Andrew login ID:______________________________________

Full Name:______________________________________

CS 15-213, Spring 2003

Exam 1

February 27, 2003

Instructions:

- Make sure that your exam is not missing any sheets, then write your full name and Andrew login ID on the front.
- Write your answers in the space provided below the problem. If you make a mess, clearly indicate your final answer.
- The exam has a maximum score of 60 points.
- The problems are of varying difficulty. The point value of each problem is indicated. Pile up the easy points quickly and then come back to the harder problems.
- This exam is OPEN BOOK. You may use any books or notes you like. No electronic devices are allowed. Good luck!

1 (1):	
2 (9):	
3 (8):	
4 (4):	
5 (8):	
6 (12):	
7 (10):	
8 (8):	
TOTAL (60):	

Problem 1. (1 points):

The correct answer to this problem is worth 1 point. An incorrect answer is worth -2 points. No answer will be scored as 0 points. Note: The answer to this question was given in lecture.

The correct answer to this problem is: ______________

Problem 2. (9 points):

Assume we are running code on an 8-bit machine using two's complement arithmetic for signed integers. Short integers are encoded using 4 bits. Sign extension is performed whenever a short is casted to an int. For this problem, assume that all shift operations are arithmetic. Fill in the empty boxes in the table below.

```
int i = -11;
unsigned ui = i;
short s = -2;
unsigned short us = s;
```

Note: You need not fill in entries marked with "–". TMax denotes the largest positive two's complement number and TMin denotes the minimum negative two's complement number. Finally, you may use hexidecimal notation for your answers in the "Binary Representation" column.

Expression	Decimal Representation	Binary Representation
Zero	0	
–	-3	
i		
i $>>$ 4		
ui		
(int) s		
(int)(s ˆ 7)		
(int) us		
TMax		
TMin		

Problem 3. (8 points):

Consider the following 7-bit floating point representation based on the IEEE floating point format:

- There is a sign bit in the most significant bit.
- The next 3 bits are the exponent. The exponent bias is 3.
- The last 3 bits are the fraction.
- The representation encodes numbers of the form: $V = (-1)^s \times M \times 2^E$, where M is the significand and E the integer value of the exponent.

Please fill in the table below. You do not have to fill in boxes with "——" in them. **If a number is NAN or infinity, you may disregard the M, E, and V fields below.** However, fill the Description and Binary fields with valid data.

Here are some guidelines for each field:

- **Description -** A verbal description if the number has a special meaning
- **Binary -** Binary representation of the number
- M **-** Significand (same as the M in the formula above)
- E **-** Exponent (same as the E in 2^E)
- V **-** Fractional Value represented

Please fill the M, E, and V fields below with rational numbers (fractions) rather than decimals or binary decimals

Description	Binary	M	E	V
——	`0 010 010`			
$2\ \frac{3}{8}$				
	`1 111 000`			
Most Negative **Normalized**				
Smallest Positive **Denormalized**				

Problem 4. (4 points):

Consider the following assembly instructions and C functions:

```
080483b4 <funcX>:
 80483b4:  55                      push   %ebp
 80483b5:  89 e5                   mov    %esp,%ebp
 80483b7:  8b 45 08                mov    0x8(%ebp),%eax
 80483ba:  8d 04 80                lea    (%eax,%eax,4),%eax
 80483bd:  8d 04 85 f6 ff ff ff    lea    0xfffffff6(,%eax,4),%eax
 80483c4:  89 ec                   mov    %ebp,%esp
 80483c6:  5d                      pop    %ebp
 80483c7:  c3                      ret
```

Circle the C function below that generates the above assembly instructions:

```
int func1(int n) {

  return n * 20 - 10;

}

int func2(int n) {

  return n * 24 + 6;

}

int func3(int n) {

  return n * 16 - 4;

}
```

Problem 5. (8 points):

Consider the following pairs of C functions and assembly code. Draw a line connecting each C function with the block(s) of assembly code that implements it. **There is not necessarily a one-to-one correspondence. Draw an X through any C and/or assembly code fragments that don't have a match.**

```
int scooby(int *a)
{
    return a[4];
}
```

```
pushl   %ebp
movl    %esp, %ebp
movl    8(%ebp), %eax
sall    $2, %eax
popl    %ebp
ret
```

```
int dooby(int a)
{
    return a*4;
}
```

```
pushl   %ebp
movl    %esp, %ebp
movl    8(%ebp), %eax
movsbl  4(%eax),%eax
popl    %ebp
ret
```

```
int doo(int a)
{
    return a<<4;
}
```

```
pushl   %ebp
movl    %esp, %ebp
movl    8(%ebp), %eax
sall    $4, %eax
popl    %ebp
ret
```

```
char scrappy(char *a)
{
    return a[4];
}
```

```
pushl   %ebp
movl    %esp, %ebp
movl    8(%ebp), %eax
leal    0(,%eax,4), %eax
popl    %ebp
ret
```

Problem 6. (12 points):

Recently, Microsoft's SQL Server was hit by the SQL Slammer worm, which exploits a known buffer overflow in the SQL Resolution Service. Today, we'll be writing our own *213 Slammer* that exploits the vulnerability introduced in `bufbomb`, the executable used in your Lab 3 assignment. And as such, `Gets` has the same functionality as in Lab 3 except that it strips off the newline character before storing the input string.

Consider the following exploit code, which runs the program into an infinite loop:

```
infinite.o:     file format elf32-i386
Disassembly of section .text:

00000000 <.text>:
   0:   68 fc b2 ff bf          push   $0xbfffb2fc
   5:   c3                      ret
   6:   89 f6                   mov    %esi,%esi
```

Here is a disassembled version of the `getbuf` function in `bufbomb`, along with the values of the relevant registers and a printout of the stack **before** the call to `Gets()`.

```
(gdb) disas
Dump of assembler code for function getbuf:
0x8048a44 <getbuf>:     push   %ebp
0x8048a45 <getbuf+1>:   mov    %esp,%ebp
0x8048a47 <getbuf+3>:   sub    $0x18,%esp
0x8048a4a <getbuf+6>:   add    $0xfffffff4,%esp
0x8048a4d <getbuf+9>:   lea    0xfffffff4(%ebp),%eax
0x8048a50 <getbuf+12>:  push   %eax
0x8048a51 <getbuf+13>:  call   0x8048b50 <Gets>
0x8048a56 <getbuf+18>:  mov    $0x1,%eax
0x8048a5b <getbuf+23>:  mov    %ebp,%esp
0x8048a5d <getbuf+25>:  pop    %ebp
0x8048a5e <getbuf+26>:  ret
0x8048a5f <getbuf+27>:  nop
End of assembler dump.

(gdb) info registers
eax            0xbfffb2fc       ecx            0xffffffff
edx            0x0              ebx            0x0
esp            0xbfffb2e0       ebp            0xbfffb308
esi            0xffffffff       edi            0x804b820

(gdb) x/20xb $ebp-12
0xbfffb2fc:   0xf0  0x17  0x02  0x40  0x18  0xb3  0xff  0xbf
0xbfffb304:   0x50  0x80  0x06  0x40  0x28  0xb3  0xff  0xbf
0xbfffb30c:   0xee  0x89  0x04  0x08  0x24  0xb3  0xff  0xbf
```

Here are the questions:

1. Write down the address of the location on the stack which contains the return address where `getbuf` is supposed to return to:

   `0x______________________`

2. Using the exploit code illustrated above, fill in the the following blanks on the stack **after** the call to `Gets()`. All the numbers must be in a **two character hexadecimal** representation of a byte. We've already filled in the terminating `\0` (0x00) character for you.

   ```
   (gdb) x/20xb $ebp-12

   0xbfffb2fc:   0x___  0x___  0x___  0x___  0x___  0x___  0x___  0x___

   0xbfffb304:   0x___  0x___  0x___  0x___  0x___  0x___  0x___  0x___

   0xbfffb30c:   0x___  0x___  0x___  0x___  0x00  0xb3  0xff  0xbf
   ```

3. During the infinite loop, what is the value of `%ebp`?

   `0x____________________________`

Problem 7. (10 points):

This problem tests your understanding of both control flow and multidimensional array layout. Consider the following assembly code for a procedure `moo()`:

```
moo:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %edi
        pushl   %esi
        pushl   %ebx
        movl    $0, %ecx
        movl    $arr1, %edi
        movl    $arr2+8, %esi
        movl    8(%ebp), %eax
        leal    0(,%eax,4), %ebx
.L5:
        leal    (%ecx,%ecx,2), %eax
        sall    $2, %eax
        movl    %ebx, %edx
        addl    (%eax,%esi), %edx
        movl    %edx, (%eax,%edi)
        incl    %ecx
        cmpl    $10, %ecx
        jle     .L5
        movl    %ecx, %eax
        popl    %ebx
        popl    %esi
        popl    %edi
        popl    %ebp
        ret
```

Based on the assembly code, fill in the blanks below in `moo`'s C source code. (Note: you may only use symbolic variables from the source code in your expressions below-do *not* use register names.) **Hint:** First figure out what the loop variable (i) is in the assembly and what the value of M is.

```
#define M _____

#define N _____


int arr1[M][N];


int arr2[M][N];


int moo(int x)
{
    int i;

    for(__________; i < M; ________ )
    {

       arr1[_____][_____] = arr2[_____][_____]+______________;
    }

    return _______;
}
```

Problem 8. (8 points):

Consider the following C declarations:

```
typedef struct {
  char          ID[2];
  double        weight;
  double        *components;
  short         momentum;
} Projectile;
```

A. Using the templates below (allowing a maximum of 24 bytes), indicate the allocation of data for structs of type `Projectile`. 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, and be sure to clearly indicate the end of the structure. Assume the Linux alignment rules discussed in class.**

`Projectile:`

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                                                       |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

B. How would you define the `Modified` structure to minimize the number of bytes allocated for the structure using the same fields as the `Projectile` structure?

```
typedef struct {



} Modified;
```

C. What is the value of `sizeof(Modified)`?