



CS107 Lab 4 – You’ve doing great!



Void Star, Function Pointers, Generics

Ben Yan, Winter 2026 ❄️

Slides Link: stanford.edu/~bbyan/cs107 :)



My Song of the Week: All (Void) Star by Smash Mouth



Stanford | ENGINEERING
Computer Science

Welcome Back to Lab!

Make sure to give yourself a pat on the back! You've learned a lot so far, and you're doing wonderful



Winter



Week 5++ (halfway mark!)



Spring



We have office hours each week in CoDa :) Feel free to stop by, get coffee, prep for midterm, ask questions, or chat with Jerry!

Seriously, amazing work so far! Jerry approves



poohbear@stanford.edu



(motto from CS106A)
computer = powerful + stupid
you = powerful + creative
ben = creative + stupid

You slaying on the midterm next week

Getting Setup for Lab 4



- ❑ To get started, log into the **myth machines**, and clone the lab starter code using the command:

```
git clone /afs/ir/class/cs107/repos/lab4/shared lab4
```

- ❑ Next, also open the **online lab check-off** linked below (or in the lab 1 handout), so you can write as you go.

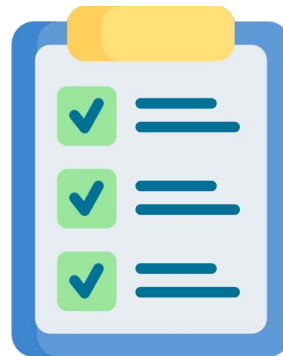


[Lab Handout](#)



[Lab Check-Off Form](#)

Don't worry at all if you fall behind on the write up! I'll aim to go over everything, though just see me after lab if anything is still left



Lecture Recap!



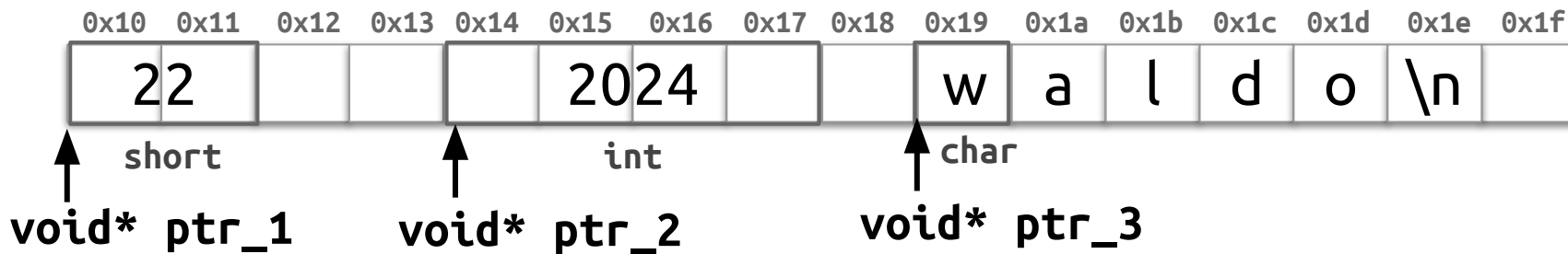
~15 minutes





Generics: One Pointer to Rule Them All

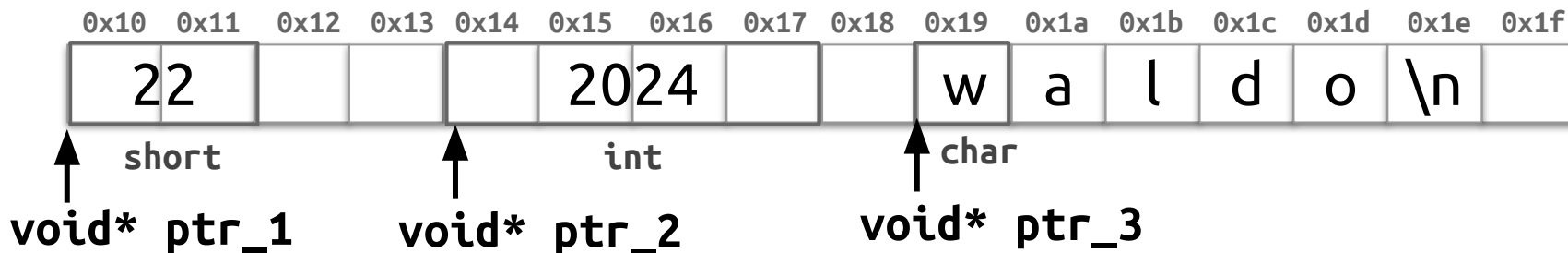
- 👉 We can use **void*** to represent a generic pointer to “something”.
- 👉 It loses information about data types, allowing for more flexibility.





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.

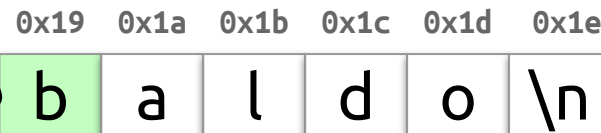


✗ We can't dereference void* pointers right away! Don't do this plz :(

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!

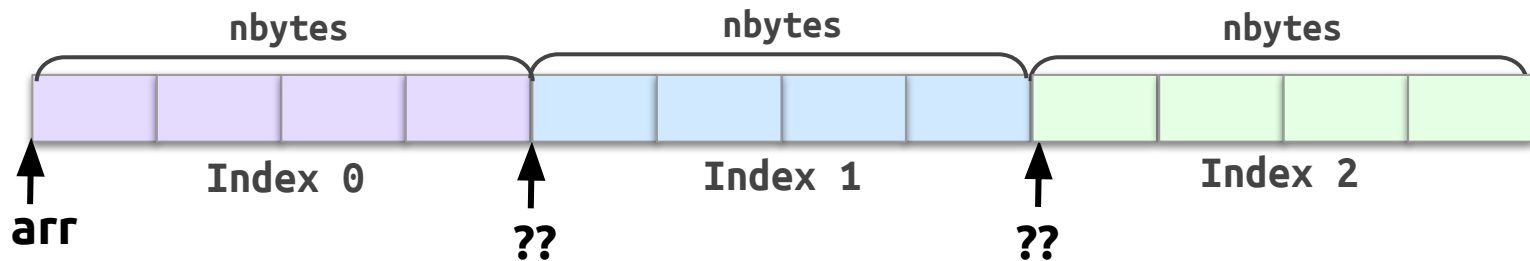
👉 Generics: Pointer Arithmetic

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?

void* elem_ptr =



👉 Generics: Pointer Arithmetic

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

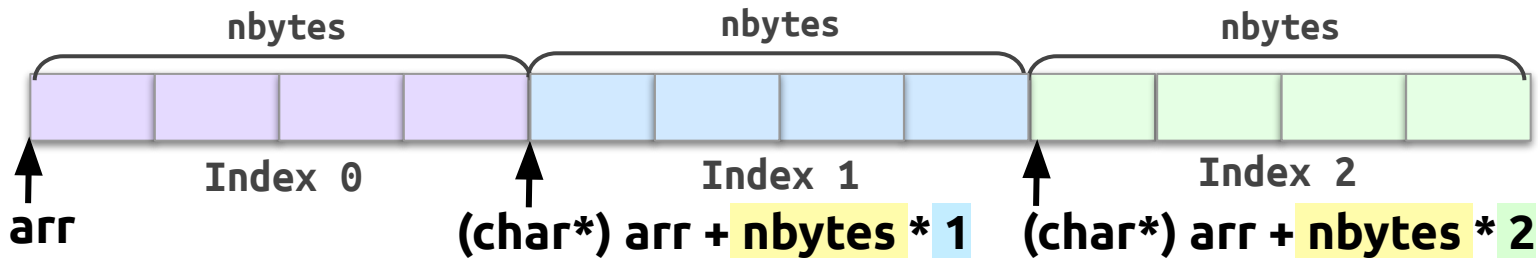
Consistency: Points to a **char** (only 1 byte), so we can work/add in terms of bytes!

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)
```



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

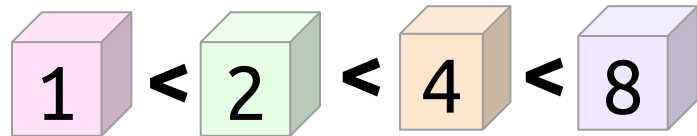


Pointer to 2nd element

We gotta teach C how to sort 🤔



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 if **a** and **b** are **equal**

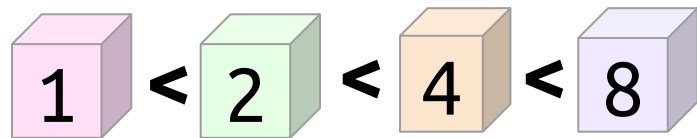
❑ <0 (Negative) if **a** comes before **b**

❑ >0 (Positive) if **a** comes after **b**

We gotta teach C how to sort 🤔



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



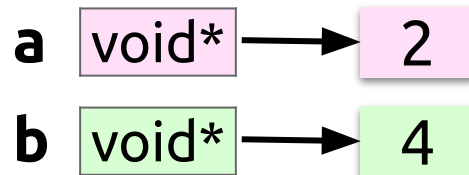
As such, to sort, we have to define a **comparison function** `cmp(a, b)`

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!**



Won't put you through another Star Wars scroll lol

Cardinal Rule of Today's Lab



👉 For callback comparison functions, **we must pass pointers to whatever values we want to compare**, not the values themselves. Try not to forget :)

Example: We have callback **int compare_strings(const void *a, const void *b)**. We want to compare strings str1 and str2, e.g., char* str1 = "blue", str2 = "red"

How should we call the function?

- `compare_strings(str1, str2)`
- `compare_strings(&str1, &str2)`



Won't put you through another Star Wars scroll lol

Cardinal Rule of Today's Lab



👉 For callback comparison functions, **we must pass pointers to whatever values we want to compare**, not the values themselves. Try not to forget :)

Example: We have callback **int compare_strings(const void *a, const void *b)**. We want to compare strings str1 and str2, e.g., char* str1 = "blue", str2 = "red"

How should we call the function?



compare_strings(str1, str2)

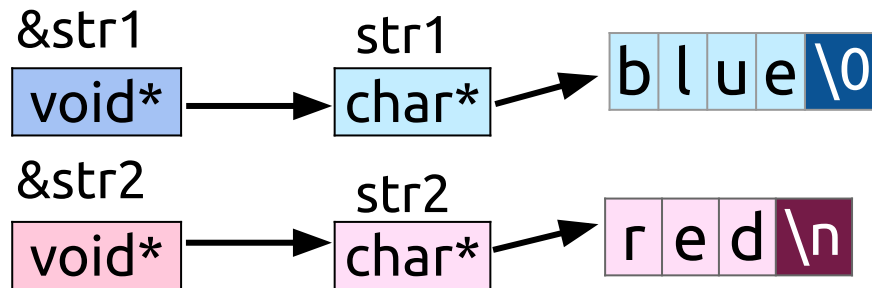


compare_strings(&str1, &str2)



As we're comparing strings, we have to pass **pointers to each string**.

A string is a **char***. Thus, we have to call the function with **char**** inputs.



&str1, &str2 are really **char**** pointers!

Happy to discuss any questions!

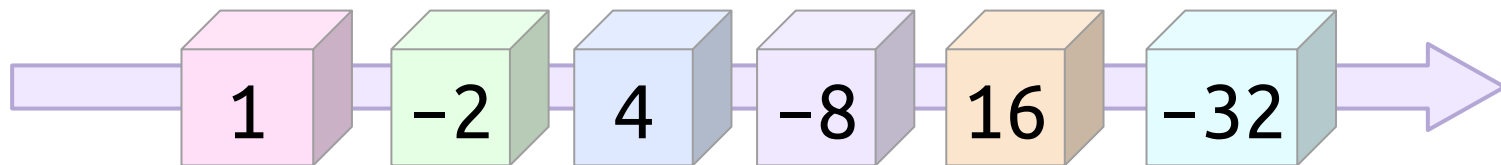
Generics, void ptrs, function ptrs, comparison functions, sorting, etc.





Check-Off Exercise: Absolute Value Sort!

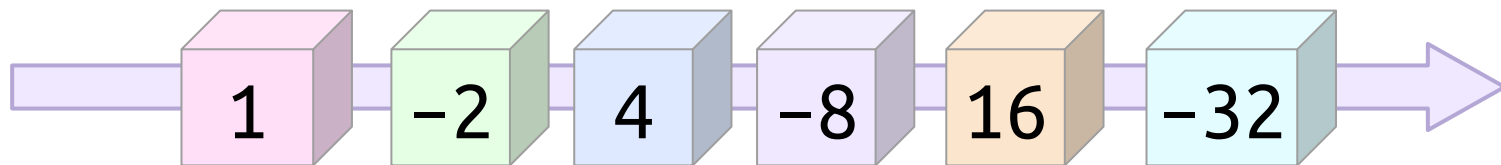
What would a callback **comparison function** (that can be passed to **qsort**) look like, if we wanted to arrange an array of **ints** in order of **increasing absolute value from left to right**? *Hint – we have a built-in `abs()` function!*





Check-Off Exercise: Absolute Value Sort!

What would a callback **comparison function** (that can be passed to **qsort**) look like, if we wanted to arrange an array of **ints** in order of **increasing absolute value from left to right**? *Hint – we have a built-in `abs()` function!*



```
int compare_abs(const void *a, const void *b){
```

```
    // your code!
```

```
}
```

Reminder! This should return:

0 if **a** and **b** are **equal**

<0 if **a** comes before **b**

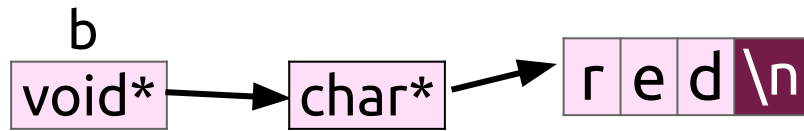
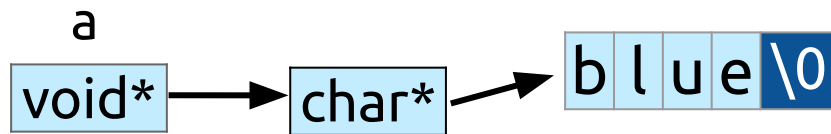
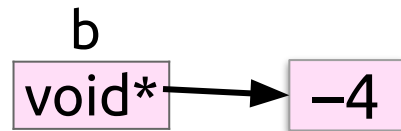
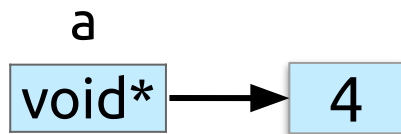
>0 if **a** comes after **b**



Writing Comparison Function: 3 Key Steps!

- 1 Cast the **void*** argument(s) to a known pointer type
- 2 Deference the **typed pointer** to access the value
- 3 Compare values to determine the result to return

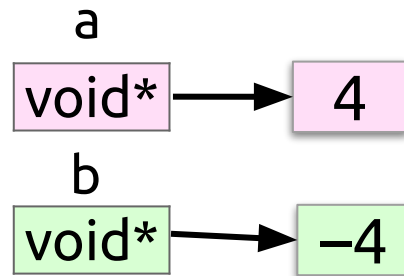
Note: Steps 1 and 2 are often combined to cast and deference in one expression!





Writing compare_abs using trilogy of steps

- 1 Cast the **void*** argument(s) to a known pointer type
- 2 Deference the **typed pointer** to access the value
- 3 Compare **values** to determine the result to return



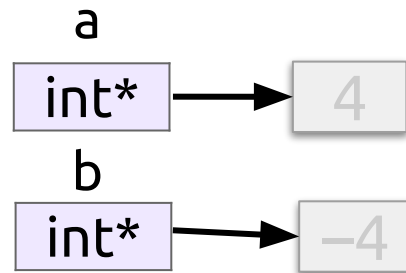
```
int compare_abs(const void *a, const void *b){
```

```
}
```



Writing compare_abs using trilogy of steps

- 1 Cast the **void*** argument(s) to a known pointer type
- 2 Deference the **typed pointer** to access the value
- 3 Compare **values** to determine the result to return



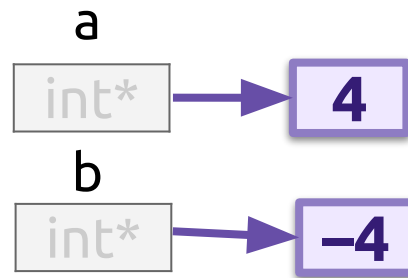
👉 Note the inputs *a* and *b* are pointers to *int*, so **cast to *int****

```
int compare_abs(const void *a, const void *b){  
    (int *)a  
    (int *)b  
  
}
```



Writing compare_abs using trilogy of steps

- 1 Cast the **void*** argument(s) to a known pointer type
- 2 **Deference the typed pointer to access the value**
- 3 Compare values to determine the result to return



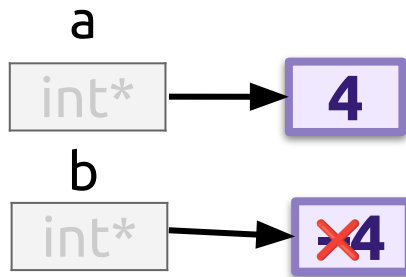
Note these are **int values**

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



Writing compare_abs using trilogy of steps

- 1 Cast the **void*** argument(s) to a known pointer type
- 2 Dereference the **typed pointer** to access the value
- 3 **Compare values** to determine the result to return



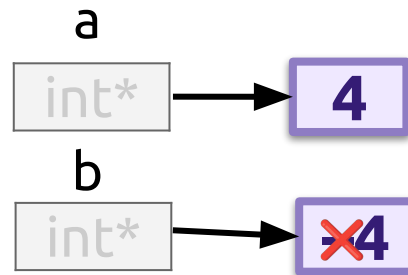
Note the absolute value *erases* *negative signs!* e.g., $\text{abs}(-4) = 4$

```
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);  
}
```



Writing compare_abs using trilogy of steps

- 1 Cast the **void*** argument(s) to a known pointer type
- 2 Deference the **typed pointer** to access the value
- 3 **Compare values** to determine the result to return

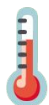


Note the absolute value *erases* *negative signs!* e.g., $\text{abs}(-4) = 4$

```
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);
```

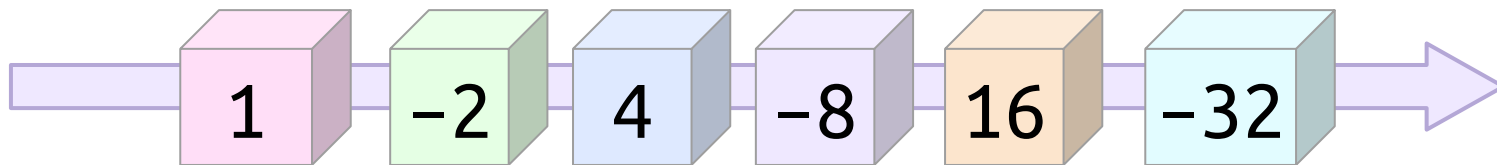
This compares the numbers
based on their absolute values!

} Can also do **return abs(*(int *)a) - abs(*(int *)b);** all in 1 line if you'd like :)



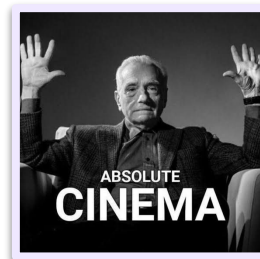
Solution: compare_abs function ✓

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**!



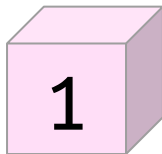
```
int compare_abs(const void *a, const void *b){  
    int a_val = *(int *)a;  
    int b_val = *(int *)b;  
    return abs(a_val) - abs(b_val);  
}
```

Validating compare_abs For 3 Cases

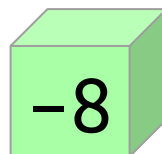


```
int compare_abs(const void *a, const void *b){  
    return abs(*(int *)a) - abs(*(int *)b);  
}
```

$$1 - 8 = -7$$



VS

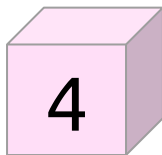


When $|a| < |b|$:

Return is **negative**, so

a goes before **b** ✓

$$4 - 4 = 0$$



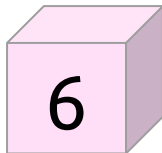
VS



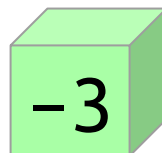
When $|a| = |b|$:

Return is **0**, so **a** is equal to **b**, order-wise ✓

$$6 - 3 = 3$$



VS



When $|a| > |b|$:

Return is **positive**, so

a goes after **b** ✓



Next, our fun lab exercises for today!

(notorious)

But first the Ben tradition of mochi diplomacy!



As everyone's been working so incredibly hard this quarter 💜, and I have too many mochi boxes (i.e. a spending problem in Tokyo), everyone gets a mochi!



Anonymous Lab Feedback Form!



Each quarter as a TA, I always explore 3 or more ways to improve my teaching, so I'd greatly appreciate any feedback! Tysm in advance 🤍



Exercise 1: void* and pitfalls

Look through the `lab4/generic.c` file, and its scintillant examples of using `qsort` on all kinds of data. **Review each comparison function** to see how they work, and **write your own next :)**

Work in groups if you can, and let's get sorting!



~ 25 min + 5 min all-lab discussion



Ciudad de México



London



San Francisco



Tokyo (東京)

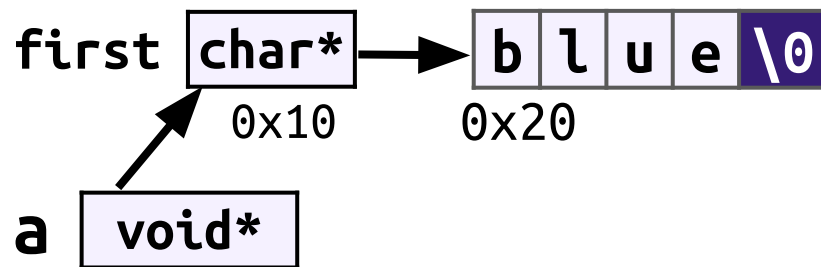
Example: Sort City Names





Q1: compare_strings_alphabetically

This function is trying to compare strings by alphabetical order. Does it work?



```
int compare_strings_alphabetically(const void *a, const void *b) {  
    const char *first = *(const char **)a;  
    const char *second = *(const char **)b;  
    return strcmp(first, second);  
}
```

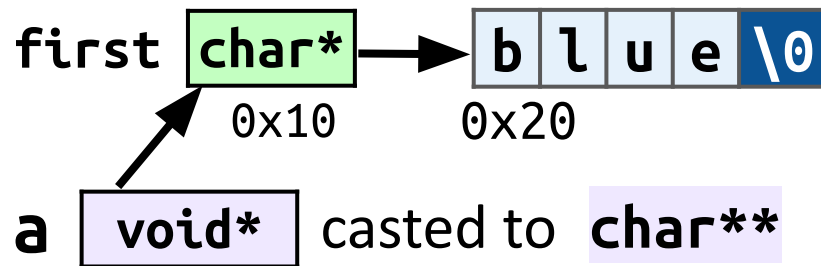


Q1: compare_strings_alphabetically

This function is trying to compare strings by alphabetical order. Does it work?

This works slay as we are passing in a pointer to a string (`char ** type`).

So casting and dereferencing once gives us the string itself (`char *`) to compare.

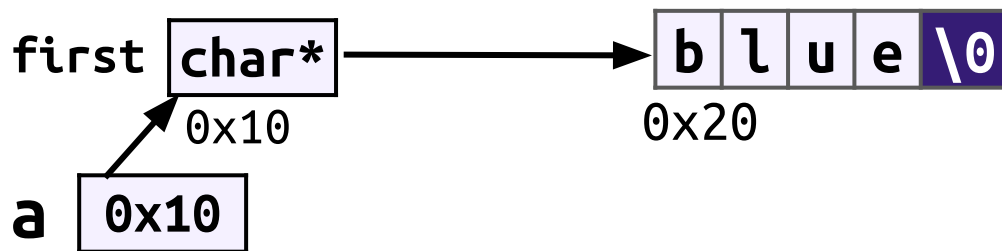


```
int compare_strings_alphabetically(const void *a, const void *b) {  
    const char *first = *(const char **)a;  
    const char *second = *(const char **)b;  
    return strcmp(first, second);  
}
```



Q1: compare_letters_alphabetically

This function tries to compare strings by **just their starting letter**. Does it work?



```
int compare_letters_alphabetically(const void *a, const void *b) {  
    unsigned char first = *(const char *)a;  
    unsigned char second = *(const char *)b;  
    return first - second;  
}
```

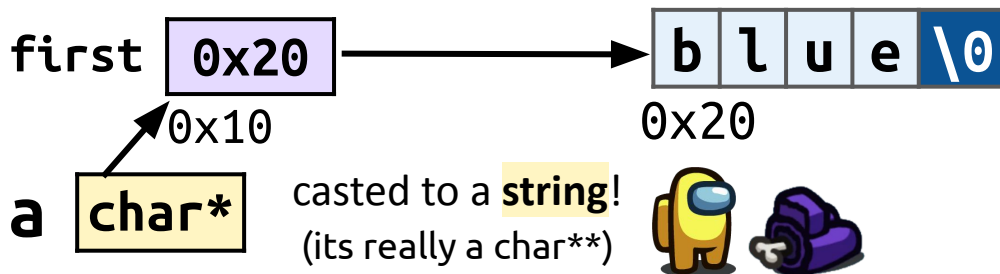


Q1: compare_letters_alphabetically

This function tries to compare strings by **just their starting letter**. Does it work?

No! :(This is cooked as we are casting to the wrong type `char*`

8-byte address (0x20) of 'b' we point to



```
int compare_letters_alphabetically(const void *a, const void *b) {  
    unsigned char first = *(const char *)a;  
    unsigned char second = *(const char *)b;  
    return first - second;  
}
```

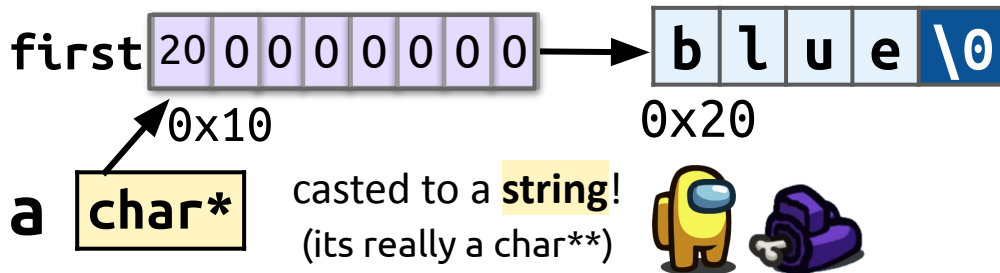


Q1: compare_letters_alphabetically

This function tries to compare strings by **just their starting letter**. Does it work?

note: 2 hex digits per byte

8-byte address (0x20) of 'b' we point to



This **tricks a** into believing it's a **string** (and **0x20** is the buffer of "characters" it points to).

casted to a **string!**
(its really a char**)



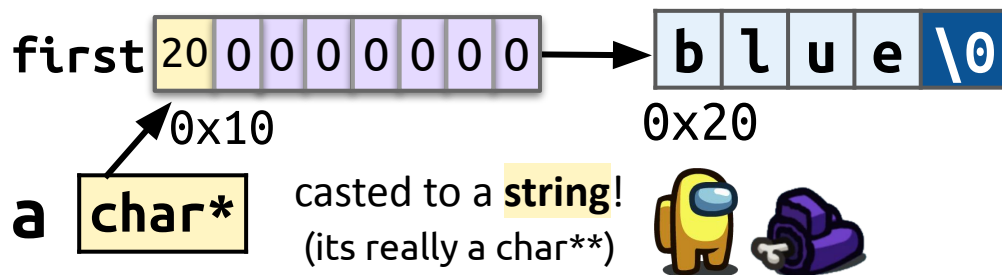
```
int compare_letters_alphabetically(const void *a, const void *b) {  
    unsigned char first = *(const char *)a;  
    unsigned char second = *(const char *)b;  
    return first - second;  
}
```



Q1: compare_letters_alphabetically

This function tries to compare strings by **just their starting letter**. Does it work?

8-byte address (0x20) of 'b' we point to



```
int compare_letters_alphabetically(const void *a, const void *b) {  
    unsigned char first = *(const char *)a;  
    unsigned char second = *(const char *)b;  
    return first - second;  
}
```

Dereferencing *a* gives us the first byte of **0x20**, or the yellow box **20**. **This isn't the value we want!**
We want the first letter **'b'** of blue.

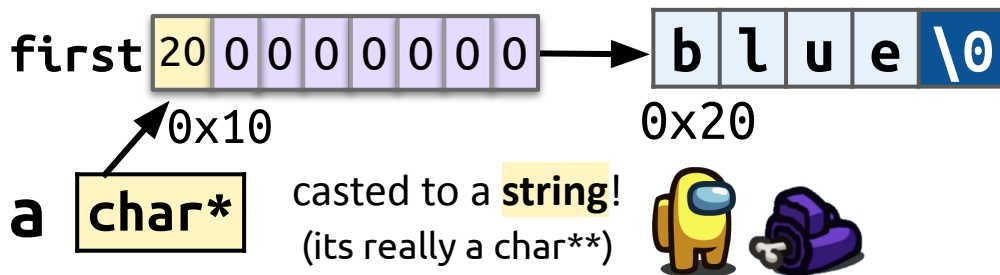


Q1: compare_letters_alphabetically

This function tries to compare strings by **just their starting letter**. Does it work?



8-byte address (0x20) of 'b' we point to



```
int compare_letters_alphabetically(const void *a, const void *b) {  
    unsigned char first = *(const char *)a;  
    unsigned char second = *(const char *)b;  
  
    In short, don't imposter your variables!  
    Cast them correctly to the right type :)  
}
```



Q2: Comparing Strings by First/Main Character

We want to sort strings in alphabetical order **purely by their first character!**

blue = beige < green < ruby = red

From Q1, we can't use `compare_letters_alphabetically`.



Are we cooked?



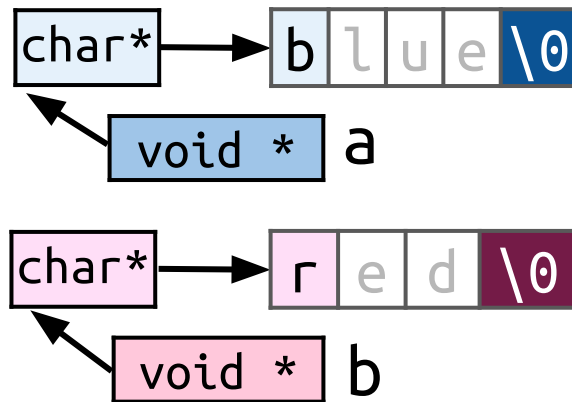
Nope, we can write an entirely **new comparison function** between strings!

- ❑ We must pass **pointers to each string as input**
- ❑ As a string is a **char ***, to compare strings, we must cast/call the function with a **char**** type





Writing compare_strings_by_first_character



```
int compare_strings_by...(const void *a, const void *b){
```

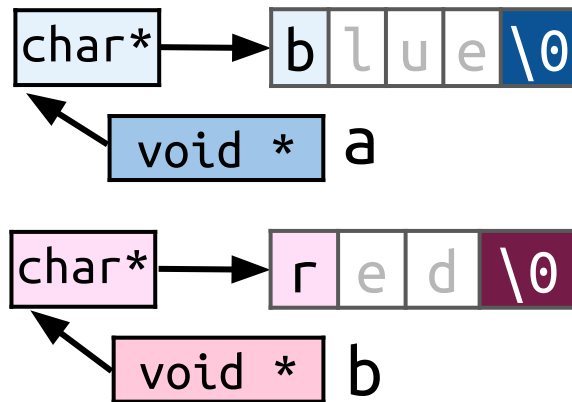
```
}
```



Writing `compare_strings_by_first_character`

Let's follow the 3 key steps!

- 1 Cast the **`void*`** argument(s) to a known pointer type
- 2 Dereference the **typed pointer** to access the value
- 3 Compare **values** to determine the result to return



```
int compare_strings_by...(const void *a, const void *b){
```

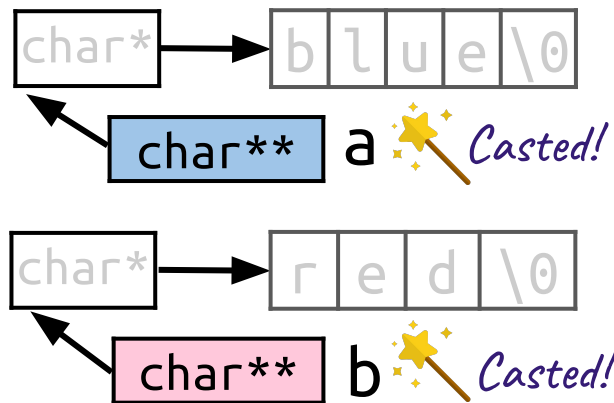
```
}
```



Writing compare_strings_by_first_character

Let's follow the 3 key steps!

- 1 Cast the **void*** argument(s) to a known pointer type
- 2 Dereference the typed pointer to access the value
- 3 Compare values to determine the result to return



```
int compare_strings_by...(const void *a, const void *b){
```

```
    (const char **) a;  
    (const char **) b;
```

👉 Note the inputs a, b are pointers to strings / char*, so cast to char**

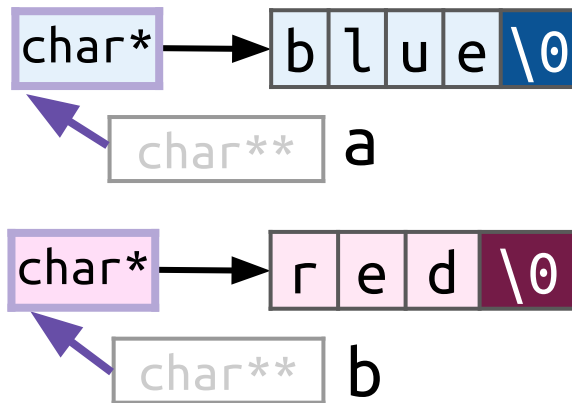
```
}
```



Writing compare_strings_by_first_character

Let's follow the 3 key steps!

- 1 Cast the **void*** argument(s) to a known pointer type
- 2 Dereference the **typed pointer** to access the value
- 3 Compare values to determine the result to return



```
int compare_strings_by...(const void *a, const void *b){
```

```
    const char *str_a = *(const char **) a;
```

```
    const char *str_b = *(const char **) b;
```

```
}
```

We have used the magic spell of dereferencing, and now we have **the strings we want to compare!**

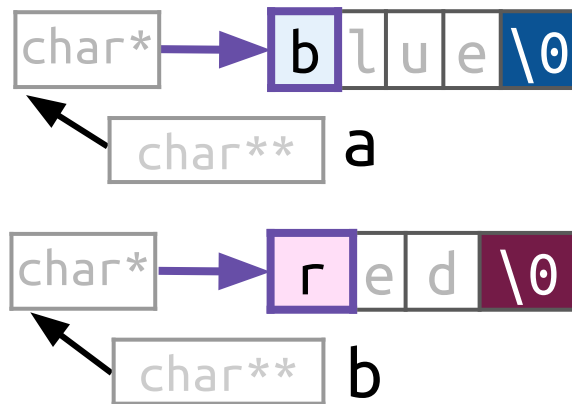




Writing compare_strings_by_first_character

Let's follow the 3 key steps!

- 1 Cast the **void*** argument(s) to a known pointer type
- 2 Dereference the typed pointer to access the value
- 3 **Compare values** to determine the result to return



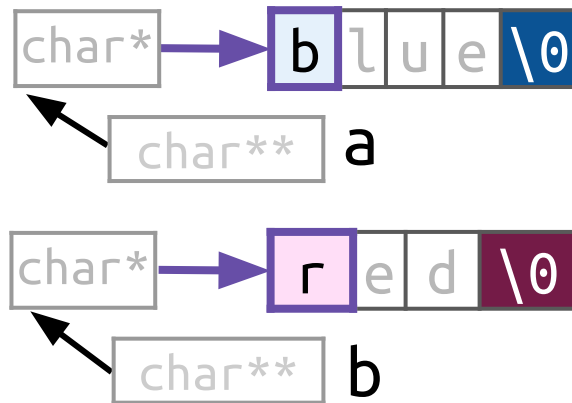
```
int compare_strings_by...(const void *a, const void *b){  
  
    const char *str_a = *(const char **) a;  
    const char *str_b = *(const char **) b;  
    return str_a[0] - str_b[0];  
}
```



Writing compare_strings_by_first_character

Let's follow the 3 key steps!

- 1 Cast the **void*** argument(s) to a known pointer type
- 2 Dereference the typed pointer to access the value
- 3 **Compare values** to determine the result to return



```
int compare_strings_by...(const void *a, const void *b){
```

```
    const char *str_a = *(const char **) a;
```

```
    const char *str_b = *(const char **) b;
```

```
    return str_a[0] - str_b[0];
```

This compares the strings **based on their first chars!**

```
} Alternatively, can write return *str_a - *str_b; (dereference to get first char!)
```

slay

Solution: compare_strings_by_first_character ✓

We want to write a cmp function that can help sort strings alphabetically but **only based on their first characters!** #mainCharacterEnergy

blue = beige < green < ruby = red

```
int compare_strings_by...(const void *a, const void *b){  
    const char *str_a = *(const char **) a;  
    const char *str_b = *(const char **) b;  
    return str_a[0] - str_b[0];  
} // or just return **(const char **) a - **(const char **) b;
```

Note: This version respects the inputs as pointers to their respective strings, or char** types

Happy to discuss any questions!

Writing cmp functions for strings, testing, handling void stars, etc.





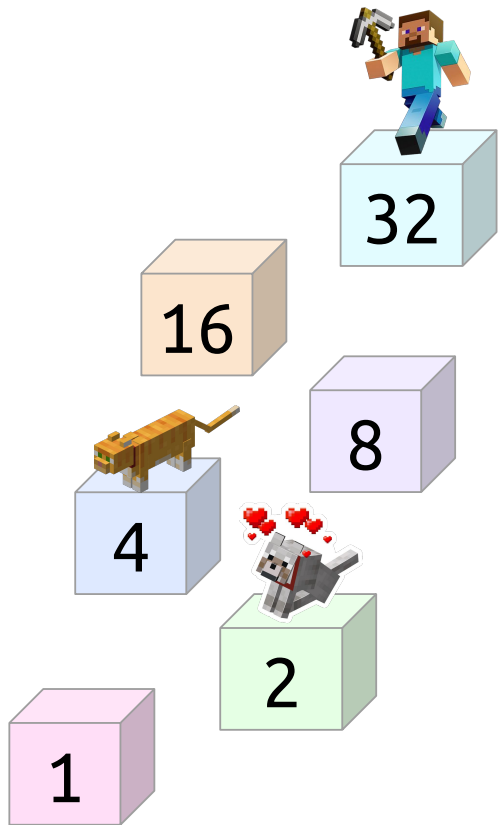
Exercise 2: Bsearch Bug

Starter code in **bsearch_bug.c**. You'll work with a standard generic function called **bsearch**, which you'll use in **Assign4**. Take a look at the code, and follow the handout to investigate the error!

And of course, work in groups if you can, and please feel free to ask me anything as you work!



~ 20 min + 10 min all-lab discussion





bsearch: Library function documentation

C library function to search for an element in a sorted array using **binary search**

Rule: The array must be sorted according to the same **cmp** function that the search is using

```
BSEARCH(3)                                Linux Programmer's Manual                                BSEARCH(3)

NAME
    bsearch - binary search of a sorted array

SYNOPSIS
    #include <stdlib.h>

    void *bsearch(const void *key, const void *base,
                  size_t nmem, size_t size,
                  int (*compar)(const void *, const void *));

DESCRIPTION
    The bsearch() function searches an array of nmem objects, the initial member of which is pointed to by base, for a member that matches the object pointed to by key. The size of each member of the array is specified by size.

    The contents of the array should be in ascending sorted order according to the comparison function referenced by compar. The compar routine is expected to have two arguments which point to the key object and to an array member, in that order, and should return an integer less than, equal to, or greater than zero if the key object is found, respectively, to be less than, to match, or be greater than the array member.
```



bsearch: Library function documentation

C library function to search for an element in a sorted array using **binary search**

Rule: The array must be sorted according to the same cmp function that the search is using

BSEARCH(3) Linux Programmer's Manual BSEARCH(3)

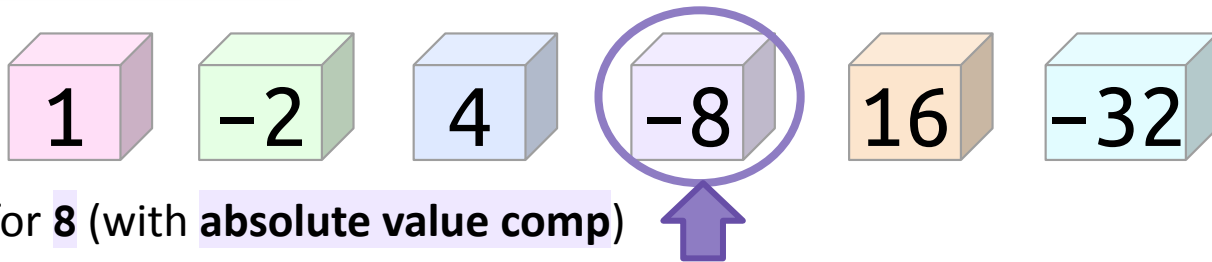
NAME
bsearch – binary search of a sorted array

SYNOPSIS
`#include <stdlib.h>`

`void *bsearch(const void *key, const void *base,
size_t nmem, size_t size,
int (*compar)(const void *, const void *));`

DESCRIPTION
The `bsearch()` function searches an array of `nmem` objects, the initial member of which is pointed to by `base`, for a member that matches the object pointed to by `key`. The size of each member of the array is specified by `size`.

The contents of the array should be in ascending sorted order according to the comparison function referenced by `compar`. The `compar` routine is expected to have two arguments which point to the `key` object and to an array member, in that order, and should return an integer less than, equal to, or greater than zero if the `key` object is found, respectively, to be less than, to match, or be greater than the array member.





However, in `bsearch_bug.c`:

```
31     qsort(words, n, sizeof(words[0]), compare_first_characters);
32
33     printf("Sorted by first letter: \n");
34     for (int i = 0; i < n; i++) {
35         printf("%s ", words[i]);
36     }
37     printf("\n");
38
39     printf("Looking for a word that starts with %s: \n", key);
40     char **found = bsearch(key, words, n, sizeof(words[0]), compare_asymmetric);
```



However, in `bsearch_bug.c`:

`sort` and `search` *should* use the same comparison function.

Why are they different here? (`compare_first_characters` and `compare_asymmetric`) Try to figure it out!

```
31  qsort(words, n, sizeof(words[0]), compare_first_characters);
32
33  printf("Sorted by first letter: \n");
34  for (int i = 0; i < n; i++) {
35      printf("%s ", words[i]);
36  }
37  printf("\n");
38
39  printf("Looking for a word that starts with %s: \n", key);
40  char **found = bsearch(key, words, n, sizeof(words[0]), compare_asymmetric);
```





What happens if we try using same cmp?

Without making any other changes to the code!

```
31 qsort(words, n, sizeof(words[0]), compare_first_characters);
```

```
... ..
```

```
40 char **found = bsearch(key, words, n, sizeof(words[0]), compare_first_characters);
```



What happens rn if we try using same cmp?

Without making any other changes to the code!

```
31 qsort(words, n, sizeof(words[0]), compare_first_characters);  
    ... ..  
40 char **found = bsearch(key, words, n, sizeof(words[0]), compare_first_characters);
```



We get a bug!

In fact, it's the most common **Assign4** bug!



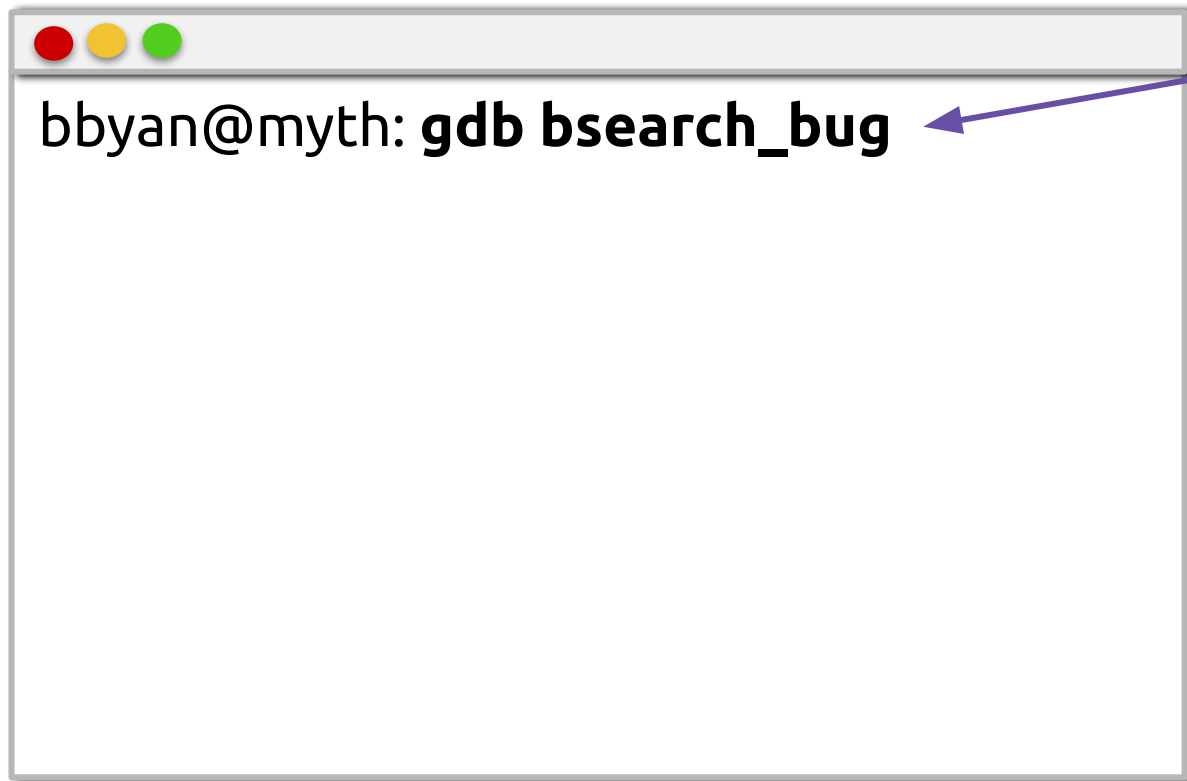
```
myth$ ./bsearch_bug  
Sorted by first letter:  
blue green red yellow  
Looking for a word that  
starts with g:
```

Segmentation fault!!!





Who's (seg)fault? A GDB whodunit mystery

A terminal window with a grey title bar containing three colored circles (red, yellow, green). The terminal text shows a user prompt and a command.

```
bbyan@myth: gdb bsearch_bug
```

Launch gdb 🚀 and run the buggy program!



Who's (seg)fault? A GDB whodunit mystery



```
bbyan@myth: gdb bsearch_bug
```

```
(gdb) run
```

```
Program SIGSEGV ⚠, Segmentation fault
```

```
compare_first_characters(p, q)
```

```
15 return **(const char**)p - **(const char**)q;
```

Launch gdb 🚀 and run the buggy program!

The crash is inside the comparison function – let's investigate **what p and q are here!**



Who's (seg)fault? A GDB whodunit mystery

```
bbyan@myth: gdb bsearch_bug
```

```
(gdb) run
```

```
Program SIGSEGV ⚠️, Segmentation fault
```

```
compare_first_characters(p, q)
```

```
15 return **(const char**)p - **(const char**)q;
```

```
(gdb) print p
```

```
(const void *) 0x55555555601e
```

```
(gdb) print **(const char**)p
```

```
Cannot access memory at address 0x6465...
```

Launch gdb 🚀 and run the buggy program!

The crash is inside the comparison function – let's investigate **what p and q are here!**

It looks like **p** is having trouble being dereferenced twice.

So perhaps **p** is not really a pointer to a string? (**not a char****)



Who's (seg)fault? A GDB whodunit mystery

(gdb) **backtrace**

```
#0 compare_first_characters(p=..., q=...), line 15  
#1 bsearch(_key=..., _base=..., _compare=...)  
#2 main(argc = 1, argv=...), line 40
```

Let's see where that the input **p** comes from – by seeing where it was passed in from.



Who's (seg)fault? A GDB whodunit mystery

(gdb) **backtrace**

```
#0 compare_first_characters(p=..., q=...), line 15  
#1 bsearch(_key=..., _base=..., _compare=...)  
#2 main(argc = 1, argv=...), line 40
```

(gdb) **up 2**

Went from compare_first_...() to main():

```
char **found = bsearch(key, words, n,  
sizeof(words[0]), compare_first_characters);
```

Let's see where that the input **p** comes from – by seeing where it was passed in from.

The comparison is ran by **bsearch()** – let's examine how it's called in main!

"up 2" ascends up 2 stack frames to main()!



Who's (seg)fault? A GDB whodunit mystery

(gdb) **backtrace**

```
#0 compare_first_characters(p=..., q=...), line 15
#1 bsearch(_key=..., _base=..., _compare=...)
#2 main(argc = 1, argv=...), line 40
```

(gdb) **up 2**

Went from compare_first_...() to main():

```
char **found = bsearch(key, words, n,
sizeof(words[0]), compare_first_characters);
```

(gdb) **print key**

```
0x55555555601e "g"
```

Let's see where that the input **p** comes from – by seeing where it was passed in from.

The comparison is ran by **bsearch()** – let's examine how it's called in main!

"up 2" ascends up 2 stack frames to main()!

We print **key** – it's the same pointer as **p**, but it **seems to be a char* ("g")**, not a **char****!!



Reflection: Original Workaround

- ❏ `compare_asymmetric` was comparing values of different types!

```
18 int compare_asymmetric(const void *p, const void *q) {  
19     return *(const char *)p - **(const char **)q;  
20 }
```



Reflection: Original Workaround

- ❑ `compare_asymmetric` was comparing values of different types!

```
18 int compare_asymmetric(const void *p, const void *q) {  
19     return *(const char *)p - **(const char **)q;  
20 }
```

- ❑ When we investigate what's really being passed into `bsearch`:

```
char *words[] = {"red", "blue", "green", "yellow"};  
int n = sizeof(words) / sizeof(*words);  
char *key = "g";  
...  
char **found = bsearch(key, words, n, sizeof(words[0]), compare_asymmetric);
```

char* (string)

char (pointer to string)**



hmm...



We're getting there!



Check-Off Q: How does calling bsearch work?

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



Check-Off Q: How does calling bsearch work?

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

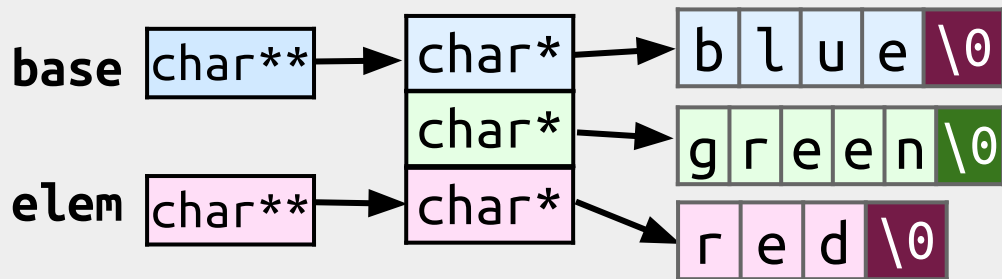
- **key**: A **pointer to the value** we're searching for, e.g., "gem"
- **base**: **Pointer to base / starting element of array** to search
- **nmemb**: Total # of elements in array
- **size/width**: # of bytes per element, e.g., 4 for array for ints
- **compar**: The **comparison function** used to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

The return value is:

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

Our cmp function expects two inputs of same **char**** type:

-  **key** pointer
-  pointer to current **elem** to be passed in, and compared!





Check-Off Q: How does calling bsearch work?

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

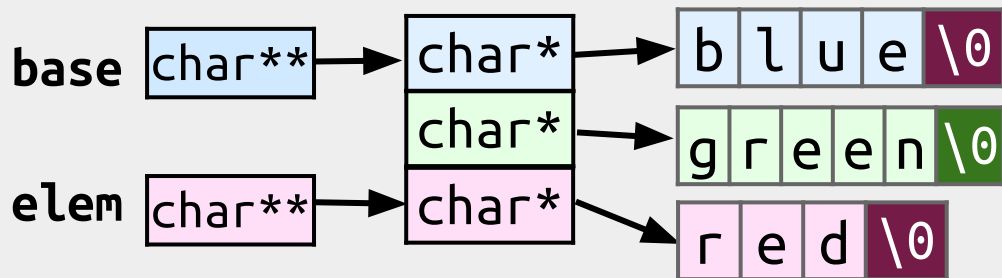
- **key**: A **pointer to the value** we're searching for, e.g., "gem"
- **base**: **Pointer to base / starting element of array** to search
- **nmemb**: Total # of elements in array
- **size/width**: # of bytes per element, e.g., 4 for array for ints
- **compar**: The **comparison function** used to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

The return value is:

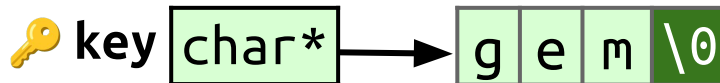
- If found, a **pointer to the matching element** in array
- If no match, **NULL** pointer is returned

Our cmp function expects two inputs of same **char**** type:

-  **key** pointer
-  pointer to current **elem** to be passed in, and compared!



Is this the case for **key** as defined here?





Check-Off Q: Our Fix! (calling bsearch correctly)

1. Use `compare_first_characters` in both, should be consistent! :)

```
qsort(words, n, sizeof(words[0]), compare_first_characters);
```

... ..

```
char **found = bsearch(key, words, n, sizeof(words[0]), compare_first_characters);
```



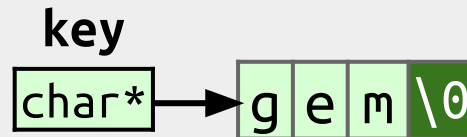
Check-Off Q: Our Fix! (calling bsearch correctly)

1. Use `compare_first_characters` in both, should be consistent! :)

```
qsort(words, n, sizeof(words[0]), compare_first_characters);  
... ..  
char **found = bsearch(key, words, n, sizeof(words[0]), compare_first_characters);
```

2. But the **key parameter** to `bsearch` must be of pointer type `char**`, as that's what the `cmp` function expects when comparing our key against each array element!

```
char *key = "gem";  
char **found = bsearch(???, words, n,  
    sizeof(words[0]), compare_first_characters);
```





Check-Off Q: Our Fix! (calling bsearch correctly)

1. Use `compare_first_characters` in both, should be consistent! :)

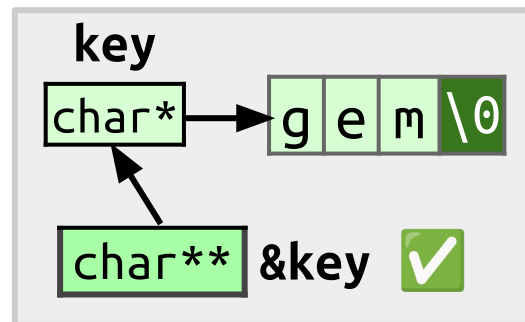
```
qsort(words, n, sizeof(words[0]), compare_first_characters);  
... ..  
char **found = bsearch(key, words, n, sizeof(words[0]), compare_first_characters);
```

2. But the **key parameter** to `bsearch` must be of pointer type `char**`, as that's what the cmp function expects when comparing our key against each array element!

```
char *key = "gem";  
char **found = bsearch(&key, words, n,  
                      sizeof(words[0]), compare_first_characters);
```

👉 Thus, we pass a **pointer to the key (&key)**, instead of the key itself.

As a `char**`, it can be de-referenced twice by the cmp function — **slay, bug is fixed!**





Exercise 3: Printing Arrays with GDB



~ 10 min

We'll do this synchronously, as printing arrays will be very helpful for upcoming assignments! ✨

We'll work with the **gdb_practice** program, and its accompanying **gdb_practice.c** code file.

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

10	20	30	40
Index 0	Index 1	Index 2	Index 3

void* ptr

Arrays: The syntax for printing **nelem** elements starting from index is:

```
(gdb) print arr[index]@nelems
```

starting element

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
```

Example: To print out the full array above 👉 **(gdb) print ((int *)ptr)[0] @ 4**



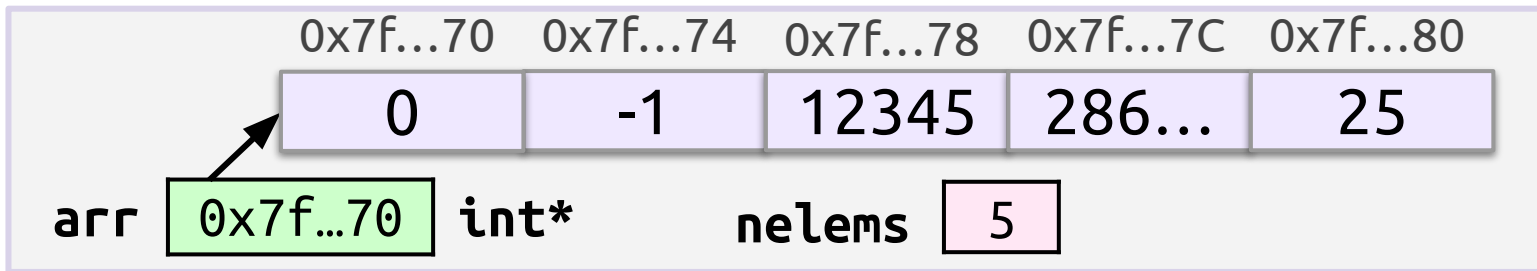
Cursed GDB Technique for Printing Arrays!



```
bbyan@myth: gdb gdb_practice  
(gdb) break my_function  
(gdb) run  
Breakpoint, my_function(arr=0x7ff...,nelems=5)  
(gdb) print arr  
0x7fffffffe770  
(gdb) print arr[0] @ nelems  
{0, -1, 12345, 286331153, 25} ✓
```

Argh, this just prints out the array's **pointer address**, not the elements! Lame

Here, we print **nelems** starting from **index 0** i.e. the **whole array**.





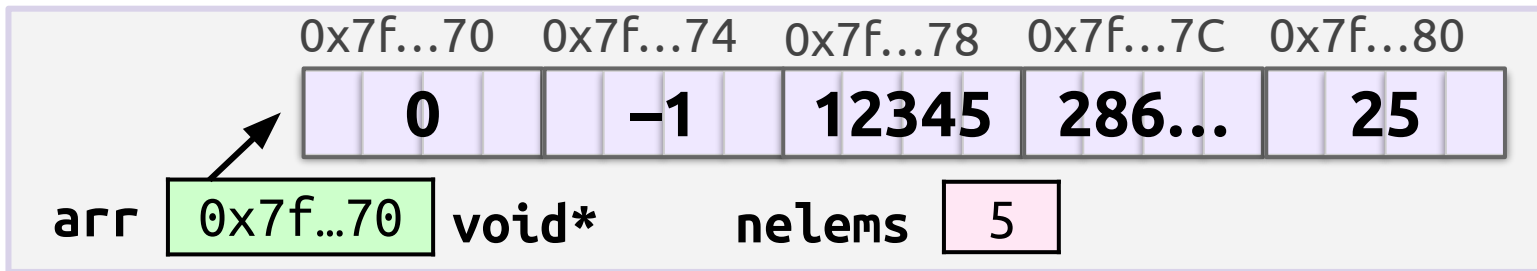
More Sorcery! Printing Array with Void* Pointer



```
(gdb) break my_function_generic
(gdb) continue
Breakpoint, my_function_generic(ptr=...,nelems=5)
(gdb) print ptr[0]@nelems
! Error! Attempt to dereference a generic ptr
(gdb) print ((int*)ptr)[0] @ nelems
{0, -1, 12345, 286331153, 25} ✓
```

This didn't work, but we know **void* ptr** is pointing to **array of integers**.

Here, we first **cast to an int***, then print **nelems** starting from **index 0** i.e. the **whole array**.





Cardinal Rule of Lab 4 (sry for bad photoshop)

*To compare values in a
callback comparison function,*

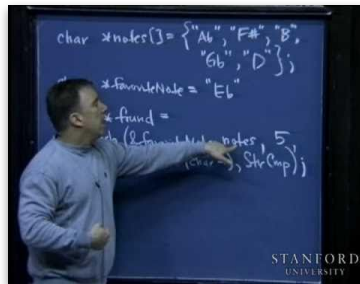
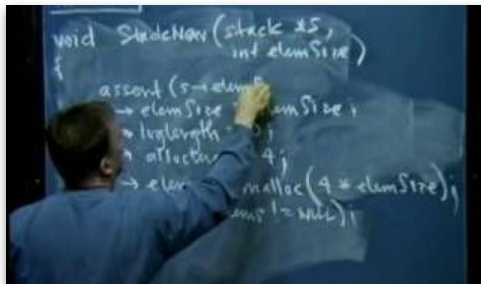
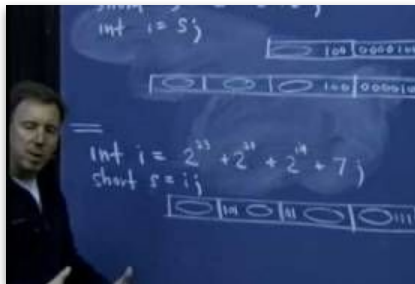
we must

pass pointers to the values !

Don't forget :)

Text: To compare values in a callback comparison function, we must **pass pointers to the values!** Don't forget :)

Thank you and have a beautiful week! 🌟



- Please reach out to me if you have any questions (e.g., on lab, midterm content), or topics you want covered!
-  Contact: bbyan@stanford.edu, email or over Slack
- Hope you enjoyed the mochi, and best of luck on our midterm! **YOU'RE GOING TO DO AMAZING!!** 💖



“Droids and void stars, there is where the fun begins” – Anakin

“Comparison functions are the thief of joy” – Teddy Roosevelt, I think