Name:_____ UID:_____ Discussion #: _____

1. How many bytes would the following array declaration allocate on a 64-bit machine?

```
char *arr[10][6];
```

2. What will the following print out?

```
typedef struct {
    char alice;
    int bob;
    char charlie;
    double dave;
} bt;

int main(void) {
    bt band[7];
    printf("%zu\n", sizeof(band));
}
```

3. What is the best* ordering of the following data types if you want to have a struct that uses all of them? What is this optimal size? In general, what is the way to order variables to guarantee that the struct will have the minimal size?
   ** the ordering that will result in the optimal usage of space – there's more than 1 answer!*

```
short apple;
double banana;
int cherry;
char durian;
char* elderberry;

struct my_struct {









};
```

4. Consider the following disassembled function:

```
000000000040102b <phase_2>:
  40102b:  55                    push   %rbp
  40102c:  53                    push   %rbx
  40102d:  48 83 ec 28           sub    $0x28,%rsp
  401031:  48 89 e6              mov    %rsp,%rsi
  401034:  e8 e3 03 00 00        callq  40141c <read_six_numbers>
  401039:  83 3c 24 01           cmpl   $0x1,(%rsp)
```

When the `read_six_numbers` function is entered (the next instruction to be executed after the `callq`), what is the value at the top of the stack?

5. For each of the following pieces of assembly, circle if the jump is taken:

`movq $5, %rax`
`movq $-7, %rbx`
`cmpq %rax, %rbx`
`jge .something`
	jump taken / jump not taken

`movq $5, %rax`
`movq $-7, %rbx`
`cmpq %rax, %rbx`
`ja .something`
	jump taken / jump not taken

`movq $8, %rax`
`movq $4, %rbx`
`subq %rax, %rbx`
`je .something`
	jump taken / jump not taken

`movq $-5, %rax`
`testq %rax, %rax`
`jl .something`
	jump taken / jump not taken

`movq $9, %rax`
`testq %rax, %rax`
`jne .something`
	jump taken / jump not taken

6. Consider the function with the following type signature:

```
int func(int a1, int a2, int a3, int a4,
         int a5, int a6, int a7, int a8);
```

And the following GDB output, taken at a breakpoint on the first instruction of `func`:

```
(gdb) info registers
rax            0x401158            0x401158
rbx            0x7fffffffd558      0x7fffffffd558
rcx            0xef                0xef
rdx            0x85                0x85
rsi            0x9                 0x9
rdi            0xfa                0xfa
rbp            0x7fffffffd440      0x7fffffffd440
rsp            0x7fffffffd428      0x7fffffffd428
r8             0x5b                0x5b
r9             0x7f                0x7f
r10            0x7fffffffd170      0x7fffffffd170
r11            0x206               0x206
r12            0x0                 0x0
r13            0x7fffffffd568      0x7fffffffd568
r14            0x7ffff7ffd000      0x7ffff7ffd000
r15            0x403df0            0x403df0
rip            0x401126            0x401126

(gdb) x/8gx $rsp
0x7fffffffd428: 0x0000000000401188    0x00000000000000e5
0x7fffffffd438: 0x000000000000006c    0x0000000000000001
0x7fffffffd448: 0x00007ffff7daecd0    0x00007fffffffd540
0x7fffffffd458: 0x0000000000401158    0x0000000100400040
```

What is the address of `func`?

What address will the function return to?

What are the values of the 8 arguments? (leaving your answer in hex is fine)

7. Consider the following C code:

```
typedef struct {
      char first;
      int second;
      short third;
      int* fourth;

} stuff;

stuff array[5];

int func0(int index, int pos, long dist) {
      char* ptr = (char*) &(array[index]._________);
      ptr += pos;
      *ptr = _________ + dist;

      return *ptr;
}

int func1() {
      int x = func0(1, ____, _____);
      return x;
}
```

Clearly some code is missing - your job is to fill in the blanks! Note that the size of the blanks is not significant. The two functions will be compiled using the following assembly code:

```
0000000000400492 <func0>:
  400492:   8d 04 17                lea    (%rdi,%rdx,1),%eax
  400495:   48 63 ff                movslq %edi,%rdi
  400498:   48 63 f6                movslq %esi,%rsi
  40049b:   48 8d 14 7f             lea    (%rdi,%rdi,2),%rdx
  40049f:   88 84 d6 60 10 60 00    mov    %al,0x601060(%rsi,%rdx,8)
  4004a6:   0f be c0                movsbl %al,%eax
  4004a9:   c3                      retq

00000000004004aa <func1>:
  4004aa:   c6 05 cb 0b 20 00 0d    movb   $0xd,0x200bcb(%rip)
                                           # 60107c <array+0x1c>
  4004b1:   b8 0d 00 00 00          mov    $0xd,%eax
  4004b6:   c3                      retq
```

You may notice that there's no call to func0 inside func1: this is because the function call has been optimized out due to an optimization known as constant propagation. Your goal is to find the arguments to func0 that would have the same effect as the assembly inside func1.