Name: SOLUTIONS	UID: CS 33

1. What's that Size?

What will be printed after running from the following code-snipet?

```
1  typedef struct {
2      char Rumi;
3      int Mira;
4      char Zoey;
5      double Celine;
6  } Huntrx;
7
8  typedef union {
9      char Jinu;
10     int Mystery;
11     char Abby;
12     double Romance;
13 } SajaBoys;
14
15 int main(int argc, char** argv) {
16     Huntrx band1[7];
17     SajaBoys band2[7];
18     printf("%d\n", (int)sizeof(band1));
19     printf("%d\n", (int)sizeof(band2));
20     return 0;
21 }
```

For the Struct:

- $(1 + (3) + 4 + 1 + (7) + 8) * 7 = 168$
- Due to alignment, we need to add the numbers in parentheses

For the Union:

- $8 * 7 = 56$

2. Struct ordering and optimal size

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? Assume a 64-bit architecture. The best ordering means the one that results in optimal space usage — there's more than one valid answer!

```
char tully;
long stark;
int greyjoy;
float* lannister;
float arryn;
double targaryen;

struct Westeros {
    /* order the above variables here */
    // Note:  this is one possible ordering
    // There are many others that work as well!
    float* lannister; // ALL pointers are 8 bytes
    double targaryen; // doubles are 8 bytes
    long stark; // longs are 8 bytes
    float arryn; // floats are 4 bytes
    int greyjoy; // ints are 4 bytes
    char tully; // chars are 1 byte
};
```

One simple strategy (the one used above) is to order the fields from the largest size to smallest, as structs are x-aligned, where x is the size of the largest data type in the struct

3. Tricky Switch

Reverse engineer the assembly code on the previous page to figure out what each case of the switch-case statement is doing. Don't forget about "break" statements!

Source Code (C) (fill in blanks!)

```
int func(int x, int y, int r)
    {
    switch (x) {
      case 0: return -4;

      case 1:
         return 5*(r+6);

      case 2: return 5*r;

      case 3: return r+(2*y);

      case 4: return r+(2*y);

      case 5: return r^y;
    }

    return  return r;
}
```

Compiled Assembly

```
func(int, int, int):
        cmpl    $5, %edi
        ja      .L9
        movl    %edi, %edi
        jmp     *.L4(,%rdi,8)

.L4:    #hint, this is the jump table!
        .quad   .L8
        .quad   .L7
        .quad   .L6
        .quad   .L5
        .quad   .L5
        .quad   .L3
.L7:
        addl    $6, %edx
.L6:
        leal    (%rdx,%rdx,4), %eax
        ret
.L5:
        leal    (%rdx,%rsi,2), %eax
        ret
.L3:
        movl    %edx, %eax
        xorl    %esi, %eax
        ret
.L8:
        movl    $-4, %eax
        ret
.L9:
        movl    %edx, %eax
        ret
```

There are multiple answers (break statements or continue to next case). To read the jumptable, you take the case index (i.e. 0) and jump to the label within the list. Thus at 2, you would jump to L6.
Note: This was Question 6 on Fall 2021 Midterm.

4. Fill in the missing C code

```c
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].first);
    ptr += pos;
    *ptr = index + dist;
    return *ptr;
}

int func1() {
    int x = func0(1, 4, 12);
    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:

```
0000000000401106 <func0>:
  401106:    48 63 c7                movslq %edi,%rax
  401109:    48 8d 04 40             leaq   (%rax,%rax,2),%rax
  40110d:    48 63 f6                movslq %esi,%rsi
  401110:    01 d7                   addl   %edx, %edi
  401112:    40 88 bc c6 60 40 40 00 movb   %dil, 0x404060(%rsi,%rax,8)
                                              # 0x404060 <array>
  40111a:    40 0f be c7             movsbl %dil, %eax
  40111e:    c3                      retq

000000000040111f <func1>:
  40111f:     ba 0c 00 00 00         movl   $0xc,%edx
  401124:     be 04 00 00 00         movl   $0x4,%esi
  401129:     bf 01 00 00 00         movl   $0x1,%edi
  40112e:     e8 d3 ff ff ff         callq  401106 <func0>
  401133:     c3                     retq
```

What operation does this function do?

The function replaces the char at (array[index] + pos) with the result of index + dist (which is 13 or 0xd). Thus, the array will be changed such that array[1].first would equal 13. The function returns this changed value.

First, we analyze the struct **stuff**. The size of stuff is equal to:

- $1 + (3) + 4 + 2 + (2) + 8 + (4) = 24$
- The () numbers are additional padding. Note, that the struct must be aligned to the biggest data type size (8) within it, thus we have the (4) extra bytes of padding.

As there is an array, the total size of the array is $24 * 5 = 120$

Second, we start with Func1. Three values are loaded into argument registers before calling func0.

- %edx (or dist) is given value of 0xc (12)
- %esi (or pos) is given value of 0x4 (4)
- %edi (or index) is given value of 0x1 (1)

Thus, we know the arguments to func0 (first part of solution.

Now, we will go line-by-line analyzing func0.

- movslq %edi,%rax
  - This moves %edi (or index) into the %rax register
- leaq (%rax,%rax,2),%rax
  - This performs the operation $\%rax + (\%rax * 2)$ or $\%rax * 3$
  - Thus, %rax now holds $index * 3$
- movslq %esi,%rsi
  - This sign-extends %esi (lower 32-bits of %rsi) in the %rsi register. This is done in preperation for addition operation below.
  - %rsi still holds the pos (sign-extended to 64 bits)
- addl %edx, %edi
  - This adds the %edx and %edi registers together and stores the result into the edi register.
  - Thus, %edi = %edi (index) + %edx (dist)
- movb %dil, 0x404060(%rsi,%rax,8)
  - %dil is the lowest byte of the edi register.
  - 0x404060 is the start position of array.
  - This instruction takes %dil and places it into memory at the position of $array + \%rsi + \%rax * 8$
  - $\%rsi + \%rax * 8$ is equal to $pos + index * 24$ as %rsi stores pos, and %rax stores index * 3
  - Note that 24 is the size of stuff. Thus, index would start on the first item within each item of the array.
- movsbl %dil, %eax
  - This sign-extends the lowest byte of edi and places it into the %rax register.

Thus, based on these results we can see that the line 11 blank is first (as it starts on the first element within stuff) and line 13 blank is index as we are doing index + dist.