

# Tutorial 7: Float

## Problem 1: Floating Point Representation

Consider the C code in Figure 1.

What are the outputs of the `printf` statements?

```
#include <stdio.h>

int main (void)
{
    static int a = 1;
    static float b = 1;
    int *c=&a;
    printf("Number \"1\" as integer = \"%x\"\n\n", *c);
    printf("Number \"1\" as pointed by an int pointer = \"%x\"\n\n", *(c+1));
    printf("Number \"1\" as float = \"%f\"\n\n", b);
    return 0;
}
```

**Figure 1: Integer Vs Float**

Integer ( $a = 1$ ) is stored as  $(0x00000001)$  in memory. Float ( $b = 1$ ) is stored as IEEE 754 format  $(0x3f800000)$  in memory. Printing the content of the memory locations where variables ( $a$ ) and ( $b$ ) are stored by the first two the `printf` statements will print  $(1)$  and  $(3f800000)$ . The last `printf` statement will print the ( $b$ ) as a decimal floating point value. In this program we rightly assume that ( $b$ ) is located at a memory location immediately after ( $a$ ). The way we access ( $b$ ) by the pointer arithmetic probably should never happen outside the `elec2041` class.

The printouts of the C program in Figure 1 are presented in Figure 2.

```
Number "1" as integer = "1"

Number "1" as pointed by an int pointer = "3f800000"

Number "1" as float = "1.000000"
```

**Figure 2: The Outputs of `printf` Statements**

## Problem 2: Conversion from float to IEEE 754

Consider the C code in Figure 3.

What are the outputs of the `printf` statements?

```

#include <stdio.h>

int main (void)
{
    float a [] = {8.0, 8.5, 8.25, 8.125, 6.0, 6.5, 6.25, 6.125};
    int *c = a, i;
    for (i=0; i < 8; i++)
    {
        printf("Number \"%.3f\" as IEEE 754 format = \"%x\"\\n\\n", a[i],
*(c+i));
    }
    return 0;
}

```

**Figure 3: Float to IEEE 754 Format**

The printouts from the C program in Figure 3 are presented in Figure 4. Program in Figure 3 prints the array (a []) elements as the float format in decimal fraction. It also prints its equivalent in IEEE 754 float format. The way it can print array (a []) in its equivalent form is by forcing an integer type pointer to point to a float type - an operation which is probably illegal outside the elec2041 class.

```

Number "8.000" as IEEE 754 format = "41000000"
Number "8.500" as IEEE 754 format = "41080000"
Number "8.250" as IEEE 754 format = "41040000"
Number "8.125" as IEEE 754 format = "41020000"
Number "6.000" as IEEE 754 format = "40c00000"
Number "6.500" as IEEE 754 format = "40d00000"
Number "6.250" as IEEE 754 format = "40c80000"
Number "6.125" as IEEE 754 format = "40c40000"

```

**Figure 4: The Outputs of printf Statements**

### Problem 3: From IEEE 754 to Scientific Notation and Float

Consider the C code in Figure 5.

What are the outputs of the printf statements?

```

#include <stdio.h>

int main (void)
{
    int a []= {0x3f000000, 0x388205ff, 0xb8324207, 0x3da8f5c3, 0x2cd31b32,
0xd4ae9f7c, 0x56a841ab, 0x5a1dbd91, 0x7fffffff, 0xffffffff, 0xff800000};
    float *c = a, i;
    for (i=0; i < 11; i++)
    {
        printf("IEEE 754 Representation \"%x\" is = \"%3.3e\", and = \"%3.3f\"
\\n\\n", a[i], *(c+i), *(c+i));
    }
    return 0;
}

```

**Figure 5: From IEEE 754 to Scientific Notation and Float**

The printouts from the C program in Figure 5 are presented in . Program in Figure 5 looks at the binary representation of a float number in IEEE 754 format and interprets it as an prints its equivalent in scientific and decimal fraction formats. The way it can print array (a []) in its equivalent form is by forcing a float type pointer to point to an integer type - an operation which is probably illegal outside the elec2041 class.

```

IEEE 754 Representation "3f000000" is = "5.000e-01", and = "0.500000"
IEEE 754 Representation "388205ff" is = "6.200e-05", and = "0.000062"
IEEE 754 Representation "b8324207" is = "-4.250e-05", and = "-0.000042"
IEEE 754 Representation "3da8f5c3" is = "8.250e-02", and = "0.082500"
IEEE 754 Representation "2cd31b32" is = "6.000e-12", and = "0.000000"
IEEE 754 Representation "d4ae9f7c" is = "-6.000e+12", and = "-
6000000106496.000000"
IEEE 754 Representation "56a841ab" is = "9.250e+13", and =
"92499997622272.000000"
IEEE 754 Representation "5a1dbd91" is = "1.110e+16", and =
"11100000452870144.000000"
IEEE 754 Representation "7fffffff" is = "nan", and = "nan"
IEEE 754 Representation "ffffffff " is = "nan", and = "nan"
IEEE 754 Representation " ff800000" is = "-inf", and = "-inf"

```

**Figure 6: The Outputs of printf Statements**

## Problem 4: From IEEE 754 to Binary Scientific Notation

Consider the C code in Figure 7.

What are the outputs of the printf statements?

```

#include <stdio.h>

int main (void)
{
    int a []= {0x3f000000, 0x388205ff, 0xb8324207, 0x3da8f5c3, 0x2cd31b32,
0xd4ae9f7c, 0x56a841ab, 0xbf800000};
    int E, i;
    float M;
    char S;
    for (i=0; i < 8; i++)
    {
        S = (a[i] < 0) ? '-' : '+';
        E = ((a[i]&0x7fffffff) >> 23) - 0x7f;
        M = (a[i]&0x7fffff)/(8388608.0)+1;
        printf("IEEE 754 Representation \"%x\" = \"%c%f X 2exp(%d)\"\\n\\n",
            a[i], S, M, E);
    }
    return 0;
}

```

**Figure 7: From IEEE 754 to Binary Scientific Notation**

The printouts from the C program in Figure 7 are presented in Figure 8. Program in Figure 7 looks at the binary representation of a 32-quantity, interprets it as an IEEE 754 format representation, and breaks it into Sign, Exponent and Mantissa parts.

```

IEEE 754 Representation "3f000000" = "+1.000000 X 2exp(-1)"
IEEE 754 Representation "388205ff" = "+1.015808 X 2exp(-14)"
IEEE 754 Representation "b8324207" = "-1.392640 X 2exp(-15)"
IEEE 754 Representation "3da8f5c3" = "+1.320000 X 2exp(-4)"
IEEE 754 Representation "2cd31b32" = "+1.649267 X 2exp(-38)"
IEEE 754 Representation "d4ae9f7c" = "-1.364242 X 2exp(42)"
IEEE 754 Representation "56a841ab" = "+1.314504 X 2exp(46)"
IEEE 754 Representation "bf800000" = "-1.000000 X 2exp(0)"

```

**Figure 8: The Outputs of printf Statements**

## Problem 5: Float Computation and Accuracy

Consider the C code in Figure 9.

What are the outputs of the printf statements?

```

include <stdio.h>

int main (void)
{
    float a = 9.25e23, b = 1.1e-23, c=1.0e-23, d;

    d = (a + b) - (a + c);
    printf("The Float Number = \"%e\"\n\n",d);
    d = (b - c);
    printf("The Float Number = \"%e\"\n\n",d);
    return 0;
}

```

**Figure 9: Float Computation and Accuracy**

The printouts from the C program in Figure 9 are presented in Figure 10. In the first print statement (a) is so large that it consumes both (b) and (c). Therefore (a + b) and (a + c) are both evaluated as (a) and (d = a - a = 0).

```

The Float Number = "0.000000e+00"

The Float Number = "1.000000e-24"

```

**Figure 10: The Outputs of printf Statements**