

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

Instructors:

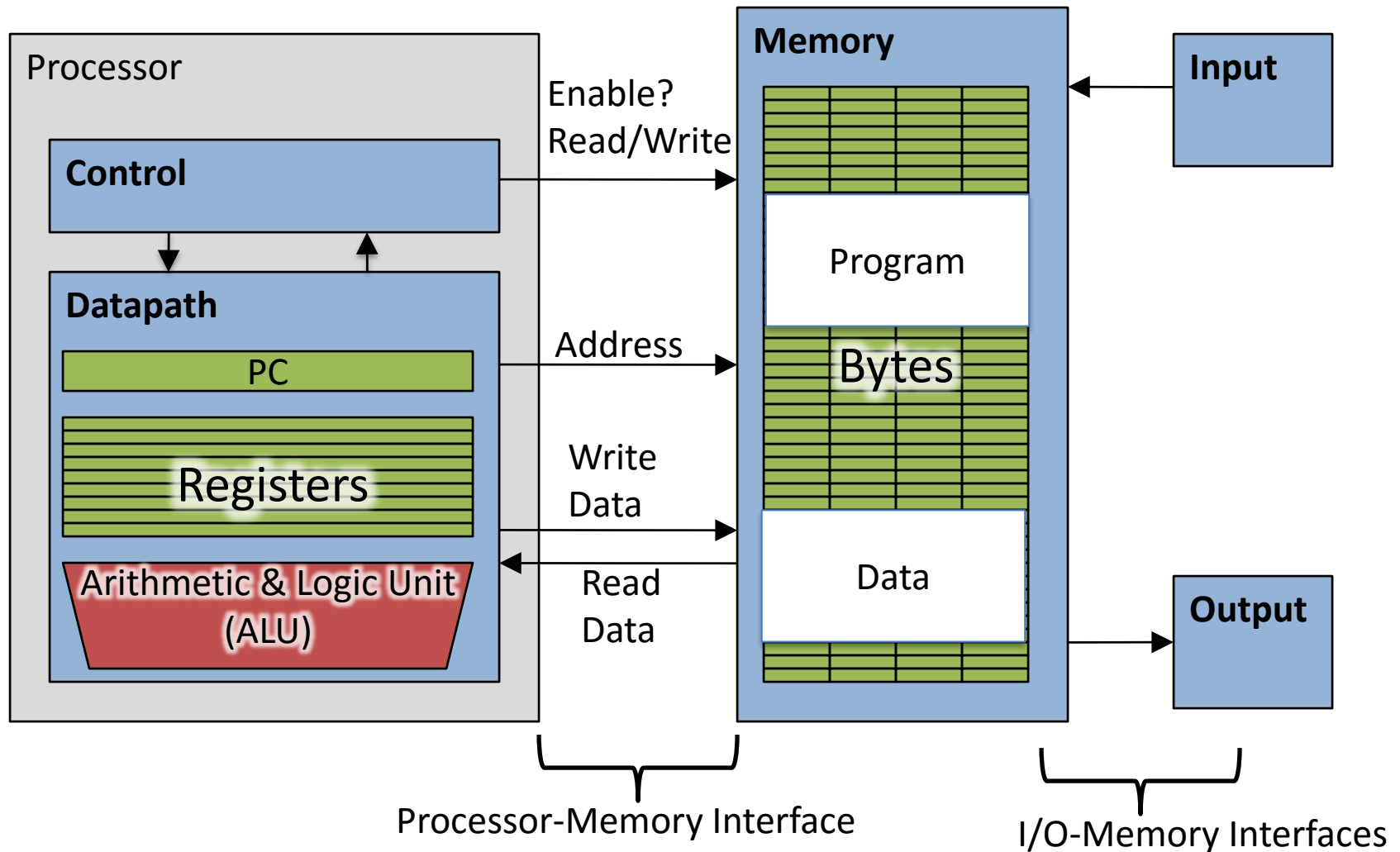
John Wawrzynek & Vladimir Stojanovic

<http://inst.eecs.Berkeley.edu/~cs61c/fa15>

# Agenda

- “Pointers” in C
- Clickers/Peer Instruction Time
- Administrivia
- Arrays in C
- News/Technology Break
- Pointers, Arrays, args to Main
- And in Conclusion, ...

# Components of a Computer



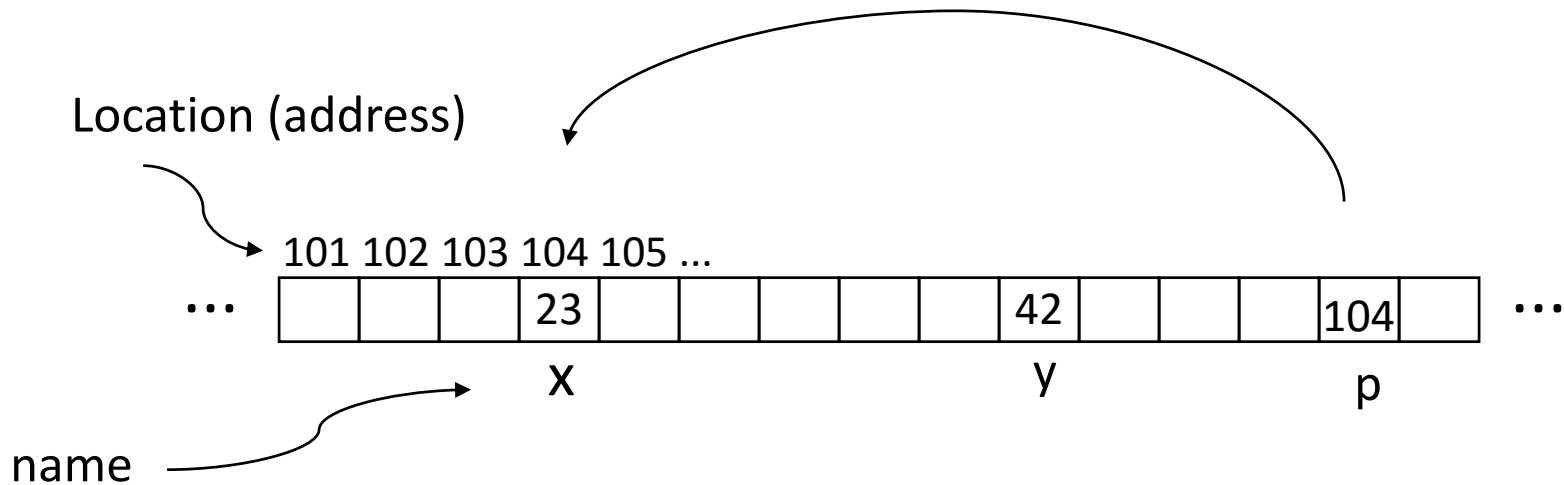
# Address vs. Value

- Consider memory to be a single huge array
  - Each cell of the array has an address associated with it
  - Each cell also stores some value
  - Are addresses signed or unsigned numbers?  
Negative address?!
- Don't confuse the address referring to a memory location with the value stored there



# Pointers

- An *address* refers to a particular memory location; e.g., it points to a memory location
- *Pointer*: A variable that contains the address of a variable



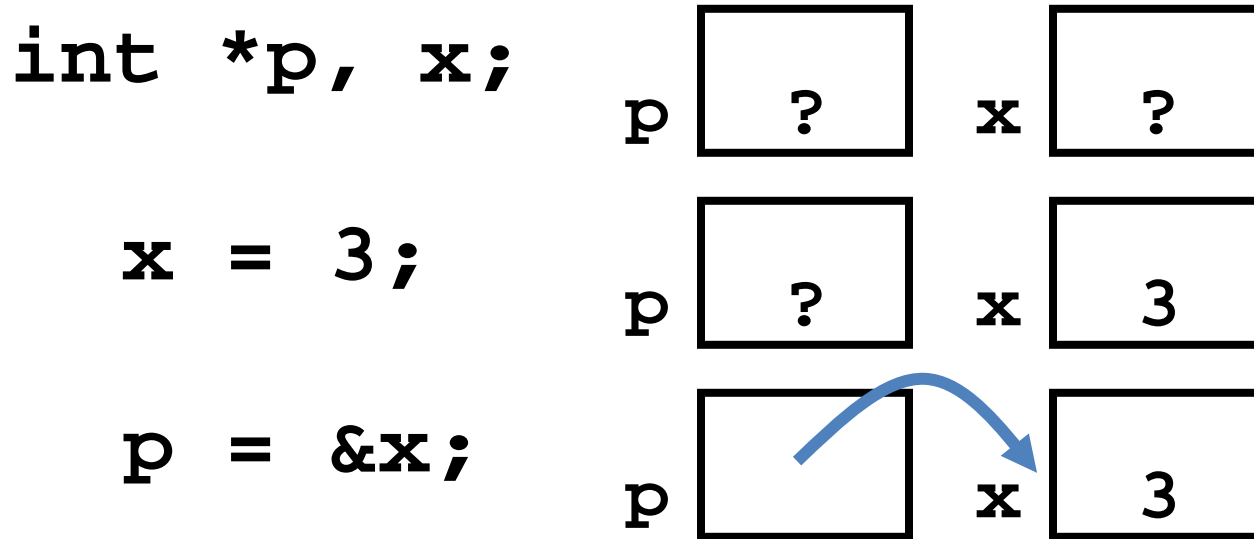
# Pointer Syntax

- `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

# Creating and Using Pointers

- How to create a pointer:

& operator: get address of a variable



Note the “\*” gets used 2 different ways in this example. In the declaration to indicate that `p` is going to be a pointer, and in the `printf` to get the value pointed to by `p`.

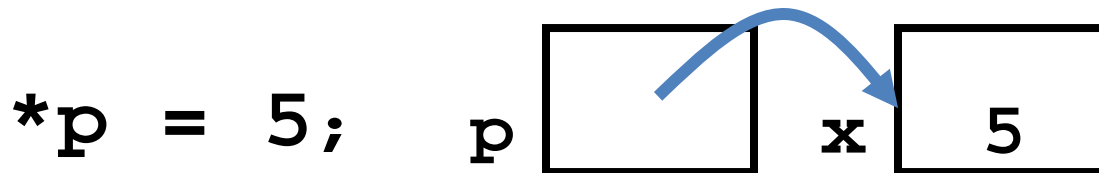
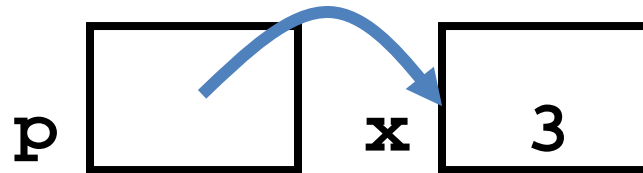
- How get a value pointed to?

“\*” (dereference operator): get the value that the pointer points to

```
printf("p points to %d\n", *p);
```

# Using Pointers for Writes

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





# 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 add_one (int x) {  
    x = x + 1;  
}  
int y = 3;  
add_one(y);
```

*y remains equal to 3*

# Pointers and Parameter Passing

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

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

```
int y = 3;
```

```
add_one(&y);
```

*y is now equal to 4*

# Types of Pointers

- Pointers are used to point to any kind of data (**int**, **char**, a **struct**, a pointer, 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 **void \*** sparingly to help avoid program bugs, and security issues, and other bad things!

# More C Pointer Dangers

- Declaring a pointer just allocates space to hold the pointer – 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;  
}
```

# Pointers and Structures

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

Point p1;
Point p2;
Point *paddr;

paddr = &p1;

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

/*structure assignment*/
p2 = p1;
```

Note, C structure assignment is not a “deep copy”. All members are copied, but not things pointed to by members.

# Pointers in C

- 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
  - Want to modify an object, not just pass its value
  - 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—coming up next lecture
    - *Dangling references* and *memory leaks*

# Why Pointers in C?

- At time C was invented (early 1970s), compilers often didn't produce efficient code
  - Computers 25,000 times faster today, compilers better
- C designed to let programmer say what they want code to do without compiler getting in way
- Today, many applications attain acceptable performance using higher-level languages without pointers
- Low-level system code still needs low-level access via pointers, hence continued popularity of C

# Video: Fun with Pointers

Worth a look.

[https://www.youtube.com/watch?v=6pmWojisM\\_E](https://www.youtube.com/watch?v=6pmWojisM_E)



# Clickers/Peer Instruction Time

```
void foo(int *x, int *y)
{   int t;
    if ( *x > *y ) { t = *y; *y = *x; *x = t; }
}
int a=3, b=2, c=1;
foo(&a, &b);
foo(&b, &c);
foo(&a, &b);
printf("a=%d b=%d c=%d\n", a, b, c);
```

Result is:

- A: a=3 b=2 c=1
- B: a=1 b=2 c=3
- C: a=1 b=3 c=2
- D: a=3 b=3 c=3
- E: a=1 b=1 c=1

# Administrivia

- We can accommodate all those on the wait-list, but you have to enroll in a lab section with space!
  - Lab section is important, but you can attend different discussion section
  - **Enroll into lab with space**, and try to swap with someone later
- HW0 out, due: Sunday 9/06 @ 11:59:59pm
- Give paper copy of mini-bio to your TA
- Get iClickers and register on bCourses! Participation points starts today!
- People with *university-related time conflict* with lectures should contact the head GSIs. We will if the participation portion of the grade can be made-up in some other way.
- Let us know about exam conflicts by the end of this week

# C Arrays

- 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

# 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;  
}
```

# Array Name / Pointer Duality

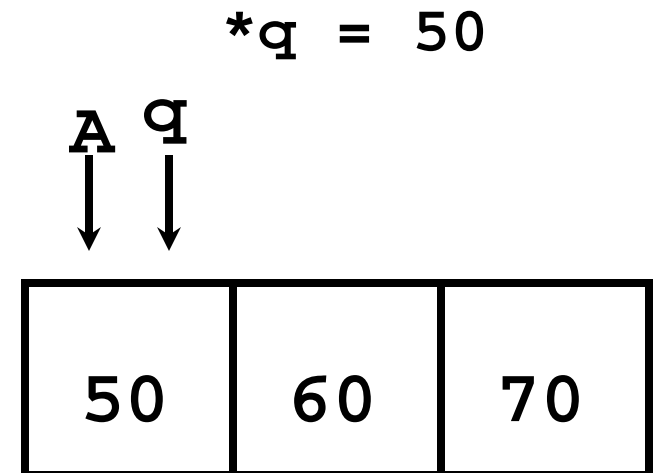
- *Key Concept:* Array variable is a “pointer” to the first (0<sup>th</sup>) element
- So, array variables almost identical to pointers
  - `char *string` and `char string[]` are nearly identical declarations
  - Differ in subtle ways: incrementing, declaration of filled arrays
- Consequences:
  - `ar` is an array variable, but works like a pointer
  - `ar[0]` is the same as `*ar`
  - `ar[2]` is the same as `*(ar+2)`
  - Can use pointer arithmetic to conveniently access arrays

# Changing a Pointer Argument?

- What if want function to change a pointer?
- What gets printed?

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

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

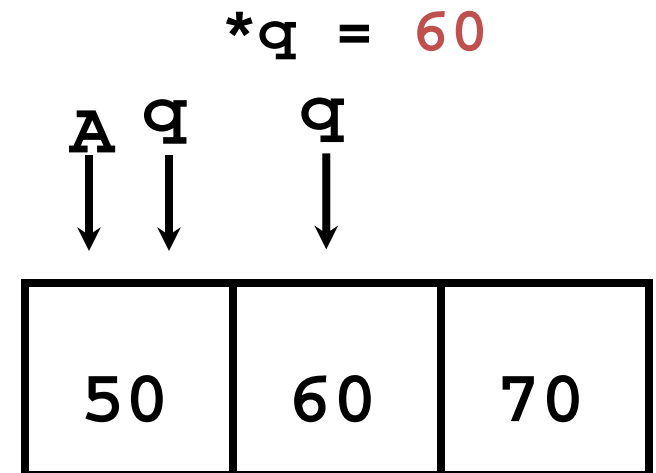


# Pointer to a Pointer

- Solution! Pass a pointer to a pointer, declared as **\*\*h**
- Now what gets printed?

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

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



# C Arrays are Very Primitive

- 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)



# Use Defined Constants

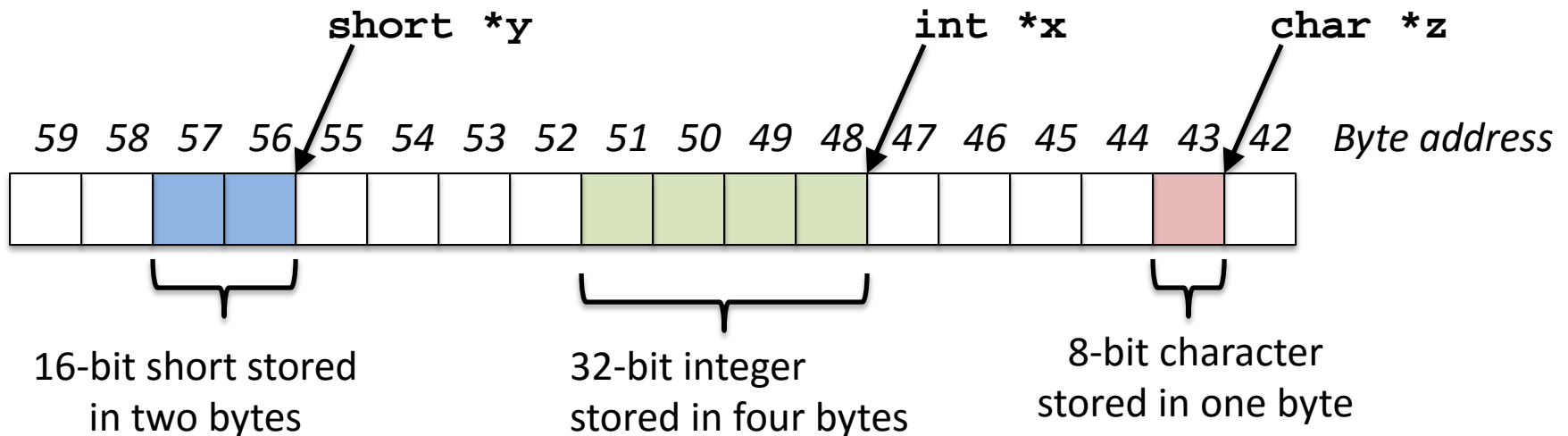
- Array size  $n$ ; want to access from  $0$  to  $n-1$ , so you should use counter AND utilize a variable for declaration & incrementation
  - Bad pattern

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

```
const 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"

# Pointing to Different Size Objects

- Modern machines are “byte-addressable”
  - Hardware’s memory composed of 8-bit storage cells, each has a unique address
- A C pointer is just abstracted memory address
- Type declaration tells compiler how many bytes to fetch on each access through pointer
  - E.g., 32-bit integer stored in 4 consecutive 8-bit bytes



# sizeof() operator

- sizeof(type) returns number of bytes in object
  - But number of bits in a byte is not standardized
    - In olden times, when dragons roamed the earth, bytes could be 5, 6, 7, 9 bits long
- By definition, sizeof(char)==1
- Can take sizeof(arr), or sizeof(structtype)
- We'll see more of sizeof when we look at dynamic memory management

# 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;
```

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

***Never code like this!!!!***

```
int *p;  
int a;  
int b;  
  
p = &a;  
p += 1;
```

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

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

*Pointer arithmetic should be used cautiously*

# 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) ...
}
```

# 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!

# Clickers/Peer Instruction Time

```
int x[] = { 2, 4, 6, 8, 10 };  
int *p = x;  
int **pp = &p;  
(*pp)++;  
(*(*pp))++;  
printf("%d\n", *p);
```

Result is:

A: 2

B: 3

C: 4

D: 5

E: None of the above



# In the News:

## Researchers produce industry's first 7nm node test chips



(July 9, 2015) An alliance led by IBM Research today announced that it has produced the semiconductor industry's first 7nm (nanometer) node test chips with functioning transistors

Read more at:

<http://phys.org/news/2015-07-industry-7nm-node-chips.html#>

Microprocessors utilizing 22nm and 14nm technology power today's servers, cloud data centers and mobile devices

# Concise strlen()

```
int strlen(char *s)
{
    char *p = s;
    while (*p++)
        ; /* Null body of while */
    return (p - s - 1);
}
```

What happens if there is no zero character at end of string?

# Point past end of array?

- 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., not cause an error

# 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 illegal since makes no sense:

- adding two pointers
- multiplying pointers
- subtract pointer from integer

# Arguments in `main ( )`

- To get arguments to the main function, use:
  - `int main(int argc, char *argv[ ])`
- What does this mean?
  - `argc` contains the number of strings on the command line (the executable counts as one, plus one for each argument). Here `argc` is 2:  
`unix% sort myFile`
  - `argv` is a *pointer* to an array containing the arguments as strings

# Example

- `foo hello 87`
- `argc = 3 /* number arguments */`
- `argv[0] = "foo",`  
`argv[1] = "hello",`  
`argv[2] = "87"`
  - Array of pointers to strings

# And In Conclusion, ...

- Pointers are an abstraction of machine memory addresses
- Pointer variables are held in memory, and pointer values are just numbers that can be manipulated by software
- In C, close relationship between array names and pointers
- Pointers know the type of the object they point to (except void \*)
- Pointers are powerful but potentially dangerous