15-122: Principles of Imperative Computation — Spring 2024

Lab 12: All Sorts of Sorts — Tuesday March 26th

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 other students in this lab!

Setup: Download the lab handout and code from the course website, and move it to your private directory in your unix.qatar.cmu.edu machine. Following that create a directory, move the handout to it, and unzip the handout file by executing the following commands:

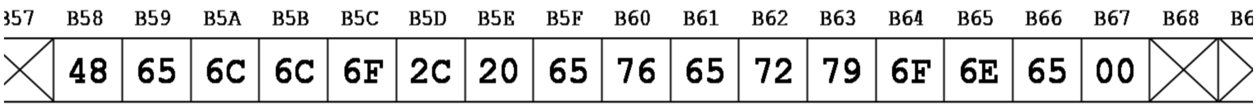
```
% mkdir lab_12
% mv 12-handout.tgz lab_12
% cd lab_12
% tar -xvf 12-handout.tgz
```

Submission:
To submit, create a tar file by executing the command below and submit it to autolab, under the lab name:

```
% tar cfzv handin.tgz gsort-test.c gsort.c
```

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 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 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.

```
typedef int compare_fn(void *x, void *y) // Compares the values at x and y
        /*@requires x != NULL && y != NULL; @*/ ;

typedef void swap_fn(void *x, void *y)   // Swaps the values at locations x and y
        /*@requires x != NULL && y != NULL; @*/ ;
```

```
void gsort(void *A, size_t count, size_t elt_size,
           compare_fn *cmp, swap_fn *swp)
/*@requires A != NULL && cmp != NULL && swp != NULL; @*/
/* requires that A is an allocation of at least count * elt_size bytes */ ;
```

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 values 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).

1.5pt

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 values given the addresses of those values. Because you are writing the sorting algorithm without knowing what values are being stored, you shouldn't ever access the memory in the array directly.

(1.a) In file `gsort.c`, complete the function `gsort` that implements a generic sort according to the strategy described above. A C0 version of selection sort is given to you below and in `sort.c0`. Don't worry about translating its contracts but do include contracts that preempt basic `NULL` pointers dereferences. You can test your code by running:

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

3pt

(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`).

4pt

Selection sort in C0

```
void sort(int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@ensures is_sorted(A, 0, n);
{
  for (int i = 0; i < n; i++)
  //@loop_invariant 0 <= i && i <= n;
  //@loop_invariant is_sorted(A, 0, i);
  //@loop_invariant le_segs(A, 0, i, A, i, n);
  {
    int min = i;
    for (int j = i + 1; j < n; j++)
    //@loop_invariant i < j && j <= n;
    //@loop_invariant i <= min && min < n;
    //@loop_invariant le_seg(A[min], A, i, j);
    {
      if (A[j] < A[min])
        min = j;
    }
    swap(A, i, min);
  }
}
```