CS107, Lecture 14
Function Pointers, Continued

Reading: K&R 5.11

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: Function Pointers

Function pointers allow us to pass functions as parameters and store functions in variables. We can use them to "pass logic around our program".

Example: bubble sort – a function anyone can import to sort an array.

- Doesn't know how to order a given pair of elements – asks caller to pass in a function that can take in two elements and specify intended order.
- Bubble sort can call this *comparison function* whenever it needs to know the ordering of two elements.
- When a program wants to use this function, they use it with a specific kind of data and would write a comparison function specifically for that kind of data and the ordering they want that time.

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

Function Pointers

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

int sort_descending(...) {
    ...
}

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_descending);
    ...
}
```

```c
// bubblesort.c

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

bubble_sort is written generically. When someone imports it into their program, they will call it specifying the sort ordering function they want to use for that specific time.

Recap: Function Pointers

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

We must have one signature for the comparison function that supports scenarios with any type of data; the only way to do this is for the comparison function signature to take in *pointers* to the two elements being compared. We use **void *** to indicate “any pointer”.

Generic Bubble Sort

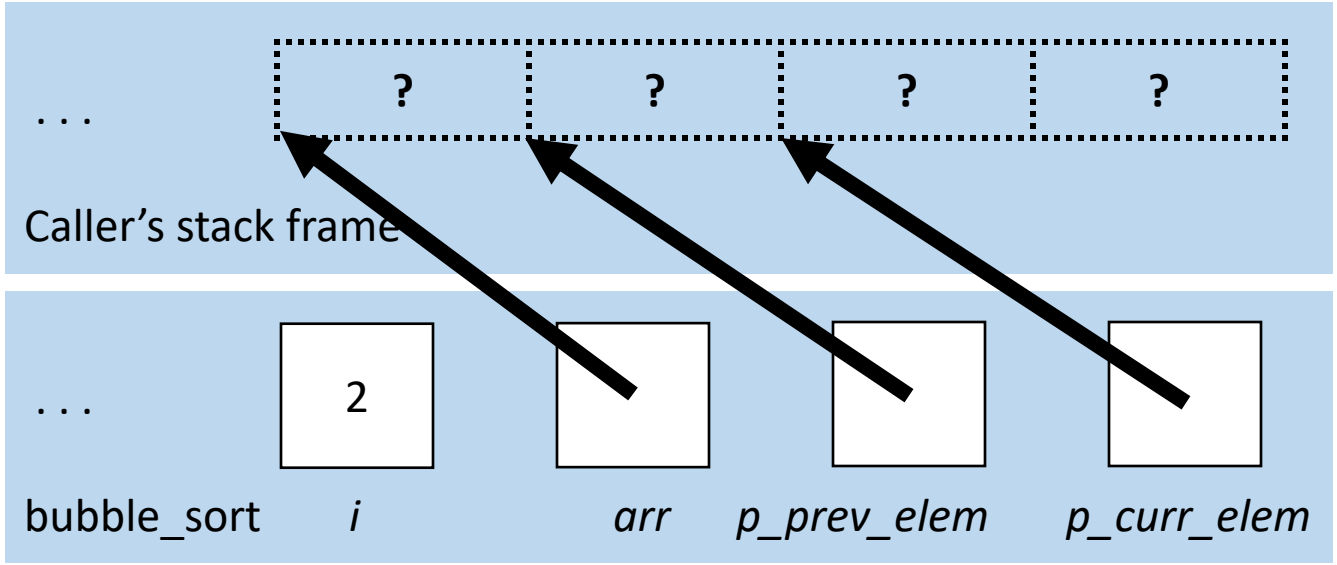
```
void bubble_sort(void *arr, size_t n, size_t elem_size_bytes,
                 bool (*should_swap)(void *, void *)) {
    while (true) {
        bool swapped = false;
        for (size_t i = 1; i < n; i++) {
            void *p_prev_elem = (char *)arr + (i - 1) * elem_size_bytes;
            void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

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;
        }
    }
}
```



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;
        }
    }
}
```

From Before: Non-Generic Comparison Function

```c
// 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);
    ...
}
```

```c
// bubblesort.c

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

Function Pointers

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

int sort_descending(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_descending);
    ...
}
```

```
// 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

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

The caller implements a specific comparison function for use in a specific scenario with a specific type of array. So even though the comparison function takes in **void *** pointers, the caller knows what type of pointer is *really* passed in within the context of this comparison function.

Function Pointers

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

// 0 if equal, neg if first before
// second, pos if second before first
int sort_descending(void *ptr1, void
*ptr2) {
    int actualElem1 = ???
    int 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_descending);
    ...
```

```
// 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

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

// 0 if equal, neg if first before
// second, pos if second before first
int sort_descending(void *ptr1, void
*ptr2) {
    int actualElem1 = *(int *)ptr1;
    int actualElem2 = *(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_descending);
    ...
```

```c
// 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

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

// 0 if equal, neg if first before
// second, pos if second before first
int sort_descending(void *ptr1, void
*ptr2) {
    int actualElem1 = *(int *)ptr1;
    int actualElem2 = *(int *)ptr2;
    return actualElem2 – actualElem1;
}

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_descending);
    ...
```

```
// 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

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

// 0 if equal, neg if first before
// second, pos if second before first
int sort_descending(void *ptr1, void
*ptr2) {
    return *(int *)ptr2 - *(int *)ptr1;
}

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_descending);
    ...
}
```

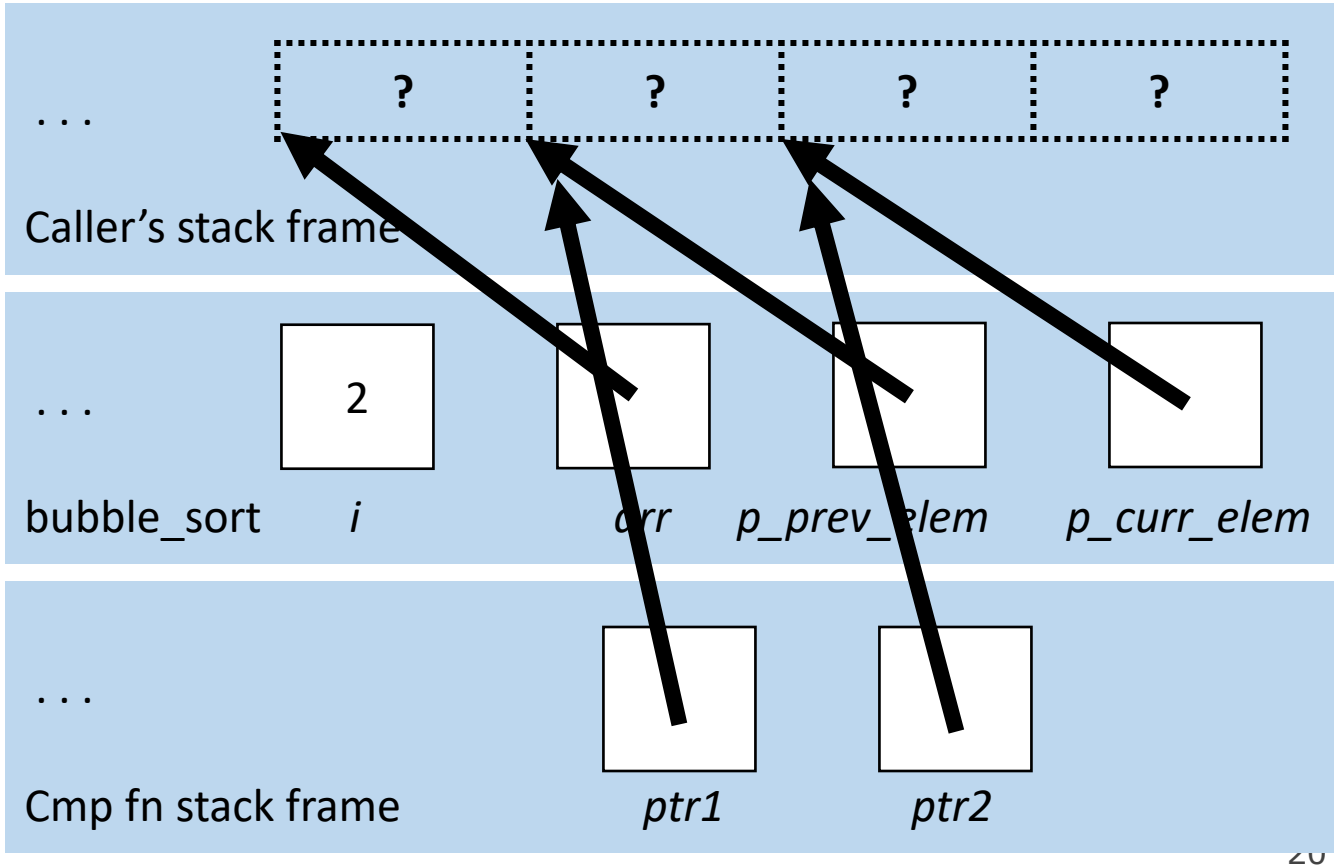
```
// bubblesort.c

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



Function Pointers

```
// not compatible – signature must match exactly
int sort_descending(int *ptr1, int *ptr2) {
    return *ptr2 - *ptr1;
}
```

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
or text CS107 to 22333 once to join.

What goes in the blank?

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:

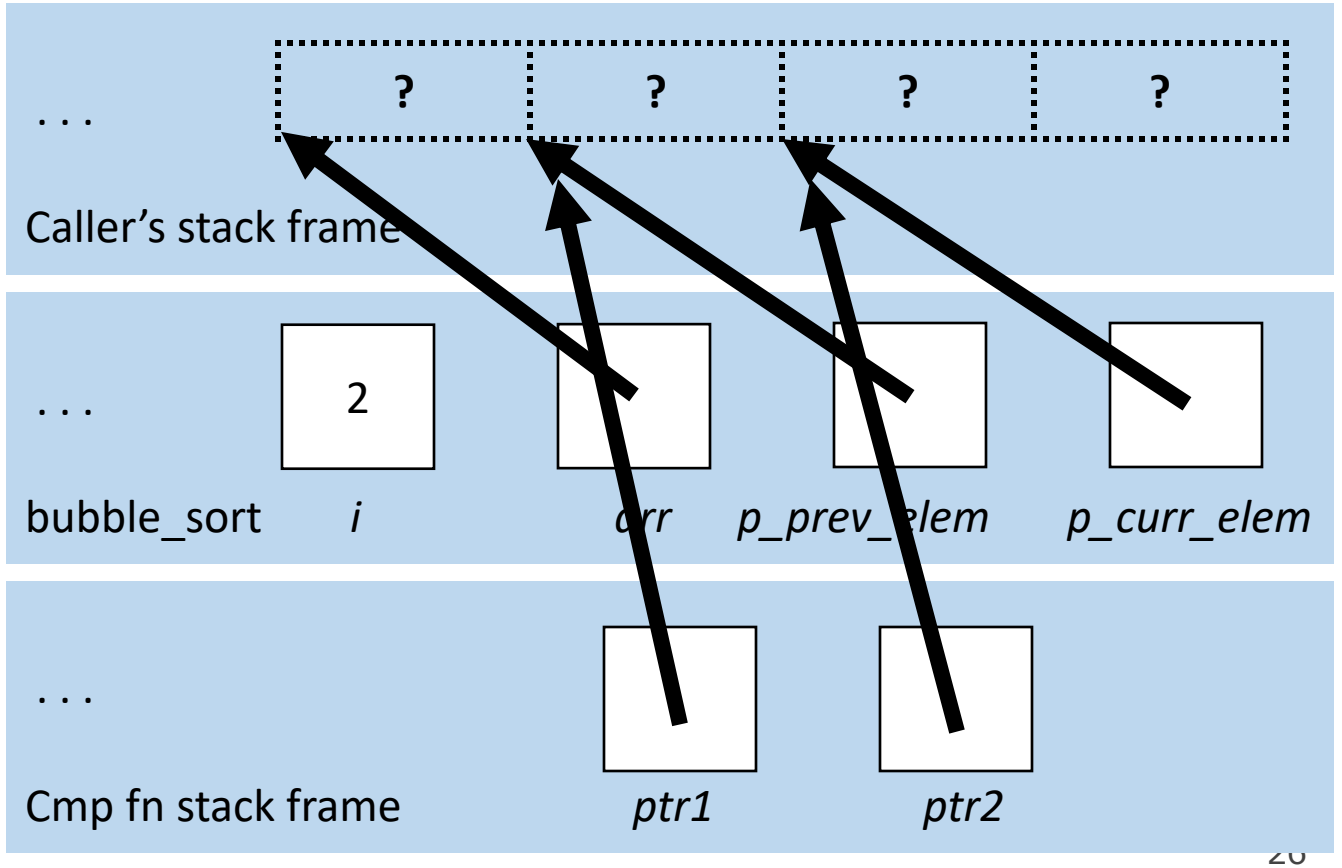
1) Cast the void *argument(s) to what they really are for that comparison function
2) Dereference the typed pointer(s) to access the values.
3) Perform the necessary operation.

(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 count the number of a certain type of element in a generic array.
- It should take in as parameters information about 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).

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

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** – I can sort an array of any type! To do that, I need you to provide me a function that can compare two elements of the kind you are asking me to sort.

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

- **scandir** – I can create a directory listing with any order and contents! To do that, I need you to provide me a function that tells me whether you want me to include a given directory entry in the listing. I also need you to provide me a function that tells me the correct ordering 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** – I can use binary search to search for a key in an array of any type! To do that, I need you to provide me a function that can compare two elements of the kind you are asking me to search.
- **lfind** – I can use linear search to search for a key in an array of any type! To do that, I need you to provide me a function that can compare two elements of the kind you are asking me to search.
- **lsearch** - I can use linear search to search for a key in an array of any type! I will also add the key for you if I can't find it. In order to do that, I need you to provide me a function that can compare two elements of the kind you are asking me to search.

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.