CS 110
Computer Architecture

Lecture 3: *Introduction to C, Part II*

Instructor:

Sören Schwertfeger

http://shtech.org/courses/ca/

School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkley's CS61C

Agenda

- C Syntax
- Pointers
- C Memory Management

A First C Program: Hello World

Original C:

```
main()
{
   printf("\nHello World\n");
}
```

ANSI Standard C:

```
#include <stdio.h>

int main(void)
{
   printf("\nHello World\n");
   return 0;
}
```

C Syntax: `main`

- When C program starts
  - C executable a.out is loaded into memory by operating system (OS)
  - OS sets up stack, then calls into C runtime library,
  - Runtime 1st initializes memory and other libraries,
  - then calls your procedure named main ()
- We'll see how to retrieve command-line arguments in main() later…

A Second C Program: Compute Table of Sines

```c
#include <stdio.h>
#include <math.h>

int main(void)
{
    int    angle_degree;
    double angle_radian, pi, value;
    /* Print a header */
    printf("\nCompute a table of the
   sine function\n\n");

    /* obtain pi once for all       */
    /* or just use pi = M_PI, where */
    /* M_PI is defined in math.h    */
    pi = 4.0*atan(1.0);
    printf("Value of PI = %f \n\n",
   pi);
```

```c
printf("angle     Sine \n");

angle_degree = 0;
/* initial angle value */
/* scan over angle     */
while (angle_degree <= 360)
/* loop until angle_degree > 360 */
   {
      angle_radian = pi*angle_degree/180.0;
      value = sin(angle_radian);
      printf (" %3d      %f \n ",
                angle_degree, value);
      angle_degree = angle_degree + 10;
      /* increment the loop index */
   }
return 0;
}
```

Second C Program Sample Output

```
Compute a table of the sine
   function

Value of PI = 3.141593

angle        Sine
   0       0.000000
  10       0.173648
  20       0.342020
  30       0.500000
  40       0.642788
  50       0.766044
  60       0.866025
  70       0.939693
  80       0.984808
  90       1.000000
 100       0.984808
 110       0.939693
 120       0.866025
 130       0.766044
 140       0.642788
 150       0.500000
 160       0.342020
 170       0.173648
 180       0.000000
```

```
190       -0.173648
200       -0.342020
210       -0.500000
220       -0.642788
230       -0.766044
240       -0.866025
250       -0.939693
260       -0.984808
270       -1.000000
280       -0.984808
290       -0.939693
300       -0.866025
310       -0.766044
320       -0.642788
330       -0.500000
340       -0.342020
350       -0.173648
360       -0.000000
```

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, more later

C Syntax : Control Flow (1/2)

- Within a function, remarkably close to Java constructs in terms of control flow
  - `if-else`
    - `if (expression) statement`
    - `if (expression) statement1`
      `else statement2`
  - `while`

    -
      ```
      while (expression)
          statement
      ```

    -
      ```
      do
          statement
      while (expression);
      ```

C Syntax : Control Flow (2/2)

- for
  - `for (initialize; check; update) statement`
- switch
  - switch syntax:

```
switch (expression){
  case const1:     statements
  case const2:     statements
  default:         statements
}
```

  - break

C Syntax: True or False

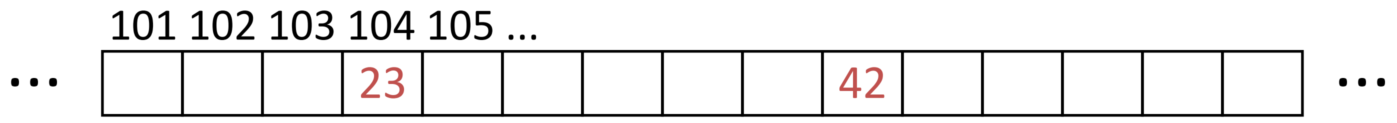
- What evaluates to FALSE in C?
  - 0 (integer)
  - NULL (a special kind of *pointer*: more on this later)
  - *No explicit Boolean type*
- What evaluates to TRUE in C?
  - Anything that isn't false is true
  - Same idea as in Python: only 0s or empty sequences are false, anything else is true!

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: ? :

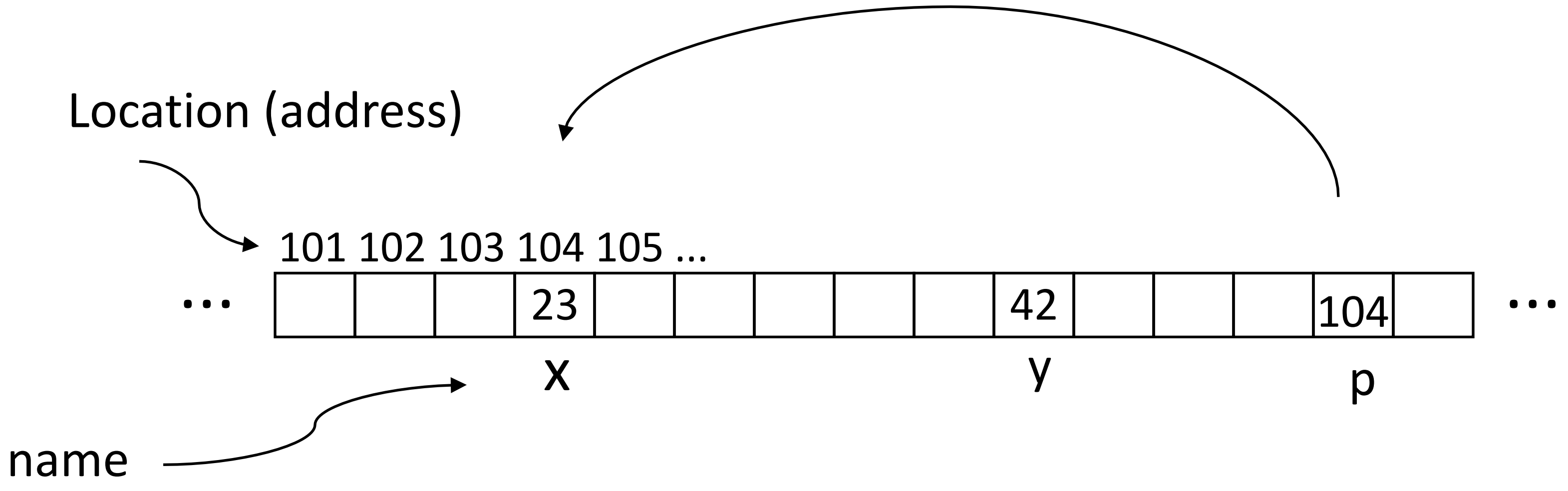
Address vs. Value

- 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

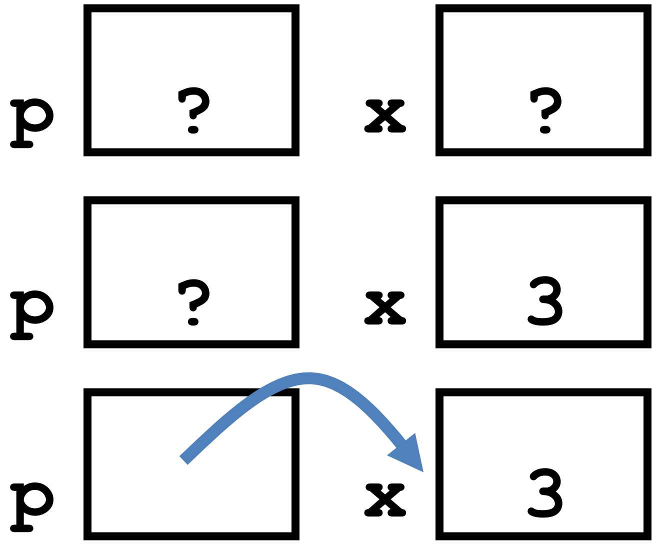
- How to create a pointer:

  `&` 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`**.

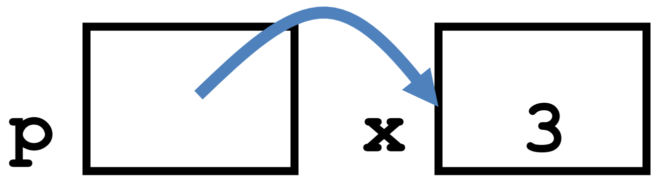
- How get a value pointed to?

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

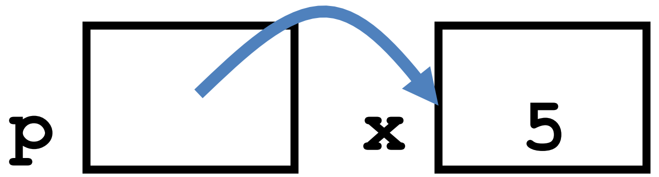
```
printf("p points to value %d\n",*p);
```

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 25,000 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

```
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`**

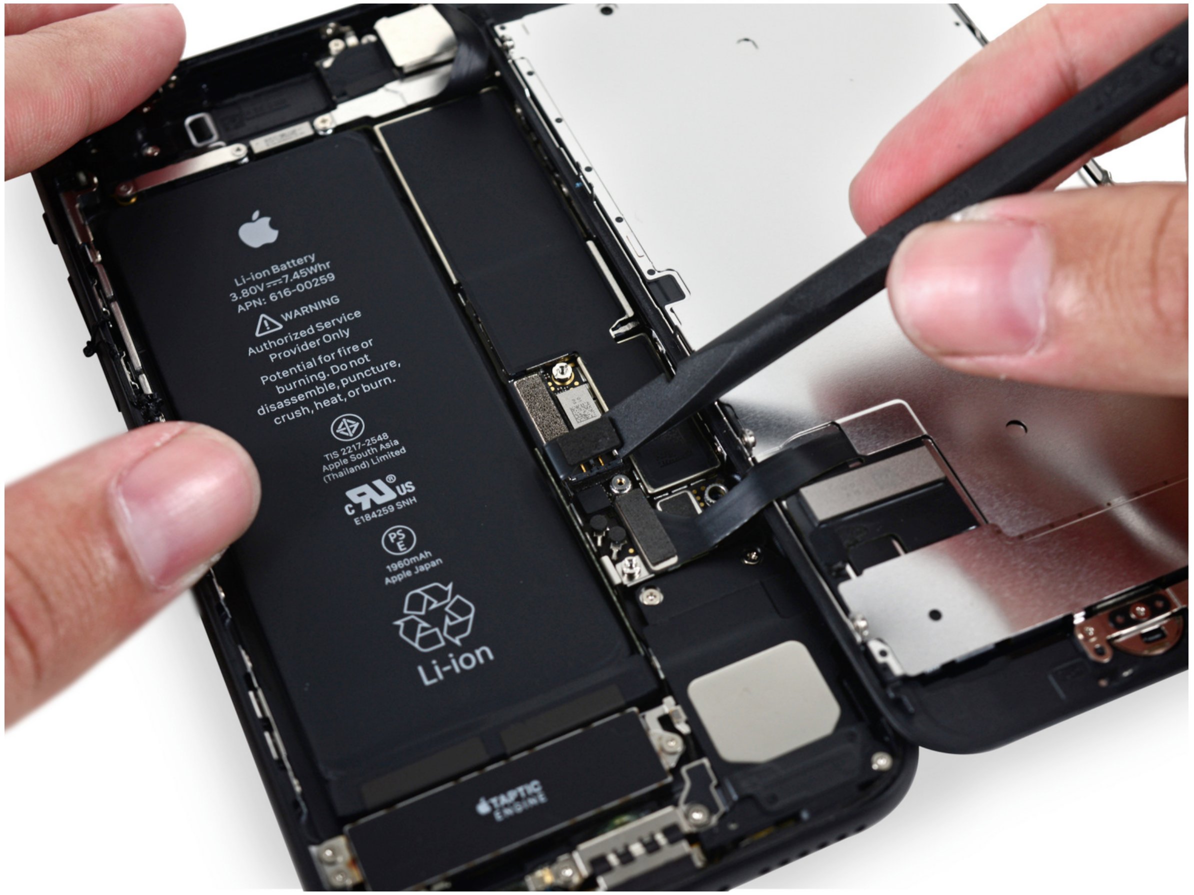
E: **`a=1  b=1  c=1`**

Administrivia

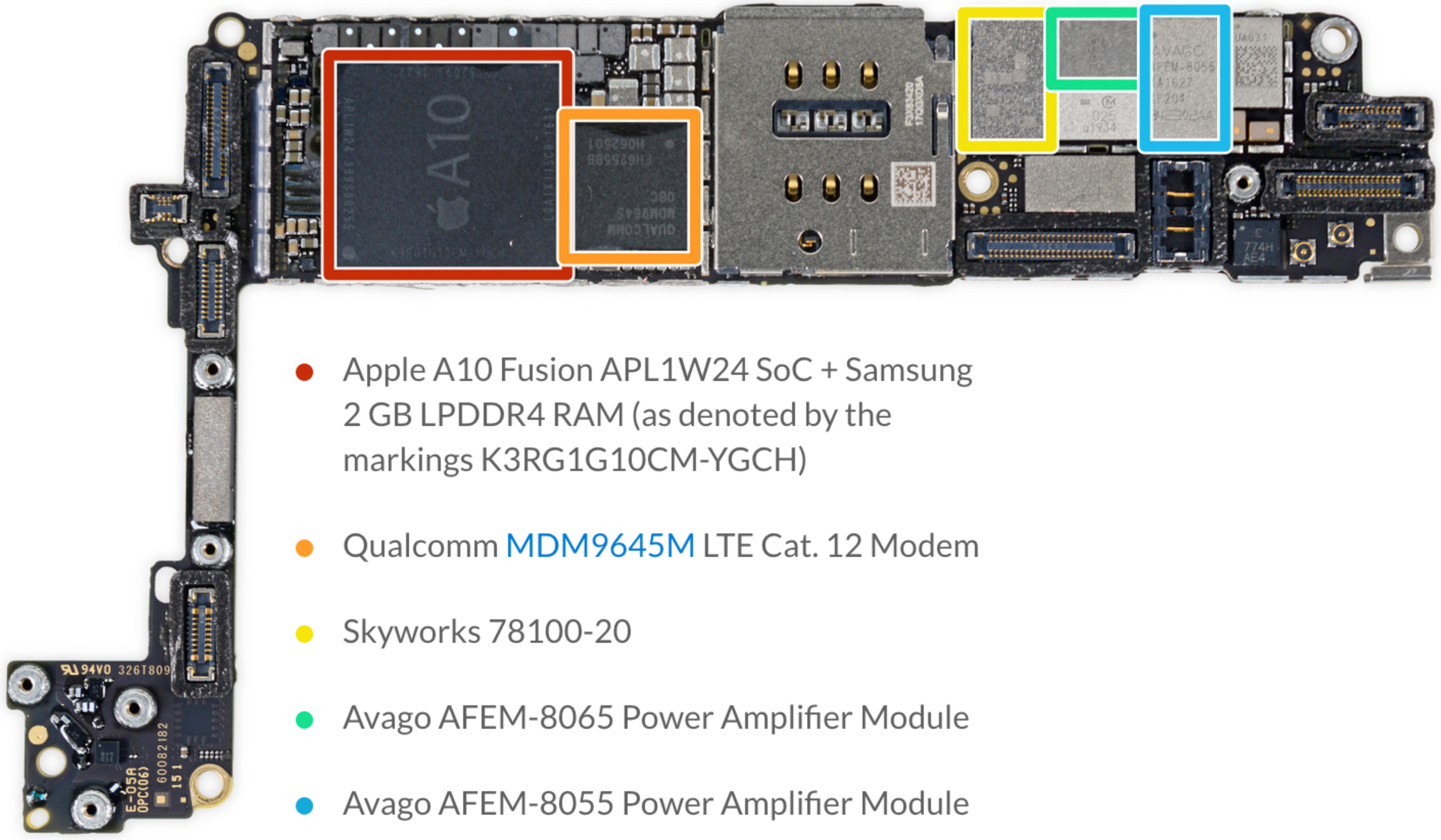
- OH started – use when you need help!
- Questions regarding HW1?

iPhone7 Teardown

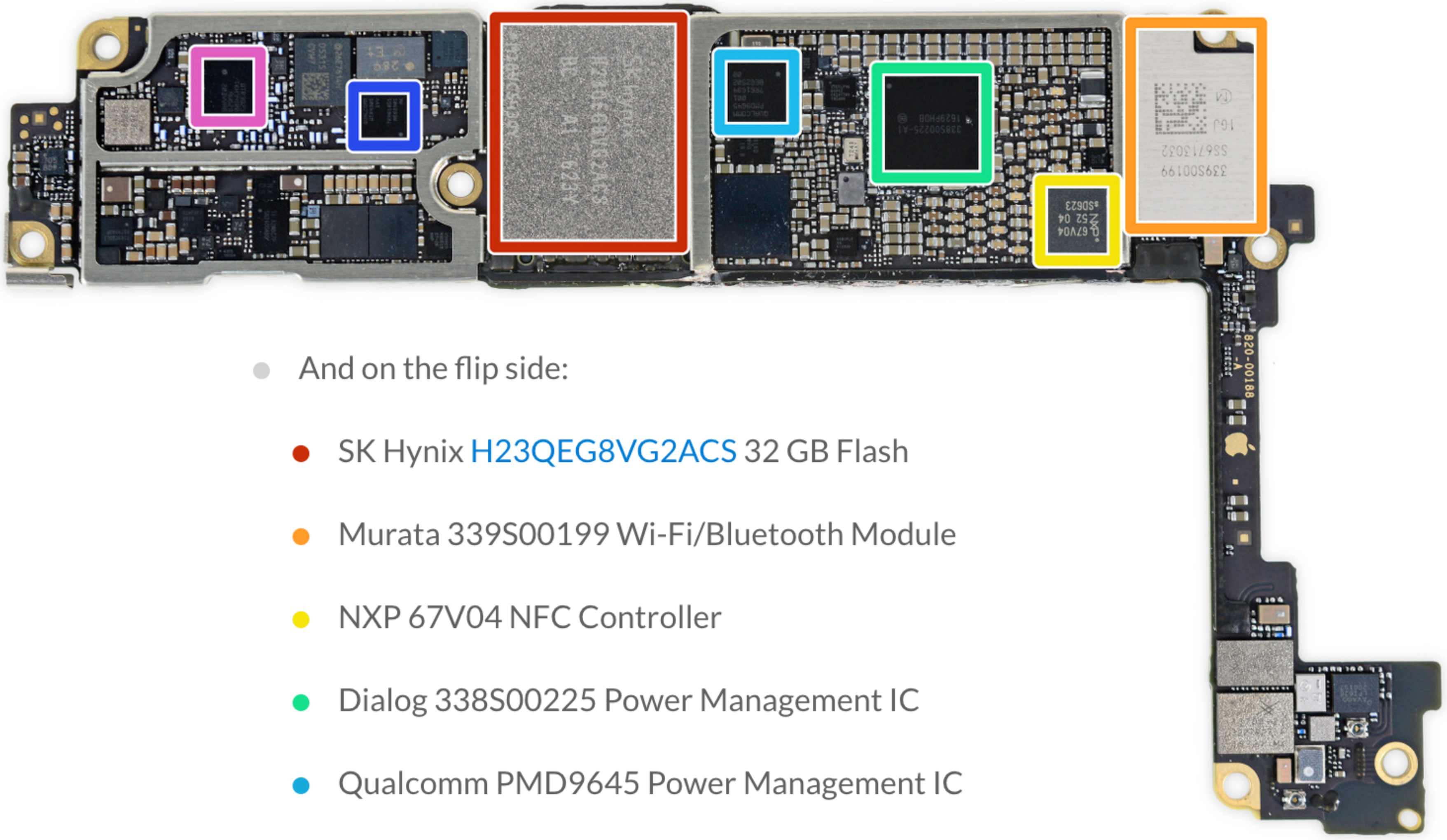
ifixit.com


Li-ion Battery
3.80V ⎓ 7.45Whr
APN: 616-00259
WARNING
Authorized Service Provider Only
Potential for fire or burning. Do not disassemble, puncture, crush, heat, or burn.
TIS 2217-2548
Apple South Asia (Thailand) Limited
c US
E184259 SNH
PSE
1960mAh
Apple Japan
Li-ion
TAPTIC ENGINE

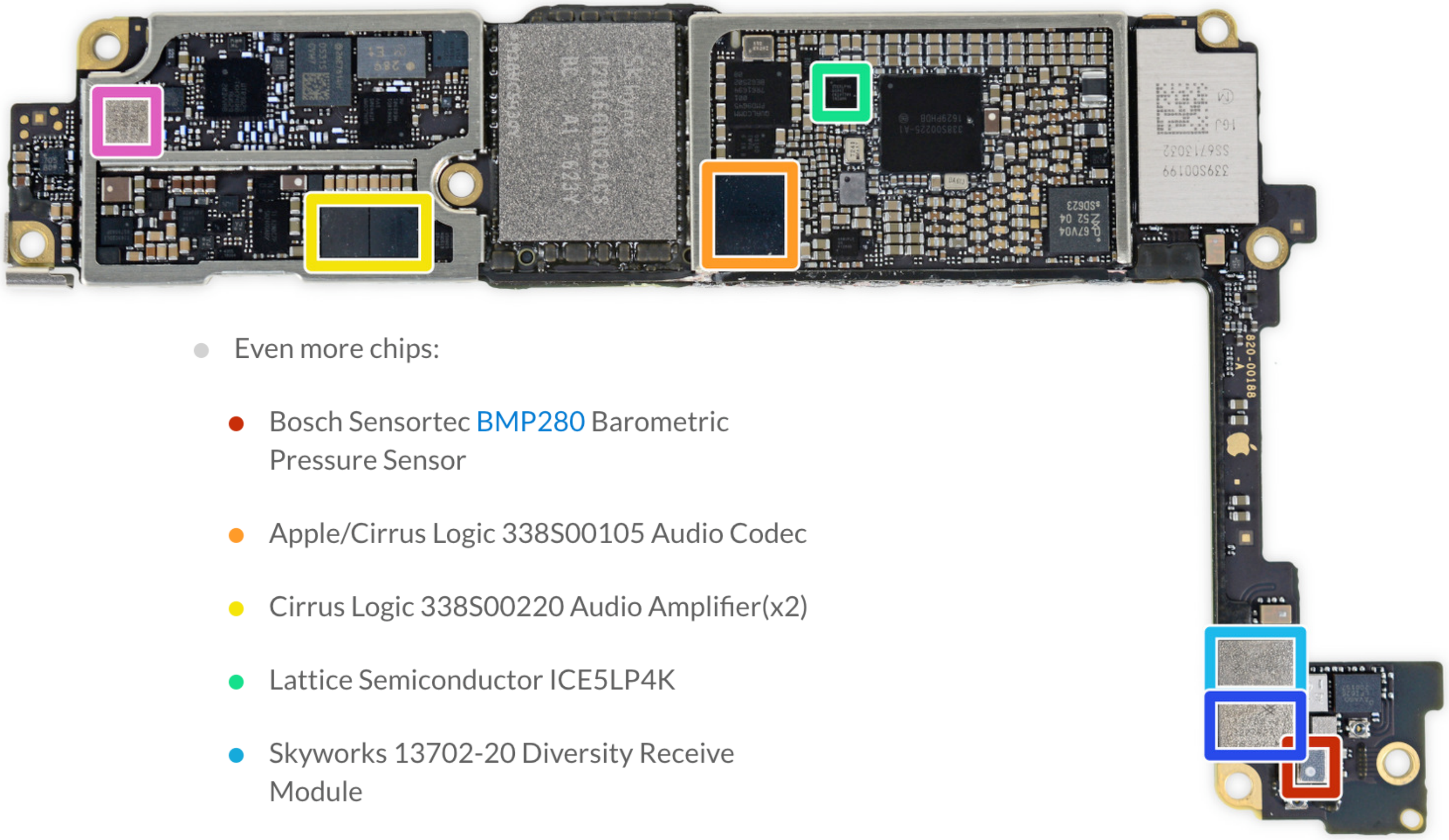
Apple 64bit System on a chip (SoC):
quad core (2 high performance, 2 low power; only 2 at a time)
125 mm^2, 3.3 billion transistors (including the GPU and caches)
2.34 GHz ARMv8 TSMC 16 nm 6-core GPU 4 Samsung LPDDR4 RAM chips



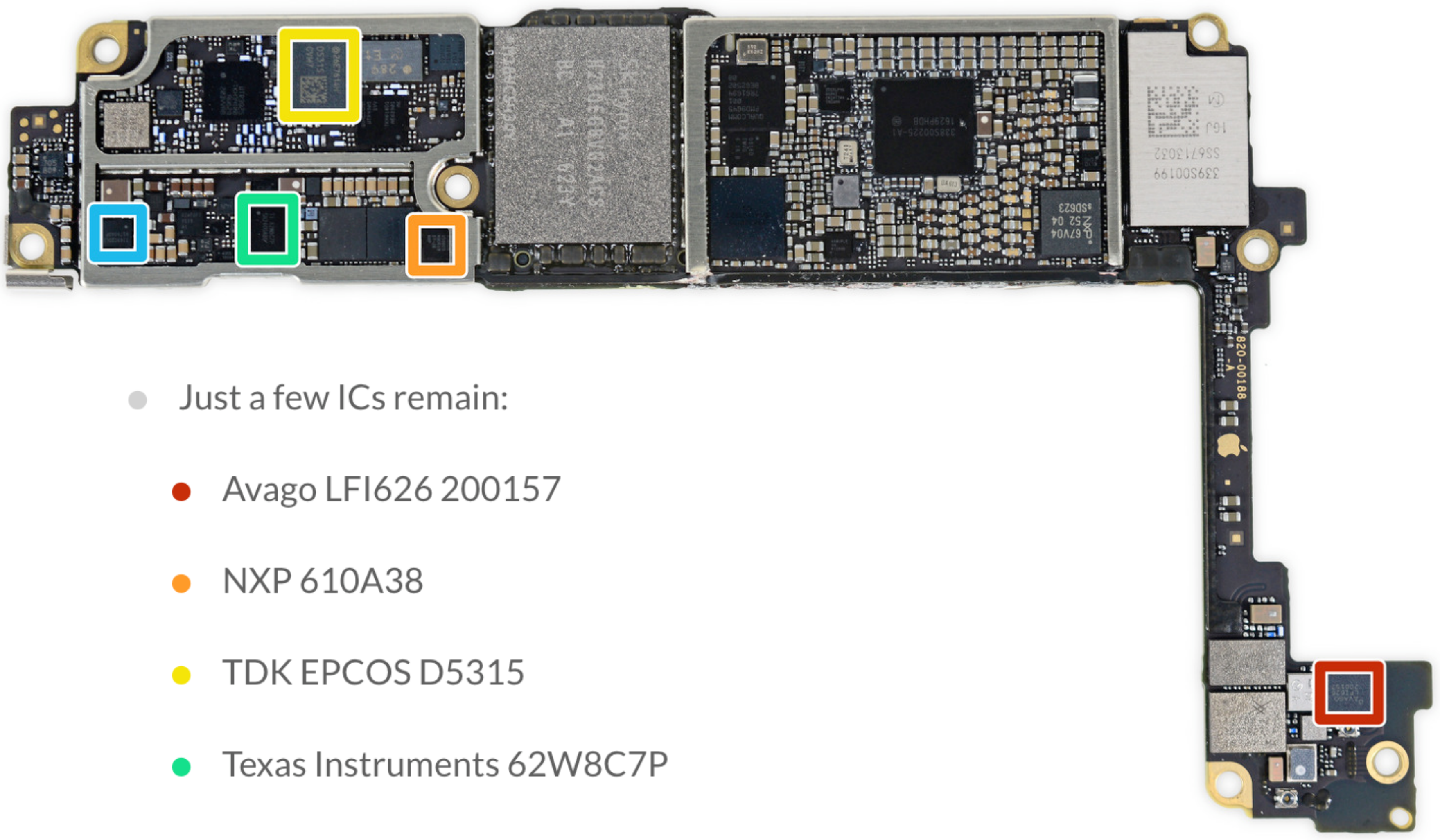
- ● Apple A10 Fusion APL1W24 SoC + Samsung 2 GB LPDDR4 RAM (as denoted by the markings K3RG1G10CM-YGCH)
- ● Qualcomm MDM9645M LTE Cat. 12 Modem
- ● Skyworks 78100-20
- ● Avago AFEM-8065 Power Amplifier Module
- ● Avago AFEM-8055 Power Amplifier Module



- And on the flip side:
  - ● SK Hynix H23QEG8VG2ACS 32 GB Flash
  - ● Murata 339S00199 Wi-Fi/Bluetooth Module
  - ● NXP 67V04 NFC Controller
  - ● Dialog 338S00225 Power Management IC
  - ● Qualcomm PMD9645 Power Management IC
  - ● Qualcomm WTR4905 Multimode LTE Transceiver
  - ● Qualcomm WTR3925 RF Transceiver



- Even more chips:
  - 🟥 Bosch Sensortec BMP280 Barometric Pressure Sensor
  - 🟧 Apple/Cirrus Logic 338S00105 Audio Codec
  - 🟨 Cirrus Logic 338S00220 Audio Amplifier(x2)
  - 🟩 Lattice Semiconductor ICE5LP4K
  - 🟦 Skyworks 13702-20 Diversity Receive Module
  - 🟦 Skyworks 13703-21 Diversity Receive Module
  - 🟪 Skyworks 77363-1



- Just a few ICs remain:
  - 🔴 Avago LFI626 200157
  - 🟠 NXP 610A38
  - 🟡 TDK EPCOS D5315
  - 🟢 Texas Instruments 62W8C7P
  - 🔵 Texas Instruments 65730A0P Power Management IC

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 (0th) 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

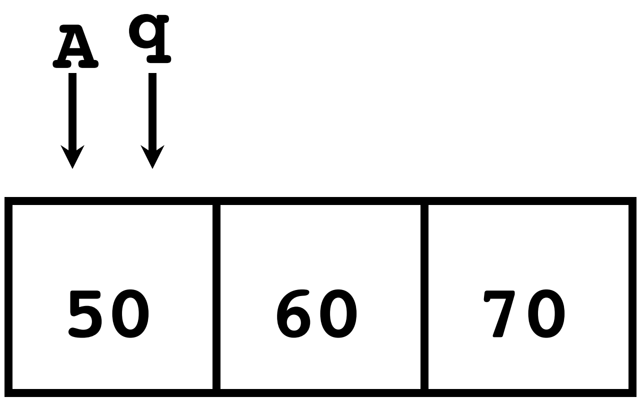
Changing a Pointer Argument?

- 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
```



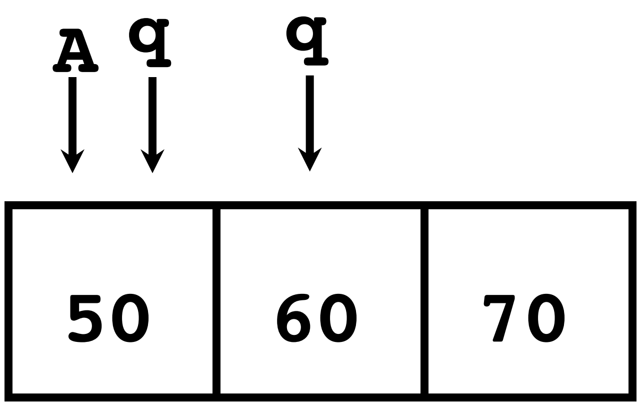
Pointer to a Pointer

- 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);
```

*q = 60



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++){ ... }
    ```

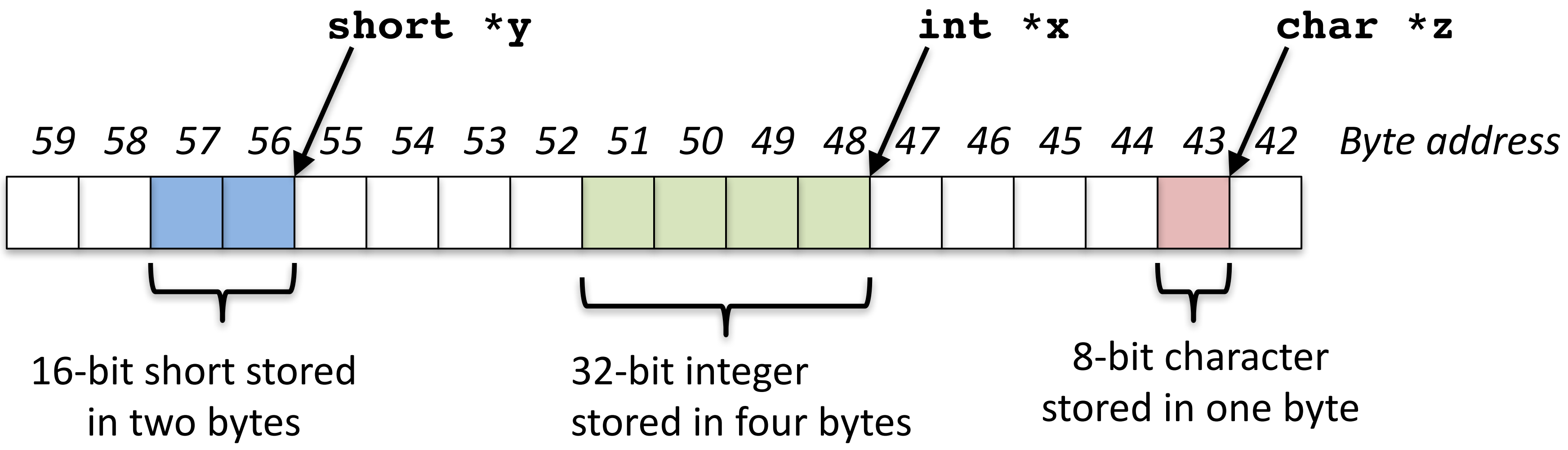
  - Better pattern

    ```
    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



short *y
int *x
char *z
59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42
Byte address
16-bit short stored in two bytes
32-bit integer stored in four bytes
8-bit character stored in one byte

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 <u>something</u> to a pointer

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

p = &a;
p += 1;
```

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

p = &a;
p += 1;
```

In each, p now points to b
(Assuming compiler doesn't reorder variables in memory. ***Never code like this!!!!***)

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

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

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

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:

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? **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? **40**