Max Simon
3295271

Exercise Sheet 01

Exercise 1

I get the following ϵ_m:

$$\begin{aligned}&\text{float} \quad \epsilon_m = 1.192 \cdot 10^{-7}\\ &\text{double} \quad \epsilon_m = 2.220 \cdot 10^{-16}\\ &\text{long double} \quad \epsilon_m = 1.084 \cdot 10^{-19}\end{aligned}$$

For the evaluation of $1 + \epsilon_m$ I get the results:

$$\begin{aligned}\text{float} \quad 1 + \epsilon_m &= 1.00000011920928955078125\\ \text{double} \quad 1 + \epsilon_m &= 1.00000000000000022204460492503\\ \text{long double} \quad 1 + \epsilon_m &= 1.00000000000000000010842021725\end{aligned}$$

but I didn't get, what is strange with these results.

Exercise 2

The results for x and y:

$$\begin{aligned}x &= 1\\ y &= 0\end{aligned}$$

The problem is the finite precession of *double*.
If I add a and b, I get of course 0. To this c is added, so I get 1.
If I add b and c, the result is still $-1 \cdot 10^{-17}$, because the 1 which is added is out of the precession of the double. If one adds a afterwards, the result is of course 0.
So x is correct.

Exercise 3

The output of this programm is:

$$\begin{aligned}i &= 100\\ j &= 99\\ k &= 100\end{aligned}$$

The result of i and k is of course clear. $0.01 * 10000 = 100$, which can be converted into an integer of value 100.

The problem with j is the typecast of the value of x to a double, which gain precession. As you can see below, the difference between the represented and the real value is 10^{-11}, which is out of floats precession. But converting it to a double, this difference is inside the precession, so we get a difference/not exactly 0.01.
The float is encoded by sign$\cdot 2^{\text{exponent}} \cdot$mantissa. Our mantissa is given by $(10100011110101110000 1010)_2$ (notice the implicit leading bit) which can be transported into decimal system by

$$\text{mantissa} = \sum_{i=0}^{\#\text{ bits}} v_i \frac{1}{2^i}, \quad v_i = \text{value of bit i, 0 or 1} \tag{1}$$

In our case, this leads to

$$\text{mantissa} = \frac{5368709}{4194304} \tag{2}$$

The exponent is decoded as -7 (after subtracting 127), which leads to an additional factor of $\frac{1}{2^7} = \frac{1}{128}$. So all in all our number 0.01 is represented as

$$\frac{5368709}{4194304} \cdot \frac{1}{128} = \frac{5368709}{536870912} \tag{3}$$

So $n = 5368709$ and $m = 536870912$. The difference to 0.01 is $3.76 * 10^{-11}$.

Exercise 4

IEEE-754 numbers are normalized.
So between 1.0 and 2.0 the exponent is 0, which means, that we can use the whole mantissa of 23 bits (52 bits for double). This yields in about $8.4 * 10^6$ possibilities ($4.5 * 10^{15}$ for double). For the range between 255.0 and 256.0 we have to use an exponent of 7 ($2^7 = 128$, exponent of 8 is already to big). So our minimal value for the mantissa is $\frac{255}{128} = 1.99219$, which - of course - determine our first 7 bits (without implicit leading bit). So there are only 16 bits remaining (45 for double), which give us about 65000 possibilities ($3.5 * 10^{13}$ for double).
So in the range between 255 and 256 one can represent about 2 magnitudes less numbers as in the range between 1.0 and 2.0.

Exercise 5

I get the following results for the resulting sum:

- Forward, without ordering, double: $-6.516e + 15$
- Backwards, without ordering, double: $-6.596e + 15$
- Forward, with ordering, double: 42
- Backward with ordering, double: 0

- Forward, with ordering, long double: 42
- Backward, with ordering, long double: 0

We see, that we get very different results for ordering/not ordering and forward or backward modes. But it seems that the usage of long double does not make a difference. The results, where we did not order the numbers are wrong. Here it depends on the position of the number, if it is take into account or not. (Many small numbers can sum up to a larger one, which could take into account; many small numbers interrupted by very great ones are neglected).
And because of the result (42) I think the forward summation is the more correct one :D.