# CS107, Lecture 9
## C Generics – Function Pointers

Reading: K&R 5.11

# Generic Lecture Plan

- Generic Swap
- **Generics Pitfalls**
- Generic Swap Ends
- Generic Stack
- Array Rotation (see Lecture 8 slides!)
- Generic Bubble Sort
- Function Pointers + Comparison Functions

# 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! (*How? Let's find out!)*

# 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!

http://i.ytimg.com/vi/10gPoYjq3EA/hqdefault.jpg

# Generic Lecture Plan

- Generic Swap
- Generics Pitfalls
- **Generic Swap Ends**
- Generic Stack
- Array Rotation (see Lecture 8 slides!)
- Generic Bubble Sort
- Function Pointers + Comparison Functions

# 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;
}

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;
}
```

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

# 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));
}

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;
}
```

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

# 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 into/beyond arr is this?

If it's an array of…

**Int?**

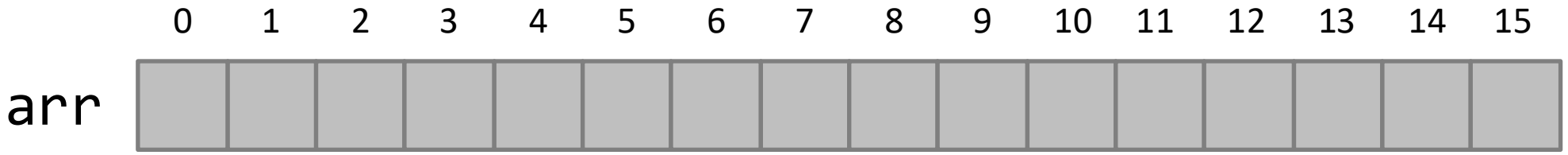
# Pointer Arithmetic

```
arr + nelems - 1
```

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

If it's an array of…

**Int?**



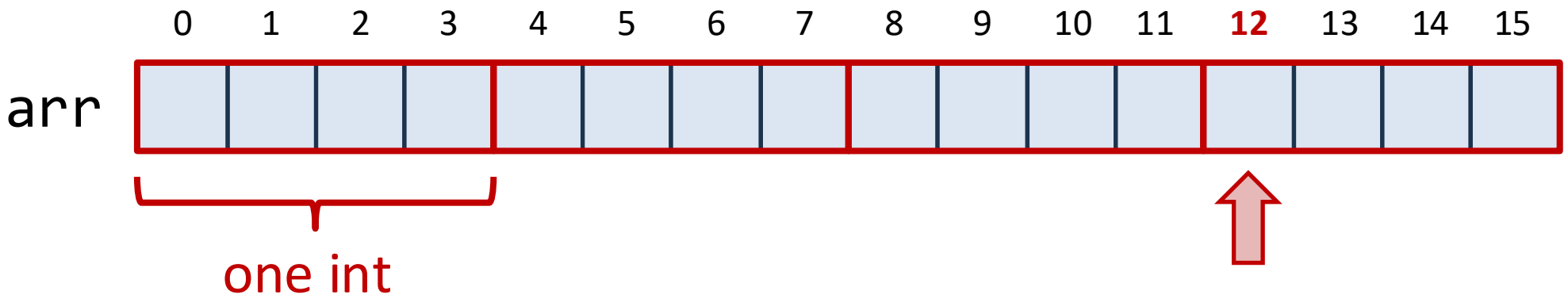
# Pointer Arithmetic

```
arr + nelems - 1
```

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

If it's an array of…

**Int?** adds 3 places to arr, and 3 * sizeof(int) = 12 bytes



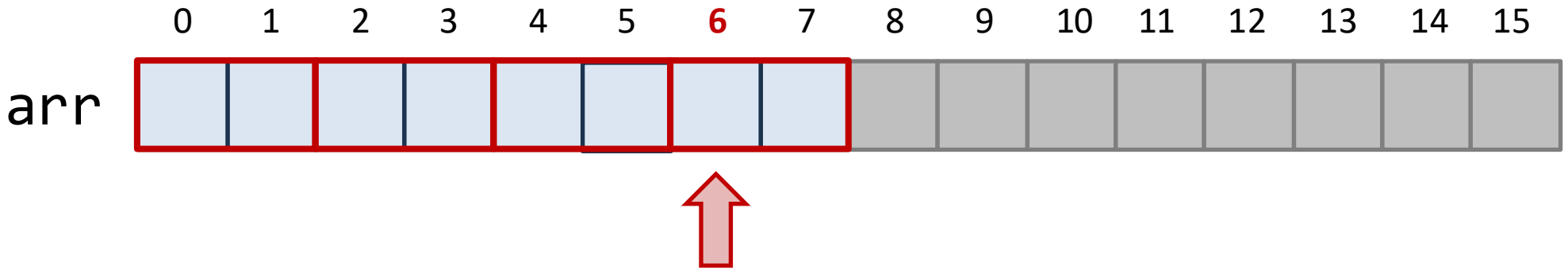
# Pointer Arithmetic

```
arr + nelems - 1
```

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

If it's an array of…

**Int?** adds 3 places to arr, and 3 * sizeof(int) = 12 bytes
**Short?** adds 3 places to arr, and 3 * sizeof(short) = 6 bytes



# Pointer Arithmetic

```
arr + nelems - 1
```

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

If it's an array of…

**Int?** adds 3 places to arr, and 3 * sizeof(int) = 12 bytes
**Short?** adds 3 places to arr, and 3 * sizeof(short) = 6 bytes
**Char *?:** adds 3 places to arr, and 3 * sizeof(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

Well, now it can swap_ends 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);
}
```

We can do pointer arithmetic with a **void *** pointer by casting it.

# Swap Ends

Lets see some examples!

```
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

Lets see some examples!

```
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

Lets see some examples!

```
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

Lets see some examples!

```
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]));
```

# Generic Lecture Plan

- Generic Swap
- Generics Pitfalls
- Generic Swap Ends
- **Generic Stack**
- Array Rotation (see Lecture 8 slides!)
- Generic Bubble Sort
- Function Pointers + Comparison Functions

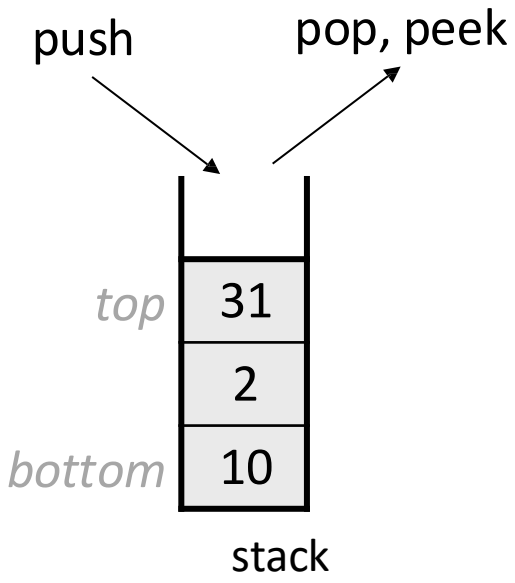
# Stacks

- 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.

# Refresher: Stacks

- A **Stack** is a data structure representing a stack of things.
- Objects can be ***pushed*** on top of or ***popped*** from the top of the stack.
- Only the top of the stack can be accessed; no other objects in the stack are visible.
- Main operations:
  - **`push(value)`**: add an element to the top of the stack
  - **`pop()`**: remove and return the top element in the stack
  - **`peek()`**: return (but do not remove) the top element in the stack

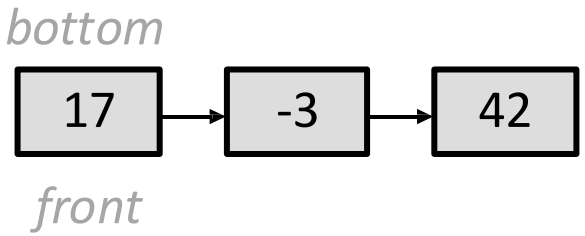


# Refresher: Stacks

A stack is often implemented using a **linked list** internally.

- "bottom" = tail of linked list
- "top" = head of linked list

```
Stack<int> s;
s.push(42);
s.push(-3);
s.push(17);
```




**Problem:** C is not object-oriented! We can't call methods on variables.

# What modifications are necessary to make a generic stack?

# Stack Structs

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


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

How might we modify the Stack data representation itself to be generic?

# Stack Structs

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


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

**Problem:** each node can no longer store the data itself, because it could be any size!

# Generic Stack Structs

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


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

**Solution:** each node stores a pointer, which is always 8 bytes, to the data somewhere else. We must also store the data size, where should that go?

# Generic Stack Structs

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


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

**Solution:** each node stores a pointer, which is always 8 bytes, to the data somewhere else. We must also store the data size in the Stack struct.

# 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;
}
```

How might we modify this function to be generic?

**From previous slide:**

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

# Generic stack_create

```
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;
}
```

**From previous slide:**

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

# int_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++;
}
```

How might we modify this function to be generic?

**From previous slide:**

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

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

# 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++;
}
```

**Problem 1:** we can no longer pass the data itself as a parameter, because it could be any size!

# Generic stack_push

```
void int_stack_push(int_stack *s, const void *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 1:** pass a pointer to the data as a parameter instead.

# Generic stack_push

```
void int_stack_push(int_stack *s, const void *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 2:** we cannot copy the existing data pointer into new_node. The data structure must manage its own copy that exists for its entire lifetime. The provided copy may go away!

# Generic stack_push

```
int main() {
    stack *int_stack = stack_create(sizeof(int));
    add_one(int_stack);
    // now stack stores pointer to invalid memory for 7!
}

void add_one(stack *s) {
    int num = 7;
    stack_push(s, &num);
}
```

# Generic stack_push

```
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 2:** make a heap-allocated copy of the data that the node points to.

# int_stack_pop

```
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;
}
```

How might we modify this function to be generic?

**From previous slide:**

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

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

# Generic stack_pop

```
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!

# Generic stack_pop

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

    s->top = n->next;

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

    return value;
}
```

While it's possible to return the heap address of the element, this means the client would be responsible for freeing it. Ideally, the data structure should manage its own memory here.

# Generic stack_pop

```
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 to that location.

# Using Generic Stack

```
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.

# Using Generic Stack

```
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.

# Using Generic Stack

```
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.

# Using Generic Stack

```
// Pop off all elements
int popped_int;
while (intstack->nelems > 0) {
    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.

# 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 Lecture Plan

- Generic Swap
- Generics Pitfalls
- Generic Swap Ends
- Generic Stack
- Array Rotation (see Lecture 8 slides!)
- **Generic Bubble Sort**
- Function Pointers + Comparison Functions

# Bubble Sort

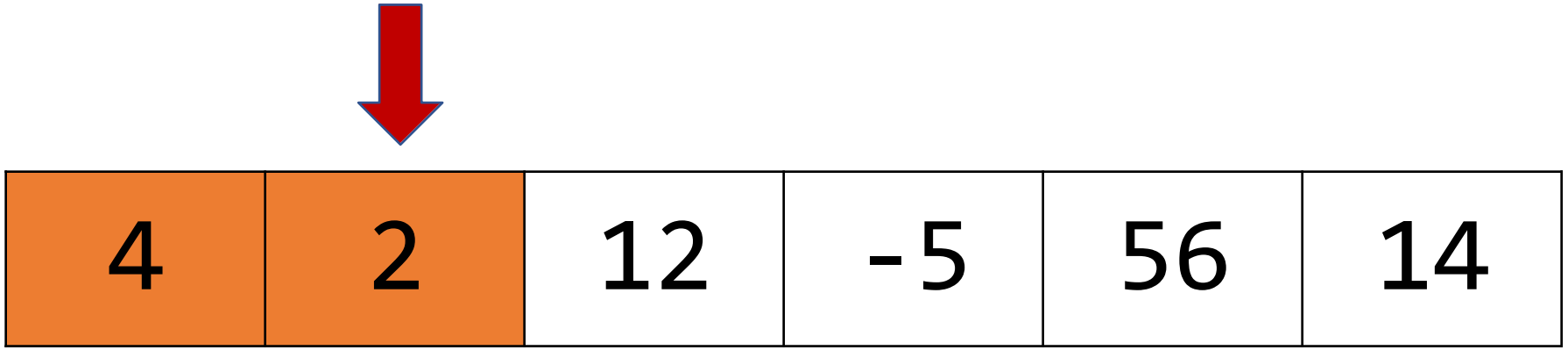
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.

| 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

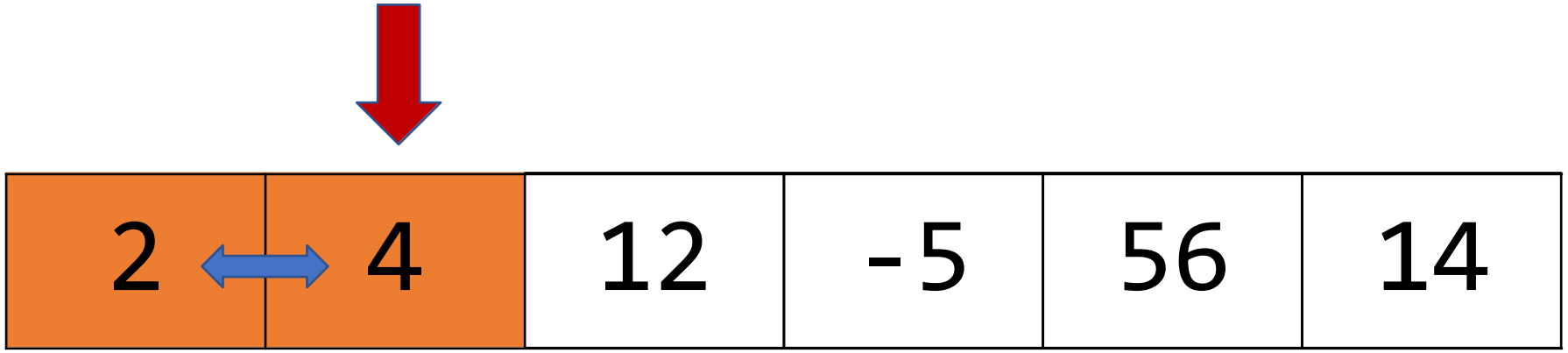
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.




- 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

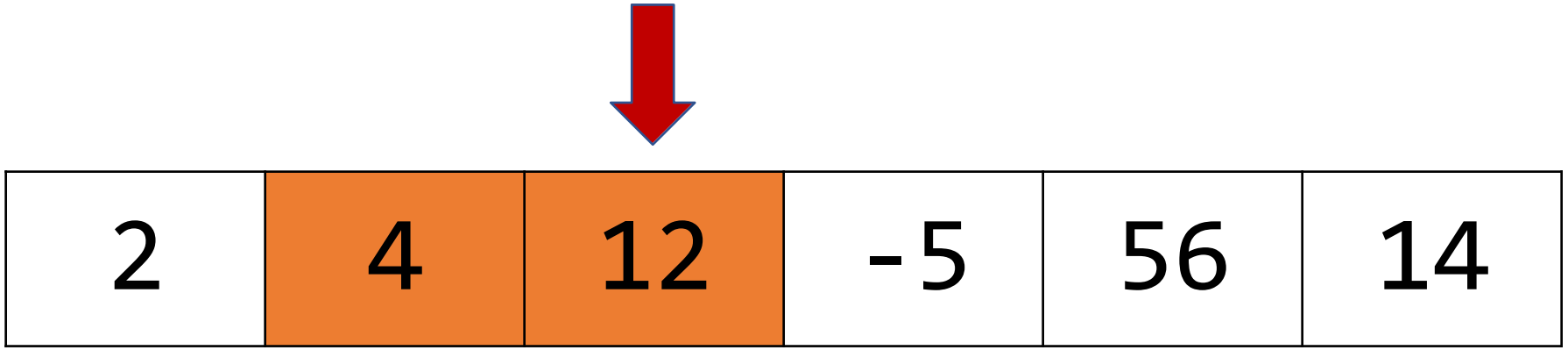
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.




- 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

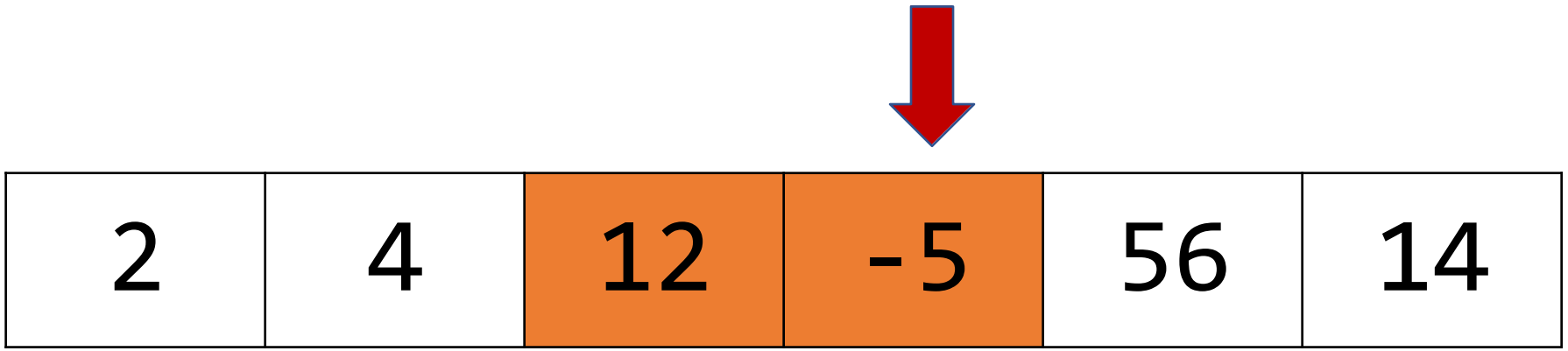
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.




- 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

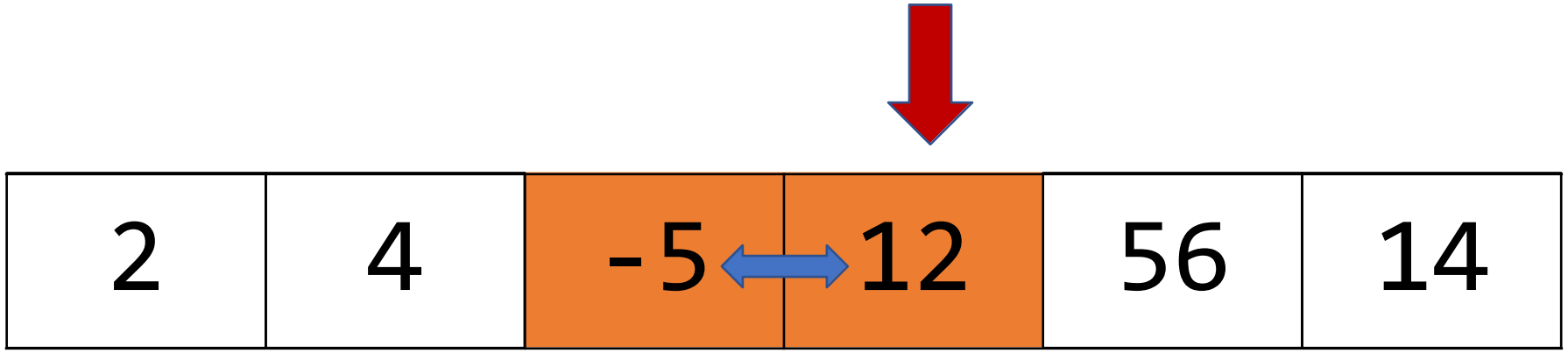
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.




- 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

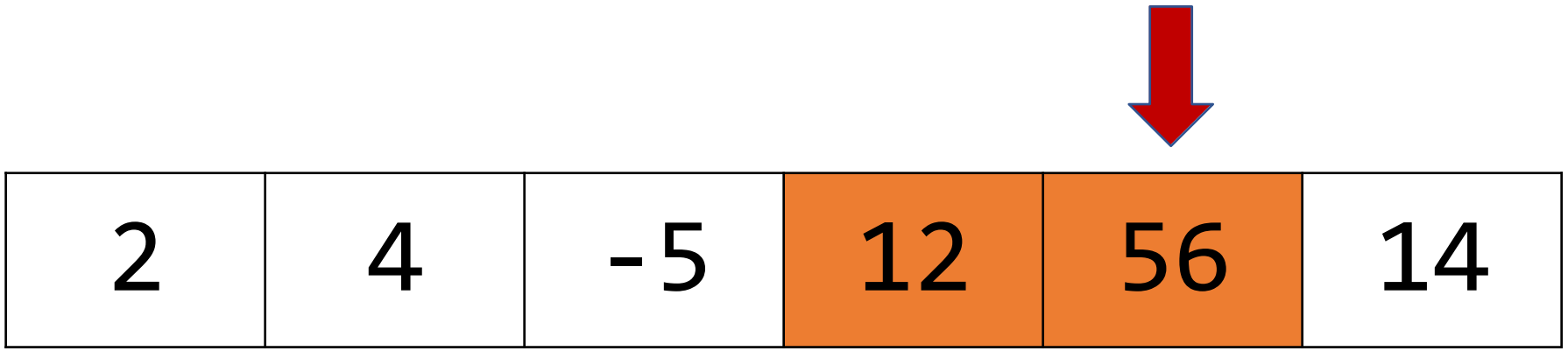
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.




- 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

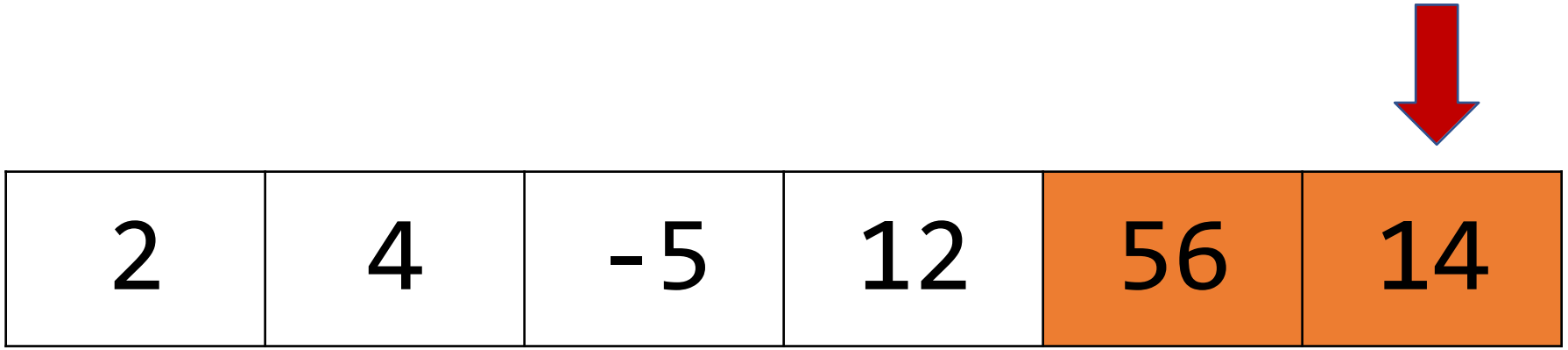
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.




- 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

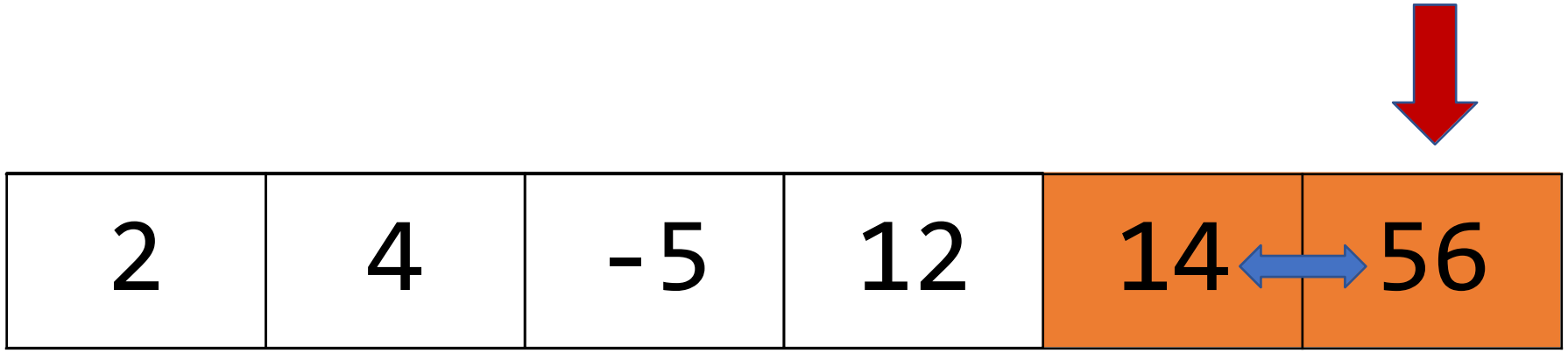
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.




- 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

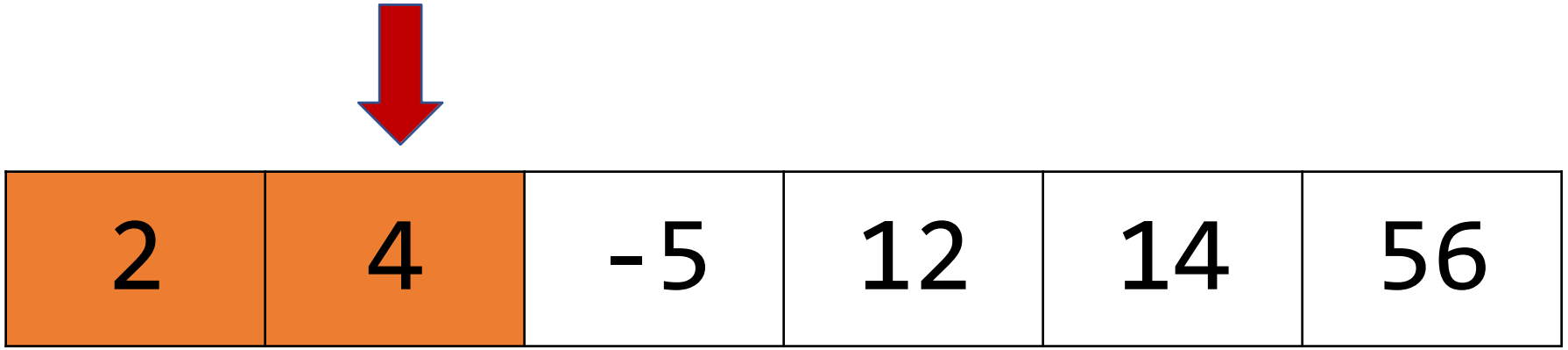
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.




- 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

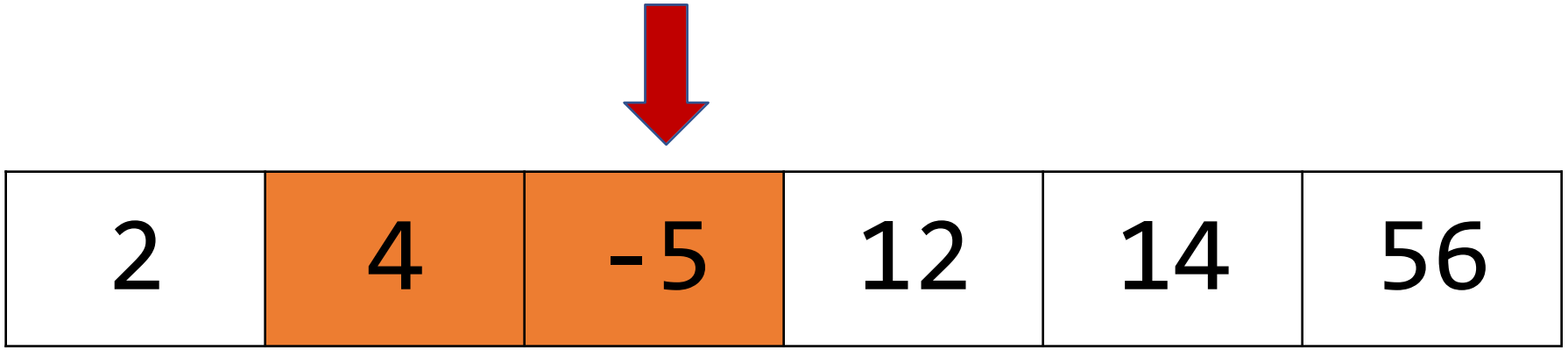
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.




- 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

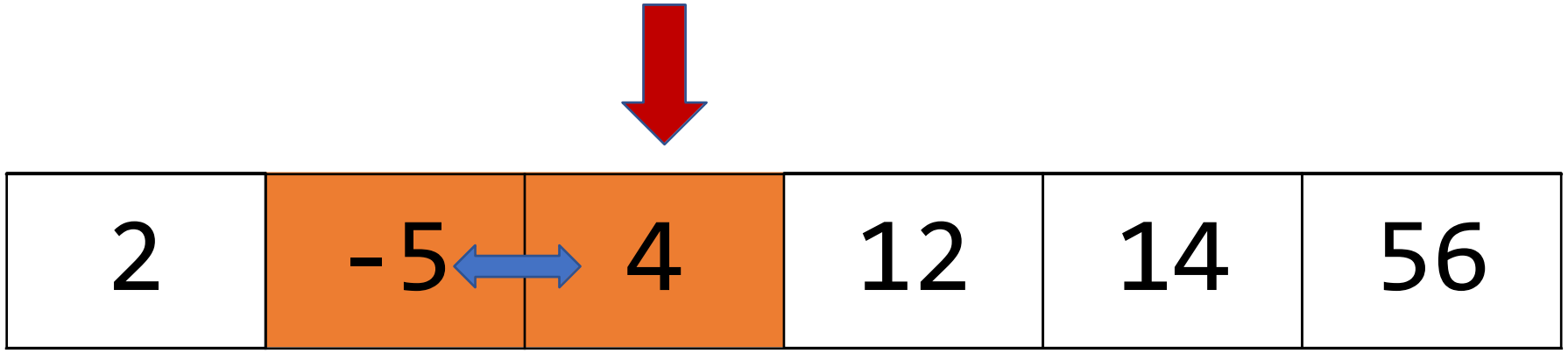
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.




- 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

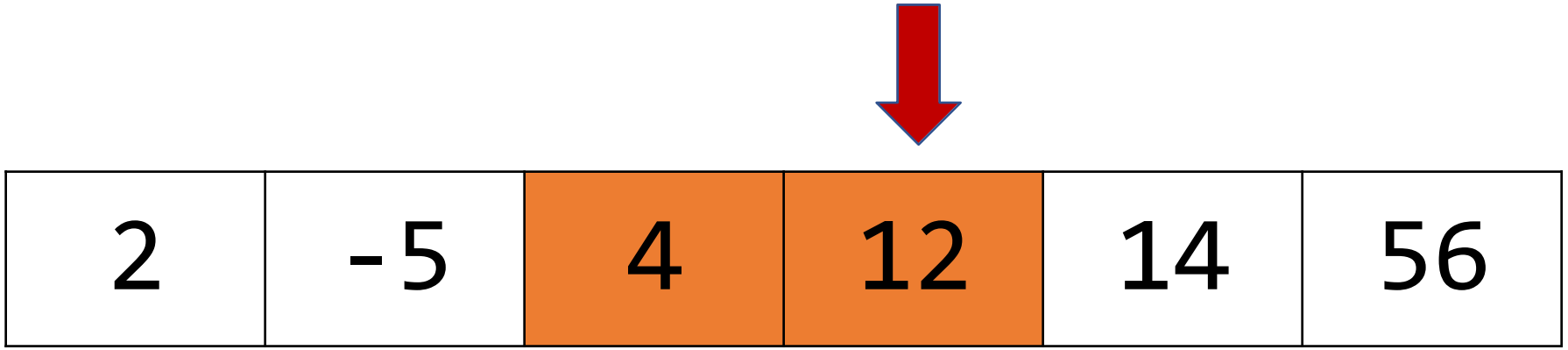
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.




- 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

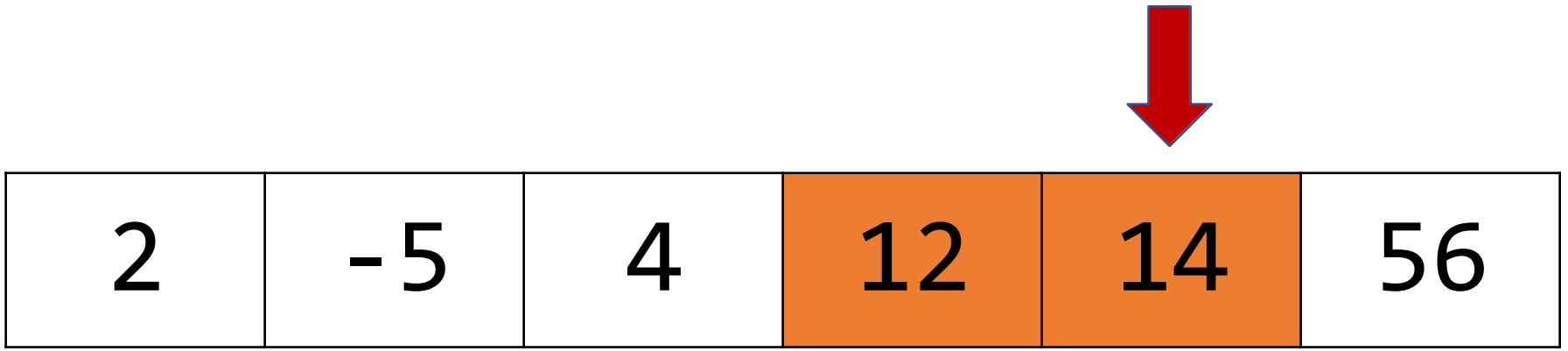
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.




- 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

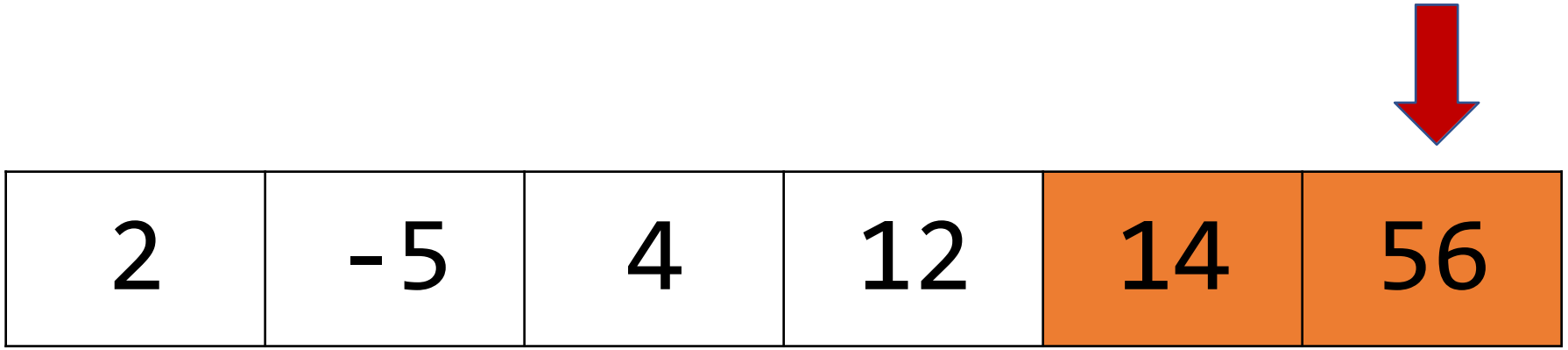
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.




- 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**.




- 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 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!

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, int 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?
(1) To start, this function always sorts in ascending order. What about other orders?

# Integer Bubble Sort

```
void bubble_sort_int(int *arr, int 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;
        }
    }
}
```

(1) 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, int 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;
        }
    }
}
```

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

# Integer Bubble Sort

```
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;
        }
    }
}
```

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

# Integer Bubble Sort

```
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;
        }
    }
}
```

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

# Generic Bubble Sort

```
void bubble_sort(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.

# Key Idea: Locating i-th Elem

A common generics idiom is getting a pointer to the i-th element of a generic array.

Recall how we located the **last** element of an array:

```
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?

# Key Idea: Locating i-th Elem

A common generics idiom is getting a pointer to the i-th element of a generic array.

Recall how we located the **last** element of an array:

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

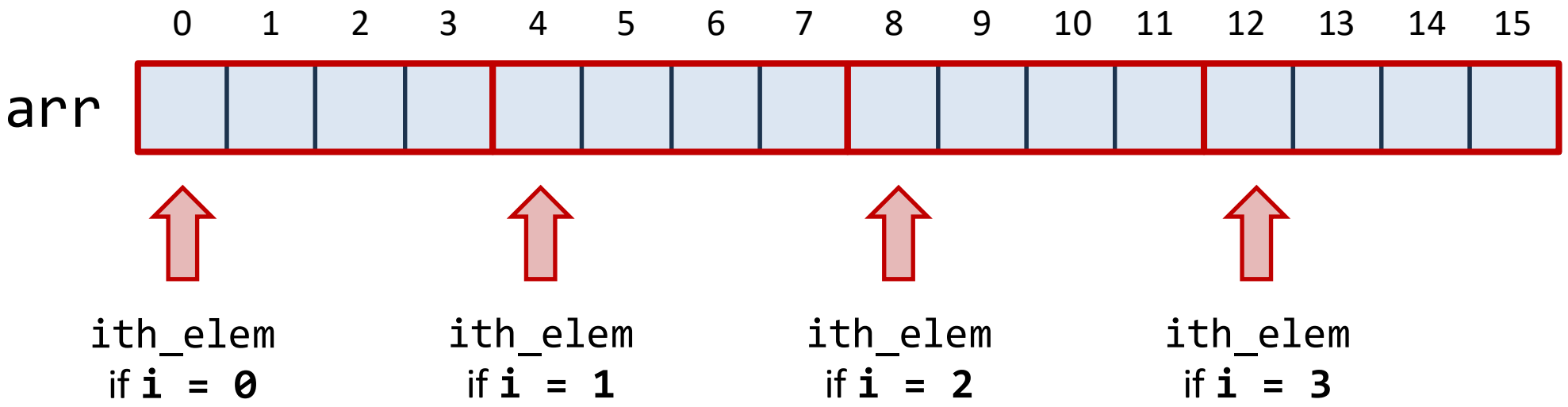
How can we generalize this to get the location of the i-th element?

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

# Key Idea: Locating i-th Elem

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

If **elem_bytes** is 4…



# Generic Bubble Sort

```
void bubble_sort(void *arr, int n, int elem_size_bytes) {
    while (true) {
        bool swapped = false;
        for (int 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 (*p_prev_elem > *p_curr_elem) {
                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, int n, int elem_size_bytes) {
    while (true) {
        bool swapped = false;
        for (int 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 (*p_prev_elem > *p_curr_elem) {
                swapped = true;
                swap(p_prev_elem, p_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 snag – 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 supplying the data….let's ask them!

# Generic Bubble Sort

```
void bubble_sort(void *arr, int n, int elem_size_bytes) {
    while (true) {
        bool swapped = false;
        for (int 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 (*p_prev_elem > *p_curr_elem) {
                swapped = true;
                swap(p_prev_elem, p_curr_elem, elem_size_bytes);
            }
        }

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

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

# Generic Bubble Sort

```
void bubble_sort(void *arr, int n, int elem_size_bytes) {
    while (true) {
        bool swapped = false;
        for (int 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 (*p_prev_elem > *p_curr_elem) {
                swapped = true;
                swap(p_prev_elem, p_curr_elem, elem_size_bytes);
            }
        }

        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, int n, int elem_size_bytes,
                                        function compare_fn) {
    while (true) {
        bool swapped = false;
        for (int 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 (compare_fn(p_prev_elem, p_curr_elem)) {
                swapped = true;
                swap(p_prev_elem, p_curr_elem, elem_size_bytes);
            }
        }

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

So, how can we receive this comparison function? The **function will be a parameter**. And it is job will be to tell us how the two elements compare.

# Generic Bubble Sort

```
void bubble_sort(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 *p_prev_elem = (char *)arr + (i - 1) * elem_size_bytes;
            void *p_curr_elem = (char *)arr + i * elem_size_bytes;
            if (compare_fn(p_prev_elem, p_curr_elem)) {
                swapped = true;
                swap(p_prev_elem, p_curr_elem, elem_size_bytes);
            }
        }

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

So the expected return will be a bool, and the function's expected input will be two void * arguments.

# 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)
```

# 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)
```

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.

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

Function pointer name
(compare_fn)

# 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)
```

Function parameters
(two void *s)

# Function Pointers

Here's the general variable type syntax:

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

# Generic Bubble Sort

```
void bubble_sort(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 *p_prev_elem = (char *)arr + (i - 1) * elem_size_bytes;
            void *p_curr_elem = (char *)arr + i * elem_size_bytes;
            if (compare_fn(p_prev_elem, p_curr_elem)) {
                swapped = true;
                swap(p_prev_elem, p_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);
    ...
}
```

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

```
bool string_compare(void *ptr1, void *ptr2) {
    ...
}

int main(int argc, char *argv[]) {
    char *classes[] = {"CS106A", "CS106B", "CS107", "CS110"};
    int arr_count = sizeof(classes) / sizeof(classes[0]);
    bubble_sort(classes, arr_count, sizeof(classes[0]), string_compare);
    ...
}
```

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.

# Bubble Sort

```
void bubble_sort(void *arr, int n, int elem_size_bytes,
                 bool (*compare_fn)(void *a, void *b))
```

- Bubble Sort is written as a generic library function to be imported into potentially many programs to be used with many types. It must have a single function signature but work with any type of data.
- Its comparison function type is part of its function signature – the comparison function signature must use one set of types but accept any data of any size. How do we do this?
  - **The function will instead accept pointers to the data via void * parameters**
  - This means that the functions must be written to handle parameters which are *pointers to the data* to be compared

# Comparison Functions

This means that functions with generic parameters must always take *pointers to the data they care about.*

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)

# Comparison Functions

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

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

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

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.

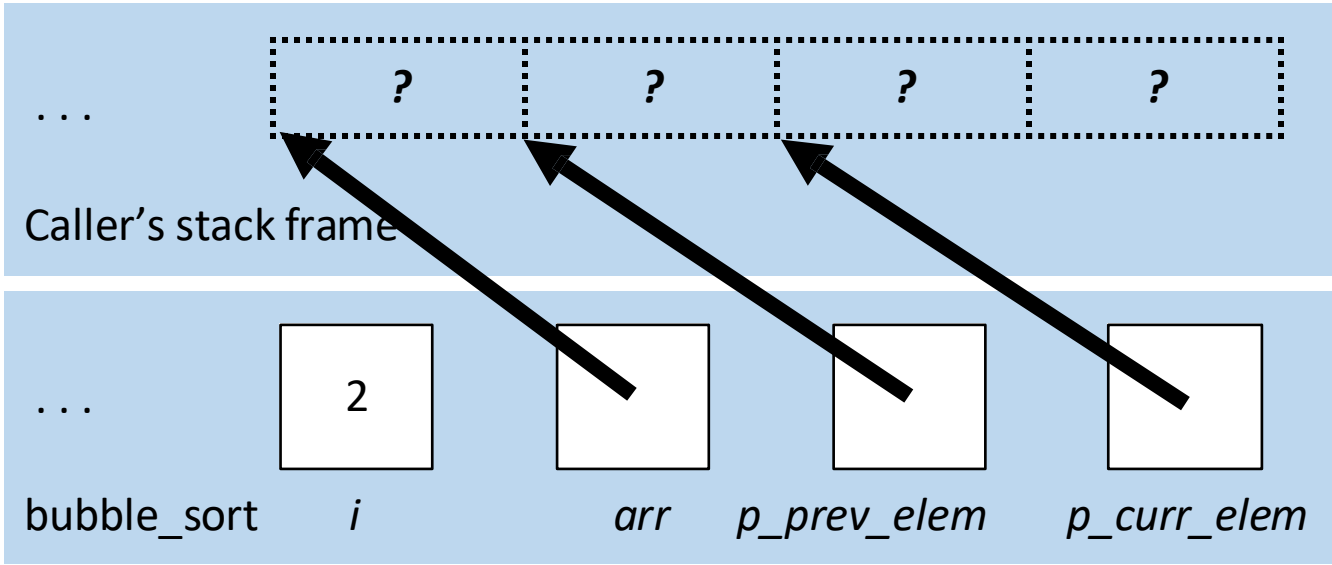
# Comparison Functions

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

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

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

However, the type of the comparison function that e.g. **`bubble_sort`** accepts must be generic, since we are writing one **`bubble_sort`** function to work with any data type.

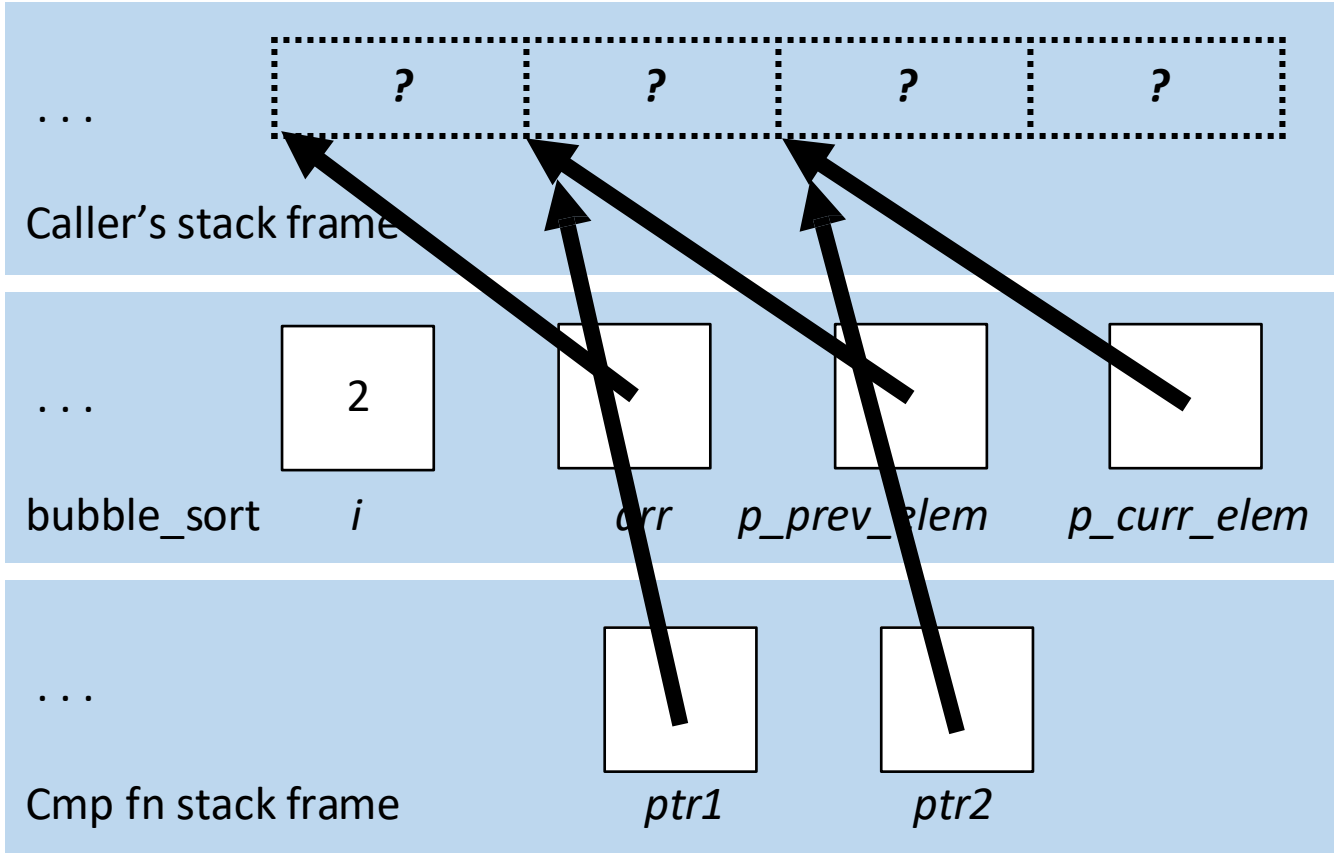
# Generic Bubble Sort

```
void bubble_sort(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 *p_prev_elem = (char *)arr + (i - 1) * elem_size_bytes;
            void *p_curr_elem = (char *)arr + i * elem_size_bytes;
            if (compare_fn(p_prev_elem, p_curr_elem)) {
                swapped = true;
                swap(p_prev_elem, p_curr_elem, elem_size_bytes);
            }
        }

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



# Comparison Functions

```
bool integer_compare(void *ptr1, void *ptr2) {
    return *(int *)ptr1 > *(int *)ptr2;
}
```



# 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 *a, void *b)
```

# Comparison Functions

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

# Generic Bubble Sort

```
void bubble_sort(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 *p_prev_elem = (char *)arr + (i - 1) * elem_size_bytes;
            void *p_curr_elem = (char *)arr + i * elem_size_bytes;
            if (compare_fn(p_prev_elem, p_curr_elem) > 0) {
                swapped = true;
                swap(p_prev_elem, p_curr_elem, elem_size_bytes);
            }
        }

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

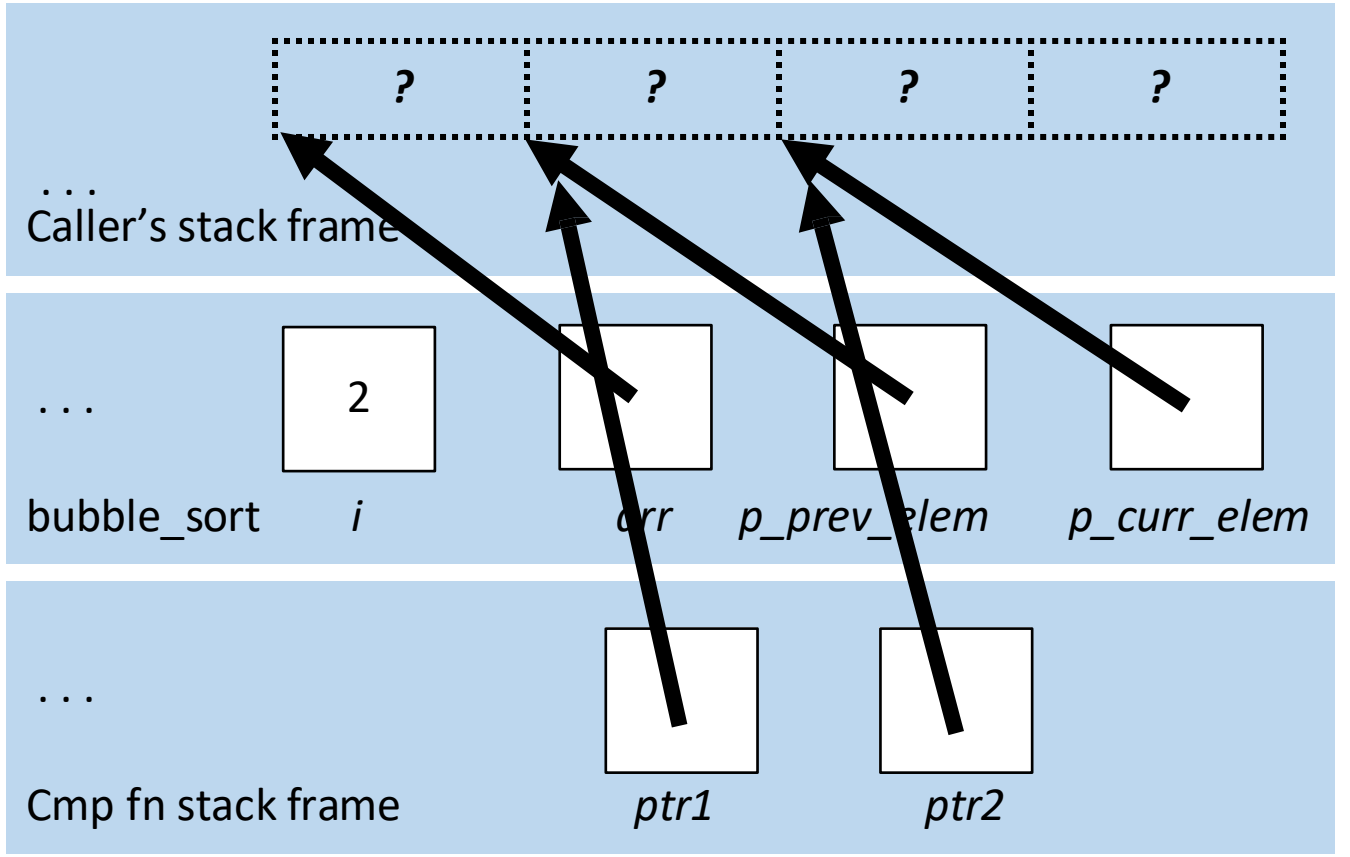
# Comparison Functions

- **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

```
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 if 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?

# 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…

# The meaning of "callback" functions

**Library writer**

- Writes generic algorithmic functions
- Relies on user-provided nelems, sizeof(elem), function pointer

```
void print_array(void *arr, size_t nelems,
                 int elem_size,
                 void(*print_fn)(void *)) {
    …
}
```

**User/caller**

- Knows the data
- Might not know the algorithm (hence the use of library function)
- Writes the callback function to pass into library function

```
void print_string(void *ptr) {
  …
}
int main(int argc, char *argv[]) {
  …
  print_array(str_array, n_elems,
        sizeof(str_array[0]), print_string);
  …
}
```

The library uses a user-written (and user-provided!) **callback function** to perform complex operations on generic data.

# Common Utility Callback Functions

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

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

- Printing function – prints out an element of a given type

```
void (*print_fn)(void *addr)
```

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

# Function Pointers As Variables

In addition to parameters, you can make normal variables that are functions.

```
1  int do_something (char *str) {
2    printf("%s\n", str);
3    return strlen(str);
4  }

5  int main(int argc, char *argv[]) {
6      // Do something with variables
7      int (*func_var)(char *) = do_something;
8      char *str = "testing";
9      int retval = func_var(str);
10     printf("%d\n", retval);
11     return 0;
12 }
```

# Generic C Standard Library Functions

- **qsort** – I can sort an array of any type! To do that, I need you to provide me a function that can compare two elements of the kind you are asking me to sort.
- **bsearch** – I can use binary search to search for a key in an array of any type! To do that, I need you to provide me a function that can compare two elements of the kind you are asking me to search.
- **lfind** – I can use linear search to search for a key in an array of any type! To do that, I need you to provide me a function that can compare two elements of the kind you are asking me to search.
- **lsearch** - I can use linear search to search for a key in an array of any type! I will also add the key for you if I can't find it. In order to do that, I need you to provide me a function that can compare two elements of the kind you are asking me to search.

# Generic C Standard Library Functions

- **scandir** – I can create a directory listing with any order and contents! To do that, I need you to provide me a function that tells me whether you want me to include a given directory entry in the listing. I also need you to provide me a function that tells me the correct ordering of two given directory entries.

# Recap

- 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.
- Function pointers also allow us to pass logic around in our programs.
- Functions handling generic data must use *pointers to the data they care about*, since any parameters must have *one type* and *one size*.

# 🌟 Common code snippets

- Generic function: **Iterate through a generic array.**

```
for (int i = 0; i < nelems; i++) {
    void *curr_p = (char *)base + i * elem_size_bytes;
    …
}
```

- User setup: **Compute the number of elements in a local array**.

```
int *int_array[] = …; // declared locally
size_t nelems = sizeof(int_array) / sizeof(int_array[0]);
```

# What would happen here?

Recall print_array:

```
void print_array(void *arr, size_t nelems, int elem_size_bytes,
                 void(*print_fn)(void *)) {
    for (int i = 0; i < nelems; i++) {
        void *elem_ptr = (char *)arr + i * elem_size_bytes;
        printf("%d: ", i + 1);
        print_fn(elem_ptr);
        printf("\n");
    }

}
```

```
1 void print_string(void *ptr) {
2   char *str = ____???____;
3   printf("%s", str);
4 }
```

1. Fill in the blank so that `print_array` can print an array of strings.

   A. `(char *) ptr`
   B. `*(char **) ptr`
   C. `*(char *) ptr`
   D. `**(char ***) ptr`

2. Why would A or D ***not*** "work"?

# What would happen here?

**Library**

```
void print_array(void *arr, size_t nelems, int elem_size,
                 void(*print_fn)(void *)) {
   …
   void *elem_ptr = (char *)arr + i * elem_size_bytes;
   print_fn(elem_ptr);
   …
}
```

**Caller**

```
void print_string(void *ptr) {
  char *str = ____???____;
  printf("%s", str);
}

int main(int argc, char *argv[]) {
  char *str_array[] = {"aardvark", "beaver", "capybara"};
  size_t n_elems = sizeof(str_array) / sizeof(str_array[0]);
  print_array(str_array, n_elems, sizeof(str_array[0]), print_string);
  …
}
```

What is the *actual* type of the parameter passed into `print_string`?

As a caller: Remember what the true types of parameters are.

# Practice: Count Matches

- Let's write a generic function *count_matches* that can count the number of a certain type of element in a generic array.
- It should take in as parameters information about the generic array, and a function parameter that can take in a pointer to a single array element and tell us if it's a match.

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

# Practice: Count Matches

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

    int match_count = 0;

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

    return match_count;
}
```