HW 5: Memory
IC221, Spring AY22

Name Umberto Fontana
90 points total

1. (20 points) Consider the program below, on the left:

```
#include <stdio.h>
#include <stdlib.h>

int * makearray(int size,int base){

  int array[size];
  int j;
  for(j=0;j<size;j++)
    array[j] = base*=2; //doubling base
  return array;
}

int main(){
  int * a1 = makearray(5,2);
  int * a2 = makearray(10,3);
  int j, sum=0;

  for(j=0;j<5;j++){
    printf("%d ",a1[j]);
    sum+=a1[j];
  }
  printf("\n");

  for(j=0;j<10;j++){
    printf("%d ",a2[j]);
    sum+=a2[j];
  }

  printf("\n");

  printf("SUM: %d\n", sum);
}
```

This program has a memory violation. Identify the memory violation and explain it:

This is a case of *dangling pointer dereference*. The function *makearray* returns an array, which can also be considered an address to the first element of the array. This address, when the function is called, is assigned to the arrays *a1* and *a2*. The problem with this is that, once the function returns and it gets popped of the memory, the local variable *array* gets popped too, so the memory that is referenced by *a1* and *a2* is now unallocated: calling an array with the *[ ]* method, which is the same as dereferencing a pointer, will result in an error, since we are dereferencing a pointer to unallocated memory.

Using a dynamic memory allocation (e.g., with `calloc()` or `malloc()`), rewrite the `makearray()` function to remove the memory violation (keep the arguments the same):

```
int * makearray(int size,int base){
  int * array = (int *) malloc(size *
sizeof(int));
  int j;
  for(j=0;j<size;j++) {
    array[j] = base*=2; //Doubling base
  }
  return array;
}
```

2. (10 points) What if you have already allocated memory using either `calloc` or `malloc`, and you later find you need to increase the allocation size? Can you do that? If so, explain how. (Hint: `man malloc`)

Yes, it can be done with the `realloc()` function, which changes the size of the memory block pointed to by a pointer to a different size in bytes. The contents of the memory will be unchanged in the range from the start of the region up to the minimum of the old and new sizes. The `reallocarray()` function can also be used to change the size of the memory block pointed to by a pointer to be large enough for an array of n elements, each of which is size bytes. Unlike `realloc()`, `reallocarray()` fails safely in the case where the multiplication would overflow.

3. (15 points) For the code below, draw the function stack diagram at each push (function call) and pop (function return). Follow the example in the course notes.

`int times(int a, int b){`
`  return a*b;`
`}`
 	
`int add(int a, int b){`
`  return a+b;`
`}`
 	
`int sub(int a, int b){`
`  return add(a,-b);`
`}`
 	
`int main(){`
`  int i = times(add(1,2),5);`
`  sub(i,6);`
`}`
	`push main     |____main____|`
	
	`push add      |    main    |`
	`              |____add_____|`
	
	`pop           |____main____|`
	
	`push times    |    main    |`
	`              |____times___|`
	
	`pop           |____main____|`
	
	`push sub      |    main    |`
	`              |____sub_____|`
	
	`push add      |    main    |`
	`              |    sub     |`
	`              |____add_____|`
	
	`pop           |    main    |`
	`              |____sub_____|`
	
	`pop           |____main____|`

4. (10 points) Consider allocating an array of 16 long integers:

```
long * larray = /*allocate with calloc and malloc*/
```

Write a C statement using `malloc()` to allocate `larray`:

```
long * larray = malloc(16 * sizeof(long));
```

Write a C statement using `calloc()` to allocate `larray`:

```
long * larray = calloc(16, sizeof(long));
```

5. (20 points) Consider the following program, which employs a double-array of a custom typed struct. Complete the deallocation routine `dealloc()`, such that there are no memory violations/leaks (Try by actually programming it.). Put your routine in the space on the right.

```c
/* mytype_todo.c */
#include <stdio.h>
#include <stdlib.h>

typedef struct{
  int * a; //array of ints
  int size; //of this size
}mytype_t;


mytype_t ** allocate(int n){
  mytype_t ** mytypes;
  int i,j;

  mytypes = calloc(n,sizeof(mytype_t*));
  for(i=0;i<n;i++){
    mytypes[i] =
      malloc(sizeof(mytype_t));

    mytypes[i]->a =
      calloc(i+1,sizeof(int));

    for(j=0;j<i+1;j++){
      mytypes[i]->a[j] = j*10;
    }

    mytypes[i]->size = i;
  }

  return mytypes;
}
```

```c
void dealloc(int n, mytype_t ** items){
  int i;
  for(i=0;i<n;i++){
    free(items[i]->a);
    free(items[i]);
  }
  free(items);
}

int main(){
  int i,j;
  mytype_t ** mytypes;

  mytypes = allocate(10);

  for(i=0; i<10; i++){
    printf("mytypes[%d] = [",i);
    for(j=0;j<mytypes[i]->size;j++){
      printf(" %d", mytypes[i]->a[j]);
    }
    printf(" ]\n");
  }

  dealloc(10, mytypes);
}
```

6. (15 points) Consider the code below that prints the bytes of the integer `a` in hexadecimal, one byte at a time.

```
#include <stdio.h>
#include <stdlib.h>

int main(){

  unsigned int a = 0xdeadbeef;
  unsigned char *p = (unsigned char *) &a;
  int i;

  for(i=0;i<4;i++){
    printf("%d: 0x%02x\n", i, p[i]);
  }

}
```

What is the output?

```
0: 0xef
1: 0xbe
2: 0xad
3: 0xde
```

Explain the output using the terms "Big Endian" or "Little Endian".

This output shows that the architectural format used for data representation by this architecture is Little Endian: the least significant byte of a multi-byte value is stored at the lowest memory address, so now that the memory address is read in order (low to high) the bytes appear backwards.