HW 5: Memory Name \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

IC221, Spring AY22 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:  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): |

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)

|  |
| --- |
|  |

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\_\_\_\_\_| |

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:

|  |
| --- |
|  |

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

|  |
| --- |
|  |

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.

|  |  |
| --- | --- |
| /\* 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;  } | void **dealloc**(int n, mytype\_t \*\* 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?

|  |
| --- |
|  |

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

|  |
| --- |
|  |