Example Structured Data And Stack Problems

Problem 1:

Consider the following source code and corresponding assembly:

```
#include <stdio.h>
char read_char()
{
    char c;
    scanf("%s",&c);
    return c;
}

int main()
{
    printf("%c\n", read_char());
}
```

```
080484c0 <read_char>:
 80484c0:       55                      push   %ebp
 80484c1:       89 e5                   mov    %esp,%ebp
 80484c3:       83 ec 10                sub    $0x10,%esp
 80484c6:       8d 45 ff                lea    0xffffffff(%ebp),%eax
 80484c9:       50                      push   %eax
 80484ca:       68 98 85 04 08          push   $0x8048598
 80484cf:       e8 6c fe ff ff          call   8048340 <_init+0x38>
 80484d4:       0f be 45 ff             movsbl 0xffffffff(%ebp),%eax
 80484d8:       89 ec                   mov    %ebp,%esp
 80484da:       5d                      pop    %ebp
 80484db:       c3                      ret
 80484dc:       8d 74 26 00             lea    0x0(%esi,1),%esi

080484e0 <main>:
 80484e0:       55                      push   %ebp
 80484e1:       89 e5                   mov    %esp,%ebp
 80484e3:       83 ec 08                sub    $0x8,%esp
 80484e6:       83 e4 f0                and    $0xfffffff0,%esp
 80484e9:       83 ec 10                sub    $0x10,%esp
 80484ec:       e8 cf ff ff ff          call   80484c0 <read_char>
 80484f1:       83 c4 08                add    $0x8,%esp
 80484f4:       0f be c0                movsbl %al,%eax
 80484f7:       50                      push   %eax
 80484f8:       68 9b 85 04 08          push   $0x804859b
 80484fd:       e8 6e fe ff ff          call   8048370 <_init+0x68>
 8048502:       89 ec                   mov    %ebp,%esp
 8048504:       5d                      pop    %ebp
 8048505:       c3                      ret
```

When the program is executed, a breakpoint is placed at the address 0x080484d4. The user inputs the single character 'a' (ASCII value 0x61). Assume that when the program is run the value of `%esp` immediately before the first instruction of `main` (0x080484e0) is executed is 0xbffff6ec. Assume that the entire stack space was initialized to zeroes before the program began executing and `main` is the first function to execute. Fill in the following table with the values observed to be on the stack when the program halts at the breakpoint. Clearly indicate (with labeled arrows) what the observed values of the registers `%ebp` and `%esp` would be. If the exact hexadecimal value cannot be determined from the information provided, but the value has a meaning (ie, return address, saved ebp), write the meaning. Your answers should be the 4-byte hexadecimal values at the stack location (**remember this is a little endian machine**. You do not need to write out all 8 digits of the hexadecimal number zero.

Address	Hex Value
0xbffff6ec	
0xbffff6e8	
0xbffff6e4	
0xbffff6e0	
0xbffff6dc	
0xbffff6d8	
0xbffff6d4	
0xbffff6d0	
0xbffff6cc	
0xbffff6c8	
0xbffff6c4	
0xbffff6c0	
0xbffff6bc	
0xbffff6b8	
0xbffff6b4	
0xbffff6b0	
0xbffff6ac	
0xbffff6a8	
0xbffff6a4	
0xbffff6a0	

If we continue executing the program (with the user having entered ’a’), will we successfully return from the function `read_char`?

If we continue executing the program (with the user having entered ’a’), will we successfully return from the function `main`?

Suppose the user had input the string ‘‘`This is a very long string.`’’. Would we successfully return from the function `read_char`?

Suppose the user had input the string ‘‘`This is a very long string.`’’. Would we successfully return from the function `main`?

Problem 2:

Consider the source code below, where `M` and `N` are constants declared with `#define`.

```
int array1[M][N];
int array2[N][M];

int copy(int i, int j)
{
  array1[i][j] = array2[j][i];
}
```

Suppose the above code generates the following assembly code:

```
copy:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        movl    8(%ebp), %ecx
        movl    12(%ebp), %edx
        leal    (%ecx,%ecx,2), %ebx
        leal    (%edx,%ebx,2), %ebx
        leal    0(,%edx,8), %eax
        subl    %edx, %eax
        addl    %ecx, %eax
        movl    array2(,%eax,4), %eax
        movl    %eax, array1(,%ebx,4)
        popl    %ebx
        popl    %ebp
        ret
```

What are the values of `M` and `N`?

`M` =

`N` =

Problem 3:

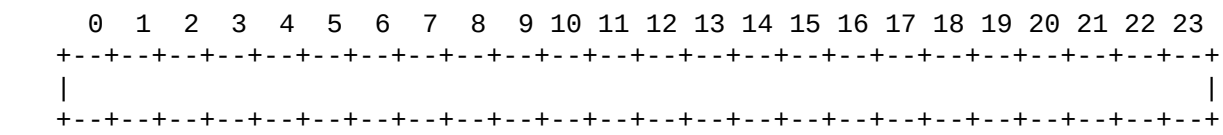
onsider the following C declarations:

```
typedef struct {
    short prime;
    short rib;
    char raw[3];
    long filet;
} cow;

typedef struct {
    short prime;
    long rib;
    char raw[5];
    short roast;
    long filet;
} bull;
```

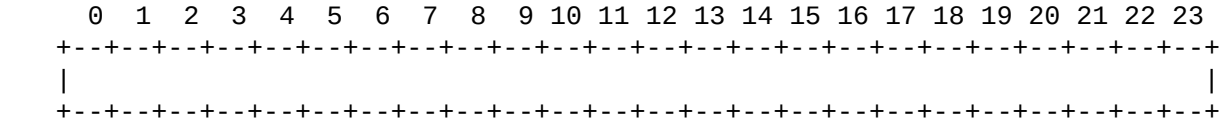
A. Using the templates below (allowing a maximum of 24 bytes), indicate the allocation of data for structs of type `cow` and `bull`. Mark off and label the areas for each individual element (arrays may be labeled as a single element). **Cross hatch the parts that are allocated, but not used (to satisfy alignment).**

Assume the Linux alignment rules discussed in class. **Clearly indicate the right hand boundary of the data structure with a vertical line.**

`cow:`



`bull:`



B. Now consider the following C code fragment:

```
void foo(cow *oldcow)
{
    bull *newcow;

    /* this zeros out all the space allocated for oldcow */
    bzero((void *)oldcow, sizeof(cow));

    oldcow->prime = 0xabcd;
    oldcow->rib = 0x1234;
    oldcow->raw[0] = 0x01;
    oldcow->raw[1] = 0x30;
    oldcow->raw[2] = 0x79;
    oldcow->raw[-3] = 0xff;
    oldcow->filet = 0x66778855;

    newcow = (bull *) oldcow;

    ...
```

Once this code has run, we begin to access the elements of `newcow`. Below, give the value of each element of `newcow` that is listed. Assume that this code is run on a Little-Endian machine such as a Linux/x86 machine. You must give your answer in hexadecimal format. **Be careful about byte ordering!**.

(a) `newcow->prime  = 0x`_____

(b) `newcow->rib  = 0x`_____

(c) `newcow->raw[0]  = 0x`_____

(d) `newcow->raw[3]  = 0x`_____

(e) `newcow->raw[-5]  = 0x`_____

Problem 4:

This problem concerns the following C code. This program reads a string on standard input and prints an integer in hexadecimal format based on the input string it read.

```
#include <stdio.h>

/* Read a string from stdin into buf */
short read_string()
{
    short buf[3];

    scanf("%s",(char*)buf);
    return buf[3];
}

int main()
{
    printf("0x%x\n", read_string());
}
```

Here is the corresponding machine code on a Linux/x86 machine:

```
08048414 <read_string>:
 8048414:       55                      push   %ebp
 8048415:       89 e5                   mov    %esp,%ebp
 8048417:       83 ec 18                sub    $0x18,%esp
 804841a:       83 c4 f8                add    $0xfffffff8,%esp
 804841d:       8d 45 f8                lea    0xfffffff8(%ebp),%eax
 8048420:       50                      push   %eax
 8048421:       68 b8 84 04 08          push   $0x80484b8
 8048426:       e8 e1 fe ff ff          call   804830c <_init+0x50>
 804842b:       0f bf 45 fe             movswl 0xfffffffe(%ebp),%eax
 804842f:       89 ec                   mov    %ebp,%esp
 8048431:       5d                      pop    %ebp
 8048432:       c3                      ret
 8048433:       90                      nop
08048434 <main>:
 8048434:       55                      push   %ebp
 8048435:       89 e5                   mov    %esp,%ebp
 8048437:       83 ec 08                sub    $0x8,%esp
 804843a:       83 c4 f8                add    $0xfffffff8,%esp
 804843d:       e8 d2 ff ff ff          call   8048414 <read_string>
 8048442:       98                      cwtl
 8048443:       50                      push   %eax
 8048444:       68 bb 84 04 08          push   $0x80484bb
 8048449:       e8 ee fe ff ff          call   804833c <_init+0x80>
 804844e:       89 ec                   mov    %ebp,%esp
 8048450:       5d                      pop    %ebp
 8048451:       c3                      ret
```

This problem tests your understanding of the stack discipline and byte ordering. Here are some notes to help you work the problem:

- `scanf("%s", buf)` reads an input string from the standard input stream (stdin) and stores it at address `buf` (including the terminating '`\0`' character). It does **not** check the size of the destination buffer.
- `printf("0x%x", i)` prints the integer i in hexadecimal format preceded by "`0x`".
- Recall that Linux/x86 machines are Little Endian.
- You will need to know the hex values of the following characters:

Character	Hex value	Character	Hex value
'd'	0x64	'v'	0x76
'r'	0x72	'i'	0x69
'.'	0x2e	'l'	0x6c
'e'	0x65	'\0'	0x00
		's'	0x73

A. Suppose we run this program on a Linux/x86 machine, and give it the string "`dr.evil`" as input on stdin.

Here is a template for the stack, showing the location of `buf`. Fill in the values written into the stack by scanf (in hexadecimal) and indicate where `ebp` points just **after** `scanf` returns to `read_string`.

```
                |<- buf
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
         ---------- addresses increase this way ------>
```

What is the 2-byte short (in hex) printed by the `printf` inside main?

`0x`_____

B. Suppose now we give it the input "`dr.evil.lives`" (again on a Linux/x86 machine).

(a) List the contents of the following memory locations just **after** `scanf` returns to `evil_read_string`. Each answer should be an unsigned 4-byte integer expressed as 8 hex digits. Remember that `buf` is a pointer to shorts.

```
*((int*)buf) = 0x__________________________

*((int*)(buf+6)) = 0x__________________________
```

(b) Immediately **before** the `ret` instruction at address `0x08048435` executes, what is the value of the frame pointer register `%ebp`?

```
%ebp  = 0x__________________________
```

You can use the following template of the stack as *scratch space*.

```
               <- buf
--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--
  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--
```

Problem 5:

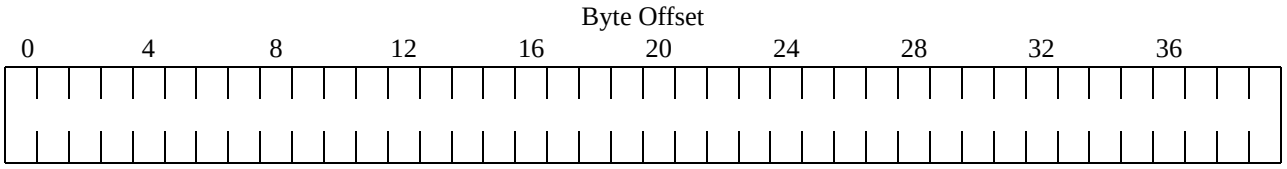
Please note the following:

- Under the IA-32 Windows alignment convention values of type `double` must be 8-byte aligned (vs. the Linux convention where they are only 4-byte aligned).
- Also note that unions have the same alignment requirements for their initial address and length as do structs in C.
- The `sizeof` function returns the amount of space a data type takes up in memory including any unused space.
- There are no syntax errors in the provided code.

Consider the following program:

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

typedef union
{
 char s[3];
 int i;
 double d;
} union1;

typedef struct
{
 char s;
 short t;
 int i;
 double *d;
 union1 un;
} struct1;

int main()
{
 struct1 s1;
 union1 u1;

 printf("union1:   &s=%#x &i=%#x  &d=%#x\n",&u1.s,&u1.i, &u1.d);
 printf("struct1:  &s=%#x  &t = %#x &i=%#x  \n\t  &d=%#x &un=%#x\n", \
                        &s1.s, &s1.t, &s1.i, &s1.d, &s1.un);
 printf("sizeof(struct1) = %d\n",sizeof(struct1));
 printf("sizeof(union1) = %d\n",sizeof(union1));
}
```

The following template is provided as an aid to help you solve this problem. You do not have to use it and **anything written in this template will not be graded**.



A. This program is compiled and executed on an i686 running Linux. Fill in the missing output:

```
union1:   &s=0xbffff84c &i= 0x_________________  &d= 0x_________________

struct1:  &s=0xbffff854  &t = 0x_________________ &i= 0x_________________

          &d= 0x_________________ &un= 0x_________________

sizeof(struct1) = _________________

sizeof(union1) = _________________
```

B. The same program is now compiled and run in a Windows environment. Assume that both union1 and struct1 end up with the same initial address. Circle the parts of your answer (if any) that would not be correctly aligned with the Windows alignment convention.

C. Taking *only* alignment issues into account, is the assumption that **both** union1 and struct1 end up with the same initial address under Windows as under Linux a valid assumption? Why or why not? Answer with one sentance. Make sure you answer for both union1 and struct1.

D. Consider the following code where R and C are constants defined earlier in the file.

```
typedef int RC_matrix[R][C];

int lollapalooza(RC_matrix m)
{
  int i = 0;
  int ret = 0;
  for(i = 0; i < R; i++)
  {
    ret += m[i][i];
  }
  return ret;
}
```

The x86 assembly for lollapalooza is given below:

```
00000000 <lollapalooza>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   53                      push   %ebx
   4:   8b 5d 08                mov    0x8(%ebp),%ebx
   7:   31 c9                   xor    %ecx,%ecx
   9:   89 ca                   mov    %ecx,%edx
   b:   90                      nop
   c:   8d 74 26 00             lea    0x0(%esi,1),%esi
  10:   8d 04 52                lea    (%edx,%edx,2),%eax
  13:   c1 e0 04                shl    $0x4,%eax
  16:   03 0c 03                add    (%ebx,%eax,1),%ecx
  19:   42                      inc    %edx
  1a:   83 fa 06                cmp    $0x6,%edx
  1d:   7e f1                   jle    10 <lollapalooza+0x10>
  1f:   89 c8                   mov    %ecx,%eax
  21:   5b                      pop    %ebx
  22:   89 ec                   mov    %ebp,%esp
  24:   5d                      pop    %ebp
  25:   c3                      ret
%
```

What are the values of R and C? Be careful to not make an off-by-one error.

R =

C =

Problem 6:

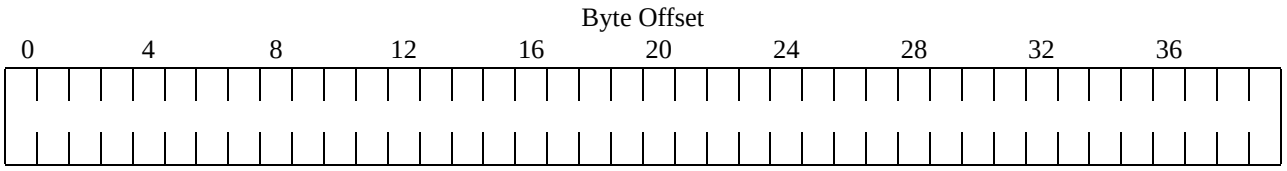
Consider the following C type definitions:

```
typedef struct {          typdef union {
     char c[2];                char buf[2];
     char *str;                long l;
     double d;                 float f;
     union1 u;            } union1
} struct1
```

Note: Assume the IA-32 **Windows** alignment convention for this problem – in particular, values of the type **double** must be 8-byte aligned. (vs. Linux where they are only 4-byte aligned).

A. What byte alignment is necessary for an object of type union1?

B. On the below template, draw out the allocation of data for an object of type struct1. Allowing for a maximum of 36 bytes, mark off and label each element in the struct as well as cross-hatch the unused, but allocated space. Clearly indicate the right hand boundary with a vertical line.



C. How many bytes need to be allocated for an object of type struct1?

D. Think about redefining the fields of struct1 in a different order so as to minimize the amount of unused, but allocated space for each variable of type struct1. Now how many bytes are required for an struct1 object?

Problem 7:

Consider the source code below, where `M` and `N` are constants declared with `#define`.

```
int squirrel[M][N];
int moose[N][M];

int bullwinkle(int i, int j)
{
  squirrel[i][j] = moose[j][i];
}
```

Suppose the above code generates the following assembly code:

```
bullwinkle:
        pushl   %ebp
        movl    %esp, %ebp
        movl    12(%ebp), %edx
        pushl   %ebx
        movl    8(%ebp), %ecx
        leal    (%edx,%edx,4), %eax
        leal    (%edx,%eax,2), %eax
        addl    %ecx, %eax
        leal    (%edx,%ecx,2), %ebx
        movl    moose(,%eax,4), %eax
        movl    %eax, squirrel(,%ebx,4)
        popl    %ebx
        popl    %ebp
        ret
```

What are the values of `M` and `N`?

`M` =

`N` =