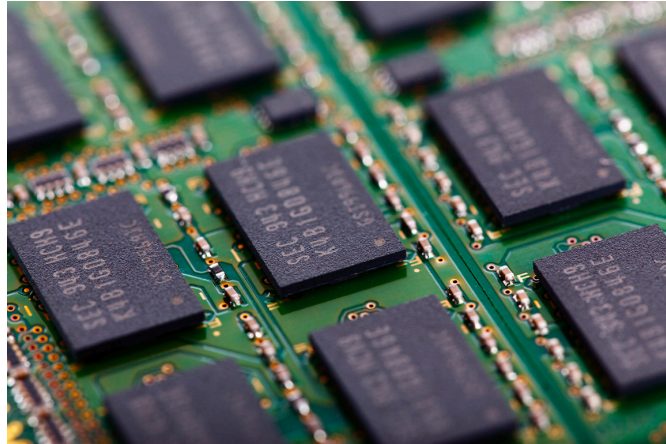


# Memory



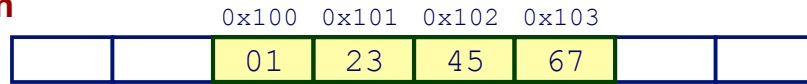
# Memory Addresses

Content:	FF	00	57	92	B3	8A	... ..	10	46	DC
Address:	000 000 000	000 000 001	000 000 002	000 000 003	000 000 004	000 000 005	...	134 217 725	134 217 726	134 217 727

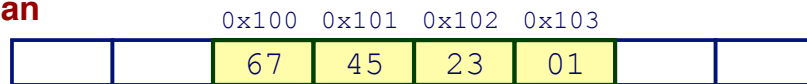
# Endianness

```
int x = 0x01234567; // stored at address 0x100
```

## Big Endian



## Little Endian



# Sample Memory

				0x20
				0x1C
80	00	00	00	0x18
				0x14
00	00	00	FF	0x10
				0x0C
				0x08
				0x04
00	00	00	13	0x00
0x03	0x02	0x01	0x00	

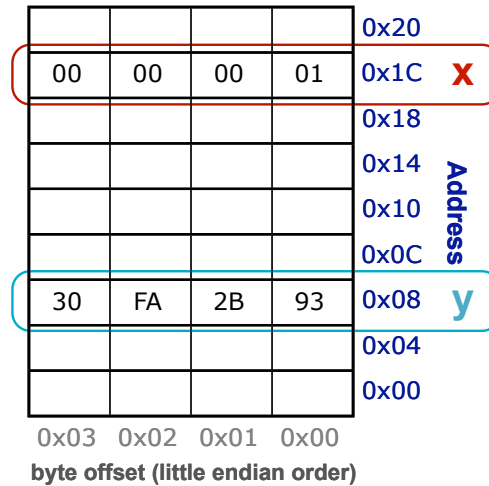
byte offset (little endian order)

Address

# Variables as Addresses

```
int x; // x at 0x1C
int y; // y at 0x08
```

```
x = 1;
y = 0x30FA2B93;
```



# Java References

```
class Thing {
    // ... instance variables ...
}
```

```
public void doStuff() {

    Thing t1 = new Thing();
    Thing t2 = new Thing();

}
```

## Java References (continued)

```
class Thing {
    // ... instance variables ...
}

public void doStuff() {

    Thing t1 = new Thing();
    Thing t2 = t1;

}
```

## Pointers

**address** = index of a byte in memory

**pointer** = a piece of data storing an address

**T\* p;** declare a pointer p that will point to something of type T

**&x** **address of** x (get a pointer to x)

**\*p** **contents at** address given by pointer p (aka “dereference p” – follow the pointer)

# Pointer Example

```
int* p; // p: 0x10
```

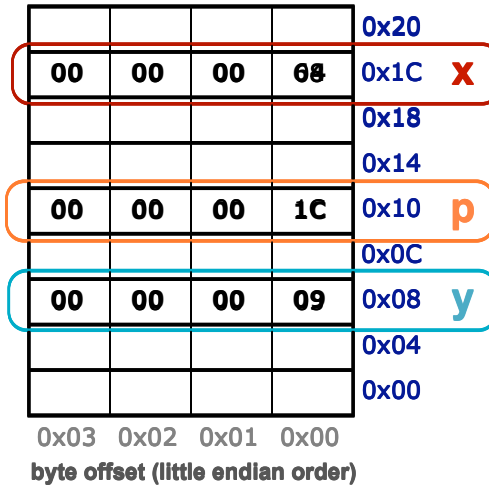
```
int x = 8; // x: 0x1C
```

```
int y = 3; // y: 0x08
```

```
p = &x;
```

```
y = 1 + *p;
```

```
*p = 100;
```



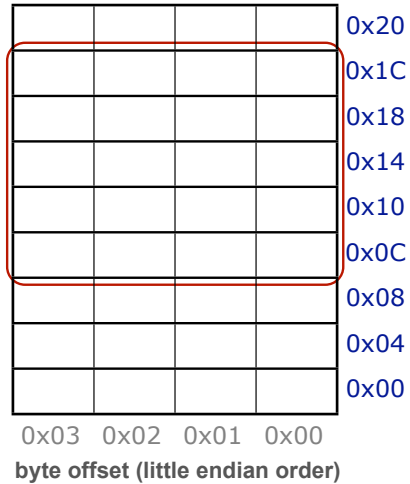
# Pointer Example

```
void do_something(int* p1, int* p2) {  
    int temp = *p1;  
    *p1 = *p2;  
    *p2 = temp;  
}
```

```
void main() {  
    int x = 5;  
    int y = 3;  
  
    printf("%d %d\n", x, y); // 5 3  
  
    do_something(&x, &y); // swap!  
  
    printf("%d %d\n", x, y); // 3 5  
}
```

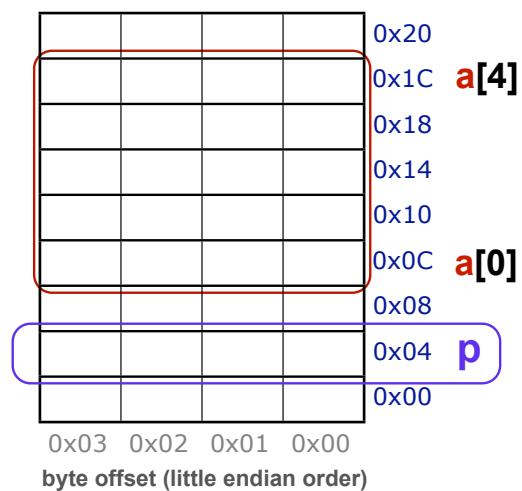
# Arrays in C

```
int a[5]; // creation  
  
a[0] = 0xFF; // indexing  
a[3] = a[0];  
  
a[5] = 0xBAD; // uh oh  
a[-1] = 0xBAD; // x2  
  
a.length; // nope!
```



# Arrays as Pointers

```
int a[5];  
  
int* p;
```

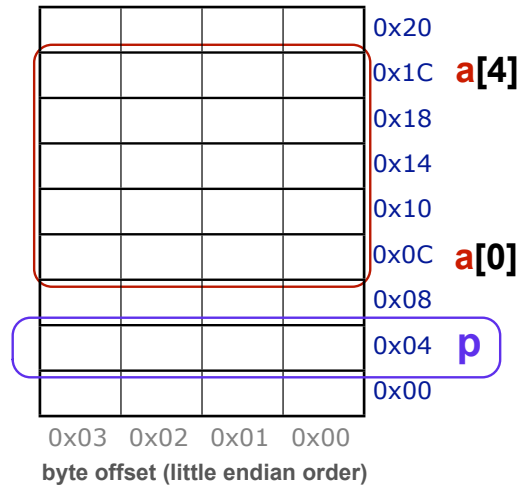


# Arrays as Pointers

```
int a[5];
```

```
int* p;
```

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

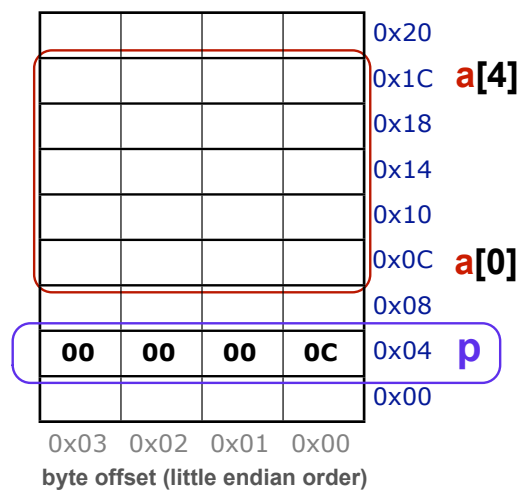


# Arrays as Pointers

```
int a[5];
```

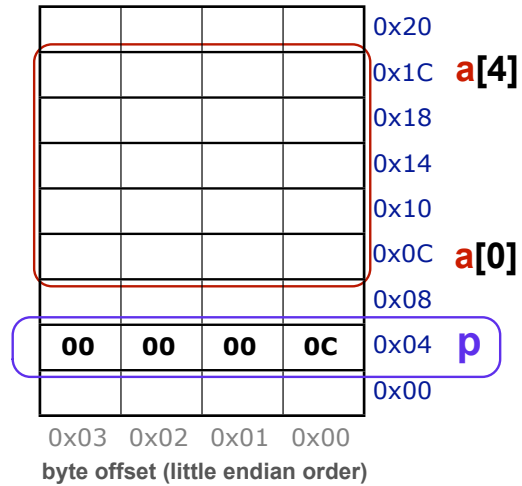
```
int* p;
```

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



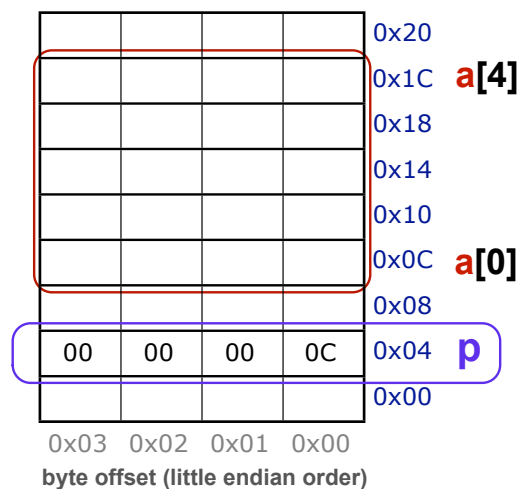
# Arrays as Pointers

```
int a[5];  
  
int* p;  
  
p = &a[0]; // or p = a;
```



# Arrays as Pointers

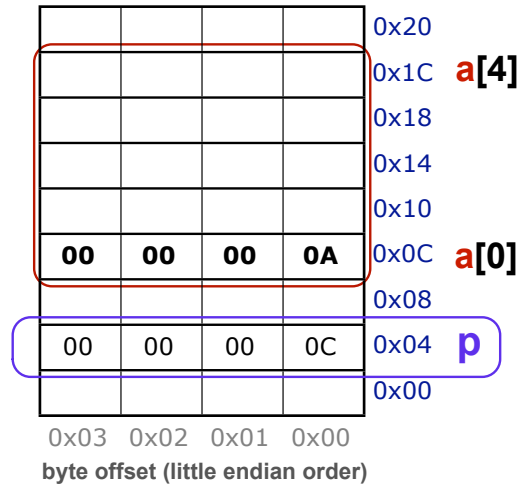
```
int a[5];  
  
int* p;  
  
p = a; // &a[0]  
  
*p = 0xA;
```





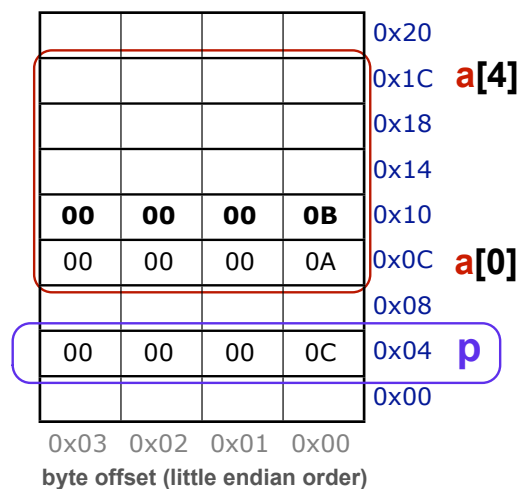
# Arrays as Pointers

```
int a[5];  
  
int* p;  
  
p = a; // &a[0]  
  
*p = 0xA;
```



# Arrays as Pointers

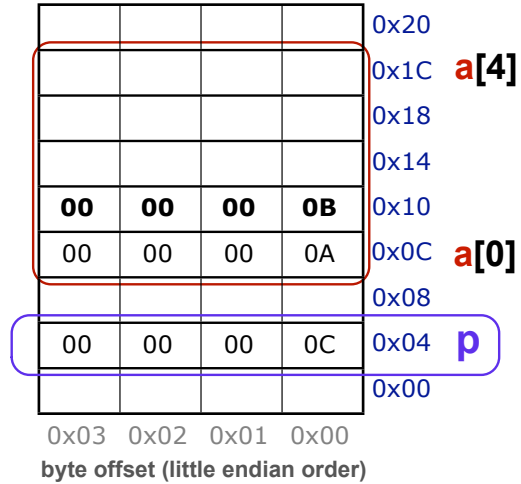
```
int a[5];  
  
int* p;  
  
p = a; // &a[0]  
  
*p = 0xA;  
p[1] = 0xB;
```



# Pointer Arithmetic

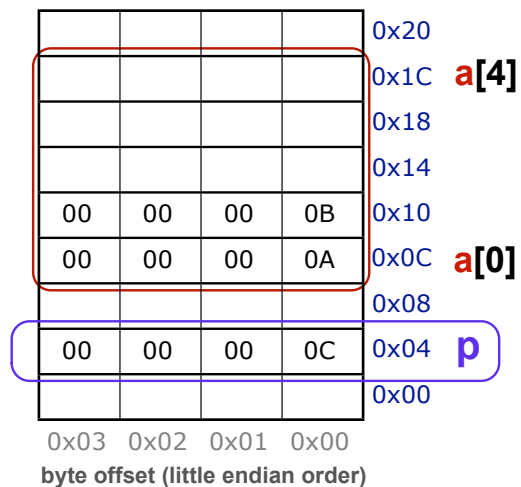
```
int a[5];  
  
int* p;  
  
p = a; // &a[0]  
  
*p = 0xA;  
p[1] = 0xB;  
*(p + 1) = 0xB;
```

Equivalent!



# Pointer Arithmetic

```
int a[5];  
  
int* p;  
  
p = a; // &a[0]  
  
*p = 0xA;  
*(p + 1) = 0xB;  
  
p = p + 2;  
*p = a[1] + 1;
```



# Pointer Arithmetic

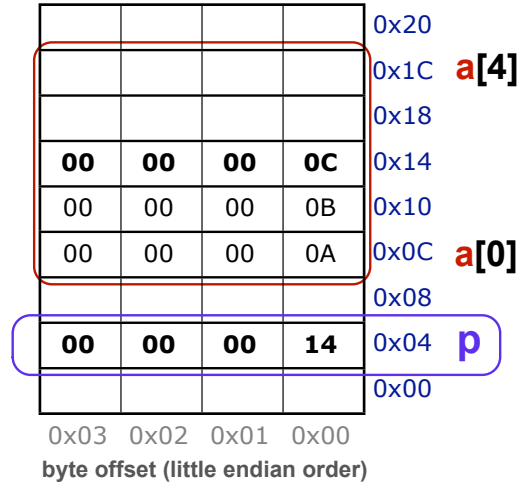
```
int a[5];

int* p;

p = a; // &a[0]

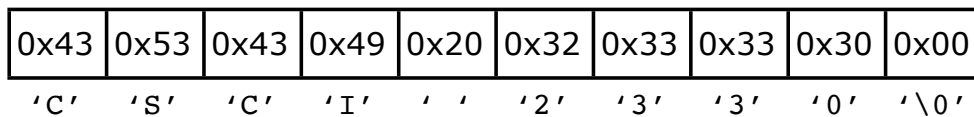
*p = 0xA;
*(p + 1) = 0xB;

p = p + 2;
*p = a[1] + 1;
```

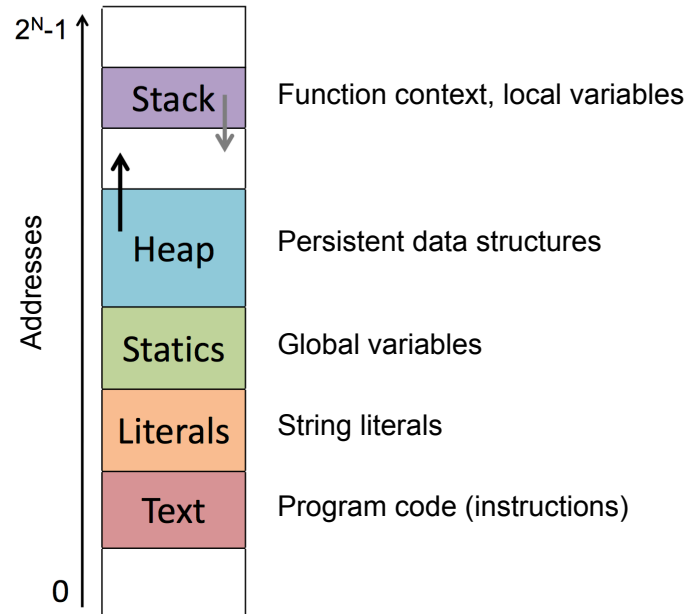


# Null-Terminated Strings

*null character*



# Memory Layout



# Dynamic Memory Allocation Example

```
#define ZIP_LENGTH 5

int* zip = (int*) malloc(sizeof(int) * ZIP_LENGTH);
if (zip == NULL) {
    perror("malloc failed");
    exit(0);
}

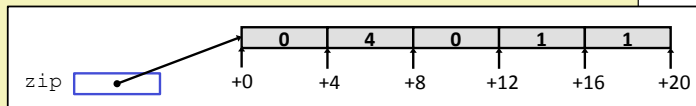
zip[0] = 0;
zip[1] = 4;
zip[2] = 0;
zip[3] = 1;
zip[4] = 1;

printf("zip is");
for (int i = 0; i < ZIP_LENGTH; i++) {
    printf(" %d", zip[i]);
}
printf("\n");

free(zip);
```

zip 0x7fedd2400dc0 0x7fff58bdd938

0	0x7fedd2400dd0
4	0x7fedd2400dcc
0	0x7fedd2400dc8
1	0x7fedd2400dc4
1	0x7fedd2400dc0



# Pointers to Pointers to ...

```
#define NUM_ZIPS 3

int** zips = (int**) malloc(sizeof(int*) * NUM_ZIPS);
...
zips[0] = (int*) malloc(sizeof(int) * ZIP_LENGTH);
...
int* first_zip = zips[0];
first_zip[0] = 0;
zips[0][1] = 4;
zips[0][2] = 0;
first_zip[3] = 1;
zips[0][4] = 1;
```

