CS107, Lecture 13
C Generics and Function Pointers

Reading: K&R 5.11

Ed Discussion: https://edstem.org/us/courses/46162/discussion/3697040

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 wrote a generic swap function last Friday. 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 beyond arr is this?

If it's an array of…

Int?

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

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

Short?

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 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 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

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 <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);
}
```

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]));
```

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]));
```

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]));
```

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 <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);
}
```

```
mystruct structs[] = …;
size_t nelems = …;
swap_ends(structs, nelems, sizeof(structs[0]));
```

Generics So Far

- **`void *`** is a variable type that represents a generic pointer "to something".
- We can't use pointer arithmetic on or dereference (without first casting) a **`void *`**.
- We can use **`memcpy`** or **`memmove`** to copy data from one memory location to another.
- To do manual pointer arithmetic with a **`void *`**, we must first cast it to a **`char *`**.
- **`void *`** and generics are powerful, but error-prone. They're error-prone because the compiler can't do type checking. That means we need to be extra careful when working with generic memory.

void * Pitfalls

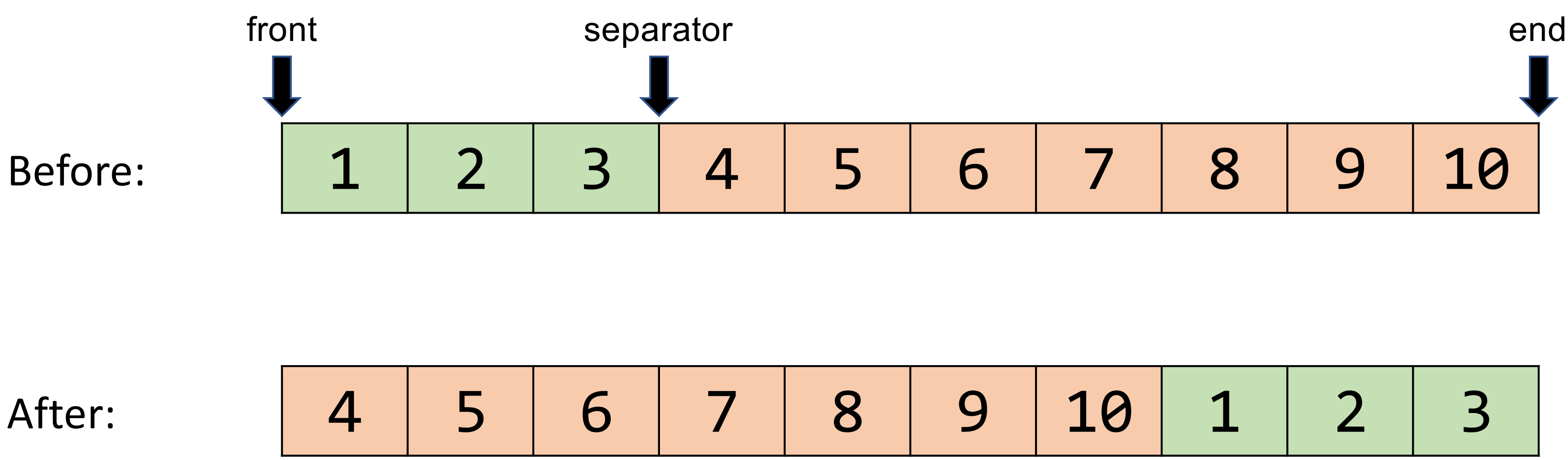
- **void ***s are powerful, but error-prone — C cannot do as much checking!
- e.g., with **int**, C would never let you swap *half* of an int. With **void *s**, it absolutely will!

```
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);
```

Exercise: Array Rotation

```
int array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
rotate(array, array + 3, array + 10);
```



Exercise: Array Rotation

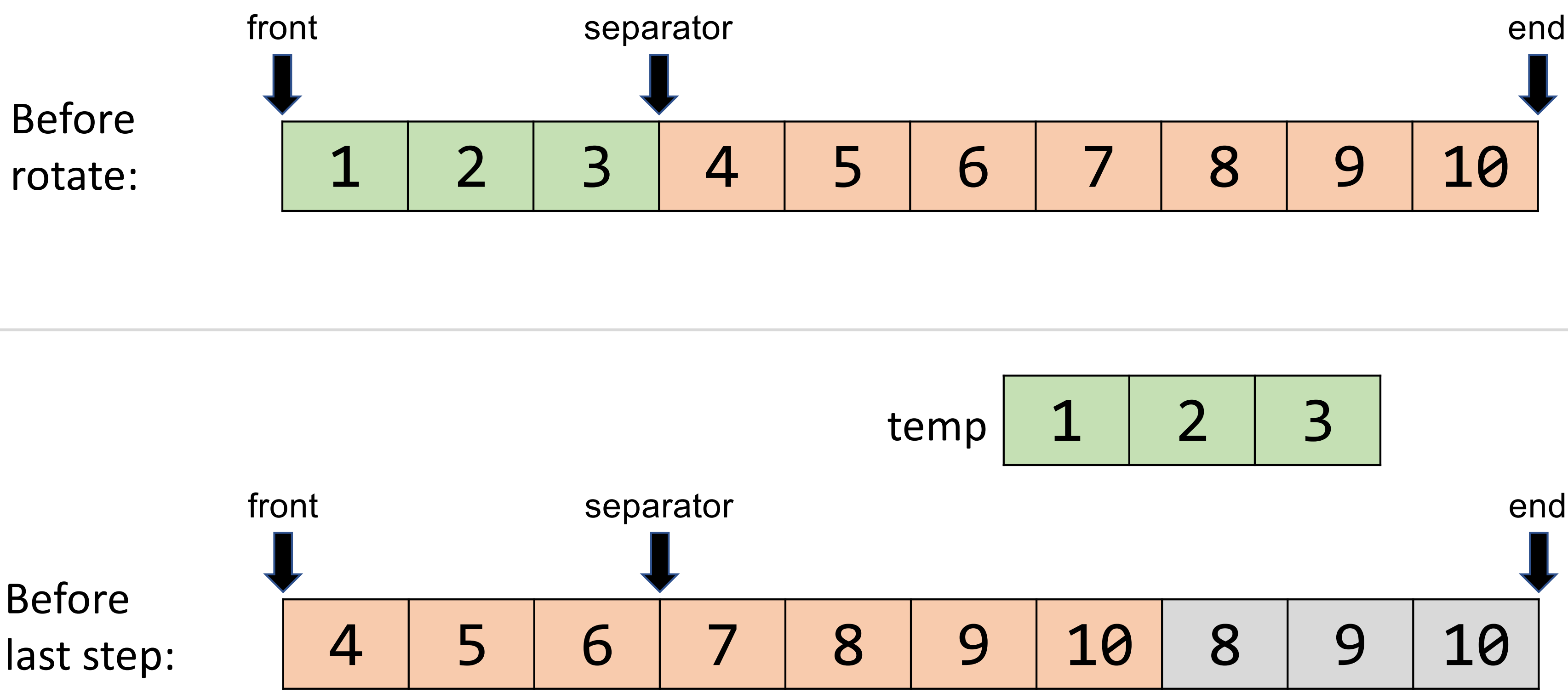
Exercise: Implement **rotate** to generate the provided output.

```
int main(int argc, char *argv[]) {
    int array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    print_int_array(array, 10); // intuit 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:

```
myth52:~/lect13$ ./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
myth52:~/lect13$
```

The inner workings of rotate



Exercise: Array Rotation

Exercise: 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);
}
```

Bubble Sort

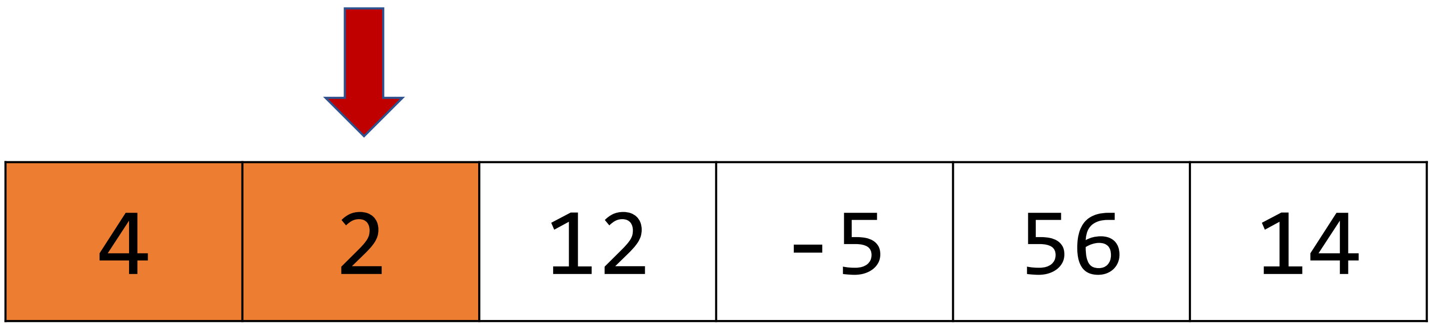
Let's write a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.

4	2	12	-5	56	14

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

Bubble Sort

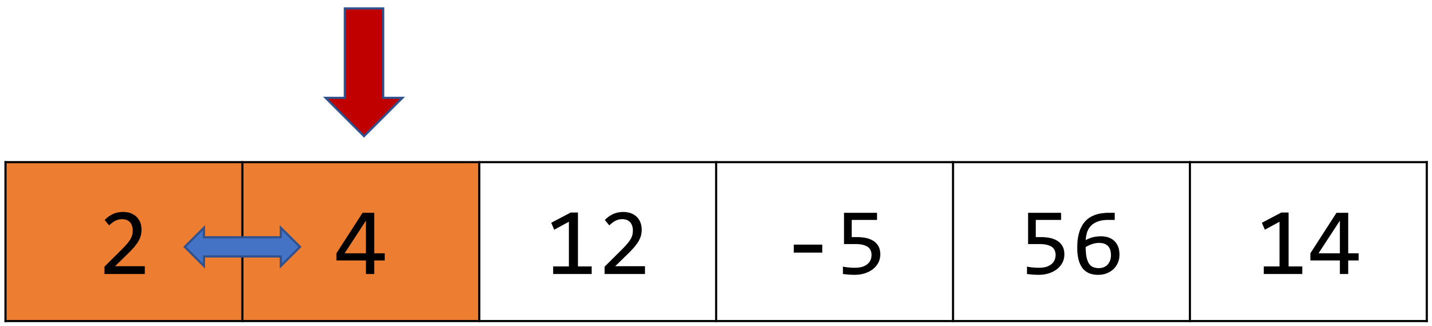
Let’s write a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.



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

Bubble Sort

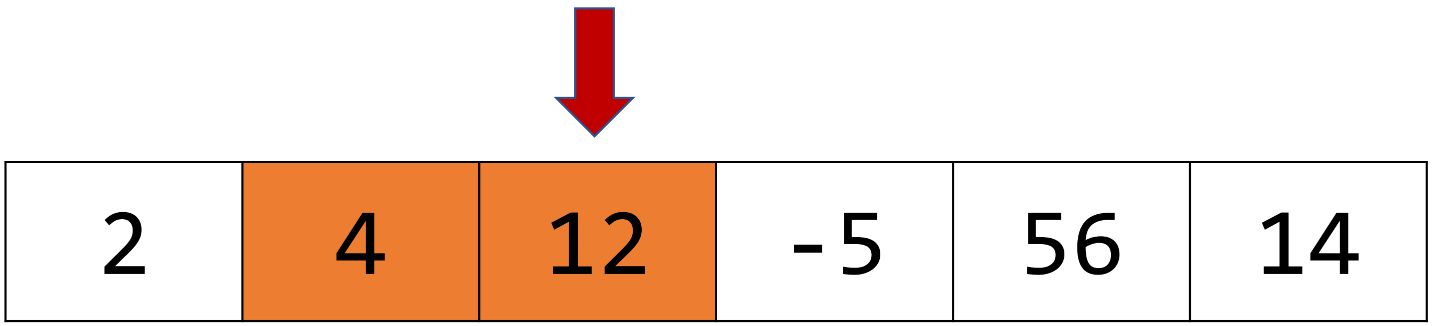
Let's write a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.



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

Bubble Sort

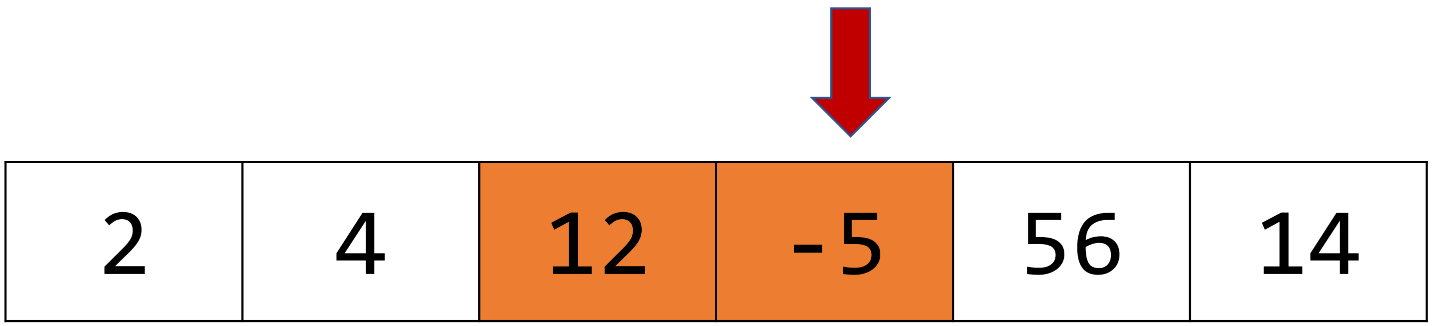
Let's write a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.



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

Bubble Sort

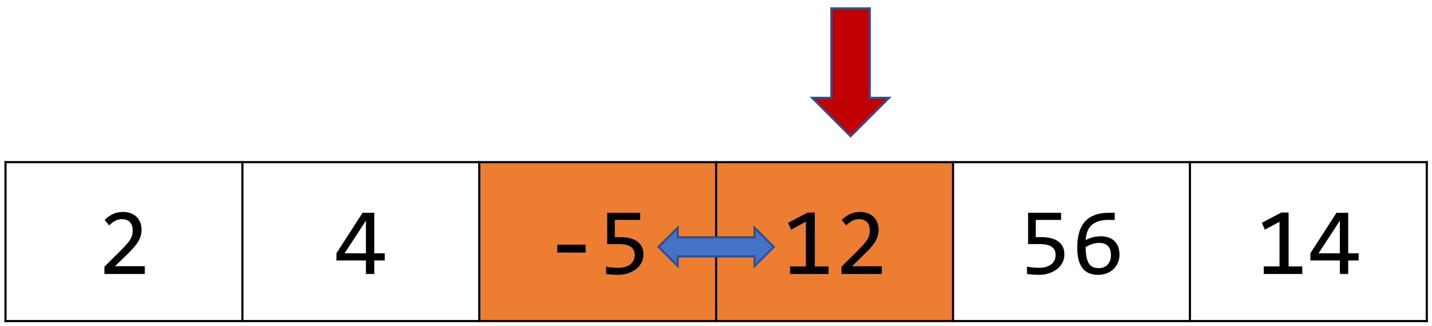
Let's write a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.



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

Bubble Sort

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



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

Bubble Sort

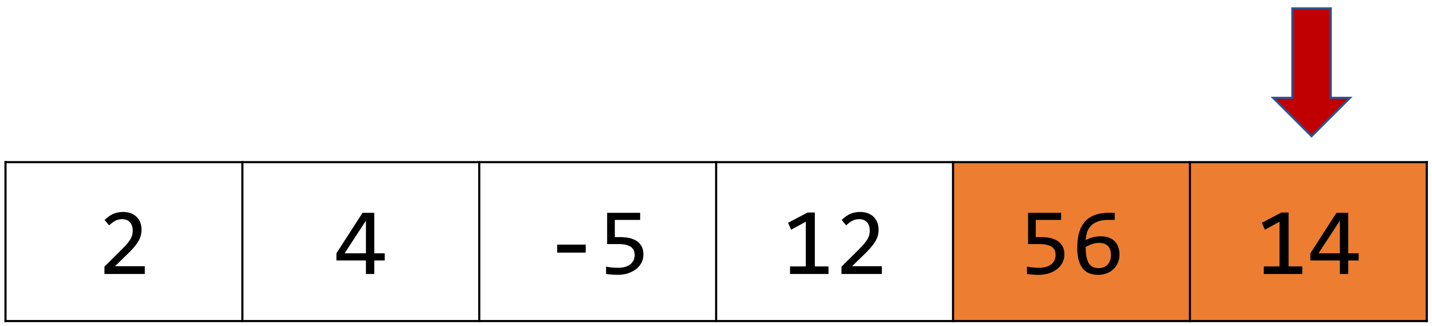
Let’s write a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.

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

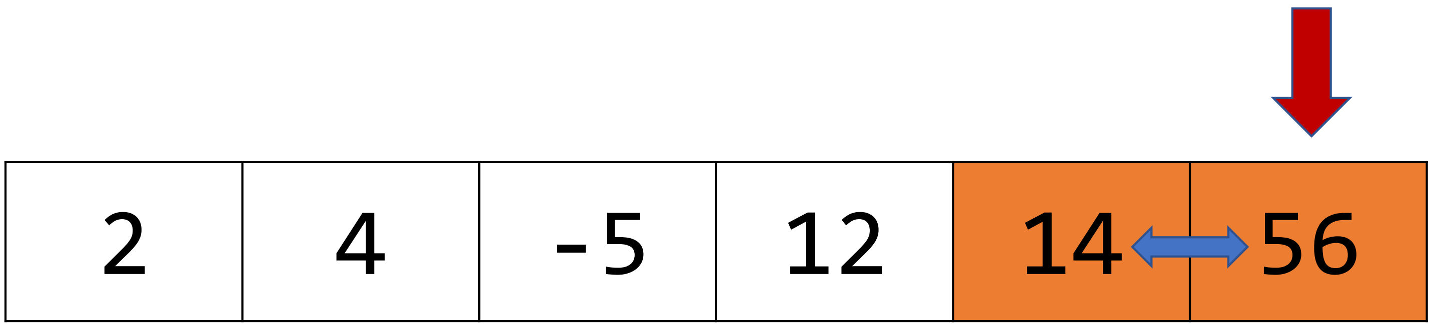
Let's write a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.



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

Bubble Sort

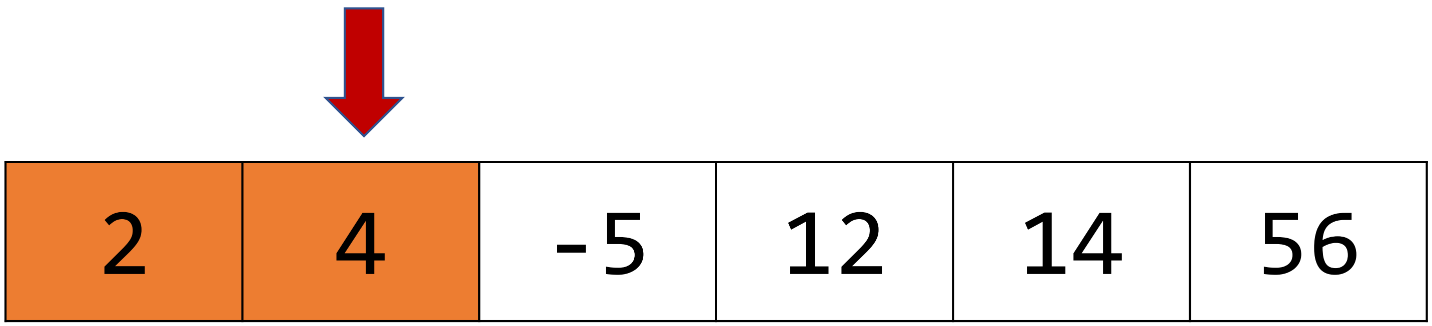
Let’s write a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.



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

Bubble Sort

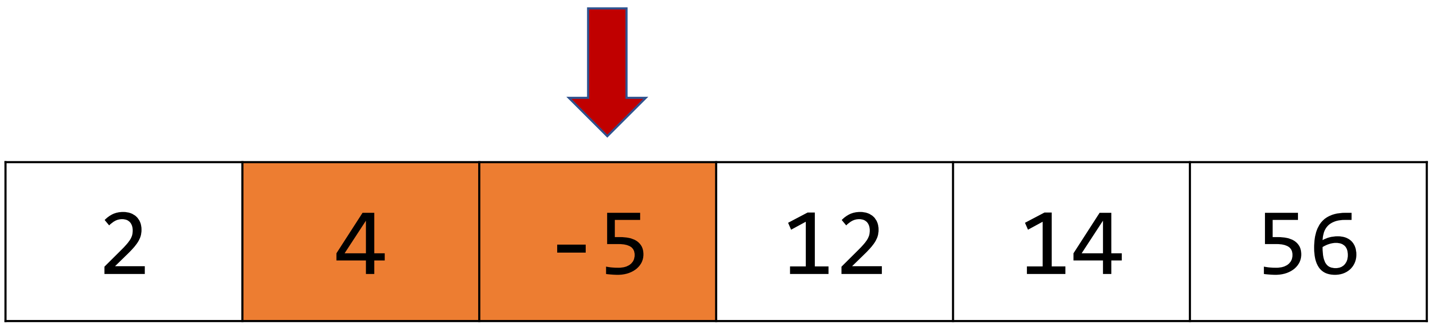
Let's write a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.



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

Bubble Sort

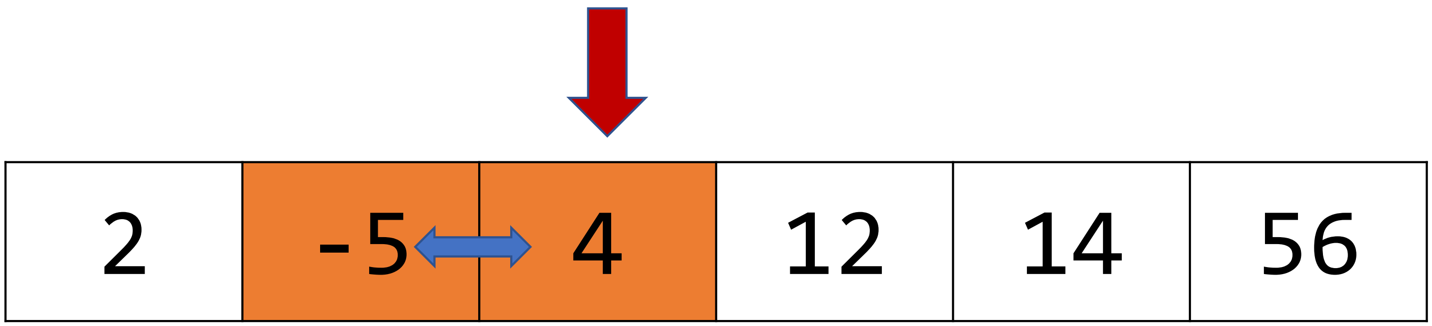
Let's write a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.



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

Bubble Sort

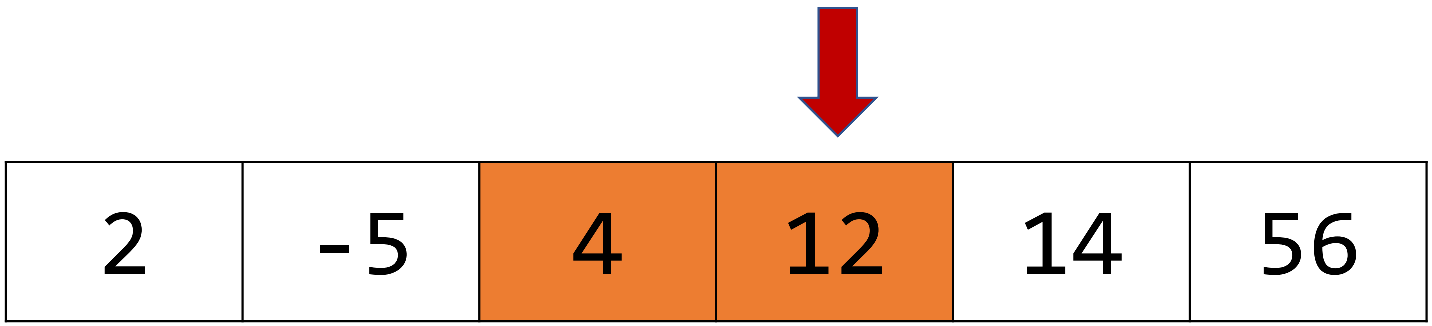
Let’s write a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.



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

Bubble Sort

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



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

Bubble Sort

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

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 **`bubble_sort_int`** to sort a list of integers using 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!

In general, bubble sort requires up to n - 1 passes to sort an array of length n, though it may end sooner if a pass doesn't swap anything.

Bubble Sort

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

-5	2	4	12	14	56

✅

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

Only two more passes are needed to arrive at the above. The first exchanges the 2 and the -5, and the second leaves everything as is.

Integer Bubble Sort

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

How can we make this function more generic? To start, this function always sorts in ascending order. What about other orders?

Integer Bubble Sort

```
void bubble_sort_int(int *arr, size_t n, bool ascending) {
    while (true) {
        bool swapped = false;
        for (size_t i = 1; i < n; i++) {
            if ((ascending && arr[i - 1] > arr[i]) ||
                (!ascending && arr[i] > arr[i - 1])) {
                swap(&arr[i - 1], &arr[i], sizeof(int));
                swapped = true;
            }
        }
        if (!swapped) {
            return;
        }
    }
}
```

We can add parameters, but they only help so much. What about other orders we can't anticipate? (odd-before-even, etc.)

Integer Bubble Sort

```
void bubble_sort_int(int *arr, size_t n) {
    while (true) {
        bool swapped = false;
        for (size_t i = 1; i < n; i++) {
            if (should_swap(arr[i - 1], arr[i])) {
                swap(&arr[i - 1], &arr[i], sizeof(int));
                swapped = true;
            }
        }
        if (!swapped) {
            return;
        }
    }
}
```

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

<u>Key Idea:</u> have the caller pass a function as a parameter that takes two `int`s and tells us whether we should swap them.

Integer Bubble Sort

```
void bubble_sort_int(int *arr, size_t n, type?? should_swap) {
    while (true) {
        bool swapped = false;
        for (size_t i = 1; i < n; i++) {
            if (should_swap(arr[i – 1], arr[i])) {
                swap(&arr[i - 1], &arr[i], sizeof(int));
                swapped = true;
            }
        }

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

Function Pointers

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

```
bool (*should_swap)(int, int)
```

Function Pointers

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

```
bool (*should_swap)(int, int)
```

Return type (bool)

Function Pointers

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

```
bool (*should_swap)(int, int)
```

Function pointer name
(should_swap)

Function Pointers

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

```
bool (*should_swap)(int, int)
```

Function parameters
(two ints)

Function Pointers

Here's the general variable type syntax:

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

Integer Bubble Sort

```
void bubble_sort_int(int *arr, size_t n, bool (*should_swap)(int, int)) {
    while (true) {
        bool swapped = false;
        for (size_t i = 1; i < n; i++) {
            if (should_swap(arr[i - 1], arr[i])) {
                swap(&arr[i - 1], &arr[i], sizeof(int));
                swapped = true;
            }
        }
        if (!swapped) {
            return;
        }
    }
}
```

Function Pointers

```
bool sort_ascending(int first_num, int second_num) {
    return first_num > second_num;
}

int main(int argc, char *argv[]) {
    int nums[] = {4, 2, -5, 1, 12, 56};
    int nums_count = sizeof(nums) / sizeof(nums[0]);
    bubble_sort_int(nums, nums_count, sort_ascending);
    ...
}
```

bubble_sort_int is written generically. When someone imports our function into their program, they will call it specifying the sort ordering they want that time.

Function Pointers

```
bool sort_descending(int first_num, int second_num) {
    return first_num < second_num;
}

int main(int argc, char *argv[]) {
    int nums[] = {4, 2, -5, 1, 12, 56};
    int nums_count = sizeof(nums) / sizeof(nums[0]);
    bubble_sort_int(nums, nums_count, sort_descending);
    ...
}
```

bubble_sort_int is written generically. When someone imports our function into their program, they will call it specifying the sort ordering they want that time.

Function Pointers

```
bool sort_abs(int first_num, int second_num) {
    return abs(first_num) < abs(second_num);
}

int main(int argc, char *argv[]) {
    int nums[] = {4, 2, -5, 1, 12, 56};
    int nums_count = sizeof(nums) / sizeof(nums[0]);
    bubble_sort_int(nums, nums_count, sort_abs);
    ...
}
```

bubble_sort_int is written generically. When someone imports our function into their program, they will call it specifying the sort ordering they want that time.

Function Pointers

- Passing a non-function as a parameter allows us to pass data around our program.
- When writing a generic function, if we don’t know how to do something and the decision about what to do should be left to the client, we can ask them to pass in a function parameter that can do it for us.
- Also called a "callback" function – function "calls back to" into caller code.
  - **Function writer:** writes generic algorithmic functions, relies on caller-provided data
  - **Function caller**: knows the data, doesn’t care how the algorithm is implemented

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.

```
int scandir(const char *dirp, struct dirent ***namelist,
            int (*filter)(const struct dirent *),
            int (*compar)(const struct dirent **, const struct dirent **));
```

- **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.

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

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)(int, int)
```

Integer Bubble Sort

```
void bubble_sort_int(int *arr, size_t n, int (*cmp_fn)(int, int)) {
    while (true) {
        bool swapped = false;
        for (size_t i = 1; i < n; i++) {
            if (cmp_fn(arr[i - 1], arr[i]) > 0) {
                swap(&arr[i - 1], &arr[i], sizeof(int));
                swapped = true;
            }
        }

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

Function Pointers

```
// 0 if equal, neg if first before second, pos if second before first
int sort_descending(int first_num, int second_num) {
    return second_num - first_num;
}

int main(int argc, char *argv[]) {
    int nums[] = {4, 2, -5, 1, 12, 56};
    int nums_count = sizeof(nums) / sizeof(nums[0]);
    bubble_sort_int(nums, nums_count, sort_descending);
    ...
}
```