HW 5

Joshua Thompson

13 FEB 2017

1. (20 points) Consider the program below (and, yes!, you should run this program to understand it further):

```
#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);
}
```

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

   > In the function makearray the returned value is only the pointer to the array, not the array itself. The array will be discarded when the function is exited.

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

   > In the first line of the makearray function replace:
   > `int array[size];`
   > with
   > `int* array = (int*)calloc(size, sizeof(int));`

   (c) Explain how your correction to `makearray()` removes the memory violation.

The correction removes the memory violation because when the pointer variable is passed, the memory it is pointing to is still there. It doesnt disappear when the function is exited.

2. (15 points) For the code below, draw the function stack diagram at each *push* (function call) and *pop* (function return)

```
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        |    add     |
                |____main____|

return 3 (pop)  |____main____|

push times      |   times    |
                |____main____|

return 15 (pop)|    main     |

push sub        |    sub     |
                |____main____|

                |    add     |
push add        |    sub     |
                |____main____|

return 9 (pop)  |    sub     |
                |____main____|

return 9 (pop)  |____main____|

exit program
```

3. (10 points) Consider allocating an array of 16 long's: write two C expression using sing `malloc()` and one using `calloc()`.

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

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

4. (10 points) What is one advantage of using `malloc()` over `calloc()`? What is one advantage of using `calloc()` over `malloc()`?

One advantage of malloc over calloc is that you have more control over what will actaully go into the memory space, whether it is going be an array or something else. Calloc's advantage over malloc is that it is simpler to use with repect to arrays. Makes code easier to read.

5. (20 points) Consider the following program, complete the deallocation routine such that there are no memory violations/leaked. (Yes! You should try programming it.)

```
#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;
}

void deallocate(mytype_t ** mytypes){

  /*Complete me*/

}

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

  mytypes = allocate(10);


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

  deallocate(mytypes);
}
```

```
void deallocate(mytype_t ** mytypes)
{

  int i,j;
  for(i=0;i<10;i++){
    free(mytypes[i]->a);
    free(mytypes[i]);
    }
  free(mytypes);
}
```

6. (15 points) Consider the code below that prints the bytes of the integer `a` in hexadecimal.

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

int main(){
  unsigned int a = 0xcafebabe;
  unsigned char *p = (unsigned char *) &a;
  int i;

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

}
```

(a) What is the output?

The bytes of the unsigned int just printed in the wrong order.

(b) Explain the output using the terms "Big Endian" and "Little Endian".

Becuase the computers we use use the Little Endian architecture for data representation, the order of the bytes is switched around so that the computer more easily recognizes it. The difference between Big Endian and Little Endian is that Little Endian puts the least significant bytes before the more significant ones.