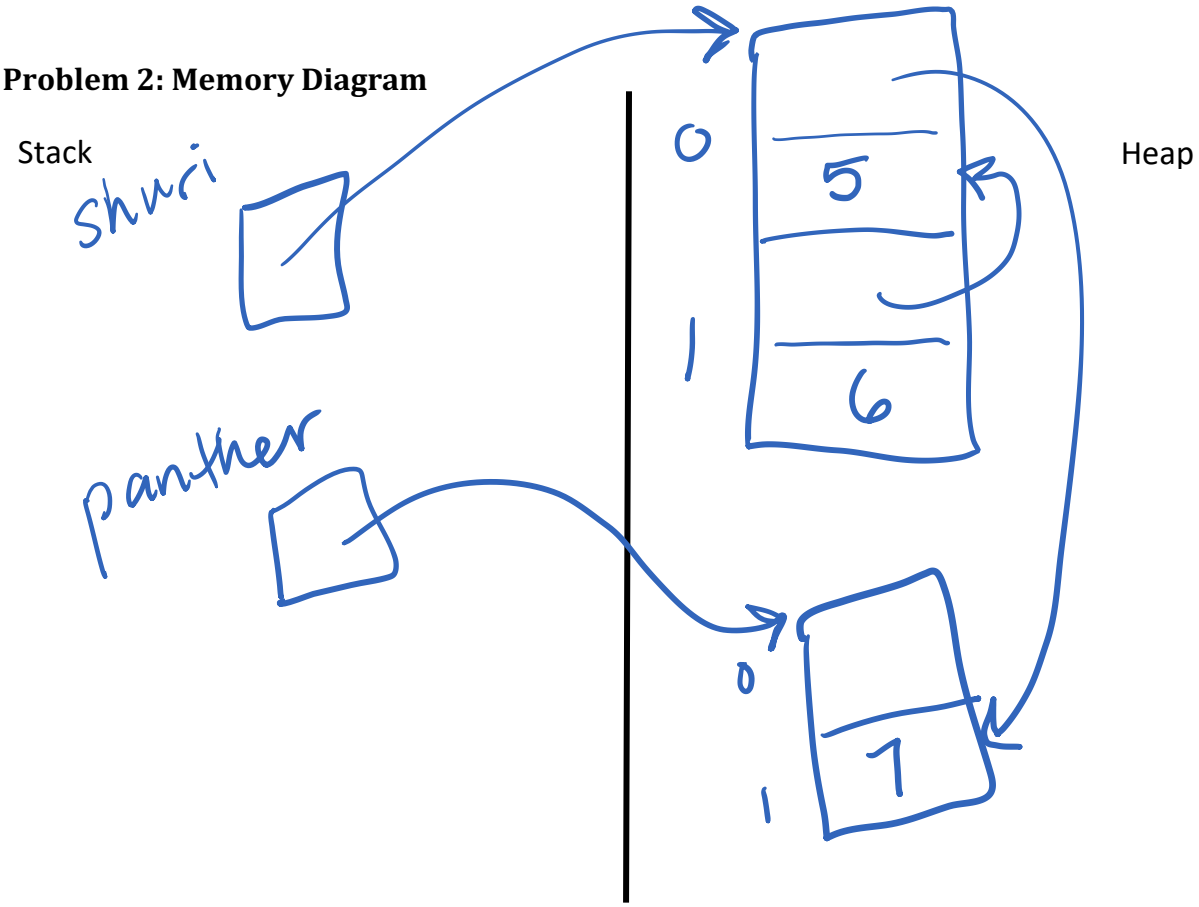
This exam is based on the final exam given in Fall 2018. The class was taught by Cynthia Lee. This was a 3-hour paper exam.

Problem 1: Floating Point

(a) 160
(b) 18
(c) 176 – we lose precision because minifloat does not have enough bits to store the entire sum of the number. This likely would not have been an issue with IEEE 32-bit floats, since there are many more bits to store numbers with more precision.

Problem 2: Memory Diagram



Problem 3: Pointers and Generics (12pts)

```
(a)   void mixup1(char *ptr1, char *ptr2) {
           swap(ptr1 + 1, ptr2 + 1, sizeof(int));
      }

(b)   -256 = 0xFFFFFF00
(c)    255 = 0x000000FF  // notice 0th byte of each is swapped
(d)   //little-endian solution (this is how myth works)
      void mixup2(int *ptr1, int *ptr2) {
          swap(ptr1 + 1, ptr2 + 2, sizeof(char));
      }

      // big-endian solution (we also gave points for this even though
      //not how myth works)
      void mixup2(int *ptr1, int *ptr2) {
          swap((char*)ptr1 + 7, (char*)ptr2 + 11, sizeof(char));
      }
```

Problem 4: Assembly (23pts)

(a) Intended solution:

```
char *vp(unsigned int aaron, char *burr)
{
    unsigned int leslie = strlen(burr) * 3;
    if (aaron < -16) {    // or <= -17 (equivalent for unsigned)
        while (leslie > 6) {
            leslie /= 4;
        }
        if (aaron >= 256) {
            pass_final_level(leslie);
        } else {
            explode_bomb();
        }
    } else {
        explode_bomb();
    }

    return burr + 3;
}
```

(b) Caller-owned registers must be saved before we use them as local/temporary storage, then restored before the function returns.

(c) The `lea` is faster than `imul`, and achieves * 3 by multiplying the value by 2 and then adding the product to itself. Shifting left by 2 with `shr` is the same as multiplying by 4 in binary arithmetic, and `shr` is faster than divide.

(d) It may look like it is not possible, because `aaron` must be both less than -16 and greater than 256. However, the comparison with -16 is unsigned, because when signed and unsigned are compared the unsigned equivalents of both values are used. So we just need a value for `aaron` that satisfies `256 <= aaron < 0xFFFFFFF0` (which is `-16U`). Some common solutions that would work: `256, 257, 300, -17, 0xFFFFFF00`, or "max `int`." Some common solutions that don't work: `255, -1` (or `0xFFFFFFFF` or "max unsigned `int`"), `0xFFFFFFF0`.

Problem 5: Heap Allocator (24pts)

(a)

```
struct Header *get_neighbor(struct Header *hdr)
{
    struct Header *next = (struct Header *)((char*)hdr + hdr->nwords * 8
                          + HDRSIZE);
    return next >= (struct Header *)segment_end ? NULL : next;
}
```

(b) Without a footer to inform us of how many bytes are in the block to the left in O(1) time (the way the header informs us of how many bytes are in the block to the right in O(1) time), we would need to traverse either the free list or the entire heap, looking for the block whose header tells us it is to our left. That looping is O(N).

(c)

```
struct Header *pay_to_hdr(struct Node *payload)
{
    // Possible solution 1
    return (struct Header *)payload – 1;

    // Possible solution 2
    return (struct Header *)((char *)payload – HDRSIZE);
}
```

(d)

```
size_t count_free_inorder()
{
   size_t nfree = 0;
   for (struct Header *curr = segment_start; curr != NULL;
                     curr = get_neighbor(curr)) {
      if (curr->used == 0) // need this because we look at both free and used
         nfree++;
   }
   return nfree;
}
```

(e)

```
size_t count_free_list()
{
   size_t nfree = 0;
   // needs NULL check in pay_to_hdr()
   for (struct Node *curr = free_list; curr != NULL;
                   curr = curr->next) {
      if (pay_to_hdr(curr)->used == 0) // not needed because all are free
         nfree++;
   }
   return nfree;
}
```

(f) You may remove the check from `count_free_list` (part (e)), for the reasons explained in the comments for (d) and (e), which is that the `free_list` contains only free blocks, so there is no need to check the free flag.

(g)

```
void update_header(struct Header *left)
{
   // remember to divide by 8, because stored as number of 8-byte words
   left->nwords += get_neighbor(left)->nwords + HDRSIZE / 8;
}
```

(h)

```
void remove_node(struct Node *remove)
{
   if (remove->prev == NULL) {
      free_list = remove->next;
   } else {
       remove->prev->next = remove->next;
   }
   if (remove->next == NULL) {
      // Empty because no later node needs its prev set.
      ;
   } else {
      remove->next->prev = remove->prev;
   }
}
```