Testing GC Code

1. We give you two programs to use to test the code you write in `hw4.c`, `main_debug.c` and `main.c`.
2. `debug_main.c` has a simple method to test if your code is working. Its `main` method looks like this:

```
int main(int argc, char** argv) {

init_gc();

for (int i=0;i<MAX_ALLOCATIONS;i++)
allocs[i]=my_malloc(i*2+128);

for (int i=0;i<MAX_ALLOCATIONS;i++)
allocs[i] = 0;
gc();
}
```

3. This code allocates 1000 pointers, and then deallocates them all and calls the garbage collector. Its `my_malloc` method will print out information about the chunk every time it is called. Note that we provide a `my_free` method with similar print statements - in order to use this, you will need to call `my_free` instead of `free` in your `hw4.c` code. (You will need to change it back to `free` to work with `main.c`.) Using these methods can be very helpful when you can't figure out what is happening with your code.
4. Answer the Gradescope question on `my_malloc`.

Testing: main.c

1. Now let's look at the `main` method in `main.c`. It starts by running the same allocated and deallocate tests as in `debug_main.c`. Then, we see this code:

```
/* allocations which all point to each other.
this checks for proper traversal of the chunk graph. */
for(int i=0;i<MAX_ALLOCATIONS;i++) {
    allocs[i]=malloc(i*2+128);
    if(i>0)
        *(void**)(allocs[i])=allocs[i-1];
}
printf("Heap after second round of allocations: %zu, free %d, inuse %d\n",
heap_size(),free_chunks(),inuse_chunks());
for(int i=0;i<MAX_ALLOCATIONS-1;i++)
    allocs[i]=0;

gc();
// here, since we keep the last entry, which points to the next-to-last
//and so on, everything should still be around
printf("Heap after clearing all but one, and gc(): %zu, free %d, inuse %d\n",
heap_size(),free_chunks(),inuse_chunks());

allocs[MAX_ALLOCATIONS-1]=0;
gc();
printf("Heap after clearing last one, and gc(): %zu, free %d, inuse %d\n",
heap_size(),free_chunks(),inuse_chunks());
```

2. What should print after our first garbage collection call? What about the second?
3. Answer the question on `main.c` on Gradescope.
4. Next, we see:

```
/* allocations which all point to each other. this checks for
proper traversal of the chunk graph. */
for(int i=0;i<MAX_ALLOCATIONS;i++) {
    allocs[i]=malloc(i*2+128);
    if(i>0) {
        void *start_of_new_alloc = allocs[i];
        void *start_of_prev_alloc = allocs[i-1];

        int offset_into_new_alloc = 8*random_up_to((i*2+120)/8);
        int offset_into_old_alloc = 8*random_up_to((i-1)*2+120)/8);
        void **location_of_pointer = (void**)(start_of_new_alloc
                                + offset_into_new_alloc);

        *location_of_pointer = start_of_prev_alloc
                             + offset_into_old_alloc;
    }
}
printf("Heap after third round of allocations: %zu, free %d, inuse %d\n",
heap_size(),free_chunks(),inuse_chunks());
for(int i=0;i<MAX_ALLOCATIONS-1;i++)
    allocs[i]=0;
gc();
// here, since we keep the last entry, which points to the next-to-last
//and so on, everything should still be around
printf("Heap after clearing all but one, and gc(): %zu, free %d, inuse %d\n",
heap_size(),free_chunks(),inuse_chunks());

allocs[MAX_ALLOCATIONS-1]=0;
gc();
printf("Heap after clearing last one, and gc(): %zu, free %d, inuse %d\n",
heap_size(),free_chunks(),inuse_chunks());
```

5. What should print after our first gc call here? What about the second?
6. Next, we call the following method:

```
/* this keeps pointers strictly on the stack, so at i==50,
we'll have allocated 100 chunks, and gc'd... 49? */
void* recursive_allocations(int i) {
    void* ptr = malloc(i*100+128);
    if(i==0) return ptr;

    void *ptr2 = recursive_allocations(i-1);
    if(i==50) {
        gc();
        printf("Recursive1: at depth 50, %zu, free %d, inuse %d\n",
        heap_size(),free_chunks(),inuse_chunks());
    }
    return ptr;
}
```

7. What should be allocated/free when we call the garbage collector at round 50? How about when we call it after the method returns?
8. Answer the question on `main.c 2` on Gradescope.
9. Next, we run this code:

```
/* here the returned pointer is stored in our local allocation
before we return. Hence at depth 50, we're not able to GC anything. */
void* recursive_allocations2(int i) {
    void** ptr = malloc(i*100+128);
    if(i==0) return ptr;

    *ptr = recursive_allocations2(i-1);
    if(i==50) {
            gc();
            printf("Recursive2: At depth 50, %zu, free %d, inuse %d\n",
            heap_size(),free_chunks(),inuse_chunks());
    }
    return ptr;
}
```

10. Note that in this code, the returned pointer from the recursive call is stored on the heap, not the stack. Since we don't free it, it should still be on the heap after the function returns. What should be allocated/free after we call `gc` within this function? What about when we call `gc` after it returns?
11. Answer the remaining questions on Gradescope.

Functions You Will Write

1. Remember that you need to write three functions: `is_pointer`, `sweep`, and `walk_region_and_mark`.
2. For `is_pointer`, you will take in a `size_t` value, and either return `NULL` if it is not within the heap or not allocated, and otherwise return a pointer to the header of the chunk containing it. We give you the start and ending addresses of the heap, so figuring out if the pointer is in that range should be trivial. Next, we recommend starting at the beginning of the heap, and going through block by block, similar to what you do in the sweep stage. You are looking for the largest block address that is smaller than your pointer, as that will be the address of the block containing it.
3. For `walk_region_and_mark`, extend the psuedo code we went over in class so that instead of just marking from one pointer, it takes a start and end address, and performs the operations below on every pointer within that range.

```
    void mark(ptr p) {
      if (b == is_ptr(p) == NULL)      return;
      if (markBitSet(b)) return;
      setMarkBit(b);
      for (i=0; i < lengthCb); i++)
        mark(b[i]);
      return;
    }
```

4. `sweep` should be almost identical to the psuedo code we went over in class. (It is below, in case you don't remember from the lecture)

```
    ptr sweep(ptr p, ptr end) {
      while (p < end) {
      if blockMarked(p)
        unmarkBlock();
          else if (blockAllocated(p))
            free(p);
        p = nextBlock(p);
        }
    }
```

What your output should look like if your code is correct

```
Heap before first round of allocations: 0, free 0, inuse 0
Heap after first round of allocations: 8048656, free 1, inuse 4000
Heap after gc, before wiping out allocs array: 8048656, free 1, inuse 495
Heap after wiping out allocs array and then gc(): 8048656, free 1, inuse 0
Heap after second round of allocations: 8048656, free 1, inuse 4000
Heap after clearing all but one, and gc(): 8048656, free 1, inuse 0
Heap after clearing last one, and gc(): 8048656, free 1, inuse 0
Heap after third round of allocations: 16556848, free 1, inuse 4000
Heap after clearing all but one, and gc(): 16556848, free 1, inuse 0
Heap after clearing last one, and gc(): 16556848, free 1, inuse 0
Now checking stack root set handling.
Before GC halfway deep in Recursive1: at depth 50, 16556848, free 1, inuse 100
After: 16556848, free 1, inuse 52
After Recursive1 16556848, free 1, inuse 0
Before GC halfway deep in Recursive2: At depth 50, 16556848, free 1, inuse 101
After: 16556848, free 1, inuse 101
After Recursive2 16556848, free 1, inuse 0
```

The numbers you should worry about are the `inuse` and `free` numbers.

Peer evaluation rubric

You can find your pairings for the lab in this sheet. Please grade your peers out of **1 point** using this form.

Session	Task	Points
Session A	Discuss questions 1, 2, and 3	1 point
Session B	Discuss questions 4, and 5	1 point

Total grade calculation

Task	Points
Turn in a Gradescope assignment	5 points
Discuss your answers	1 point
Evaluate another student	1 point
Total points	7 points