CS107 Lecture 9
More Generics in C: Function pointers

Reading: K&R 5.11

Slides by Jerry Cain and Lisa Yan, who leveraged prior work by Nick Troccoli, Julie Zelenski, Marty Stepp, Cynthia Lee, Chris Gregg, and others.

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

Plan For Today

- **Finish up:** Generic Stack
- Function Pointers
- **Example:** Bubble Sort

```
cp -r /afs/ir/class/cs107/samples/lectures/lect9 .
```

Plan For Today

- **Finish up:** Generic Stack
- Function Pointers
- **Example:** Bubble Sort

Plan For Today

- **Finish up:** Generic Stack
- Function Pointers
- **Example:** Bubble Sort

Bubble Sort

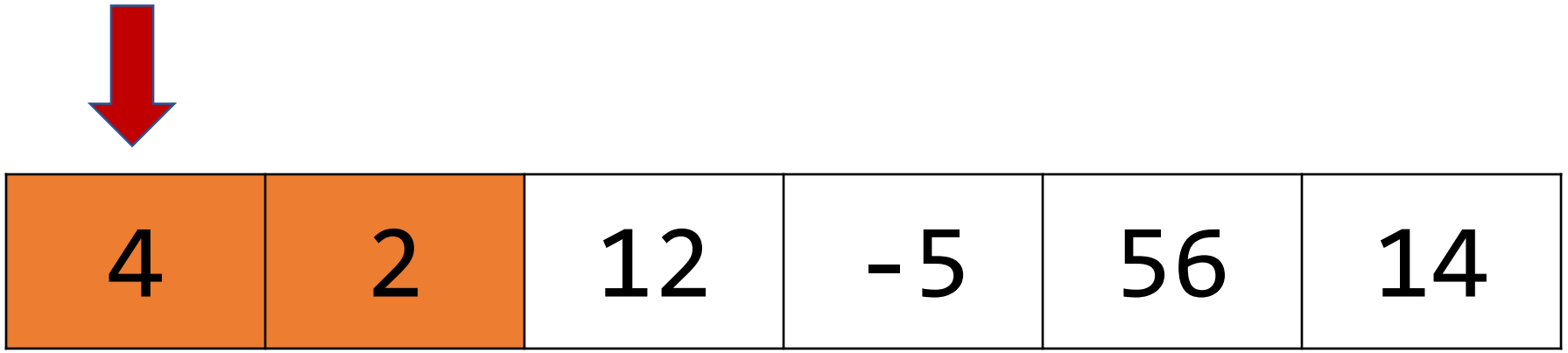
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.

4	2	12	-5	56	14

- Bubble sort repeatedly passes over the array, exchanging neighboring elements when they're out of order.

Bubble Sort

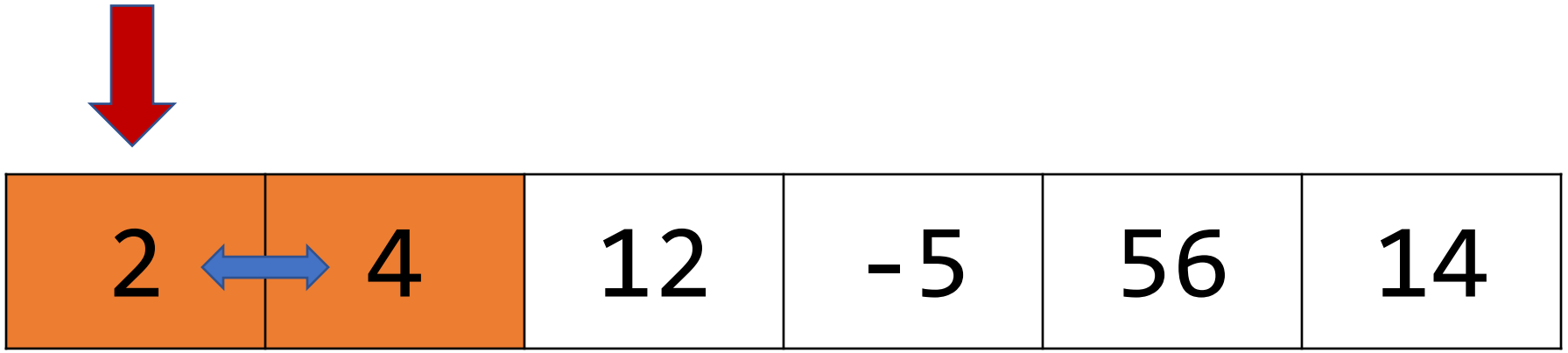
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- Bubble sort repeatedly passes over the array, exchanging neighboring elements when they're out of order.

Bubble Sort

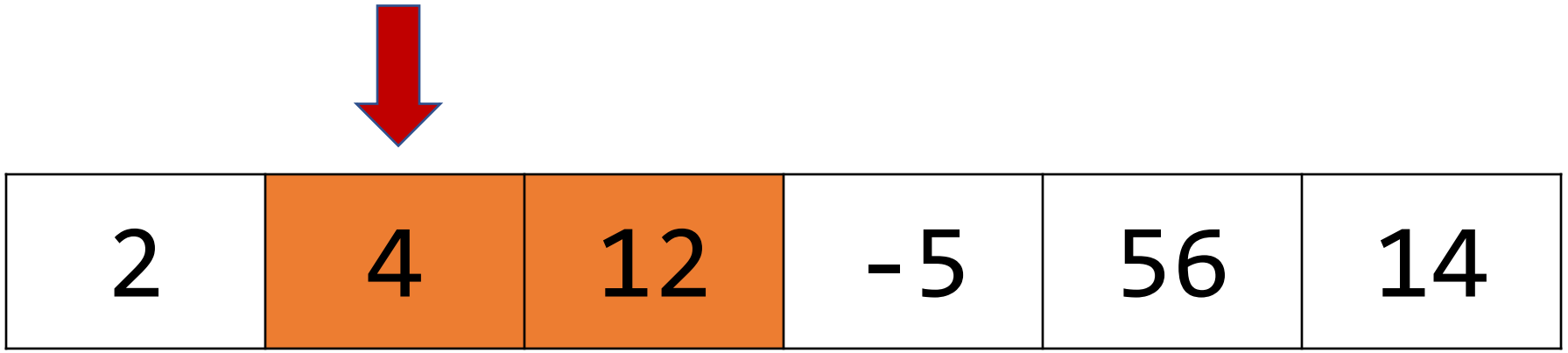
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- Bubble sort repeatedly passes over the array, exchanging neighboring elements when they're out of order.

Bubble Sort

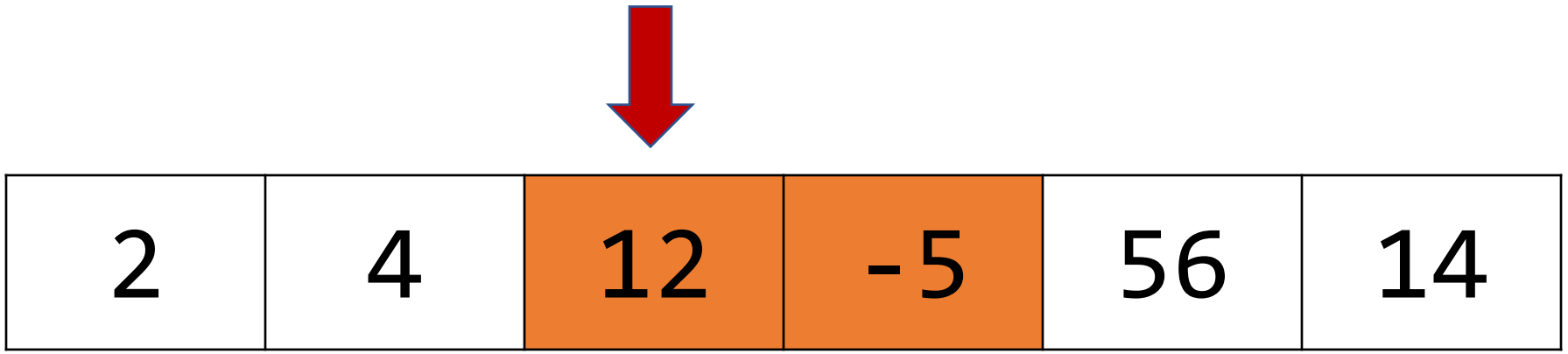
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- Bubble sort repeatedly passes over the array, exchanging neighboring elements when they're out of order.

Bubble Sort

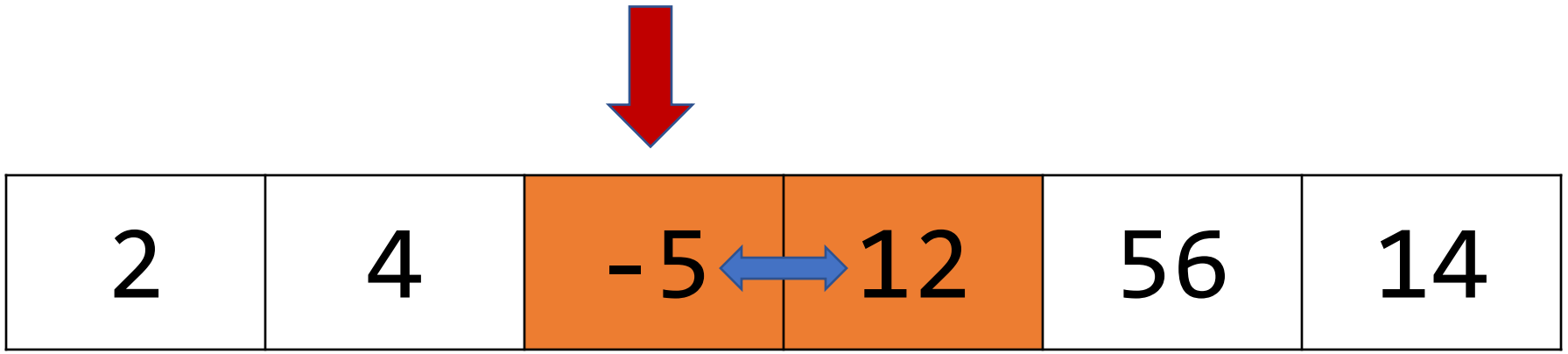
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- Bubble sort repeatedly passes over the array, exchanging neighboring elements when they're out of order.

Bubble Sort

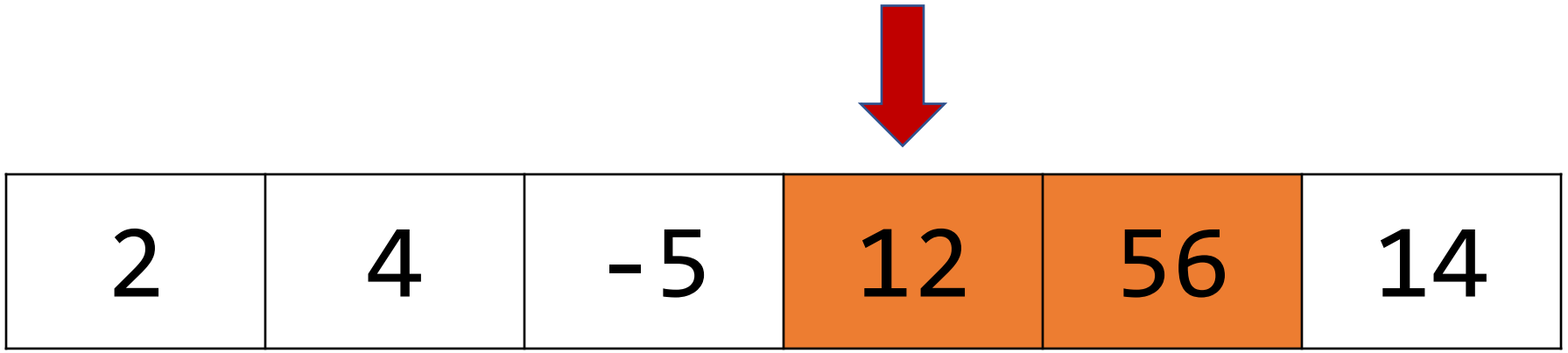
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- Bubble sort repeatedly passes over the array, exchanging neighboring elements when they're out of order.

Bubble Sort

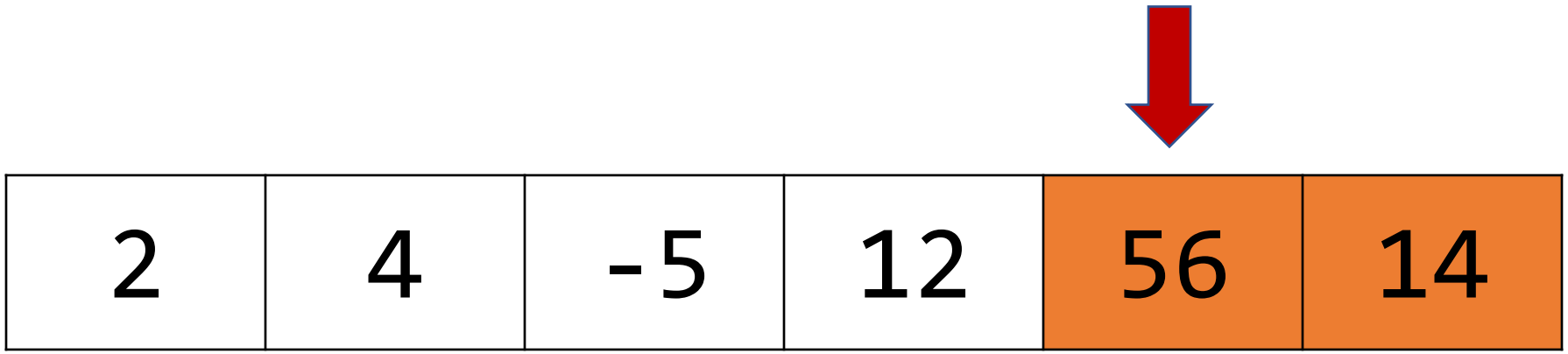
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- Bubble sort repeatedly passes over the array, exchanging neighboring elements when they're out of order.

Bubble Sort

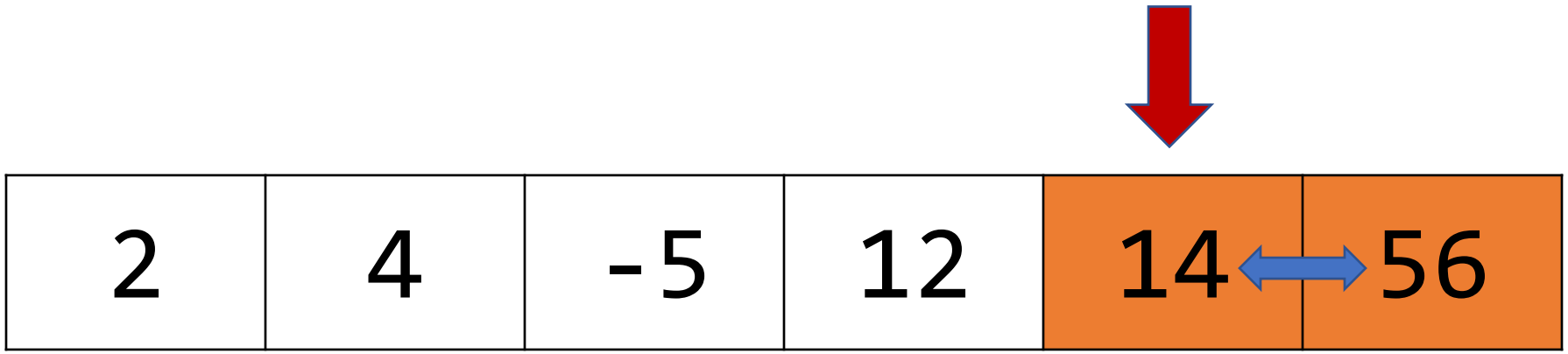
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- Bubble sort repeatedly passes over the array, exchanging neighboring elements when they're out of order.

Bubble Sort

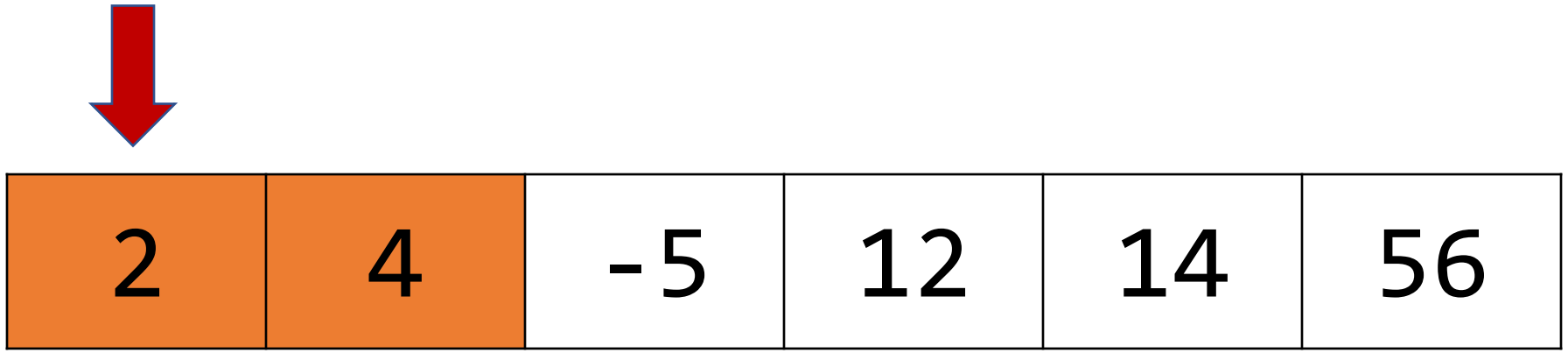
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- Bubble sort repeatedly passes over the array, exchanging neighboring elements when they're out of order.

Bubble Sort

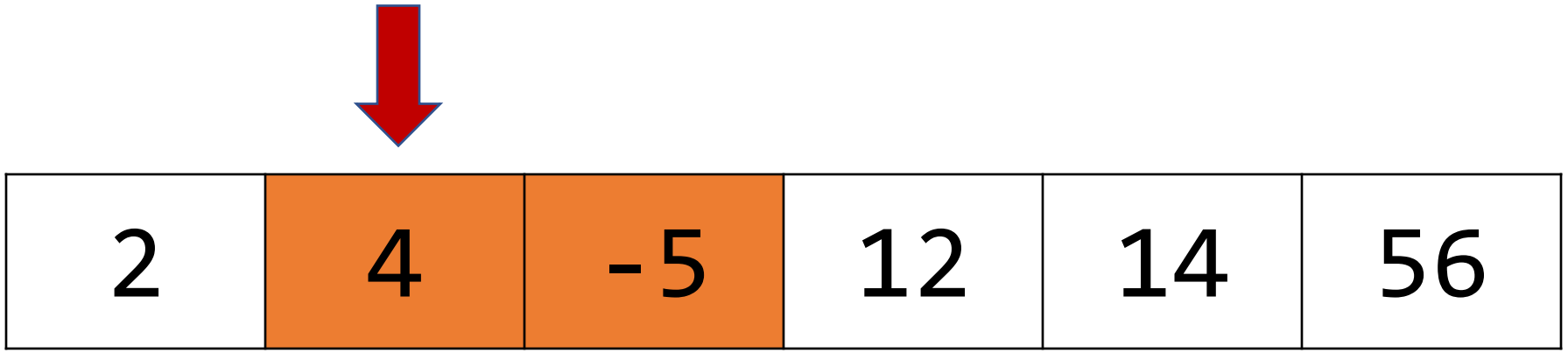
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- Bubble sort repeatedly passes over the array, exchanging neighboring elements when they're out of order.

Bubble Sort

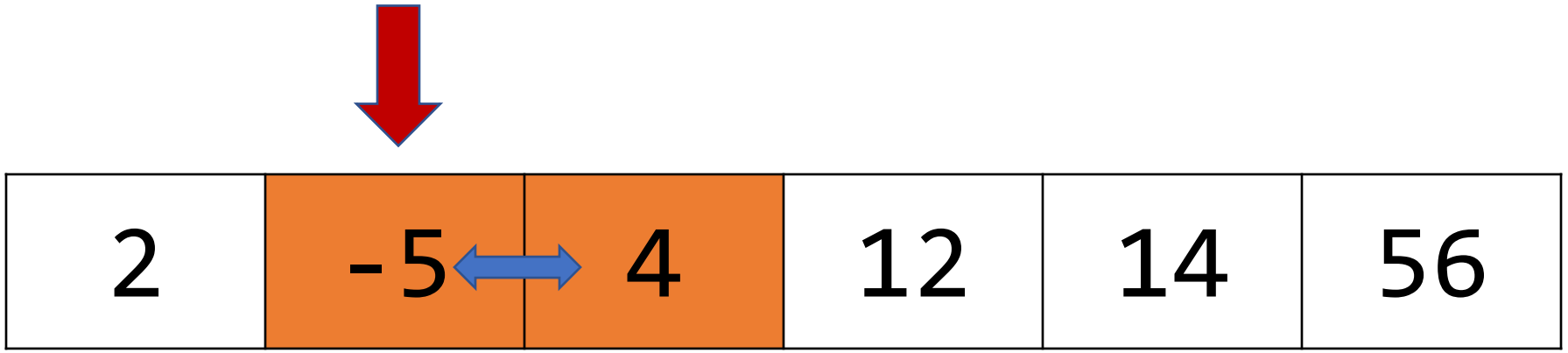
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- Bubble sort repeatedly passes over the array, exchanging neighboring elements when they're out of order.

Bubble Sort

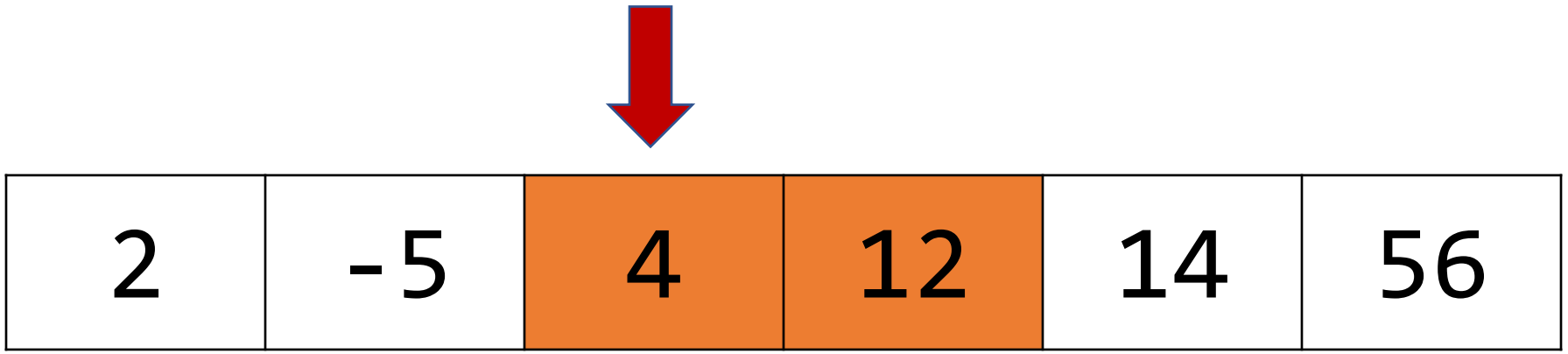
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- Bubble sort repeatedly passes over the array, exchanging neighboring elements when they're out of order.

Bubble Sort

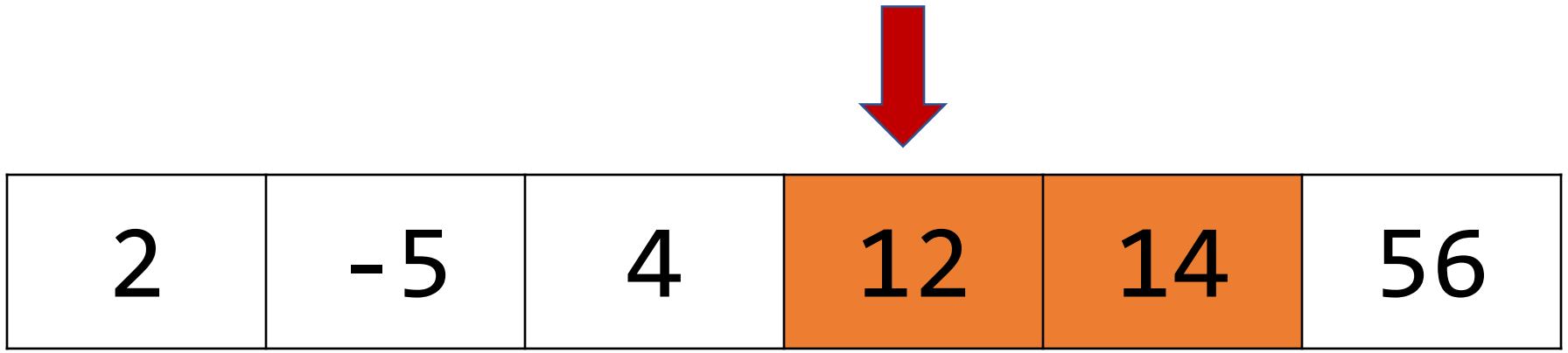
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- Bubble sort repeatedly passes over the array, exchanging neighboring elements when they're out of order.

Bubble Sort

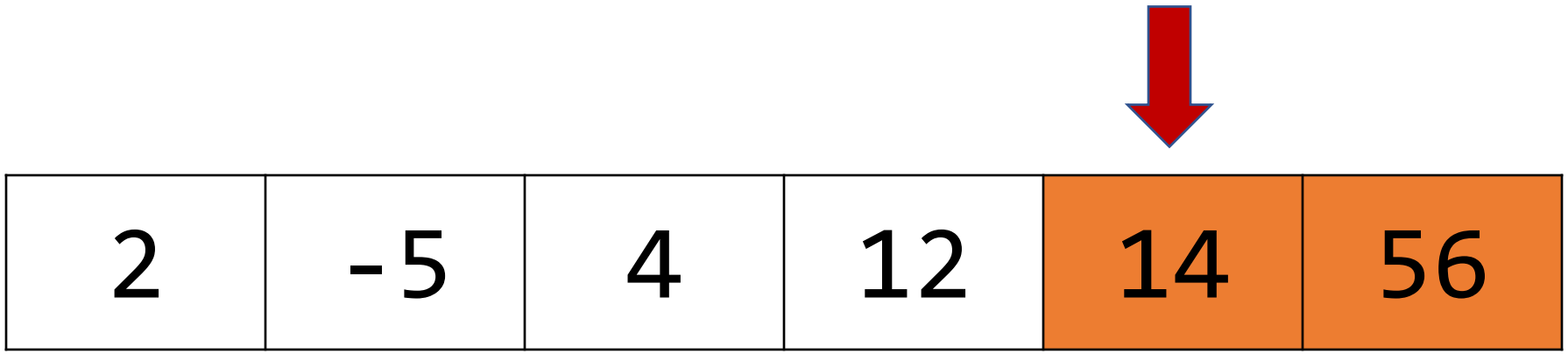
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- Bubble sort repeatedly passes over the array, exchanging neighboring elements when they're out of order.

Bubble Sort

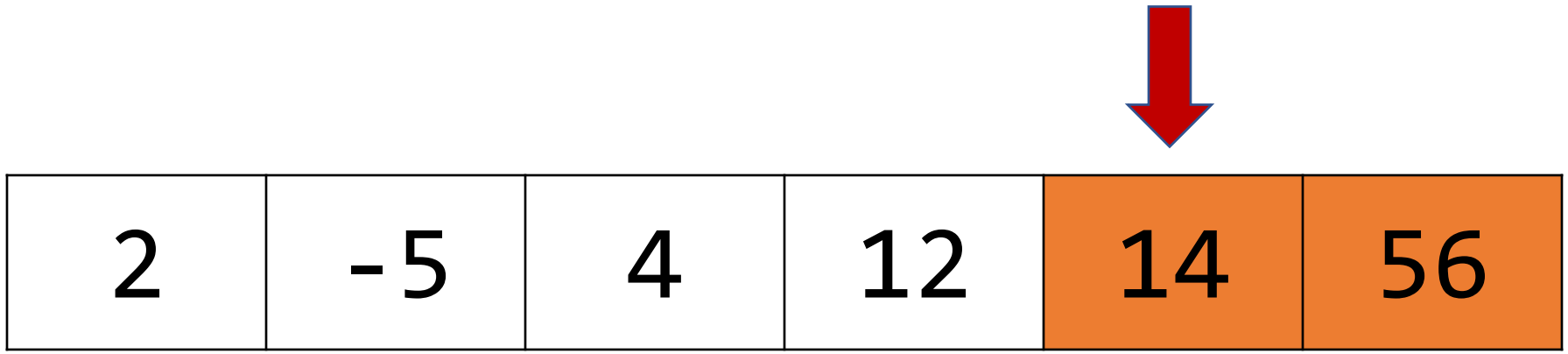
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- Bubble sort repeatedly passes over the array, exchanging neighboring elements when they're out of order.

Bubble Sort

- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- Bubble sort repeatedly passes over the array, exchanging neighboring elements when they're out of order.

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 to sort a list of integers. We'll use the **bubble sort algorithm**.

-5	2	4	12	14	56

- Bubble sort repeatedly passes over the array, exchanging neighboring elements when they're out of order.

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 arr[], size_t n) {

    for (size_t i = 0; i < n - 1; i++) {
        bool swapped = false;
        for (size_t j = 0; j < n - 1; j++) {
            if (arr[j] > arr[j + 1]) { // out of order, so swap!
                swapped = true;
                swap(&arr[j], &arr[j + 1], sizeof(int));
            }
        }

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

How can we make this function generic, to sort an array of *any type?*

Integer Bubble Sort

```
void bubble_sort(int arr[], size_t n) {

    for (size_t i = 0; i < n - 1; i++) {
        bool swapped = false;
        for (size_t j = 0; j < n - 1; j++) {
            if (arr[j] > arr[j + 1]) { // out of order, so swap!
                swapped = true;
                swap(&arr[j], &arr[j + 1], sizeof(int));
            }
        }

        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 width) {

    for (size_t i = 0; i < n - 1; i++) {
        bool swapped = false;
        for (size_t j = 0; j < n - 1; j++) {
            if (arr[j] > arr[j + 1]) { // out of order, so swap!
                swapped = true;
                swap(&arr[j], &arr[j + 1], width);
            }
        }

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

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

Recall 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 locate the **last** element:

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

How can we generalize this to get the i^{th} element?

```
void *addr = (char *)arr + i * width;
```

Generic Bubble Sort

```
void bubble_sort(void *arr, size_t n, size_t width) {

    for (size_t i = 0; i < n - 1; i++) {
        bool swapped = false;
        for (size_t j = 0; j < n - 1; j++) {
            void *first = (char *) arr + j * width;
            void *second = (char *) arr + (j + 1) * width;
            if (arr[j] > arr[j + 1]) { // out of order, so swap!
                swapped = true;
                swap(first, second, width);
            }
        }

        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 width) {

    for (size_t i = 0; i < n - 1; i++) {
        bool swapped = false;
        for (size_t j = 0; j < n - 1; j++) {
            void *first = (char *) arr + j * width;
            void *second = (char *) arr + (j + 1) * width;
            if (*first > *second) { // out of order, so swap!
                swapped = true;
                swap(first, second, width);
            }
        }

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

Wait a minute…this doesn't work! We can't dereference **void** *s OR compare any element with **>**, since they may not be numbers!

A Generics Conundrum

- We've hit a snag: There's no way to generically compare elements. They could be any type, and < isn't always the right way to compare (e.g. think C strings)
- How can we write code to compare any two elements of the same type*?*
- That's not something that a generic bubble sort knows how to do. The caller, however, should know—because they're supplying the data.

Generic Bubble Sort

```
void bubble_sort(void *arr, size_t n, size_t width) {

    for (size_t i = 0; i < n - 1; i++) {
        bool swapped = false;
        for (size_t j = 0; j < n - 1; j++) {
            void *first = (char *) arr + j * width;
            void *second = (char *) arr + (j + 1) * width;
            if (*first > *second) { // out of order, so swap!
                swapped = true;
                swap(first, second, width);
            }
        }

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

bubble_sort (inner voice): hey, you, person who invoked me. Do you know how to compare the items at these two addresses?

Generic Bubble Sort

```
void bubble_sort(void *arr, size_t n, size_t width) {

    for (size_t i = 0; i < n - 1; i++) {
        bool swapped = false;
        for (size_t j = 0; j < n - 1; j++) {
            void *first = (char *) arr + j * width;
            void *second = (char *) arr + (j + 1) * width;
            if (*first > *second) { // out of order, so swap!
                swapped = true;
                swap(first, second, width);
            }
        }

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

Caller: yeah, I know how to compare them. You don't know what data type they are, but I do. I have a function that can do the comparison for you and tell you the result.

Generic Bubble Sort

```
void bubble_sort(void *arr, size_t n, size_t width,
                 function_type cmpfn) {
    for (size_t i = 0; i < n - 1; i++) {
        bool swapped = false;
        for (size_t j = 0; j < n - 1; j++) {
            void *first = (char *) arr + j * width;
            void *second = (char *) arr + (j + 1) * width;
            if (cmpfn(first, second) > 0) { // out of order, so swap!
                swapped = true;
                swap(first, second, width);
            }
        }

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

How can we compare these elements? They can pass us this **function as a parameter**. The function's job is to tell us how two elements compare.

Generic Bubble Sort

```
void bubble_sort(void *arr, size_t n, size_t width,
                 int (*cmpfn)(const void *a, const void *b)) {
    for (size_t i = 0; i < n - 1; i++) {
        bool swapped = false;
        for (size_t j = 0; j < n - 1; j++) {
            void *first = (char *) arr + j * width;
            void *second = (char *) arr + (j + 1) * width;
            if (cmpfn(first, second) > 0) { // out of order, so swap!
                swapped = true;
                swap(first, second, width);
            }
        }

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

How can we compare these elements? They can pass us this **function as a parameter**. The function's job is to tell us how two elements compare.

Function Pointers

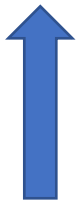
A function pointer is the type used to pass a function as a parameter. Here is how the parameter's type is declared:

```
int (*cmpfn)(const void *a, const void *b)
```

Function Pointers

A function pointer is the type used to pass a function as a parameter. Here is how the parameter's type is declared:

```
int (*cmpfn)(const void *a, const void *b)
```

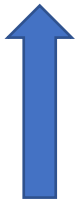


Return type
(int)

Function Pointers

A function pointer is the type used to pass a function as a parameter. Here is how the parameter's type is declared:

```
int (*cmpfn)(const void *a, const void *b)
```

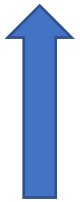


Function pointer name
(cmpfn)

Function Pointers

A function pointer is the type used to pass a function as a parameter. Here is how the parameter's type is declared:

```
int (*cmpfn)(const void *a, const void *b)
```



Function parameters
(two void *s that promise to read but not change the data)

Comparison Functions

- Function pointers are used in cases like this to compare two values of the same type. These are called **comparison functions**.
- When implementing a comparison function, it's often expected the return value provide comparison information the same way **strcmp** does.
  - < 0 if first value is "less" than the second
  - > 0 if first value is "greater" than the second
  - 0 if first value and second value are equivalent

```
int (*cmpfn)(const void *a, const void *b)
```

Function Pointers

```
int int_cmp(const void *ptr1, const void *ptr2) {
    ...
}

int main(int argc, char *argv[]) {
    int numbers[] = {4, 2, -5, 1, 12, 56};
    size_t count = sizeof(numbers)/sizeof(numbers[0]);
    bubble_sort(numbers, count, sizeof(int), int_cmp);
    return 0;
}
```

bubble_sort is generic and works for any type. But the **caller** knows the specific type of data being sorted and provides a comparison function specifically for that data type.

Function Pointers

```
int int_cmp(const void *a, const void *b) {
    return *(const int *)a - *(const int *)b;
}
```

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

Comparison Functions

- **Exercise:** how can we write a comparison function for bubble sort to sort strings in alphabetical order?
- When implementing a comparison function, it's often expected the return value provide comparison information the same way **strcmp** does.
  - < 0 if first value is "less" than the second
  - > 0 if first value is "greater" than the second
  - 0 if first value and second value are equivalent

```
int (*cmpfn)(const void *a, const void *b)
```

String Comparison Function

```
int str_cmp(const void *a, const void *b) {
    const char *str1 = *(const char **)a;
    const char *str2 = *(const char **)b;
    return strcmp(str1, str2);
}

int main(int argc, char *argv[]) {
    char *names[] = {"Nathan", "Monica", "Brent", "Sasha"};
    size_t count = sizeof(names)/sizeof(names[0]);
    bubble_sort(names, count, sizeof(char *), str_cmp);
    return 0;
}
```

Recap

- We can pass functions as parameters to pass logic throughout our programs.
- Comparison functions are often passed as parameters to generically compare two elements. There are other use cases for function pointers, and you'll see several in this week's lab and assignment.
- Functions handling generic data must use *pointers to the data they care about*, since the data could be any size. That's why function pointers are so important when implementing generics.

Plan For Today

- Finish up: Generic Stack
- Function Pointers
- Example: Bubble Sort

Next time: Floats in C