15-122: Principles of Imperative Computation

Recitation 22 — Josh Zimmerman

Function pointers

Think back to the memory layout recitation. Remember that in the layout of memory there is a section for code.

In C, we can get the address of anything, including things in this code section.

So, we can get pointers to functions, normally referred to as *function pointers*.

Let's consider the following code snippet:

```c
int square(int a) {
  return a * a;
}
int add1(int a) {
  return a + 1;
}
void map(int (*f)(int), int A[], size_t n) {
  for (int i = 0; i < n; i++) {
    A[i] = f(A[i]);
  }
}
```

In this example, we wrote a generic function that applies a given function to every value in an array of `int`s. The function can be anything we want it to be, which gives us more flexibility: instead of writing an "add 1 to all in array" function and a "square all in array" function, we can just write the `map` function and pass in a function pointer.

The syntax for function pointers is a bit ugly, so we'll use `typedef`s to make it easier to deal with. (A handy way to remember `typedef` syntax: if you remove the `typedef` from `typedef foo bar;` then you're declaring a variable named `bar` of type `foo`. For instance, when you do `typedef int elem;` if you remove the `typedef`, you're declaring an `int` named `elem`.)

We can use function pointers to make our data structures more adaptable. For instance, we can modify the `min_index` function of our selection sort to take a generic comparison function:

```c
// Everything that has the type "compare_fun" takes two elems and returns an int
typedef void* elem;
typedef int (*compare_fun)(elem, elem);
int min_index(elem *A, int lower, int upper, compare_fun elem_compare) {
 REQUIRES(A != NULL && 0 <= lower && lower < upper && elem_compare != NULL);
 int min_idx = lower;
 for (int i = lower + 1; i < upper; i++) {
  if ((*elem_compare)(A[i], A[min_idx]) < 0) {
   min_idx = i;
  }
 }
 return min_idx;
}
```

Similarly, we can modify our `ht_insert` function to take a function pointer that points to a hash function. This allows us to make the hash table implementation much more generic and reusable: we don't have to recompile for each different type of element. We can call the hash function and maintain hash tables for many different types at once, unlike before when we had to separately compile depending on the type and the hash function.

Specifically, we can `typedef int (*hashFunction)(void *, int);` and we'll have a type called `hashFunction`. Variables of this type take a `void *` (the thing we want to hash) and an `int` (the size of the hash table) and return the hash of the element.

Then, if we have something of type `hashFunction`, we can pass that as an argument to `ht_new`, which would then store a pointer to the hash function along with the hash table, allowing us to have multiple hash tables that can hash different types. (We could make one hash table that has a hash function to hash `int`s, another to hash strings, and so on.)

Modifying `heaps.c` to free non-empty heaps

In Huffman lab, we had to make sure that the priority queue was empty before we freed it. But what if instead we use a function pointer to let us free non-empty queues without leaking memory?

We need to modify our heaps to have a `elem_free` function as part of the data structure.

Let's write this now. Here's the start of the code for the new version of `pq_free`, and some modified code to support a function pointer.

```
typedef void (*elem_free_t)(elem);
struct heap_header {
  int limit;              /* limit = capacity+1 */
  int next;               /* 1 <= next && next <= limit */
  elem* data;             /* \length(data) == limit */
  elem_free_t elem_free; /* Non-NULL pointer to function that frees elems */
};
typedef struct heap_header* heap;

heap pq_new(int capacity, elem_free_t elem_free) {
  REQUIRES(capacity > 0);
  REQUIRES(elem_free != NULL);
  heap H = xcalloc(1, sizeof(struct heap_header));
  H->limit = capacity+1;
  H->next = 1;
  H->data = xcalloc(capacity+1, sizeof(elem));
  H->elem_free = elem_free;
  ENSURES(is_heap(H));
  return H;
}

void pq_free(heap H) {
  REQUIRES(is_heap(H));
  for (int i = __; i < ___________________; i++) {
      ______________________________________________________________ // Free elements of the heap
  }
  ______________________________
  _______________
}
```

Casting, revisited

We can play around some more with casting, to help get more familiar with it.

The file `signed-casting.c` is available on the course website if you want to look at it after recitation.