Review Questions

1) If the word size of a machine is 64-bits, which of the following is usually true? (pick all that apply)
    a) 64 bits is the size of a pointer
    b) 64 bits is the size of an integer
    c) 64 bits is the width of a register
2) (True/False) By looking at the bits stored in memory, I can tell if a particular 4-bytes is being used to represent an integer, floating point number, or instruction
3) If the size of a pointer on a machine is 7 bits, the address space is how many bytes?

Memory, Data, & Addressing II

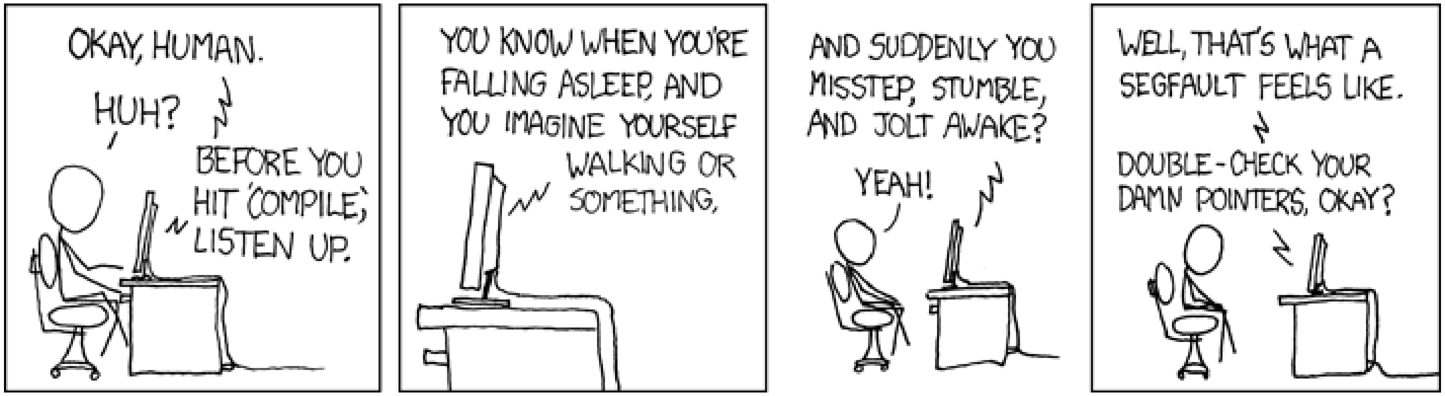
CSE 351 Autumn 2016

Instructor:

Justin Hsia

Teaching Assistants:

Chris Ma
Hunter Zahn
John Kaltenbach
Kevin Bi
Sachin Mehta
Suraj Bhat
Thomas Neuman
Waylon Huang
Xi Liu
Yufang Sun



http://xkcd.com/371/

Administrivia

- Lab 0 due today @ 5pm
  - Credit/no credit – we'll talk about topics in depth later
- Lab 1 released tomorrow @ 6pm
  - Some progress due Monday 10/10, Lab 1 due Friday 10/14
- Survey results:
  - Hoping to get out
    - Linux/Unix, C, Assembly
    - How a computer works
  - Concerns
    - Fast-paced & course load
    - Having little background in this area
    - Not familiar with C or Linux

Memory, Data, and Addressing

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

Addresses and Pointers in C

`*` is also used with variable declarations

- `&` = "address of" operator
- `*` = "value at address" or "dereference" operator

```
int* ptr;
```

Declares a variable, `ptr`, that is a pointer to (i.e. holds the address of) an `int` in memory

```
int x = 5;
int y = 2;
```

Declares two variables, `x` and `y`, that hold `int`s, and sets them to 5 and 2, respectively

```
ptr = &x;
```

Sets `ptr` to the address of `x` ("`ptr` points to `x`")

```
y = 1 + *ptr;
```

"Dereference `ptr`"

Sets `y` to "1 plus the value stored at the address held by `ptr`. Because `ptr` points to `x`, this is equivalent to `y=1+x;`

What is `*(&y)` ?

Assignment in C

- A variable is represented by a memory location
- Declaration ≠ initialization (initially holds "garbage")
- `int x, y;`
  - `x` is at address 0x04, `y` is at 0x18

0x00	0x01	0x02	0x03		
A7	00	32	00	0x00	
00	01	29	F3	0x04	x
EE	EE	EE	EE	0x08	
FA	CE	CA	FE	0x0C	
26	00	00	00	0x10	
00	00	10	00	0x14	
01	00	00	00	0x18	y
FF	00	F4	96	0x1C	
DE	AD	BE	EF	0x20	
00	00	00	00	0x24	

Assignment in C

32-bit example
(pointers are 32-bits wide)

- A variable is represented by a memory location
- Declaration ≠ initialization (initially holds "garbage")
- `int x, y;`
  - `x` is at address 0x04, `y` is at 0x18

0x00	0x01	0x02	0x03		
				0x00	
00	01	29	F3	0x04	x
				0x08	
				0x0C	
				0x10	
				0x14	
01	00	00	00	0x18	y
				0x1C	
				0x20	
				0x24	

Assignment in C

32-bit example
(pointers are 32-bits wide)

`&` = "address of"
`*` = "dereference"

- left-hand side = right-hand side;
  - LHS must evaluate to a memory *location*
  - RHS must evaluate to a *value* (could be an address)
  - Store RHS value at LHS location
- `int x, y;`
- `x = 0;`

0x00	0x01	0x02	0x03		
				0x00	
00	00	00	00	0x04	x
				0x08	
				0x0C	
				0x10	
				0x14	
01	00	00	00	0x18	y
				0x1C	
				0x20	
				0x24	

Assignment in C

32-bit example
(pointers are 32-bits wide)

`&` = "address of"
`*` = "dereference"

- left-hand side = right-hand side;
  - LHS must evaluate to a memory *location*
  - RHS must evaluate to a *value* (could be an address)
  - Store RHS value at LHS location
- `int x, y;`
- `x = 0;`
- `y = 0x3CD02700;`

little endian!

0x00	0x01	0x02	0x03		
				0x00	
00	00	00	00	0x04	x
				0x08	
				0x0C	
				0x10	
				0x14	
00	27	D0	3C	0x18	y
				0x1C	
				0x20	
				0x24	

Assignment in C

32-bit example
(pointers are 32-bits wide)

`&` = "address of"
`*` = "dereference"

- left-hand side = right-hand side;
  - LHS must evaluate to a memory *location*
  - RHS must evaluate to a *value* (could be an address)
  - Store RHS value at LHS location
- `int x, y;`
- `x = 0;`
- `y = 0x3CD02700;`
- `x = y + 3;`
  - Get value at `y`, add 3, store in `x`

0x00	0x01	0x02	0x03		
				0x00	
03	27	D0	3C	0x04	x
				0x08	
				0x0C	
				0x10	
				0x14	
00	27	D0	3C	0x18	y
				0x1C	
				0x20	
				0x24	

Assignment in C

32-bit example
(pointers are 32-bits wide)

`&` = "address of"
`*` = "dereference"

- left-hand side = right-hand side;
  - LHS must evaluate to a memory *location*
  - RHS must evaluate to a *value* (could be an address)
  - Store RHS value at LHS location
- `int x, y;`
- `x = 0;`
- `y = 0x3CD02700;`
- `x = y + 3;`
  - Get value at `y`, add 3, store in `x`
- `int* z;`
  - `z` is at address 0x20

0x00	0x01	0x02	0x03		
				0x00	
03	27	D0	3C	0x04	x
				0x08	
				0x0C	
				0x10	
				0x14	
00	27	D0	3C	0x18	y
				0x1C	
DE	AD	BE	EF	0x20	z
				0x24	

Assignment in C

32-bit example
(pointers are 32-bits wide)

`&` = "address of"
`*` = "dereference"

- left-hand side = right-hand side;
  - LHS must evaluate to a memory *location*
  - RHS must evaluate to a *value* (could be an address)
  - Store RHS value at LHS location
- `int x, y;`
- `x = 0;`
- `y = 0x3CD02700;`
- `x = y + 3;`
  - Get value at `y`, add 3, store in `x`
- `int* z = &y + 3;`
  - Get address of `y`, "add 3", store in `z`

Pointer arithmetic

0x00	0x01	0x02	0x03		
				0x00	
03	27	D0	3C	0x04	x
				0x08	
				0x0C	
				0x10	
				0x14	
00	27	D0	3C	0x18	y
				0x1C	
24	00	00	00	0x20	z
				0x24	

Pointer Arithmetic

- Pointer arithmetic is scaled by the size of target type
  - In this example, `sizeof(int)` = 4
- `int* z = &y + 3;`
  - Get address of `y`, add `3*sizeof(int)`, store in `z`
  - `&y` = `0x18` = $1*16^1 + 8*16^0 = 24$
  - $24 + 3*(4) = 36 = 2*16^1 + 4*16^0 =$ `0x24`

- Pointer arithmetic can be dangerous!
  - Can easily lead to bad memory accesses
  - Be careful with data types and *casting*

Assignment in C

32-bit example
(pointers are 32-bits wide)

`&` = "address of"
`*` = "dereference"

- `int x, y;`
- `x = 0;`
- `y = 0x3CD02700;`
- `x = y + 3;`
  - Get value at `y`, add 3, store in `x`
- `int* z = &y + 3;`
  - Get address of `y`, add **12**, store in `z`
- `*z = y;`
  - What does this do?

0x00	0x01	0x02	0x03		
				0x00	
03	27	D0	3C	0x04	x
				0x08	
				0x0C	
				0x10	
				0x14	
00	27	D0	3C	0x18	y
				0x1C	
24	00	00	00	0x20	z
				0x24	

Assignment in C

32-bit example
(pointers are 32-bits wide)

`&` = "address of"
`*` = "dereference"

- `int x, y;`
- `x = 0;`
- `y = 0x3CD02700;`
- `x = y + 3;`
  - Get value at `y`, add 3, store in `x`
- `int* z = &y + 3;`
  - Get address of `y`, add **12**, store in `z`
- `*z = y;`
  - Get value of `y`, put in address stored in `z`

The target of a pointer is also a memory location

0x00	0x01	0x02	0x03		
				0x00	
03	27	D0	3C	0x04	x
				0x08	
				0x0C	
				0x10	
				0x14	
00	27	D0	3C	0x18	y
				0x1C	
24	00	00	00	0x20	z
00	27	D0	3C	0x24	

Arrays in C

Arrays are adjacent locations in memory storing the same type of data object

a is a name for the array's address

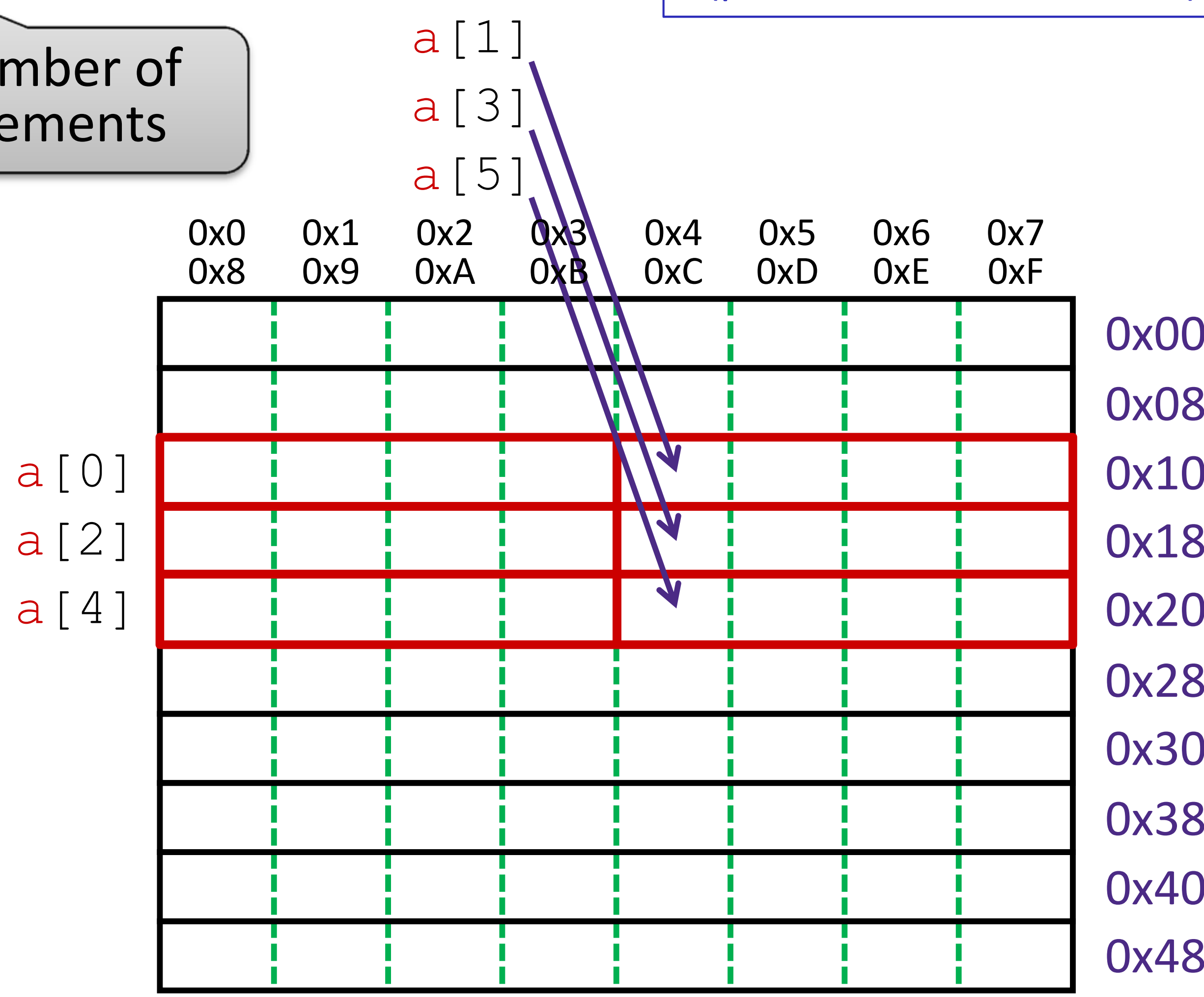
Declaration: `int a[6];`

element type

name

number of elements

64-bit example
(pointers are 64-bits wide)



Arrays in C

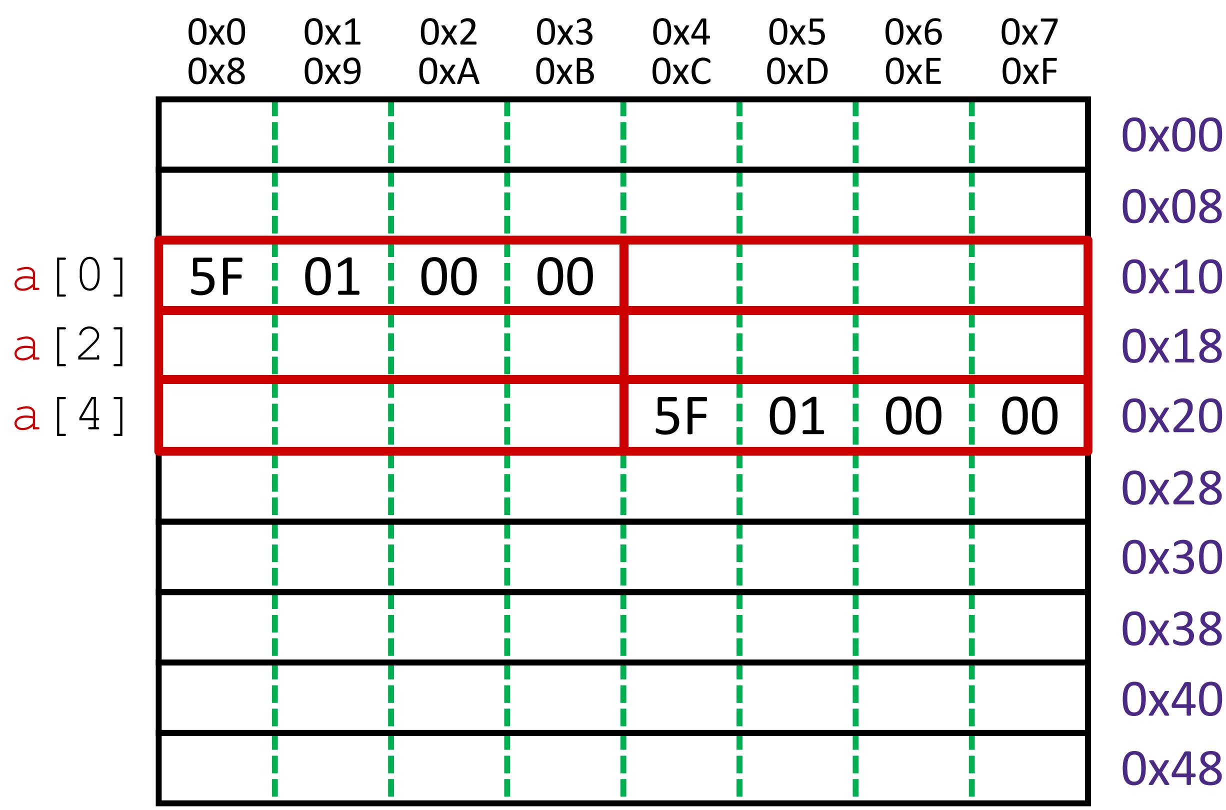
Arrays are adjacent locations in memory storing the same type of data object

`a` is a name for the array's address

The address of `a[i]` is the address of `a[0]` plus `i` times the element size in bytes

Declaration: `int a[6];`

Indexing:
```
a[0] = 0x015f;
a[5] = a[0];
```



Arrays in C

Arrays are adjacent locations in memory storing the same type of data object

`a` is a name for the array's address

The address of `a[i]` is the address of `a[0]` plus `i` times the element size in bytes

Declaration: `int a[6];`

Indexing:
```
a[0] = 0x015f;
a[5] = a[0];
```

No bounds checking:
```
a[6] = 0xBAD;
a[-1] = 0xBAD;
```

	0x0 0x8	0x1 0x9	0x2 0xA	0x3 0xB	0x4 0xC	0x5 0xD	0x6 0xE	0x7 0xF	
									0x00
					AD	0B	00	00	0x08
a[0]	5F	01	00	00					0x10
a[2]									0x18
a[4]					5F	01	00	00	0x20
	AD	0B	00	00					0x28
									0x30
									0x38
									0x40
									0x48

Arrays in C

Arrays are adjacent locations in memory storing the same type of data object

a is a name for the array's address

The address of `a[i]` is the address of `a[0]` plus `i` times the element size in bytes

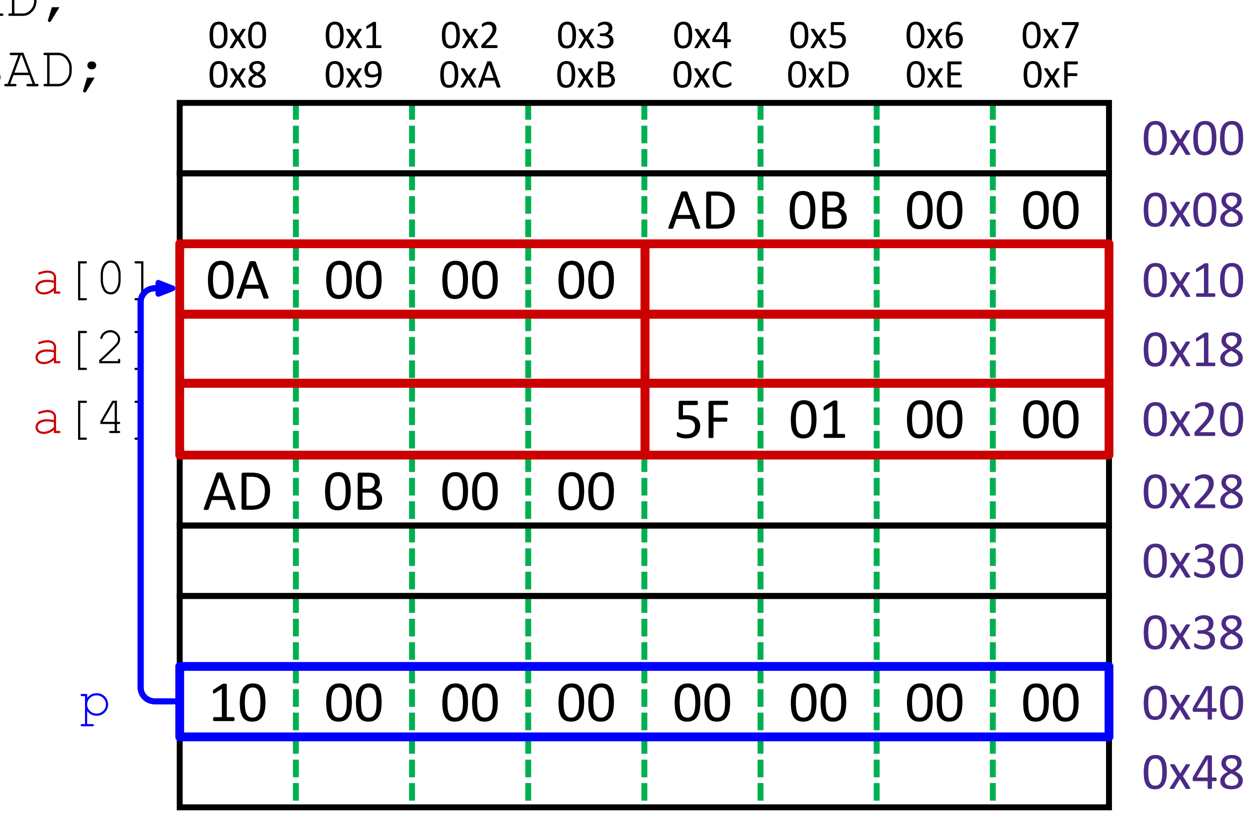
Declaration: `int a[6];`

Indexing:
```
a[0] = 0x015f;
a[5] = a[0];
```

No bounds checking:
```
a[6] = 0xBAD;
a[-1] = 0xBAD;
```

Pointers: `int* p;`

equivalent {
```
p = a;
p = &a[0];
```
}
```
*p = 0xA;
```



Arrays in C

Arrays are adjacent locations in memory storing the same type of data object

`a` is a name for the array's address

The address of `a[i]` is the address of `a[0]` plus `i` times the element size in bytes

Declaration: `int a[6];`

Indexing:
```
a[0] = 0x015f;
a[5] = a[0];
```

No bounds checking:
```
a[6] = 0xBAD;
a[-1] = 0xBAD;
```

Pointers:
```
int* p;
```
equivalent {
```
p = a;
p = &a[0];
```
```
*p = 0xA;
```

array indexing = address arithmetic
(both scaled by the size of the type)

equivalent {
```
p[1] = 0xB;
*(p+1) = 0xB;
```
```
p = p + 2;
```

	0x0 0x8	0x1 0x9	0x2 0xA	0x3 0xB	0x4 0xC	0x5 0xD	0x6 0xE	0x7 0xF	
									0x00
					AD	0B	00	00	0x08
a[0]	0A	00	00	00	0B	00	00	00	0x10
a[2]									0x18
a[4]					5F	01	00	00	0x20
	AD	0B	00	00					0x28
									0x30
									0x38
p	10	00	00	00	00	00	00	00	0x40
									0x48

Arrays in C

Arrays are adjacent locations in memory storing the same type of data object

`a` is a name for the array's address

The address of `a[i]` is the address of `a[0]` plus `i` times the element size in bytes

Declaration: `int a[6];`

Indexing:
```
a[0] = 0x015f;
a[5] = a[0];
```

No bounds checking:
```
a[6] = 0xBAD;
a[-1] = 0xBAD;
```

Pointers: `int* p;`

equivalent {
```
p = a;
p = &a[0];
```
}
```
*p = 0xA;
```

array indexing = address arithmetic
(both scaled by the size of the type)

equivalent {
```
p[1] = 0xB;
*(p+1) = 0xB;
```
}
```
p = p + 2;
```

```
*p = a[1] + 1;
```

	0x0 0x8	0x1 0x9	0x2 0xA	0x3 0xB	0x4 0xC	0x5 0xD	0x6 0xE	0x7 0xF	
									0x00
					AD	0B	00	00	0x08
a[0]	0A	00	00	00	0B	00	00	00	0x10
a[2]	0C	00	00	00					0x18
a[4]					5F	01	00	00	0x20
	AD	0B	00	00					0x28
									0x30
									0x38
p	18	00	00	00	00	00	00	00	0x40
									0x48

http://PollEv.com/justinh

Question: The variable values after Line 3 executes are shown on the right. What are they after Line 4 & 5?

```
1    void main() {
2        int a[] = {5,10};
3        int *p = a;
4          p =  p + 1;
5        *p = *p + 1;
6    }
```

	Data (decimal)	Address (decimal)
a[0]	5	100
a[1]	10	
⋮	⋮	
p	100	

	p	*p	a[0]	a[1]		p	*p	a[0]	a[1]
(A)	101	10	5	10	then	101	11	5	11
(B)	104	10	5	10	then	104	11	5	11
(C)	100	6	6	10	then	101	6	6	10
(D)	100	6	6	10	then	104	6	6	10

Representing strings

- C-style string stored as an array of bytes (`char *`)
  - Elements are one-byte ASCII codes for each character
  - No “String” keyword, unlike Java

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**

ASCII: American Standard Code for Information Interchange

Null-Terminated Strings

- **Example:** "Donald Trump" stored as a 13-byte array

Decimal:	68	111	110	97	108	100	32	84	114	117	109	112	0
Hex:	0x44	0x6F	0x6E	0x61	0x6C	0x64	0x20	0x54	0x72	0x75	0x6D	0x70	0x00
Text:	D	o	n	a	l	d		T	r	u	m	p	\0

- Last character followed by a 0 byte (`'\0'`) (a.k.a. "null terminator")
  - Must take into account when allocating space in memory
  - Note that `'0'` ≠ `'\0'` (i.e. character 0 has non-zero value)
- How do we compute the length of a string?
  - Traverse array until null terminator encountered

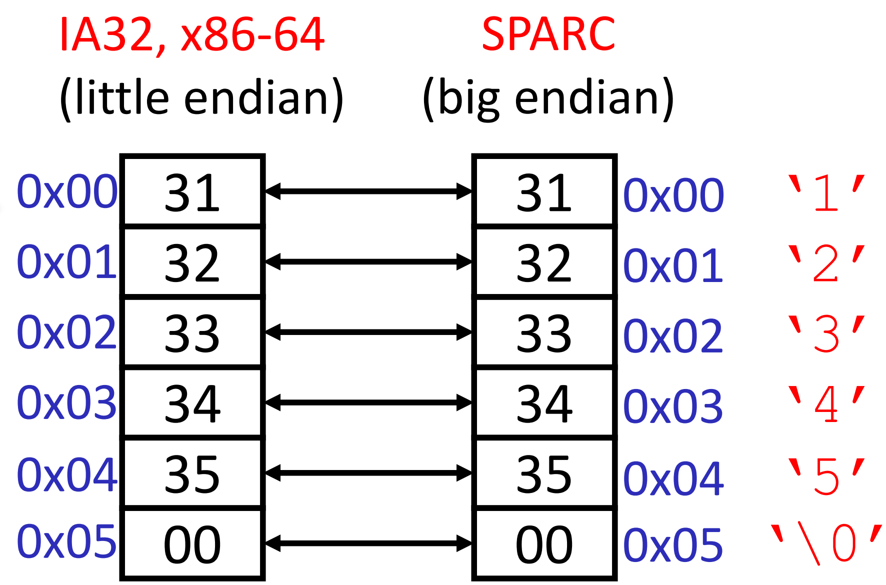
Endianness and Strings

C (char = 1 byte)

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

String literal

0x31 = 49 decimal = ASCII '1'



- Byte ordering (endianness) is not an issue for 1-byte values
  - The whole array does not constitute a single value
  - Individual elements are values; chars are single bytes
- Unicode characters – up to 4 bytes/character
  - ASCII codes still work (just add leading zeros)
  - Unicode 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`
  - C has unchecked casts !! DANGER !!

```
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");
}
```

`printf` directives:

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

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`
  - C has unchecked casts !! DANGER !!

```
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));
}
```

show_bytes Execution Example

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

- Result (Linux x86-64):
  - **Note:** The addresses will change on each run (try it!), but fall in same general range

```
int a = 12345;
0x7fffb7f71dbc      0x39
0x7fffb7f71dbd      0x30
0x7fffb7f71dbe      0x00
0x7fffb7f71dbf      0x00
```

Summary

- Assignment in C results in value being put in memory location
- Pointer is a C representation of a data address
  - `&` = "address of" operator
  - `*` = "value at address" or "dereference" operator
- Pointer arithmetic scales by size of target type
  - Convenient when accessing array-like structures in memory
  - Be careful when using – particularly when *casting* variables
- Arrays are adjacent locations in memory storing the same type of data object
  - Strings are null-terminated arrays of characters (ASCII)