CS107, Lecture 13
C Generics – Function Pointers

Reading: K&R 5.11

😷 masks recommended

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
- **Motivating Example:** Bubble Sort
- Function Pointers
- Generic Function Pointers

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

Lecture Plan

- Generics So Far
- **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 perform pointer arithmetic with or dereference (without casting first) a **`void *`**.
- We can use **`memcpy`** or **`memmove`** to copy data from one memory location to another.
- To do pointer arithmetic with a **`void *`**, we must first cast it to a **`char *`**.
- **`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.

Generic Array Swap

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

We can cast to a **char *** in order to perform manual byte arithmetic with void * pointers.

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 = 0xeeeeeeee;
swap(&x, &y, sizeof(short));

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

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 3rd entry’s bytes to 1s
```

Lecture Plan

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

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

Bubble Sort

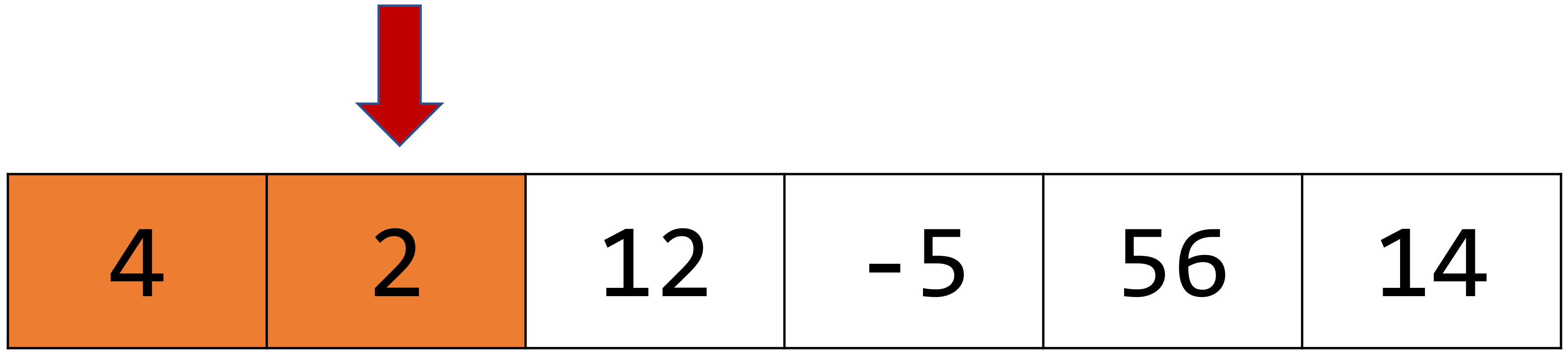
Let’s write a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.

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

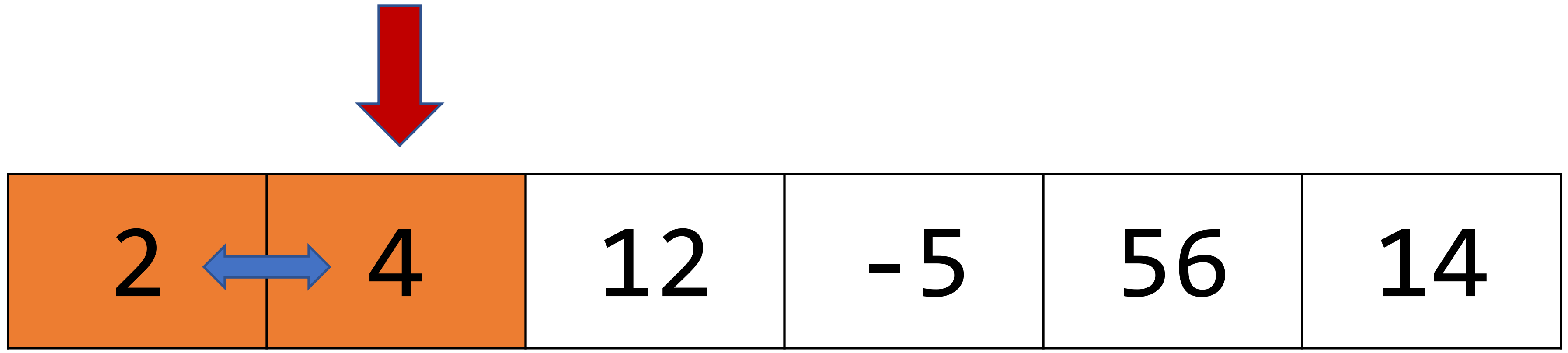
Let's write a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.



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

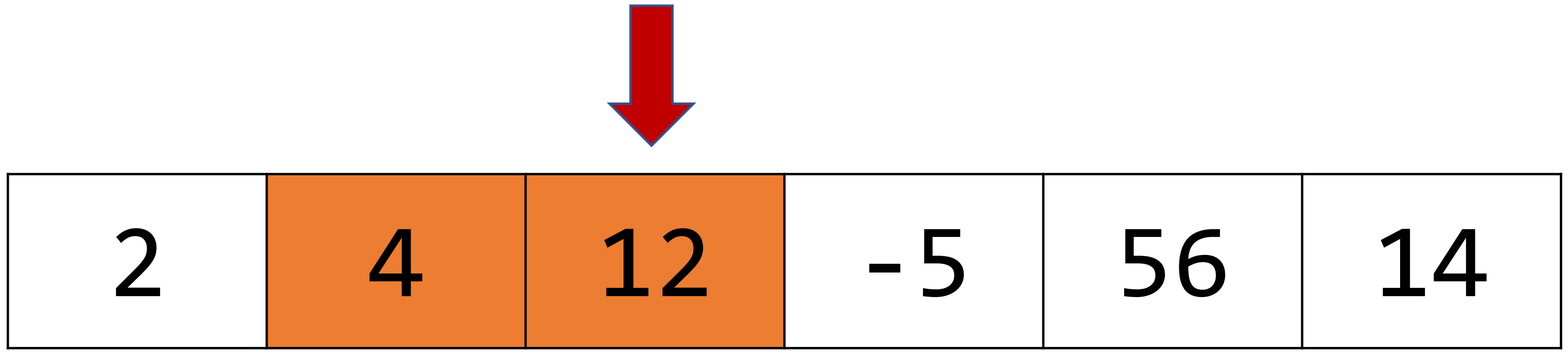
Let’s write a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.



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

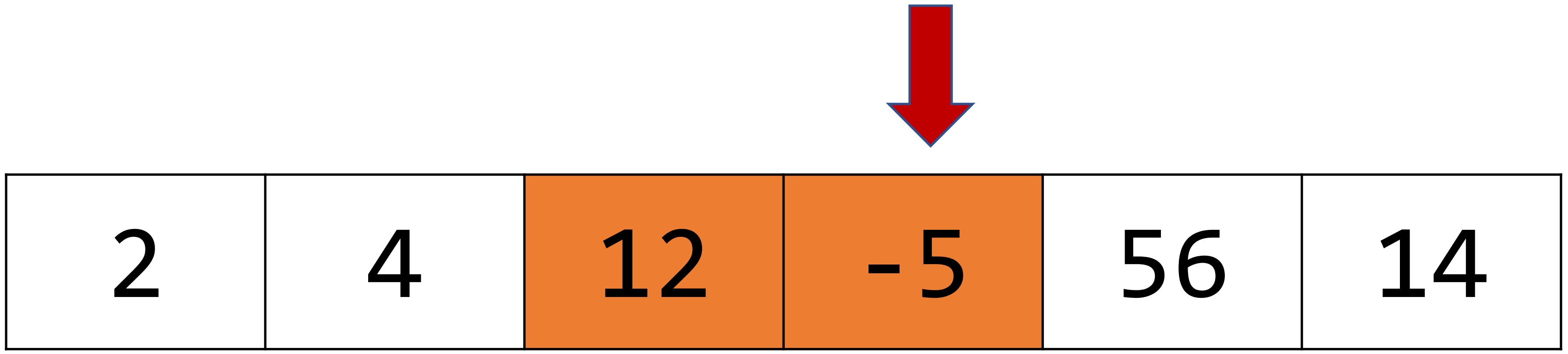
Let’s write a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.



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

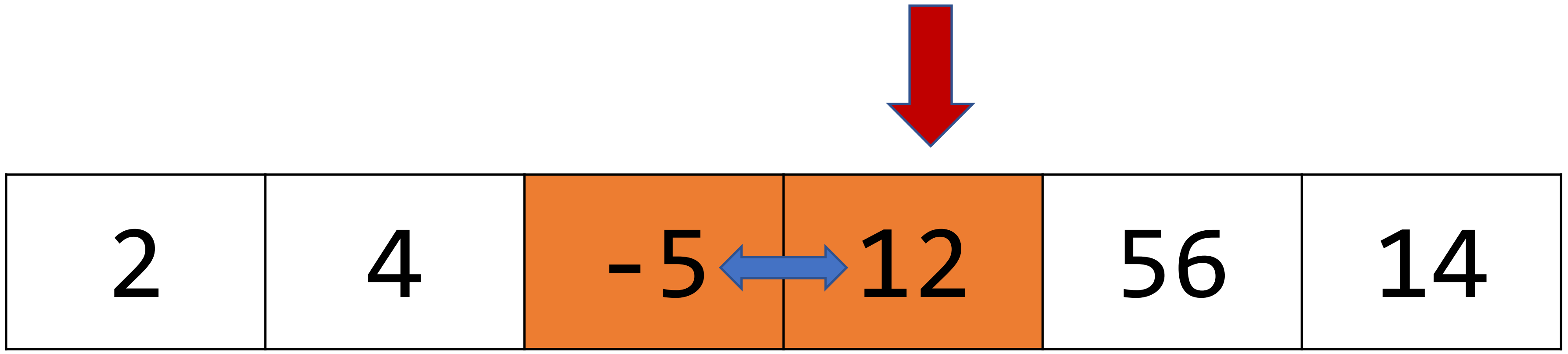
Let’s write a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.



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

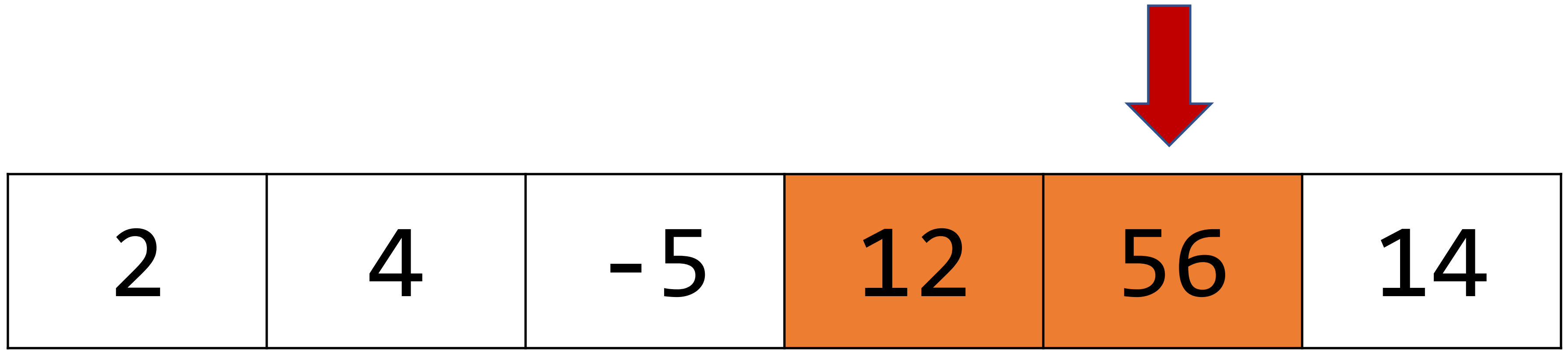
Let’s write a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.



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

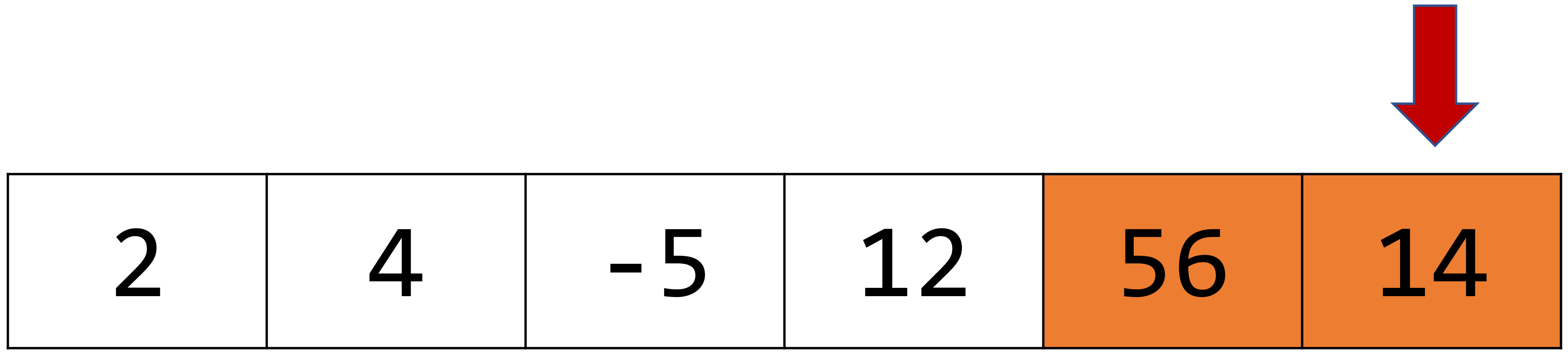
Let's write a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.



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

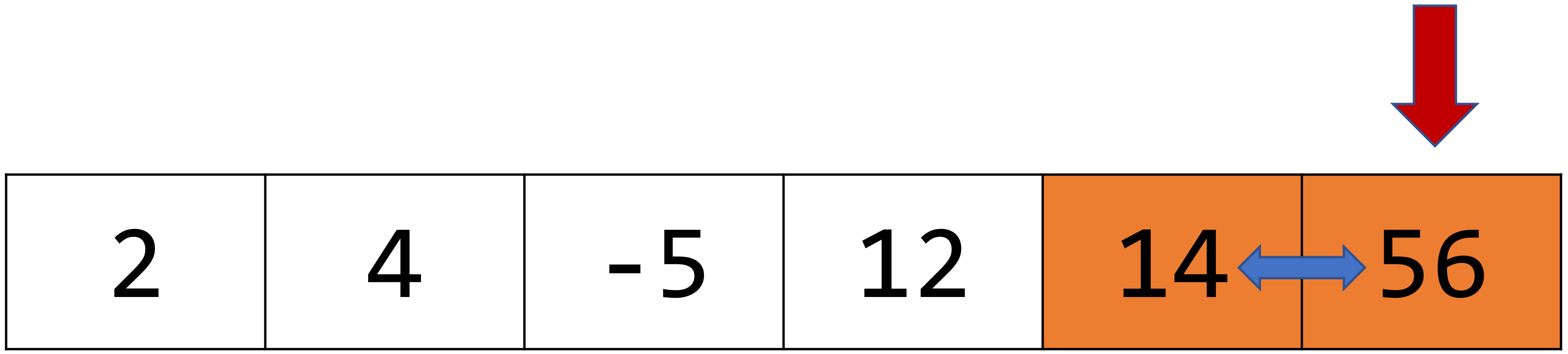
Let's write a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.



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

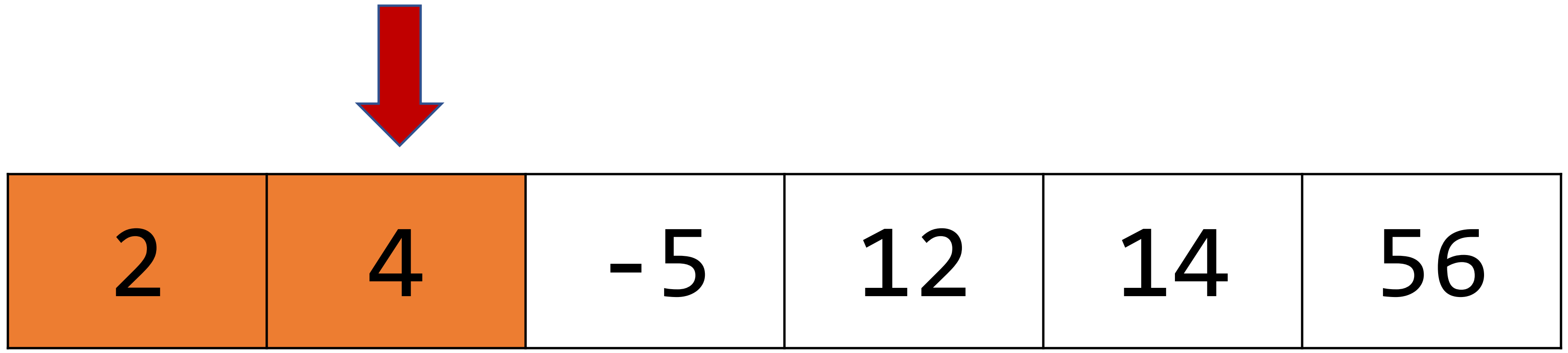
Let's write a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.



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

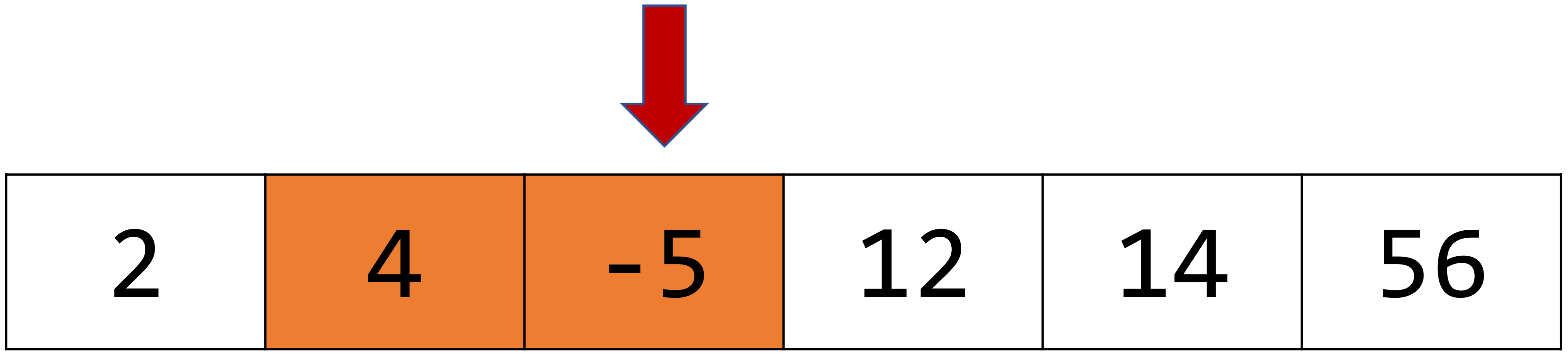
Let’s write a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.



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

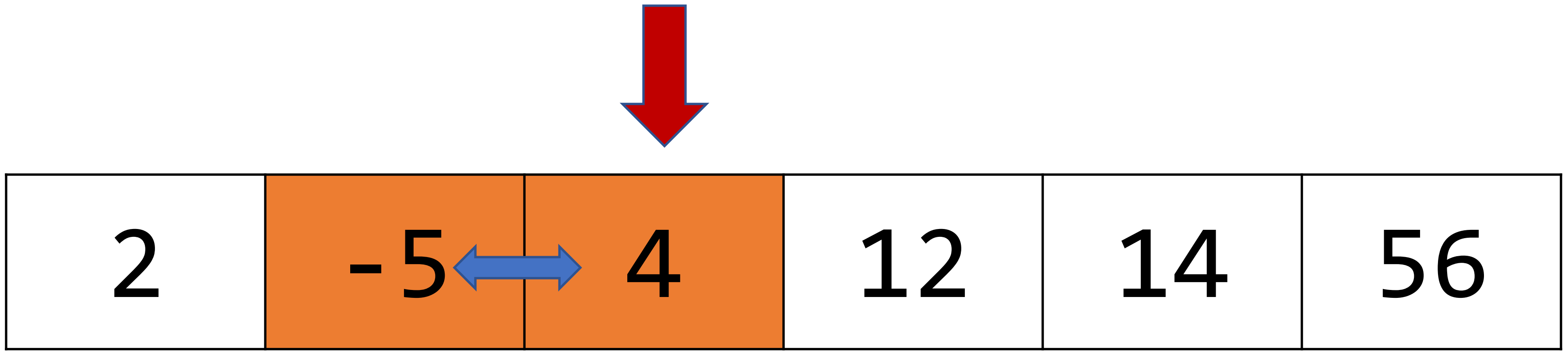
Let’s write a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.



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

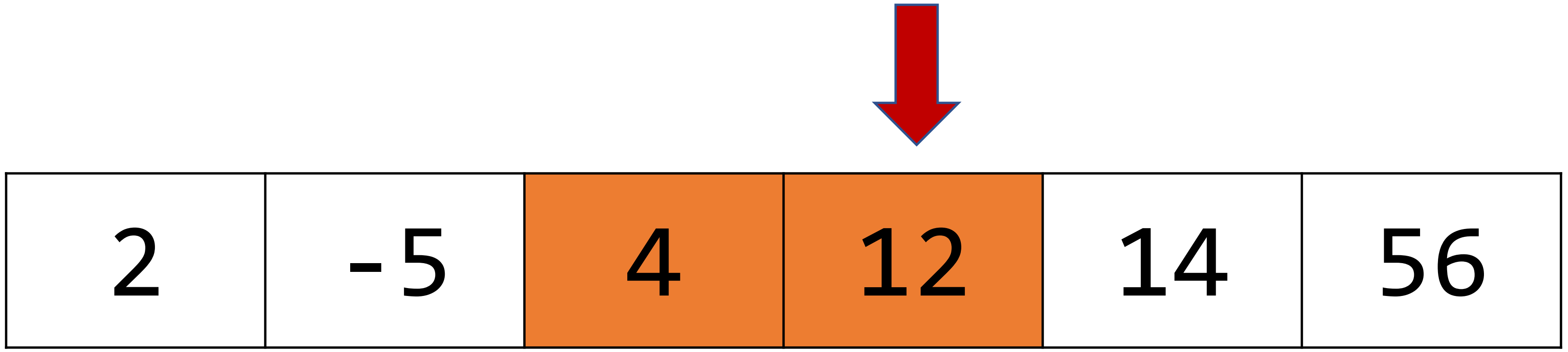
Let’s write a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.



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

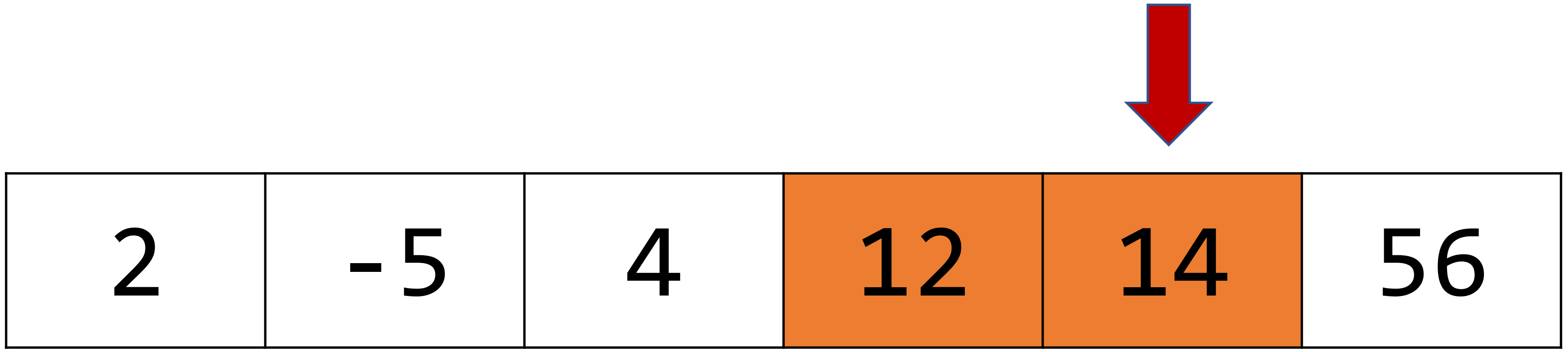
Let's write a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.



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

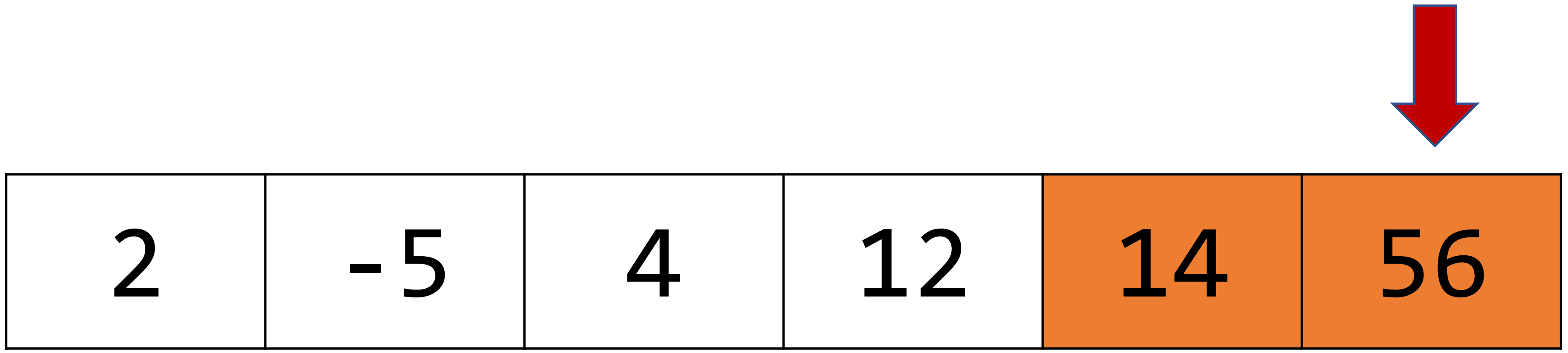
Let's write a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.



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 `int`s 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
- **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

```
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);
    ...
}
```

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

```
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);
    ...
}
```

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

```
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);
    ...
}
```

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
- **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[]) {
    ...
}
```

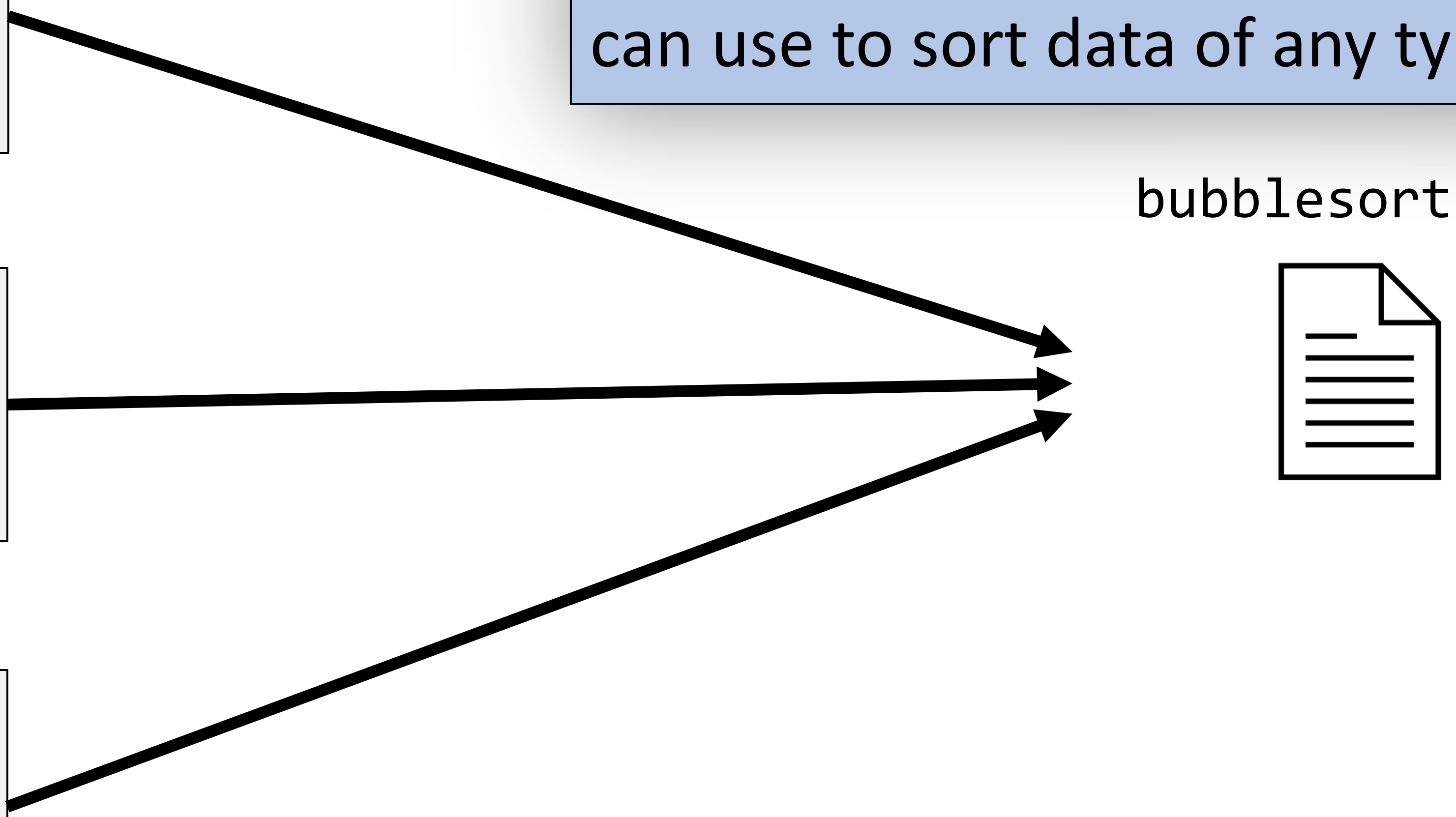
file_that_sorts_strings.c

```
#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.



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, 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.

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

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,
                 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 last lecture, 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);
}
```

How can we generalize this 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)(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;
        }
    }
}
```

Now let’s make the comparison function generic.

Comparison Functions

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

- The standard comparison function in many C functions provides even more information. 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
- This is the same return value format as **strcmp**!

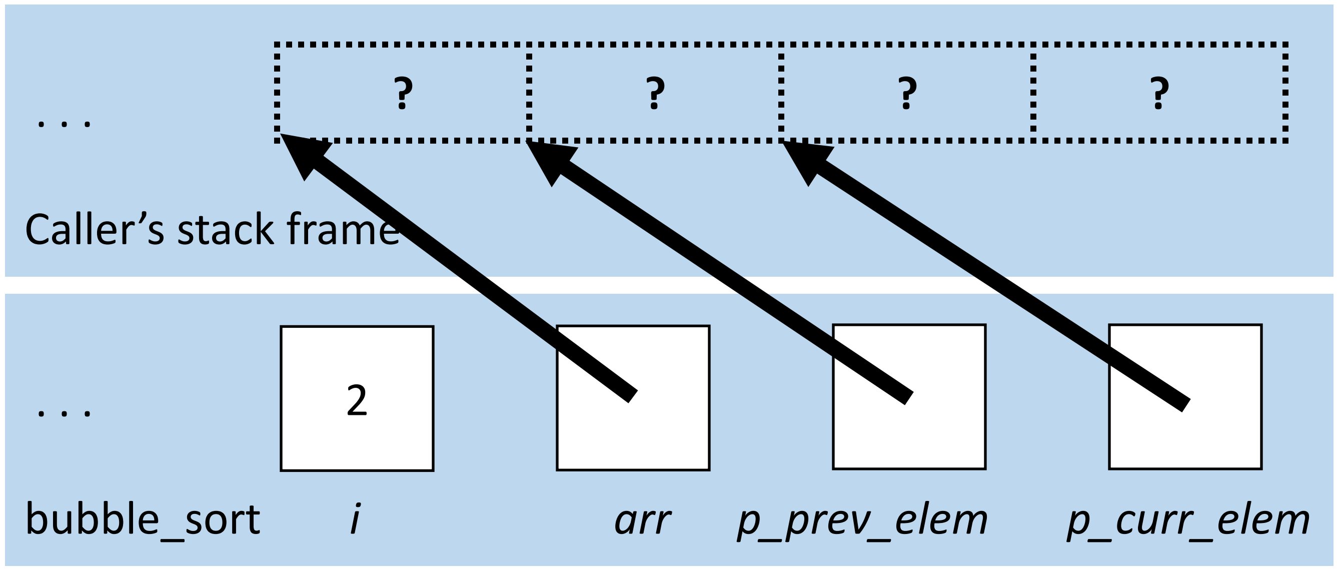
```
int (*compare_fn)(void *, void *)
```

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;
        }
    }
}
```

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;
        }
    }
}
```



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)

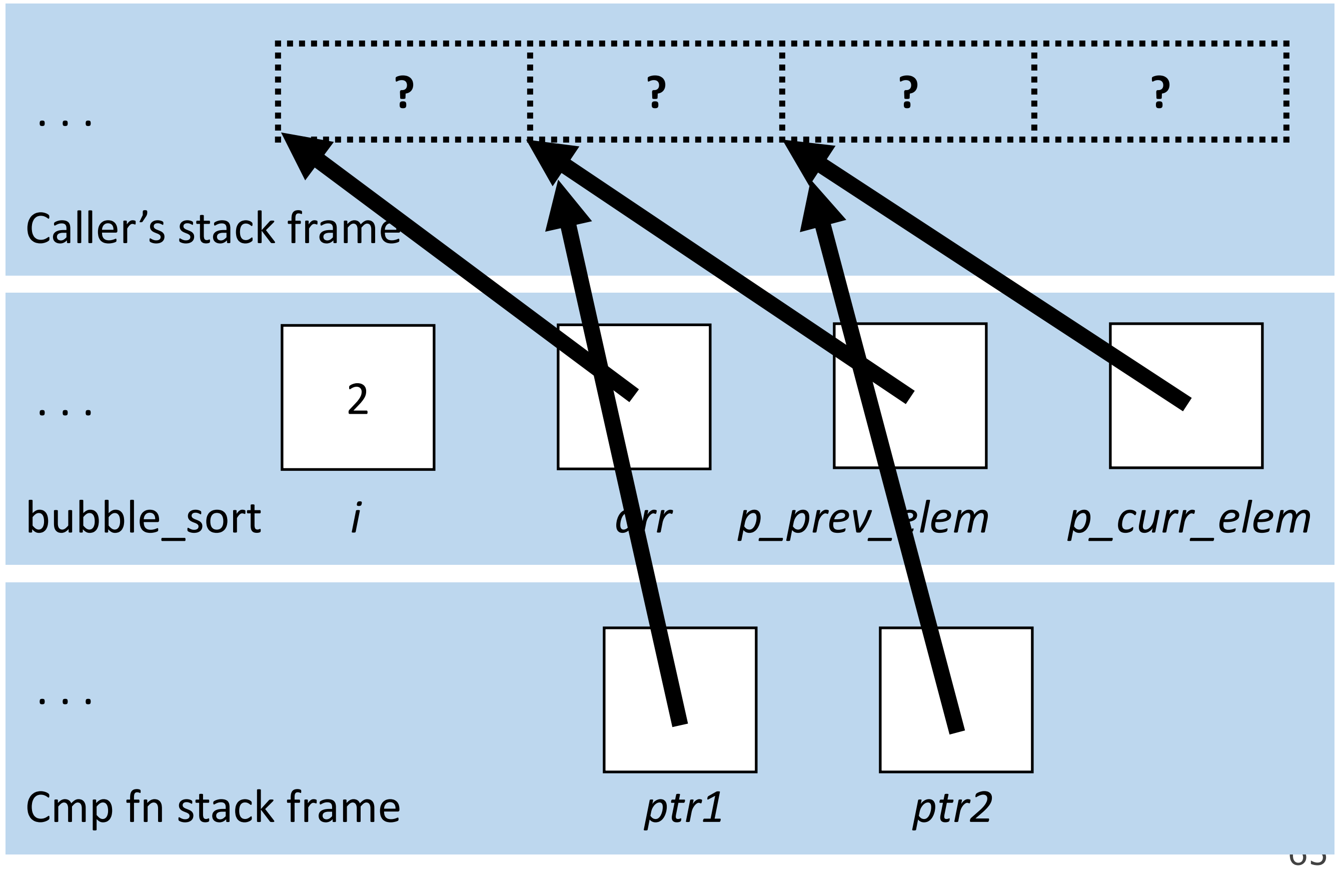
Function Pointers

```
int sort_descending(void *ptr1, void *ptr2) {
    // 1) cast arguments to what they really are here
     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;
}
```



Recap

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

Lecture 13 takeaway: A function pointer is a type of variable that stores a function. 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.