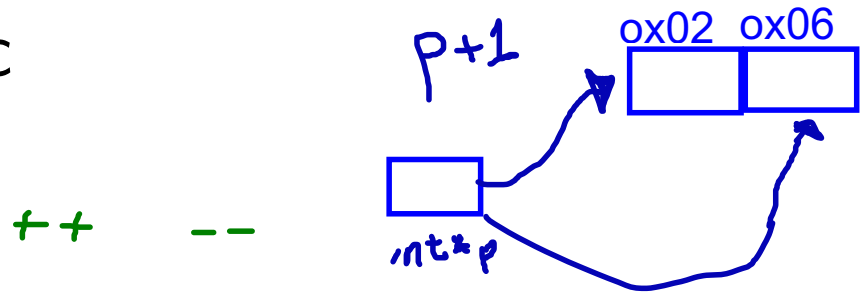
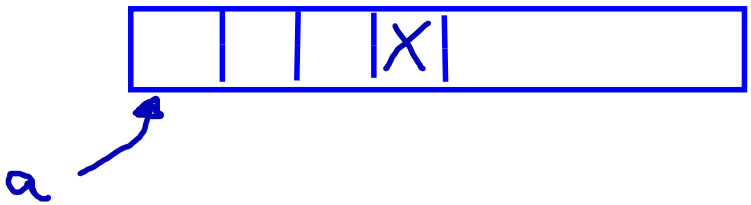
Pointer Arithmetic



- +, -, +=, -= for other pointers or integers
- Most often used on pointers that are arrays
- Doesn't add the actual number, it adds that number times how many bytes each element takes up
  - for variable int * p, code "p+1" will in fact add 4 bytes (sizeof(int)) to p's address
- ptr1 = ptr2 assignment works for pointers of same type
- ptr1 == ptr2 etc makes sense to compare ptrs ("do they point to the same memory location?"), and ptr == NULL

pointer arithmetic takes into account the type of the pointer we are adding to or subtracting from. So, for instance, if p is a pointer of type char *, then p + 1 will be adding oner byte to p because p is of type char *, but if p were of type double * for instance, then p + 1 would be adding 8 bytes to it, because size of double is 8.

Pointer Arithmetic and Arrays



- A declared array, say int `a[10]`, is "really" just an address that starts a block of memory.
- Writing a is generally the same as writing `&a[0]`
- `a[3]` is a synonym for `*(a + 3)` (offset three from pointer to start of array)
- `&a[3]` is a synonym for `a + 3`

if you write the array name just by itself, it is the base address (address of the very first element) of the array.

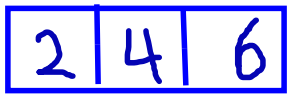
Pointer Arithmetic and Arrays

pointerArith.c:

```
#include <stdio.h>
#include <stdlib.h>
int main() {

    int array[] = {2, 4, 6};

    printf("array[1] = %d, ", array[1]);
    printf(" *(array + 1) = %d, ", *(array+1));
    printf(" array = %p\n", (void *)array);
    printf(" &array[1] = %p, ",(void *)  &array[1]);
    printf(" array + 1 = %p\n", (void *)(array + 1));
    return 0;
}
```




```
$ gcc -std=c99 -Wall -Wextra -pedantic pointerArith.c
$ ./a.out
array[1] = 4,  *(array + 1) = 4,  array = 0x7ffebacbd214
 &array[1] = 0x7ffebacbd218,  array + 1 = 0x7ffebacbd218
```

What is the correct output?