CS107, Lecture 9
C Generics – Function Pointers

Reading: K&R 5.11

Learning Goals

- Learn how to write C code that works with any data type.
- Learn how to pass functions as parameters
- Learn how to write functions that accept functions as parameters

Lecture Plan


```
cp -r /afs/ir/class/cs107/lecture-code/lect09 .
```

Lecture Plan


```
cp -r /afs/ir/class/cs107/lecture-code/lect09 .
```

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.

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.

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.

Void * Pitfalls

- **void** *s are powerful, but dangerous - C cannot do as much checking!
- E.g. with **int**, C would never let you swap *half* of an int. With **void *s**, this can happen!

```
int x = 0xffffffff;
int y = 0xeeeeeeee;
swap(&x, &y, sizeof(short));

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

memset

memset is a function that sets a specified amount 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


```
cp -r /afs/ir/class/cs107/lecture-code/lect09 .
```

Bubble Sort

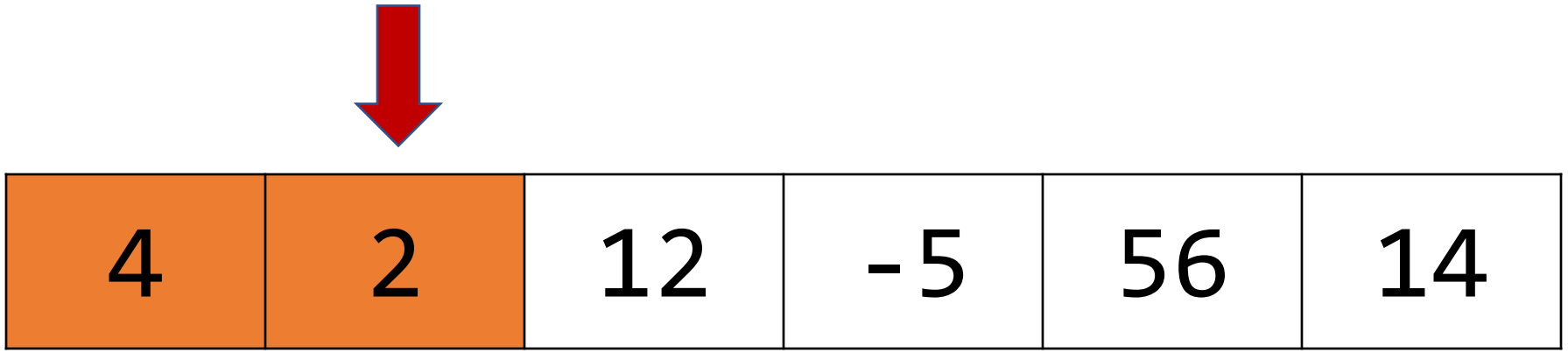
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.

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

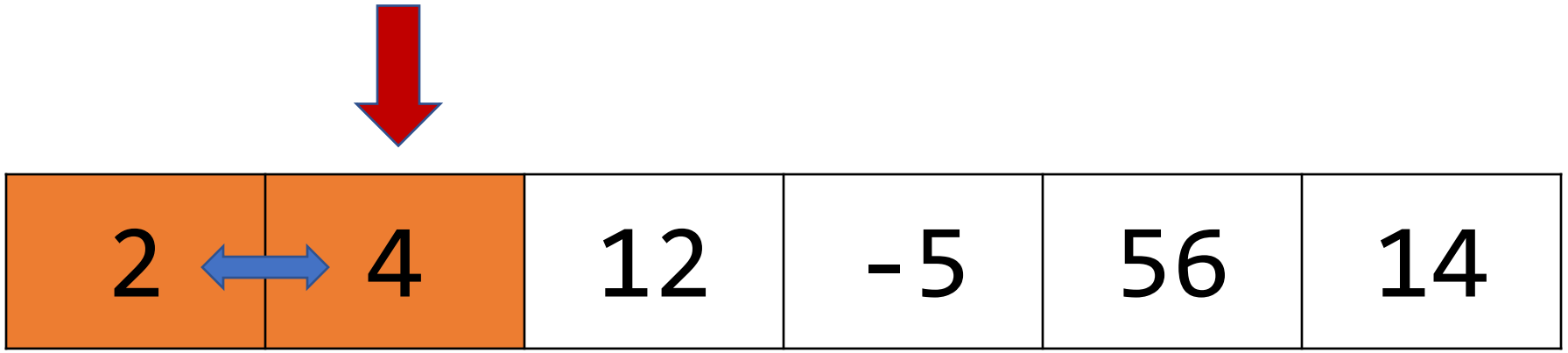
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- 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

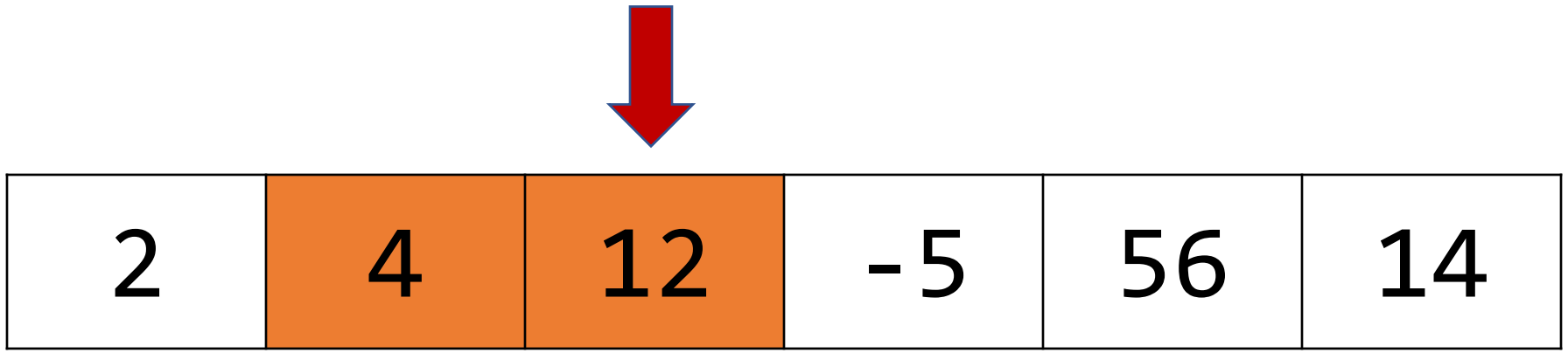
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- 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

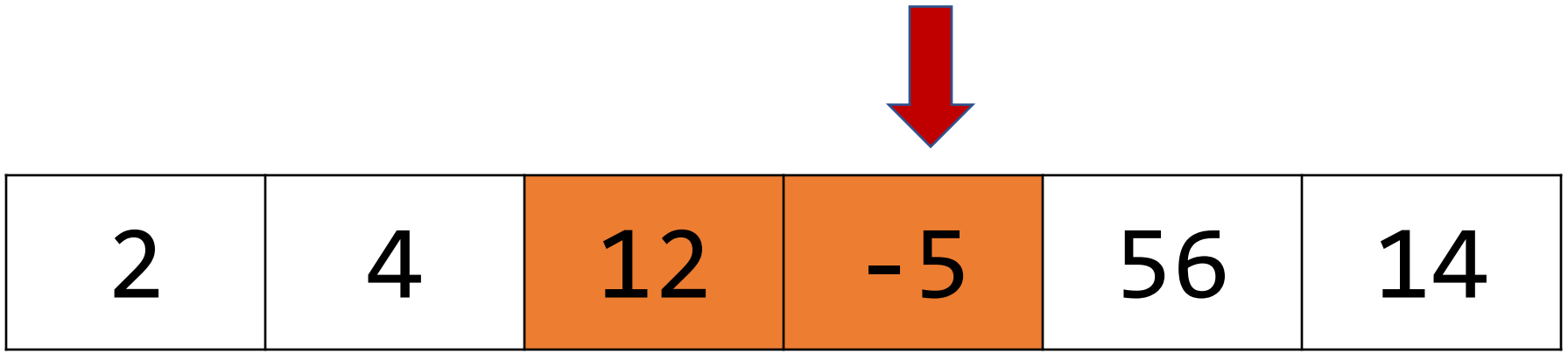
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- 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

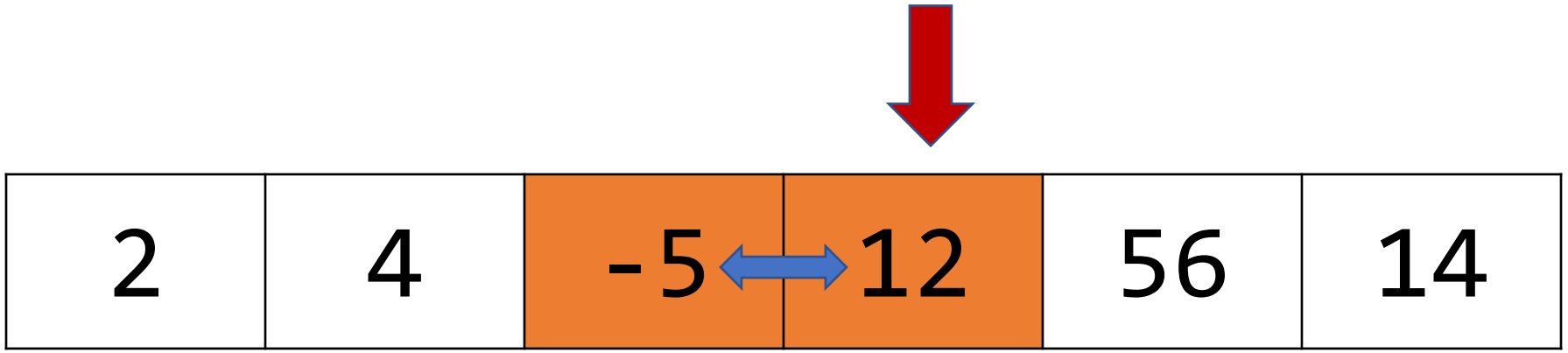
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- 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

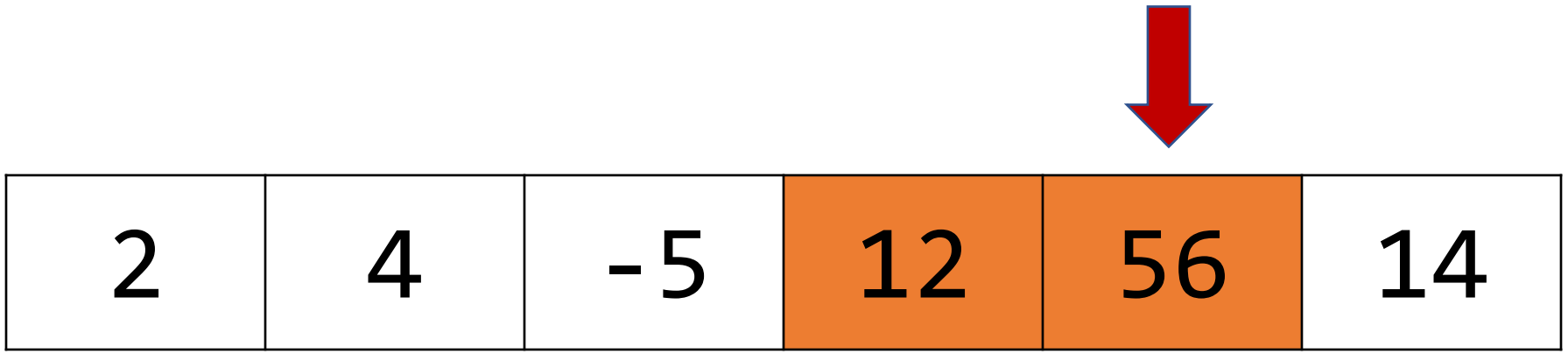
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- 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

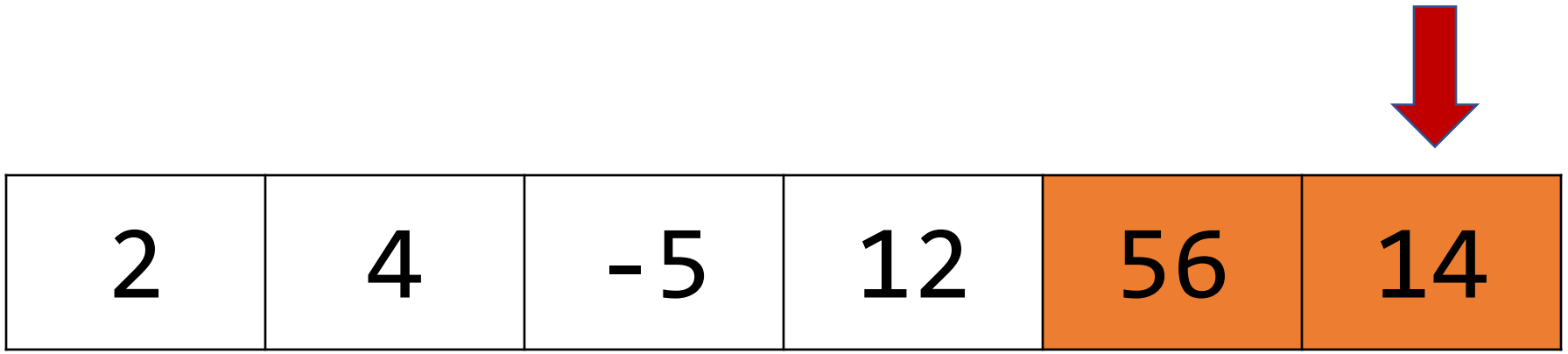
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- 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

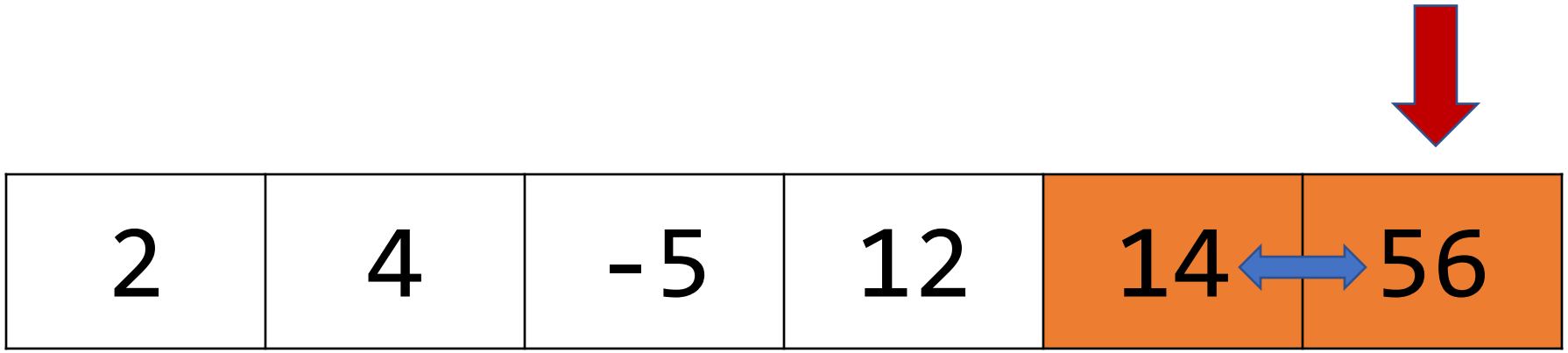
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- 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

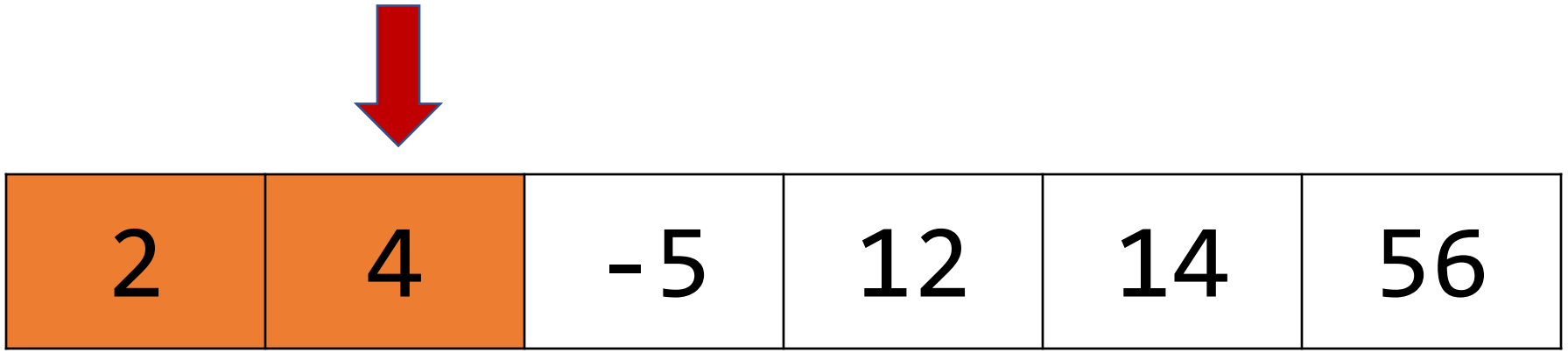
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- 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

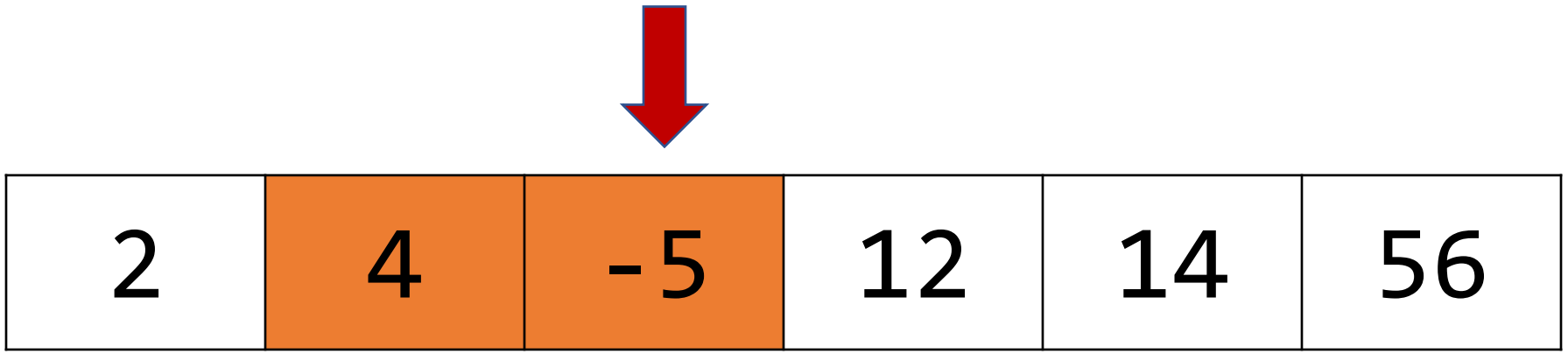
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- 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

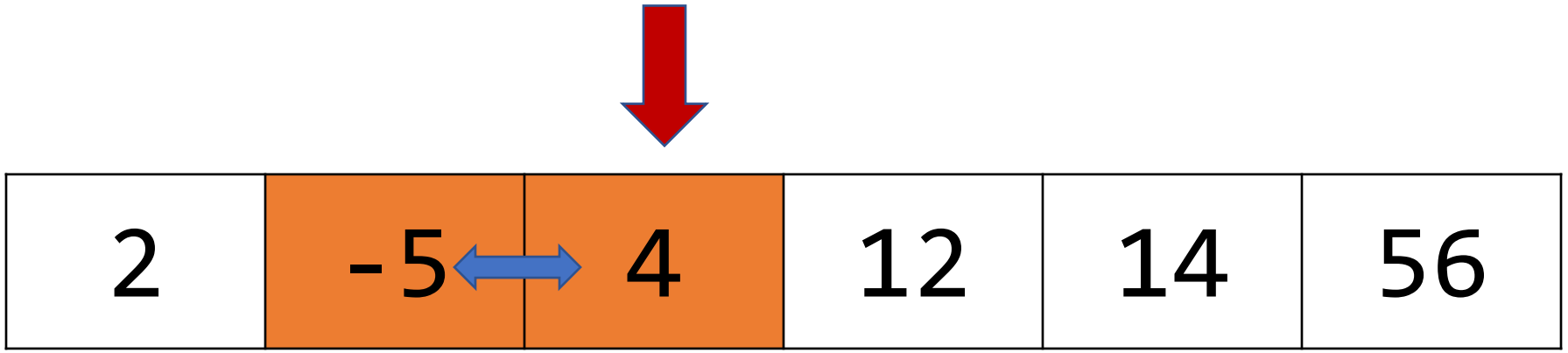
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- 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

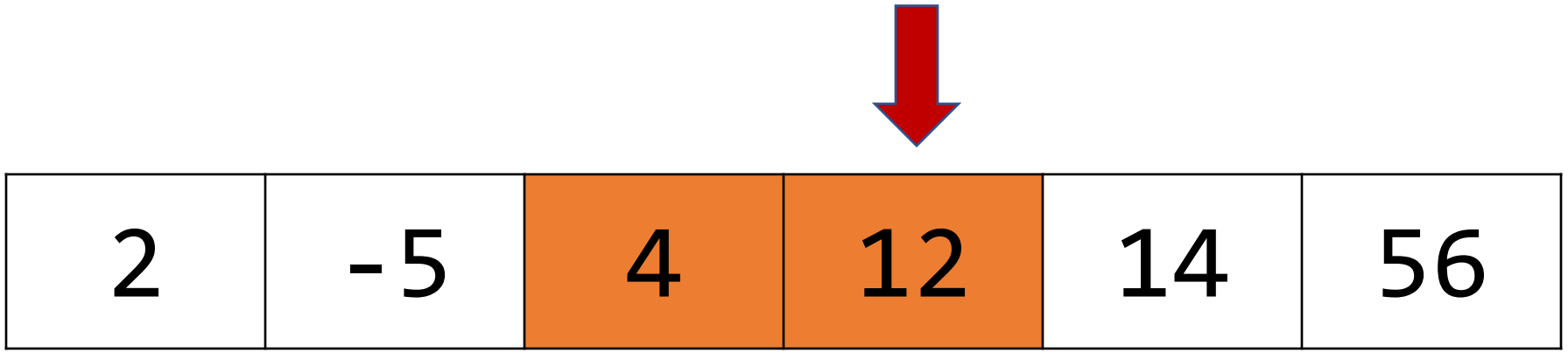
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- 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

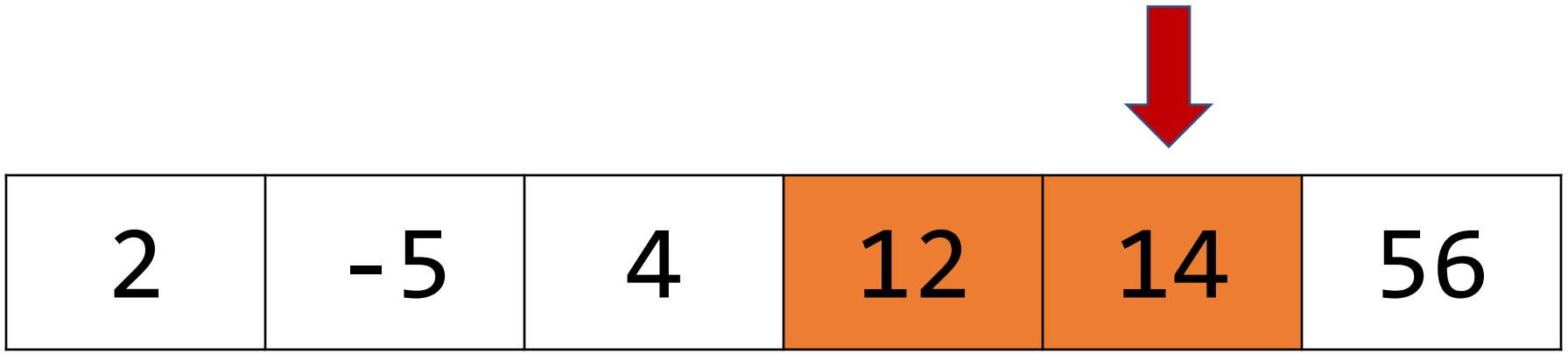
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- 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

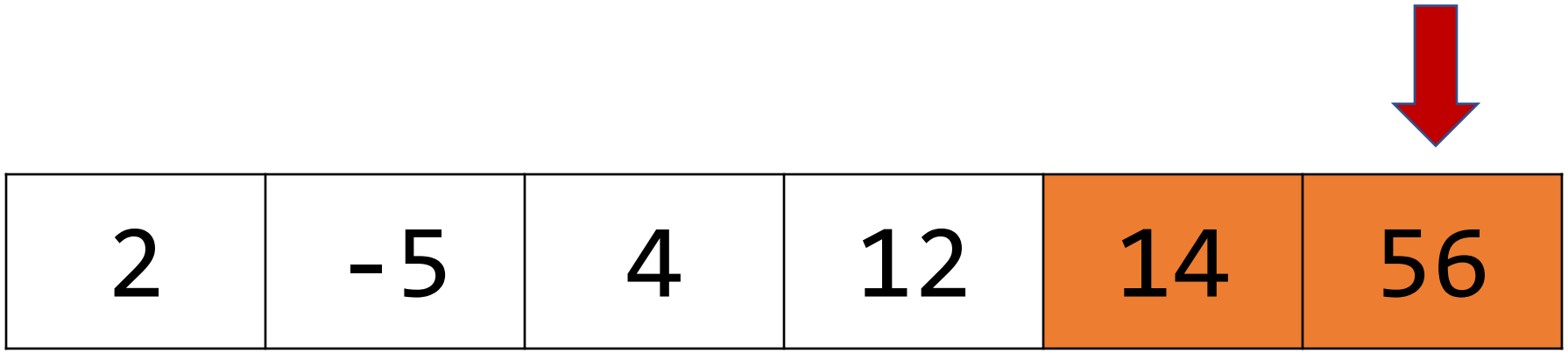
- Let's write a function to sort a list of integers. We'll use the **bubble sort algorithm**.



- 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 (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;
        }
    }
}
```

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


```
cp -r /afs/ir/class/cs107/lecture-code/lect09 .
```

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.

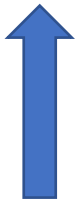
```
bool (*compare_fn)(void *a, void *b)
```

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[] = {"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.

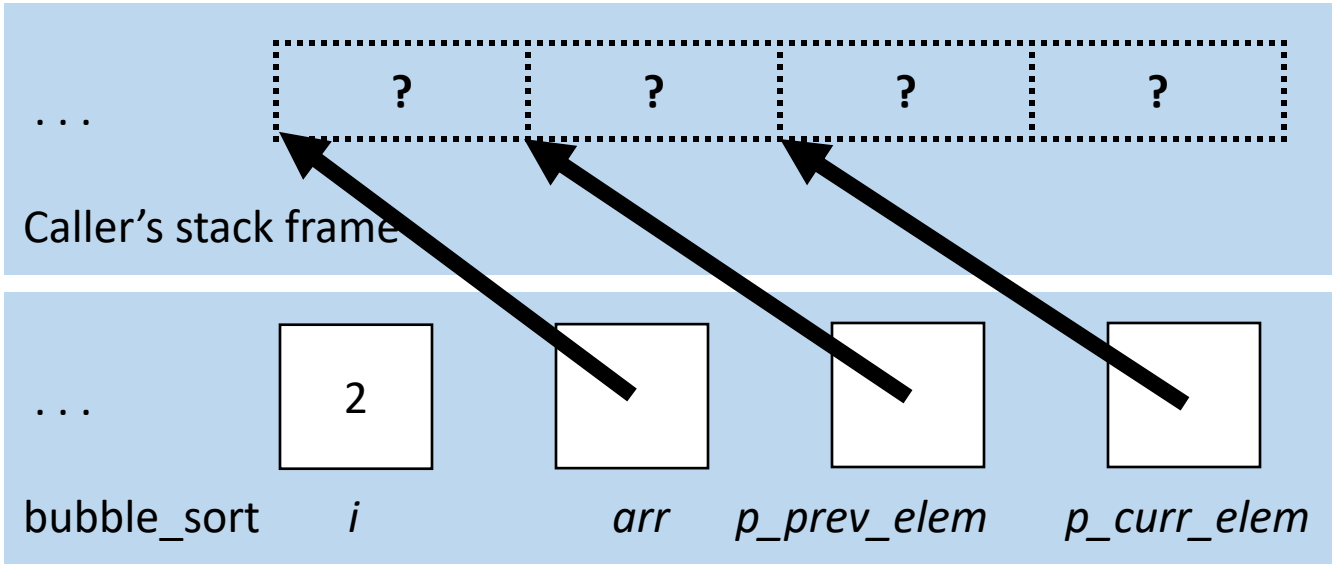
Bubble Sort

```
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

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

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.

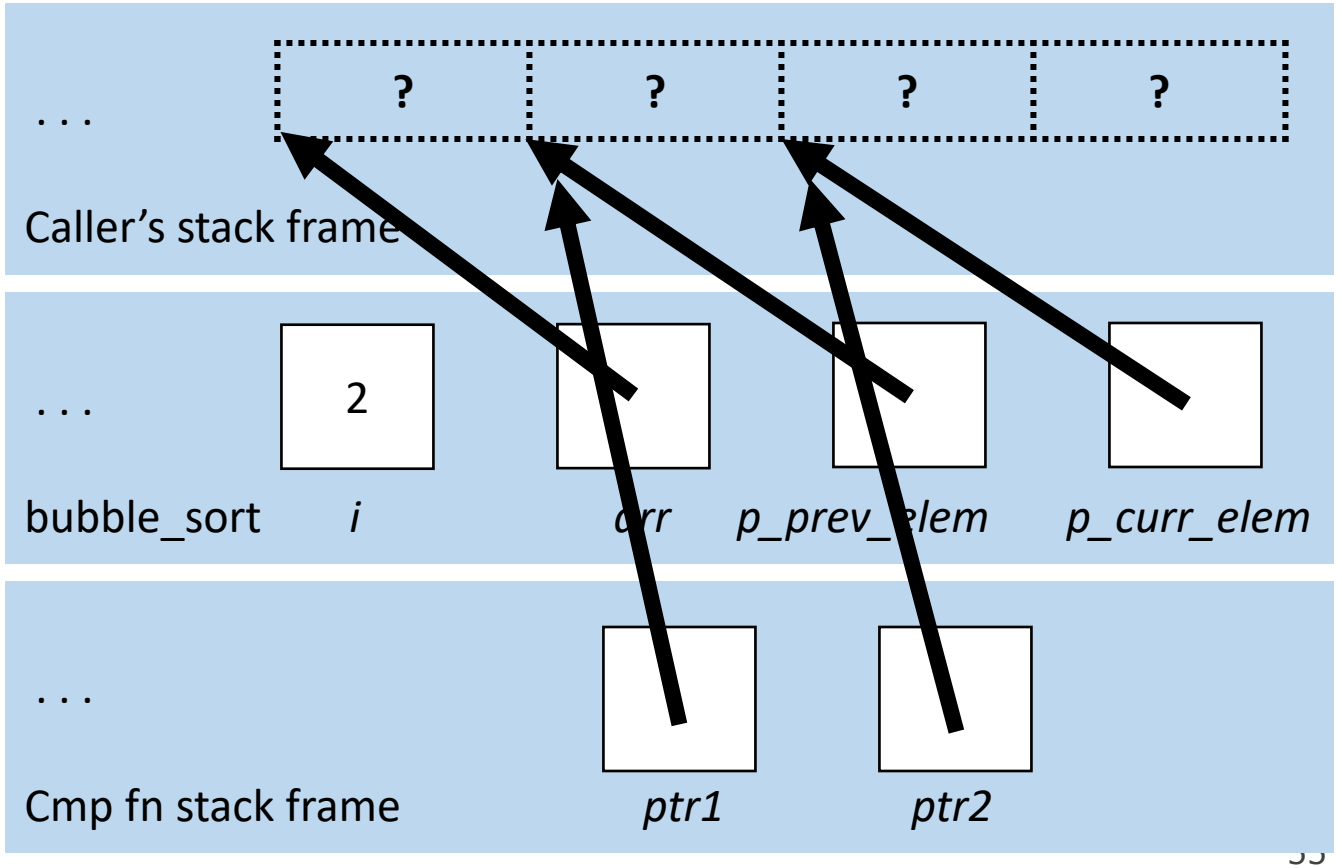
Function Pointers

```
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;
        }
    }
}
```

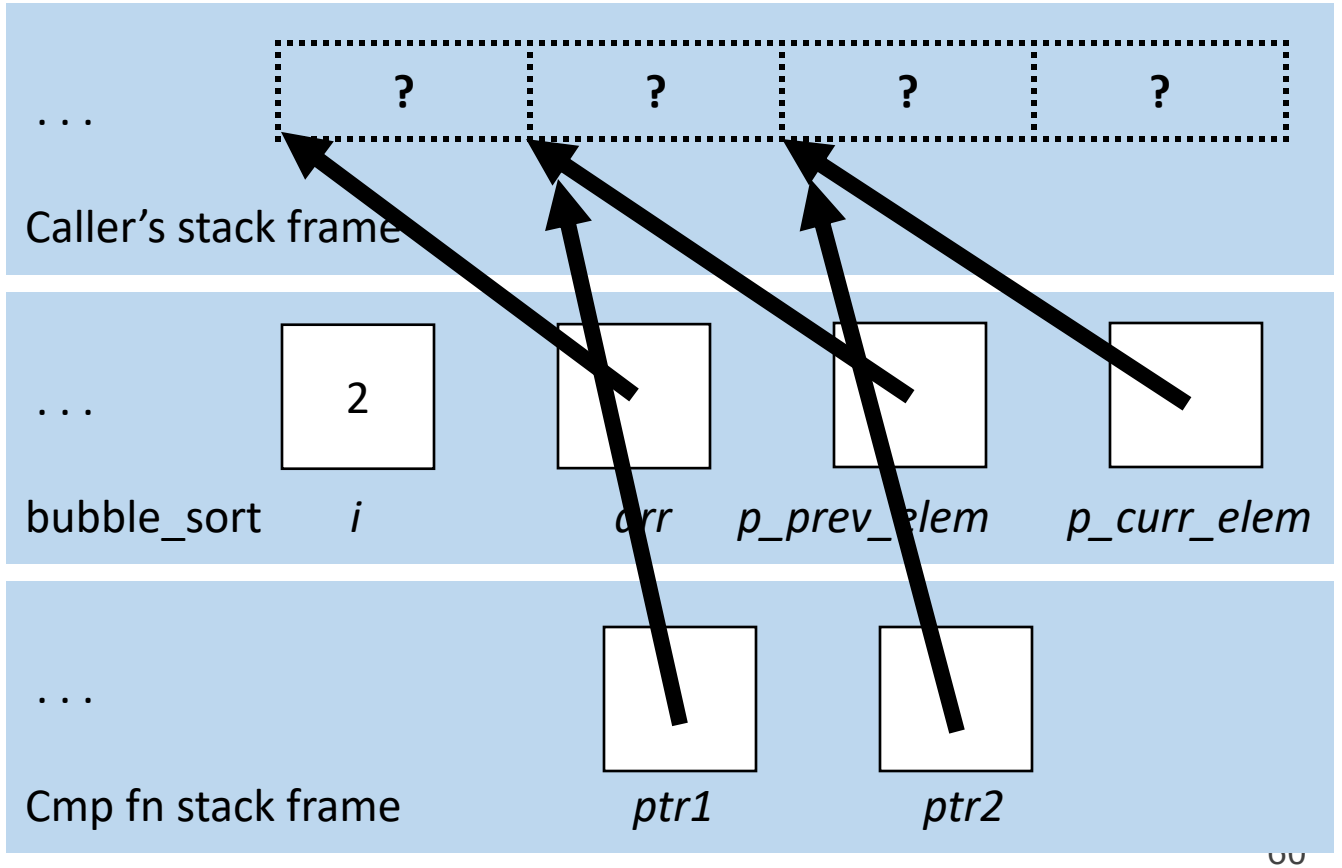
Comparison Functions

- **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


```
cp -r /afs/ir/class/cs107/lecture-code/lect09 .
```

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.

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.

Recap

- 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*.

Generics Overview

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

Lecture Plan


```
cp -r /afs/ir/class/cs107/lecture-code/lect09 .
```

Recall: Vulnerabilities Equities Process

The US federal government is one of the largest discoverers and purchasers of 0-day vulnerabilities.

It follows a “Vulnerabilities Equities Process” (VEP) to determine which vulnerabilities to responsibly disclose and which to keep secret and use for espionage or intelligence gathering.

VEP claimed in 2017 that 90% of vulnerabilities are disclosed, but it is not clear what the impact or scope of the un-disclosed 10% of vulnerabilities are.

Case Study: EternalBlue

2012-2017: NSA secretly stores the EternalBlue Microsoft vulnerability and uses it to spy on both US and non-US citizens.

early 2017: EternalBlue stolen by hacker group the ShadowBrokers. NSA discloses EternalBlue to Microsoft.

March 14, 2017: Microsoft releases a patch for the vulnerability.

May 12, 2017: EternalBlue is the basis of the WannaCry and other ransomware attacks, leading to downtime in critical hospital and city systems and over $1 billion of damages.

Microsoft on EternalBlue

"[T]his attack provides yet another example of why the **stockpiling of vulnerabilities** by governments is such a problem. ...

We need governments to consider the **damage to civilians** that comes from hoarding these vulnerabilities and the use of these exploits.

This is one reason we called in February for a new "Digital Geneva Convention" to govern these issues, including a **new requirement for governments to report vulnerabilities to vendors**, rather than stockpile, sell, or exploit them.

And it's why we've pledged our support for **defending every customer everywhere** in the face of cyberattacks**, regardless of their nationality**."

If intentionally exploiting a vulnerability to access private or encrypted information would be wrong if a private citizen did it, is it equally wrong when a government does it?

Some people argue that we have special obligations to our own country or to other citizens of our own countries.

For example, President Obama described surveillance program PRISM as "a circumscribed, narrow system directed at **us being able to protect our people**."

What would it mean to have special obligations to citizens of ones own country?

Partiality

Partiality holds that it is acceptable to give preferential treatment to some people on the basis of our relationships to them or shared group membership with them.

Impartiality, by contrast, involves "acting from a position that acknowledges that all persons are ... equally entitled to fundamental conditions of well-being and respect."

Partiality

Degrees of Partiality

Partiality: preference towards own family, friends, and state is morally acceptable or even required

Partial Cosmpolitanism: limited preference towards own state acceptable

Universal Care: preference towards family acceptable but not towards state

Impartial Benevolence: same moral responsibilities towards all people

Kwame Appiah's Partial Cosmopolitanism

"In the final message my father left for me and my sisters, he wrote, 'Remember you are citizens of the world.' But as a leader of the independence movement in what was then the Gold Coast, he never saw a conflict between local partialities and a universal morality — between being part of the place you were and a part of a broader human community" (Cosmopolitanism xviii).

Appiah advocates consideration for compatriots, but within a recognition of broader human community that creates universal moral obligations.

Mozi's Inclusive Care (jian ai)

"Suppose people were for others' states as for their own state (impartiality). Then who alone would deploy his state to attack others' states? One would be for others as for oneself.

... That being so, then states and cities not attacking and assaulting each other, people and clans not disordering and injuring each other, is this harm to the world? Or is it benefit to the world? We must say, It is benefit to the world."

Mozi advocates for partiality towards family members but not partiality towards one state over another.

Sāntideva's Argument for Impartial Benevolence

“I should dispel the suffering of others because it is suffering like my own suffering. I should help others too because of their nature as beings, which is like my own being.

When happiness is dear to me and others equally, what is so special about me that I strive after happiness only for myself?

When fear and suffering are disliked by me and others equally, what is so special about me that I protect myself and not the other?” (C&S: 96)

Impartial benevolence has no special role for partial relationships. All beings have an equal claim to my help in promoting happiness and diminishing suffering.

Should I be loyal to my country, a citizen of the world, or both?

When should I give preference to my family members and when should I strive to treat all equally?

What you choose matters:

the moral obligations you take on constitute who you are.

What would these four positions advocate in the case of EternalBlue?

basis of Obama's moral claim

basis of Microsoft's moral claim

Partiality: preference towards own family, friends, and state is morally acceptable or even required	Partial Cosmpolitanism: limited preference towards own state acceptable	Universal Care: preference towards family acceptable but not towards state	Impartial Benevolence: same moral responsibilities towards all people

This may depend on how morally bad the underlying violation of privacy is.

Even partiality theorists do not argue that you would behave with morally admirable partiality if you help your friend kill someone. In fact, you are harming your friend (and the person killed!) by enabling your friend to do something morally wrong.

Where do violations of privacy fall? Next time on Embedded EthiCS ...

Thank you!

Office Hours: calendly.com/kathleencreel

Recap

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

Next time: assembly language

Live Session Slides

Plan For Today

- **10 minutes:** general review
- **5 minutes:** post questions or comments on Ed for what we should discuss
- **15 minutes**: open Q&A
- **25 minutes**: extra practice

Lecture 9 takeaway: A function pointer is a type of variable that stores a function. We can store functions in variables and pass them as parameters. A common use case is to pass comparison functions to generic functions like bubble sort that need to compare elements. We also learned about partiality and how it can influence security.

🌟 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]);
```

Function pointers as variables

```
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 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

file_that_sorts_ints.c

```
#include <bubblesort.h>

int main(int argc, char *argv[]) {
    ...
}
```

file_that_sorts_strings.c

```
#include <bubblesort.h>

int main(int argc, char *argv[]) {
    ...
}
```

file_that_sorts_structs.c

```
#include <bubblesort.h>

int main(int argc, char *argv[]) {
    ...
}
```

Goal: write 1 implementation of bubblesort that any program can use to sort data of any type.

bubblesort.h/c

```
void bubble_sort(void *arr, int n, int
elem_size_bytes, function compare_fn)
```

Generic Bubble Sort

file_that_sorts_ints.c

```
#include <bubblesort.h>

int main(int argc, char *argv[]) {
    ...
}
```

file_that_sorts_strings.c

```
#include <bubblesort.h>

int main(int argc, char *argv[]) {
    ...
}
```

file_that_sorts_structs.c

```
#include <bubblesort.h>

int main(int argc, char *argv[]) {
    ...
}
```

Goal: write 1 implementation of bubblesort that any program can use to sort data of any type.

bubblesort.h/c

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

Generic Bubble Sort

file_that_sorts_ints.c

```
#include <bubblesort.h>

int main(int argc, char *argv[]) {
    ...
}
```

file_that_sorts_strings.c

```
#include <bubblesort.h>

int main(int argc, char *argv[]) {
    ...
}
```

file_that_sorts_structs.c

```
#include <bubblesort.h>

int main(int argc, char *argv[]) {
    ...
}
```

Just like other types, the types of function parameters must match exactly.

bubblesort.h/c

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

Generic Bubble Sort

file_that_sorts_ints.c

```
#include <bubblesort.h>

int main(int argc, char *argv[]) {
    ...
}
```

file_that_sorts_strings.c

```
#include <bubblesort.h>

int main(int argc, char *argv[]) {
    ...
}
```

file_that_sorts_structs.c

```
#include <bubblesort.h>

int main(int argc, char *argv[]) {
    ...
}
```

Generic comparison functions take pointers *to the elements* to compare.

bubblesort.h/c

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

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 *)) {
    …
}
```

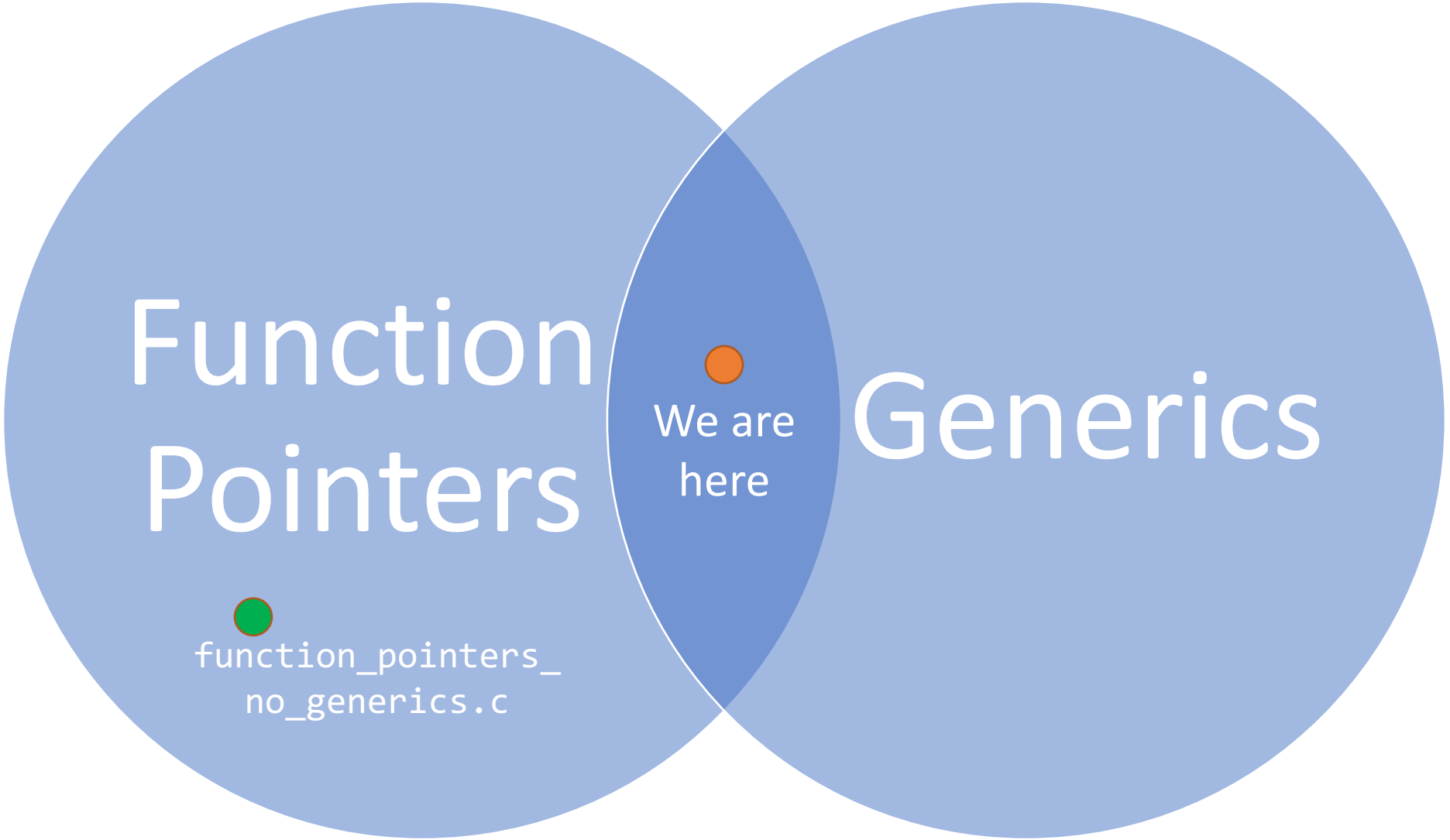
User/caller

- 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.

Function Pointers and Generics



Plan For Today

- **10 minutes:** general review
- **5 minutes:** post questions or comments on Ed for what we should discuss
- **15 minutes**: open Q&A
- **25 minutes**: extra practice

Lecture 9 takeaway: A function pointer is a type of variable that stores a function. We can store functions in variables and pass them as parameters. A common use case is to pass comparison functions to generic functions like bubble sort that need to compare elements. We also learned about partiality and how it can influence security.

Pre-lecture Quiz 4 Q1

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"?

Caller: Pre-lecture Quiz 4 Q1

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. **Draw pictures**!

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;
}
```