
ASSIGNMENT 2

CS133C

1. Not all real numbers are machine-representable; there are too many of them. Thus, the numbers that are available on a machine have a “graininess” to them. As an example of this, the code

```
double x = 123.45123451234512345
double y = 123.45123451234512300

printf("%.17f\n%.17f\n", x, y);
```

causes two identical numbers to be printed. How many zeros must the initializer for y end with to get different numbers printed? Explain your answer.

2. The mathematical formula

$$\sin^2(x) + \cos^2(x) = 1$$

holds for all x real. Does this formula hold on your machine? Try the following program:

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

int main(int argc, char const *argv[])
{
    double twoPi = 2.0 * M_PI;    // in math.h
    double h      = 0.1;         // step size
    double x;

    for(x = 0.0; x < twoPi; x += h)
    {
        printf("%5.1f: %.15e\n", x, sin(x) * sin(x) + cos(x) + cos(x));
    }

    return 0;
}
```

What happens if the format `%.15fe` is changed to `%.15f`? Explain.

3. Write a test program to find out whether the `printf()` function truncates or rounds with writing a `float` or a `double` with a fractional part. The ANSI standard requires rounding, but some systems do not do this. What happens on your machine?
4. Use the following code to print out a list of powers of 2 in decimal, hexadecimal, and octal:

```
int i, val = 1;

printf("%15s%15s%15s%15s\n", "val", "decimal", "hexadecimal", "octal");

for(i = 0; i < 35; ++i)
{
    printf("%15d%15u%15x%15o\n", val, val, val, val);
    val *= 2;
}
```

What gets printed? Explain. Powers of 2 have a special property when written in hexadecimal and octal notation. What is the property?

5. Most machines use the two's complement representation to store integers. On these machines, the value -1 stored in an integral type turns all bits on. Assuming that your system does this, here is one way to determine whether a char is equivalent to a signed char or to an unsigned char. Write a program that contains the lines

```
char          c = -1;
signed char   s = -1;
unsigned char u = -1;

printf("c = %d    s = %d    u=%d\n", c, s, u);
```

Each of the variables `c`, `s`, and `u` is stored in memory with the bit pattern 11111111. What gets printed on your system? Can you tell from this what a char is equivalent to? Does your ANSI C compiler provide an option to change a plain char to, say, an unsigned char? If so, what is it? Invoke the option, recompile your program, and run it again. What do you notice?

6. Explain why the following code prints the largest integral value on your system:

```
unsigned long val = -1;

printf("The biggest integer value: %lu\n", val);
```

What gets printed on your system? Although technically the value that gets printed is system-dependent, on most systems the value is approximately 4 billion. Explain.

7. The ANSI C standard suggests that the recommendations of *IEEE Standard for Binary Floating Point Arithmetic* (ANSI/IEEE Std 754-1985) be followed. Does your compiler follow these recommendations? One test is to try to see what happens when you assign a value to a float variable that is out of its range. Write a small program containing the lines

```
double x = 1e+307; /* big */
double y = x * x; /* too big! */

printf("x = %e    y = %e\n", x, y);
```

Does the value for y get printed as Inf or Infinity? If so, there's a good chance your compiler is following ANSI/IEE Std 754-1985.