

# Lecture 3: Pointers

Vikram Iyer

*ca*

Adapted from material by Blake Hannaford

# Administrative

- Lab access- ECE front desk told me everyone should have access, if you can't get in just let them know
- Lab kit pickup  
Mon-Fri: 9am-12pm and 1pm-5pm
- Partner matching form DUE TODAY: <https://bit.ly/cse474-24sp-partners>
- C Programming Assignment 1 posted



# Assignment 1

CSE/ECE 474: Embedded Systems

Home

Syllabus

Calendar

Assignments

Ed Discussion

24sp

## C Programming Assignments

### C Programming 1

Due Apr 12 by 11:59pm on [Canvas](#)

[Instructions](#), [Assignment files \(zip\)](#)

CSE/ECE 474: Embedded Systems

Home

Syllabus

Calendar

Assignments

Ed Discussion

24sp

April				
Monday	Tuesday	Wednesday	Thursday	Friday
9:30-11:30 OH (Deeksha) 01 ECE 345 11:30-13:30 OH (Alex) ECE 345	13:00-15:00 OH (Zach) 02 ECE 345	9:30-11:30 OH (Deeksha) 03 ECE 345 12:30-14:20 Lecture MOR 230 <i>Lecture 3: Pointers in C</i> 14:00-15:00 OH (Vikram) ECE 345	10:00-12:00 OH (Zach) 04 ECE 345 12:30-14:30 OH (Alex) ECE 345	12:30-14:20 Lecture 05 MOR 230 <i>Lecture 4: Number Representation and Bitwise Operators</i> 14:00-15:00 OH (Vikram) ECE 345
9:30-11:30 OH (Deeksha) 08 ECE 345 11:30-13:30 OH (Alex) ECE 345	13:00-15:00 OH (Zach) 09 ECE 345	9:30-11:30 OH (Deeksha) 10 ECE 345 12:30-14:20 Lecture MOR 230 <i>Lecture 5: Hardware and Machine Organization</i> 14:00-15:00 OH (Vikram) ECE 345	10:00-12:00 OH (Zach) 11 ECE 345 12:30-14:30 OH (Alex) ECE 345	12:30-14:20 Lecture 12 MOR 230 <i>Lecture 6: Working with Registers and IMU Demo</i> 14:00-15:00 OH (Vikram) ECE 345 23:59 C Programming 1 due

# Last time

- Intro to C programming
  - Splitting code into multiple files
  - main()
  - Function prototypes
  - Header files
  - Hello World in Arduino and C
  - Variable and function scope

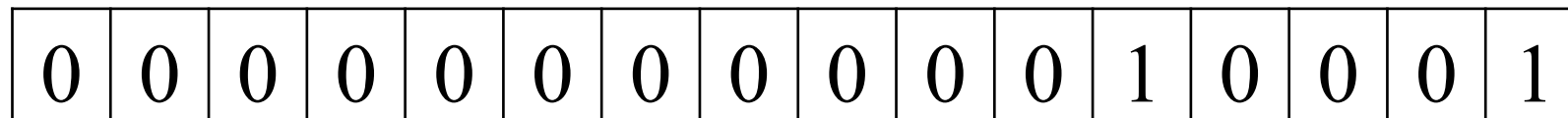
# Plan for today

- Pointers and memory
  - How is your data actually stored in memory?
  - Introduction to memory addresses
  - Pointers and pointer operations

# Data in memory

- The **key** to understanding C code and writing C code is think about data in the computer memory as 0's and 1's.
- What does our data look like?
  - Assuming that the size of an int is 2 bytes (16 bits), the C compiler sets up our declaration as follows:

```
int c_int_variable = 17;
```



# Data in memory

- The **key** to understanding C code and writing C code is think about data in the computer memory as 0's and 1's.
- What does our data look like?
  - Assuming that the size of an int is 2 bytes (16 bits), the C compiler sets up our declaration as follows:

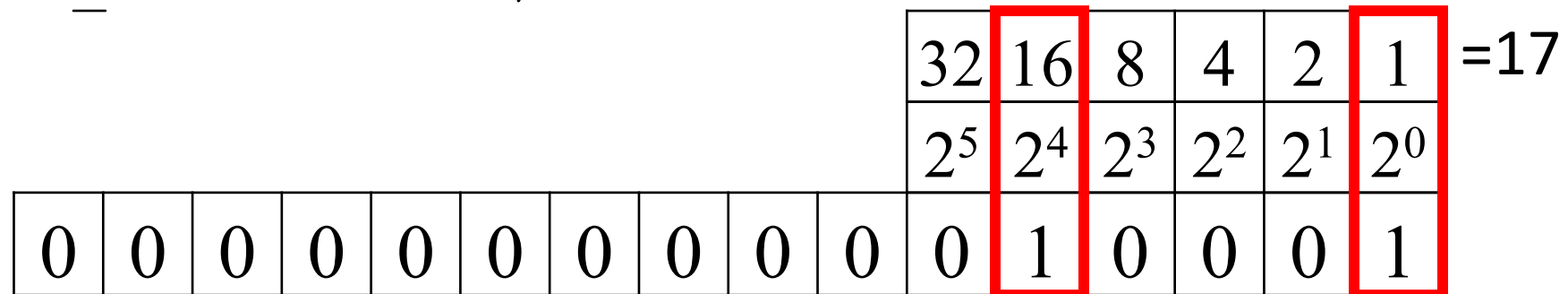
```
int c_int_variable = 17;
```

											32	16	8	4	2	1
											$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1

# Data in memory

- The **key** to understanding C code and writing C code is think about data in the computer memory as 0's and 1's.
- What does our data look like?
  - Assuming that the size of an int is 2 bytes (16 bits), the C compiler sets up our declaration as follows:

```
int c_int_variable = 17;
```



# Data in memory: characters

```
char * c_char_array = "OK";
```

# Data in memory: characters

```
char * c_char_array = "OK";
```

C converts the two characters 'O' and 'K' to 8-bit binary versions in the ASCII (UTF-8) code:

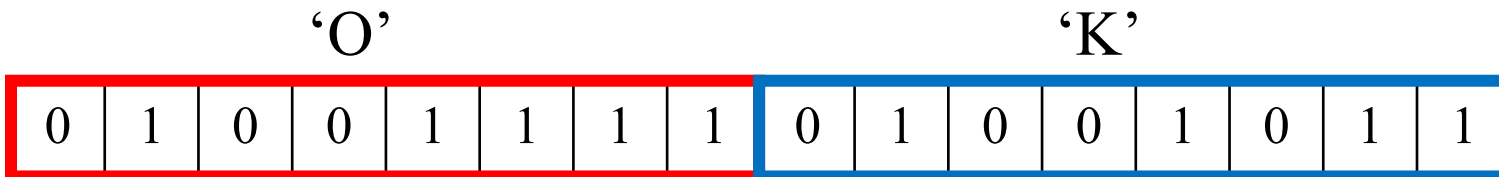
character	numerical ASCII code
'O'	<u>79</u> (dec), 1001111(bin)
'K'	<u>75</u> (dec), 1001011(bin)

# Data in memory: characters

```
char * c_char_array = "OK";
```

C converts the two characters 'O' and 'K' to 8-bit binary versions in the ASCII (UTF-8) code:

character	numerical ASCII code
'O'	79 (dec), 1001111(bin)
'K'	75 (dec), 1001011(bin)



*null terminator*

*0*



# Where does the data get stored?

Example memory location for `c_int_variable`:

2457568: → 

0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Example memory location for `c_char_array`:

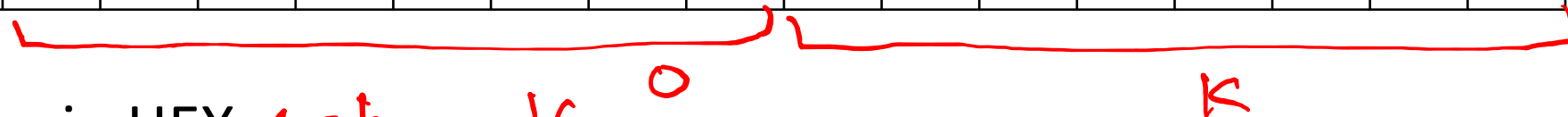
4200984 : → 

0	1	0	0	1	1	1	1	0	1	0	0	1	0	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# Where does the data get stored?

- Another layout with sequential locations

Address	Byte 1								Byte 0							
<b>2457568</b>	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1
<b>2457570</b>	0	1	0	0	1	1	1	1	0	1	0	0	1	0	1	1



- Same thing in HEX ← base 16

Address	Byte 1								Byte 0							
<b>257FE0</b>	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1
<b>257FE2</b>	0	1	0	0	1	1	1	1	0	1	0	0	1	0	1	1

# Where does the data get stored?

- C keeps a table of your variable names (it's called the "symbol table") which for our examples would be:

name	address(hex)	type
<i>IT</i> → c_int_variable	0x257FE0	int
<i>"OK!"</i> c_char_array	0x257FE2	char
c_int_01	0x<some hex addr>	int
c_int_02	0x<some hex addr>	int

# Where does the data get stored?

## What is a pointer?

A variable whose value is the memory address of another variable

- // `my_int_pointer` is a pointer to an `int`

```
int * my_int_pointer;
```

`my_int_pointer` is set up to hold the *address* of an `int`

Address	Byte 1								Byte 0								
<b>257FE0</b>	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	17
<b>257FE2</b>	0	1	0	0	1	1	1	1	0	1	0	0	1	0	1	1	6K

# Assigning value of a pointer

The ampersand (&) symbol refers to the memory address of a variables

```
// make my_int_pointer point to c_int_variable  
int* my_int_pointer = &c_int_variable ;
```

**Stored value**

**257FE0** → 

0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

// the pointer now points to c\_int\_01 instead!

```
my_int_pointer = &c_int_01 ;
```

**Stored value**

**257FE2** → 

0	1	0	0	1	1	1	1	0	1	0	0	1	0	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# Dereferencing a pointer

Dereferencing = getting the value stored in that address with \*

When not used in a declaration (e.g. `int * x = 0x2457854`), asterisk (\*) is known as the “dereferencing operator”.

```
int c = int-variable = 17;    c-int-variable +1;
```

```
// my_int_pointer "points to" c_int_variable
```

```
my_int_pointer = & c_int_variable;
```

```
int x = *my_int_pointer; assigning 17
```

```
if (x == 17) printf(" I understand pointers!\n");
```

Stored value

<b>257FE0</b>	→	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1
---------------	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

*257FE0*

# Examples

```
int *pa = 0x3000; // starting address is 0x3000
*pa = 0x000B; ← update memory address
*(pa++) = 0x0010;
*(pa+1) = 0x00C0;
```

What does memory look like? What are the values of

`*pa + 1 == ??`

`(int)pa == ??`

# Examples

Name	Addr				
	3000	0	0	1	0
pa→	3004	0	0	0	0
	3008	0	0	C	0
	300C	0	0	0	0

\*pa + 1 == 0001

(int)pa == 3004

# Pointer Types and increments

Pointers are typed so that they “know” how big the thing is they point to. Example:

address	byte	byte	byte	byte	
0x257FE0				0x01	=1
0x257FE4				0x02	=2
0x257FE8			0x00	0x10	=16
0x257FEC			0x02	0x01	=513

# Pointer Types and increments

```
int * p1 = 0x257FE0
printf("%d", *p1); // prints 1
printf("%d", *(p1+1) ); // prints 2
// (added 4 to byte address)
printf("%d", *(p1+2) ); // prints 16
// (added 8 to byte address)
```

	address	byte	byte	byte	byte
$p1 \rightarrow$	0x257FE0				0x01
$(p1+1) \rightarrow$	0x257FE4				0x02
$(p1+2) \rightarrow$	0x257FE8			0x00	0x10
$p2 \rightarrow$	0x257FEC			0x02	0x01

*Handwritten notes: A red circle around 0x10 with "=16" next to it. A red bracket under the last two bytes of the last row with an "x" below it.*

C Pointers advance in increments of the thing they point to.

```
int x = 0;
int* p2 = &x;
x = 1
```

*Handwritten note: "Change's what's in memory" with an arrow pointing to the variable x.*

# Typecasting

You can change variables from one type to another by “casting”

```
// "cast" my_int_pointer to instead BE an int  
int x = (int) my_int_pointer;  
print(x); ← 2577 ... memory address
```

Creates an `int` with the address in it and prints it like any other integer.

# Arrays

An array is an ordered set of memory elements.

int j[3] sets up memory for three ints :

j[0], j[1], j[2]

```
int a[5]=0; // array of 5 ints
```

```
           // each int is 2 bytes
```

*↙ pointer*

Element name	a[0]	a[1]	a[2]	a[3]	a[4]
Address	3000	3002	3004	3006	3008

# Initializing Arrays

```
int a[5]; // Declaring an array
```

Enough storage is allocated for 5 integers (typically  $5 \times 32$  bits or 40 bytes). But you should not count on any initial value.

Initializing an array:

Method 1: 

```
int q[3] = {0, 1, 2};
```

Method 2:

```
int i, q[3];  
for (i=0; i<3; i++) q[i] = 0;
```

# Arrays and pointers

Arrays are implemented inside C just like pointers.

Example:

```
int a[10], *p;
```

```
p = &a[0];
```

*← memory address of first element*

a[3] <-----> \* (p+3) are exactly the same

a <-----> &a[0]

are exactly the same

# Generic (void) Pointers

Sometimes we want a pointer which is not locked to a specific type and can potentially point to anything.

```
void *name ; // declare a generic pointer called name
```

- name can point to anything in the computer.
- name cannot be dereferenced with \*
- Must instead assign value of void pointer to a pointer of the type you want.

# Generic (void) Pointers

## Example:

```
void* myGenericPtr;  
int t, *ip, myvalue = 3;  
myGenericPtr = &myvalue; // OK  
t = *myGenericPtr; // NO!!  
  
//*****  
  
ip = myGenericPtr;  
t = *ip; // OK!!
```

# Null pointers

If a pointer has the value `NULL`, it points to nothing. `NULL` is a predefined constant in `<stddef.h>` or use `#define NULL 0`

- `NULL` is illegal to dereference.
- `NULL` can be tested for:
- `int i,*ip = NULL;`

```
if(ip == NULL) {  
    // I haven't defined ip yet  
}  
else { // OK, now I can use it!
```

# Function pointers

C can have pointers to functions.

```
type (* functionpointer) (arg list)
```

## Examples

```
int (*IntFuncPtr) ();
```

```
\\ IntFuncPtr is a pointer to a function with  
\\ no arguments which returns an int
```

```
double (*doubleFuncPtr) (int arg1, char arg2)
```

```
\\ doubleFuncPtr is a pointer to a function with  
\\ an int and a char arg which returns a double.
```

# Function pointers

Assignment to function pointers:

```
int (* IFP)(int x) = NULL; // empty function pointer
int realfunction( int x); // an actual function

IFP = &realfunction;
IFP = realfunction;    // can skip &
```

Dereferencing function pointers:

```
(*IFP)(5)    // call function realfunction with arg 5

IFP(5)       // can also use function pointer just like
              // original function name
```

# Pointer arithmetic

A powerful feature of pointers is the ability to compute with them like ints, but only some operations are allowed with pointers.

## Allowed:

- Add a scalar to a pointer
- Subtract pointers

*Scalars*

## Not Allowed

- Adding two pointers
- Multiplying or dividing
- Multiplying /dividing by scalar

# Pointer arithmetic

## Example: Find the midpoint in an array.

```
#define SIZE 100
int length, buffer[SIZE]; int *ptr1,
*ptr2, *ptr3;
ptr1 = buffer;           // points to start of array
ptr2 = ptr1 + 100;      // points to end of array
length = ptr2 - ptr1;   // length = 100
ptr3 = ptr1 + length/2  // ptr3 points to mid-point of array
```

## Pointer comparisons

- `==, !=` Determine if two pointers are equal or not.
- `<, <=, >=, >` Which pointer points to a higher address in memory?  
Which way will subtraction come out?