CS107 Lecture 13

C Generics and Function Pointers, Take II

Reading: K&R 5.11

Based on slides created by Cynthia Lee, Chris Gregg, Jerry Cain, Lisa Yan, Nick Troccoli, and others.

Standard Comparator Paradigm

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

The standard comparison function **typically provides more information than match-versus-no-match** in the form of a **`bool`**. More often, it returns:

- < 0 if the first value is less than the second
- > 0 if the first value is greater than the second
- 0 if the first value and the second value are equal

To conform to industry standard, our callback function—that is, our comparator—should be of type:

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

Integer Bubble Sort: Going Generic

The **`bubble_sort_int`** presented below is the **most general we can be**, assuming of course we know we're sorting **`int`**s.

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

As the implementation is specific to **`int`**s, it passes **copies** of adjacent **`int`**s to the supplied comparison function to **learn whether they are out of order**.

The comparison function **returns 0** whenever **the two elements are equal**. It returns a **positive** value whenever **the first element is larger than the second**, and it returns a **negative** value whenever **the second is larger than the first**.

The comparison function gets to decide what **equal**, **larger**, and **smaller** mean. **#deep**

Obvious next question: What about other element types?

Integer Bubble Sort: Going Generic

To write one generic **bubble sort** function, **we need a single prototype that interfaces cleanly with all array element types**, **`int`**s or otherwise.

Version 1:

```
void bubble_sort(int arr[], size_t n,
                 int (*cmp_fn)(int, int));
```

This prototype is the one we've coded to thus far. Our implementation can sort **any integer array**, provided **`cmp_fn`** is clear what it means for one **`int`** to be larger or smaller than another.

Version 2:

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

This now accepts **an array of any type** but requires we supply the element size, much as was required for **`swap`** and **`swap_ends`**. The comparison function is still **`int`**-specific, so **we're not done just yet**.

Version 3:

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

This is what we want! A fully generic implementation won't know the element type, so it can't pass element copies to **`cmp_fn`**. It can, however, **pass the addresses of neighboring elements as generic pointers**. Because **`cmp_fn`** is written as a callback, **it'll know what those pointers really are** (and how to cast them properly ☺).

Integer Bubble Sort: Going Generic

Here's an implementation of a fully generic **bubble_sort**.

```
void bubble_sort(void *arr, size_t n, size_t elem_size,
                 int (*cmp_fn)(void *, void *)) {
   while (true) {
      bool swapped = false;
      for (size_t i = 1; i < n; i++) {
         void *first = (char *) arr + (i - 1) * elem_size;
         void *second = (char *) arr + i * elem_size;
         if (cmp_fn(first, second) > 0) {
            swap(first, second, elem_size);
            swapped = true;
         }
      }
      if (!swapped) return;
    }
}
```

The implementation knows **where the array begins and how large each element is**, but it **doesn't know what the elements themselves are**. The best we can do is compute the **addresses** of the elements at positions **`i - 1`** and **`i`**.

Note **we're now passing element addresses to the client-supplied comparison function**. Again, **that's the best we can do**, as we aren't permitted to dereference **`void`** `*`s when we **lack type information**.

Integer Bubble Sort: Going Generic

Before we can properly use our generic bubble sort, **we need to understand what bubble_sort knows about the array it's sorting**.

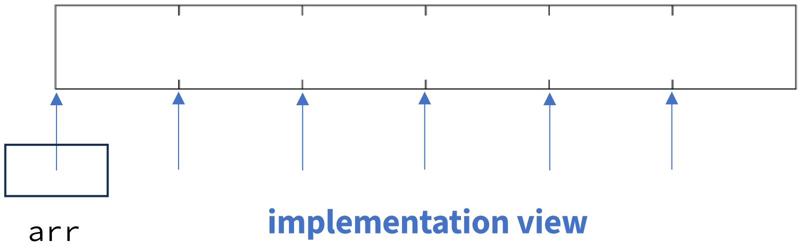
```
void bubble_sort(void *arr, size_t n, size_t elem_size,
                 int (*cmp_fn)(void *, void *)) {
   while (true) {
      bool swapped = false;
      for (size_t i = 1; i < n; i++) {
         void *first = (char *) arr + (i - 1) * elem_size;
         void *second = (char *) arr + i * elem_size;
         if (cmp_fn(first , second) > 0) {
            swap(first, second, elem_size);
            swapped = true;
         }
      }
      if (!swapped) return;
    }
}
```

client view

4	2	12	-5	56	14

Assume the client **declares and initializes** an **`int`** array of length 6 as above. The client, of course, has near perfect information.

The **`bubble_sort`** implementation **does not**! It only knows the **array's base address, the number of elements, and the element size**. From this trio of facts, **`bubble_sort`** can **compute the addresses** of the array elements, but that's it.



implementation view

Integer Bubble Sort: Going Generic

Let's **reproduce the client program** that sorts an integer array from low to high. The **difficult part** is understanding what the **void** *s passed to the comparator really are.

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

int sort_ascending(void *first, void *second) {
   return *(int *)first - *(int *)second;
}

int main(int argc, char *argv[]) {
   int nums[] = {4, 2, 12, -5, 56, 14};
   size_t count = sizeof(nums) / sizeof(nums[0]);
   bubble_sort(nums, count, sizeof(int), sort_ascending);
   ...
   return 0;
}
```

The comparison function is **required** to take two **void** *s. That's part of the **contract** we have with a **bubble_sort** that's capable of **sorting all array types**.

Because the client knows the element addresses are **int** *s, it knows those **int** *s are **disguised** as **void** *s—**necessarily** so, cause, **contract**—when passed to the comparator.

The implementation of **sort_ascending** **must cast** each of the two **void** *s to be the **int** *s it knows them to be. It uses the **int** * cast to **tell the truth about what data elements are** being compared to one another for this specific call to **bubble_sort**.

Integer Bubble Sort: Going Generic

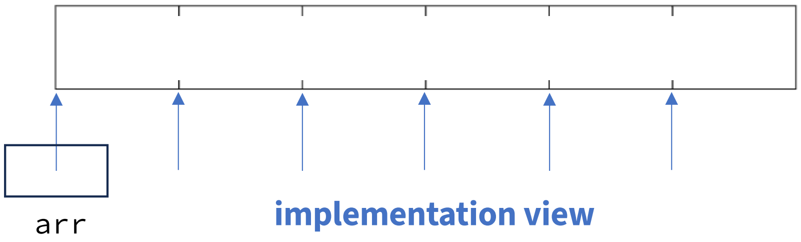
And what about **C string arrays**? What does a **comparison function look like for those**?

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

int string_cmp(void *first, void *second) {
   char *one = *(char **) first;
   char *two = *(char **) second;
   return strcmp(one, two);
}

int main(int argc, char *argv[]) {
   char *words[] = {
      "sabotage", "bumfuzzle", "winsome", "ablution", "gravamen", "crepuscular"
   };
   size_t count = sizeof(words) / sizeof(words[0]);
   bubble_sort(words, count, sizeof(char *), string_cmp);
   ...
   return 0;
}
```

The base address of the **`words`** array is of type **`char`** `**`, since each of its elements are **`char`** `*`s. In fact, **all addresses computed** by **`bubble_sort`** are **`char`** `**`s **disguised as** **`void`** `*`s. That means the **`void`** `*`s passed to **`string_cmp`** should be cast as **`char`** `**`s.



implementation view

The casts within **`string_cmp`** must always **expose the truth** about what the incoming **`void`** `*`s really are.

Generic C Standard Library Functions

The C standard libraries include **many generic search and sort routines**, the most frequently used being **`qsort`** and **`bsearch`**. You'll use **both** in **`assign4`**.

- **`qsort`**: I can sort an array of any type! To do that, I need 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 *));
```

- **`bsearch`**: I can use binary search to search for a key in an array of any type! To do that, I need a function that can **compare two elements of the kind you are asking me to search**.

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

- **`lfind`**: I can use linear search to search for a key in an array of any type! To do that, I need a function that can **compare two elements of the kind you are asking me to search**.

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

- **`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. To do that, I need a function that can **compare two elements of the kind you are asking me to search**.

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

When you find a library implementation
of quicksort so you don't have
to write bubble sort anymore