# HW 5

Mike Hanling

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.

The function makearray() returns the address of a locally defined array.
This yields a Seg Fault because the memory allocated for the local makearray instance (its stack) is cleared when it returns, but the main tries to access part of that memory which is not allowed.

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

```
int * makearray(int size,int base){

  int* array = calloc(size, sizeof(int));
  int j;
```

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

    return array;
  }
```

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

This now declares the array to return on the heap which is accessable everywhere in a program. Now the main can access the array just fine.

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);
}
```

Diagram start:

```
push main      |____main____|

push add       |    add     |
               |____main____|

  ...
```

```
push main      |____main____|

push add       |    add     |
               |____main____|

pop            |____main____|

push times     |____times___|
               |____main____|

pop            |____main____|

push sub       |____sub_____|
               |____main____|

push add       |____add_____|
               |____sub_____|
               |____main____|

pop            |____sub_____|
               |____main____|

pop            |____main____|

pop            |____________|
```

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

Malloc does not have to go and clear the allocated memory, therefore it is faster. Calloc does clear the memorym so this makes using very easy because you can assume such.

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){

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

```
0: 0xbe
1: 0xba
2: 0xfe
3: 0xca
```

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

Big Endian stores bytes how humans read: most significant bytes first.
Little Endian does the opposite: least significant bytes first.
Since most architectures (including the one I used) is Little Endian, printing the bytes "in order" from memory returns what we would understand as backwards.