Section 1: Memory, Data, and Addressing

- Preliminaries
- Representing information as bits and bytes
- Organizing and addressing data in memory
- Manipulating data in memory using C
- Boolean algebra and bit-level manipulations

Arrays

- **Arrays represent adjacent locations in memory storing the same type of data object**
  - e.g., int big_array[128];
    allocates 512 adjacent bytes in memory starting at 0x00ff0000
- **Pointer arithmetic can be used for array indexing in C (if pointer and array have the same type!):**
  - int *array_ptr;

```
array_ptr = big_array;             0x00ff0000
array_ptr = &big_array[0];         0x00ff0000
array_ptr = &big_array[3];         0x00ff000c
array_ptr = &big_array[0] + 3;     0x00ff000c (adds 3 * size of int)
array_ptr = big_array + 3;         0x00ff000c (adds 3 * size of int)
*array_ptr = *array_ptr + 1;       0x00ff000c (but big_array[3] is incremented)
array_ptr = &big_array[130];       0x00ff0208 (out of bounds, C doesn't check)
```

  - In general: &big_array[i] is the same as (big_array + i),
    which implicitly computes: *&bigarray[0] + i*sizeof(bigarray[0]);*

Representing strings

- **A C-style string is represented by an array of bytes.**
  - Elements are one-byte ASCII codes for each character.
  - A 0 byte marks the end of the array.

32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	”	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	’	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	I	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	del

Null-terminated strings

- For example, “Harry Potter” can be stored as a 13-byte array.

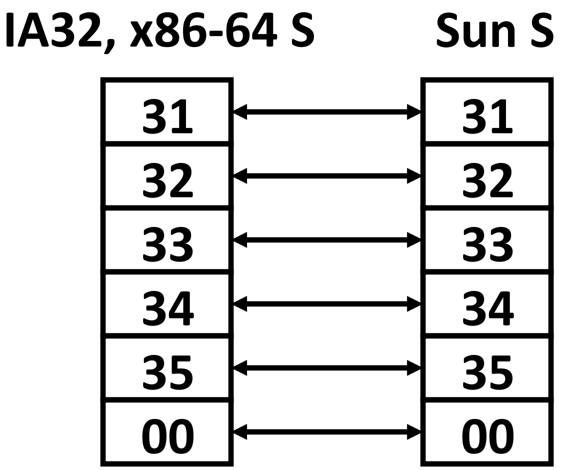
72	97	114	114	121	32	80	111	116	116	101	114	0
H	a	r	r	y		P	o	t	t	e	r	\0

- Why do we put a 0, or null zero, at the end of the string?
  - Note the special symbol: `string[12] = '\0';`

- How do we compute the string length?

Compatibility

```
char S[6] = "12345";
```



IA32, x86-64 S

Sun S

IA32, x86-64 S	Sun S
31	31
32	32
33	33
34	34
35	35
00	00

- **Byte ordering (endianness) is not an issue for standard C strings (char arrays)**
- **Unicode characters – up to 4 bytes/character**
  - ASCII codes still work (just add leading 0 bits) but can support the many characters in all languages in the world
  - Java and C have libraries for Unicode (Java commonly uses 2 bytes/char)

Examining Data Representations

- **Code to print byte representation of data**
  - Any data type can be treated as a *byte array* by casting it to `char`

```
void show_bytes(char *start, int len) {
  int i;
  for (i = 0; i < len; i++)
    printf("%p\t0x%.2x\n", start+i, *(start+i));
  printf("\n");
}
```

```
void show_int (int x) {
  show_bytes( (char *) &x, sizeof(int));
}
```

printf directives:

%p	**Print pointer**
\t	**Tab**
%x	**Print value as hex**
\n	**New line**

show_bytes Execution Example

```
int a = 12345; // represented as 0x00003039
printf("int a = 12345;\n");
show_int(a);   // show_bytes( (byte *) &a, sizeof(int));
```

Result:

```
int a = 12345;
0x7fff6f330dcc  0x39
0x7fff6f330dcd  0x30
0x7fff6f330dce  0x00
0x7fff6f330dcf  0x00
```