Joel Mire
Professor Lee
CS/ECE 250
2/1/18

Q1

(a) binary:

$+45_{10}$ = 2^5 + 2^3 + 2^2 + 2^0
= 101101_2 which, in 8b
= 00101101_2

hexadecimal:

consider $+45_{10}$ = 00101101_2 as 0010_2 1101_2
0010_2 = 2_{16} and 1101_2 = D_{16}
$+45_{10}$ = 0x2D

(b) binary:

-35_{10}
35_{10} = 2^5 + 2^1 + 2^0
= 100011_2 which, in 8b
= 00100011_2 which, when inverted
= 11011100_2
\+ 1_2
-35_{10} = 11011101_2

hexadecimal:

consider -35_{10} = 11011101 as 1101_2 1101_2
1101_2 = D_{16} and 1101_2 = D_{16}
-35_{10} = $0xDD_{16}$

(c) IEEE floating point in binary:

47.0
47 = 2^5 + 2^3 + 2^2 + 2^1 + 2^0
= 101111
now considering the .0 as well,
= 101111.00000…
= 1.0111100000… x 10^5
bias of 127 + 5 = 132 which is 10000100
note that the sign is (+), so the first bit is 0. thus,
47.0 = 01000010001111000000000000000000

IEEE floating point in hexadecimal:

consider 01000010001111000000000000000000 as
0100 0010 0011 1100 0000 0000 0000 0000, which in hexadecimal are
4 2 3 c 0 0 0 0 , then
47.0 = 0x423C0000

(d) IEEE floating point in binary:

-0.625
0 = 0
now considering the 0.625,
0.625 x 2 = 1.25 1
0.25 x 2 = 0.5 0
0.5 x 2 = 1 1
0.0 x 0 = 0 0 … so, combining the bit for 0,
= 0.10100000…
= 1.0100000… x 10^-11
bias of 127 - 1 = 126 which is 01111110
note that the sign is (-), so the first bit is 1. thus,

```
-0.625 = 1 01111110 10100000000000000000000
```

<u>IEEE floating point in hexadecimal:</u>

```
consider 1 01111110 10100000000000000000000 as
1011 1111 0101 0000 0000 0000 0000 0000, which in hexadecimal are
 b    f    5    0    0    0    0    0 , then
-0.625 = 0xbf500000
```

(e) <u>“Strings for 250!\n” in hexadecimal:</u>

```
S -> 83 -> 01010011 -> 53
t -> 116 -> 01110100 -> 74
r -> 114 -> 01110010 -> 72
i -> 105 -> 01101001 -> 69
n -> 110 -> 01101110 -> 6E
g -> 103 -> 01100111 -> 67
s -> 115 -> 01110011 -> 73
[space] -> 32 -> 00100000 -> 20
f -> 102 -> 01100110 -> 66
o -> 111 -> 01101111 -> 6F
r -> 114 -> 01110010 -> 72
[space] -> 32 -> 00100000 -> 20
2 -> 50 -> 00110010 -> 32
5 -> 53 -> 00110101 -> 35
0 -> 48 -> 00110000 -> 30
\n -> 10 -> 00001010 -> 0A
```

thus,
“Strings for 250!\n” = 0x537472696E677320666F72203235300A

(f) 64 bits
If the value is signed, then we assume 2’s complement. Then the largest possible value is 2^63 - 1, which is 9,223,372,036,854,775,807. The bit representation is:
111111111111111111111111111111111111111111111111111111111111111.
For any integer n > 0, 2^63 - 1 + n will cause an integer overflow, and cannot be represented in 64 bits. Therefore, 9,223,372,036,854,775,808 and all greater integer values cannot be represented as a 64 bit signed integer.

Q2

(a)
- (a) a lives on the stack.
- (b) b_ptr lives on the stack.
- (c) *b_ptr lives on the heap because it is an array which is allocated memory via malloc.
- (d) e_ptr lives in the global data because it is declared outside of a function.
- (e) *e_ptr lives on the stack because it is the value stored at &a, which is in the stack.

(b) The value returned by main() is 0. Comments in the code below show work.

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

float* e_ptr;

float foo(float* x, float *y, float* z){
    if (*x > *y + *z) { // if (1.2 > 11.0)
//        printf("wrong track\n");
        return *x;
    }
    else { //since (!(1.2 > 11.0))
//        printf("right track\n");
        return *y+*z; // return 11.0
    }
}

int main() {
    float a = 1.2;
    e_ptr = &a;
    float* b_ptr = (float*)malloc (2*sizeof(float)); // 2 * 4 = 8 bytes allocated
    b_ptr[0] = 7.0;
    b_ptr[1] = 4.0;
//    printf("e_ptr is: %u\t b_ptr is : %u\t b_ptr+1 is :%u\n", e_ptr, b_ptr, b_ptr + 1);
    float c = foo(e_ptr, b_ptr, b_ptr+1); //foo(&a, &7.0, &4.0) ... c takes value of 11.0
    free(b_ptr);
    if (c > 10.5) { //since 11.0 > 10.5
//        printf("right track\n");
        return 0;  //return 0
    }
    else {
//        printf("wrong track\n");
        return 1;
    }
}
```

Q3

```
g++ -O0 -o myProgramUnopt prog.c
time ./myProgramUnopt

    C[111][392]=-1801792042

    real    0m0.892s
    user    0m0.878s
    sys     0m0.004s
```

The output above is from one run. I take the average of ten runs to find a roughly average user time. User time outputs: 0.878, 0.797, 0.870, 0.794, 0.848, 1.051, 0.947, 1.053, 0.82, 0.883. The average user time for the unoptimized version, then, is 0.894.

```
g++ -O3 -o myProgramOpt prog.c
time ./myProgramOpt

C[111][392]=-1801792042

real    0m0.355s
user    0m0.344s
sys     0m0.003s
```

The output above is from one run. I take the average of ten runs to find a roughly average user time. User time outputs: 0.344, 0.387, 0.346, 0.425, 0.483, 0.371, 0.349, 0.357, 0.353, 0.369. The average user time for the optimized version, then, is 0.378.

While the system time does not vary much in the runtimes for the unoptimized vs. optimized compiled versions of prog.c, user time varies significantly. Based on my data, the optimized version is 2.37 times faster than the unoptimized version.