CS107, Lecture 14
Function Pointers, Continued

Reading: K&R 5.11

CS107 Topic 4

How can we use our knowledge of memory and data representation to write code that works with any data type?

Why is answering this question important?

- Writing code that works with any data type lets us write more generic, reusable code while understanding potential pitfalls (previously)
- Allows us to learn how to pass functions as parameters, a core concept in many languages (last time + today)

assign4: implement your own version of the **ls** command, a function to generically find and insert elements into a sorted array, and a program using that function to sort the lines in a file like the **sort** command.

Learning Goals

- Learn how to pass functions as parameters
- Learn how to write functions that accept functions as parameters

Lecture Plan

- **Recap:** Function Pointers
- **Example:** Count Matches
- **Introduction:** Assembly

```
cp -r /afs/ir/class/cs107/lecture-code/lect14 .
```

Lecture Plan

- **Recap: Function Pointers**
- **Example:** Count Matches
- **Introduction:** Assembly

```
cp -r /afs/ir/class/cs107/lecture-code/lect14 .
```

Recap: Function Pointers

Function pointers allow us to pass functions as parameters and store functions in variables. We can use them to "pass logic around our program".

- **Example – Bubble Sort:** we want to write a function anyone can use to sort an array. We ask the caller to pass in a function as a parameter that can take in two of the elements and tell us what order they should be in. Bubble Sort can call this function whenever it needs to know the ordering of two elements.
  - We want this "comparison function" to support scenarios with any type of data; the only way to do this is for the comparison function signature to take in *pointers* to the two elements being compared. We use **void *** to indicate "any pointer".

```
void bubble_sort(void *arr, size_t n, size_t
elem_size_bytes, int (*cmp_fn)(void *, void *))
```

Comparison Functions

Function pointers are used often in cases like this to compare two values of the same type. These are called **comparison functions**.

- The standard comparison function in many C functions provides even more information. It should return:
  - < 0 if first value should come before second value
  - > 0 if first value should come after second value
  - 0 if first value and second value are equivalent
- This is the same return value format as **strcmp**!

```
int (*compare_fn)(void *, void *)
```

Recap: Function Pointers

- **Example – Bubble Sort:** we want to write a function anyone can use to sort an array. We ask the caller to pass in a function as a parameter that can take in two of the elements and tell us what order they should be in.

```
void bubble_sort(void *arr, size_t n, size_t
elem_size_bytes, int (*cmp_fn)(void *, void *))
```

When a program wants to use this function, they use it with a specific kind of data and would write a comparison function specifically for that kind of data and the ordering they want. Let's see an example!

Demo: Bubble Sort

`bubble_sort_generic.c`

Calling Generic Bubble Sort

```
// 0 if equal, neg if first before second, pos if second before first
int sort_descending(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]), sort_descending);
    ...
}
```

Key idea: now the comparison function is passed pointers to the elements being compared.

Function Pointers

```
int sort_descending(void *ptr1, void *ptr2) {
    // 1) cast arguments to what they really are here
    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 num2 - num1;
}
```

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

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

Comparison Functions

- **Exercise:** how can we write a comparison function for bubble sort to sort strings in alphabetical order?
- 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 = ___???___ptr1;
    char *str2 = ___???___ptr2;

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

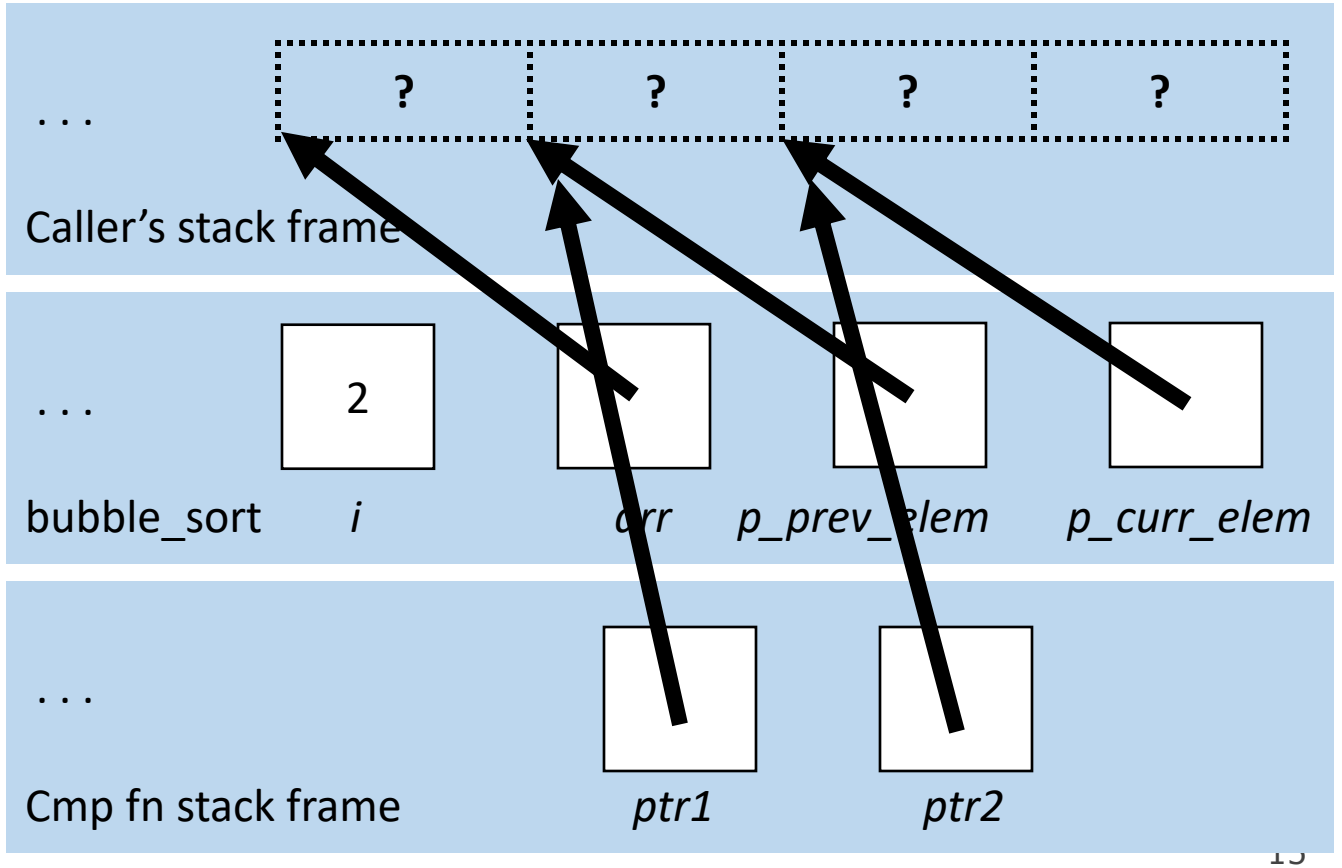
What should we put in the blanks (same expression in both) to correctly implement a comparison function for strings? **Respond with your thoughts on PollEv:** pollev.com/cs107 or text CS107 to 22333 once to join.

Hint: remember what the true types of the parameters are. **Draw pictures**!

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);
}
```



Hint: remember what the true types of the parameters are. **Draw pictures**!

Lecture Plan

- **Recap:** Function Pointers
- **Example: Count Matches**
- **Introduction:** Assembly

```
cp -r /afs/ir/class/cs107/lecture-code/lect14 .
```

Function Pointers

- We will commonly see function pointers used for comparison functions for various library functions. But if we implement a function, we can specify any function we want for the caller to pass in.
- 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…

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, size_t nelems,
                  size_t elem_size_bytes,
                  bool (*match_fn)(void *));
```

Demo: Count Matches

`count_matches.c`

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?

More Function Pointer Details

- Function pointers can be set to NULL.
- Function pointers can be more than just parameters – we can also make variables!

Function pointers as variables

```
 1 int main(int argc, char *argv[]) {
 2     int (*cmp)(void *, void *) = sort_ascending:
 3     if (...) cmp = sort_descending;
 4     else if (...) cmp = sort_odd_then_even;
 5     ...
 6
 7     bubble_sort(nums, count, sizeof(nums[0]), cmp);
 8     ...
 9 }
10
11
12
```

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.

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

- **scandir** – I can create a directory listing with any order and contents! To do that, I need you to provide me a function that tells me whether you want me to include a given directory entry in the listing. I also need you to provide me a function that tells me the correct ordering of two given directory entries.

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

Generic C Standard Library Functions

- **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.

Generics 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*.

Lecture Plan

- **Recap:** Function Pointers
- **Example:** Count Matches
- **Introduction:** Assembly

```
cp -r /afs/ir/class/cs107/lecture-code/lect14 .
```

CS107 Topic 5: How does a computer interpret and execute C programs?

CS107 Topic 5

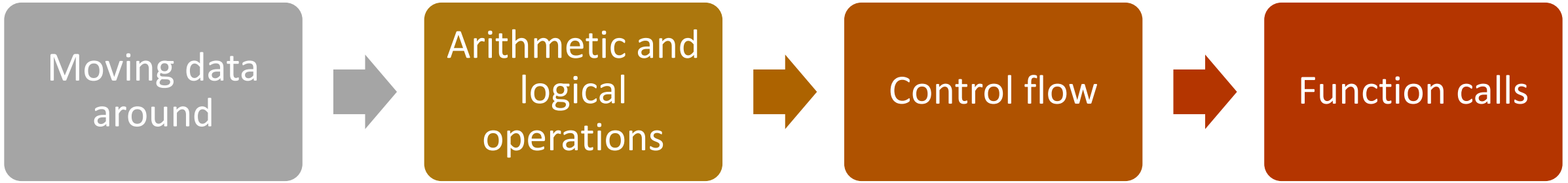
How does a computer interpret and execute C programs?

Why is answering this question important?

- Learning how our code is really translated and executed helps us write better code
- We can learn how to reverse engineer and exploit programs at the assembly level

assign5: find and exploit vulnerabilities in an ATM program, reverse engineer a program without seeing its code, and de-anonymize users given a data leak.

Learning Assembly



Bits all the way down

Data representation so far

- Integer (`unsigned int`, 2's complement signed `int`)
- `char` (ASCII)
- Address (unsigned long)
- Aggregates (arrays, `structs`)

The code itself is binary too!

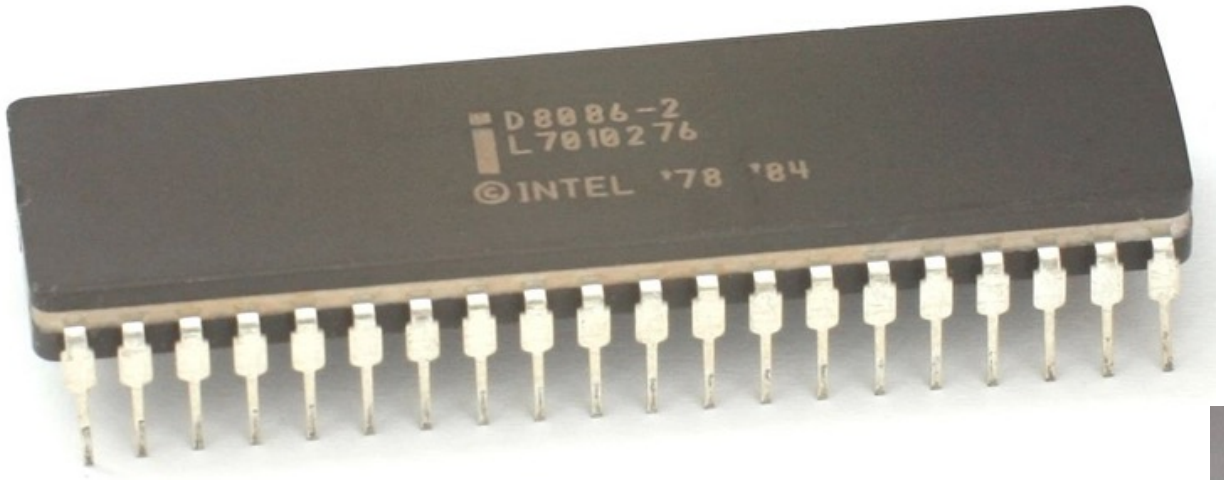
- Instructions (machine encoding)

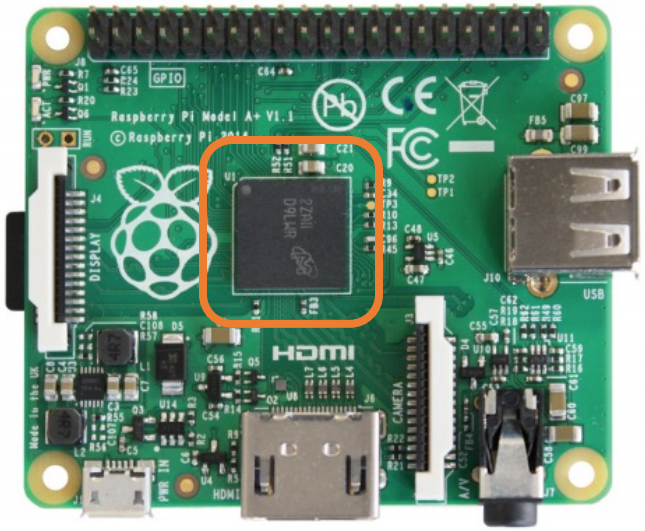
GCC

- **GCC** is the compiler that converts your human-readable code into machine-readable instructions.
- C, and other languages, are high-level abstractions we use to write code efficiently. But computers don't really understand things like data structures, variable types, etc. Compilers are the translator!
- Pure machine code is 1s and 0s – everything is bits, even your programs! But we can read it in a human-readable form called **assembly**. (Engineers used to write code in assembly before C).
- There may be multiple assembly instructions needed to encode a single C instruction.
- We're going to go behind the curtain to see what the assembly code for our programs looks like.

Central Processing Units (CPUs)

Intel 8086, 16-bit microprocessor ($86.65, 1978)





Raspberry Pi BCM2836
32-bit **ARM** microprocessor
($35 for everything, 2015)

Intel Core i9-9900K 64-bit
8-core multi-core processor
($449, 2018)

Recap

- **Recap:** Function Pointers
- **Example:** Count Matches
- **Introduction:** Assembly

Lecture 14 takeaway: A common use case for function pointers is to pass comparison functions to generic functions like bubble sort that need to compare elements; but there are many use cases, such as with counting matches. Our next topic, assembly, will take us on a deep dive of what a compiled program looks like!