CS107 Lecture 12
C Generics and Function Pointers

Reading: K&R 5.11

Going Generic: swap_ends

Let's write an **int**-specific version of a function called **swap_ends_int** that **accepts an array** and **exchanges its two endpoints**.

```
int main(int argc, char *argv[]) {
   int nums[] = {7, 2, 3, 4, 5, 6, 1};
   size_t len = sizeof(nums) / sizeof(nums[0]);
   swap_ends_int(nums, len);
   printf("nums[0] = %d, nums[%zu] = %d\n",
          nums[0], len - 1, nums[len - 1]);
   return 0;
} // should print nums[0] = 1, nums[6] = 7

void swap_ends_int(int arr[], size_t len) {
   int tmp = arr[0];
   arr[0] = arr[len - 1];
   arr[len - 1] = tmp;
}
```

Algorithmically straightforward.

We, however, have already written a generic **swap** routine that can replace the three-statement implementation of **swap_ends_int**.

Before we do that, let's say some smart things:

- **arr[0]** is really ***arr**
- **arr[len - 1]** is really ***(arr + len - 1)**. Restated, this computes the address that is **(len - 1)** integers—or rather **(len - 1) * sizeof(int)** bytes—**beyond** the array's base and **dereferences it to access the last element**.
- **Array notation is syntactic sugar for pointer notation**. We now understand even more so how elements in an array are located.

Going Generic: swap_ends

Let's write an **int**-specific version of a function called **swap_ends_int** that **accepts an array** and **exchanges its two endpoints**.

```
int main(int argc, char *argv[]) {
   int nums[] = {7, 2, 3, 4, 5, 6, 1};
   size_t len = sizeof(nums) / sizeof(nums[0]);
   swap_ends_int(nums, len);
   printf("nums[0] = %d, nums[%zu] = %d\n",
          nums[0], len - 1, nums[len - 1]);
   return 0;
}

void swap_ends_int(int arr[], size_t len) {
   swap(&arr[0], &arr[len - 1], sizeof(arr[0]));
}
```

Truth be told, the win here is largely an **academic one**, and most would be **perfectly thrilled** with the original three-statement version.

Still, it's a **victory** to understand how the **raw bit pattern exchange** managed by **swap** exchanges the **sizeof(int)**-byte figures on behalf of **swap_ends_int**.

There's very little about the implementation that feels all that **int**-specific.

What would **swap_ends_short**, **swap_ends_float**, and **swap_ends_string** look like?

Going Generic: swap_ends

What would **swap_ends_short**, **swap_ends_float**, and **swap_ends_string** look like?

```
void swap_ends_int(int arr[], size_t len) {
   swap(&arr[0], &arr[len - 1], sizeof(arr[0]));
}

void swap_ends_short(short arr[], size_t len) {
   swap(&arr[0], &arr[len - 1], sizeof(arr[0]));
}

void swap_ends_float(float arr[], size_t len) {
   swap(&arr[0], &arr[len - 1], sizeof(arr[0]));
}

void swap_ends_string(const char *arr[], size_t len) {
   swap(&arr[0], &arr[len - 1], sizeof(arr[0]));
}
```

All four versions are one-line wrappers around a call to **swap**, and the **three expressions passed as prameters are precisely the same** in all cases.

Of course, the **sizeof** expressions produce **different numbers**. And the **pointer arithmetic** associated with **&arr[len – 1]** is computed by **scaling len – 1** by the **relevant element size**.

Surely there's some way to **unify** these four implementations to a **single code base**, much as we did for **swap**. Right?

Going Generic: swap_ends

Here's our first attempt at a **fully generic** implementation of **swap_ends**.

```
void swap_ends(void *arr, size_t len, size_t size) {
   swap(arr, arr + len - 1, size);
}
```

What works?

The **base address** of the array—whatever the element type—can be accepted as a **void***.

And provided the **element size** is passed through as an additional parameter to **swap_ends**, we know to pass that value on verbatim as the third parameter to **swap**.

What doesn't?

The expression **arr** + **len** - **1** is **invalid**.

Since **arr** is a **void***, it can't be dereferenced and can't participate in pointer arithmetic. Pointer arithmetic requires a known pointed-to type so the compiler can scale addresses by **sizeof(type)**.

Because that information is missing, expressions like ***arr**, **arr[0]**, and **arr + len – 1** don't make sense an pure C-compliant compiler.

Going Generic: swap_ends

Let's build up a **generic** implementation of **swap_ends** that truly works.

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

Numerically, the second argument to **swap** needs to be the address that is **literally** **(len – 1) * size** bytes beyond **arr**.

The trick—standard in generic C code and referred to as the "**char *** hack" in CS107 circles—is to **cast the leading address** to be a **char ***—**regardless of what it truly addresses**—so that pointer arithmetic is reduced to **byte-by-byte arithmetic**.

The expression **(char *) + (len – 1) * size** is of type **char ***, but **swap**'s second parameter is of type **void *** and therefore happy to accept it as a generic address.

Going Generic: swap_ends

Let's build up a **generic** implementation of **swap_ends** that truly works.

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

How should **swap_ends** be called? Check this out:

```
int numbers[] = {7, 2, 3, 4, 5, 6, 1};
size_t count = sizeof(numbers) / sizeof(numbers[0]);
swap_ends(numbers, count, sizeof(numbers[0]));
```

```
short primes[] = {23, 3, 5, 7, 11, 13, 17, 19, 2};
size_t count = sizeof(primes) / sizeof(primes[0]);
swap_ends(primes, count, sizeof(primes[0]));
```

```
const char *french[] = {"prie", "vous", "en", "je"};
size_t count = sizeof(french) / sizeof(french[0]);
swap_ends(french, count, sizeof(words[0]));
```

```
float math[] = {4.6692, 2.7183, 3.1416, 1.6180};
size_t count = sizeof(math) / sizeof(math[0]);
swap_ends(math, count, sizeof(math[0]));
```

```
typedef struct {
   int num;
   int denom;
} fraction;

fraction ratios[] = {{5, 7}, {11, 18}, {13, 27}};
size_t count = sizeof(ratios) / sizeof(ratios[0]);
swap_ends(ratios, count, sizeof(ratios[0]));
```

Generics Etude: `rotate`

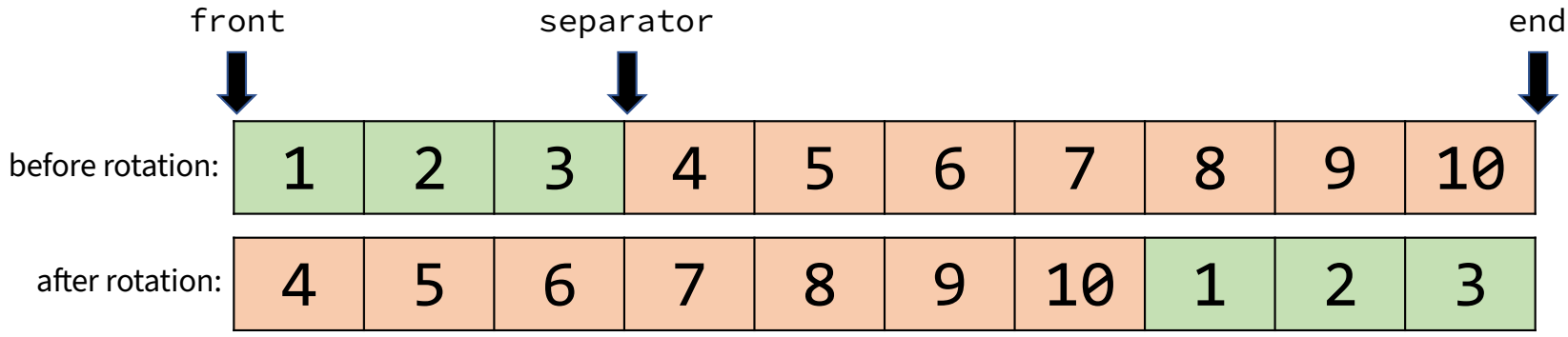
Let's implement a C generic that **rotates the byte range** **`[front, end)`** so the subrange **`[front, separator)`** moves to the end. All pointers reach into the same contiguous memory block, and relative byte order must otherwise be preserved.

Here's the general prototype.

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

Here's some client code and some pretty diagrams to illustrate how rotate works.

```
int array[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
rotate(array, array + 3, array + 10);
assert(array[0] == 4 && array[3] == 7 && array[7] == 1);
```



Here's Doris.

Generics Etude: `rotate`

Let's implement a C generic that **rotates the byte range** **`[front, end)`** so the subrange **`[front, separator)`** moves to the end. All three addresses point into the same contiguous memory block, and relative byte order must otherwise be preserved.

```
void rotate(void *front, void *separator, void *end) {
   assert(front <= separator && separator <= end);
   size_t width = (char *)end - (char *)front;
   size_t prefix_width = (char *)separator - (char *)front;
   size_t suffix_width = width - prefix_width;

   if (prefix_width == 0 || suffix_width == 0) return;

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

Here we compute the **`width`** of the entire array and **the sizes of the two figures** that travel as a unit.

`temp` is **just big enough to store a copy of the prefix** to be pushed to the rear of the array.

There are **three memory moves** here for the **same reason** there were three in the generic **`swap`**. The key difference here, of course, is that the two figures being exchanged are usually **different sizes**.

Sadness: there's a **logical flaw** with the second of the three **memcpy** calls because the **source and the destination might overlap**.

Introducing memmove

memmove copies a specified number of bytes from one memory location to another, **correctly handling scenarios where the source and destination ranges overlap**.

```
void *memmove(void *dst, const void *src, size_t len);
```

- **memmove** behaves just like **memcpy** but is defined to work properly **even when the source and destination regions overlap**. It copies data in a safe direction **to avoid overwriting data that has not yet been copied**.
- If you know the two regions **can't possibly overlap**, the **correct CS107 thing to do** is to call **memcpy** instead.
- **memmove** and **memcpy** each return whatever **dst** is. The return value is **almost always ignored**.
- **Hot Take**: **memmove** and **memcpy** both operate much like **strncpy**, except they **don't stop when they encounter a zero byte**. The number of bytes copied is dictated—in all cases—by the **value of the third argument**.

Quick Example: Here's a function that converts a Pascal string of length 255 or less to a C string. Pascal strings aren't terminated by a zero byte the way C strings are. Instead, they store their length in the 0^{th} position, and the characters themselves start at position 1. This version converts in place.

```
void pascal_to_c_string(char *s) {
   size_t len = s[0];
   memmove(s, s + 1, len);
   s[len] = '\0';
}
```

Generics Etude: `rotate`

Let's implement a C generic that **rotates the byte range** **`[front, end)`** so the subrange **`[front, separator)`** moves to the end. All three addresses point into the same contiguous memory block, and relative byte order must otherwise be preserved.

```
void rotate(void *front, void *separator, void *end) {
   assert(front <= separator && separator <= end);
   size_t width = (char *)end - (char *)front;
   size_t prefix_width = (char *)separator - (char *)front;
   size_t suffix_width = width - prefix_width;

   if (prefix_width == 0 || suffix_width == 0) return;

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

Unless the caller supplies bogus addresses, it's **impossible** for the material in between **`front`** and **`end`** to overlap with the memory set aside for **`temp`**.

That means **`memcpy`** is **perfectly valid for the first and third of the three moves**. And yes, while we could call **`memmmove`** anyway, you shouldn't call **`memmmove`** if **`memcpy`** works in all scenarios.

Whenever there's a **genuine threat of overlap** as there is with the second of the three moves, you must call **`memmmove`**.

Motivating Example: Bubble Sort

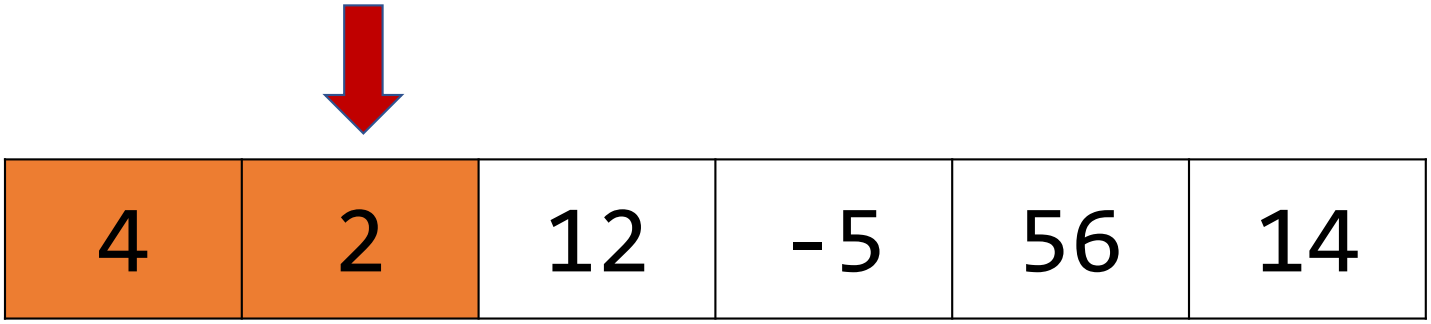
Let's implement a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.

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.

Motivating Example: Bubble Sort

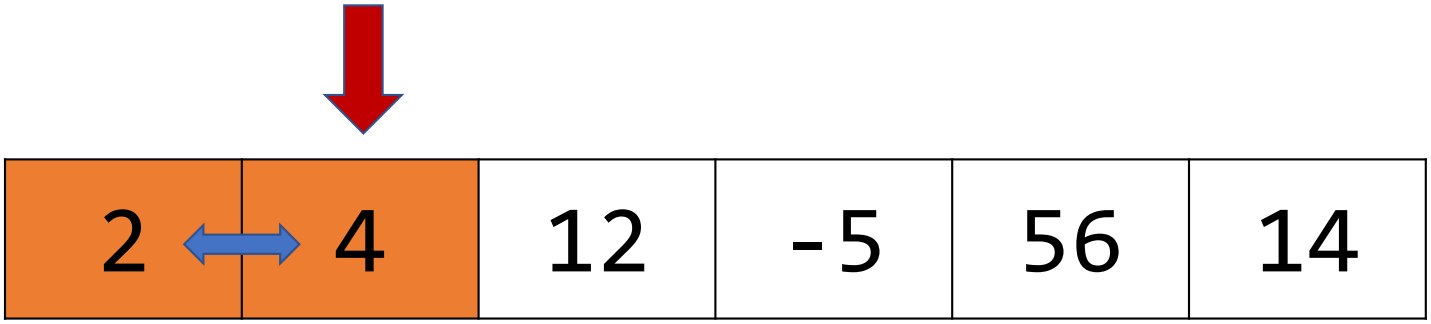
Let's implement a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.



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.

Motivating Example: Bubble Sort

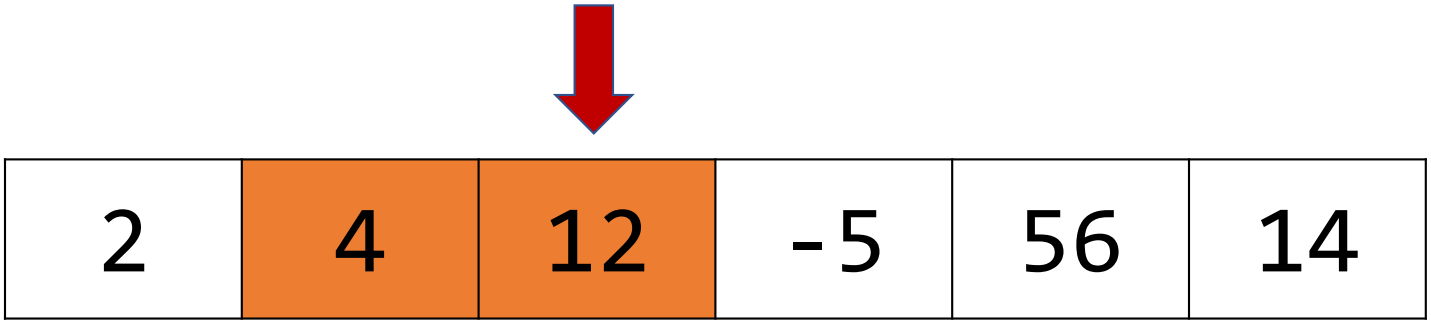
Let's implement a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.



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.

Motivating Example: Bubble Sort

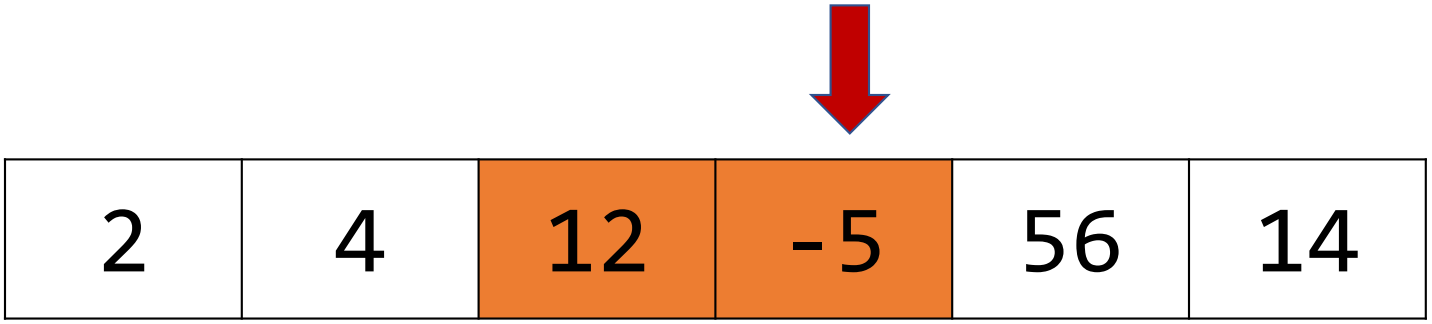
Let's implement a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.



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.

Motivating Example: Bubble Sort

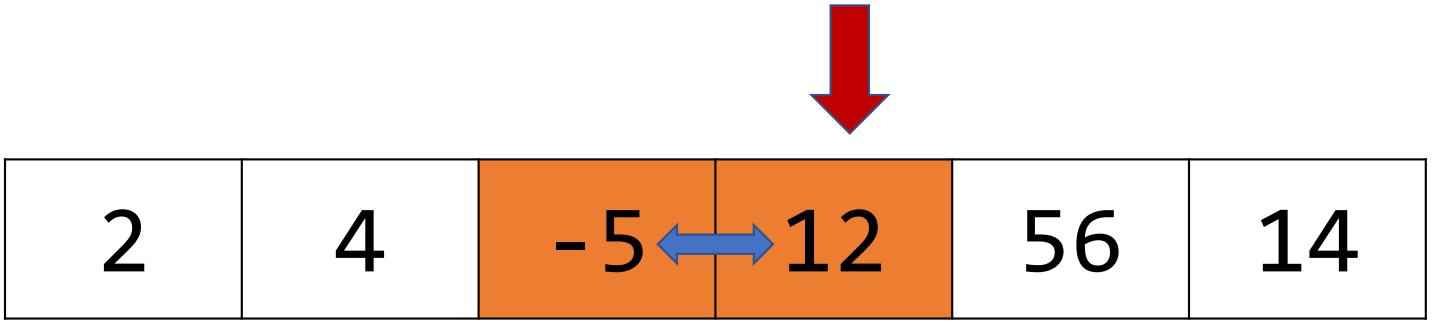
Let's implement a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.



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.

Motivating Example: Bubble Sort

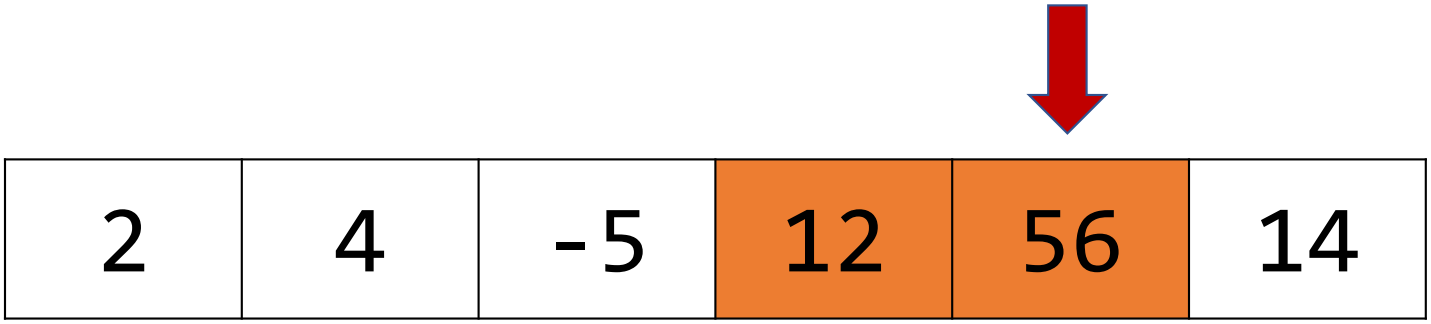
Let's implement a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.



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.

Motivating Example: Bubble Sort

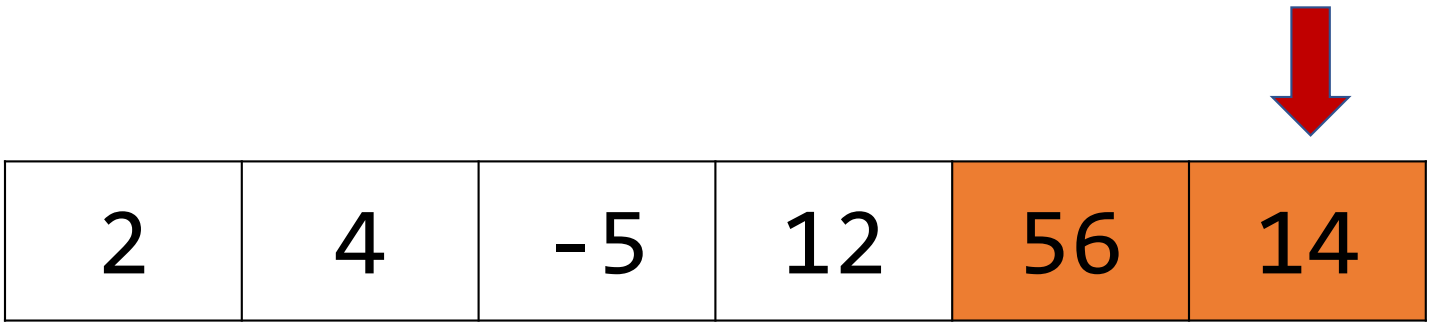
Let's implement a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.



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.

Motivating Example: Bubble Sort

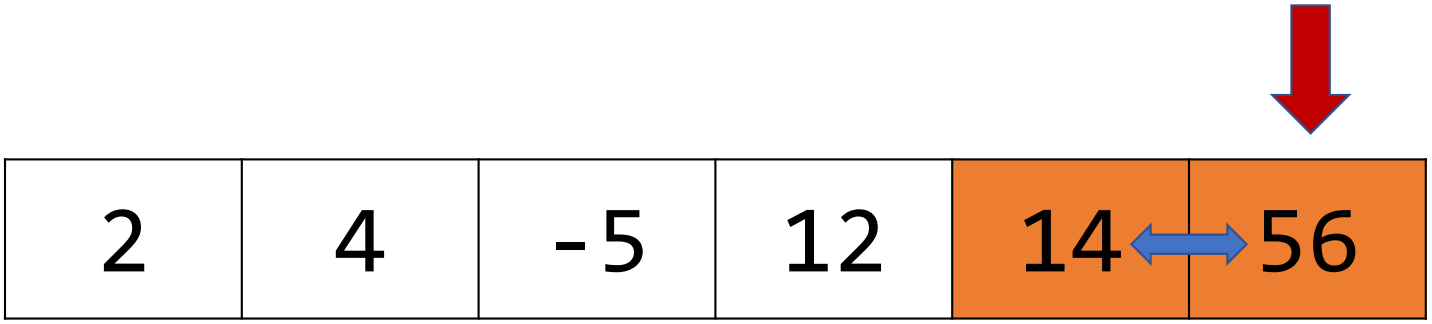
Let's implement a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.



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.

Motivating Example: Bubble Sort

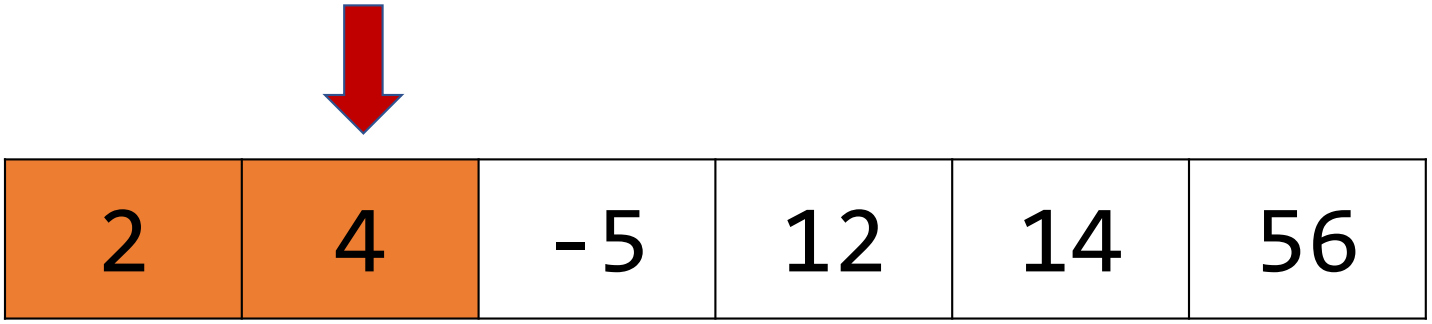
Let's implement a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.



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.

Motivating Example: Bubble Sort

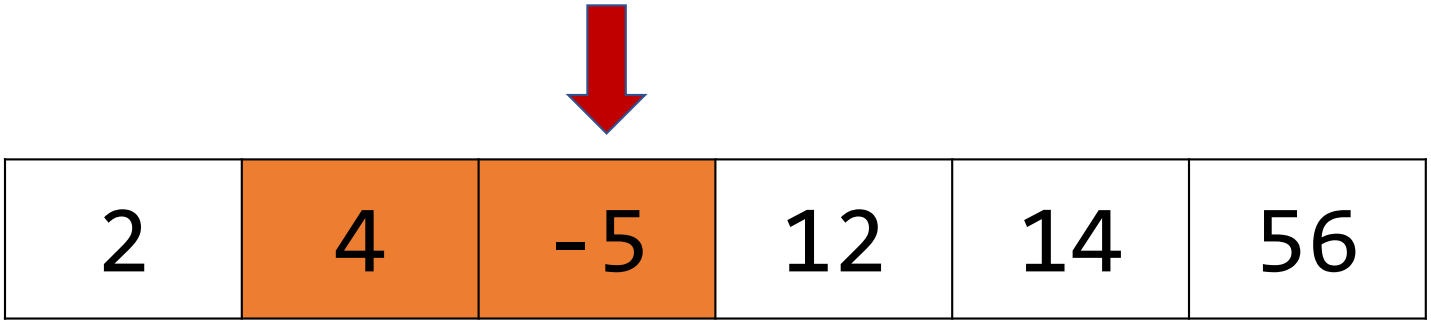
Let's implement a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.



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.

Motivating Example: Bubble Sort

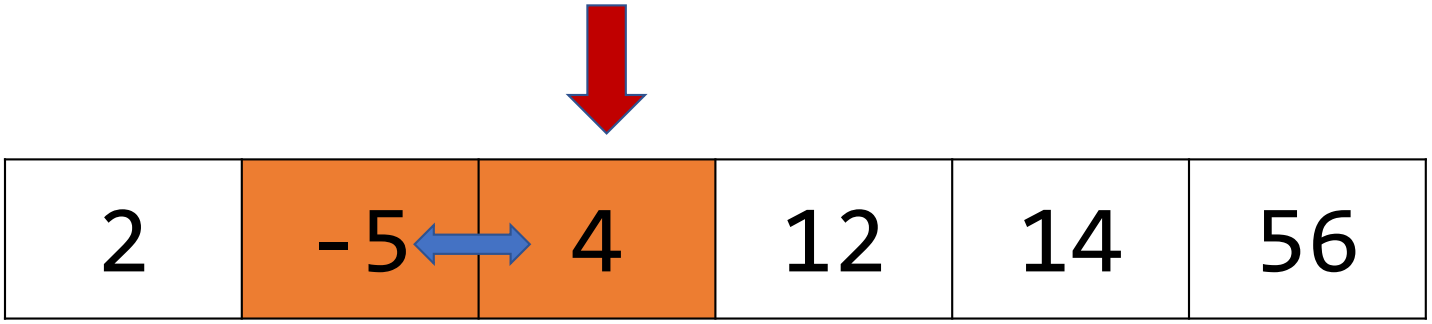
Let's implement a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.



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.

Motivating Example: Bubble Sort

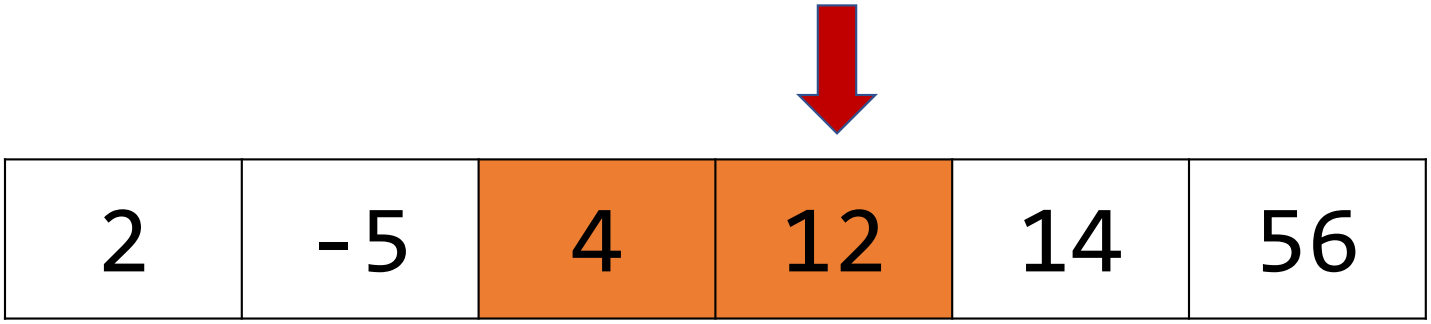
Let's implement a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.



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.

Motivating Example: Bubble Sort

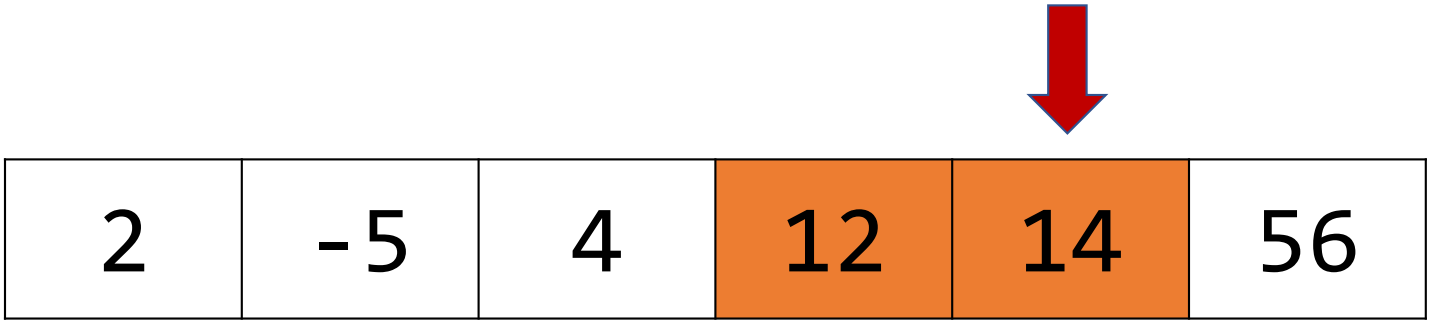
Let's implement a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.



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.

Motivating Example: Bubble Sort

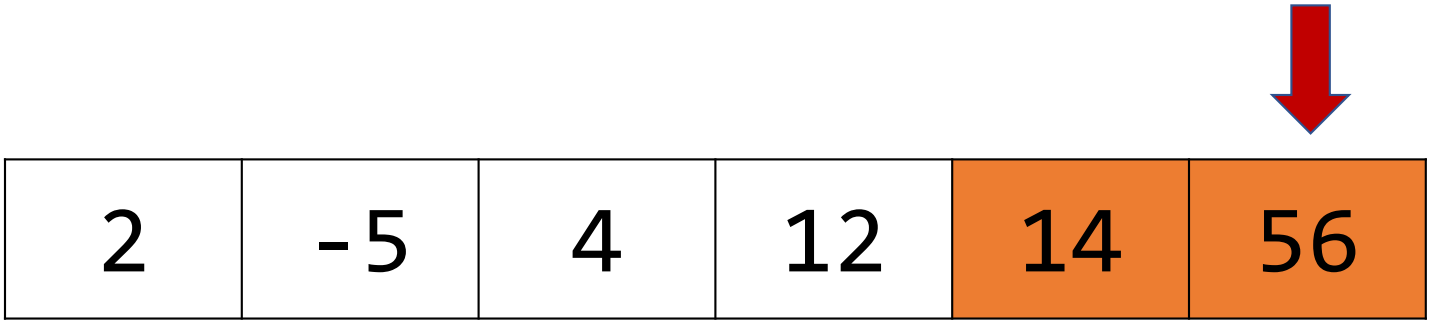
Let's implement a function **`bubble_sort_int`** to sort a list of integers using the **bubble sort algorithm**.



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.

Motivating Example: Bubble Sort

Let's implement 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.

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 at all.

Motivating Example: Bubble Sort

Let's implement 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 what you see above. The first of those two additional passes exchanges the 2 and the -5, and the second leaves everything as is.

Integer Bubble Sort: Going Generic

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

```
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;
    }
}
```

The **`while (true)`** loop gives the impression that we could in theory loop **forever**, but that's not true.

The **largest element bubbles to the end of the array** via the **`while`** loop's first iteration. That's **guaranteed**.

The second largest bubbles to sit **to the left of the largest element** by the end of the **`while`** loop's second iteration.

A la induction, the smallest element **occupies index 0** after at most **`n - 1`** iterations.

Can we generalize this version just a bit?

Integer Bubble Sort: Going Generic

Here's a slightly more elaborate (though still **`int`**-specific) implementation, that allows the client to choose **ascending versus descending order**.

```
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 - 1] < arr[i])) {
            swap(&arr[i - 1], &arr[i], sizeof(int));
            swapped = true;
         }
      }
      if (!swapped) return;
   }
}
```

Here we rely on **a single Boolean value**—a **`bool`** called **`ascending`**—to **brute-force decide** between one of two comparisons.

What about **different orderings**? Odd before even? Distance from 0?

We could replace the **`bool`** with an **`int`** code or some enumerated type **to dispatch between one of several comparisons** instead of just two, but that **won't scale very well**.

Integer Bubble Sort: Going Generic

What we really want is a version **where the decision to swap or not** is made by some predicate function, here called **`should_swap`**.

```
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;
   }
}
```

Here, **`should_swap`** is a standalone function that returns **`true`** if and only if the two elements being compared **are out of order**.

The problem is that **`should_swap`** has no access to **call-specific intent**. Its behavior is **fixed for the entire program**, so there's no easy way to sort one **`int`** array from low to high, another by decreasing absolute value, and a third using some other custom ordering.

A well-decomposed function **isn't the same** as a generic one.

Integer Bubble Sort: Going Generic

The solution here is **to require the client pass a function** that accepts two **`int`**s and returns **`true`** if and only if the two are out of order.

```
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;
   }
}
```

The third parameter is of type **function pointer**, which means it **accepts a function** that, in this case, itself takes two **`int`**s and returns a **`bool`**.

A function name **is internally recognized as the address of the first assembly code instruction** associated with the function. That's why a function name can be used as a **function pointer**.

The name of any function can be passed in as the third parameter here, provided it takes two **`int`**s and returns a **`bool`**.

Integer Bubble Sort: Going Generic

Look at just how versatile **`bubble_sort_int`** is now.

```
bool sort_ascending(int one, int two) {
   return one > two;
}

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

```
bool sort_descending(int one, int two) {
   return one < two;
}

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

```
bool sort_abs(int one, int two) {
   return abs(one) > abs(two);
}

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

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 require the client to pass a function** that can do it for us.

Functions passed as parameters are often called **callback functions**, because they **call back** into **client-specific, context-aware** code.

Next time, we'll work to **upgrade this** to work for **any element type**.