CS107, Lecture 14
C Generics and Function Pointers, Take II

Reading: K&R 5.11

Ed Discussion: https://edstem.org/us/courses/46162/discussion/3714841

This document is copyright (C) Stanford Computer Science, licensed under Creative Commons Attribution 2.5 License. All rights reserved.
Based on slides created by Cynthia Lee, Chris Gregg, Jerry Cain, Lisa Yan, Nick Troccoli, and others.

Integer Bubble Sort

```
void bubble_sort_int(int *arr, size_t n, int (*cmp_fn)(int, int)) {
    while (true) {
        bool swapped = false;
        for (size_t i = 1; i < n; i++) {
            if (cmp_fn(arr[i – 1], arr[i]) > 0) {
                swap(&arr[i - 1], &arr[i], sizeof(int));
                swapped = true;
            }
        }

        if (!swapped) {
            return;
        }
    }
}
```

bubble_sort_int now supports any possible sort ordering. But it's not fully generic - it still only supports arrays of ints. What about arrays of other types?

Generic Bubble Sort

file_that_sorts_ints.c

```
#include <bubblesort.h>

int main(int argc, char *argv[]) {
    ...
}
```

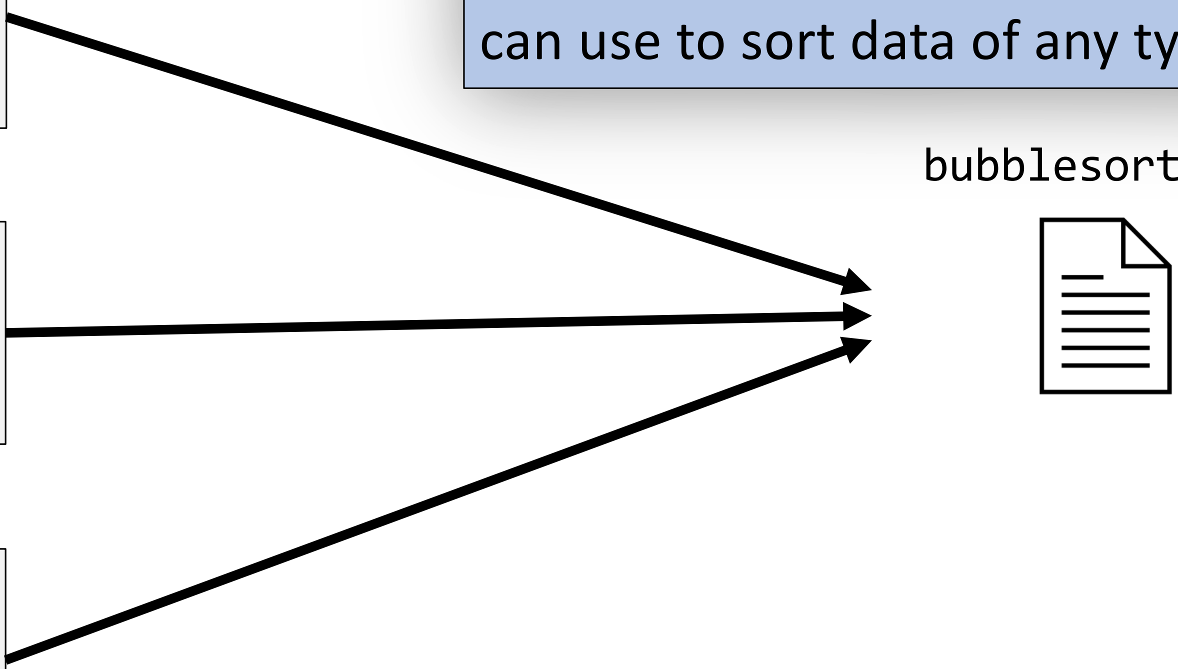
file_that_sorts_strings.c

```
#include <bubblesort.h>

int main(int argc, char *argv[]) {
    ...
}
```

file_that_sorts_structs.c

```
#include <bubblesort.h>

int main(int argc, char *argv[]) {
    ...
}
```

Goal: write 1 implementation of bubblesort that any program can use to sort data of any type.



Generic Bubble Sort

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

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

Generic Bubble Sort

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

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

Problem: we need **one comparison function signature** that works with any type.

Generic Bubble Sort

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

```
void bubble_sort_int(void *arr, size_t n,
    size_t elem_size_bytes, int (*cmp_fn)(int, int));
void bubble_sort_long(void *arr, size_t n,
    size_t elem_size_bytes, int (*cmp_fn)(long, long));
void bubble_sort_str(void *arr, size_t n,
    size_t elem_size_bytes, int (*cmp_fn)(char *, char *));
...
```

How can we write a function that can take in parameters of *any* type?

Generic Parameters

- Let’s say I want to write a function **generic_func** that takes in one parameter, but it could be any type. What should we specify as the parameter type?

```
generic_func(type param1) { …
```

- **Problem**: C needs the parameter to be a single specified size. But in theory it could be infinitely big (e.g., a large struct).
- **Key Idea:** require the caller to pass in a *pointer to the data*. Pointers are always 8 bytes, regardless of what they address.
- **Problem:** which pointer type should I pick? e.g., int *, char *? If it doesn’t match the actual type, the caller will have to cast (yuck).
- **Key Idea #2:** make the parameter type a **void ***, which means "any pointer".

Generic Bubble Sort

- We will use the same idea for bubble sort’s comparison function. Make its parameters **void *s**. Then we must call them by specifying *pointers to what we want to compare*, not the elements themselves.

Let’s write a generic version of bubblesort:

1. Make the parameters and swap functionality generic
2. Make the comparison function usage generic

Generic Bubble Sort

```
void bubble_sort(int *arr, size_t n, int (*cmp_fn)(int, int)) {
    while (true) {
        bool swapped = false;
        for (size_t i = 1; i < n; i++) {
            if (cmp_fn(arr[i - 1], arr[i]) > 0) {
                swap(&arr[i - 1], &arr[i], sizeof(int));
                swapped = true;
            }
        }

        if (!swapped) {
            return;
        }
    }
}
```

Let's start by making the parameters generic.

Generic Bubble Sort

```
void bubble_sort(void *arr, size_t n,
                 size_t elem_size_bytes, int (*cmp_fn)(int, int)) {
    while (true) {
        bool swapped = false;
        for (size_t i = 1; i < n; i++) {
            if (cmp_fn(arr[i - 1], arr[i]) > 0) {
                swap(&arr[i – 1], &arr[i], elem_size_bytes);
                swapped = true;
            } // arguments passed to swap won’t compile! must fix!
        }

        if (!swapped) {
            return;
        }
    }
}
```

Let’s start by making the parameters generic.

Generic Bubble Sort

```
void bubble_sort(void *arr, size_t n,
                 size_t elem_size_bytes, int (*cmp_fn)(int, int)) {
    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 (cmp_fn(arr[i - 1], arr[i]) > 0) {
                swap(p_prev_elem, p_curr_elem, elem_size_bytes);
                swapped = true;
            }
        }
        if (!swapped) {
            return;
        }
    }
}
```

Let's start by making the parameters generic.

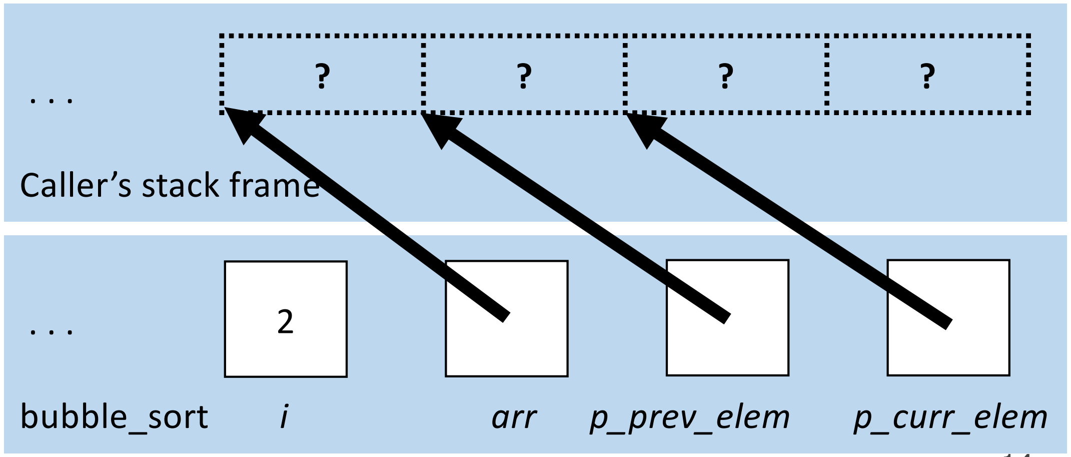
Generic Bubble Sort

```
void bubble_sort(void *arr, size_t n,
                 size_t elem_size_bytes, int (*cmp_fn)(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 (cmp_fn(p_prev_elem, p_curr_elem) > 0) {
                swap(p_prev_elem, p_curr_elem, elem_size_bytes);
                swapped = true;
            }
        }
        if (!swapped) {
            return;
        }
    }
}
```

Let's start by making the parameters generic.

Generic Bubble Sort

```
void bubble_sort(void *arr, size_t n, size_t elem_size_bytes,
                 int (*cmp_fn)(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 (cmp_fn(p_prev_elem, p_curr_elem) > 0) {
                swap(p_prev_elem, p_curr_elem, elem_size_bytes);
                swapped = true;
            }
        }

        if (!swapped) {
            return;
        }
    }
}
```



Calling Generic Bubble Sort

```
// 0 if equal, neg if first before second, pos if second before first
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);
    ...
}
```

Key idea: now the comparison function is passed pointers to the elements being compared.

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) and set typed pointers equal to them.
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)

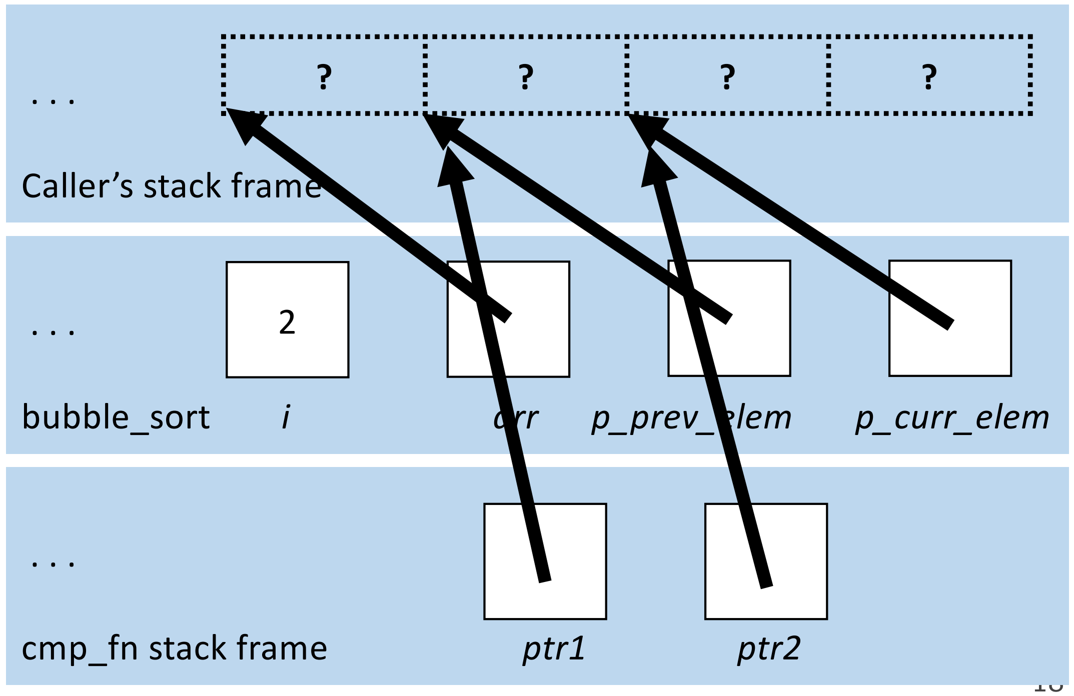
Function Pointers

```
int sort_descending(void *ptr1, void *ptr2) {
    // 1) cast arguments to int *s
     int *num1ptr = (int *)ptr1;
     int *num2ptr = (int *)ptr2;

     // 2) dereference typed points to access values
     int num1 = *num1ptr;
     int num2 = *num2ptr;

     // 3) perform operation
     return num2 – num1;
}
```

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.

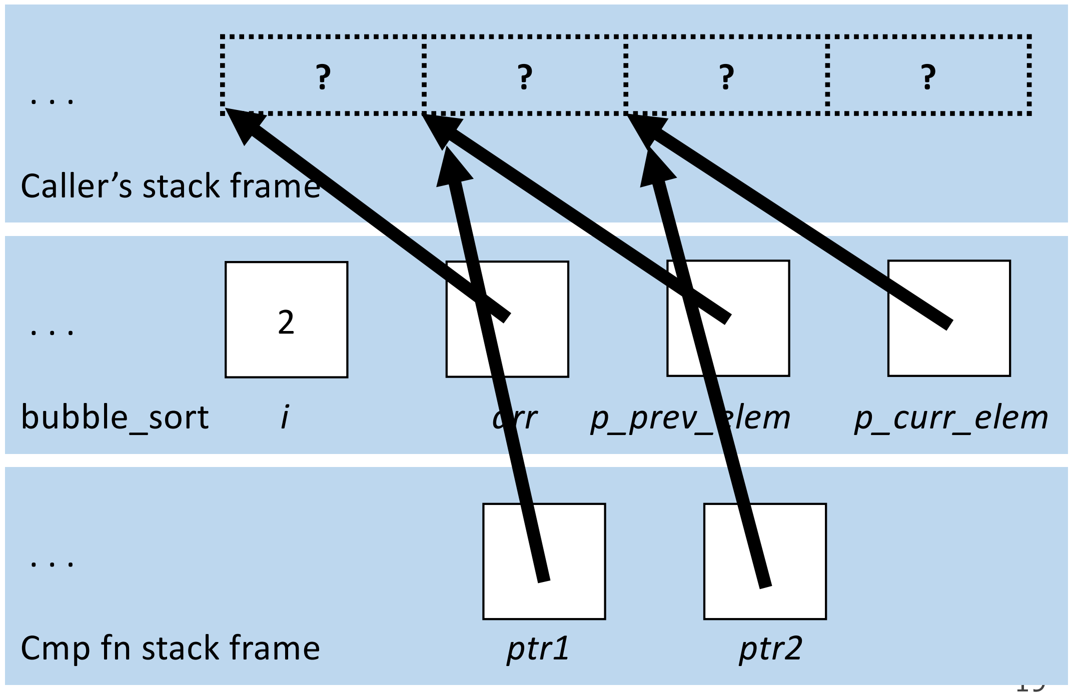
Function Pointers

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



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);
}
```



Function Pointer Pitfalls

- If a function takes a function pointer as a parameter, any function with the expected prototype can be passed in, even if it's the wrong function.
- Think about what happens if you pass in a string comparison function when sorting an integer array?

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.

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

Demo: Count Matches

`count_matches.c`

Practice Solution: Count Matches

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

    int match_count = 0;

    for (size_t i = 0; i < nelems; i++) {
        void *curr_p = (char *)base + i * elem_size_bytes;
        if (match_fn(curr_p)) {
            match_count++;
        }
    }

    return match_count;
}
```

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.
- **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.