# CS 61c: Great Ideas in Computer Architecture

## Arrays, Strings, and Some More Pointers

**Instructor:** Alan Christopher

June 24, 2014

# Review of Last Lecture

- C Basics
  - Variables, functioss, control flow, types, structs
  - Only 0 and `NULL` evaluate to false
- Pointers hold addresses
  - Address vs. Value
  - Allows for efficient and powerful code, but error prone
- C functions are "pass by value"
  - Passing pointers circumvents this

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

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

(blue) Prints 5

(green) Prints garbage

(purple) Guaranteed to crash

(yellow) Probably crashes

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

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

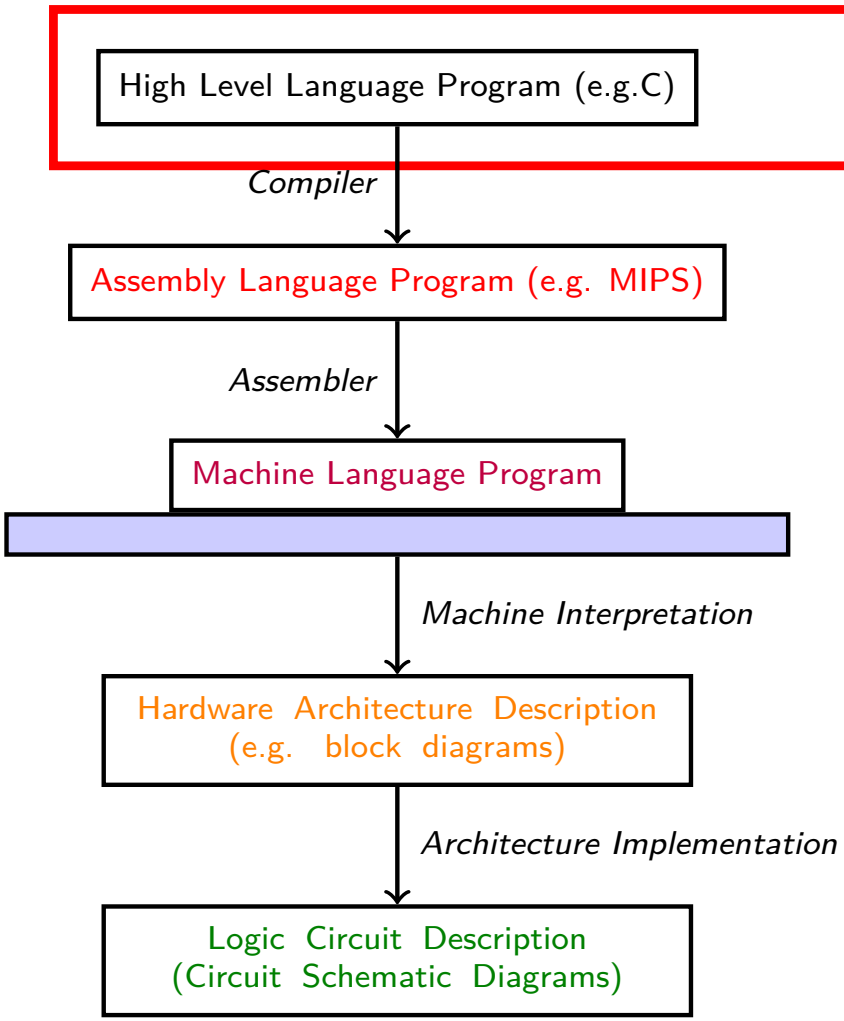
(blue) Prints 5

(green) Prints garbage

(purple) Guaranteed to crash

(yellow) Probably crashes

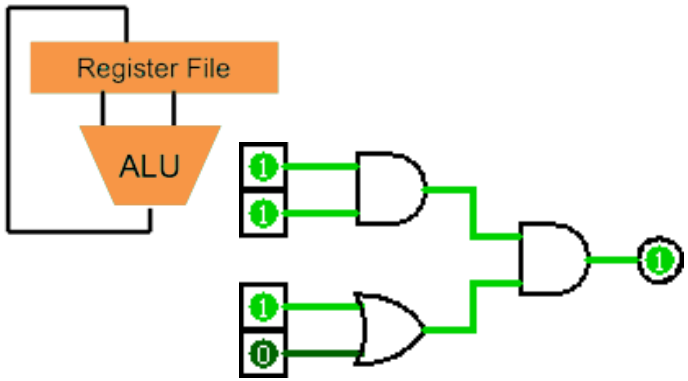
# Great Idea #1: Levels of Representation/Interpretation




```
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)
```

```
1000 1100 0100 1000 0000 0000 0000 0000
1000 1100 0100 1001 0000 0000 0000 0100
1010 1100 0100 1001 0000 0000 0000 0000
1010 1100 0100 1000 0000 0000 0000 0100
```



# Outline

# Expansion on Struct Declarations

- Structure definition:
  - Does NOT declare a variable
  - Variable type is "`struct name`"

```
struct name bob, *pn, name_arr[3];
```

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

- Joint struct definition and `typedef` possible

```
struct nm {
  /* fields */
};
typedef struct nm name;
name n1;
```

Equiv. ⟷

```
typedef struct nm {
  /* fields */
} name;
name n1;
```

# Assignment and Equality

- One of the most common errors for beginning C programmers
  - `(a = b)` is an *assignment*
  - `(a == b)` is an *equality test*
- Comparisons will use assigned values
  - Assignments return the value assigned
  - `if (a = b) { ... }` is legal, but probably not what you meant
- A trick for avoiding this mistake
  - Put the constant on the left when comparing

    `if (3 == a) { ... }` ← Correct

    `if (3 = a) { ... }` ← Compilation Error

# Operator Precedence

| Operators | Associativity |
|---|---|
| `() [] -> .` | left to right |
| `! ~ ++ -- + - * (type) sizeof` | right to left |
| `* / %` | left to right |
| `+ -` | left to right |
| `<< >>` | left to right |
| `< <= > >=` | left to right |
| `== !=` | left to right |
| `&` | left to right |
| `^` | left to right |
| `\|` | left to right |
| `&&` | left to right |
| `\|\|` | left to right |
| `?:` | right to left |
| `= += -= *= %= &= ^= \|= <<= >>=` | right to left |
| `,` | left to right |

# 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)`, rather than `(x & 1) == 0`
- Pre-increment (++`p`) takes effect *first*
- Post-increment (`p`++) takes effect *last*

# Increment and Dereference

- Dereference operator (*) and (in/de)crement operators are the same level of precedence and are applied from *right to left*
  *p++ returns *p, then increments p
  - ++ binds to p before *, but takes effect last

# Increment and Dereference

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

  `*p++` returns `*p`, then increments `p`
  - `++` binds to `p` before `*`, but takes effect last

  `*--p` decrements `p`, returns whatever is at that address
  - `--` binds to `p` before `*`, and takes effect first

# Increment and Dereference

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

  `*p++` returns `*p`, then increments `p`
  - `++` binds to `p` before `*`, but takes effect last

  `*--p` decrements `p`, returns whatever is at that address
  - `--` binds to `p` before `*`, and takes effect first

  `++*p` increments `*p`, then returns that value
  - `*` binds to `++` before `*`

# Increment and Dereference

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

  `*p++` returns `*p`, then increments `p`
  - `++` binds to `p` before `*`, but takes effect last

  `*--p` decrements `p`, returns whatever is at that address
  - `--` binds to `p` before `*`, and takes effect first

  `++*p` increments `*p`, then returns that value
  - `*` binds to `++` before `*`

  `(*p)++` returns `*p`, then increments in memory
  - `*` binds to `p` before `++`, and takes effect first

**Question:** What is the output of the following code?

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

| | 1 | 2 |
|---|---|---|
| blue | 7 | 8 |
| green | 7 | 1 |
| purple | 1 | 1 |
| yellow | 1 | C |

**Question:** What is the output of the following code?

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

| | 1 | 2 |
|---|---|---|
| blue | 7 | 8 |
| green | 7 | 1 |
| purple | 1 | 1 |
| yellow | 1 | C |

# Outline

# Array Syntax

- **Declaration:**
  `int ar[2];` declares a 2-element array of integers
  `int ar[] = {795, 635};` declares and initialized a 2-element integer array
- **Accessing elements:**
  `ar[num]` returns the `num`-th element of `ar`
  - Zero-indexed

# Array Pitfalls

- **Pitfall:** An array in C does not know its own length, and its bounds are not checked!
  - We can accidentally access elements past the end of an array
    - Not even guaranteed to fail when that happens!
  - We must pass the array *and its size* (or use sentinel values, more on that later) to any procedure manipulating it.
  - Mistakes with array bounds manifest as *segmentation faults* and *bus errors*
    - Very difficult to find, best to be careful when coding to avoid them as much as possible.

# Accessing Arrays

- Array size `n`: can access entries in the range [0,`n`−1]
- Use a variable or constant for declaration of length

```
/* Blegh, magic numbers! */
int i, arr[10];
for (i = 0; i < 10; i ++) { ... }
```

# Accessing Arrays

- Array size `n`: can access entries in the range [0,`n`-1]
- Use a variable or constant for declaration of length

```
/* Blegh, magic numbers! */
int i, arr[10];
for (i = 0; i < 10; i ++) { ... }

/* Single source of truth. Much better. */
int ARRAY_SIZE = 10;
int i, arr[ARRAY_SIZE];
for (i = 0; i < ARRAY_SIZE; i ++) { ... }
```

# 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 0-th element
  - `ar[0]` same as `*ar` and `ar[2]` same as `*(ar + 2)`
  - We can use pointer arithmetic to conveniently access arrays
- An array variable is read-only (no assignment)
  - cannot use `ar = anything`

# Array and Pointer Example

- Remember: `ar[i]` is treated as `*(ar + i)`
- Three different ways of zeroing an array
  1. `for (i = 0; i < SIZE; i++) ar[i] = 0;`

# Array and Pointer Example

- Remember: `ar[i]` is treated as `*(ar + i)`
- Three different ways of zeroing an array
  1. `for (i = 0; i < SIZE; i++) ar[i] = 0;`
  2. `for (i = 0; i < SIZE; i++) *(ar + i) = 0;`

# Array and Pointer Example

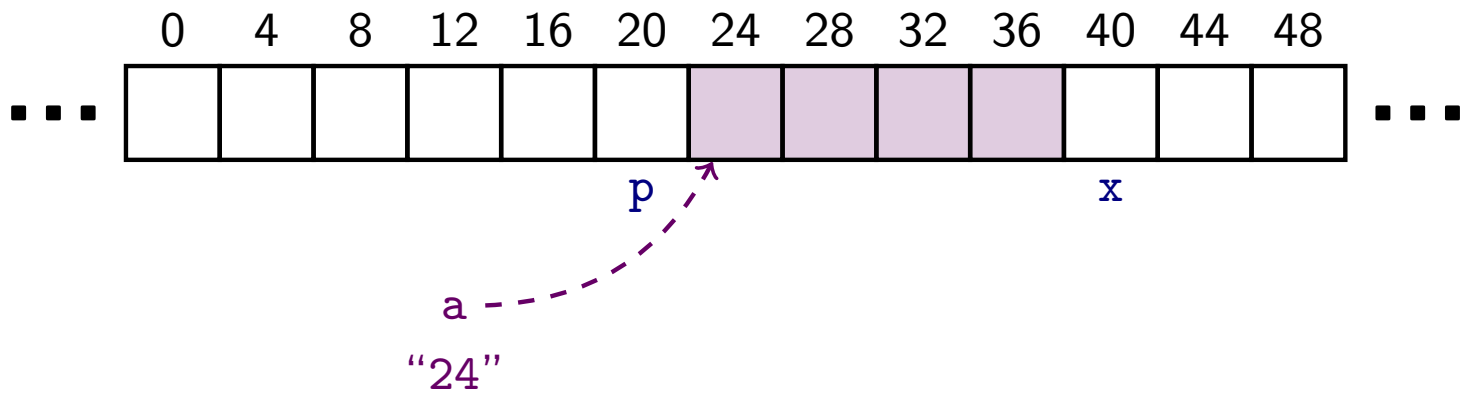
- Remember: `ar[i]` is treated as `*(ar + i)`
- Three different ways of zeroing an array
  1. `for (i = 0; i < SIZE; i++) ar[i] = 0;`
  2. `for (i = 0; i < SIZE; i++) *(ar + i) = 0;`
  3. `for (p = ar; p < ar + SIZE; p++) *p = 0;`

# Array and Pointer Example

- Remember: `ar[i]` is treated as `*(ar + i)`
- Three different ways of zeroing an array
  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'll cover in more detail shortly

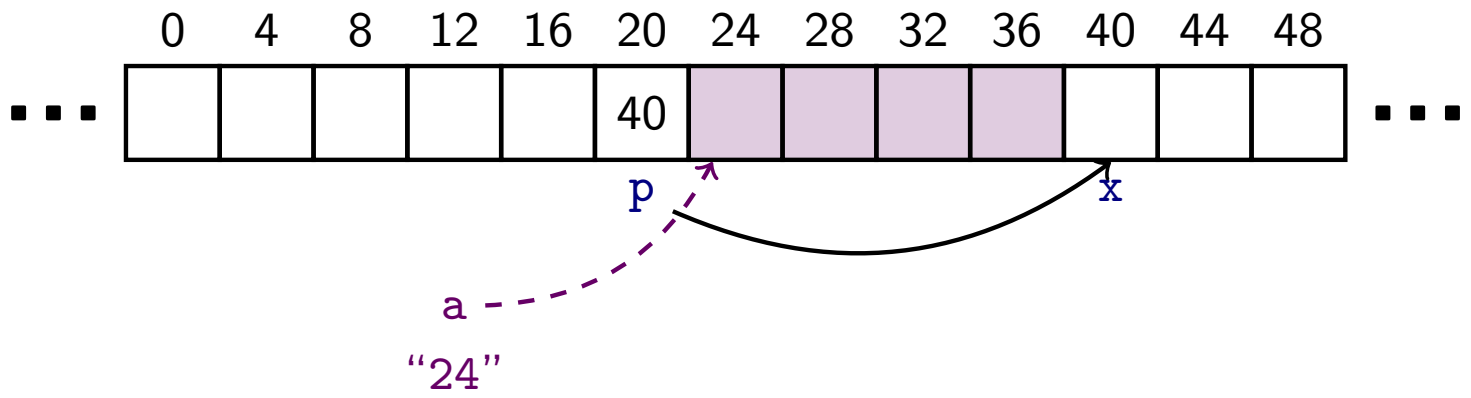
# Arrays Stored Differently Than Pointers

```
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);
}
```



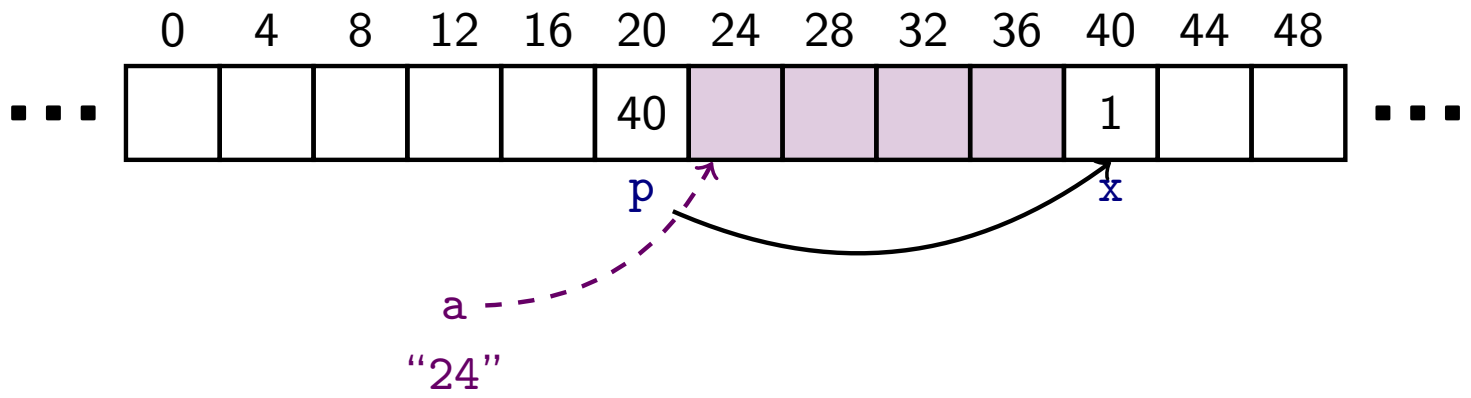
# Arrays Stored Differently Than Pointers

```
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);
}
```



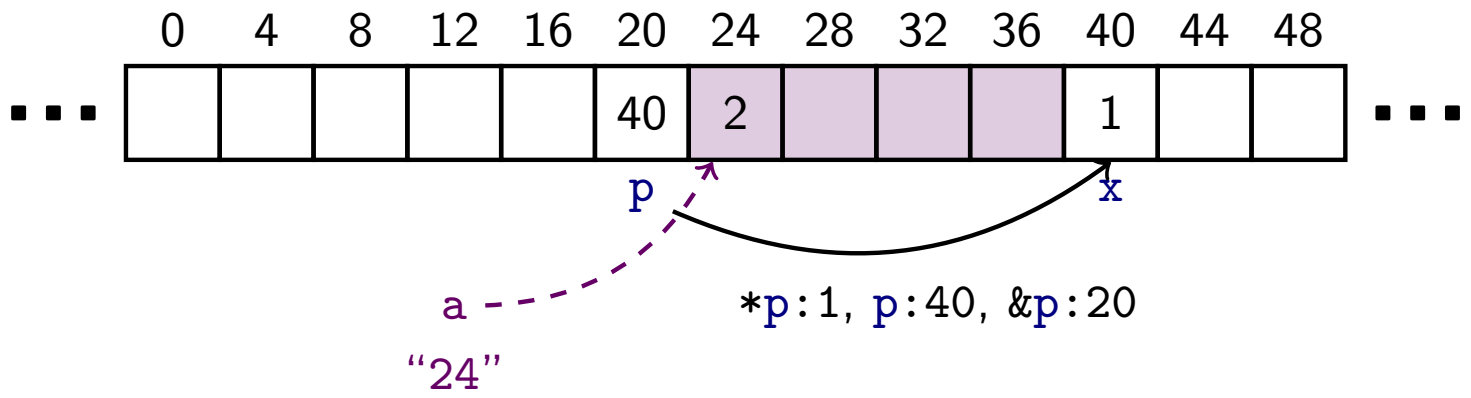
# Arrays Stored Differently Than Pointers

```
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);
}
```



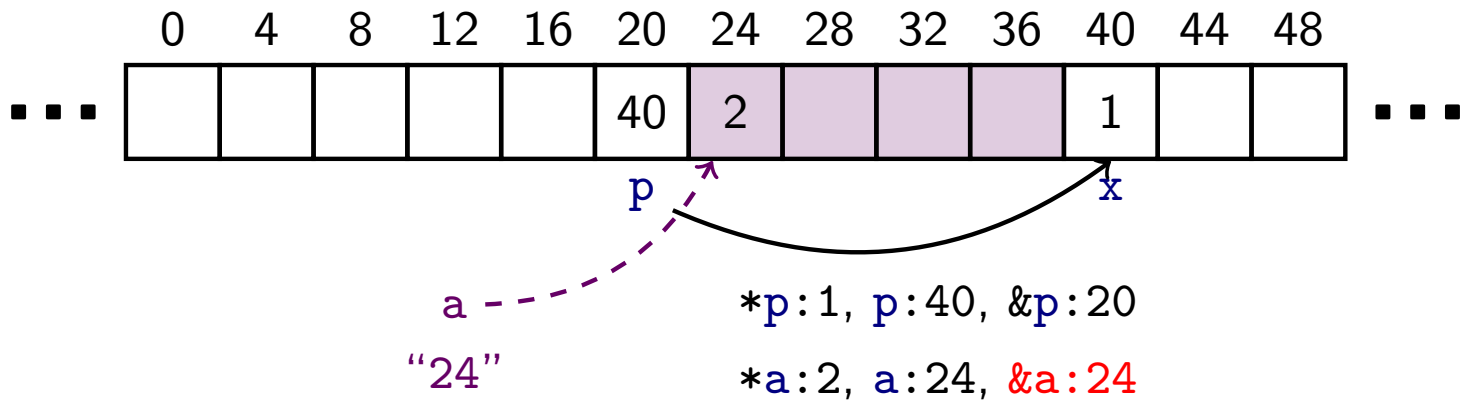
# Arrays Stored Differently Than Pointers

```
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);
}
```



# 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);
}
```



# Arrays and Functions

- Declared arrays only allocated while the scope is valid:

```
/** This function is EVIL. */
char *foo() {
    char string[32]; ...;
    return string;
}
```

- An array is passed to a function as a pointer

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

# Arrays and Functions

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

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

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

# Arrays and Functions

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

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

# Arrays and Functions

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

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

`sizeof(int *)`

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

`10*sizeof(int)`

# Outline

# Administrivia

- Lab 2 tomorrow
- HW1 due this Sunday

# Outline

# C Strings

- A string in C is just an array of characters

```
char string[] = "abc"; // 4 bytes needed
```

- How do you tell how long a string is?
  - Last character is followed by a *null terminator* (`'\0' == 0`)
  - *Need extra space in array for null terminator*

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

# C String Libraries

- Accessible with `#include <string.h>`
- `int strlen(char *string);`
  - Returns the length of `string` (*excluding* the null terminator)
- `int strcmp(char *str1, char *str2);`
  - Compares `str1` and `str` according to a lexical ordering
  - 0 if `str1` is identical to `str2` (how different from `str1 == str2`?)
- `char *strcpy(char *dst, char *src);`
  - Copies the contents of `src` to the memory pointed to by `dst`. Caller must ensure that `dst` is large enough to hold the copied data
  - Why not `dst = src`?

# String Examples

```
#include <stdio.h>
#include <string.h>
int main () {
    char s1[10], s2[10];
    char s3[]="hello", *s4="hola";
    strcpy(s1,"hi"); strcpy(s2,"hi");
}
```

Values of the following expressions?

1. `sizeof(s1)`
2. `strlen(s1)`
3. `s1 == s2`
4. `strcmp(s1,s2)`
5. `strcmp(s1,s3)`
6. `strcmp(s1,s4)`

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

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

(blue) Always throws an error

(green) changes characters in string `t` to the next character in the string `s`

(purple) Copies a string at address `t` to the string at address `s`

(yellow) Appends the string at address `t` to the end of the string at address `s`

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

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

(blue) Always throws an error

(green) changes characters in string `t` to the next character in the string `s`

(purple) Copies a string at address `t` to the string at address `s`

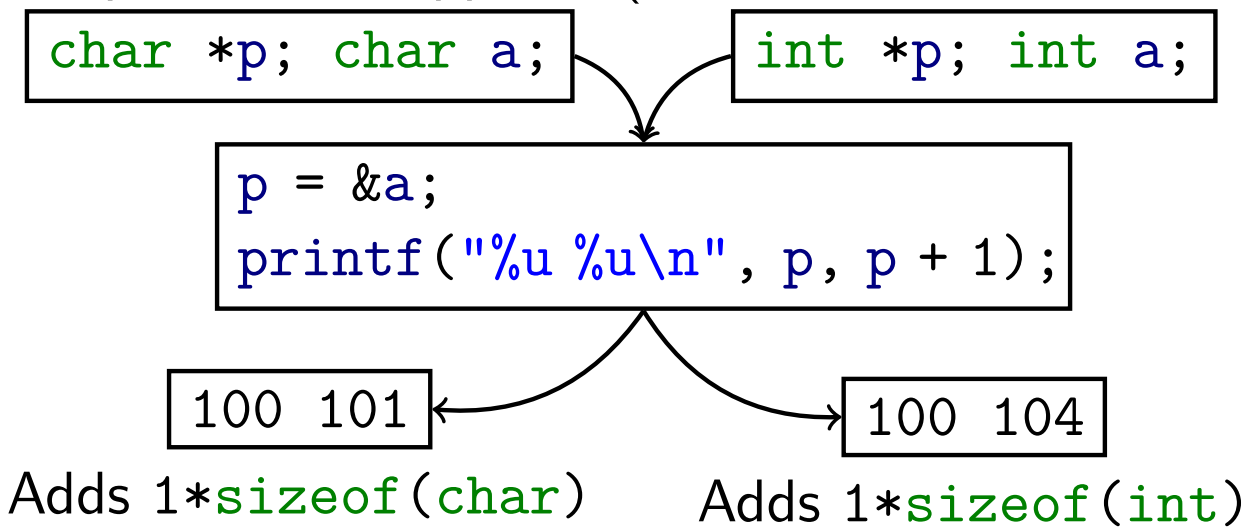
(yellow)

Appends the string at address `t` to the end of the string at address `s`

# Outline

# Pointer Arithmetic

- pointer ± number
  - e.g. `p + 1` adds 1 something to `p`
- Compare what happens: (assume `a` at address 100)



# 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 large structs?
  - Struct declaration tells C the size to use, so handled like basic types

# Pointer Arithmetic

- What constitutes valid pointer arithmetic?
  - Add an integer to a pointer
  - Substract 2 pointers (in the same array)
  - Compare pointers (<, <=, ==, !=,>, >=)
  - Compare pointer to `NULL`
- Everything else is illegal since it makes no sense:
  - Adding two pointers
  - Multiplying pointers
  - Subtracting a pointer from an 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 += 1) {
        *to++ = *from++;
    }
}
```

- Note: we have to pass the size (`n`) to `copy`

**Question:** The first `printf` outputs `100 5 5 10`. What will the next two `printf`s 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]);
}
```

(blue) 101 10 5 10 then 101 11 5 11
(green) 104 10 5 10 then 104 11 5 11
(purple) 100 6 6 10 then 101 6 6 10
(yellow) 100 6 6 10 then 104 6 6 10

**Question:** The first `printf` outputs `100 5 5 10`. What will the next two `printf`s 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]);
}
```

(blue) 101 10 5 10 then 101 11 5 11

(green) 104 10 5 10 then 104 11 5 11

(purple) 100 6 6 10 then 101 6 6 10

(yellow) 100 6 6 10 then 104 6 6 10

# Delayed Icebreaker/Technology Break

- Here are the rules
  - You say your name, your question for me, and your answer to that question.
  - Then I answer your question and the next person goes.

# Delayed Icebreaker/Technology Break

- Here are the rules
  - You say your name, your question for me, and your answer to that question.
  - Then I answer your question and the next person goes.
- Who's first?

# Pointers and Allocation

- When you declare a pointer (e.g. `int *ptr;`), it doesn't actually point to anything yet
  - I points somewhere, but we don't know where
  - Dereferencing will usually cause an error
- **Option 1:** Point to something that already exists
  - `int *ptr, var; var = 5; ptr = &var;`
  - `var` has space implicitly allocated for it (declaration)
- **Option 2:** Allocate room in memory for something new to point to (next lecture)

# Pointers and Structures

Variable declarations:

```
struct point {
  int x;
  int y;
  /* As close to containing
   * an instance of ourself
   * as is possible. */
  struct point *p;
};

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

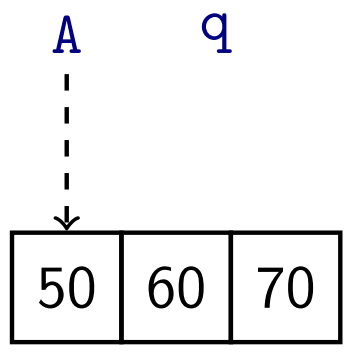
Some Valid operations:

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

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

/* struct assignment.
 * Copies contents. */
pt1 = pt2;
```

# Handles

- A pointer to a pointer, declared as `int **h` (of course, doesn't have to be an `int` handle.)
- Example:

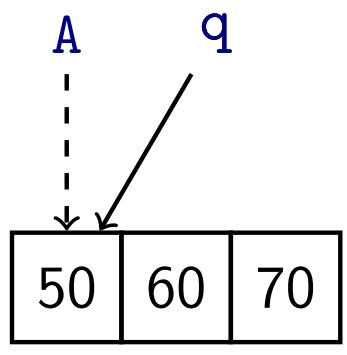
```
void incr_ptr(int **h) {
    *h = *h + 1;
}

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



# Handles

- A pointer to a pointer, declared as `int **h` (of course, doesn't have to be an `int` handle.)
- Example:

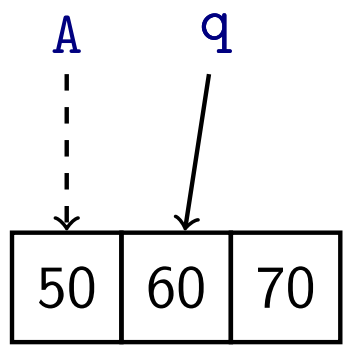
```
void incr_ptr(int **h) {
    *h = *h + 1;
}

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



# Handles

- A pointer to a pointer, declared as `int **h` (of course, doesn't have to be an `int` handle.)
- Example:

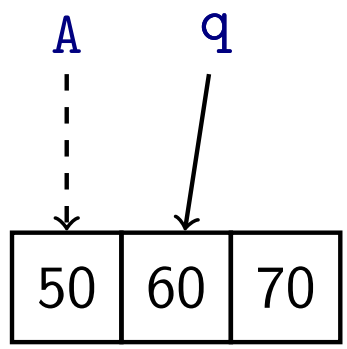
```
void incr_ptr(int **h) {
    *h = *h + 1;
}

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



# Handles

- A pointer to a pointer, declared as `int **h` (of course, doesn't have to be an `int` handle.)
- Example:

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

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




`*q = 60`

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

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

(blue) address
(green) data
(purple) invalid

1. `&p`
2. `p->name`
3. `p[7]->next`
4. `*(*(p + 2))`
5. `*(p[0]->next)`
6. `(*p)->next->name`

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

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

(blue) address
(green) data
(purple) invalid

1. `&p`
2. `p->name`
3. `p[7]->next`
4. `*(*(p + 2))`
5. `*(p[0]->next)`
6. `(*p)->next->name`

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

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

(blue) address
(green) data
(purple) invalid

1. `&p`
2. `p->name`
3. `p[7]->next`
4. `*(*(p + 2))`
5. `*(p[0]->next)`
6. `(*p)->next->name`

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

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

(blue) address
(green) data
(purple) invalid

1. `&p`
2. `p->name`
3. `p[7]->next`
4. `*(*(p + 2))`
5. `*(p[0]->next)`
6. `(*p)->next->name`

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

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

(blue) address
(green) data
(purple) invalid

1. `&p`
2. `p->name`
3. `p[7]->next`
4. `*(*(p + 2))`
5. `*(p[0]->next)`
6. `(*p)->next->name`

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

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

(blue) address
(green) data
(purple) invalid

1. `&p`
2. `p->name`
3. `p[7]->next`
4. `*(*(p + 2))`
5. `*(p[0]->next)`
6. `(*p)->next->name`

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

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

(blue) address
(green) data
(purple) invalid

1. `&p`
2. `p->name`
3. `p[7]->next`
4. `*(*(p + 2))`
5. `*(p[0]->next)`
6. `(*p)->next->name`

# 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