

# COMP 201

## Computer Systems & Programming

Lecture #11 – Function Pointers



**KOÇ**  
**UNIVERSITY**

Aykut Erdem // Koç University // Spring 2024



# Recap

- Overview: Generics
- Generic Swap
- Generics Pitfalls
- Generic Array Swap

# 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

- Generics So Far
- Motivating Example: Bubble Sort
- Function Pointers
- Example: Generic Printing
- Example: Counting Matches

**Disclaimer:** Slides for this lecture were borrowed from  
—Nick Troccoli's Stanford CS107 class

# Lecture Plan

- Generics So Far
- Motivating Example: Bubble Sort
- Function Pointers
- Example: Generic Printing
- Example: Counting Matches

# Generics So Far

- **void \*** is a variable type that represents a generic pointer “to something”.
- We cannot perform pointer arithmetic with or dereference 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.

# Recap: Generic Swap

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

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

# Recap: Generic Swap

```
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);  
}
```

```
int x = 2;  
int y = 5;  
swap(&x, &y, sizeof(x));
```



# Recap: Generic Swap

```
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);  
}
```

```
short x = 2;  
short y = 5;  
swap(&x, &y, sizeof(x));
```

# Recap: Generic Swap

```
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);  
}
```

```
char *x = "2";  
char *y = "5";  
swap(&x, &y, sizeof(x));
```

# Recap: Generic Array Swap

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

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

# Recap: Swap Ends

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

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

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

# Recap: Swap Ends

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

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

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



# Recap: Swap Ends

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

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

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

# Array Rotation

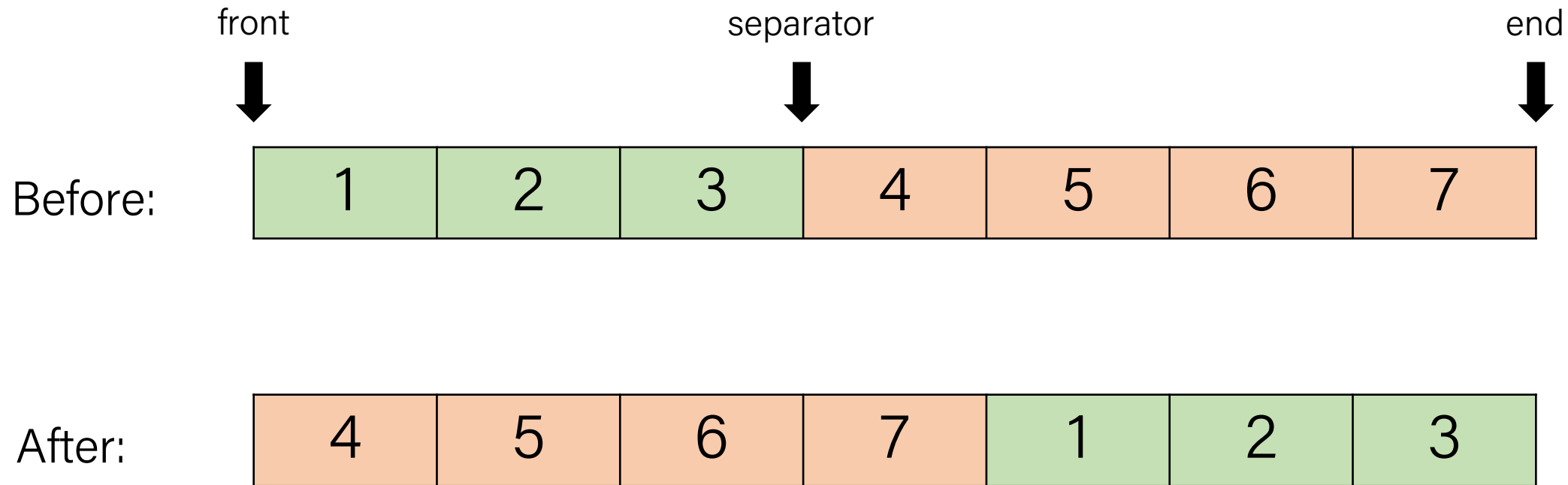
You're asked to provide an implementation for a function called **rotate** with the following prototype:

```
void rotate(void *front, void *separator, void *end);
```

The expectation is that **front** is the base address of an array, **end** is the past-the-end address of the array, and **separator** is the address of some element in between. **rotate** moves all elements in between **front** and **separator** to the end of the array, and all elements between **separator** and **end** move to the front.

# Array Rotation

```
int array[7] = {1, 2, 3, 4, 5, 6, 7};  
rotate(array, array + 3, array + 7);
```



# Demo: *Array Rotation*



rotate.c

# Array Rotation

A properly implemented **rotate** will prompt the following program to generate the provided output.

```
int main(int argc, char *argv[]) {
    int array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    print_int_array(array, 10); // intuitive implementation 😊
    rotate(array, array + 5, array + 10);
    print_int_array(array, 10);
    rotate(array, array + 1, array + 10);
    print_int_array(array, 10);
    rotate(array + 4, array + 5, array + 6);
    print_int_array(array, 10);
    return 0;
}
```

Output:

```
linuxpool :~/lect11$ ./rotate
Array: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Array: 6, 7, 8, 9, 10, 1, 2, 3, 4, 5
Array: 7, 8, 9, 10, 1, 2, 3, 4, 5, 6
Array: 7, 8, 9, 10, 2, 1, 3, 4, 5, 6
linuxpool:~/lect11$
```



# Array Rotation

A properly implemented **rotate** will prompt the following program to generate the provided output.

And here's that properly implemented function!

```
void rotate(void *front, void *separator, void *end) {
    int width = (char *)end - (char *)front;
    int prefix_width = (char *)separator - (char *)front;
    int suffix_width = width - prefix_width;

    char temp[prefix_width];
    memcpy(temp, front, prefix_width);
    memmove(front, separator, suffix_width);
    memcpy((char *)end - prefix_width, temp, prefix_width);
}
```

# memset

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

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

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

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

# Why are void \* pointers useful?

Because each parameter and return type must be a *single type* with a *single size*.

# Why Are `void *` Pointers Useful?

- Each parameter and return type must be a *single* type with a *single* size.
- **Problem #1:** for a function parameter to accept multiple data types, it needs to be able to accept data of different sizes.
  - **Key Idea #1:** pointers are all the same size regardless of what they point to. To pass different sizes of data via a single parameter type, make the parameter be a pointer to the data instead.
- **Problem #2:** we still might pass either a `char *`, `int *`, etc. These are the same size, but still different declared types. What should the parameter type be?
  - **Key Idea #2:** A `void *` encompasses all these types – it represents a “pointer to something”. A `char *`, `int *`, etc. all implicitly cast to `void *`.
- **Solution:** to pass one of multiple types via a single parameter/return, that parameter/return’s type can be **`void *`**, and we can pass a pointer to the data.

# Lecture Plan

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



# Bubble Sort

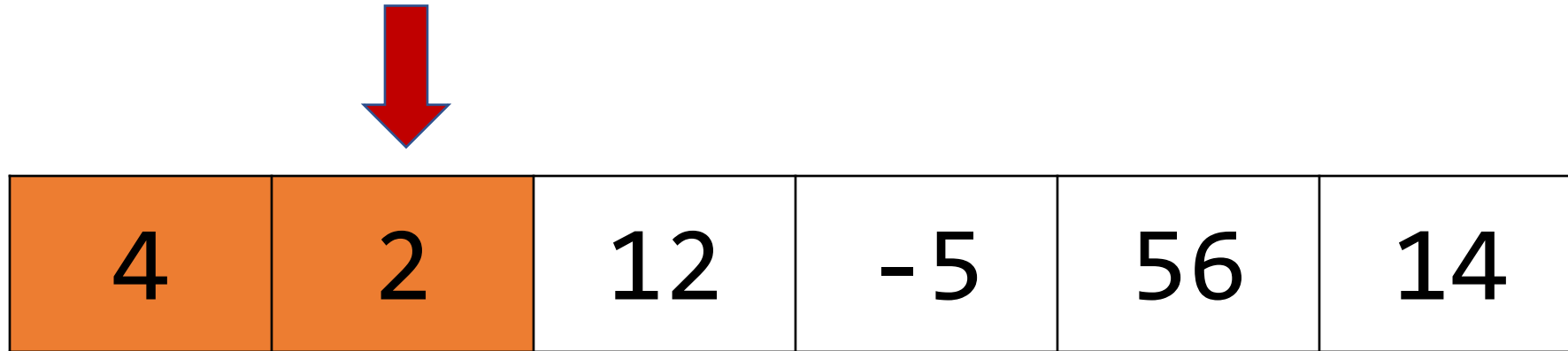
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.

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

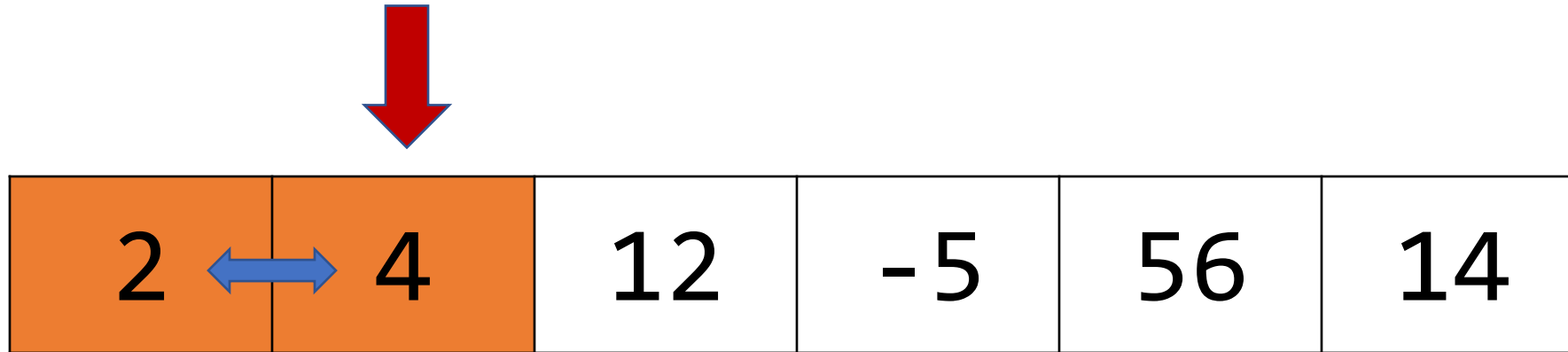
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- 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

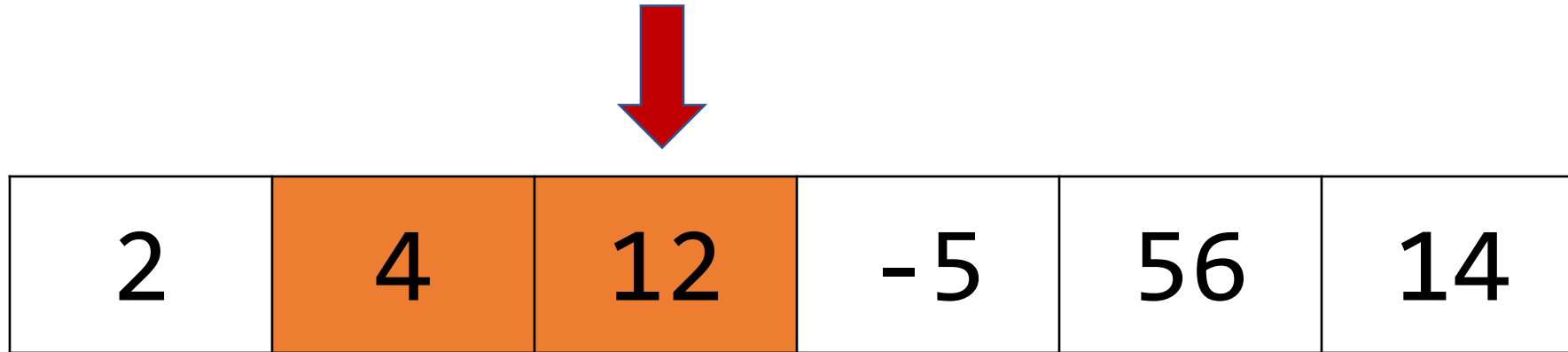
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- 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

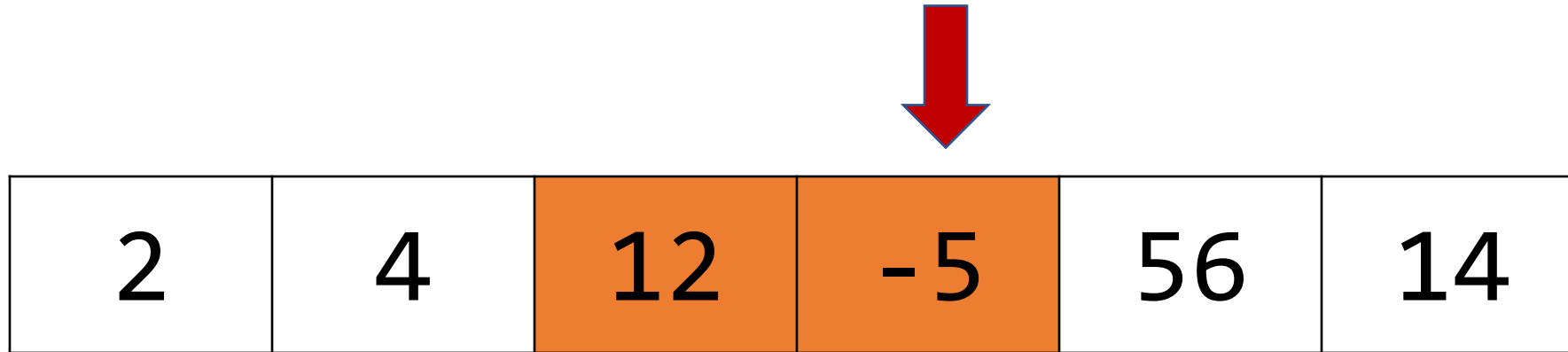
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- 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

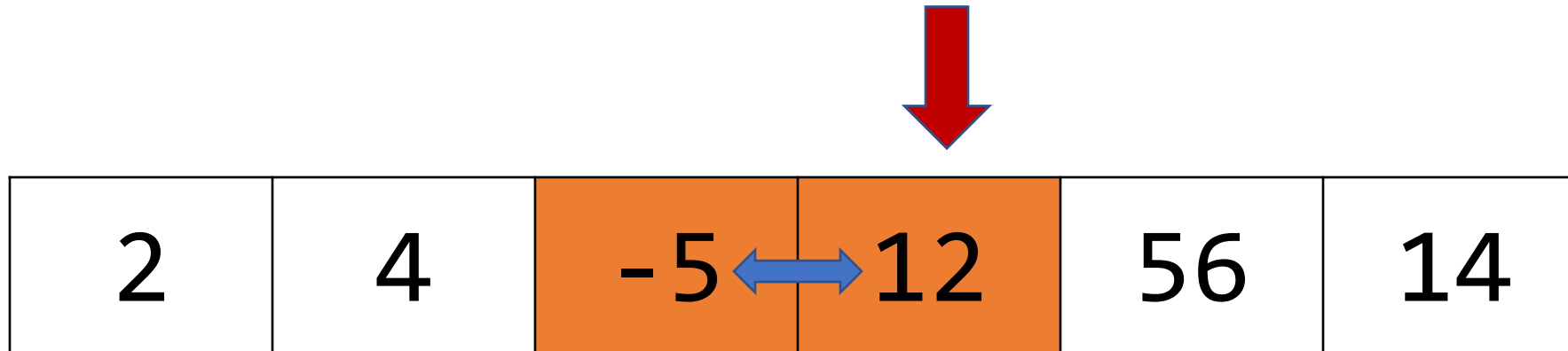
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- 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

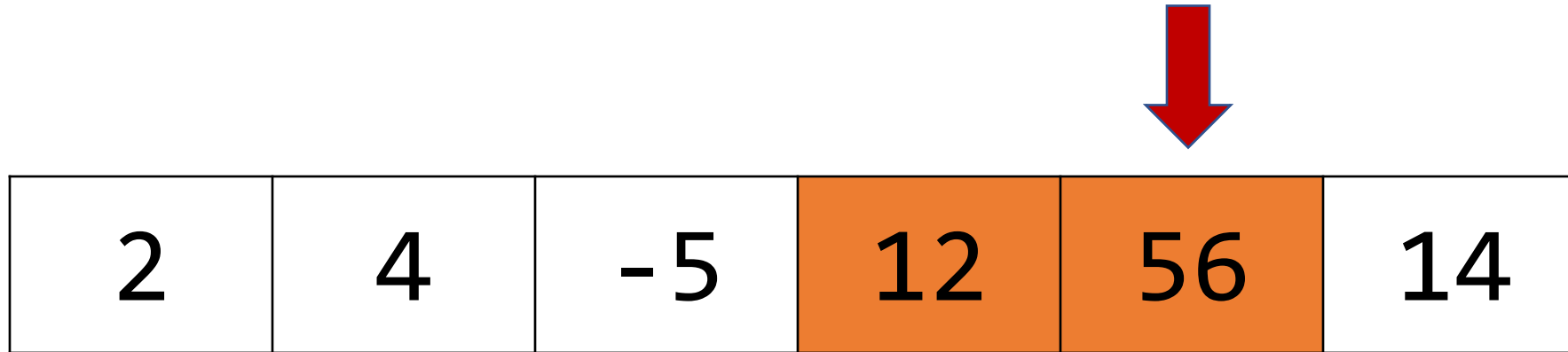
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- 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

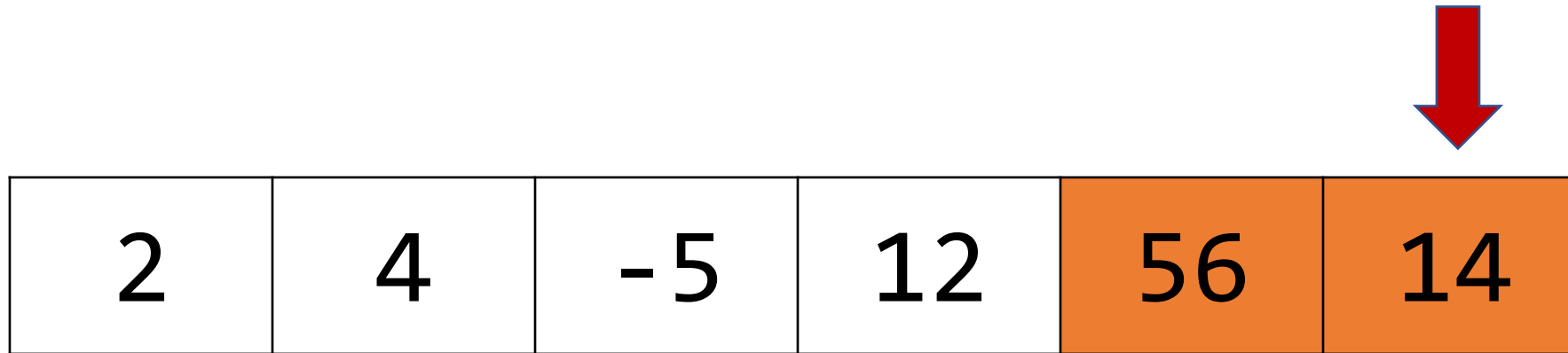
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- 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**.

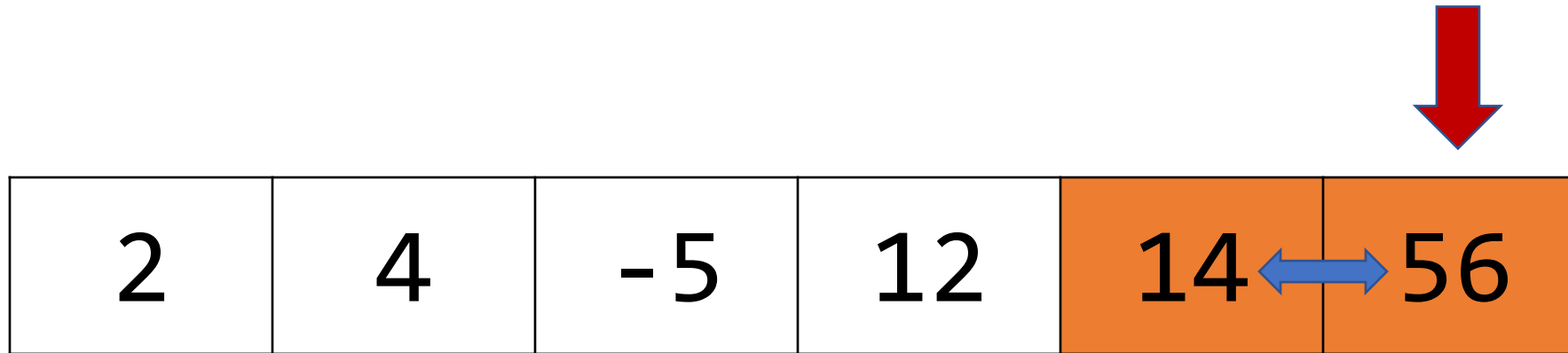


- Bubble sort repeatedly goes through the array, swapping any pairs of elements that are out of order. When there are no more swaps needed, the array is sorted!



# Bubble Sort

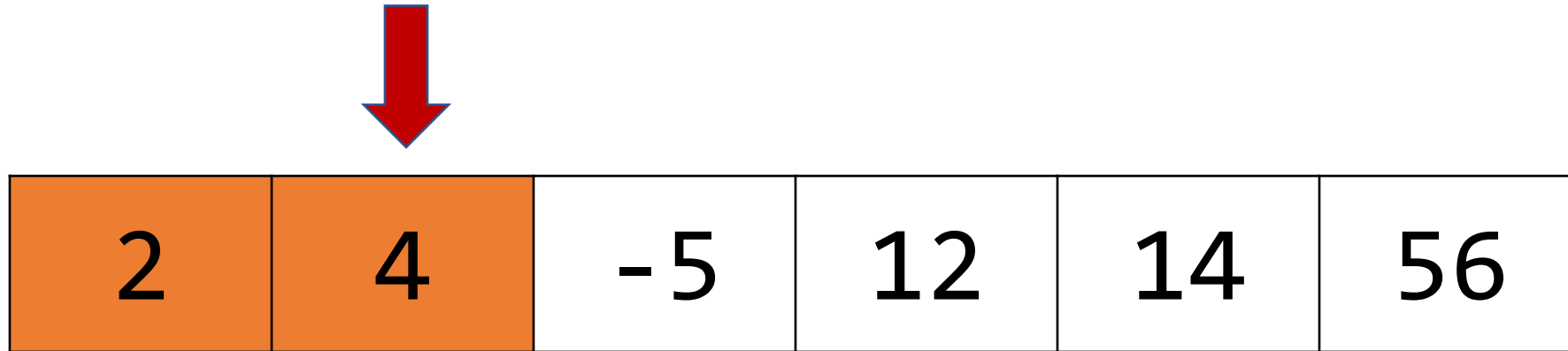
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- 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

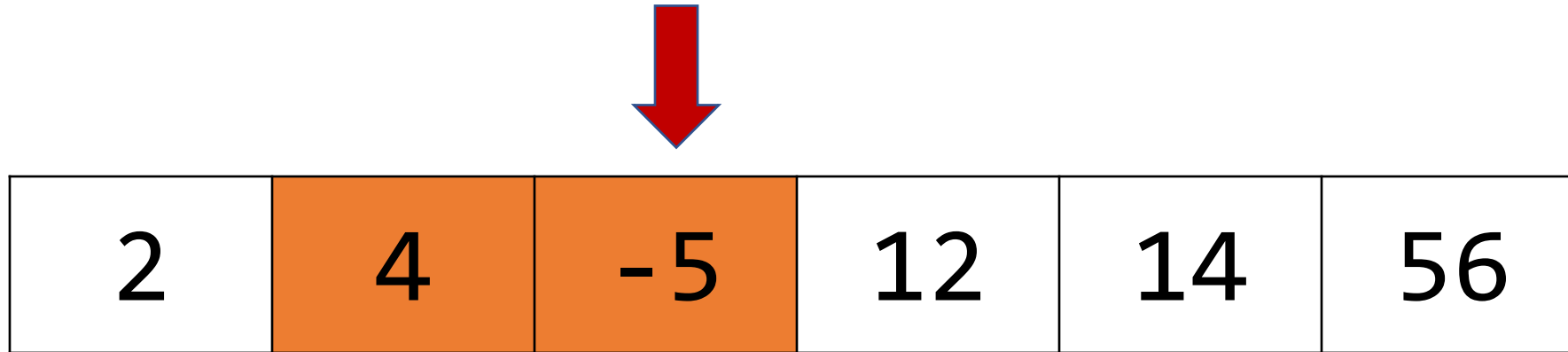
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- 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

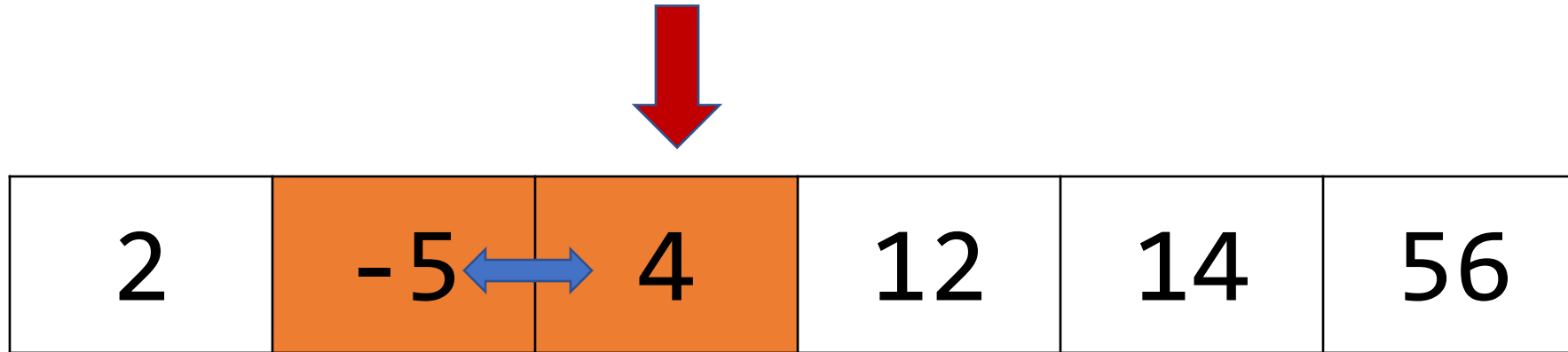
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- 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

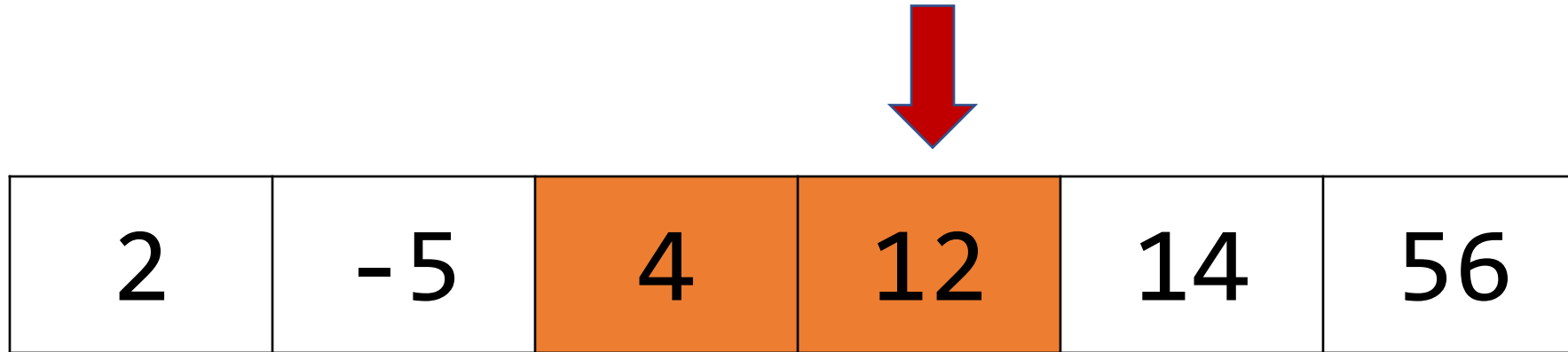
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- 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

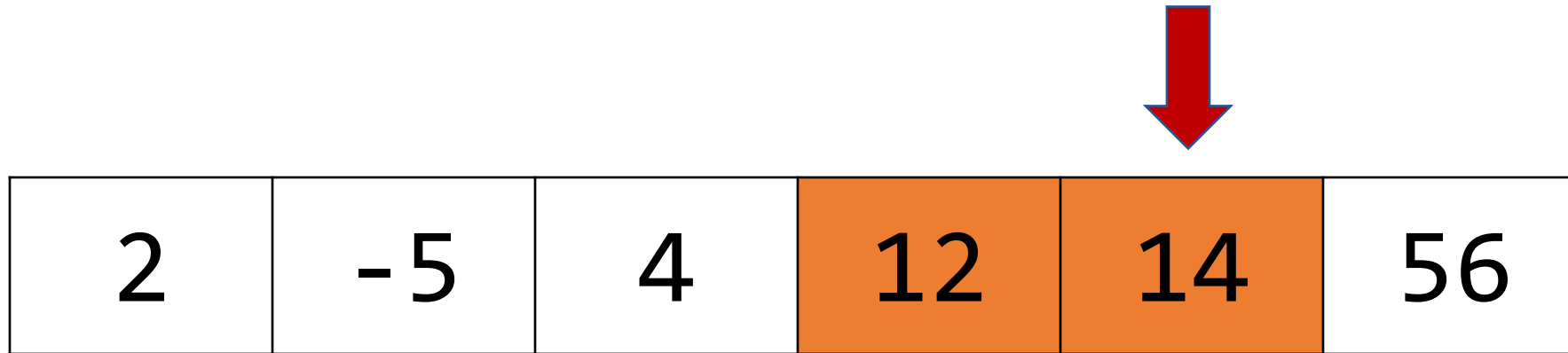
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- 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

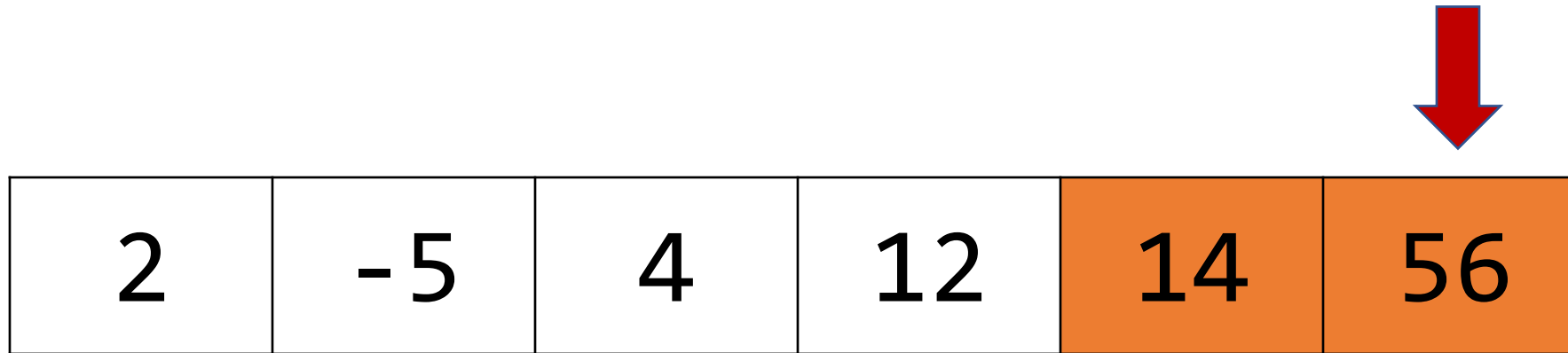
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- 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. 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. Only two more passes are needed to arrive at the above. The first pass 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 (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*?

# 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;  
        }  
    }  
}
```

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. From last lecture, we know how to locate the **last** element:

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

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

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

# Generic Bubble Sort

```
void bubble_sort(void *arr, 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;  
        }  
    }  
}
```

Are we done? Is something not right?



# 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!



# Lecture Plan

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

# 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;
        }
    }
}
```

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, 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;
        }
    }
}
```

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 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[] = {"COMP100", "COMP132", "COMP201", "COMP202"};  
    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.

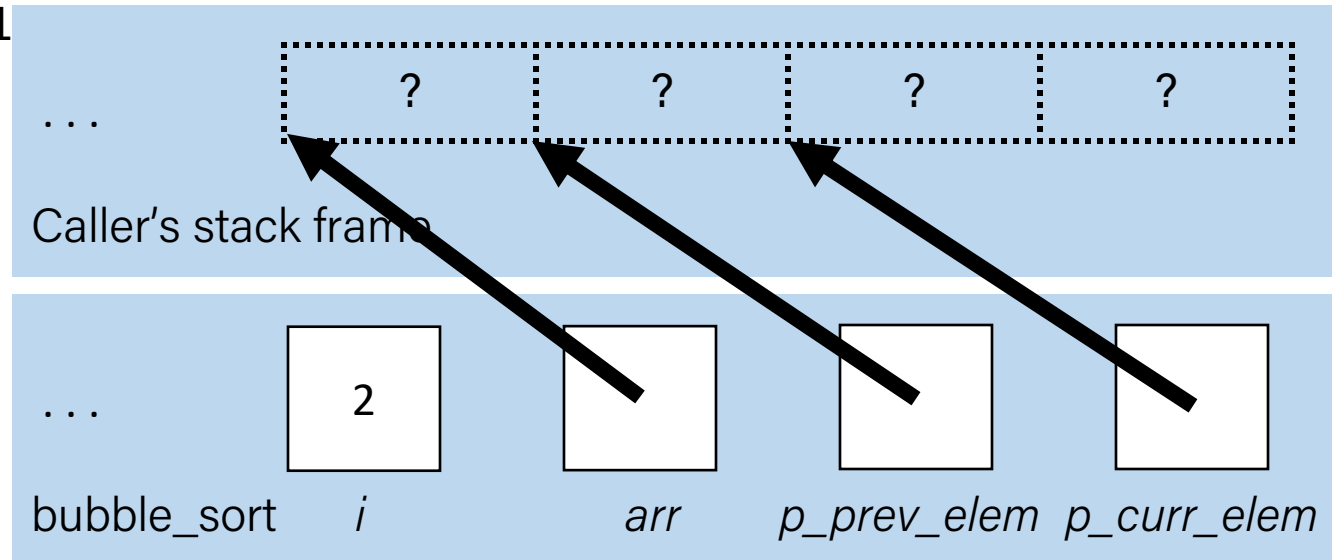
# Bubble Sort

```
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

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



# Function Pointers

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)



# Function Pointers

```
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.

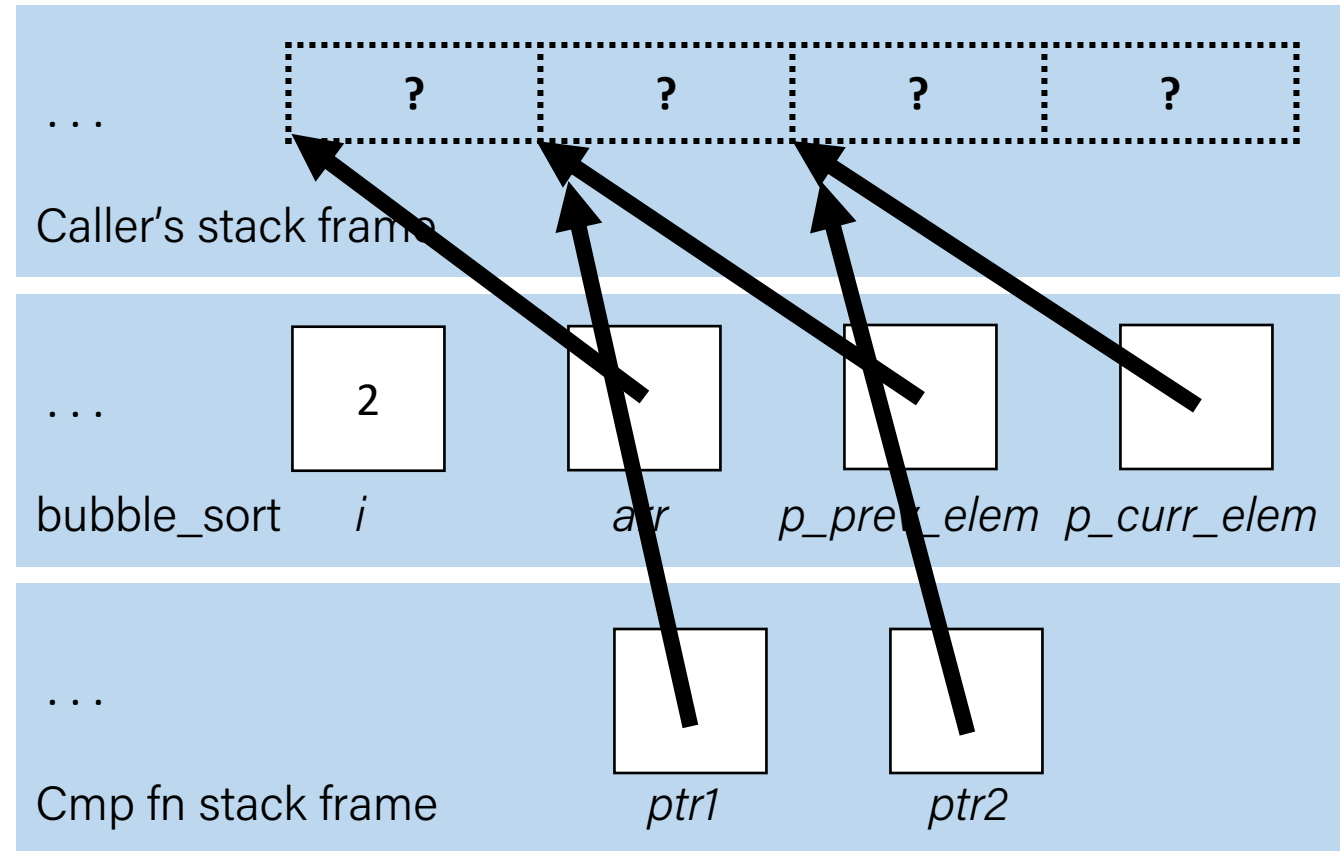
# Function Pointers

```
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.

# Function Pointers

```
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;
        }
    }
}
```

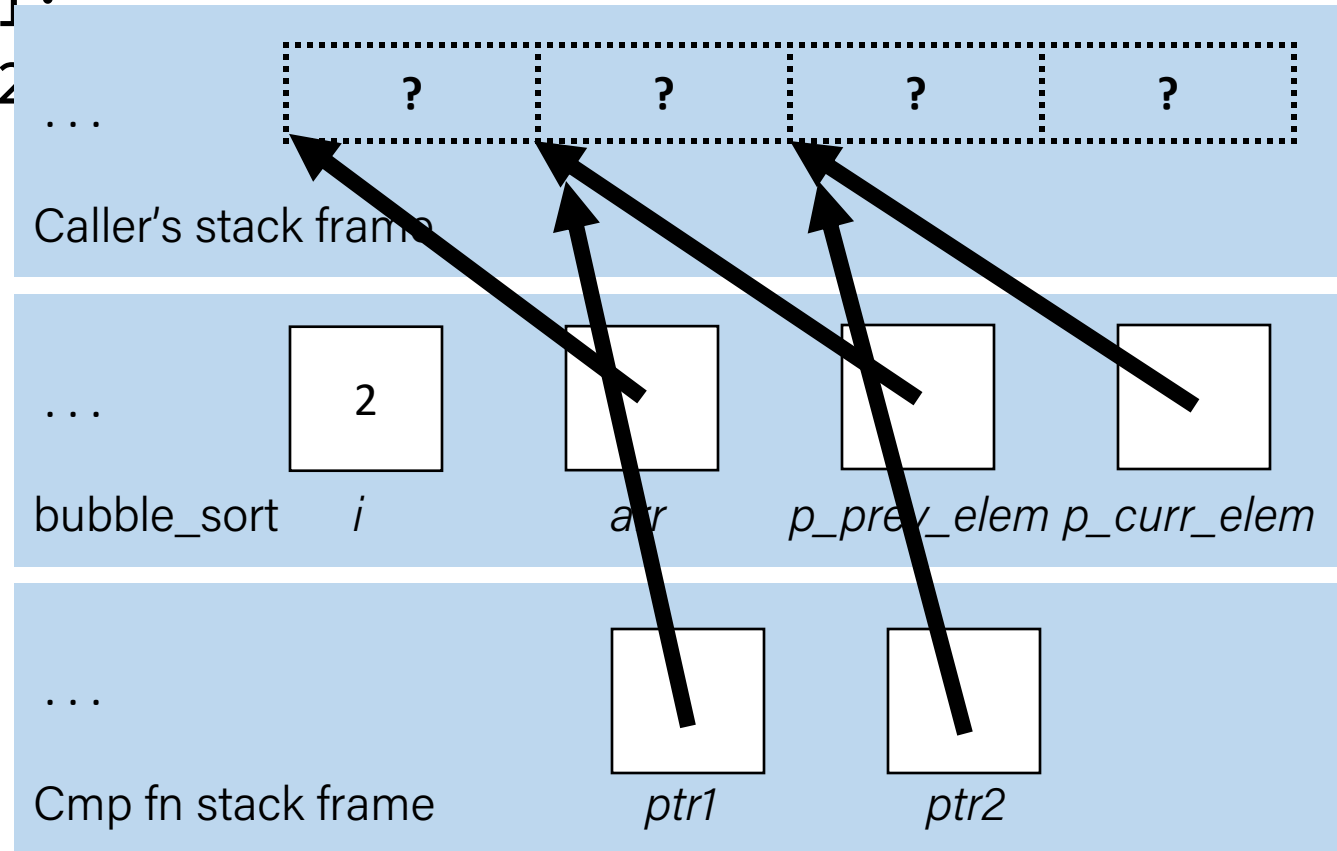
# Comparison Functions

- **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?

# Lecture Plan

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

# 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: Generic Printing



print\_array.c

# 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.

# Lecture Plan

- Generics So Far
- Motivating Example: Bubble Sort
- Function Pointers
- Example: Generic Printing
- Example: Counting Matches

# Demo: Count Matches



```
count_matches.c
```



# 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 *));
```

# 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;
}
```

# Function Pointers As Variables

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

```
int do_something(char *str) {  
    ...  
}  
  
int main(int argc, char *argv[]) {  
    ...  
    int (*func_var)(char *) = do_something;  
    ...  
    func_var("testing");  
    return 0;  
}
```

# 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.

# Summary: Function Pointers

- We can pass functions as parameters to pass logic around in our programs.
- Comparison functions are one common class of functions passed as parameters to generically compare the elements at two addresses.
- Functions handling generic data must use *pointers to the data they care about*, since any parameters must have *one type* and *one size*.

# Summary: Generics

- 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.

# Recap

- Generics So Far
- Motivating Example: Bubble Sort
- Function Pointers
- Example: Generic Printing
- Example: Counting Matches

Next time: *const*, *Structures*