CS 110
Computer Architecture

Lecture 2: *Introduction to C I*

Instructors:
Sören Schwertfeger & Chundong Wang

https://robotics.shanghaitech.edu.cn/courses/ca/20s/

School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkley's CS61C

Online Class: Rules I

- Displayed names have to be your name in pinyin.
- I will ask some questions during the lecture. We will try two ways - let's see how they work:
  - A) I will enable that everybody can unmute himself - a student who wants to answer unmutes and gives the answer.
  - B) If that is too chaotic: Students that want to answer write a 'I' in the chat. I will then select and unmute one of those students to give the answer.

Online Class: Rules II

- The use of chat is allowed with the following rules:
  - Only use English.
  - No irrelevant chatter.
  - Write short and concise.
  - You may report on technical aspects there or
  - **You may ask questions during the lecture in the chat!**

Participation

- Participation is mandatory!
- We are required to record your participation and ensure your attention =>
- Quiz!
  - May be on piazza or
  - May be on autolab or
  - May be on gradescope!
  - Make sure your logins to all 3 are working!

Online Class Overview

- The online class has three parts:
  - Live lecture (30-40 minutes):
    - New content will be taught.
      - Typically this covers the same content of the first video of the next lecture. All videos of the next lecture will be published after this class.
      - You are encouraged to ask questions directly in chat – or use the chat to indicate that you want to ask a question live (you will be unmuted then).
  - Discussion round (up to 30 minutes):
    - Dedicated time to ask questions
  - Online Quiz (20 minutes)
    - Answer questions about previous content AND content of this lecture.

Introduction Chundong Wang

Chundong Wang (王春东)

2008

西安交通大学
XI'AN JIAOTONG UNIVERSITY

2014

NUS
National University of Singapore

2017

Agency for Science, Technology and Research
SINGAPORE

2020

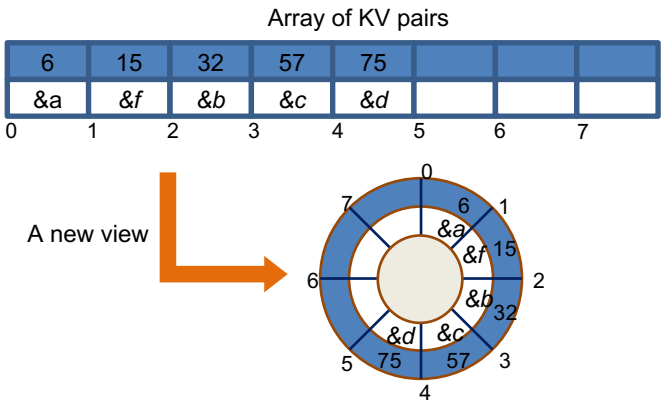
SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Since 2020

上海科技大学
ShanghaiTech University

Latest Research

Circ-Tree: a B+-tree variant for in-memory KV database



Higher **cache efficiency** for Insertion/Deletion

SweynTooth: a family of vulnerabilities of Bluetooth Low Energy Implementations

(a) FitBit Inspire (b) Eve Energy (c) August Smart Lock

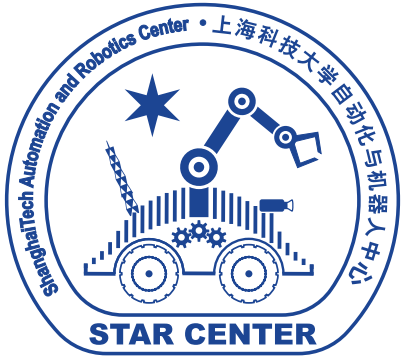
(d) CubiTag (e) eGeeTouch

More …

TOAST LAB
吐 司 实 验 室

http://toast-lab.tech

Introduction Sören Schwertfeger



Sören Schwertfeger

- Assistant Prof at ShanghaiTech since Aug. 2014
- Running the **M**obile **A**utonomous **R**obotic **S**ystems Lab (MARS Lab)
- 2005 – 2012: Ph.D. in Computer Science, Jacobs University Bremen, Germany
- **Robotics Enthusiast**

- Worked on intelligent functions for ground, underwater, aerial and space robots:
  - Autonomy; Mapping; Vision; Artificial Intelligence
- System Integration of Complex Robotics Software

MARS Lab

2 PhD Students; 7 Master Students; Several Undergraduates

Research on Mapping (SLAM)

Agenda

- Everything is a Number
- Compile vs. Interpret
- Pointers
- Pointers & Arrays

Agenda

- Everything is a Number
- Compile vs. Interpret
- Pointers
- Pointers & Arrays

BIG IDEA: Bits can represent anything!!

- Characters?
  - 26 letters $\Rightarrow$ 5 bits ($2^5 = 32$)
  - upper/lower case + punctuation
    $\Rightarrow$ 7 bits (in 8) ("ASCII")
  - standard code to cover all the world's languages $\Rightarrow$ 8,16, 32 bits ("Unicode")
    www.unicode.com

- Logical values?
  - 0 $\rightarrow$ False, 1 $\rightarrow$ True
- colors ? Ex: ***Red (00)*** ***Green (01)*** ***Blue (11)***
- locations / addresses? commands?
- MEMORIZE: N bits $\Leftrightarrow$ at most 2^N things

Ways to Make Two's Complement

- For N-bit word, complement to 2_{ten}^{N}
  - For 4 bit number $3_{ten}=0011_{two}$, two's complement

    (i.e. -3_{ten}) would be

    $16_{ten}-3_{ten}=13_{ten}$ or $10000_{two} - 0011_{two} = 1101_{two}$
- Here is an easier way:
  - Invert all bits and add 1
  - Computers actually do it like this, too

3_{ten} 0011_{two}

Bitwise complement 1100_{two}

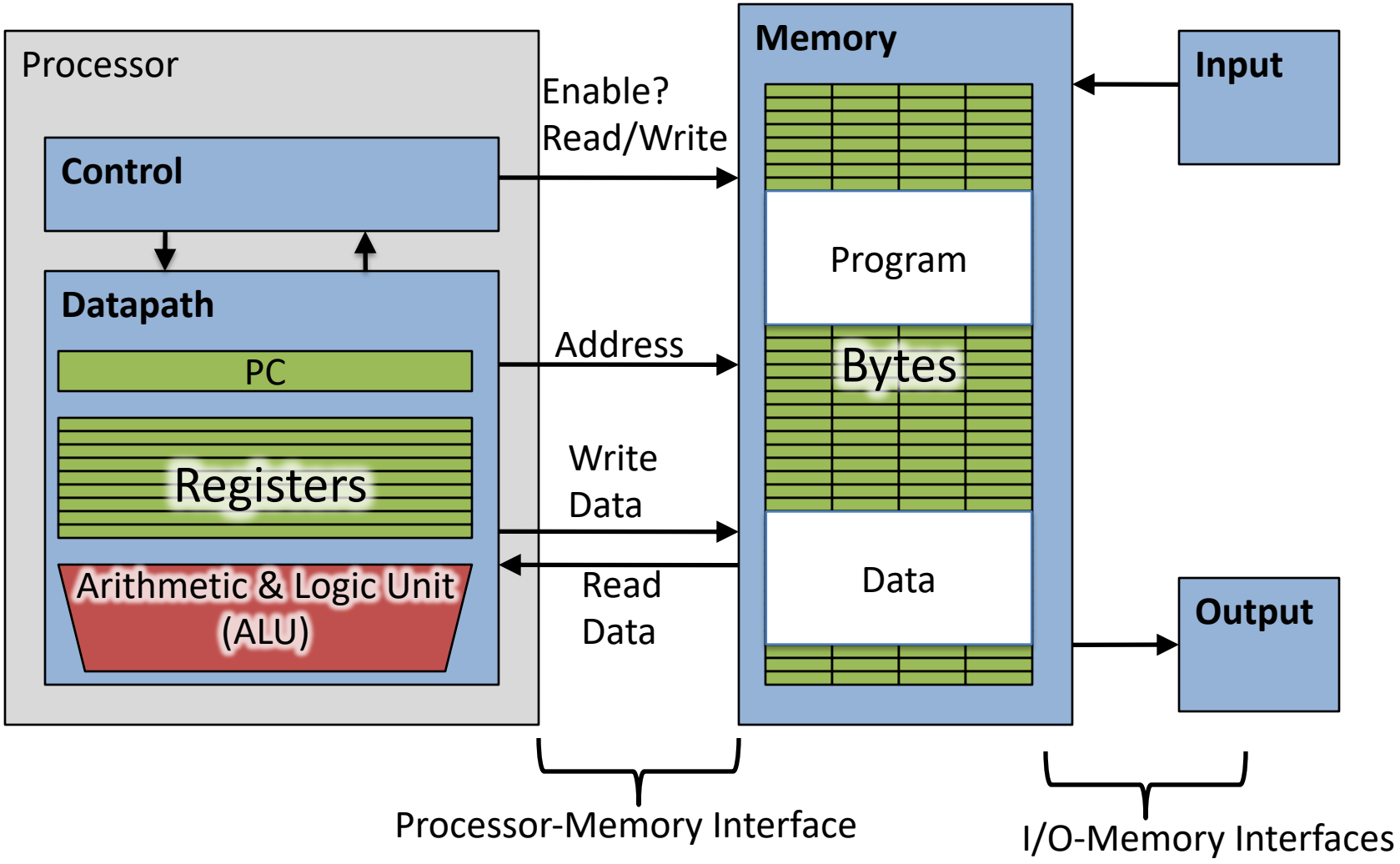
$+ \quad 1_{two}$

-3_{ten} 1101_{two}

Agenda

- Everything is a Number
- Compile vs. Interpret
- Pointers
- And in Conclusion, …

Components of a Computer



Great Idea: 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., RISC-V)

```
lw    t0, 0(s2)
lw    t1, 4(s2)
sw    t1, 0(s2)
sw    t0, 4(s2)
```

Assembler

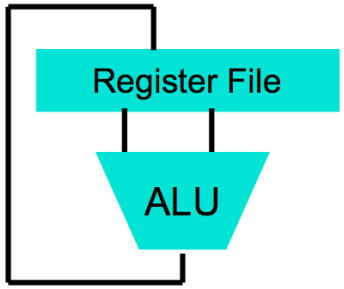
Machine Language Program (RISC-V)

```
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

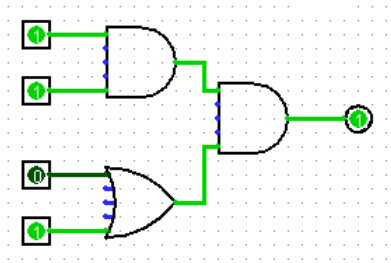
Machine Interpretation

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

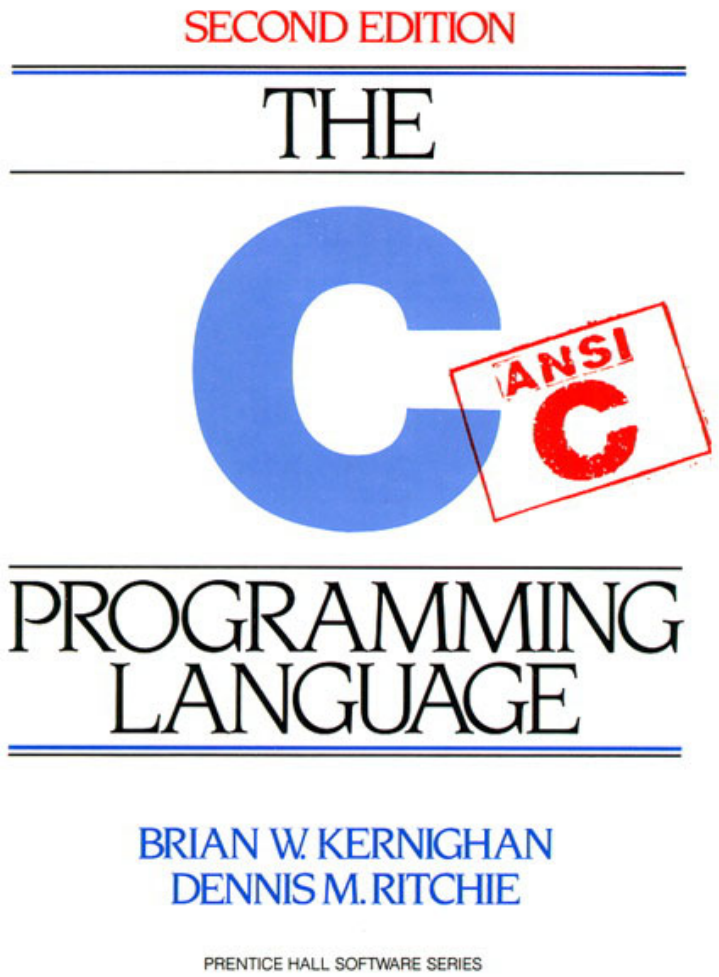


Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)



Introduction to C
“The Universal Assembly Language”



Intro to C

- *C is not a "very high-level" language, nor a "big" one, and is not specialized to any particular area of application. But its absence of restrictions and its generality make it more convenient and effective for many tasks than supposedly more powerful languages.*

  – Kernighan and Ritchie

- Enabled first operating system not written in assembly language: *UNIX* - A portable OS!

Intro to C

- Why C?: *we can write programs that allow us to exploit underlying features of the architecture – memory management, special instructions, parallelism*
- C and derivatives (C++/Obj-C/C#) still one of the most popular application programming languages after >40 years!

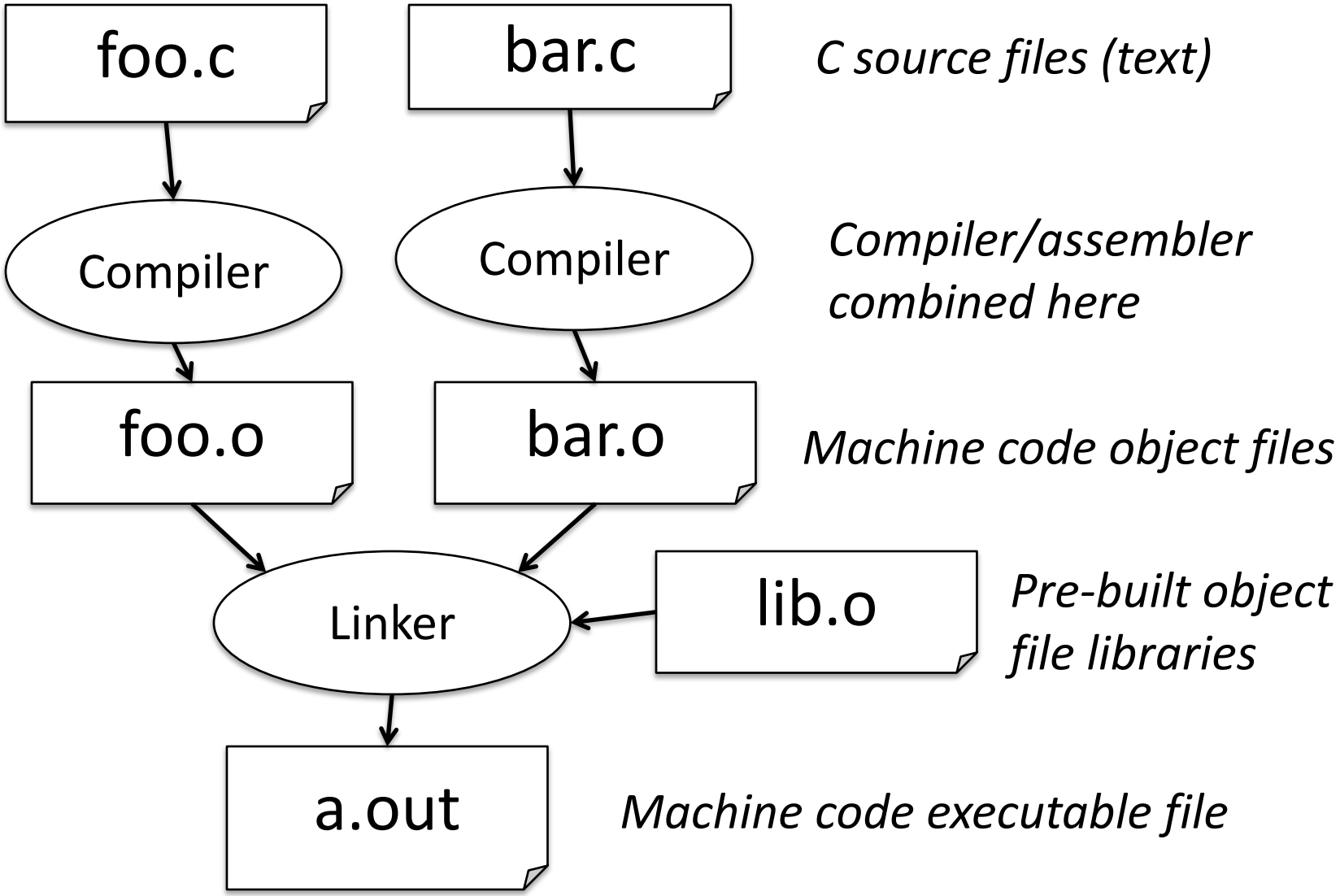
Disclaimer

- You will not learn how to fully code in C in these lectures! You'll still need your C reference for this course
  - K&R is a must-have
    - Check online for more sources
- Key C concepts: Pointers, Arrays, Implications for Memory management
- We will use ANSI C89 – original "old school" C
  - Because it is closest to Assembly

Compilation: Overview

- C *compilers* map C programs into architecture-specific machine code (string of 1s and 0s)
  - Unlike *Java*, which converts to architecture-independent *bytecode*
  - Unlike *Python* environments, which *interpret* the code
  - These differ mainly in exactly when your program is converted to low-level machine instructions ("levels of interpretation")
  - For C, generally a two part process of compiling .c files to .o files, then linking the .o files into executables;
  - Assembling is also done (but is hidden, i.e., done automatically, by default); we'll talk about that later

C Compilation Simplified Overview (more later in course)



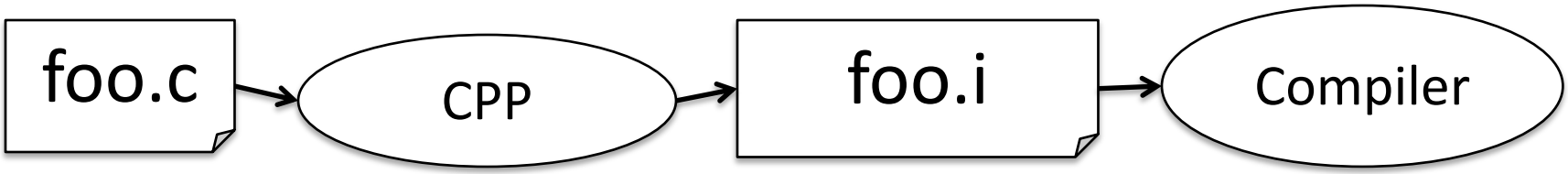
Compilation: Advantages

- Excellent run-time performance: generally much faster than Scheme or Java for comparable code (because it optimizes for a given architecture)
- Reasonable compilation time: enhancements in compilation procedure (Makefiles) allow only modified files to be recompiled

Compilation: Disadvantages

- Compiled files, including the executable, are architecture-specific, depending on processor type (e.g., MIPS vs. RISC-V) and the operating system (e.g., Windows vs. Linux)
- Executable must be rebuilt on each new system
  - I.e., "porting your code" to a new architecture
- "Change → Compile → Run [repeat]" iteration cycle can be slow during development
  - but Make tool only rebuilds changed pieces, and can do compiles in parallel (linker is sequential though -> Amdahl's Law)

C Pre-Processor (CPP)



- C source files first pass through macro processor, CPP, before compiler sees code
- CPP replaces comments with a single space
- CPP commands begin with “#”
- #include “file.h” /* Inserts file.h into output */
- #include <stdio.h> /* Looks for file in standard location */
- #define M_PI (3.14159) /* Define constant */
- #if/#endif /* Conditional inclusion of text */
- Use -save-temps option to gcc to see result of preprocessing
- Full documentation at: **http://gcc.gnu.org/onlinedocs/cpp/**

Piazza Question: CPP Macro
Which one is the correct way?

Piazza: "Lecture 1 CPP Macro"

```
// Magnitude (Length) of Vector (x, y)
1) #define mag(x, y) = sqrt( x*x + y*y );
2) #define mag(x, y) = sqrt( x*x + y*y )
3) #define mag(x, y) = (sqrt( x*x + y*y ))
4) #define mag(x, y) sqrt( x*x + y*y );
5) #define mag(x, y) sqrt( x*x + y*y )
6) #define mag(x, y) (sqrt( x*x + y*y ))
7) #define mag(x, y) = sqrt( (x)*(x) + (y)*(y) )
8) #define mag(x, y) = sqrt( (x*x) + (y*y) );
9) #define mag(x, y) sqrt( (x*x) + (y*y) )
10)#define mag(x, y) (sqrt( (x*x) + (y*y) );)
11)#define mag((x), (y)) (sqrt( (x*x) + (y*y) ))
```

Piazza Question: CPP Macro

- Correct answer: **NONE**
- Most correct solution:

```
#define MAG(x, y) (sqrt( (x)*(x) + (y)*(y)))
```

- Rules:
  - Convention: macros are CAPITALIZED
  - Put parenthesis around arguments – if missing:
  - `#define MAG(x, y) (sqrt( (x*x) + (y*y)))`
  - `MAG( a+2, 1-b) =>`
    `sqrt( (a+2*a+2) + (1-b*1-b) =>`
    `sqrt( (3*a+2) + (1-2*b))`

Piazza Question: CPP Macro

- More Pitfalls:
  - Put the whole macro body in parentheses:
    - `#define ADD(a, b) (a) + (b)`
    - `int result = 3 * ADD( 2, 3); // is 15!? =>`
    - `int result = 3 * 2 + 3; // is 9`
  - => Convention: put parenthesis EVERYWHERE!
  - Never put a semicolon after the macro:
    - `#define MAG(x, y) (sqrt( (x)*(x) + (y)*(y)));`
    - May work for:
    - `double len = MAG(a+2, 1-b);`
    - But not, for example, here:
    - `printf("Magnitude: %f ", MAG(a, b));`

Piazza Question: CPP Macro II

Piazza: "Lecture 1 CPP Macro II"

- Most Correct version:
- `#define MAG(x, y) (sqrt( (x)*(x) + (y)*(y)))`
- `int val = 3;`
- `double len = MAG(++val, 4);`
- `Printf(" val: %d  len^2: %f \n", val, len*len);`

A: val: 3 len^2: 25 B: val: 4 len^2: 25 C: val: 5 len^2: 25

D: val: 3 len^2: 32 E: val: 4 len^2: 32 F: val: 5 len^2: 32

G: val: 3 len^2: 36 H: val: 4 len^2: 36 I: val: 5 len^2: 36

J: val: 3 len^2: 41 K: val: 4 len^2: 41 L: val: 5 len^2: 41

M: val: 6 len^2: 25 N: val: 6 len^2: 32 O: val: 6 len^2: 36

P: val: 6 len^2: 41

Piazza Question: CPP Macro II

- Most Correct version:
- `#define MAG(x, y) (sqrt( (x)*(x) + (y)*(y)))`
- `int val = 3;`
- `double len = MAG(++val, 4);`
- `Printf(" val: %d  len^2: %f \n", val, len*len);`
- **Answer: I: val: 5 len^2: 36:**

```
double len = (sqrt((++val)*(++val) + (4)*(4)));
double len = (sqrt((4)*(5) + (4)*(4)));
```

```
test.c:14:19: warning: multiple unsequenced modifications to 'val'
      [-Wunsequenced]
       double len = MAG(++val, 4);
                        ^~
test.c:8:27: note: expanded from macro 'MAG'
#define MAG(x, y) (sqrt( (x)*(x) + (y)*(y)))
                          ^   ~
1 warning generated.
```

Piazza Question: CPP Macro II

- Avoid using macros whenever possible
- NO or very tiny speedup.
- Instead use C functions – e.g. inline function:

```
double mag(double x, double y);
double inline mag(double x, double y)
{  return sqrt( x*x + y*y ); }
```

- Read more…
- https://chunminchang.gitbooks.io/cplusplus-learning-note/content/Appendix/preprocessor_macros_vs_inline_functions.html

Typed Variables in C

```
int   variable1   = 2;
float variable2   = 1.618;
char  variable3   = 'A';
```

- Must declare the type of data a variable will hold
  - Types can't change

Type	Description	Examples
int	integer numbers, including negatives	0, 78, -1400
unsigned int	integer numbers (no negatives)	0, 46, 900
long	larger signed integer	-6,000,000,000
char	single text character or symbol	'a', 'D', '?'
float	floating point decimal numbers	0.0, 1.618, -1.4
double	greater precision/big FP number	10E100

Integers: Python vs. Java vs. C

Language	sizeof(int)
Python	>=32 bits (plain ints), infinite (long ints)
Java	32 bits
C	Depends on computer; 16 or 32 or 64

- C: `int` should be integer type that target processor works with most efficiently
- Only guarantee: sizeof(`long long`) ≥ sizeof(`long`) ≥ sizeof(`int`) ≥ sizeof(`short`)
  - Also, `short` >= 16 bits, `long` >= 32 bits
  - All could be 64 bits

Consts and Enums in C

- Constant is assigned a typed value once in the declaration; value can't change during entire execution of program

```
const float golden_ratio = 1.618;
const int days_in_week = 7;
```

- You can have a constant version of any of the standard C variable types
- Enums: a group of related integer constants. Ex:

```
enum cardsuit {CLUBS,DIAMONDS,HEARTS,SPADES};
enum color {RED, GREEN, BLUE};
```

Compare "`#define PI 3.14`" and "`const float pi=3.14`" – which is true?

A: Constants "PI" and "pi" have same type

B: Can assign to "PI" but not "pi"

C: Code runs at same speed using "PI" or "pi"

D: "pi" takes more memory space than "PI"

E: Both behave the same in all situations

C Syntax: Variable Declarations

- *All* variable declarations must appear before they are used (e.g., at the beginning of the block)
- A variable may be initialized in its declaration; if not, it holds garbage!
- Examples of declarations:
  - Correct:
```
{
    int a = 0, b = 10;
    ...
}
```
  - Incorrect:
```
for (int i = 0; i < 10; i++)
```

Newer C standards are more flexible about this…

C Syntax: True or False

- What evaluates to FALSE in C?
  - 0 (integer)
  - NULL (a special kind of *pointer*: more on this later)
  - *No explicit Boolean type*
- What evaluates to TRUE in C?
  - Anything that isn't false is true
  - Same idea as in Python: only 0s or empty sequences are false, anything else is true!

C operators

- arithmetic: +, -, *, /, %
- assignment: =
- augmented assignment: +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=
- bitwise logic: ~, &, |, ^
- bitwise shifts: <<, >>
- boolean logic: !, &&, ||
- equality testing: ==, !=
- subexpression grouping: ( )
- order relations: <, <=, >, >=
- increment and decrement: ++ and --
- member selection: ., ->
- conditional evaluation: ? :

Online Class: Admin

- Next: Question and Answer
- After: Discussion material presented by head TA Yanjie Song
- Then: Online Quiz

Q & A

Discussion by Yanjie Song

Online Quiz

Go to piazza and answer the 3 polls named:

Online Lecture 1 Feedback: XXX

CS 110
Computer Architecture

Lecture 2: *Introduction to C, Part I*
Video 2: Pointers

Instructors:
Sören Schwertfeger & Chundong Wang

https://robotics.shanghaitech.edu.cn/courses/ca/20s/

School of Information Science and Technology SIST

ShanghaiTech University

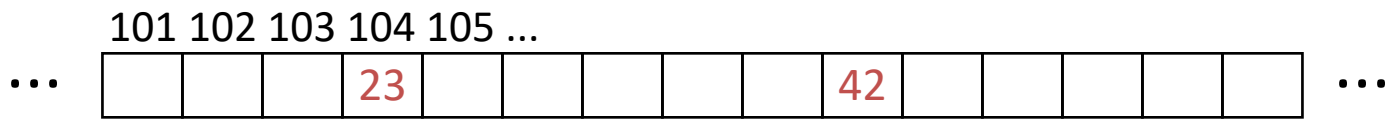
Slides based on UC Berkley's CS61C

Agenda

- Everything is a Number
- Compile vs. Interpret
- Pointers
- Pointers & Arrays

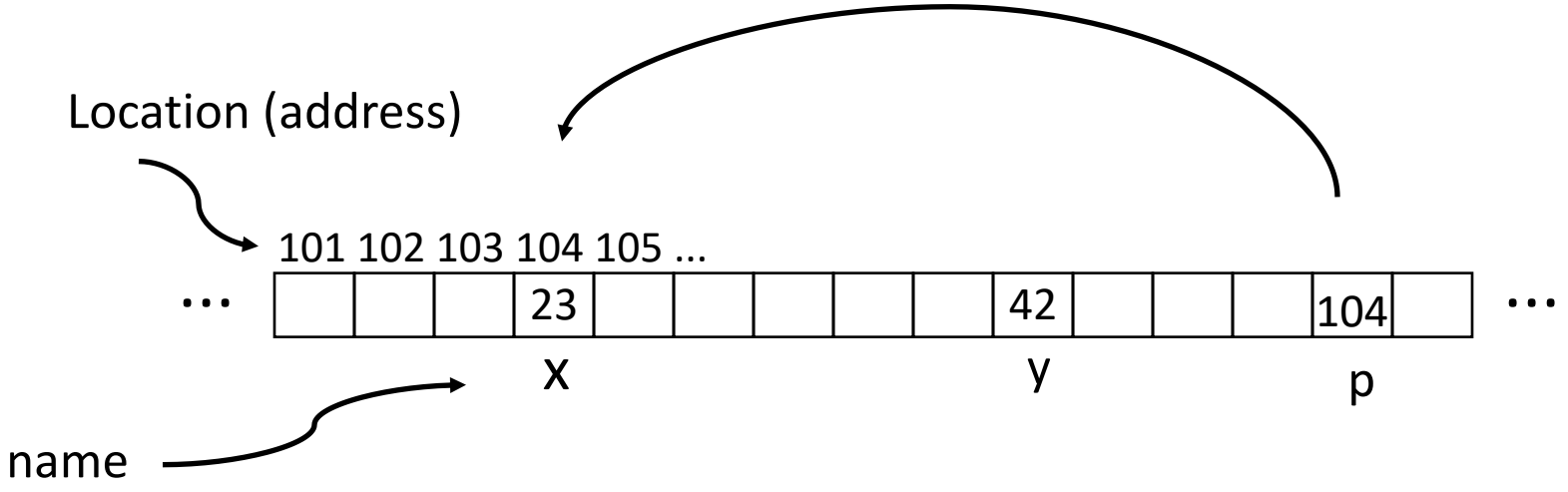
Address vs. Value

- 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
  - For addresses do we use 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

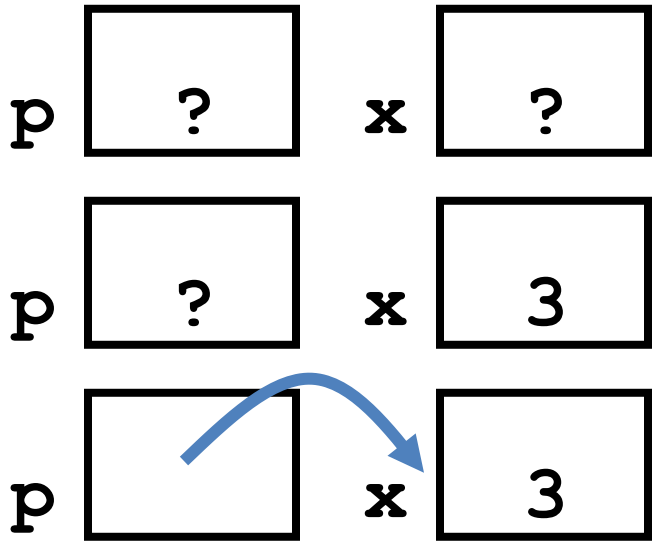
- How to create a pointer:

  & operator: get address of a variable

```
int *p, x;

   x = 3;

   p = &x;
```



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**.

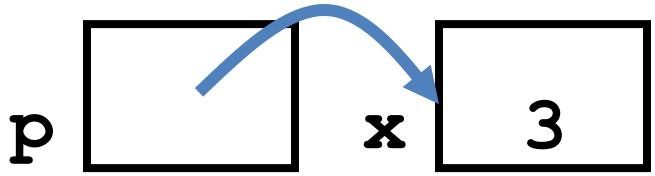
- How get a value pointed to?

  "*" (dereference operator): get the value that the pointer points to

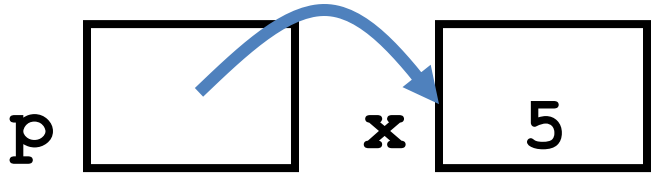
```
printf("p points to value %d\n",*p);
```

Using Pointer for Writes

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




```
*p = 5;
```



Pointers and Parameter Passing

- C passes 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

What would you use in C++?

Call by reference:

```
void add_one (int &p) {
   p = p + 1;   // or  p += 1;
}
```

Types of 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 **`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 – it 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 {
    int x;
    int y;
} Point;

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

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

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

/* This works too */
p1 = p2;
```

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
  - 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 week
    - *Dangling references* and *memory leaks*

Why Pointers in C?

- At time C was invented (early 1970s), compilers often didn't produce efficient code
  - Computers 100,000 times faster today, compilers better
- C designed to let programmer say what they want code to do without compiler getting in way
  - Even give compilers hints which registers to use!
- Today's compilers produce much better code, so may not need to use pointers in application code
- Low-level system code still needs low-level access via pointers

Quiz: Pointers

Piazza: "Lecture 2 Pointer poll"

```
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=3 c=2**

C: **a=3 b=3 c=3**

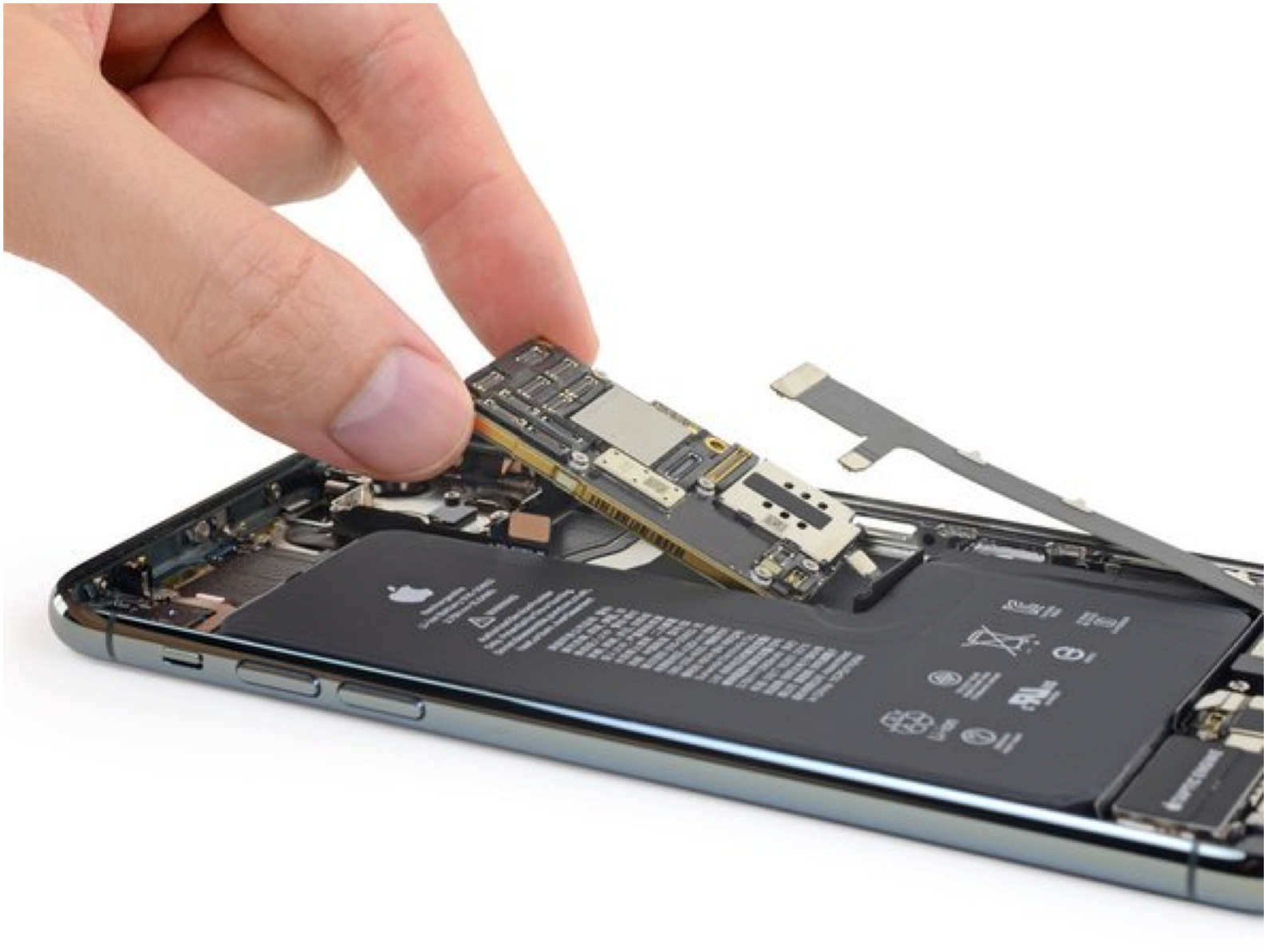
D: **a=1 b=2 c=3**

E: **a=1 b=1 c=1**

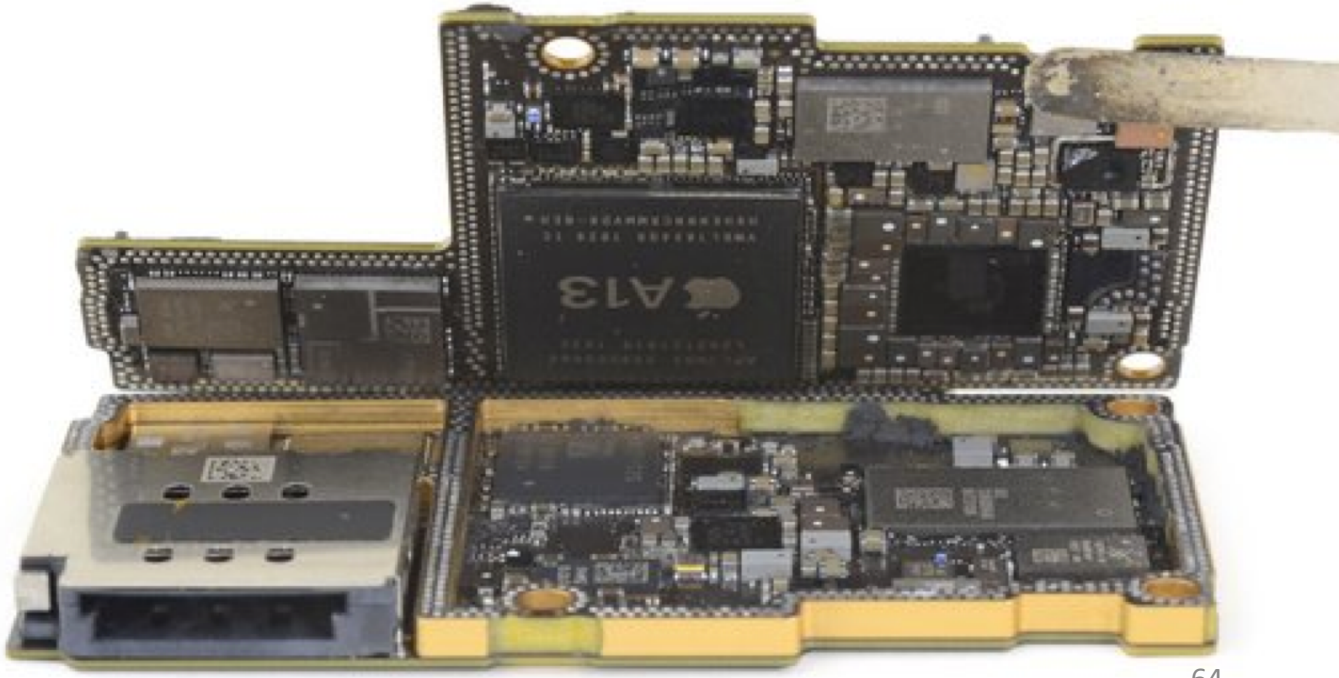
iPhone 11 Pro Max Teardown

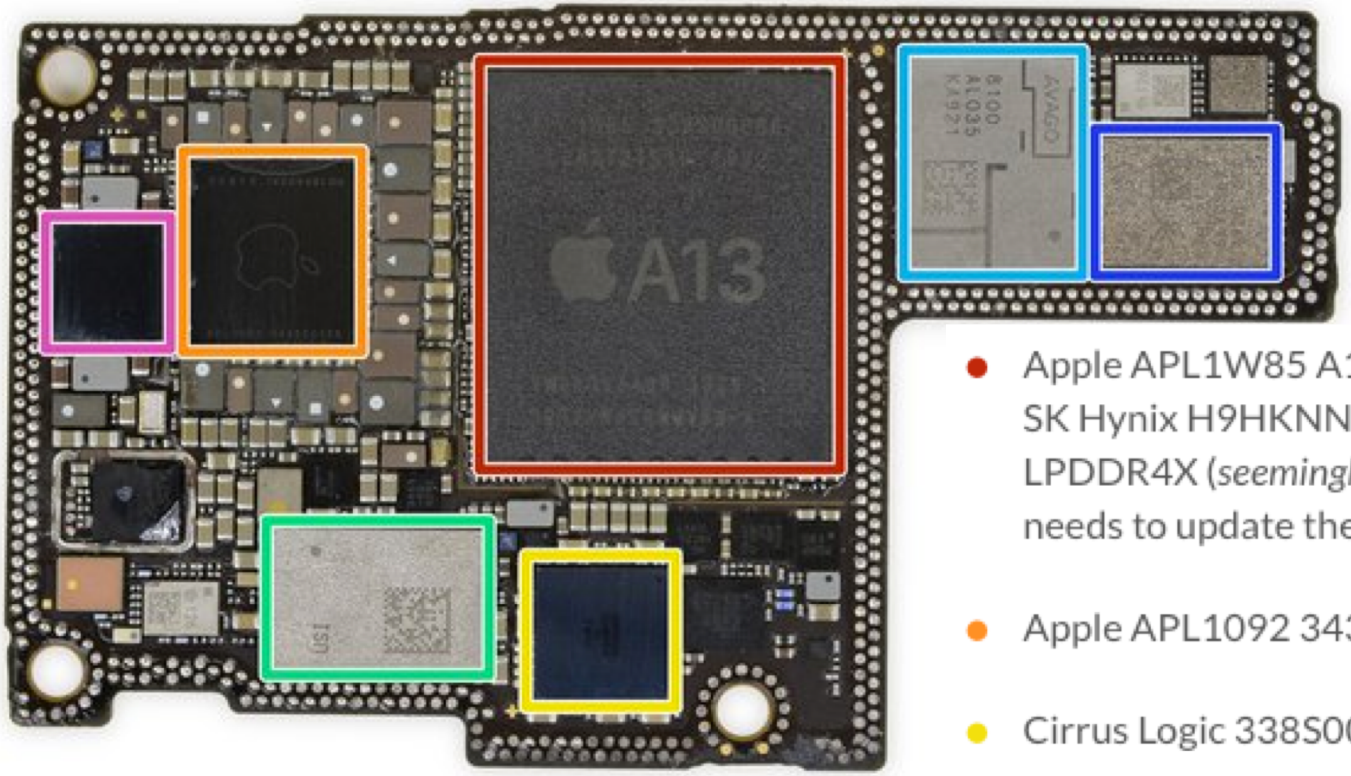
ifixit.com

Get logic board out
dual layer design



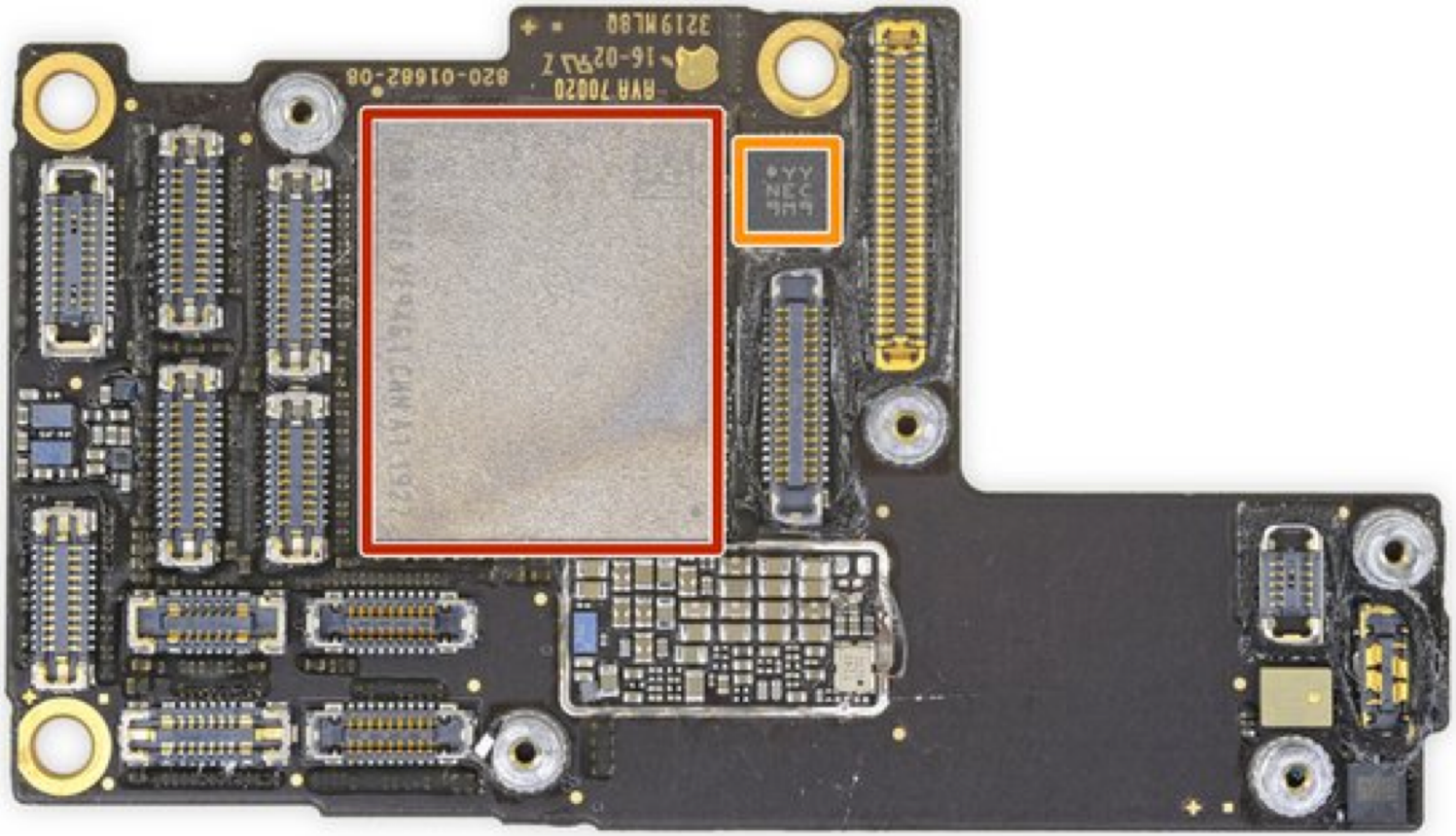
- Apple 64bit System on a chip (SoC); A13:
  - Hexa core (2 high performance (up to 2.66 GHz), 4 low power)
  - Apple designed GPU
  - Motion Processor; Image Processor; Neural Engine
  - 4 GB LPDDR4X (memory)
  - L1 cache: 128 KB instruction, 128 KB data (fast cores)
  - L2 cache: 8 MB; (fasy cores; 4MB slow cores)
  - L3 cache : yes, 16MB, shared with other cores (e.g. GPU)



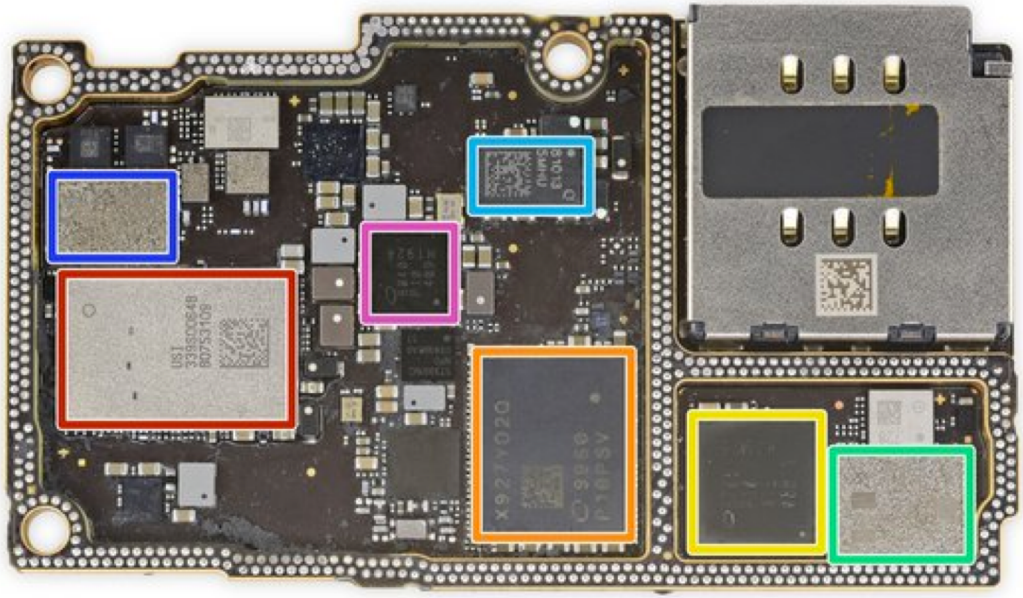


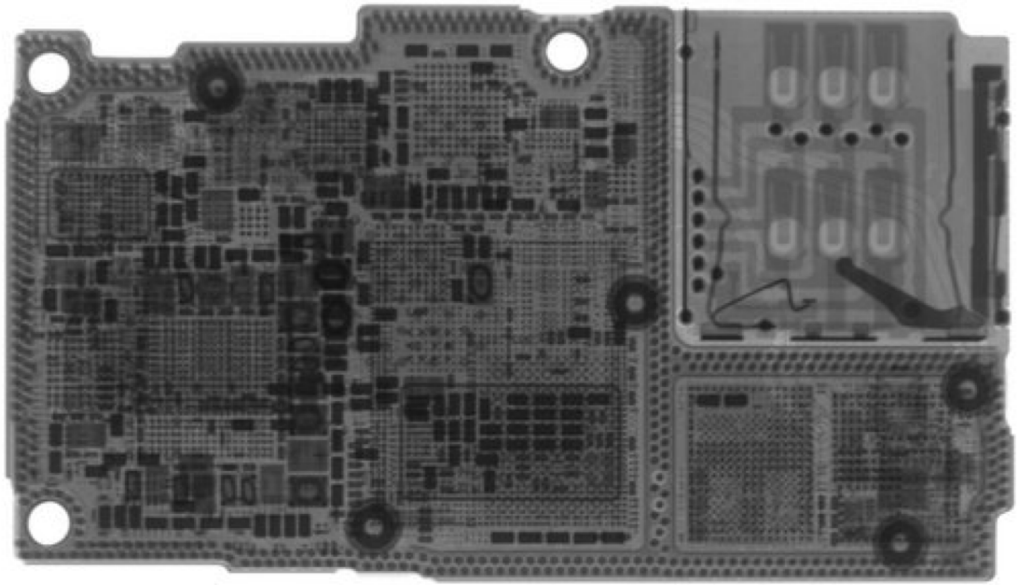
- Apple APL1W85 A13 Bionic SoC layered over SK Hynix H9HKNNNCRMMVDR-NEH LPDDR4X (*seemingly* 4 GB, but SK Hynix needs to update their decoder)
- Apple APL1092 343S00355 PMIC
- Cirrus Logic 338S00509 audio codec
- Unmarked USI module—**teardown update**: it turns out that this *is* where Apple's new U1 ultra-wideband chip is hiding. Read all about it about it in our blog post.
- Avago 8100 Mid/High band PAMiD
- Skyworks 78221-17 low-band PAMiD
- STMicrolectronics STB601A0N power management IC

- Toshiba TSB 4226VE9461CHNA1 1927 64 GB flash storage
- YY NEC 9M9 (likely accel/gyro)



RF board





- Apple/USI 339S00648 WiFi/Bluetooth SoC
- Intel X927YD2Q (likely XMM7660) modem
- Intel 5765 P10 A15 08B13 H1925 transceiver
- Skyworks 78223-17 PAM
- 81013 - Qorvo Envelope Tracking
- Skyworks 13797-19 DRx
- Intel 6840 P10 409 H1924 baseband PMIC


IFIXIT

CS 110
Computer Architecture

Lecture 2: *Introduction to C, Part I*
Video 4: Pointers & Arrays

Instructors:
Sören Schwertfeger & Chundong Wang

https://robotics.shanghaitech.edu.cn/courses/ca/20s/

School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkley's CS61C

Agenda

- Everything is a Number
- Compile vs. Interpret
- Pointers
- Pointers & Arrays

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^{th}) 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

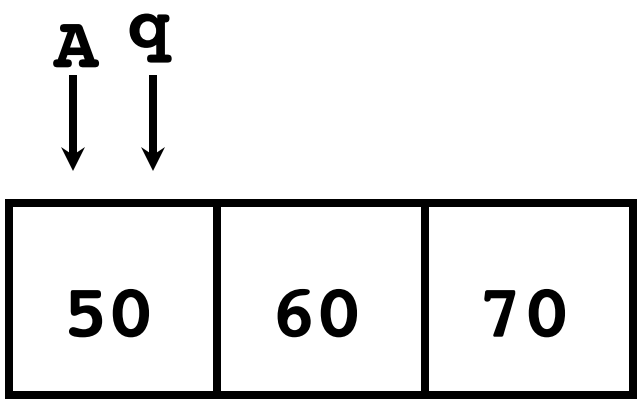
Changing a Pointer Argument?

- 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);
```

*q = 50

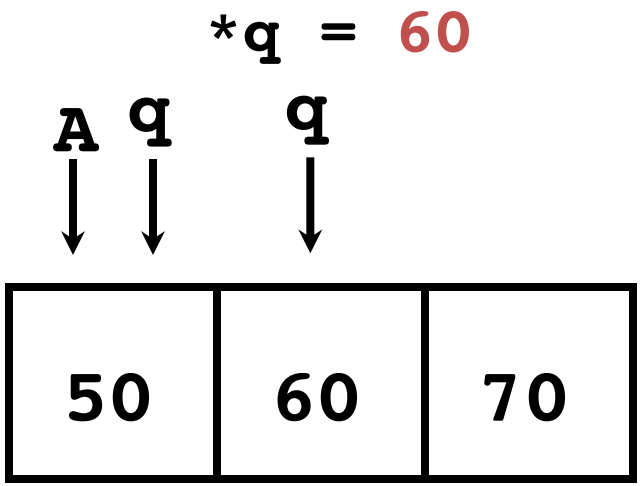


Pointer to a Pointer

- 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!

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++){ ... }
```

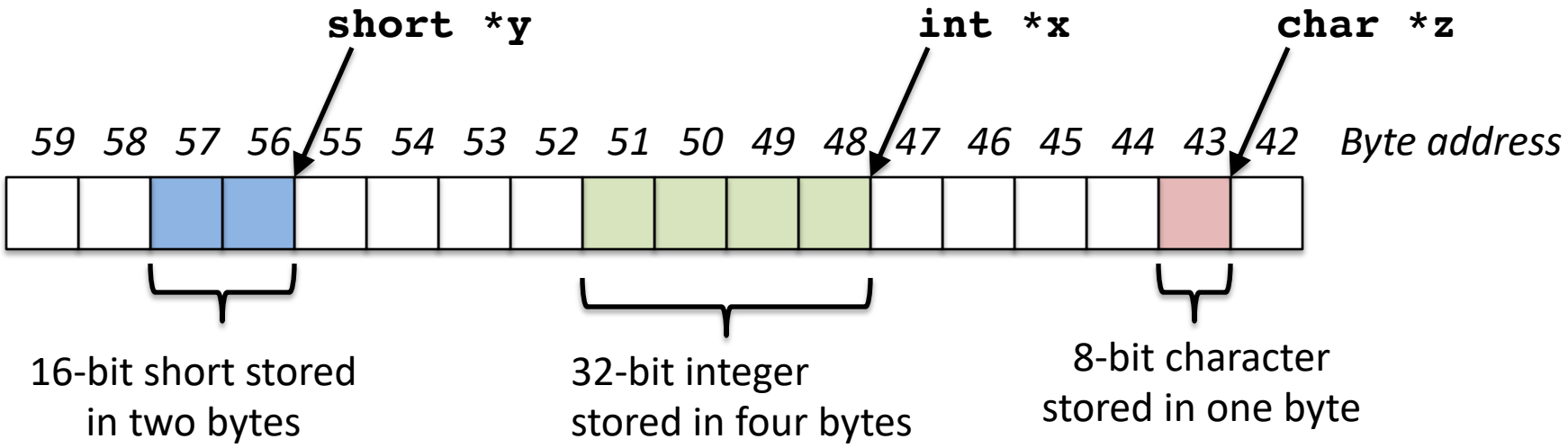
  - Better pattern

```
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

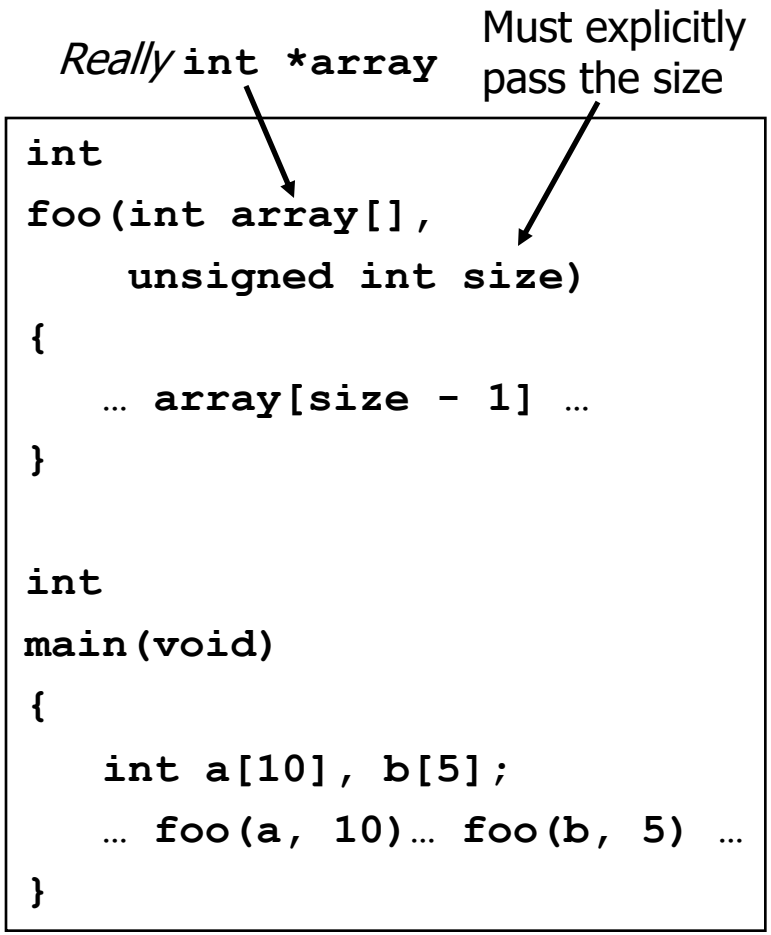
- Array ≈ pointer to the initial (0th) array element

```
a[i] ≡ *(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!

Passing arrays:



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 (32bit)? **4**

... because **`array`** is really a pointer (and a pointer is architecture dependent, but likely to be 8 on modern machines!)

What does this print (32bit)? **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!

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;
}
```

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

Quiz:

Piazza: "Lecture 2 PP poll"

```
int x[] = { 0, 2, 4, 6, 8, 10, 12, 14 };
int *p = x;
int **pp = &p;
(*pp)++;
(*pp)+= **pp;
(*(*pp))++;
printf("%d\n", *p);
```

Select the result in the poll.

Summary

- “Lowest High-level language”
  - Use ANSI C89 in class
  - => closest to assembler
- Pointers: powerful but dangerous
- Pointer arithmetic and arrays useful