

CS107, Lecture 13

C Generics and Function Pointers

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)

Announcements

- Midterm exam **Tues 5/6 7-9PM in CEMEX Auditorium** - exam webpages posted with review materials and exam information. Review session being scheduled, time TBD!
- **Mid-Quarter Evaluation available:** open through Sun 5/4 at 11:59PM PDT. We greatly appreciate any feedback!

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 (last time)
- Allows us to learn how to pass functions as parameters, a core concept in many languages (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 write C code that works with any data type
- Learn how to pass functions as parameters
- Learn how to write functions that accept functions as parameters

Lecture Plan

- Generics So Far
- Generic Array Swap
- **Motivating Example:** Bubble Sort
- Function Pointers
- Generic Function Pointers

```
cp -r /afs/ir/class/cs107/lecture-code/lect13 .
```

Lecture Plan

- **Generics So Far**
- Generic Array Swap
- **Motivating Example:** Bubble Sort
- Function Pointers
- Generic Function Pointers

```
cp -r /afs/ir/class/cs107/lecture-code/lect13 .
```

Generics So Far

- **void *** is a variable type that represents a generic pointer “to something”.
- We cannot directly dereference a **void ***.
- We can use **memcpy** or **memmove** to copy data from one memory location to another.
- **void *** and generics are powerful but dangerous because of the lack of type checking, so we must be extra careful when working with generic memory.

Generic Swap

```
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {  
    char temp[nbytes];  
    memcpy(temp, data1ptr, nbytes);  
    memcpy(data1ptr, data2ptr, nbytes);  
    memcpy(data2ptr, temp, nbytes);  
}
```

We can use **void *** to represent a pointer to any data, and **memcpy/memmove** to copy arbitrary bytes.

Void * Pitfalls

void *s are powerful, but dangerous - C cannot do as much checking!

- E.g. with **int**, C would never let you swap *half* of an int. With **void** *s, this can happen!

```
int x = 0xffffffff;
int y = 0xeeeeeeeee;
swap(&x, &y, sizeof(short));
```

```
// now x = 0xffffeeee, y = 0xeeeeffff!
printf("x = 0x%x, y = 0x%x\n", x, y);
```

Void *Pitfalls

Void * has more room for error because it manipulates arbitrary bytes without knowing what they represent. This can result in some strange memory Frankensteins!



NEW: memset

memset is a function that sets a specified number of bytes starting at an address to a certain value.

```
void *memset(void *s, int c, size_t n);
```

It fills *n* bytes starting at memory location *s* with the byte *c*. (It also returns *s*).

```
int counts[5];  
memset(counts, 0, 3);           // zero out first 3 bytes at counts  
memset(counts + 3, 0xff, 4)    // set index 3 entry's bytes to 1s
```

Lecture Plan

- Generics So Far
- **Generic Array Swap**
- **Motivating Example:** Bubble Sort
- Function Pointers
- Generic Function Pointers

```
cp -r /afs/ir/class/cs107/lecture-code/lect13 .
```

Swap Ends

You're asked to write a function that swaps the first and last elements in an array of numbers.

```
void swap_ends_int(int *arr, size_t nelems) {  
    int tmp = arr[0];  
    arr[0] = arr[nelems - 1];  
    arr[nelems - 1] = tmp;  
}
```

Wait – we just wrote a generic swap function. Let's use that!

```
int main(int argc, char *argv[]) {  
    int nums[] = {5, 2, 3, 4, 1};  
    size_t nelems = sizeof(nums) / sizeof(nums[0]);  
    swap_ends_int(nums, nelems);  
    // want nums[0] = 1, nums[4] = 5  
    printf("nums[0] = %d, nums[4] = %d\n", nums[0], nums[4]);  
    return 0;  
}
```

Swap Ends

You're asked to write a function that swaps the first and last elements in an array of numbers.

```
void swap_ends_int(int *arr, size_t nelems) {  
    swap(arr, arr + nelems - 1, sizeof(*arr));  
}
```

Wait – we just wrote a generic swap function. Let's use that!

```
int main(int argc, char *argv[]) {  
    int nums[] = {5, 2, 3, 4, 1};  
    size_t nelems = sizeof(nums) / sizeof(nums[0]);  
    swap_ends_int(nums, nelems);  
    // want nums[0] = 1, nums[4] = 5  
    printf("nums[0] = %d, nums[4] = %d\n", nums[0], nums[4]);  
    return 0;  
}
```

Swap Ends

Let's write out what some other versions would look like (just in case).

```
void swap_ends_int(int *arr, size_t nelems) {  
    swap(arr, arr + nelems - 1, sizeof(*arr));  
}
```

```
void swap_ends_short(short *arr, size_t nelems) {  
    swap(arr, arr + nelems - 1, sizeof(*arr));  
}
```

```
void swap_ends_string(char **arr, size_t nelems) {  
    swap(arr, arr + nelems - 1, sizeof(*arr));  
}
```

```
void swap_ends_float(float *arr, size_t nelems) {  
    swap(arr, arr + nelems - 1, sizeof(*arr));  
}
```

The code seems to be the same regardless of the type!

Swap Ends

Let's write a version of `swap_ends` that works for any type of array.

```
void swap_ends(void *arr, size_t nelems) {  
    swap(arr, arr + nelems - 1, sizeof(*arr));  
}
```

Is this generic? Does this work?

Swap Ends

Let's write a version of `swap_ends` that works for any type of array.

```
void swap_ends(void *arr, size_t nelems) {  
    swap(arr, arr + nelems - 1, sizeof(*arr));  
}
```

Is this generic? Does this work?

Unfortunately not. First, we no longer know the element size. Second, pointer arithmetic depends on the type of data being pointed to. With a `void *`, we lose that information!

Swap Ends

Let's write a version of `swap_ends` that works for any type of array.

```
void swap_ends(void *arr, size_t nelems) {  
    swap(arr, arr + nelems - 1, sizeof(*arr));  
}
```

We need to know the element size, so let's add a parameter.

Swap Ends

Let's write a version of `swap_ends` that works for any type of array.

```
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {  
    swap(arr, arr + nelems - 1, elem_bytes);  
}
```

We need to know the element size, so let's add a parameter.

Pointer Arithmetic

`arr + nelems - 1`

Let's say `nelems = 4`. How many bytes beyond `arr` is this?

If it's an array of...

Int?

Pointer Arithmetic

`arr + nelems - 1`

Let's say `nelems = 4`. How many bytes beyond `arr` is this?

If it's an array of...

Int: adds 3 places to `arr`, and $3 * \text{sizeof}(\text{int}) = 12$ bytes

Pointer Arithmetic

```
arr + nelems - 1
```

Let's say `nelems = 4`. How many bytes beyond `arr` is this?

If it's an array of...

Int: adds 3 places to `arr`, and $3 * \text{sizeof}(\text{int}) = 12$ bytes

Short?

Pointer Arithmetic

```
arr + nelems - 1
```

Let's say `nelems = 4`. How many bytes beyond `arr` is this?

If it's an array of...

Int: adds 3 places to `arr`, and $3 * \text{sizeof}(\text{int}) = 12$ bytes

Short: adds 3 places to `arr`, and $3 * \text{sizeof}(\text{short}) = 6$ bytes

Pointer Arithmetic

```
arr + nelems - 1
```

Let's say `nelems = 4`. How many bytes beyond `arr` is this?

If it's an array of...

Int: adds 3 places to `arr`, and $3 * \text{sizeof}(\text{int}) = 12$ bytes

Short: adds 3 places to `arr`, and $3 * \text{sizeof}(\text{short}) = 6$ bytes

Char *: adds 3 places to `arr`, and $3 * \text{sizeof}(\text{char} *) = 24$ bytes

In each case, we need to know the element size to do the arithmetic.

Swap Ends

Let's write a version of `swap_ends` that works for any type of array.

```
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {  
    swap(arr, arr + nelems - 1, elem_bytes);  
}
```

How many bytes past `arr` should we go to get to the last element?

`(nelems - 1) * elem_bytes`

Swap Ends

Let's write a version of swap_ends that works for any type of array.

```
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {  
    swap(arr, arr + (nelems - 1) * elem_bytes, elem_bytes);  
}
```

How many bytes past arr should we go to get to the last element?

$(nelems - 1) * elem_bytes$

Swap Ends

Let's write a version of `swap_ends` that works for any type of array.

```
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {  
    swap(arr, arr + (nelems - 1) * elem_bytes, elem_bytes);  
}
```

But C still can't do arithmetic with a `void*`. We need to tell it to not worry about it, and just add bytes. **How can we do this?**

Swap Ends

Let's write a version of swap_ends that works for any type of array.

```
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {  
    swap(arr, (char *)arr + (nelems - 1) * elem_bytes, elem_bytes);  
}
```

But C still can't do arithmetic with a void*. We need to tell it to not worry about it, and just add bytes. **How can we do this?**

char * pointers already add bytes!

Swap Ends

You're asked to write a function that swaps the first and last elements in an array of numbers. Well, now it can swap for an array of anything!

```
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {  
    swap(arr, (char *)arr + (nelems - 1) * elem_bytes, elem_bytes);  
}
```

```
int nums[] = {5, 2, 3, 4, 1};  
size_t nelems = sizeof(nums) / sizeof(nums[0]);  
swap_ends(nums, nelems, sizeof(nums[0]));
```

Swap Ends

You're asked to write a function that swaps the first and last elements in an array of numbers. Well, now it can swap for an array of anything!

```
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {  
    swap(arr, (char *)arr + (nelems - 1) * elem_bytes, elem_bytes);  
}
```

```
short nums[] = {5, 2, 3, 4, 1};  
size_t nelems = sizeof(nums) / sizeof(nums[0]);  
swap_ends(nums, nelems, sizeof(nums[0]));
```

Swap Ends

You're asked to write a function that swaps the first and last elements in an array of numbers. Well, now it can swap for an array of anything!

```
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {  
    swap(arr, (char *)arr + (nelems - 1) * elem_bytes, elem_bytes);  
}
```

```
char *strs[] = {"Hi", "Hello", "Howdy"};  
size_t nelems = sizeof(strs) / sizeof(strs[0]);  
swap_ends(strs, nelems, sizeof(strs[0]));
```

Swap Ends

You're asked to write a function that swaps the first and last elements in an array of numbers. Well, now it can swap for an array of anything!

```
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {  
    swap(arr, (char *)arr + (nelems - 1) * elem_bytes, elem_bytes);  
}
```

```
mystruct structs[] = ...;  
size_t nelems = ...;  
swap_ends(structs, nelems, sizeof(structs[0]));
```

Lecture Plan

- Generics So Far
- Generic Array Swap
- **Motivating Example: Bubble Sort**
- Function Pointers
- Generic Function Pointers

```
cp -r /afs/ir/class/cs107/lecture-code/lect13 .
```

Bubble Sort

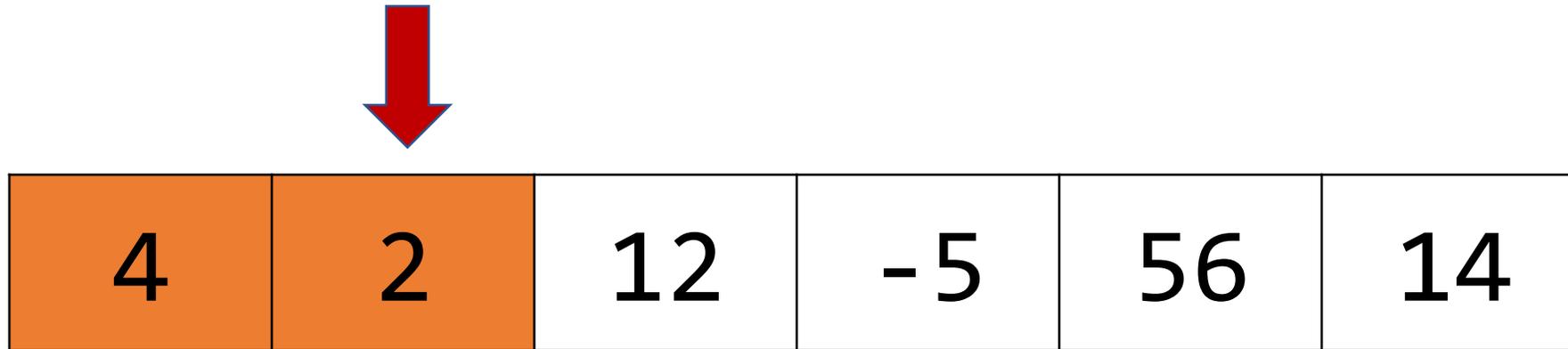
Let's write a function `bubble_sort_int` to sort a list of integers using the **bubble sort algorithm**.

4	2	12	-5	56	14
---	---	----	----	----	----

Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!

Bubble Sort

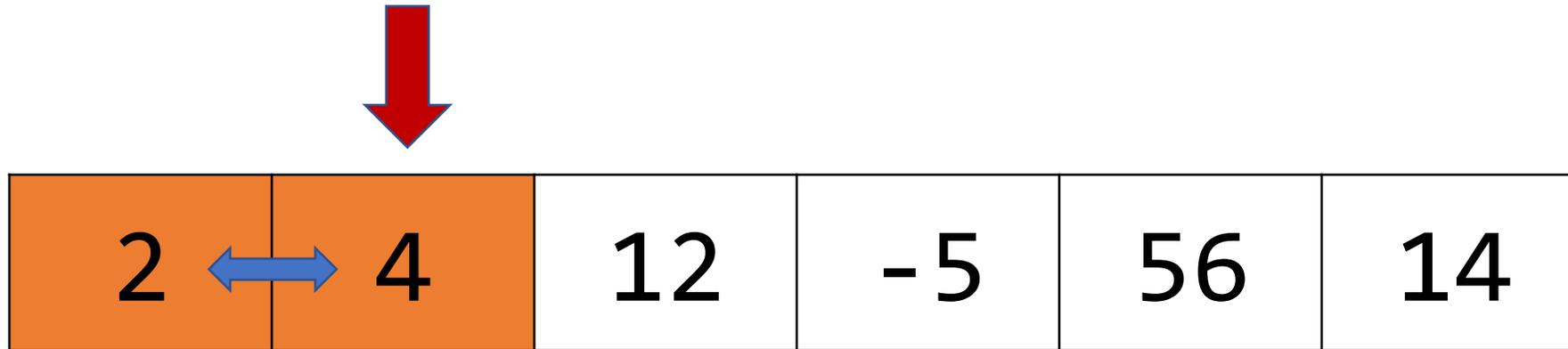
Let's write a function `bubble_sort_int` to sort a list of integers using the **bubble sort algorithm**.



Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!

Bubble Sort

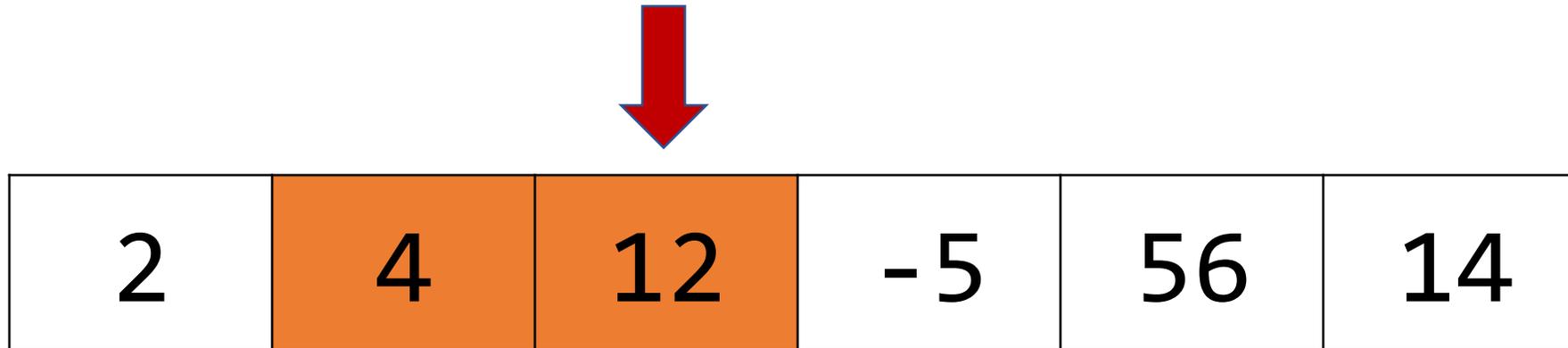
Let's write a function `bubble_sort_int` to sort a list of integers using the **bubble sort algorithm**.



Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!

Bubble Sort

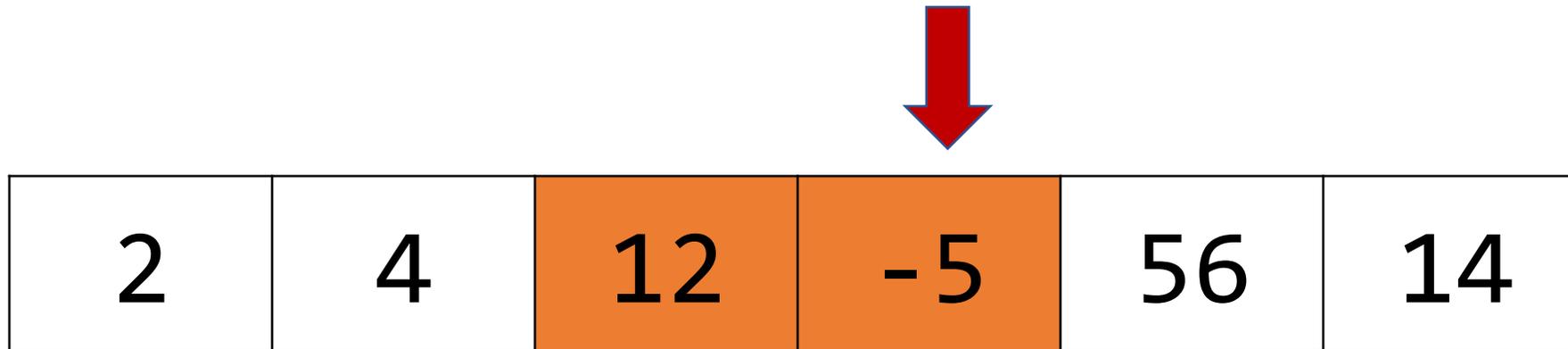
Let's write a function `bubble_sort_int` to sort a list of integers using the **bubble sort algorithm**.



Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!

Bubble Sort

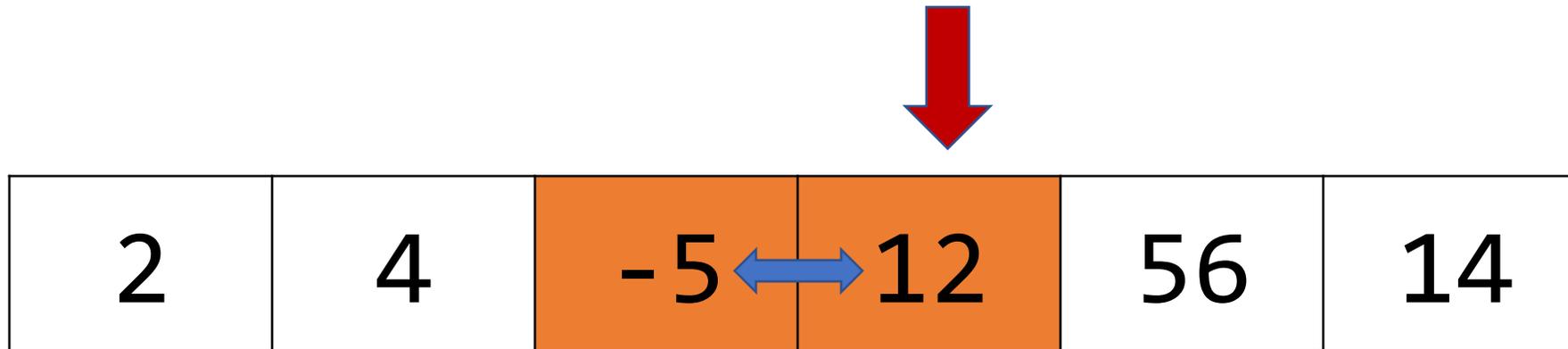
Let's write a function `bubble_sort_int` to sort a list of integers using the **bubble sort algorithm**.



Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!

Bubble Sort

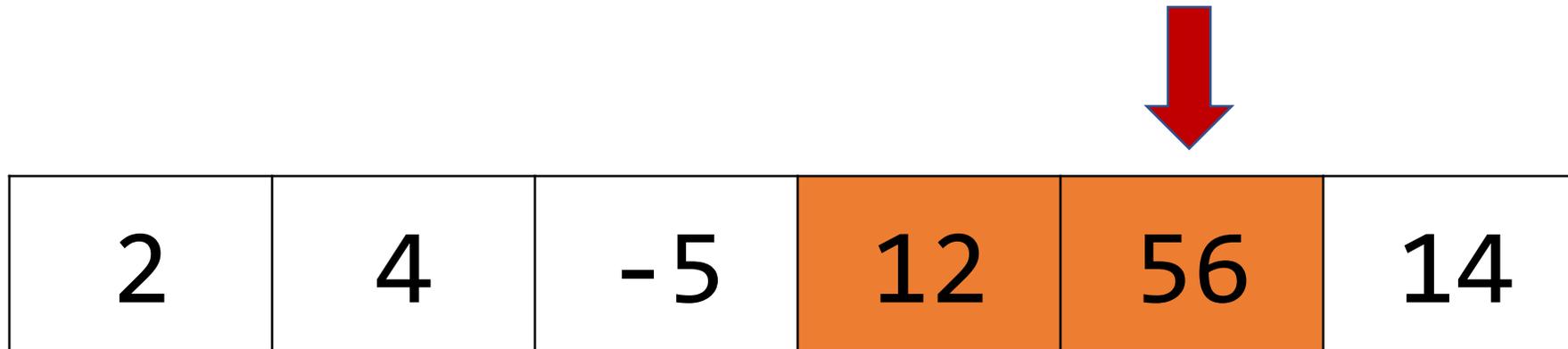
Let's write a function `bubble_sort_int` to sort a list of integers using the **bubble sort algorithm**.



Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!

Bubble Sort

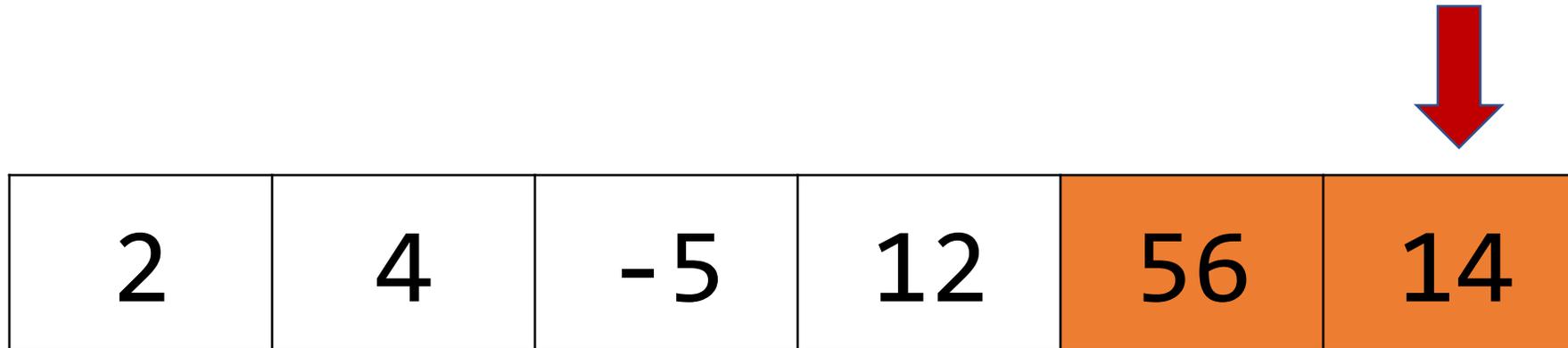
Let's write a function `bubble_sort_int` to sort a list of integers using the **bubble sort algorithm**.



Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!

Bubble Sort

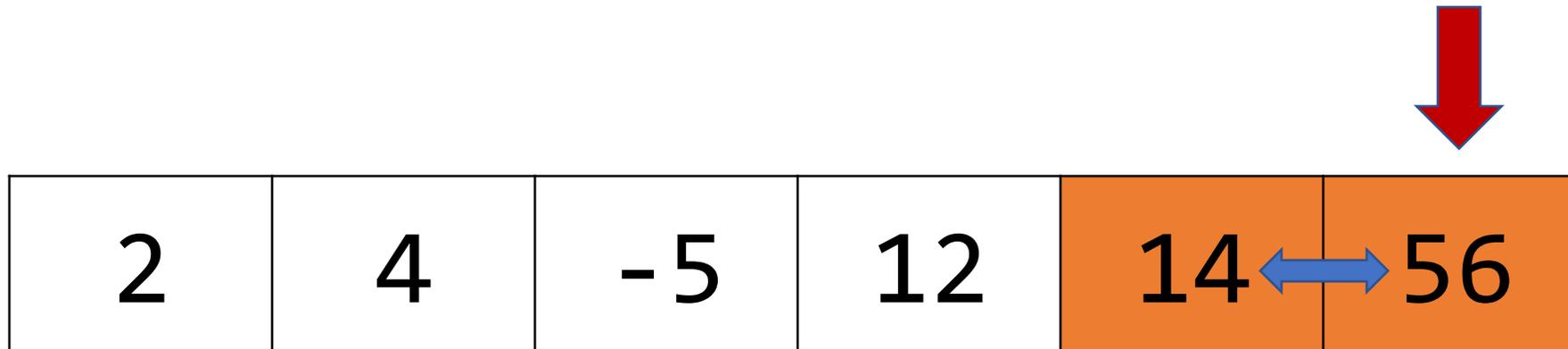
Let's write a function `bubble_sort_int` to sort a list of integers using the **bubble sort algorithm**.



Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!

Bubble Sort

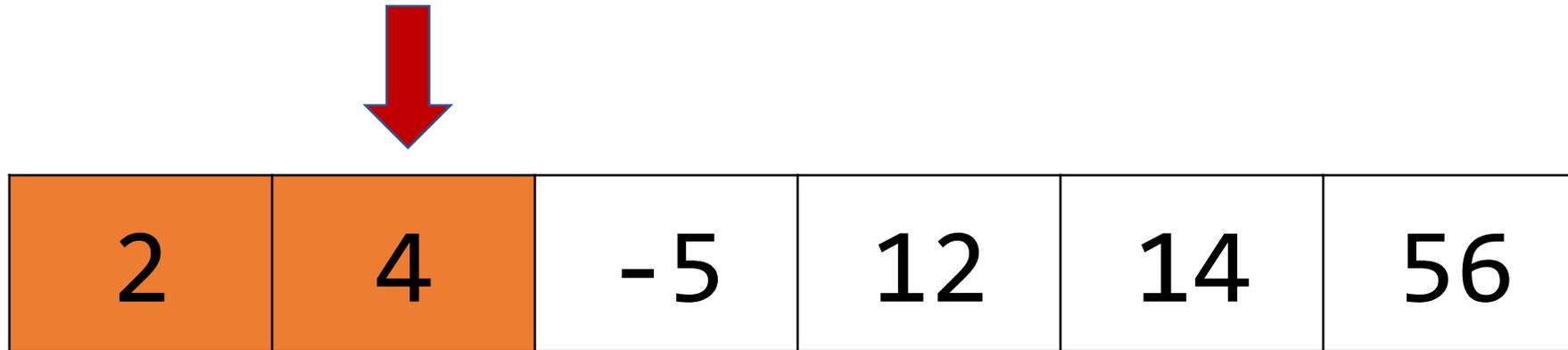
Let's write a function `bubble_sort_int` to sort a list of integers using the **bubble sort algorithm**.



Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!

Bubble Sort

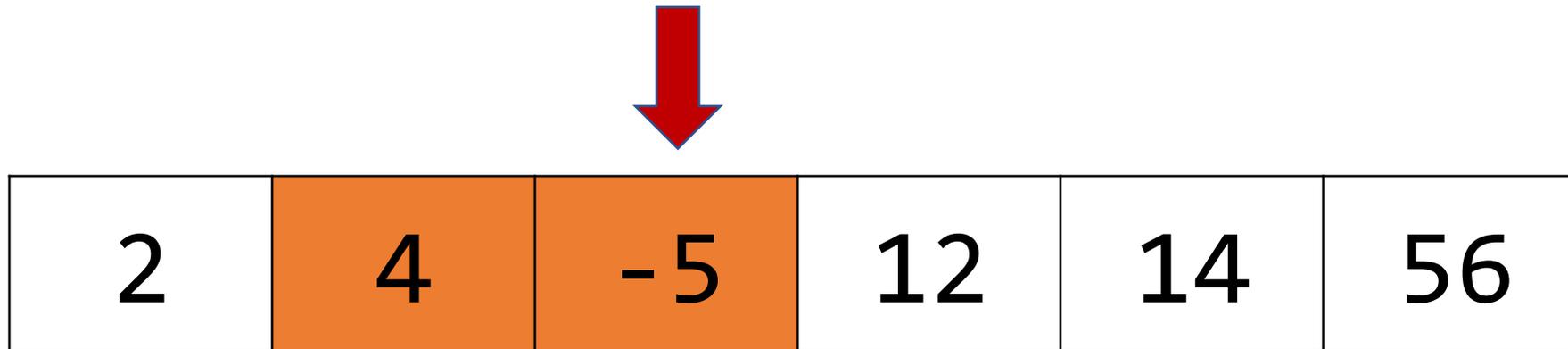
Let's write a function `bubble_sort_int` to sort a list of integers using the **bubble sort algorithm**.



Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!

Bubble Sort

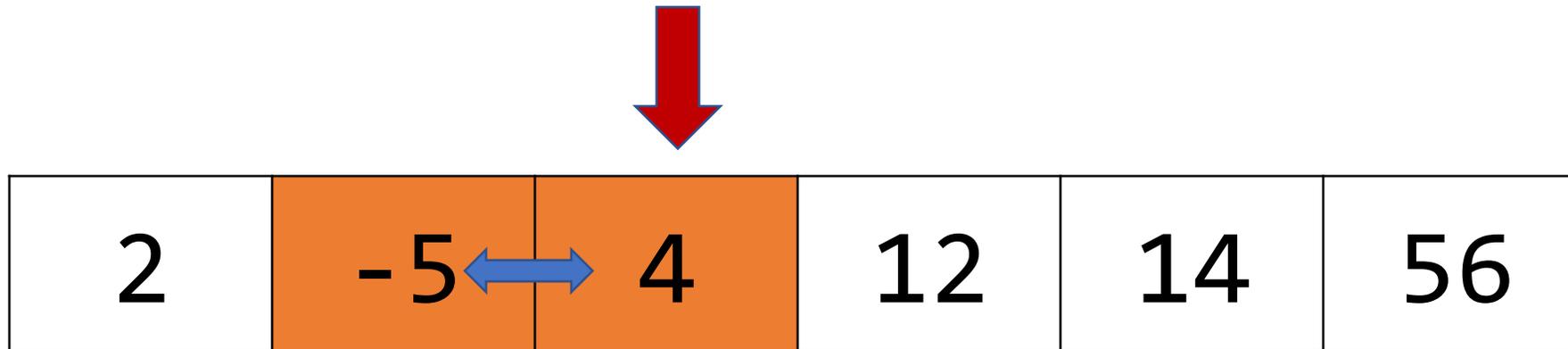
Let's write a function `bubble_sort_int` to sort a list of integers using the **bubble sort algorithm**.



Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!

Bubble Sort

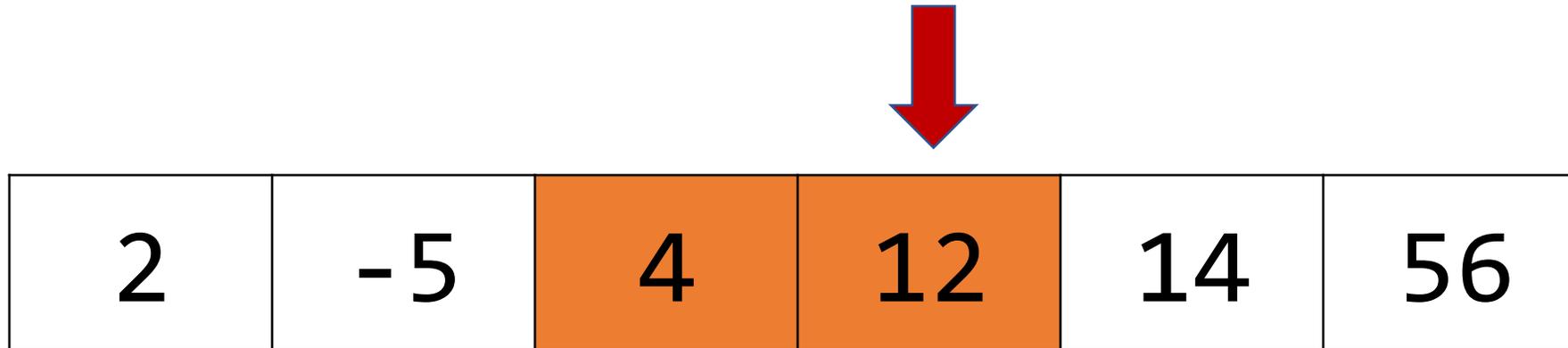
Let's write a function `bubble_sort_int` to sort a list of integers using the **bubble sort algorithm**.



Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!

Bubble Sort

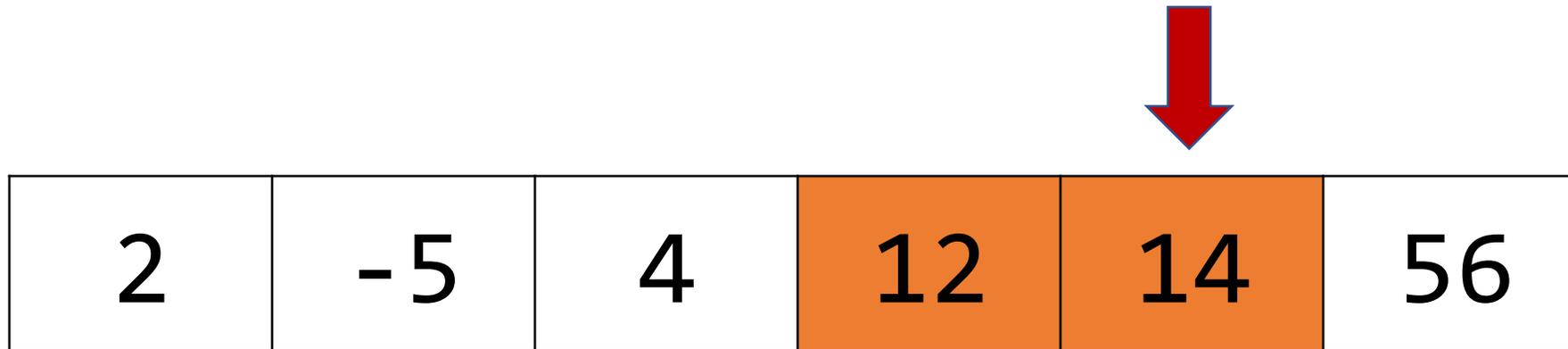
Let's write a function `bubble_sort_int` to sort a list of integers using the **bubble sort algorithm**.



Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!

Bubble Sort

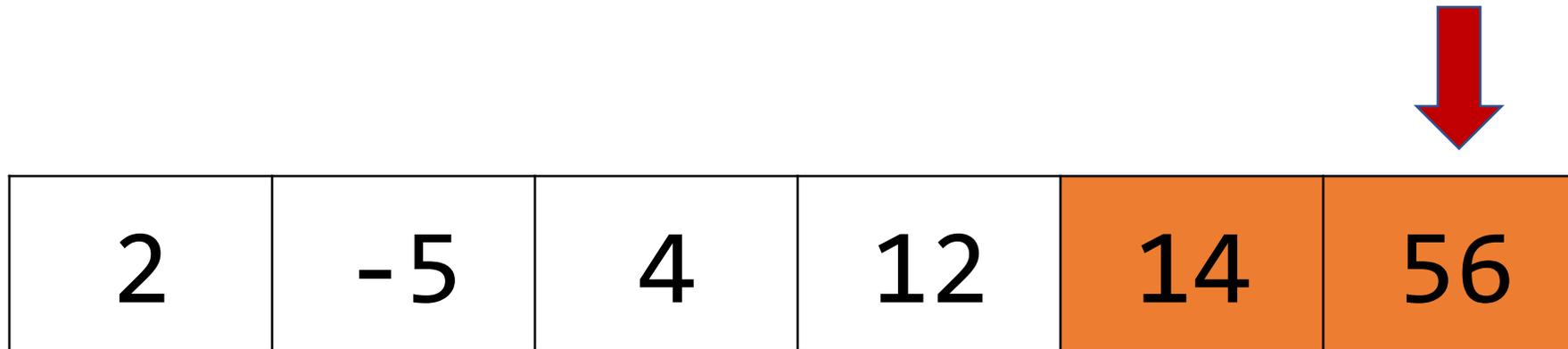
Let's write a function `bubble_sort_int` to sort a list of integers using the **bubble sort algorithm**.



Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!

Bubble Sort

Let's write a function `bubble_sort_int` to sort a list of integers using the **bubble sort algorithm**.



Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!

In general, bubble sort requires up to $n - 1$ passes to sort an array of length n , though it may end sooner if a pass doesn't swap anything.

Bubble Sort

Let's write a function `bubble_sort_int` to sort a list of integers using the **bubble sort algorithm**.

-5	2	4	12	14	56
----	---	---	----	----	----



Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!

Only two more passes are needed to arrive at the above. The first exchanges the 2 and the -5, and the second leaves everything as is.

Integer Bubble Sort

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

How can we make this function more generic?
To start, this function always sorts in ascending order. What about other orders?

Integer Bubble Sort

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

We can add parameters, but they only help so much. What about other orders we can't anticipate? (odd-before-even, etc.)

Integer Bubble Sort

```
void bubble_sort_int(int *arr, size_t n) {
    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;
        }
    }
}
```

What we really want is this – but we don't know how to implement this function...the person calling this function does, though!

**Key Idea: have the caller
pass a function as a
parameter that takes two
ints and tells us whether
we should swap them.**

Integer Bubble Sort

```
void bubble_sort_int(int *arr, size_t n, type?? should swap) {
    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;
        }
    }
}
```

Lecture Plan

- Generics So Far
- Generic Array Swap
- **Motivating Example:** Bubble Sort
- **Function Pointers**
- Generic Function Pointers

```
cp -r /afs/ir/class/cs107/lecture-code/lect13 .
```

Function Pointers

A *function pointer* is the variable type for passing a function as a parameter. Here is how the parameter's type is declared in this case.

```
bool (*should_swap)(int, int)
```

Function Pointers

A *function pointer* is the variable type for passing a function as a parameter. Here is how the parameter's type is declared in this case.

bool (*should_swap)(int, int)



Return type
(bool)

Function Pointers

A *function pointer* is the variable type for passing a function as a parameter. Here is how the parameter's type is declared in this case.

```
bool (*should_swap)(int, int)
```

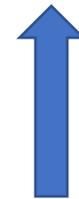


Function pointer name
(should_swap)

Function Pointers

A *function pointer* is the variable type for passing a function as a parameter. Here is how the parameter's type is declared in this case.

```
bool (*should_swap)(int, int)
```



Function parameters
(two ints)

Function Pointers

Here's the general variable type syntax:

[return type] ([name]) ([parameters])*

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;
        }
    }
}
```

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.

Function Pointers

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

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

```
// 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.

Function Pointers

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

bool sort_odd_then_even(int first_num,
                        int second_num) {
    return (second_num % 2 != 0) &&
           (first_num % 2 == 0);
}

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_odd_then_even);
    ...
}
```

```
// 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.

Function Pointers

Passing a non-function as a parameter allows us to pass data around our program. Passing a function as a parameter allows us to pass logic around our program.

- When writing a generic function, if we don't know how to do something in the way the caller wants, we can ask them to pass in a function parameter that can do it for us.
- Also called a “callback” function – function “calls back to” the caller.
 - **Function writer:** writes generic algorithmic functions, relies on caller-provided data
 - **Function caller:** knows the data, doesn't know how the algorithm works

Lecture Plan

- Generics So Far
- Generic Array Swap
- **Motivating Example:** Bubble Sort
- Function Pointers
- **Generic Function Pointers**

```
cp -r /afs/ir/class/cs107/lecture-code/lect13 .
```

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 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.

bubblesort.h/c



Generic Bubble Sort

```
void bubble_sort(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;  
        }  
    }  
}
```

Let's start by making the parameters and swap generic.

Generic Bubble Sort

```
void bubble_sort(void *arr, size_t n, size_t elem_size_bytes,
                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;
                swap(&arr[i - 1], &arr[i], elem_size_bytes);
            }
        }
        if (!swapped) {
            return;
        }
    }
}
```

Let's start by making the parameters and swap generic.

Generic Bubble Sort

```
void bubble_sort(void *arr, size_t n, size_t elem_size_bytes,
                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;
                swap(&arr[i - 1], &arr[i], elem_size_bytes);
            }
        }
        if (!swapped) {
            return;
        }
    }
}
```

Problem: we can't do pointer arithmetic / indexing with **void *s!**

Generic Bubble Sort

```
void bubble_sort(void *arr, size_t n, size_t elem_size_bytes,
                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;
                swap(arr + i - 1, arr + i, elem_size_bytes);
            }
        }
        if (!swapped) {
            return;
        }
    }
}
```

Problem: we can't do pointer arithmetic / indexing with **void *s!**

Key Idea: Locating i-th Elem

A common generics idiom is getting a pointer to the i-th element of a generic array. From previously, we know how to locate the **last** element:

```
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {  
    swap(arr, (char *)arr + (nelems - 1) * elem_bytes, elem_bytes);  
}
```

What can we use as an expression to get the location of the i-th element?

Respond on PollEv:
pollev.com/cs107



What is the expression to get a pointer to the i -th element of a generic array?

Nobody has responded yet.

Hang tight! Responses are coming in.

Key Idea: Locating i-th Elem

A common generics idiom is getting a pointer to the i-th element of a generic array. We know how to locate the **last** element:

```
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {  
    swap(arr, (char *)arr + (nelems - 1) * elem_bytes, elem_bytes);  
}
```

What can we use as an expression to get the location of the i-th element?

```
void *ith_elem = (char *)arr + i * elem_bytes;
```

Generic Bubble Sort

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

Let's start by making the parameters and swap generic.

Generic Bubble Sort

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

How do we make the comparison function generic?

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(int *arr, size_t n, bool (*should_swap)(int, int));
```

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)(int, int));  
void bubble_sort(void *arr, size_t n,  
    size_t elem_size_bytes, bool (*should_swap)(long, long));  
void bubble_sort(void *arr, size_t n,  
    size_t elem_size_bytes, bool (*should_swap)(char *, char *));  
...  
// what we really want is...  
void bubble_sort(void *arr, size_t n,  
    size_t elem_size_bytes, bool (*should_swap)(anything, anything));
```

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. large struct).
- **Key Idea:** require the caller to pass in a *pointer to the data*. Pointers are always 8 bytes big, regardless of what they point to!
- **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

```
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(arr[i - 1], arr[i])) {
                swapped = true;
                swap(p_prev_elem, p_curr_elem, elem_size_bytes);
            }
        }
        if (!swapped) {
            return;
        }
    }
}
```

The comparison function must work with pointers *to the elements*, not the elements themselves.

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;
        }
    }
}
```

The comparison function must work with pointers *to the elements*, not the elements themselves.

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. We'll see some examples next time!

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

Recap

- Generics So Far
- Generic Array Swap
- **Motivating Example:**
Bubble Sort
- Function Pointers
- Generic Function Pointers

Lecture 13 takeaway: We can do pointer arithmetic with a **void *** by casting it to **char *** and adding/subtracting # bytes. A function pointer is a type of variable that stores a function, and we can pass them as parameters. A common use case is to pass comparison functions to generic functions like bubble sort that need to compare elements.