

COMP 206 – Intro to Software Systems

Lecture 8 – Working with Strings

Sept 26, 2018

Plan today

- Understand how text data is stored
 - How to represent single characters
 - How to build them into strings
- Practice doing useful things with strings
 - Via basic C commands on the data itself
 - Using pointers.

Recall, different variable types. Char for text.

Description	Type	Bits	Range
Integer	short	16	+/-32 thousand
	int	32	+/-2.1 billion
	long	64	+/- 9.2 x 10 ¹⁸
Floating point	float	32	+/- 10 ³⁸
	double	64	+/- 10 ³⁰⁸
	long double	128	+/- 10 ⁴⁹³²
Character	char	8	-127 to 128
	unsigned char		0 to 255
Pointer	char*	64	0 to 1.8 x 10 ¹⁹
	int* (etc)		

ASCII TABLE

Memory
is binary

- Char type is really an 8-bit integer
- The mapping to text characters is defined by a standard!

Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]	48	30	110000	60	0	96	60	1100000	140	`
1	1	1	1	[START OF HEADING]	49	31	110001	61	1	97	61	1100001	141	a
2	2	10	2	[START OF TEXT]	50	32	110010	62	2	98	62	1100010	142	b
3	3	11	3	[END OF TEXT]	51	33	110011	63	3	99	63	1100011	143	c
4	4	100	4	[END OF TRANSMISSION]	52	34	110100	64	4	100	64	1100100	144	d
5	5	101	5	[ENQUIRY]	53	35	110101	65	5	101	65	1100101	145	e
6	6	110	6	[ACKNOWLEDGE]	54	36	110110	66	6	102	66	1100110	146	f
7	7	111	7	[BELL]	55	37	110111	67	7	103	67	1100111	147	g
8	8	1000	10	[BACKSPACE]	56	38	111000	70	8	104	68	1101000	150	h
9	9	1001	11	[HORIZONTAL TAB]	57	39	111001	71	9	105	69	1101001	151	i
10	A	1010	12	[LINE FEED]	58	3A	111010	72	:	106	6A	1101010	152	j
11	B	1011	13	[VERTICAL TAB]	59	3B	111011	73	;	107	6B	1101011	153	k
12	C	1100	14	[FORM FEED]	60	3C	111100	74	<	108	6C	1101100	154	l
13	D	1101	15	[CARRIAGE RETURN]	61	3D	111101	75	=	109	6D	1101101	155	m
14	E	1110	16	[SHIFT OUT]	62	3E	111110	76	>	110	6E	1101110	156	n
15	F	1111	17	[SHIFT IN]	63	3F	111111	77	?	111	6F	1101111	157	o
16	10	10000	20	[DATA LINK ESCAPE]	64	40	1000000	100	@	112	70	1110000	160	p
17	11	10001	21	[DEVICE CONTROL 1]	65	41	1000001	101	A	113	71	1110001	161	q
18	12	10010	22	[DEVICE CONTROL 2]	66	42	1000010	102	B	114	72	1110010	162	r
19	13	10011	23	[DEVICE CONTROL 3]	67	43	1000011	103	C	115	73	1110011	163	s
20	14	10100	24	[DEVICE CONTROL 4]	68	44	1000100	104	D	116	74	1110100	164	t
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]	69	45	1000101	105	E	117	75	1110101	165	u
22	16	10110	26	[SYNCHRONOUS IDLE]	70	46	1000110	106	F	118	76	1110110	166	v
23	17	10111	27	[ENG OF TRANS. BLOCK]	71	47	1000111	107	G	119	77	1110111	167	w
24	18	11000	30	[CANCEL]	72	48	1001000	110	H	120	78	1111000	170	x
25	19	11001	31	[END OF MEDIUM]	73	49	1001001	111	I	121	79	1111001	171	y
26	1A	11010	32	[SUBSTITUTE]	74	4A	1001010	112	J	122	7A	1111010	172	z
27	1B	11011	33	[ESCAPE]	75	4B	1001011	113	K	123	7B	1111011	173	{
28	1C	11100	34	[FILE SEPARATOR]	76	4C	1001100	114	L	124	7C	1111100	174	
29	1D	11101	35	[GROUP SEPARATOR]	77	4D	1001101	115	M	125	7D	1111101	175	}
30	1E	11110	36	[RECORD SEPARATOR]	78	4E	1001110	116	N	126	7E	1111110	176	~
31	1F	11111	37	[UNIT SEPARATOR]	79	4F	1001111	117	O	127	7F	1111111	177	[DEL]
32	20	100000	40	[SPACE]	80	50	1010000	120	P					
33	21	100001	41	!	81	51	1010001	121	Q					
34	22	100010	42	"	82	52	1010010	122	R					
35	23	100011	43	#	83	53	1010011	123	S					
36	24	100100	44	\$	84	54	1010100	124	T					
37	25	100101	45	%	85	55	1010101	125	U					
38	26	100110	46	&	86	56	1010110	126	V					
39	27	100111	47	'	87	57	1010111	127	W					
40	28	101000	50	(88	58	1011000	130	X					
41	29	101001	51)	89	59	1011001	131	Y					
42	2A	101010	52	*	90	5A	1011010	132	Z					
43	2B	101011	53	+	91	5B	1011011	133	[
44	2C	101100	54	,	92	5C	1011100	134	\					
45	2D	101101	55	-	93	5D	1011101	135]					
46	2E	101110	56	.	94	5E	1011110	136	^					
47	2F	101111	57	/	95	5F	1011111	137	_					

C Code to work with single characters

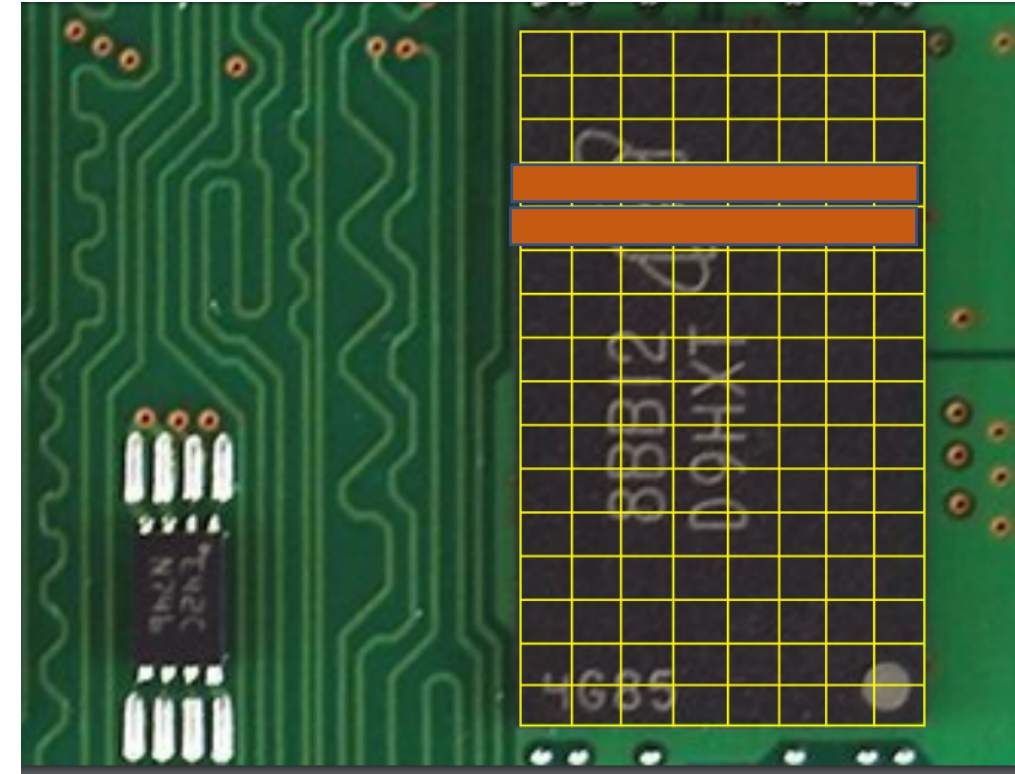
- Single quotes for literals:
- Math allows moving alphabetically forward or backwards, finding relative positions
- Logic works via alphabetical order
- `char char_variable = 'w';`
- `char_variable++;` (it now = 'x');
- `char_variable - 'a'` (tells you what position in alphabet, 23 here)
- `char_variable == 'x'` (evals true)
- `char_variable > 'z'` (evals false)

C Strings

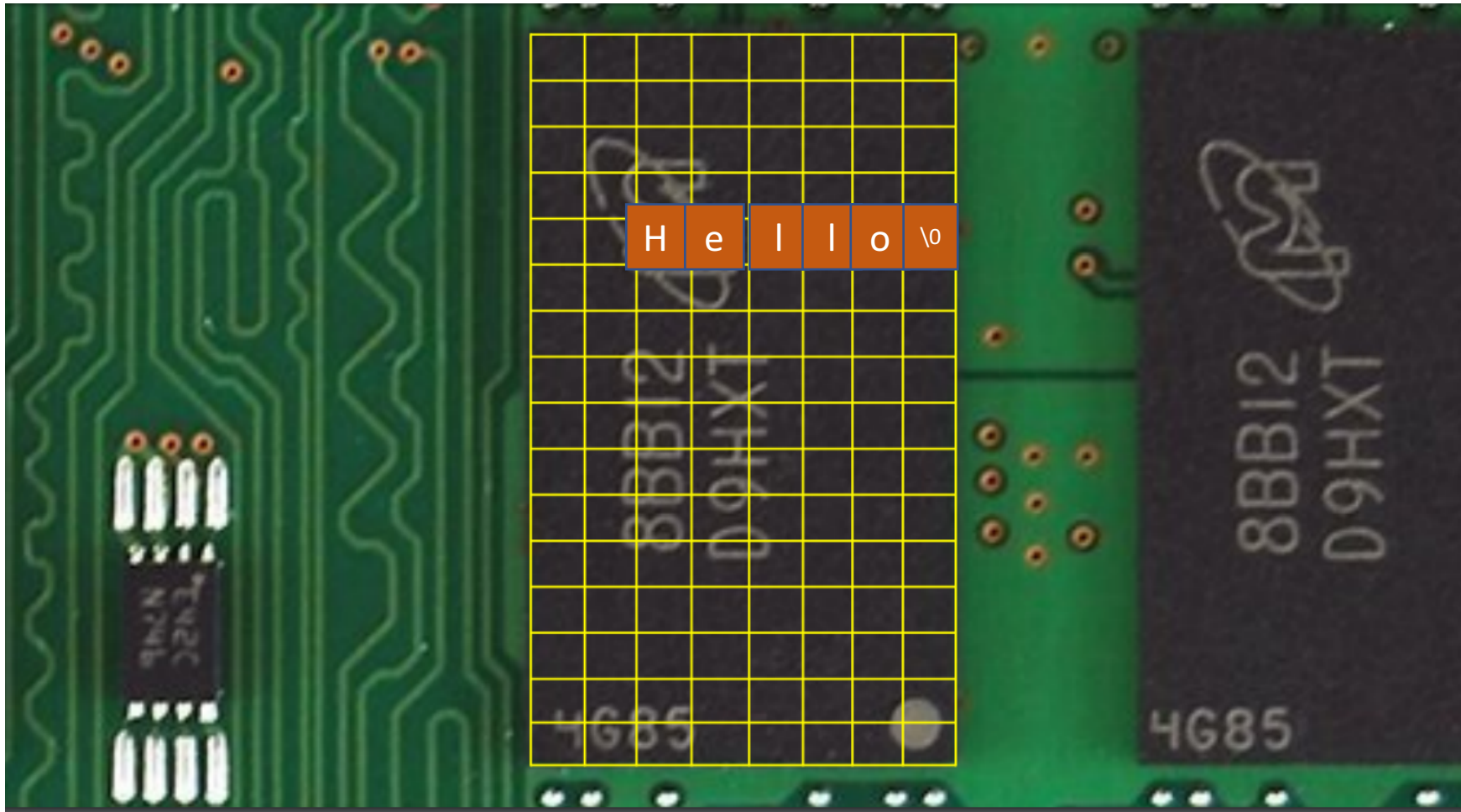
- Arrays of characters
 - E.g. `char name[100] = "David";`
- Each element is a character stored using the ASCII table
 - A mapping between our printable letters and the 0's and 1's in memory
- Must be "null terminated" with the special `'\0'` NULL value, with ASCII integer representation 0
- Claim: Without the `\0`, `printf` could not work.
 - Do you agree? Why or why not?

Arrays and Memory

- Arrays take multiple “slots”, each of the underlying type
 - E.g. `float array[2];`
- They are guaranteed to be stored in order
- The number as you specify on array creation does not change



Strings in memory



C Code to work with char arrays

- Double quotes for literals:
- Doesn't work with math
- Logic operators don't work
- `char str_var[100] = "hello";`
- `str_var++;` (error);
- `str_var - "jello"` (nothing related to 'h' - 'j')
- `str_var == "hello"` (incorrect)
- `str_var > "jello"` (incorrect)

So what are the correct operations? Break down the string into its characters

- To compare or change a string, we want to think about the characters inside. We will typically write loops that iterate char by char
- Let's build up to that, first off, print a string one char at a time:

```
char str_var[100] = "hello";  
for( int pos=0; pos<100; pos++ ){  
    if( str_var[pos] == '\0' ) break;  
    printf( "%c", str_var[pos] );  
}
```

Building up, what about finding length?

```
#include <stdio.h>

int main(){

    char str_var[100] = "world";

    int length = 0;
    for( int pos=0; pos<100; pos++ ){

        if( str_var[pos] == '\0' ) break;
        length++;
    }

    printf( "The string %s has length %d.\n", str_var, length );

    return 0;
}
```

Danger Young Programmer!

Don't forget the \0

- We get \0 at the end of our string for free when we create it with the “...” syntax.
- This can lead us to start forgetting to add it when we create, e.g.,
 - `char str[10];`
 - `str[0] = 'h';`
 - `str[1] = 'i';`
- Hopefully you can see from our previous examples, this is not safe, we can get the wrong length and may print garbage data.

Danger Young Programmer!

Don't forget the \0

- We get \0 at the end of our string for free when we create it with the “...” syntax.
- This can lead us to start forgetting to add it when we create, e.g.,
 - char str[10];
 - str[0] = 'h';
 - str[1] = 'i';
 - str[2] = '\0';
- Hopefully you can see from our previous examples, this is not safe, we can get the wrong length and may print garbage data.



Correction!

More tools

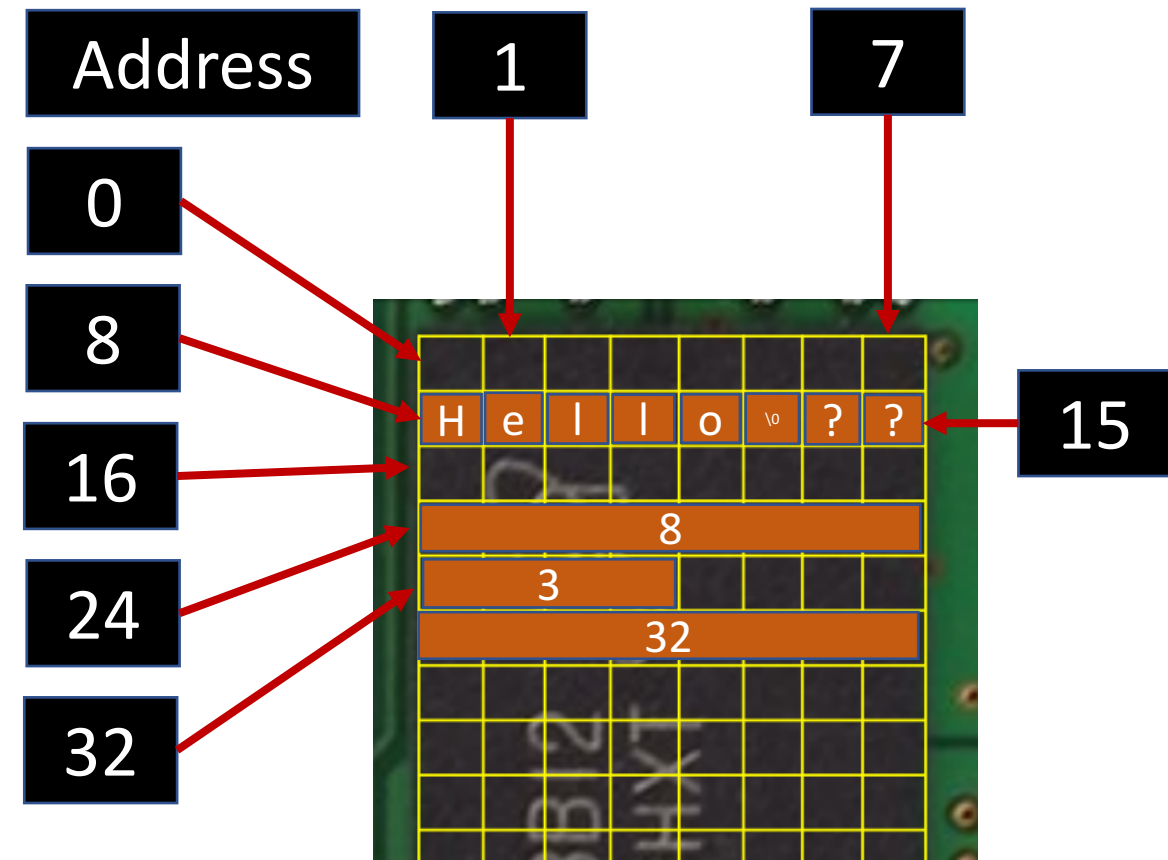
- You can do everything you'd like with the array representation of strings
- However, they become a bit difficult to work with as we always need to keep the array index. Breaking up the string and moving around different parts of it requires many variables for book-keeping.
- The "Thinking in C" approach starts now, it's easier to work with addresses!

C strings are also pointers.

Wait... what is a pointer?

- A pointer is variable that stores an address.
- Pointers allow us to move around our strings (think iterators, lists indices):

```
char str_var[100] = "Hello";  
char *start = str_var;  
char *mid = str_var+3;
```
- In future lectures: we will think about how pointers can be used with other types of variables



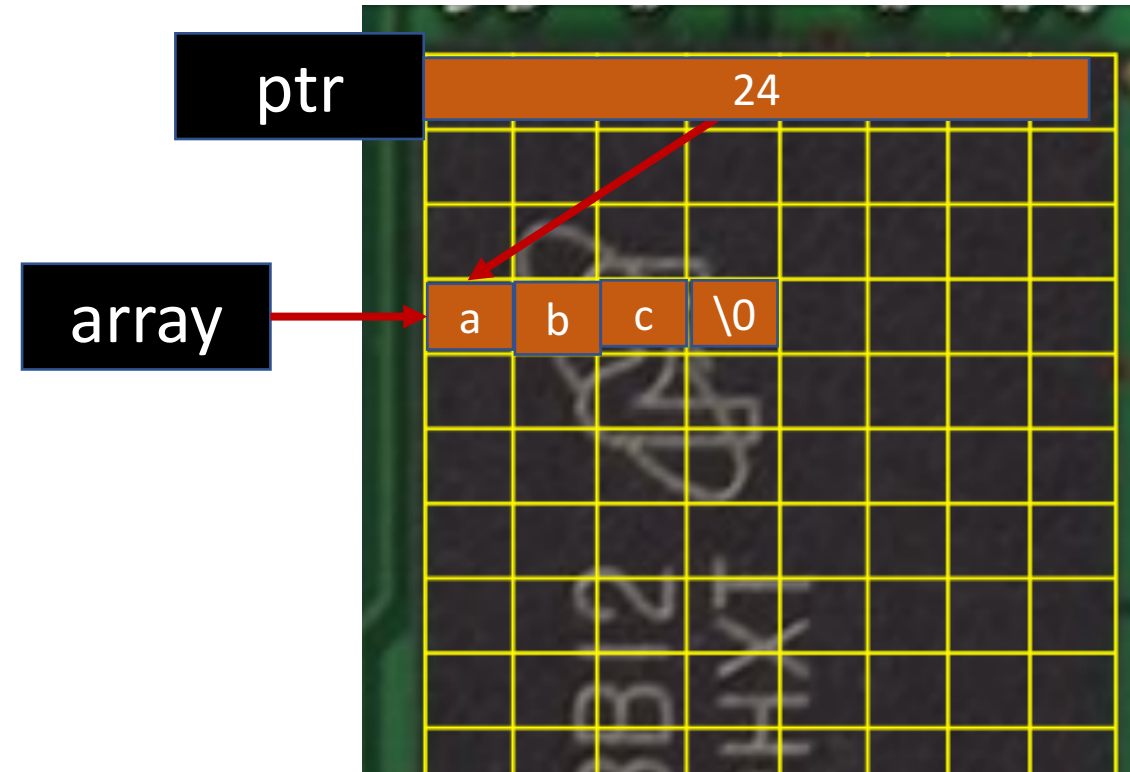
Note: all data is of course binary, but we display chars or ints in each byte to make life easier.

C Syntax for pointers

- Declare a non-pointer variable with:
 - `TYPE VARNAME;`
- Declare a pointer with:
 - `TYPE * VARNAME;`
 - Star can be anywhere between the type and var
 - `VARNAME` holds the value “the address of a variable or array of `TYPE`”

Pointers and Arrays

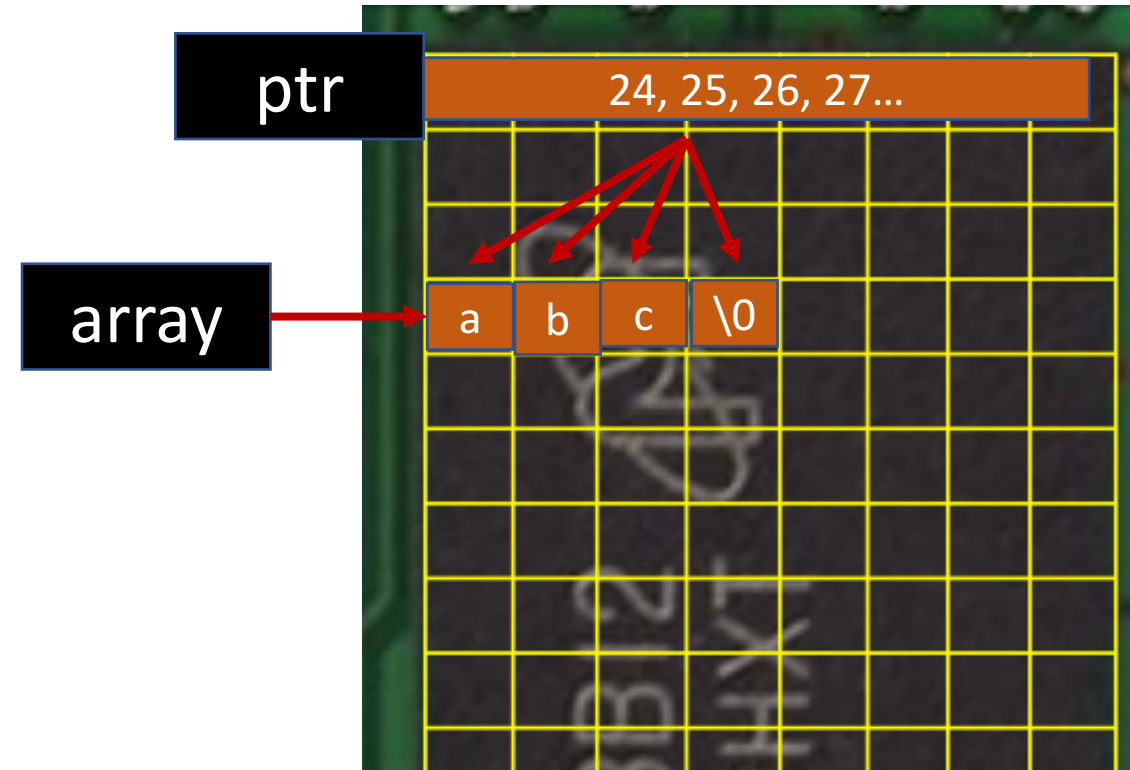
- In some ways, they are interchangeable
- An array in C is implemented as the address of its first entry. So, we “point to” the array
- Example:
 `char array[4] = “abc”;`
 `char *ptr = array;`



Pointers and Arrays

- In some ways they differ:
- The array variable holds the address to the start of this memory always, while the pointer is more flexible
- Example:

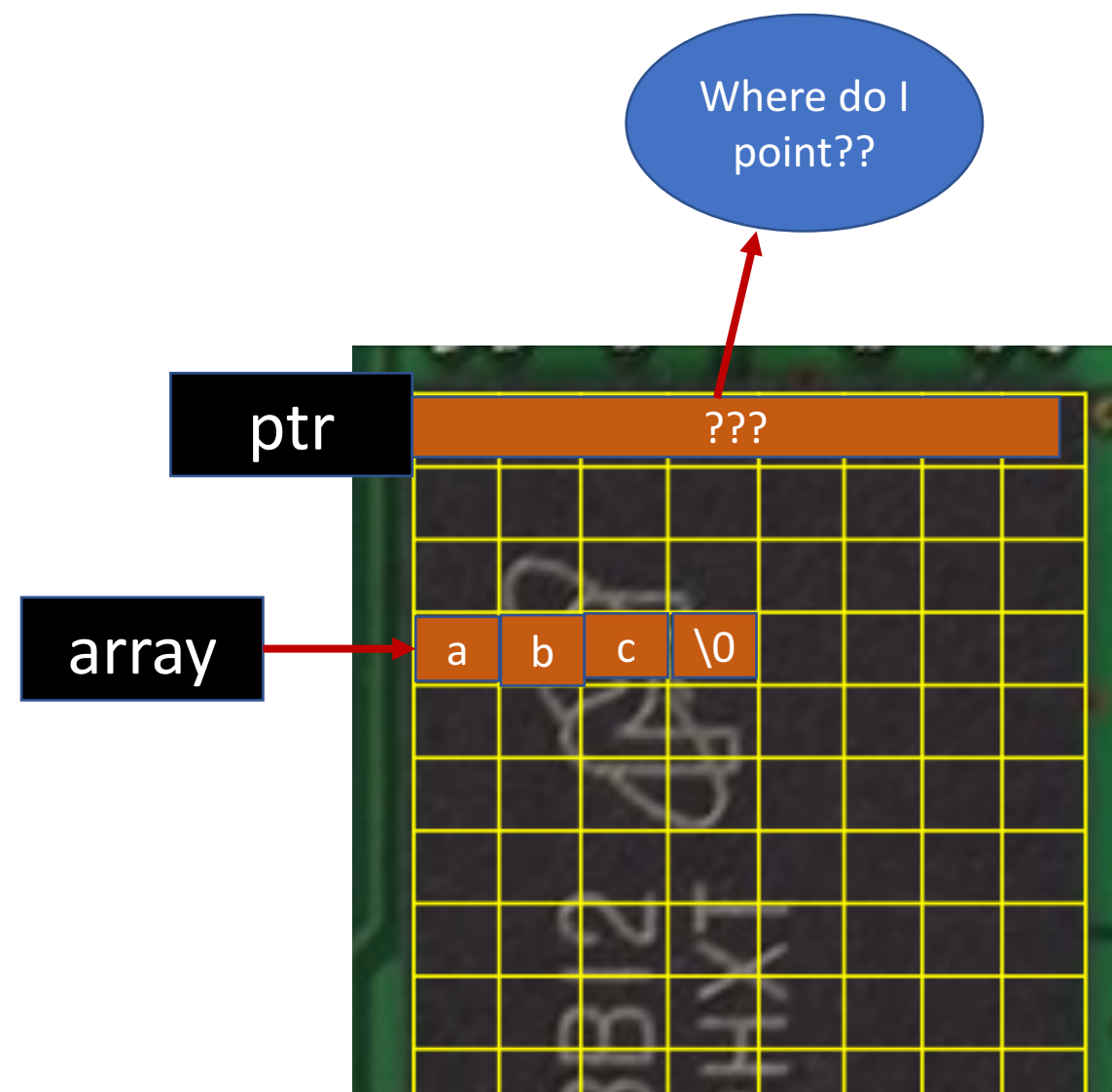
```
char array[4] = "abc";  
char *ptr = array;  
ptr++; ptr++; ptr++; ptr++; // These work  
  
array++; // This is an error!
```



Pointers and Arrays

- In some ways they differ:
 - An array says how much memory it requires at the beginning
 - A ptr declaration alone does not request memory to store real data
- Example:

```
char array[4];  
array[0] = 'a'; // This line is OK  
  
Char *ptr;  
ptr[0] = 'a';   // This line may seg fault
```



String length again, this time “the C way”

```
#include <stdio.h>

int main(){

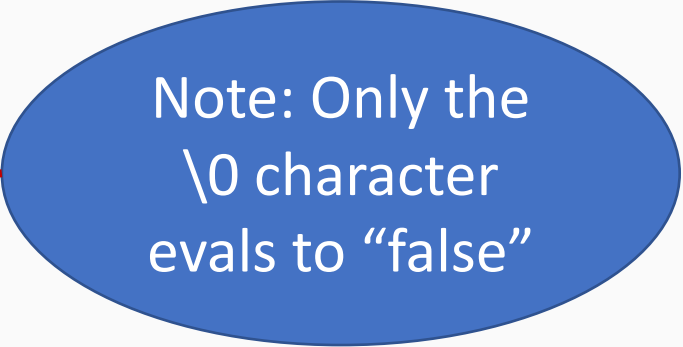
    char str_var[100] = "world";

    int length = 0;
    char *ptr = str_var;

    while( *ptr ){
        ptr++;
        length++;
    }

    printf( "The string %s has length %d.\n", str_var, length );

    return 0;
}
```



Note: Only the `\0` character evals to “false”

String length again, this time “the C way”

```
#include <stdio.h>

int main(){

    char str_var[100] = "world";

    int length = 0;
    char *ptr = str_var;

    while( *ptr ){
        ptr++;
        length++;
    }

    printf( "The string %s has length %d.\n", str_var, length );

    return 0;
}
```

C strings using pointers

- Pointing to start of literal works
 - Pointer math moves us around the string and computes distances
 - Logic is based on the pointer position
- `char str_array[100] = "hello";`
 - `char *ptr = str_array;`
 - `ptr = ptr + 3; // Now points to lo`
 - `ptr = ptr - 1; // Back to llo`
 - `char *ptr2 = str_array;`
 - `ptr - ptr2; // Gives position
// difference, 2 here`
 - `ptr = ptr2; // False, not same spots`
 - `ptr > ptr2; // True, ptr is farther along`

The final step (for today)

- We can now manipulate our strings ourselves
- For you, take a look at `<string.h>` library. These functions let you do things the easy way!

Exercises

- Write a C program to reverse the characters in its first argument and output to terminal
- Assignment 2 2nd question