

CS107, Lecture 14

Function Pointers, Continued

Reading: K&R 5.11

This document is copyright (C) Stanford Computer Science and Nick Troccoli, licensed under Creative Commons Attribution 2.5 License. All rights reserved.

Based on slides created by Cynthia Lee, Chris Gregg, Jerry Cain, Lisa Yan and others.

NOTICE RE UPLOADING TO WEBSITES: This content is protected and may not be shared, uploaded, or distributed. (without expressed written permission)

CS107 Topic 4

How can we use our knowledge of memory and data representation to write code that works with any data type?

Why is answering this question important?

- Writing code that works with any data type lets us write more generic, reusable code while understanding potential pitfalls (previously)
- Allows us to learn how to pass functions as parameters, a core concept in many languages (last time + today)

assign4: implement your own version of the **ls** command, a function to generically find and insert elements into a sorted array, and a program using that function to sort the lines in a file like the **sort** command.

Learning Goals

- Learn how to pass functions as parameters
- Learn how to write functions that accept functions as parameters

Lecture Plan

- **Recap and continuing:** Function Pointers
- **Example:** Count Matches

```
cp -r /afs/ir/class/cs107/lecture-code/lect14 .
```

Lecture Plan

- **Recap and continuing: Function Pointers**
- **Example: Count Matches**

```
cp -r /afs/ir/class/cs107/lecture-code/lect14 .
```

Recap: Passing Functions as Parameters

1. Taking a function as a parameter lets us implement code where a piece of the logic can be customized by the caller.

Integer Bubble Sort

```
void bubble_sort_int(int *arr, size_t n, bool (*should_swap)(int, int)) {
    while (true) {
        bool swapped = false;
        for (size_t i = 1; i < n; i++) {
            if (should_swap(arr[i - 1], arr[i])) {
                swapped = true;
                int tmp = arr[i - 1];
                arr[i - 1] = arr[i];
                arr[i] = tmp;
            }
        }
        if (!swapped) {
            return;
        }
    }
}
```

bubble_sort_int lets the caller specify any sort order they want via the **should_swap** function.

Recap: Passing Functions as Parameters

1. Taking a function as a parameter lets us implement code where a piece of the logic can be customized by the caller.
- 2. When the caller wants to use our function, they pass in the name of a function as a parameter that they implemented specifically for what they want to do that time.**

Function Pointers

```
// my_program.c
#include "bubblesort.h"

bool sort_ascending(int first_num,
                    int second_num) {
    return first_num > second_num;
}

int main(int argc, char *argv[]) {
    int nums[] = {4, 2, -5, 1, 12, 56};
    int nums_count = sizeof(nums) /
                      sizeof(nums[0]);
    bubble_sort_int(nums, nums_count,
                    sort_ascending);
    ...
}
```

```
// bubblesort.c

void bubble_sort_int(int *arr, size_t
n, bool (*should_swap)(int, int)) {
    ...
}
```

bubble_sort_int is written generically. When someone imports our function into their program, they will call it specifying the sort ordering they want that time.

Recap: Passing Functions as Parameters

1. Taking a function as a parameter lets us implement code where a piece of the logic can be customized by the caller.
2. When the caller wants to use our function, they pass in the name of a function as a parameter that they implemented specifically for what they want to do that time.
3. **If we want to write a fully-generic version of bubble sort that works with any data type, we must pick *one* signature for our comparison function.**

Generic Bubble Sort

To write one generic bubblesort function, we must create one function signature that works for any scenario with any type.

```
void bubble_sort(void *arr, size_t n,  
                size_t elem_size_bytes, bool (*should_swap)(??, ??));
```

Any functions passed in as the **should_swap** parameter must exactly match this signature.

Challenge: for **should_swap** parameters, C has no way to specify “a variable of any type”.

Recap: Function Pointers

```
void bubble_sort(void *arr, size_t n, size_t  
elem_size_bytes, bool (*cmp_fn)(void *, void *))
```

Solution: have comparison function take in **void *** parameters – these will be *pointers to the elements* being compared, not the elements themselves.

Functions handling generic data must use *pointers to the data they care about*, since any parameters must have *one type* and *one size*.

Generic Bubble Sort

```
void bubble_sort(void *arr, size_t n, size_t elem_size_bytes,
                 bool (*should_swap)(void *, void *)) {
    while (true) {
        bool swapped = false;
        for (size_t i = 1; i < n; i++) {
            void *p_prev_elem = (char *)arr + (i - 1) * elem_size_bytes;
            void *p_curr_elem = (char *)arr + i * elem_size_bytes;
            if (should_swap(p_prev_elem, p_curr_elem)) {
                swapped = true;
                swap(p_prev_elem, p_curr_elem, elem_size_bytes);
            }
        }
        if (!swapped) {
            return;
        }
    }
}
```

Passing Functions as Parameters

1. Taking a function as a parameter lets us implement code where a piece of the logic can be customized by the caller.
2. When the caller wants to use our function, they pass in the name of a function as a parameter that they implemented specifically for what they want to do that time.
3. If we want to write a fully-generic version of bubble sort that works with any data type, we must pick *one* signature for our comparison function.
4. **There is a common signature used for these “comparison functions” – instead of returning a bool, we return an int.**

Comparison Functions

Function pointers are used often in cases like this to compare two values of the same type. These are called **comparison functions**.

C has a standard format for comparison function, returning an **int** instead of a **bool** to provide more information about order. It's like **strcmp** - it should return:

- < 0 if first value should come before second value
- > 0 if first value should come after second value
- 0 if first value and second value are equivalent

```
int (*compare_fn)(void *, void *)
```

Generic Bubble Sort

```
void bubble_sort(void *arr, size_t n, size_t elem_size_bytes,
                int (*should_swap)(void *, void *)) {
    while (true) {
        bool swapped = false;
        for (size_t i = 1; i < n; i++) {
            void *p_prev_elem = (char *)arr + (i - 1) * elem_size_bytes;
            void *p_curr_elem = (char *)arr + i * elem_size_bytes;
            if (should_swap(p_prev_elem, p_curr_elem) > 0) {
                swapped = true;
                swap(p_prev_elem, p_curr_elem, elem_size_bytes);
            }
        }
        if (!swapped) {
            return;
        }
    }
}
```

Function Pointers

```
// my_program.c
#include "bubblesort.h"

int sort_ascending(void *ptr1, void
*ptr2) {
    ???
}

int main(int argc, char *argv[]) {
    int nums[] = {4, 2, -5, 1, 12, 56};
    int nums_count = sizeof(nums) /
        sizeof(nums[0]);
    bubble_sort(nums, nums_count,
        sizeof(nums[0]),
        sort_ascending);
    ...
}
```

```
// bubblesort.c

void bubble_sort(void *arr, size_t n,
                size_t elem_size_bytes,
                int (*should_swap)(void *, void *)) {
    ...
}
```

Key idea: now the comparison function is passed pointers to the elements being compared. How does the caller implement one?

Function Pointers

- Comparison function now takes in **void *** *pointers to the elements* being compared
- The caller implements a specific comparison function for use in a specific scenario with a specific type of array (e.g. sort ints in ascending order).
- Thus, the parameters are technically **void *** parameters, **but** the caller knows what they really are within the context of that comparison function!

Function Pointers

```
// my_program.c
#include "bubblesort.h"

// 0 if equal, (-) if first before
// second, (+) if first after second
int sort_ascending(void *ptr1, void
*ptr2) {

}

int main(int argc, char *argv[]) {
    int nums[] = {4, 2, -5, 1, 12, 56};
    int nums_count = sizeof(nums) /
        sizeof(nums[0]);
    bubble_sort(nums, nums_count,
        sizeof(nums[0]),
        sort_ascending);
    ...
}
```

```
// bubblesort.c

void bubble_sort(void *arr, size_t n,
                size_t elem_size_bytes,
                int (*should_swap)(void *, void *)) {
    ...
}
```

Key question: *in*
sort_ascending, what type
are **ptr1** and **ptr2** really?

Function Pointers

```
// my_program.c
#include "bubblesort.h"

// 0 if equal, (-) if first before
// second, (+) if first after second
int sort_ascending(void *ptr1, void
*ptr2) {

}

int main(int argc, char *argv[]) {
    int nums[] = {4, 2, -5, 1, 12, 56};
    int nums_count = sizeof(nums) /
        sizeof(nums[0]);
    bubble_sort(nums, nums_count,
        sizeof(nums[0]),
        sort_ascending);
    ...
}
```

```
// bubblesort.c

void bubble_sort(void *arr, size_t n,
                size_t elem_size_bytes,
                int (*should_swap)(void *, void *)) {
    ...
}
```

Key question: *in*
sort_ascending, what type
are **ptr1** and **ptr2** really?
*int **

Function Pointers

```
// my_program.c
#include "bubblesort.h"

// 0 if equal, (-) if first before
// second, (+) if first after second
int sort_ascending(int *ptr1, int
*ptr2) {
```

Unfortunately we can't do this,
since the signature must match
bubble sort exactly.

```
int nums_count = sizeof(nums) /
sizeof(nums[0]);
bubble_sort(nums, nums_count,
sizeof(nums[0]),
sort_ascending);
...
}
```

```
// bubblesort.c
```

```
void bubble_sort(void *arr, size_t n,
size_t elem_size_bytes,
int (*should_swap)(void *, void *)) {
...
}
```

Key question: *in*
sort_ascending, what type
are **ptr1** and **ptr2** really?
*int **

Function Pointers

```
// my_program.c
#include "bubblesort.h"

// 0 if equal, (-) if first before
// second, (+) if first after second
int sort_ascending(void *ptr1, void
*ptr2) {
    int *realPtr1 = ptr1;
    int *realPtr2 = ptr2;
    ...
}

int main(int argc, char *argv[]) {
    int nums[] = {4, 2, -5, 1, 12, 56};
    int nums_count = sizeof(nums) /
        sizeof(nums[0]);
    bubble_sort(nums, nums_count,
        sizeof(nums[0]),
        sort_ascending);
    ...
}
```

```
// bubblesort.c

void bubble_sort(void *arr, size_t n,
                size_t elem_size_bytes,
                int (*should_swap)(void *, void *)) {
    ...
}
```

Key question: *in*
sort_ascending, what type
are **ptr1** and **ptr2** really?
int *

Function Pointers

```
// my_program.c
#include "bubblesort.h"

// 0 if equal, (-) if first before
// second, (+) if first after second
int sort_ascending(void *ptr1, void
*ptr2) {
    int *realPtr1 = ptr1;
    int *realPtr2 = ptr2;
    int actualElem1 = *realPtr1;
    int actualElem2 = *realPtr2;

}

int main(int argc, char *argv[]) {
    int nums[] = {4, 2, -5, 1, 12, 56};
    int nums_count = sizeof(nums) /
        sizeof(nums[0]);
    bubble_sort(nums, nums_count,
        sizeof(nums[0]),
```

```
// bubblesort.c

void bubble_sort(void *arr, size_t n,
                size_t elem_size_bytes,
                int (*should_swap)(void *, void *)) {
    ...
}
```

Key question: *in*
sort_ascending, what type
are **ptr1** and **ptr2** really?
*int **

Function Pointers

```
// my_program.c
#include "bubblesort.h"

// 0 if equal, (-) if first before
// second, (+) if first after second
int sort_ascending(void *ptr1, void
*ptr2) {
    int *realPtr1 = ptr1;
    int *realPtr2 = ptr2;
    int actualElem1 = *realPtr1;
    int actualElem2 = *realPtr2;
    return actualElem1 - actualElem2;
}

int main(int argc, char *argv[]) {
    int nums[] = {4, 2, -5, 1, 12, 56};
    int nums_count = sizeof(nums) /
        sizeof(nums[0]);
    bubble_sort(nums, nums_count,
        sizeof(nums[0]),
```

```
// bubblesort.c

void bubble_sort(void *arr, size_t n,
                size_t elem_size_bytes,
                int (*should_swap)(void *, void *)) {
    ...
}
```

Key question: *in*
sort_ascending, what type
are **ptr1** and **ptr2** really?
*int **

Function Pointers

```
// my_program.c
#include "bubblesort.h"

// 0 if equal, (-) if first before
// second, (+) if first after second
int sort_ascending(void *ptr1, void
*ptr2) {
    int actualElem1 = *(int *)ptr1;
    int actualElem2 = *(int *)ptr2;
    return actualElem1 - actualElem2;
}

int main(int argc, char *argv[]) {
    int nums[] = {4, 2, -5, 1, 12, 56};
    int nums_count = sizeof(nums) /
        sizeof(nums[0]);
    bubble_sort(nums, nums_count,
        sizeof(nums[0]),
        sort_ascending);
    ...
}
```

```
// bubblesort.c

void bubble_sort(void *arr, size_t n,
                size_t elem_size_bytes,
                int (*should_swap)(void *, void *)) {
    ...
}
```

Key question: *in*
sort_ascending, what type
are **ptr1** and **ptr2** really?
*int **

Function Pointers

```
// my_program.c
#include "bubblesort.h"

// 0 if equal, (-) if first before
// second, (+) if first after second
int sort_ascending(void *ptr1, void
*ptr2) {
    return *(int *)ptr1 - *(int *)ptr2;
}

int main(int argc, char *argv[]) {
    int nums[] = {4, 2, -5, 1, 12, 56};
    int nums_count = sizeof(nums) /
        sizeof(nums[0]);
    bubble_sort(nums, nums_count,
        sizeof(nums[0]),
        sort_ascending);
    ...
}
```

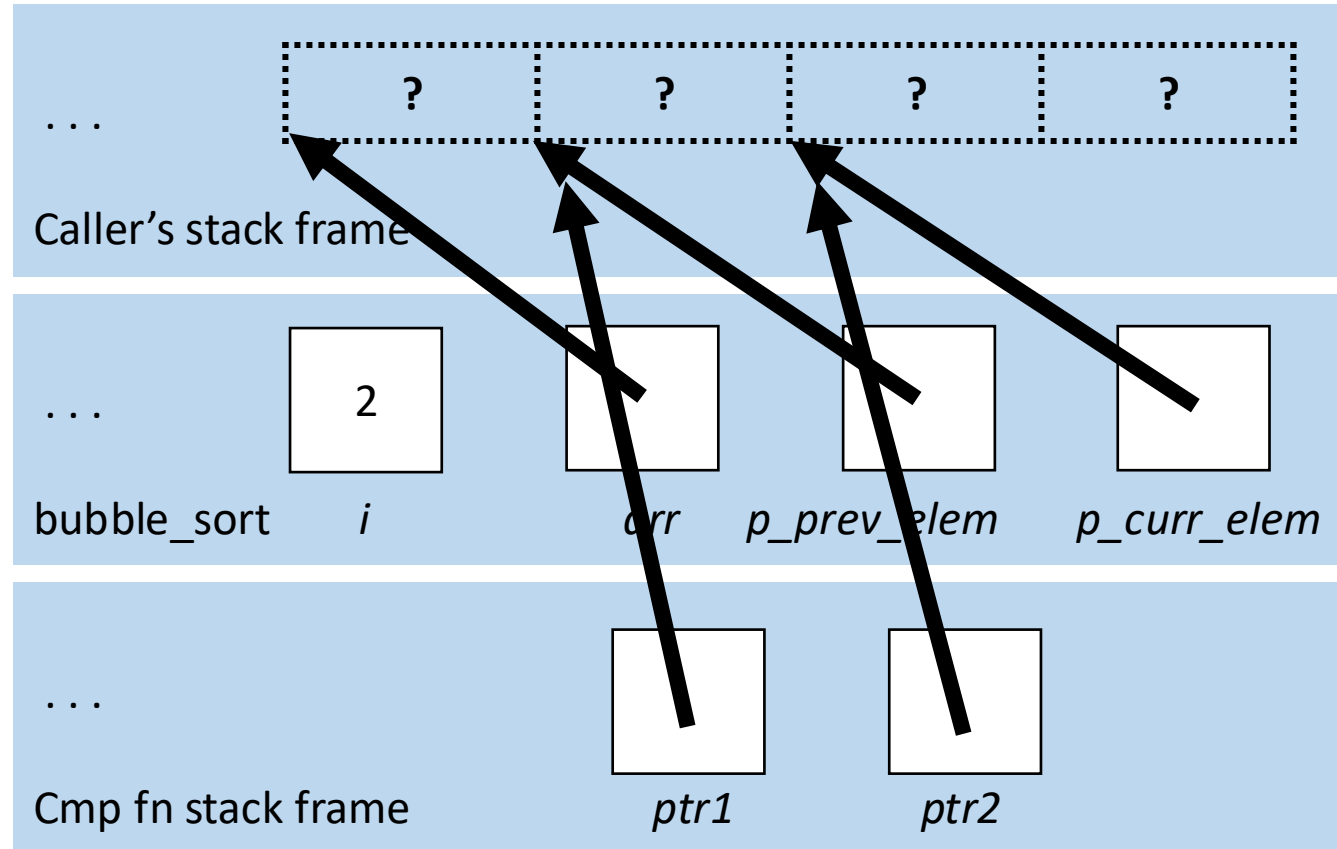
```
// bubblesort.c

void bubble_sort(void *arr, size_t n,
                size_t elem_size_bytes,
                int (*should_swap)(void *, void *)) {
    ...
}
```

This function is created by the caller *specifically* to compare integers, knowing their addresses are necessarily disguised as void *so that **bubble_sort** can work for any array type.

Function Pointers

```
int sort_ascending(void *ptr1, void *ptr2) {  
    return *(int *)ptr1 - *(int *)ptr2;  
}
```



Demo: Bubble Sort



```
sort_example.c
```

Pollev: what should go in the blanks?

```
int sort_strings_alphabetical(void *ptr1, void *ptr2) {  
    char *str1 = ___???___ptr1;  
    char *str2 = ___???___ptr2;  
  
    return strcmp(str1, str2);  
}
```

Respond on Pollev:
pollev.com/cs107



L14. 1. What goes in the blank?

`(char *)`

0%

`*(char **)`

0%

`*(char *)`

0%

`** (char **)`

0%

Function Pointers

How does the caller implement a comparison function that bubble sort can use?
The key idea is now the comparison function is passed pointers to the elements that are being compared.

We can use the following pattern:

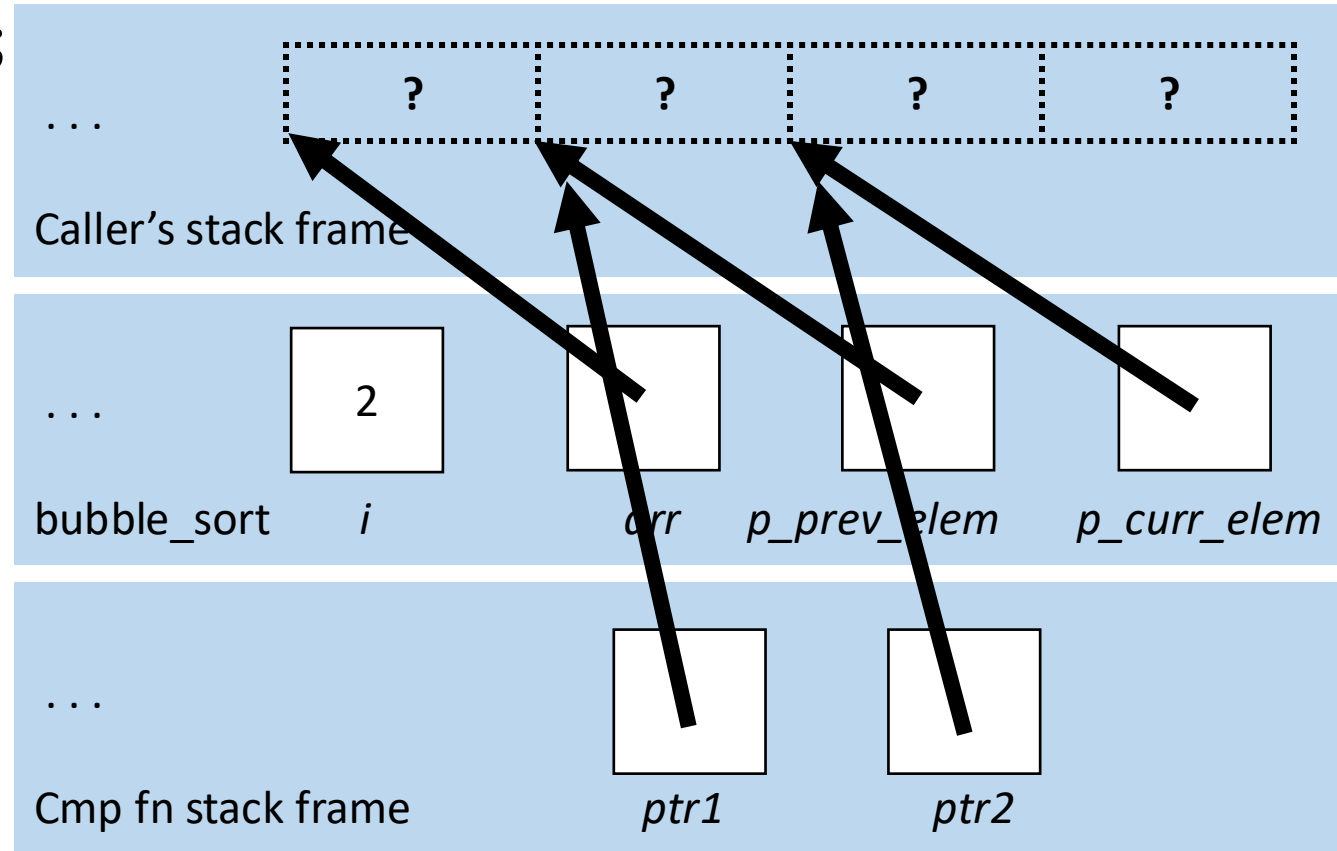
- 1) Cast the void *argument(s) to what they really are for that comparison function
- 2) Dereference the typed pointer(s) to access the values.
- 3) Perform the necessary operation.

(steps 1 and 2 can often be combined into a single step)

String Comparison Function

```
int string_compare(void *ptr1, void *ptr2) {  
    // cast arguments and dereference  
    char *str1 = *(char **)ptr1;  
    char *str2 = *(char **)ptr2;  
  
    // perform operation  
    return strcmp(str1, str2);  
}
```

Hint: remember what the true types of the parameters are. **Draw pictures!**



Lecture Plan

- **Recap and continuing:** Function Pointers
- **Example: Count Matches**

```
cp -r /afs/ir/class/cs107/lecture-code/lect14 .
```

Function Pointers

- We will commonly see function pointers used for comparison functions for various library functions. But if we implement a function, we can specify any function we want for the caller to pass in.
- Function pointers can be used in a variety of ways. For instance, you could have:
 - A function to compare two elements of a given type
 - A function to print out an element of a given type
 - A function to free memory associated with a given type
 - And more...

Practice: Count Matches

Let's write a generic function *count_matches* that can return a count of the number of a certain type of element in a generic array. Whether an element should be counted or not is up to the caller.

```
int count_matches(??);
```

(bubble sort signature for reference):

```
void bubble_sort(void *arr, size_t n,  
                size_t elem_size_bytes, int (*cmp_fn)(void *, void *))
```

Discuss with your table and then individually input your response – what could the function signature for this function be?

pollev.com/cs107



L14. 2. What could the function signature look like for `count_matches`?

Nobody has responded yet.

Hang tight! Responses are coming in.

Practice: Count Matches

Let's write a generic function *count_matches* that can return a count of the number of a certain type of element in a generic array. Whether an element should be counted or not is up to the caller.

```
int count_matches(void *base, size_t nelems,  
                 size_t elem_size_bytes,  
                 bool (*match_fn)(void *));
```

Parameters: the generic array, and a function parameter that can take in a pointer to a single array element and tell us if it's a match (returns true if match, false otherwise).

Demo: Count Matches



```
count_matches.c
```

Function Pointer Pitfalls

- If a function takes a function pointer as a parameter, it will accept it if it fits the specified signature.
- *This is dangerous!* E.g. what happens if you pass in a char comparison function when sorting an integer array? (see lab4 for more!)

Function pointers as variables

- Function pointers can be set to NULL.
- Function pointers can be more than just parameters – we can also make variables! For example:

```
1 int main(int argc, char *argv[]) {
2     int (*cmp)(void *, void *) = sort_ascending;
3     if (...) cmp = sort_descending;
4     else if (...) cmp = sort_odd_then_even;
5     ...
6
7     bubble_sort(nums, count, sizeof(nums[0]), cmp);
8     ...
9 }
10
```

Generic C Standard Library Functions

- **qsort** – generic library array sorting function (just like bubble sort):

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

- **scandir** – Creates a directory listing with any order and contents. Requires a function that specifies whether to include a given directory entry in the listing, and another function to specify the order of two given directory entries.

```
int scandir(const char *dirp, struct dirent ***namelist,  
           int (*filter)(const struct dirent *),  
           int (*compar)(const struct dirent **, const struct dirent **));
```

Generic C Standard Library Functions

- **bsearch** – generic library array binary search function
- **find** – generic library array linear search function
- **lsearch** – generic library array linear search function, plus inserts if not found.

Generics Recap

- We can pass functions as parameters to pass logic around in our programs.
- Comparison functions are one common class of functions passed as parameters to generically compare the elements at two addresses.
- Functions handling generic data must use *pointers to the data they care about*, since any parameters must have *one type* and *one size*.

Recap

- **Recap and continuing:** Function Pointers
- **Example:** Count Matches

Lecture 14 takeaway: A common use case for function pointers is to pass comparison functions to generic functions like bubble sort that need to compare elements; but there are many use cases, such as with counting matches.

Next time: introduction to assembly