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

• Generics So Far
• **Motivating Example:** Bubble Sort
• Function Pointers
• **Example:** Generic Printing
Lecture Plan

• Generics So Far

• Motivating Example: Bubble Sort

• Function Pointers

• Example: Generic Printing

```
cp -r /afs(ir/class/cs107/lecture-code/lect9)
```
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.
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    memcpy(temp, data1ptr, nbytes);
    memcpy(data1ptr, data2ptr, nbytes);
    memcpy(data2ptr, temp, nbytes);
}

Generic Swap

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

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

// now x = 0xffffffeeee, y = 0xeeeeefffff!
printf("x = 0x%x, y = 0x%x\n", x, y);
```
memset is a function that sets a specified amount of bytes at one address to a certain value.

```c
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).

```c
int counts[5];
memset(counts, 0, 3); // zero out first 3 bytes at counts
memset(counts + 3, 0xff, 4) // set 3rd entry’s bytes to 1s
```
Why are void * pointers useful?

Because each parameter and return type must be a single type with a single size.
Why Are void * Pointers Useful?

• Each parameter and return type must be a single type with a single size.

• Problem #1: for a function parameter to accept multiple data types, it needs to be able to accept data of different sizes.
  • Key Idea #1: pointers are all the same size regardless of what they point to. To pass different sizes of data via a single parameter type, make the parameter be a pointer to the data instead.

• Problem #2: we still might pass either a char *, int *, etc. These are the same size, but still different declared types. What should the parameter type be?
  • Key Idea #2: A void * encompasses all these types — it represents a “pointer to something”. A char *, int *, etc. all implicitly cast to void *.

• Solution: to pass one of multiple types via a single parameter/return, that parameter/return’s type can be void *, and we can pass a pointer to the data.
Lecture Plan

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

```
```
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!
• 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!
• Let’s write a function to sort a list of integers. We’ll use the **bubble sort** algorithm.

![Bubble Sort Diagram]

- 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!
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**.

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>12</td>
<td>-5</td>
<td>56</td>
<td>14</td>
</tr>
</tbody>
</table>

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

![Bubble Sort](image)

• 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!
• Let’s write a function to sort a list of integers. We’ll use 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!
• 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!
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!
• Let’s write a function to sort a list of integers. We’ll use the **bubble sort** algorithm.

![Bubble Sort Example](image)

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

  ![Array](image)

  - 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!
• Let’s write a function to sort a list of integers. We’ll use the bubble sort algorithm.

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>-5</td>
<td>4</td>
<td>12</td>
<td>14</td>
<td>56</td>
<td></td>
</tr>
</tbody>
</table>

• 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.
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!

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.
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*?
Let’s start by making the parameters and swap generic.
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.
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:

```c
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?

```c
void *ith_elem = (char *)arr + i * elem_bytes;
```
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;
        }
    }
}
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;
        }
    }
}
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;
        }
    }
}
A Generics Conundrum

• We’ve hit a snag – there is no way to generically compare elements. They could be any type and have complex ways to compare them.

• How can we write code to compare any two elements of the same type?

• That’s not something that bubble sort can ever know how to do. BUT – our caller should know how to do this, because they’re supplying the data….let’s ask them!
Lecture Plan

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

```bash
cp -r /afs(ir/class/cs107/lecture-code/lect9 .
```
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;
        }
    }
}
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;
        }
    }
}
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;
        }
    }
}
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;
        }
    }
}
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)`
A function pointer is the variable type for passing a function as a parameter. Here is how the parameter’s type is declared.

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

Return type (bool)
A function pointer is the variable type for passing a function as a parameter. Here is how the parameter’s type is declared.

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

**Function pointer name**
(compare_fn)
A function pointer is the variable type for passing a function as a parameter. Here is how the parameter’s type is declared.

```c
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]])
```
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

```c
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.
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.
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
void bubble_sort(void *arr, int n, int elem_size_bytes, 
  bool (*compare_fn)(void *, void *)) {
  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)
bool integer_compare(void *ptr1, void *ptr2) {
    // 1) cast arguments to int *
    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.
bool integer_compare(void *ptr1, void *ptr2) {
    // 1) cast arguments to int *
    int *num1ptr = (int *)ptr1;
    int *num2ptr = (int *)ptr2;

    // 2) dereference typed pointers 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.
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`

```c
int (*compare_fn)(void *a, void *b)
```
Comparison Functions

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

```c
int (*compare_fn)(void *a, void *b)
```
```c
int string_compare(void *ptr1, void *ptr2) {
    // cast arguments and dereference
    char *str1 = *(char **)ptr1;
    char *str2 = *(char **)ptr2;

    // perform operation
    return strcmp(str1, str2);
}
```
Function Pointer Pitfalls

• If a function takes a function pointer as a parameter, it will accept it if it fits the specified signature.

• *This is dangerous!* E.g. what happens if you pass in a string comparison function when sorting an integer array?
Lecture Plan

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

```bash
```
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.

\[ \text{int} \ (\text{cmp_fn})(\text{void} \ *\text{addr1}, \ \text{void} \ *\text{addr2}) \]

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

\[ \text{void} \ (\text{print_fn})(\text{void} \ *\text{addr}) \]

• There are many more! You can specify any functions you would like passed in when writing your own generic functions.
In addition to parameters, you can make normal variables that are functions.

```c
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 operations generic.
Recap

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

**Next time:** assembly language
Question Break

Post any questions you have to the lecture thread on the discussion forum for today’s lecture!
Extra Practice
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.

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