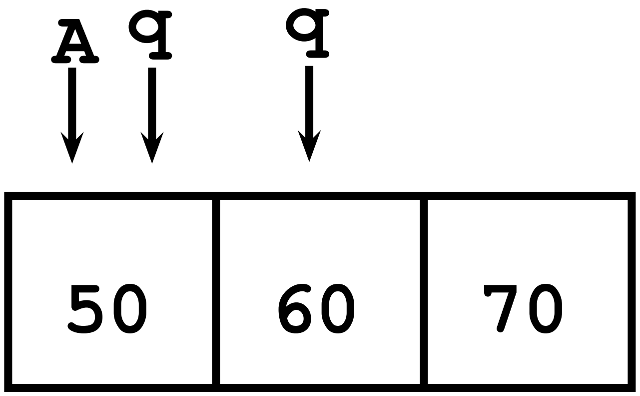
Copies contents

# Pointers to Pointers

- *Pointer to a pointer*, declared as `**h`
- Example:

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

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




***q: 60**

## **Question:** *Struct and Pointer Practice*

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

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

- ☐ address
- ☐ data
- ☐ invalid

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*

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

- ☐ address
- ☐ data
- ☐ invalid

**1) `&p`:**

address (ptr to ptr to ptr)

"address of" operator returns an address

## **Answers:** *Struct and Pointer Practice*

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

- ☐ address
- ☐ data
- ☐ invalid

**`2) p->name:`**

invalid

Attempt to access field of a pointer address

# **Answers:** *Struct and Pointer Practice*

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

- ☐ address
- ☐ data
- ☐ invalid

**3) `p[7]->next`**

invalid

Increment p into unknown memory, then dereference

# **Answers:** *Struct and Pointer Practice*

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

- ☐ address
- ☐ data
- ☐ invalid

**4) `*(*(p + 2))`**

data (struct node)

Move along array, access pointer, then access struct

# **Answers:** *Struct and Pointer Practice*

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

- ☐ address
- ☐ data
- ☐ invalid

**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`.

## **Answers:** *Struct and Pointer Practice*

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

- ☐ address
- ☐ data
- ☐ invalid

**6) `(*p)->next->name`**

address (char array)

`next` field points to struct, access `name` field, which is, itself, a pointer (string)

## **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

# 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++;
    }
}
```

- We have to pass the size(`n`) to `copy`