1.) Explain why an int that is stored in 8 bytes cannot always be exactly stored in an 8 byte double? Meaning, why in some cases would the value stored in the double be an approximation of the original int value?

* A double only has 53 bits to represent significant digits in the fractional field versus an 8 byte int has 63 bits (excluding the sign bit) to encode significant digits. So, not every number that can be stored in 63 bits of an int could be stored precisely/exactly in the 53 bits in the fractional field of a double.

2.) Assume the variables d and c are both doubles. Explain how the following equation can evaluate to false. 

\[( d + c ) - d == ( d - d ) + c\]

This would be problematic if d was a very large number and c was a very small number. When d is added to c, c would basically be lost due to the limited number of significant digits that can be represented in a double. Thus, (d+c)-d would look like 0 but (d-d)+c would look like c.

3.) Explain the relationship between the x86_64 registers %rax and %eax.

\%rax is a 64 bit register. \%eax is a 32 bit register which is the 32 lower bits of register \%rax.

4.) Assume that an array variable of type long has been declared and called array. Assume that a long is stored in 8 bytes. Write C code using the array operator (i.e. \[\]) to indicate which element of array is being accessed by each of the 3 x86_64 memory accesses below.

For this problem, assume the starting address of array is 1000 and that the following registers store the specified values:

- %rdi = 1000
- %rsi = 4
- %rax = 16

For example, (%rdi) is an access to array[0].

The memory accesses are:

- 24 ( %rdi ) \hspace{1cm} array[3]
- ( %rdi, %rsi, 8 ) \hspace{1cm} array[4]
- 48 ( %rdi, %rax ) \hspace{1cm} array[8]

Make sure your answer is clear about which memory access each piece of C code corresponds to.