15-122: Principles of Imperative Computation

Lab C: Code of a generic sort

Rob Simmons

Collaboration: In lab, we encourage collaboration and discussion as you work through the problems. These activities, like recitation, are meant to get you to review what we've learned, look at problems from a different perspective and allow you to ask questions about topics you don't understand. We encourage discussing problems with your neighbors as you work through this lab!

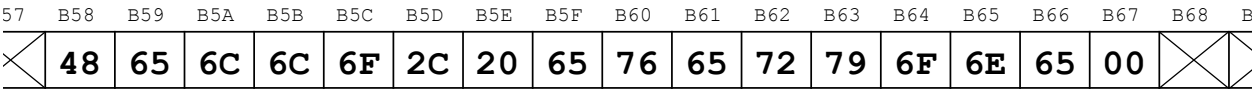
Setup: Copy the lab code from our public directory to your private directory:

```
% cd private/15122
% cp -R /afs/andrew/course/15/122/misc/lab-gsort .
% cd lab-gsort
```

Grading: Finish task (1.1) for 2 points, and finish both tasks (1.1) and (1.2) for 3 points.

Generic sort

Today we will explore the difference between `void*` in C0 and `void*` in C, and exploit that difference to write a generic sort in C.



The image above shows a 16 byte allocation where each byte contains an 8-bit value. The actual size of a byte in C is implementation-defined and so can be either 8 bits *or more*. You can pretty much count on a byte being 8 bits on current computers.

If we have a pointer `A` whose value is `0xB58`, then the way we interpret that pointer *depends on the type of the pointer*. As a `char*`, the pointer `A` points to the value `0x48` or `'H'`, the first character in the NUL-terminated string `"Hello, everyone"`. As an `int32_t*`, the pointer `A` points to a signed integer, the first element in an array of four integers. (According to the implementation-defined behavior we usually expect, this array contains the four integers 1819043144, 1696607343, 2037540214, and 6647407.) If `A` is a `void*`, then we know *nothing* about how to interpret, read from, or write to this block of memory.

In this lab, we will write a function that sorts arrays without knowing anything about how to interpret, read from, or write to the memory addresses in that array. The client will tell us the how many elements there are (`count`) and the size of each element (`elt_size`). The client will also tell us how to interpret (with a comparison function) and manipulate (with a swap function) elements of the array.

```
1  typedef void swap_fn(void *x, void *y)
2    /*@requires x != NULL && y != NULL; @*/ ;
3
4  // Compares the objects at memory locations x and y
5  typedef int compare_fn(void *x, void *y)
6    /*@requires x != NULL && y != NULL; @*/
7    /*@ensures -1 <= \result && \result <= 1; @*/ ;
8
9  void gsort(void *A, size_t count, size_t elt_size,
10            swap_fn *swp,
11            compare_fn *comapr)
12   /*@requires A != NULL && swp != NULL && compar != NULL; @*/
13   /* requires that A is an allocation of at least count * elt_size */ ;
```

The interface above is provided in `lib/gsort.h`. Your implementation in `gsort.c` is one of the very small number of cases in C where it is acceptable to cast a `void*` to another pointer type without being absolutely certain what type the `void*` originally was. (Remember that it was *never* acceptable to do this in C1.)

If a client asks us to sort the 5 two-byte objects starting at the `void` pointer `0xB58`, then we know that the five elements in the array have the addresses `0xB58`, `0xB5A`, `0xB5C`, `0xB5E`, and `0xB60`. We can calculate the address of the array element that the client thinks of as `A[3]` by casting `A` to a `char*` and then writing either `(A + 6)` or `&A[6]`. This makes sense because a `char` is always one byte and because 6 is the array offset (3) times the size of an array element in bytes (2).

All we need to do to sort is calculate the addresses where array elements begin and pass these addresses to the client functions. The client's functions know how to compare and swap objects given the addresses of those objects. Because you are writing the sorting algorithm without knowing what objects are being stored, you shouldn't ever access the memory in the array directly.

(1.a) Write a generic sort in `gsort.c` according to the strategy described above. Make sure your `gsort.c` includes `"lib/gsort.h"`. A C0 version of insertion sort is given to you below and in `sort.c0`. Don't worry about translating the contracts. You can test your code by running:

```
% make
% ./a.out
```

(1.b) Modify `gsort-test.c` to sort arrays of frequency counts by both frequency and by word (according to the `string.h` function `strcmp`).

Insertion sort in C0

```
1  void sort(int[] A, int n)
2  //@requires 0 <= n && n <= \length(A);
3  //@ensures is_sorted(A, 0, n);
4  {
5    for (int i = 0; i < n; i++)
6    //@loop_invariant 0 <= i && i <= n;
7    //@loop_invariant is_sorted(A, 0, i);
8    //@loop_invariant le_segs(A, 0, i, A, i, n);
9    {
10     int min = i;
11     for (int j = i + 1; j < n; j++)
12     //@loop_invariant i < j && j <= n;
13     //@loop_invariant i <= min && min < n;
14     //@loop_invariant le_seg(A[min], A, i, j);
15     {
16       if (A[j] < A[min]) {
17         min = j;
18       }
19     }
20     swap(A, i, min);
21   }
22 }
```