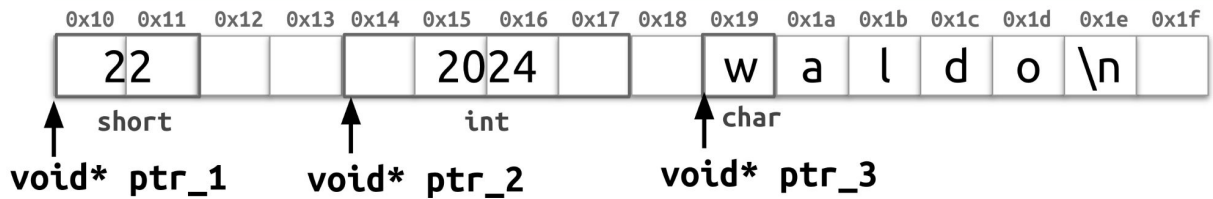




Generics: One Pointer to Rule Them All



- 👉 We can use **void*** to represent a generic pointer to “something”.
- 👉 It loses information about data types, allowing for more flexibility.

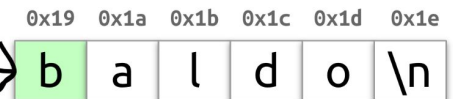


❌ We can't dereference **void*** pointers right away! Don't do this plz :(

To access/change memory a **void*** points to:

- ★ Use the generic memcpy / memmove
- ★ Or, if we know what type is actually being pointed to, we **first cast, then dereference**

***(char *) ptr_3 = 'b';**



Generics: Pointer Arithmetic



For pointer arithmetic with **void*** pointers, why do we cast to a **char***?

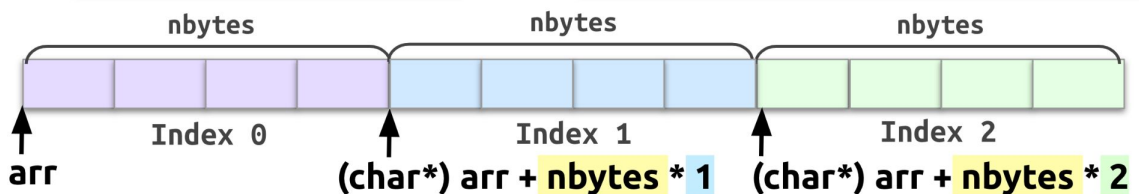
Consistency: Points to a **char** (only 1 byte), so we can work/add in terms of bytes!

Check-Off Q Consider a generic array **arr** where each element has width **nbytes**. How do we get a **pointer to the ith element** of the array?

Solution: **void* elem_ptr = (char *) arr + i * nbytes;**

Cast to **char*** so we can work in individual bytes

Each element is **nbytes**, so multiply with number of elements (**i**) to move



Function Pointers: Sharing is Caring!



- 👉 We can **pass functions as parameters to other functions**, by specifying one or more function pointer parameters. These are termed **callback functions**.

- ❑ A **function pointer** lets us pass logic/wisdom from caller to callee
- ❑ For example, we can pass a function to **bubble_sort** that knows how to compare 2 elements of the type we are sorting

Function pointers have the form:

return_type (*function_name) (arg1,...)

For instance,

int (*sat_add) (int a, int b)

Comparison functions in particular have the following generic prototype:

int (*my_compare) (const void *a, const void *b)

Sign of **return value** tells us which elem goes first when sorting

👉 **Pointer to 1st element**

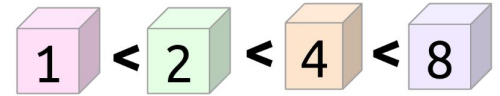
👉 **Pointer to 2nd element**



Teaching C How to Sort: Comparison Functions!



💡 Sorting is really just knowing **how to compare** items (i.e. which items go before which other items)



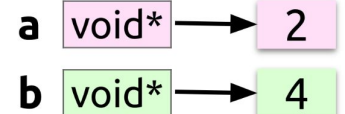
As such, to sort, we have to define a **comparison function** `cmp(a, b)`

This function returns:

- ❑ `<0` (Negative) if **a comes before b**
- ❑ `0` if **a and b are equal**
- ❑ `>0` (Positive) if **a comes after b**

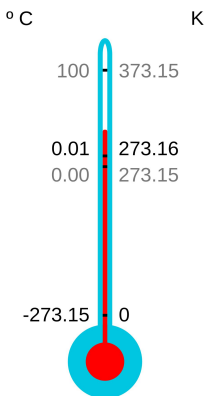
Most comparison functions we write, if not all, will have the blueprint:

```
int my_compare(const void *a, const void *b){
    // code to cast & dereference the received elements
    // code to compare the values, often by subtracting them
}
```

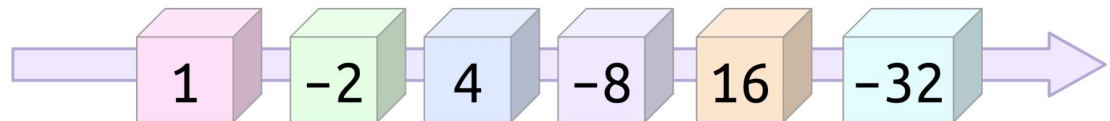


Due to generic void* args, we must input pointers to what we're comparing!

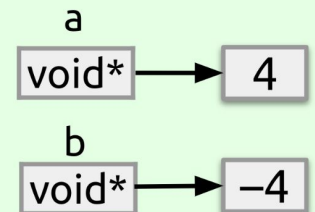
Example: Absolute Value Sort!



Write a **comparison function** (that can be passed to `qsort`) to help arrange an array of `ints` in order of **increasing absolute value from left to right!**



```
int compare_abs(const void *a, const void *b){
    int a_val = *(int *)a;
    int b_val = *(int *)b;
    return abs(a_val) - abs(b_val);
}
```



bsearch() Generic Library Function



As input, we always pass the following into `bsearch`:

- **key**: A **pointer to the value** we're searching for, e.g., "gem"
- **base**: **Pointer to base / starting element of array** to search
- **nmemb**: Total # of elements in array
- **size/width**: # of bytes per element, e.g., 4 for array for ints
- **compar**: The **comparison function** used to find a patch

The return value is:

- If found, a **pointer to the matching element** in array
- If no match, **NULL** pointer is returned

e.g., searching for 8 (with `abs` value comp)



Rule: Array must be **sorted according to the same cmp function** the search uses

(GDB) Printing Arrays



Arrays: To printing `nelem` elements starting from index:

> (gdb) print `arr[index]@nelem`

For a generic array, we have to cast first, e.g., for `void*` `ptr` to an array of ints,

> (gdb) print `((int*) ptr)[index]@nelems`

`int a[] = {10, 20, 30, 40};`



Here, (gdb) print `((int *)ptr)[0] @ 4`