Lecture 3: Pointers

Vikram Iyer

Adapted from material by Blake Hannaford

Administrative

- Office hours and website updated

CSE/ECE 474: Embedded Systems | Home | Syllabus | Calendar | Ed Discussion | 23sp

CSE/ECE 474: Introduction to Embedded Systems- Spring 2023

Instructor: Vikram Iyer (vsiyer [at] cs.washington.edu)
Teaching Assistants: Kyle Johnson, Vicente Arroyos, Joe Breda

Lectures: 12:30-1:50, Wed/Fri in MOR 230, Recordings (Login required)
Lab and Office Hours: EEB 345

Assignment Submission and Grades: Canvas
Contact Info: Please use Ed Discussion whenever possible. This is the fastest way to get an answer to your question from the course staff or even others who have had the same issue. The answer to your question is likely to be helpful to others in the class. Feel free to use private messages on the discussion board for questions that contain detailed code or should not be shared with the rest of the class. For other questions, including unexpected emergencies or other personal circumstances, email the instructor directly.

Lab kit pickup (EE Store- EEB 137):

- MONDAY: 12:00 - 2:00
- TUESDAY: 11:30 - 5:30
- WEDNESDAY: 12:00 - 2:00
- THURSDAY: 10:00 - 2:00
- FRIDAY: 10:00 - 12:00

Lecture Topics

Date	Description

Administrative

- Office hours and website updated

CSE/ECE 474: Embedded Systems | Home | Syllabus | Calendar | Ed Discussion | 23sp

Schedule

Subscribe to this calendar (google, iCal, etc.)

March				
Monday	**Tuesday**	**Wednesday**	**Thursday**	**Friday**
27 09:00-11:00 OH (Vicente) ECE 345 13:30-15:30 OH (Joe) ECE 345	28 09:00-11:00 OH (Kyle) ECE 345	29 09:00-11:00 OH (Vicente) ECE 345 12:30-13:50 Lecture MOR 230 *Course overview* slides	30 09:00-11:00 OH (Kyle) ECE 345 13:30-15:30 OH (Joe) ECE 345	31 12:30-13:50 Lecture MOR 230 *Intro to C Programming* slides, code

April				
Monday	**Tuesday**	**Wednesday**	**Thursday**	**Friday**
03 09:00-11:00 OH (Vicente) ECE 345 13:30-15:30 OH (Joe) ECE 345	04 09:00-11:00 OH (Kyle) ECE 345	05 09:00-11:00 OH (Vicente) ECE 345 12:30-13:50 Lecture MOR 230 *Pointers in C*	06 09:00-11:00 OH (Kyle) ECE 345 13:30-15:30 OH (Joe) ECE 345	07 12:30-13:50 Lecture MOR 230 *Number Representation and Bitwise Operators*
10 09:00-11:00 OH (Vicente) ECE 345 13:30-15:30 OH (Joe) ECE 345	11 09:00-11:00 OH (Kyle) ECE 345	12 09:00-11:00 OH (Vicente) ECE 345 12:30-13:50 Lecture MOR 230 *Hardware and Machine Organization* 23:59 HW1 due	13 09:00-11:00 OH (Kyle) ECE 345 13:30-15:30 OH (Joe) ECE 345	14 12:30-13:50 Lecture MOR 230 *Working with Registers*

Assignment 1

- Assignment 1 due next Wed 4/12
- Currently on Canvas, will be on the website as well after class
- Turn in on Canvas

canvas
Account
Dashboard
Courses
Calendar
Inbox
History
Help

CSE 474 A › Assignments › C Programming 1

Spring 2023

Home
Announcements
Assignments
Quizzes
Grades
Panopto Recordings
Files
People
UW Resources

C Programming 1

Start Assignment

Due Apr 12 by 11:59pm **Points** 20 **Submitting** a file upload **File Types** c

Instructions

Assignment files

C Programming 1

Criteria	Ratings				Pts
Looping, String Manipulation, Basic Arithmetic	4 pts Full Marks	3.2 pts Mostly working	2 pts Partially working	0 pts No Marks	4 pts
Compact Data Representation, Memory Manipulation, Pointers & Arrays	14 pts Full Marks	11.2 pts Mostly Working	7.47 pts Partially working	0 pts No Marks	14 pts
Compile warnings and coding style	2 pts Full Marks	1.2 pts Some Warnings	0.8 pts Many Warnings	0 pts No Marks	2 pts

Total Points: 20

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

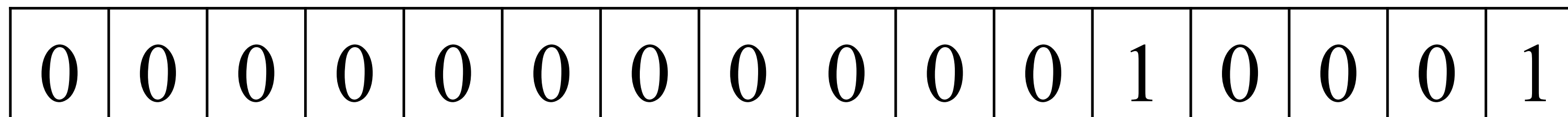
Plan for today

- 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	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;
```

										32	16	8	4	2	1	=17
										2^5	2^4	2^3	2^2	2^1	2^0	
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	

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 ASCI (UTF-8) code:

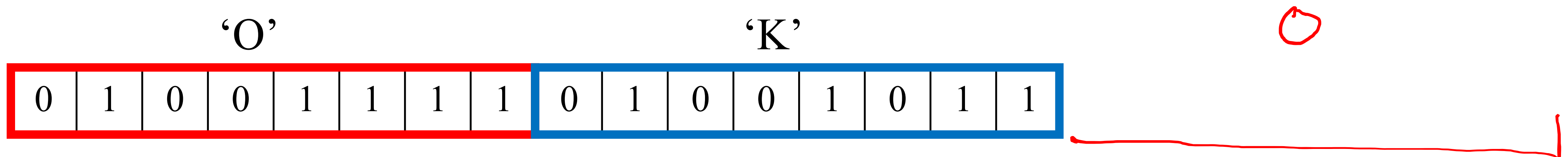
character	numerical ASCII code
‘O’	79 (dec), 1001111(bin)
‘K’	75 (dec), 1001011(bin)

Data in memory: characters

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

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

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



Where does the data get stored?

17

Example memory location for `c_int_variable`: +1

memory address

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							
2457568	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1
2457570	0	1	0	0	1	1	1	1	0	1	0	0	1	0	1	1

- Same thing in HEX

Address	Byte 1								Byte 0							
257FE0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1
257FE2	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
c_int_variable	0x257FE0	int
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?

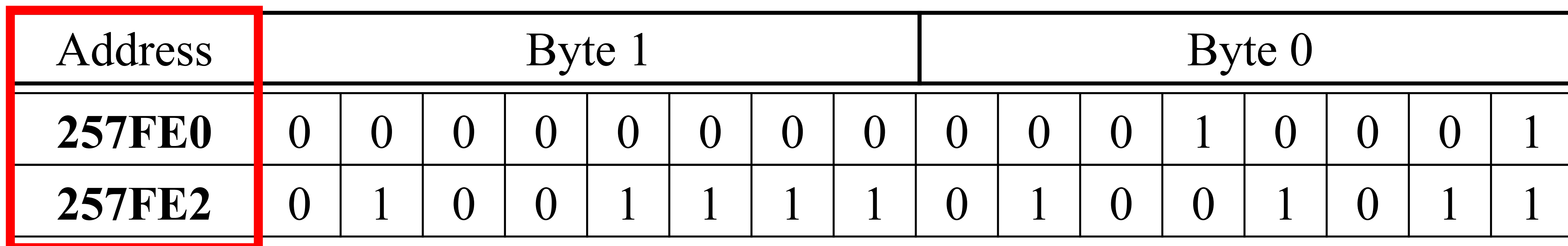
What is a pointer?

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							
257FE0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1
257FE2	0	1	0	0	1	1	1	1	0	1	0	0	1	0	1	1

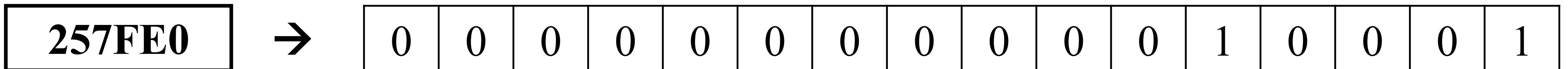
Assigning value of a pointer

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 ;
```

get the memory address

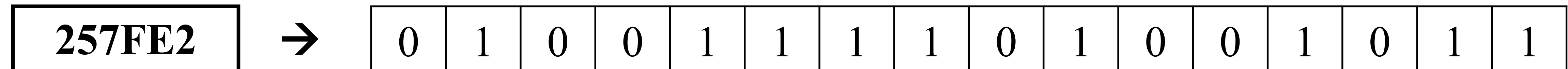
Stored value




```
// the pointer now points to c_int_01 instead!
my_int_pointer = &c_int_01 ;
```

257FE0

Stored value



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”.

```
// my_int_pointer "points to" c_int_variable
my_int_pointer = & c_int_variable;
int x = *my_int_pointer;
if (x == 17) printf(" I understand pointers!\n");
```

Stored value

257FE0	→	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1

Examples

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

What does memory look like? What are the values of

dereferencing

```
*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

+1

```
*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
0x257FE0				0x01
0x257FE4				0x02
0x257FE8			0x00	0x10
0x257FEC			0x02	0x01

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

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);
```

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.

number of ints to store

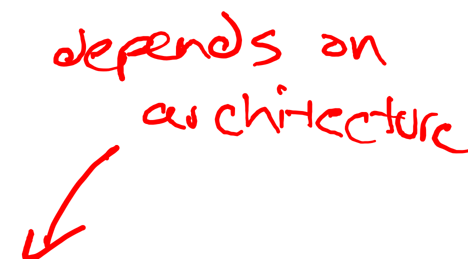
`int j[3]` sets up memory for three ints :

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

```
int a[5]=0; // array of 5 ints
            // each int is 2 bytes
```

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 5×32 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];


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!!
```

know type = int

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

return type → no inputs

```
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

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?