CS107, Lecture 9
C Generics – Function Pointers

Reading: K&R 5.11

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

- **Recap:** Generics with Void *
- **Finish up:** Generic Stack
- Function Pointers
- **Example:** Bubble Sort
- **Example:** Stack Cleanup

Plan For Today

- **Recap:** Generics with Void *
- **Finish up:** Generic Stack
- Function Pointers
- **Example:** Bubble Sort
- **Example:** Stack Cleanup

Generics

- We always strive to write code that is as general-purpose as possible.

- Generic code reduces code duplication, and means you can make improvements and fix bugs in one place rather than many.

- Generics is used throughout C for functions to sort any array, search any array, free arbitrary memory, and more.

Generic Swap

Wouldn't it be nice if we could write *one* function that would work with any parameter type, instead of so many different versions?

```
void swap_int(int *a, int *b) { … }
void swap_float(float *a, float *b) { … }
void swap_size_t(size_t *a, size_t *b) { … }
void swap_double(double *a, double *b) { … }
void swap_string(char **a, char **b) { … }
void swap_mystruct(mystruct *a, mystruct *b) { … }
…
```

Generic Swap

```c
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    memcpy(temp, data1ptr, nbytes);
    // copy data2 to location of data1
    memcpy(data1ptr, data2ptr, nbytes);
    // copy data in temporary storage to location of data2
    memcpy(data2ptr, temp, nbytes);
}
```

We can use **void *** to represent a pointer to any data, and **memcpy** 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 = 0xeeeeeeee;
swap(&x, &y, sizeof(short));

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

Swap Ends

Let's write a function that swaps the first and last elements in an array. How can we make this generic?

```
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));
}
```

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 <u>places</u> to arr, and 3 * sizeof(int) = 12 bytes

Short: adds 3 <u>places</u> to arr, and 3 * sizeof(short) = 6 bytes

Char *: adds 3 <u>places</u> to arr, and 3 * sizeof(char *) = 24 bytes

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

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 an array of <u>anything</u>!

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

Plan For Today

- **Recap:** Generics with Void *
- **Finish up:** Generic Stack
- Function Pointers
- **Example:** Bubble Sort
- **Example:** Stack Cleanup

Stacks

- C generics are particularly powerful in helping us create generic data structures.

- Let's see how we might go about making a Stack in C.

```
typedef struct int_node {
    struct int_node *next;
    int data;
} int_node;


typedef struct int_stack {
    int nelems;
    int_node *top;
} int_stack;
```

Stack Functions

- **`int_stack_create()`**: creates a new stack on the heap and returns a pointer to it
- **`int_stack_push(int_stack *s, int data)`**: pushes data onto the stack
- **`int_stack_pop(int_stack *s):`** pops and returns topmost stack element

int_stack_create

```
int_stack *int_stack_create() {
    int_stack *s = malloc(sizeof(int_stack));
    s->nelems = 0;
    s->top = NULL;
    return s;
}
```

Why must this be heap-allocated?

```c
void int_stack_push(int_stack *s, int data) {
    int_node *new_node = malloc(sizeof(int_node));
    new_node->data = data;

    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}
```

```c
int int_stack_pop(int_stack *s) {
    if (s->nelems == 0) {
        error(1, 0, "Cannot pop from empty stack");
    }
    int_node *n = s->top;
    int value = n->data;

    s->top = n->next;

    free(n);
    s->nelems--;

    return value;
}
```

What modifications are necessary to make a generic stack?

Generic Stack Structs

Each node can no longer store the data itself, because it could be any size! Instead, it stores a *pointer to the data* somewhere else.

```
typedef struct node {
    struct node *next;
    void *data;
} node;



typedef struct stack {
    int nelems;
    int elem_size_bytes;
    node *top;
} stack;
```

```
stack *stack_create(int elem_size_bytes) {
    stack *s = malloc(sizeof(stack));
    s->nelems = 0;
    s->top = NULL;
    s->elem_size_bytes = elem_size_bytes;
    return s;
}
```

```
void int_stack_push(int_stack *s, int data) {
    int_node *new_node = malloc(sizeof(int_node));
    new_node->data = data;

    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}
```

Problem: we can no longer pass the data itself as a parameter, because it could be any size! We also cannot copy the data into the node itself.

Generic stack_push

```
void int_stack_push(int_stack *s, int data) {
    int_node *new_node = malloc(sizeof(int_node));
    new_node->data = data;

    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}
```

Solution: pass a pointer to the data as a parameter instead, and make a heap-allocated copy of it that the node points to.

```
void stack_push(stack *s, const void *data) {
    node *new_node = malloc(sizeof(node));
    new_node->data = malloc(s->elem_size_bytes);
    memcpy(new_node->data, data, s->elem_size_bytes);

    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}
```

Solution: pass a pointer to the data as a parameter instead, and make a heap-allocated copy of it that the node points to.

Generic stack_push

```
void stack_push(stack *s, const void *data) {
    node *new_node = malloc(sizeof(node));
    new_node->data = data;

    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}
```

Why can't we do this?

```
void stack_push(stack *s, const void *data) {
    node *new_node = malloc(sizeof(node));
    new_node->data = data;

    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}
```

Why can't we do this? **Because we don't know where data points to.** It could point to stack memory that goes away in the future! This stack must have its own copy to control its lifetime.

```c
int int_stack_pop(int_stack *s) {
    if (s->nelems == 0) {
        error(1, 0, "Cannot pop from empty stack");
    }
    int_node *n = s->top;
    int value = n->data;

    s->top = n->next;

    free(n);
    s->nelems--;

    return value;
}
```

Problem: we can no longer return the data itself, because it could be any size!

```c
int int_stack_pop(int_stack *s) {
    if (s->nelems == 0) {
        error(1, 0, "Cannot pop from empty stack");
    }
    int_node *n = s->top;
    int value = n->data;

    s->top = n->next;

    free(n);
    s->nelems--;

    return value;
}
```

Solution: have the caller pass a memory location as a parameter, and copy the data value to that location.

```
void stack_pop(stack *s, void *addr) {
    if (s->nelems == 0) {
        error(1, 0, "Cannot pop from empty stack");
    }
    node *n = s->top;
    memcpy(addr, n->data, s->elem_size_bytes);
    s->top = n->next;

    free(n->data);
    free(n);
    s->nelems--;
}
```

Solution: have the caller pass a memory location as a parameter, and copy the data value to that location.

```
int_stack *intstack = int_stack_create();
for (int i = 0; i < TEST_STACK_SIZE; i++) {
    int_stack_push(intstack, i);
}
```

We must now pass the *address* of an element to push onto the stack, rather than the element itself.

Using Generic Stack

```
stack *intstack = stack_create(sizeof(int));
for (int i = 0; i < TEST_STACK_SIZE; i++) {
    stack_push(intstack, &i);
}
```

We must now pass the *address* of an element to push onto the stack, rather than the element itself.

```
int_stack *intstack = int_stack_create();
int_stack_push(intstack, 7);
```

We must now pass the *address* of an element to push onto the stack, rather than the element itself.

```
stack *intstack = stack_create(sizeof(int));
int num = 7;
stack_push(intstack, &num);
```

We must now pass the *address* of an element to push onto the stack, rather than the element itself.

Using Generic Stack

```
// Pop off all elements
while (intstack->nelems > 0) {
    printf("%d\n", int_stack_pop(intstack));
}
```

We must now pass the *address* of where we would like to store the popped element, rather than getting it directly as a return value.

```
// Pop off all elements
int popped_int;
while (intstack->nelems > 0) {
    int_stack_pop(intstack, &popped_int);
    printf("%d\n", popped_int);
}
```

We must now pass the *address* of where we would like to store the popped element, rather than getting it directly as a return value.

Plan For Today

- **Recap:** Generics with Void *
- **Finish up:** Generic Stack
- Function Pointers
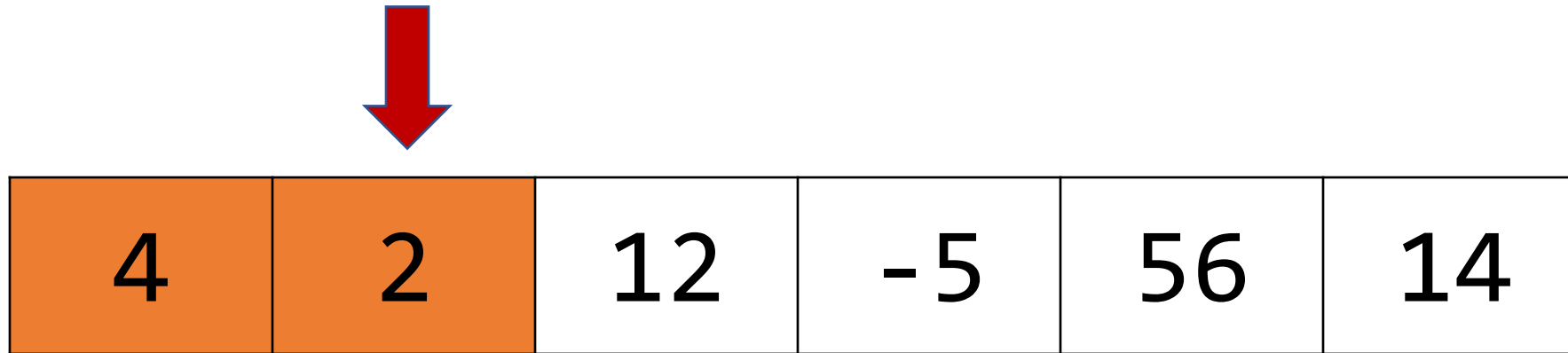- **Example:** Bubble Sort
- **Example:** Stack Cleanup

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 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!
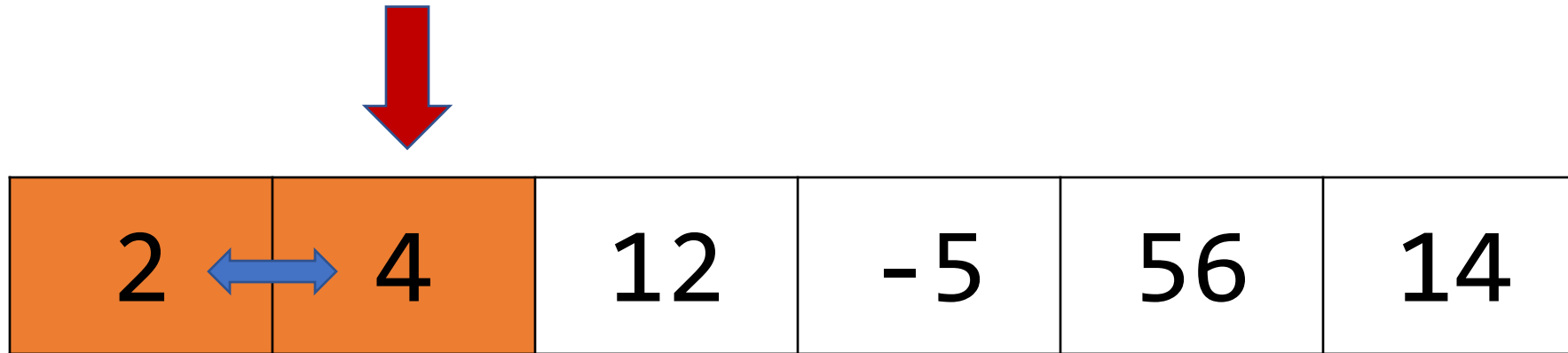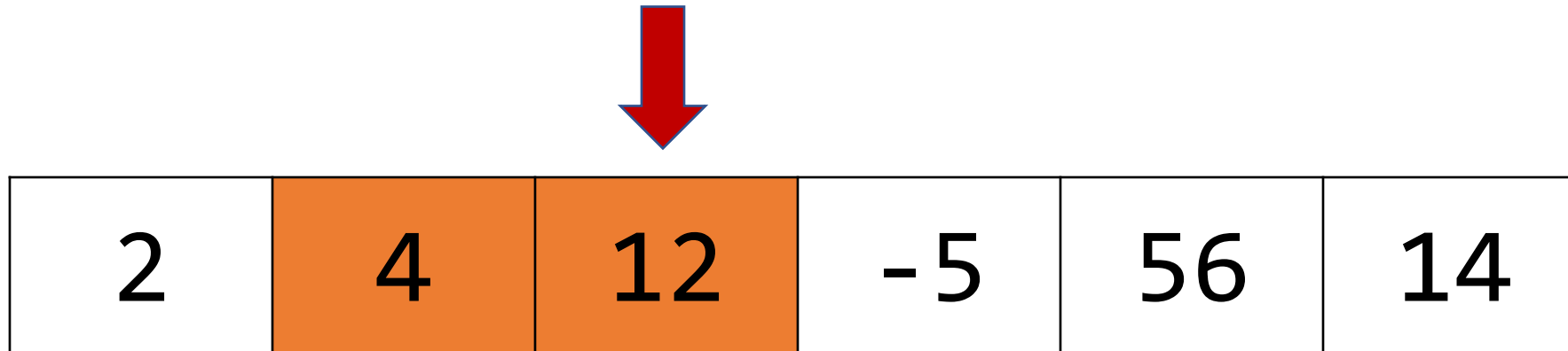
Bubble Sort

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

| 2 | 4 | 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!
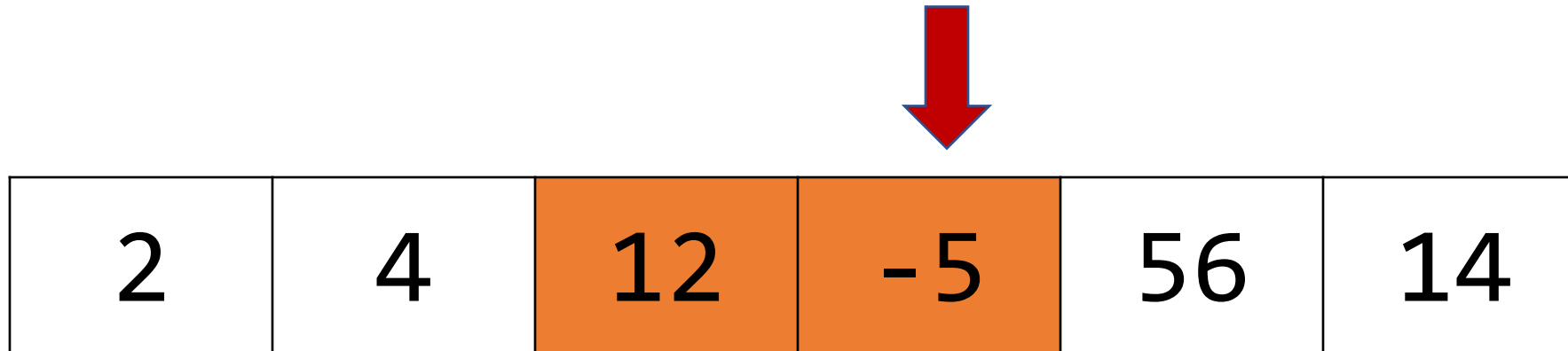
Bubble Sort

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

2	4	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!
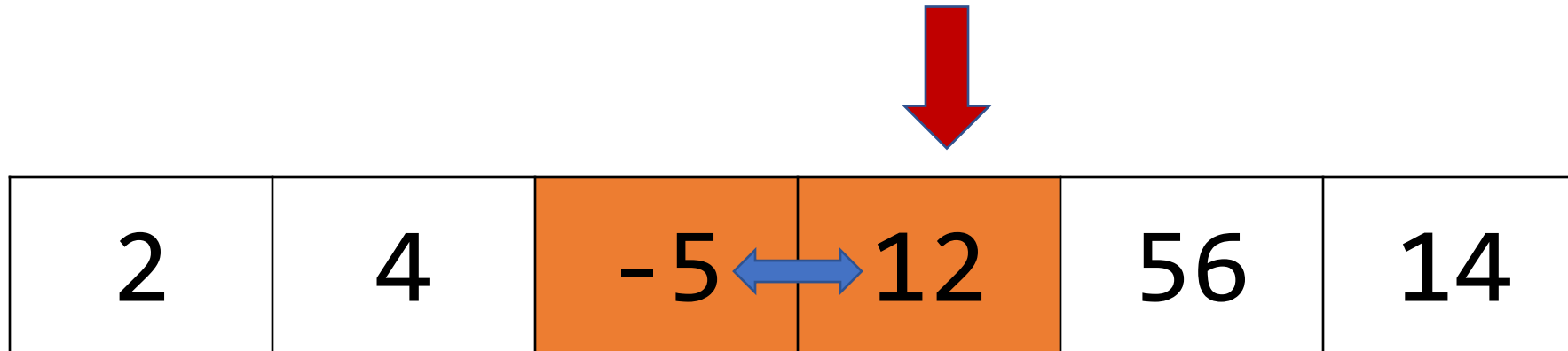
Bubble Sort

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

2	4	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!
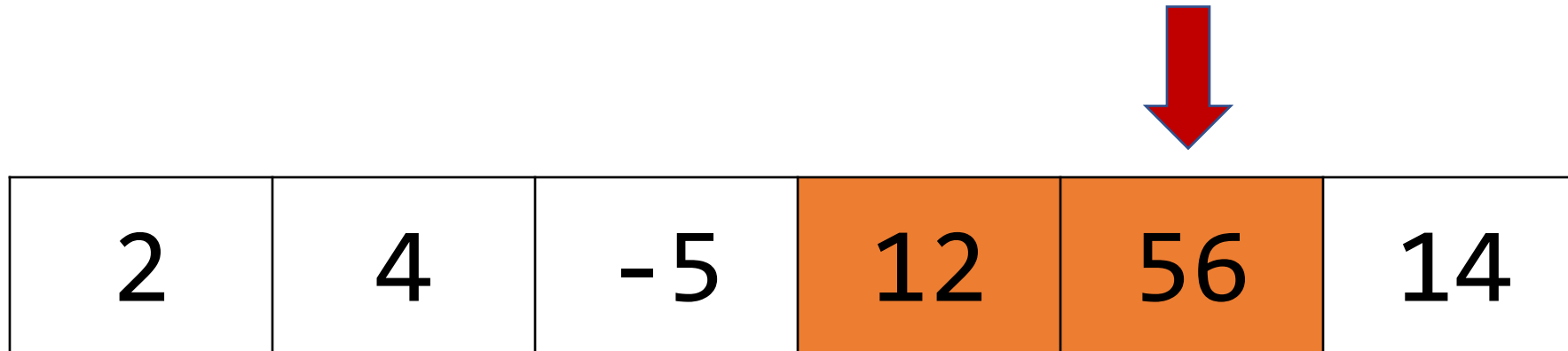
Bubble Sort

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

2	4	-5	12	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!
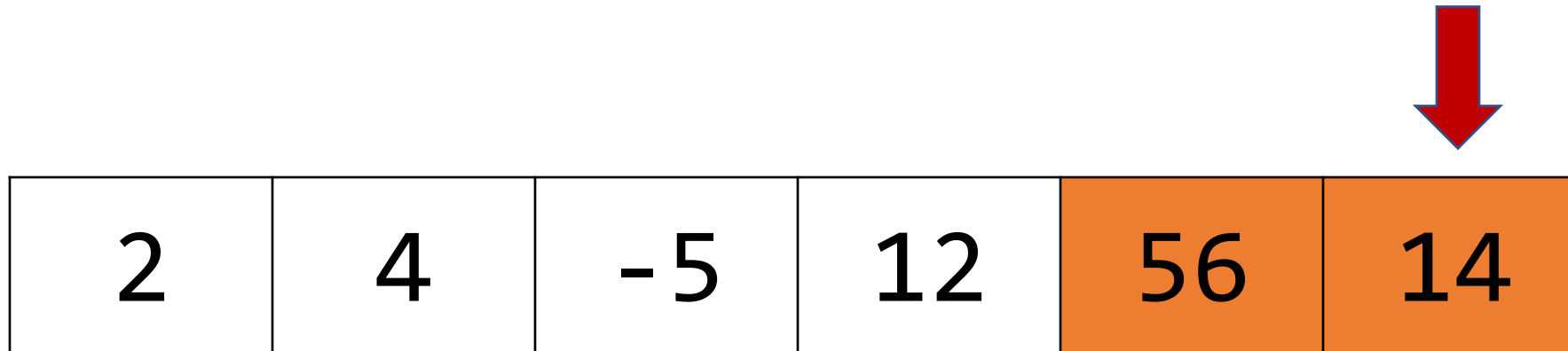
Bubble Sort

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

| 2 | 4 | -5 | 12 | 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 to sort a list of integers. We'll use the **bubble sort algorithm**.

2	4	-5	12	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!
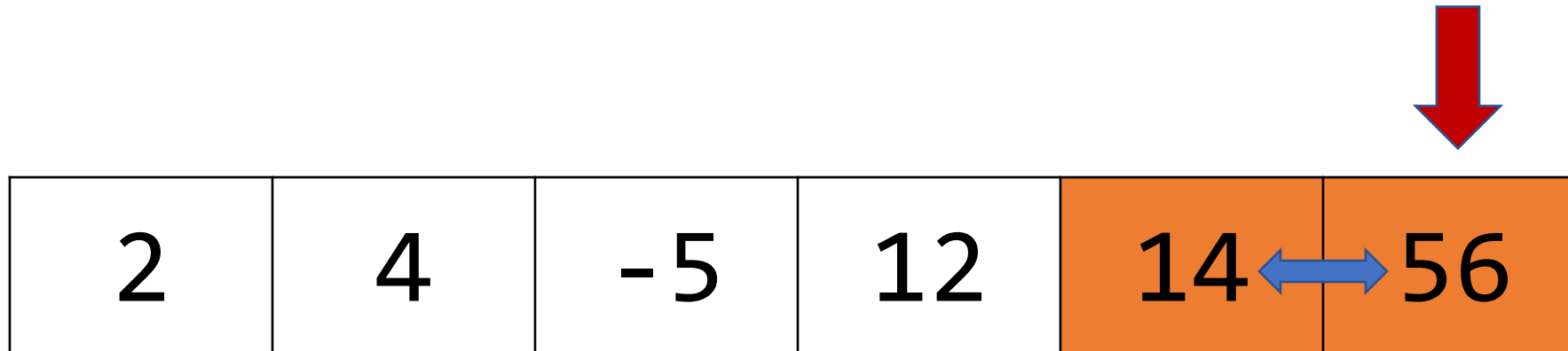
Bubble Sort

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

2	4	-5	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!
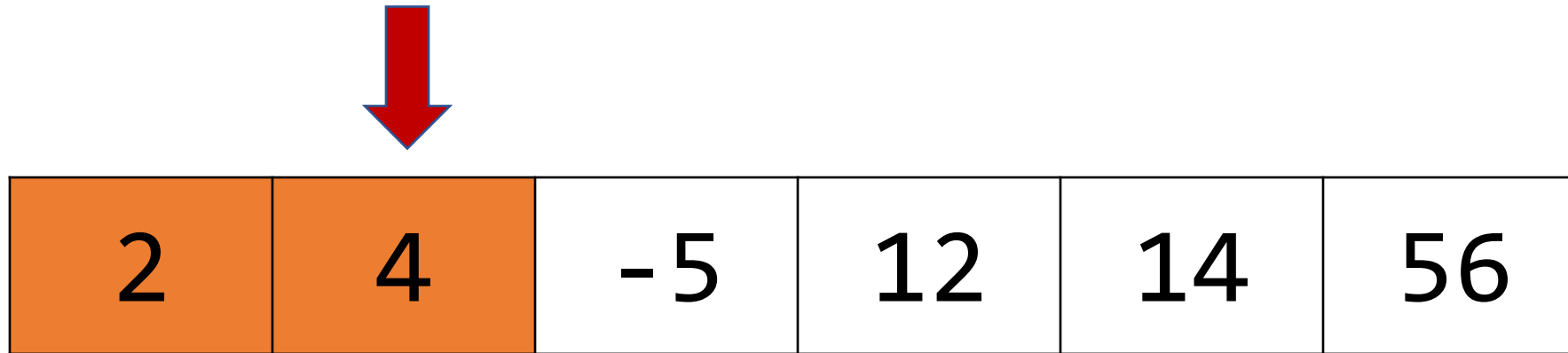
Bubble Sort

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

2	4	-5	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!
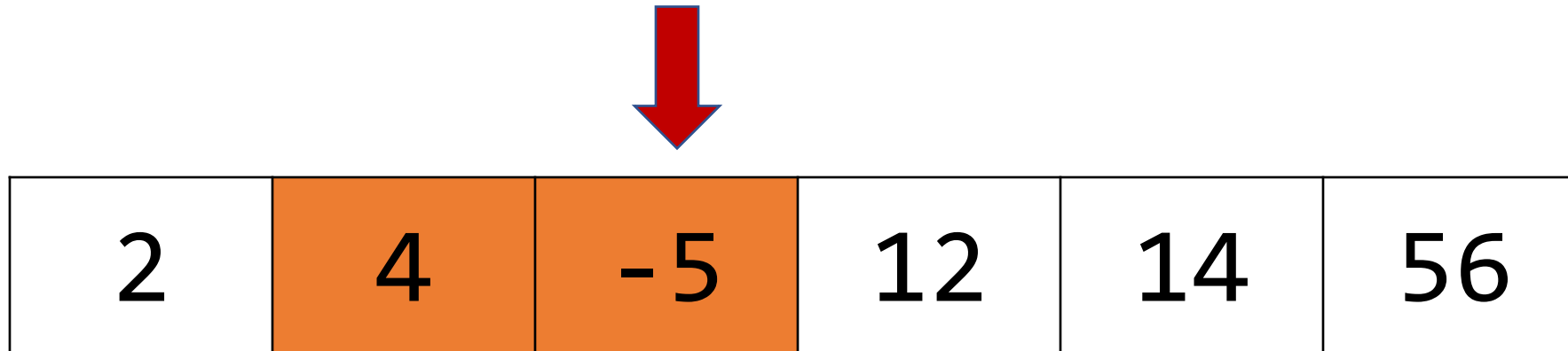
Bubble Sort

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

2	4	-5	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!
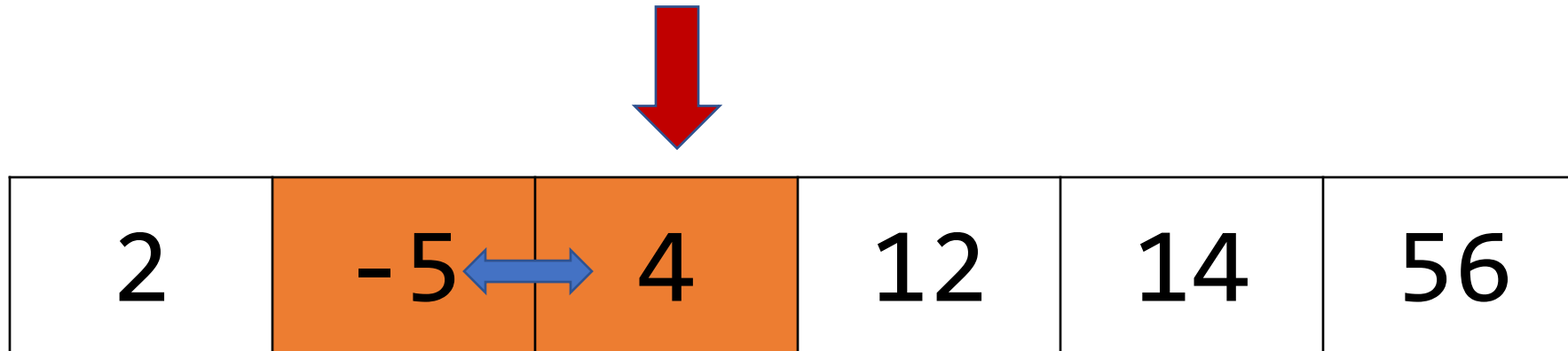
Bubble Sort

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

| 2 | -5 | 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!
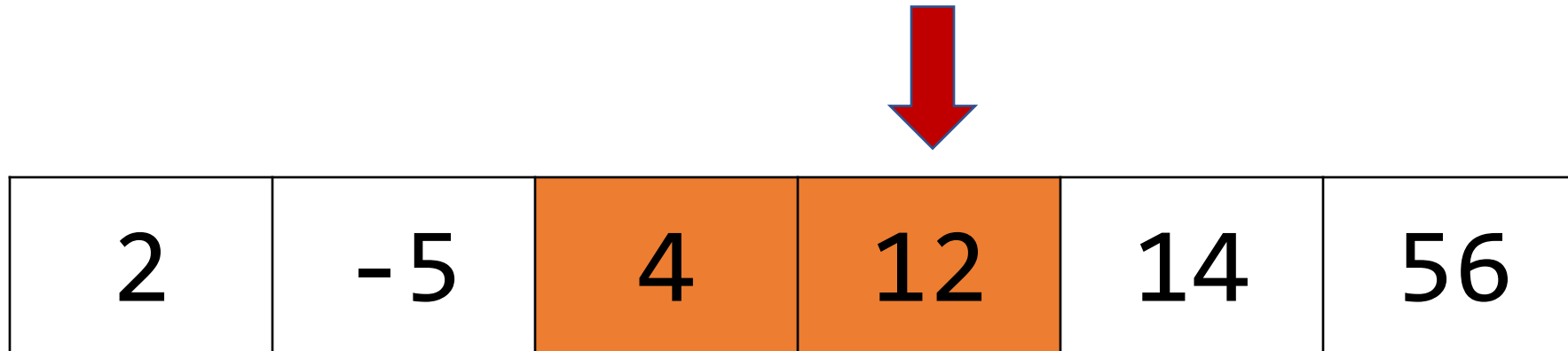
Bubble Sort

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

2	-5	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!

Bubble Sort

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

2	-5	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!
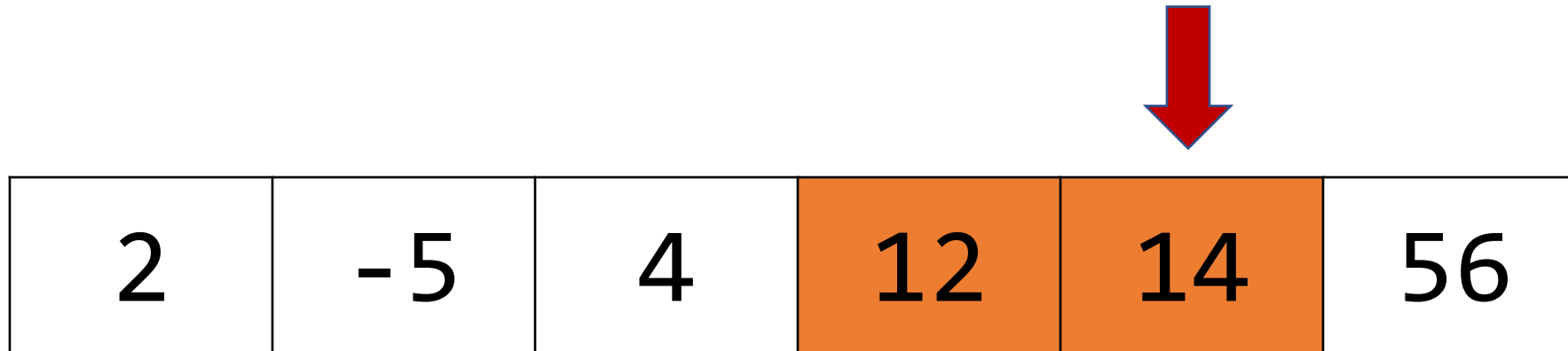
Bubble Sort

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

2	-5	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!
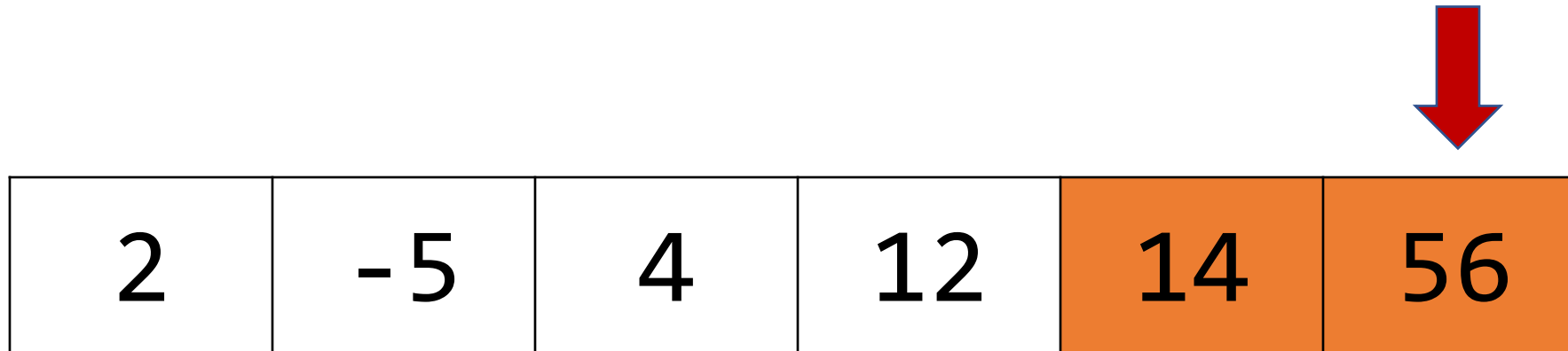
Bubble Sort

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

| 2 | -5 | 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!
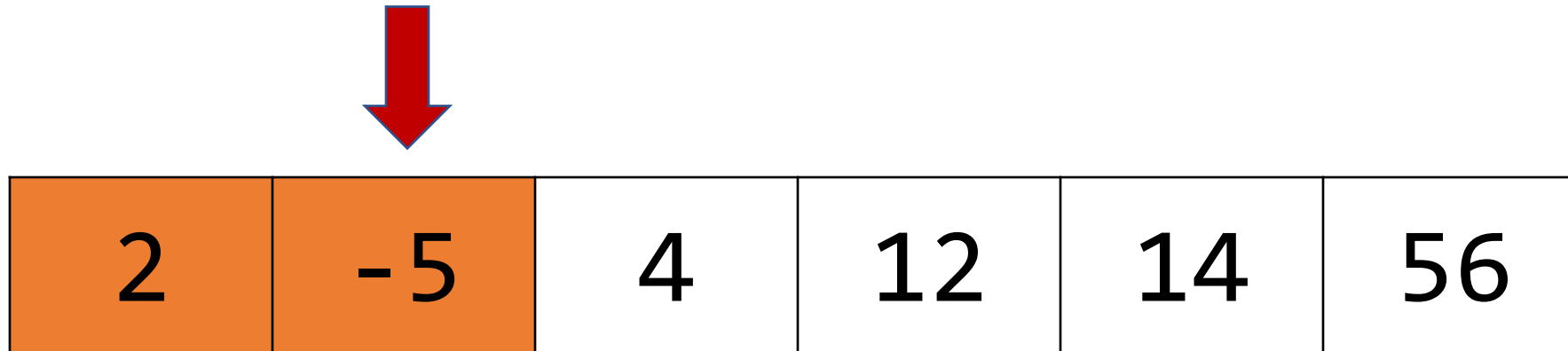
Bubble Sort

- 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 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!
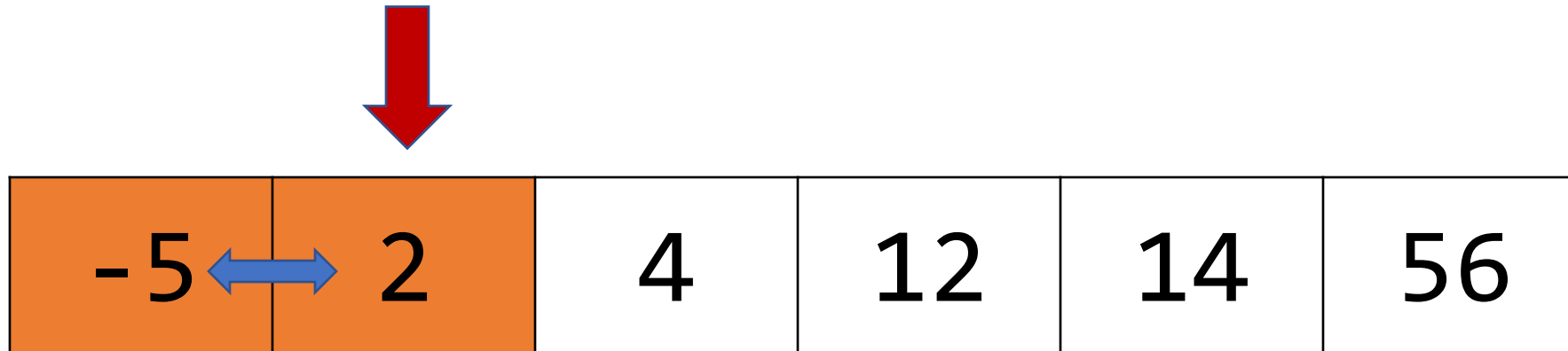
Bubble Sort

- Let's write a function to sort a list of integers. We'll use 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!

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 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!
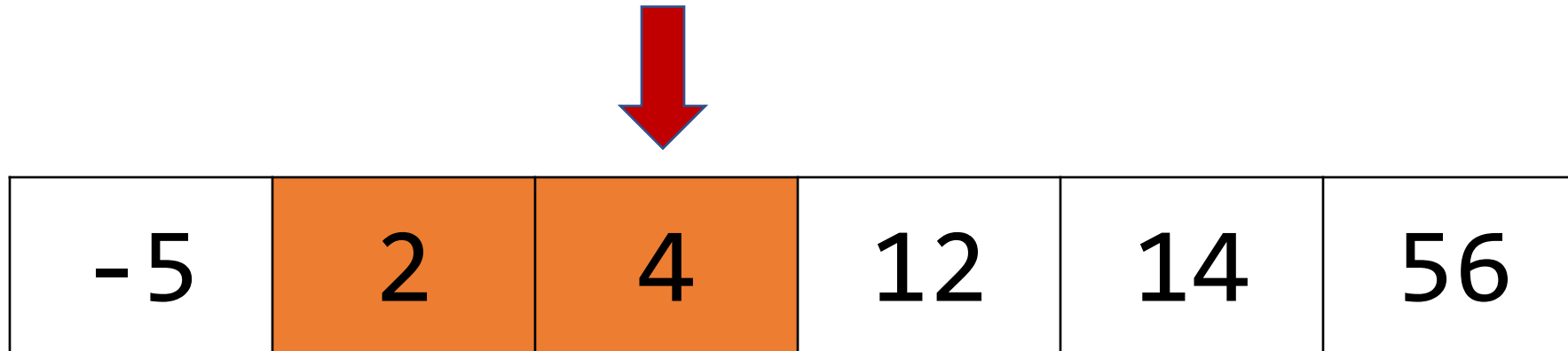
Bubble Sort

- Let's write a function to sort a list of integers. We'll use 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!
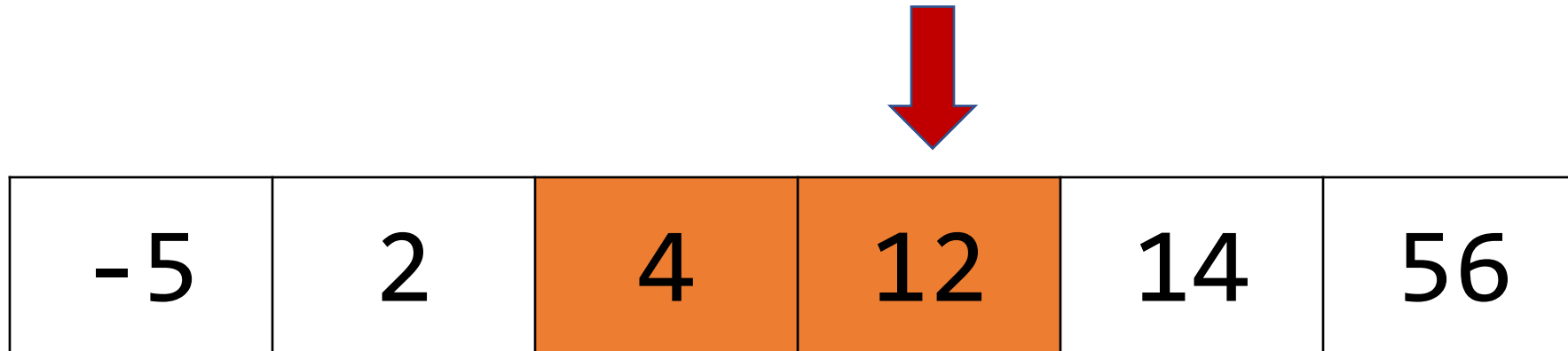
Bubble Sort

- 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 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!
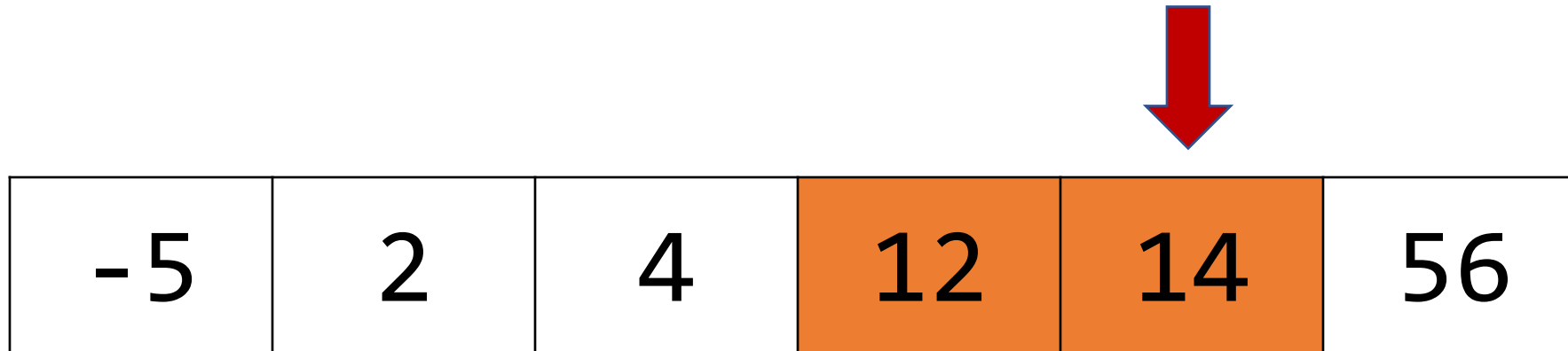
Bubble Sort

- Let's write a function to sort a list of integers. We'll use 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!
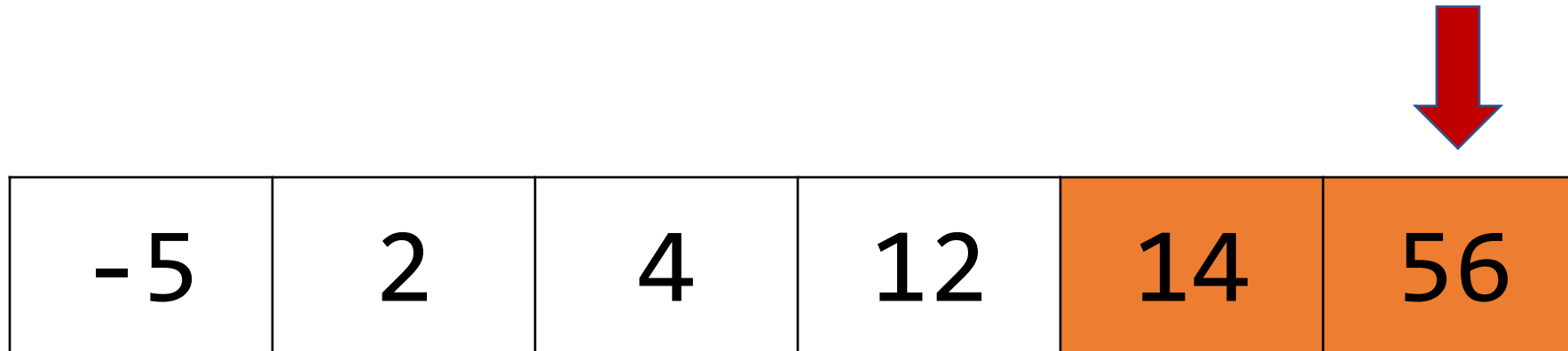
Bubble Sort

- 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 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 to sort a list of integers. We'll use 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!
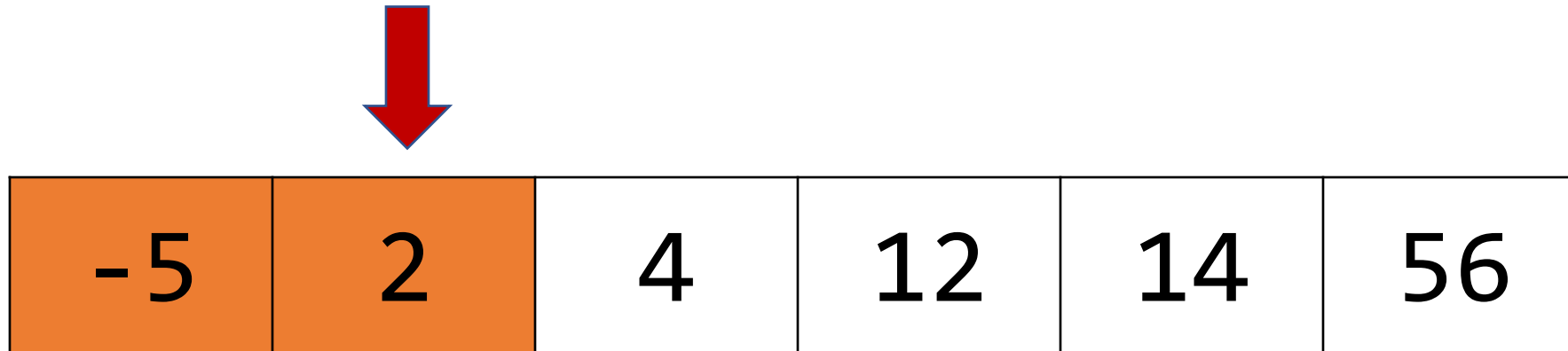
Bubble Sort

- 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 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!
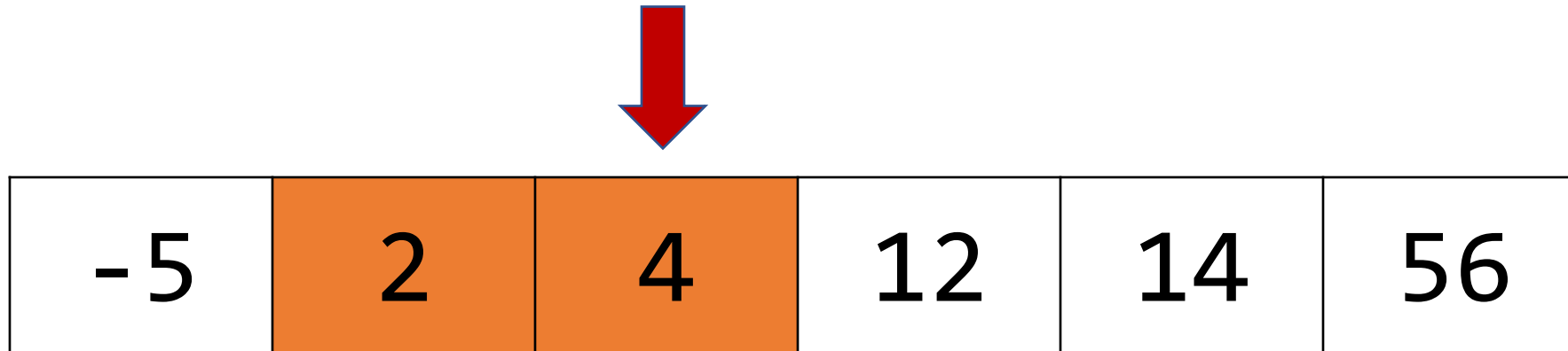
Bubble Sort

- Let's write a function to sort a list of integers. We'll use 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!

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

        if (!swapped) {
            return;
        }
    }
}
```
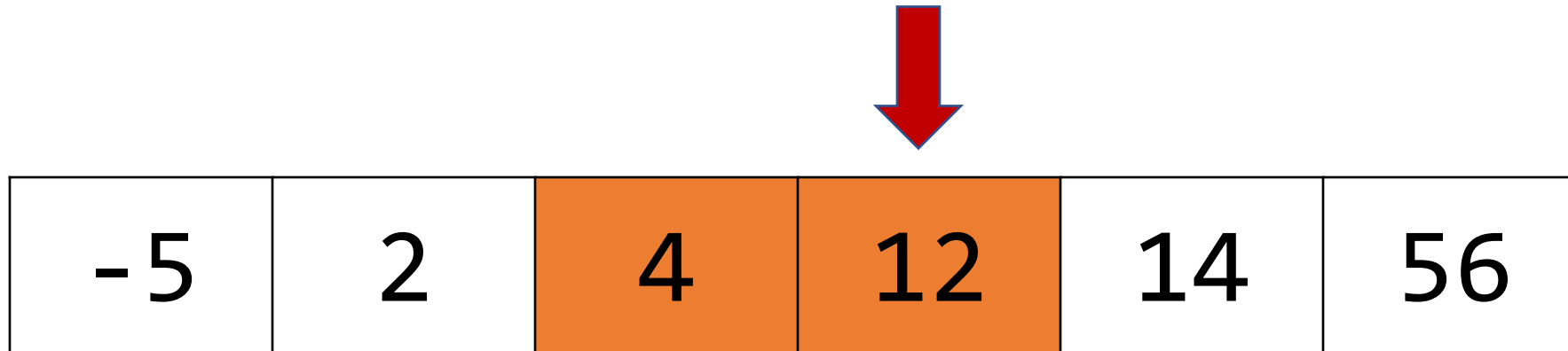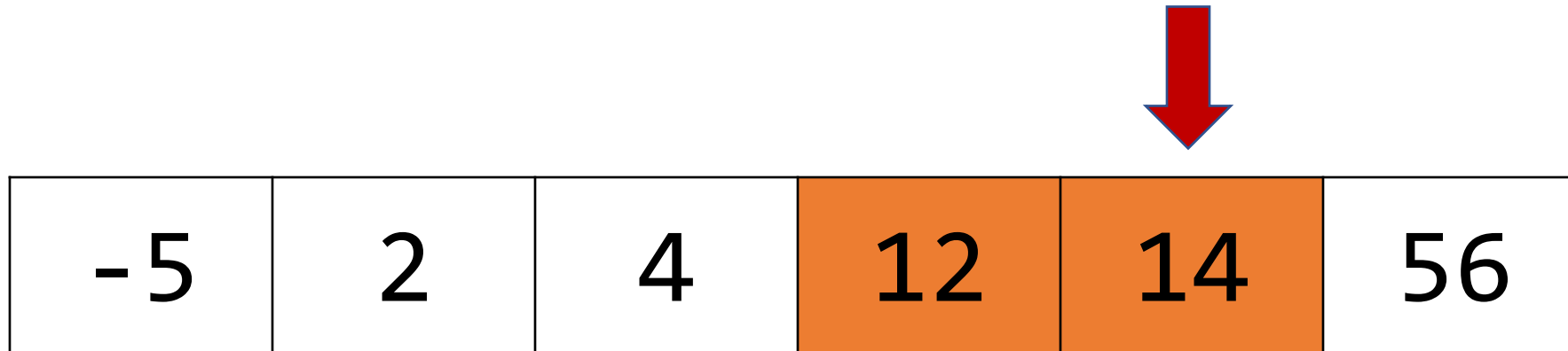
How can we make this function generic, to sort an array of *any type?*

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

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

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

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

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

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

Generic Bubble Sort

```c
void bubble_sort_int(void *arr, int n, int elem_size_bytes) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            if (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_int(void *arr, int n, int elem_size_bytes) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            void *prev_elem = (char *)arr + (i-1)*elem_size_bytes;
            void *curr_elem = (char *)arr + i*elem_size_bytes;
            if (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.

```
void bubble_sort_int(void *arr, int n, int elem_size_bytes) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            void *prev_elem = (char *)arr + (i-1)*elem_size_bytes;
            void *curr_elem = (char *)arr + i*elem_size_bytes;
            if (*prev_elem > *curr_elem) {
                swapped = true;
                swap(prev_elem, curr_elem, elem_size_bytes);
            }
        }

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

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

```
void bubble_sort_int(void *arr, int n, int elem_size_bytes) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            void *prev_elem = (char *)arr + (i-1)*elem_size_bytes;
            void *curr_elem = (char *)arr + i*elem_size_bytes;
            if (*prev_elem > *curr_elem) {
                swapped = true;
                swap(prev_elem, curr_elem, elem_size_bytes);
            }
        }

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

```
void bubble_sort_int(void *arr, int n, int elem_size_bytes) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            void *prev_elem = (char *)arr + (i-1)*elem_size_bytes;
            void *curr_elem = (char *)arr + i*elem_size_bytes;
            if (*prev_elem > *curr_elem) {
                swapped = true;
                swap(prev_elem, curr_elem, elem_size_bytes);
            }
        }

        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 wall – there is no way to generically compare elements. They could be any type, and have complex ways to compare them.

- How can we write code to compare *any two elements of the same type?*

- That's not something that bubble sort can ever know how to do. **BUT** – our caller should know how to do this, because they're passing in the data….let's ask them!

```
void bubble_sort_int(void *arr, int n, int elem_size_bytes) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            void *prev_elem = (char *)arr + (i-1)*elem_size_bytes;
            void *curr_elem = (char *)arr + i*elem_size_bytes;
            if (*prev_elem > *curr_elem) {
                swapped = true;
                swap(prev_elem, curr_elem, elem_size_bytes);
            }
        }

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

Us: hey, you, person who called us. Do you know how to compare these two elements? Can you help us?

Generic Bubble Sort

```
void bubble_sort_int(void *arr, int n, int elem_size_bytes) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            void *prev_elem = (char *)arr + (i-1)*elem_size_bytes;
            void *curr_elem = (char *)arr + i*elem_size_bytes;
            if (*prev_elem > *curr_elem) {
                swapped = true;
                swap(prev_elem, curr_elem, elem_size_bytes);
            }
        }

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

> **Caller:** yeah, I know how to compare those. I have a function that can do the comparison for you and tell you the result.

```
void bubble_sort_int(void *arr, int n, int elem_size_bytes) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            void *prev_elem = (char *)arr + (i-1)*elem_size_bytes;
            void *curr_elem = (char *)arr + i*elem_size_bytes;
            if (compare_fn(prev_elem, curr_elem)) {
                swapped = true;
                swap(prev_elem, curr_elem, elem_size_bytes);
            }
        }

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

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

74

```
void bubble_sort_int(void *arr, int n, int elem_size_bytes,
                     bool (*compare_fn)(void *a, void *b)) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            void *prev_elem = (char *)arr + (i-1)*elem_size_bytes;
            void *curr_elem = (char *)arr + i*elem_size_bytes;
            if (compare_fn(prev_elem, curr_elem)) {
                swapped = true;
                swap(prev_elem, curr_elem, elem_size_bytes);
            }
        }

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

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

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.

```
bool (*compare_fn)(void *a, void *b)
```

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

bool (*compare_fn)(void *a, void *b)

Return type
(bool)

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

```
bool (*compare_fn)(void *a, void *b)
```

Function pointer name
(compare_fn)

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

```
bool (*compare_fn)(void *a, void *b)
```

Function parameters
(two void *s)

```
void bubble_sort_int(void *arr, int n, int elem_size_bytes,
                     bool (*compare_fn)(void *a, void *b)) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            void *prev_elem = (char *)arr + (i-1)*elem_size_bytes;
            void *curr_elem = (char *)arr + i*elem_size_bytes;
            if (compare_fn(prev_elem, curr_elem)) {
                swapped = true;
                swap(prev_elem, curr_elem, elem_size_bytes);
            }
        }

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

Function Pointers

```
bool integer_compare(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]), integer_compare);
    return 0;
}
```

Function Pointers

- Function pointers must always take *pointers to the data they care about,* since the data could be any size!

When writing a callback, 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)

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

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

        // perform operation
        return num1 > num2;
}
```

```
bool integer_compare(void *ptr1, void *ptr2) {
        // cast arguments to int *s and dereference
        int num1 = *(int *)ptr1;
        int num2 = *(int *)ptr2;

        // perform operation
        return num1 > num2;
}
```

Comparison Functions

- Function pointers are used often in cases like this to compare two values of the same type.
- 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 *a, void *b)
```

Function Pointers

```c
int integer_compare(void *ptr1, void *ptr2) {
    // cast arguments to int *s and dereference
    int num1 = *(int *)ptr1;
    int num2 = *(int *)ptr2;

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

```
void bubble_sort_int(void *arr, int n, int elem_size_bytes,
                     int (*compare_fn)(void *a, void *b)) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            void *prev_elem = (char *)arr + (i-1)*elem_size_bytes;
            void *curr_elem = (char *)arr + i*elem_size_bytes;
            if (compare_fn(prev_elem, curr_elem) > 0) {
                swapped = true;
                swap(prev_elem, curr_elem, elem_size_bytes);
            }
        }

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

Comparison Functions

- **Exercise:** how can we write a comparison function for bubble sort to sort strings in alphabetical order?

- The common prototype 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

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

String Comparison Function

```c
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, it will accept it as long as it fits the specified signature.

- *This is dangerous*! E.g. what happens if you pass in a string comparison function when sorting an integer array?

Plan For Today

- **Recap:** Generics with Void *
- **Finish up:** Generic Stack
- Function Pointers
- **Example:** Bubble Sort
- **Example:** Stack Cleanup

Function Pointers

- Function pointers can be used in a variety of ways. For instance, you could have:
  - A function to compare two elements of a given type
  - A function to print out an element of a given type
  - A function to free memory associated with a given type
  - And more…

Function Pointers

- Function pointers can be used in a variety of ways. For instance, you could have:
  - A function to compare two elements of a given type
  - A function to print out an element of a given type
  - **A function to free memory associated with a given type**
  - And more…

Demo: Revisiting Our Generic Stack

Common Utility Callback Functions

- Comparison function – compares two elements of a given type.

```
int (*cmp_fn)(const void *addr1, const void *addr2)
```

- Cleanup function – cleans up heap memory associated with a given type.

```
void (*cleanup_fn)(void *addr)
```

- There are many more! You can specify any functions you would like passed in when writing your own generic functions.

Generics Overview

- We use **void *** pointers and memory operations like **memcpy** and **memmove** to make data operations generic.

- We use **function pointers** to make logic/functionality operations generic.

Plan For Today

- **Recap:** Generics with Void *
- **Finish up:** Generic Stack
- Function Pointers
- **Example:** Bubble Sort
- **Example:** Stack Cleanup

Next time: Floats in C