Contents

Over and Underflow

Overflow

- Assume a car has a 4-digit odometer, and currently, it shows `9999`. What does the odometer show if you drive the car another mile? As you might guess, it shows `0000` while it should show `10000`. The reason is the odometer does not have a counter for the fifth digit. Similarly, in C#, when you do arithmetic operations on integral data, the result may not fit in the corresponding data type. This situation is called an **overflow** error.
- In an unsigned data type variable with N bits, we can store the numbers from 0 to $2^N - 1$. In signed data type variables, the high order bit represents the sign of the number as follows:
- 0 means zero or a positive value
- 1 means a negative value
- With the remaining $N - 1$ bits, we can represent $2^(N - 1)$ values. Hence, considering the sign bit, we can store a number from $-2^(N - 1)$ to $2^(N - 1) - 1$ in the variable.
- In some programming languages like C and C++, overflow errors cause undefined behavior, and can crash your program. In C#, however, the extra bits are just ignored, and the program will continue executing even though the value in the variable may not make sense. If the programmer is not careful to check for the possibility of overflow errors, they can lead to unwanted program behavior and even severe security problems.
- For example, assume a company gives loans to its employee. Couples working for the company can get loans separately, but the total amount cannot exceed $10,000. The following program looks like it checks loan requests to ensure they are below the limit, but it can be attacked using an overflow error. (This program uses notions you may have not studied yet, but that should not prevent you from reading the source code and executing it.)

```
!include code/snippets/overflowExample.cs
```

- If the user enters 2 and 4,294,967,295, we expect to see the error message ("Error: the sum of loans exceeds the maximum allowance."). However, this is not what will happen, and the request will be accepted even though it should not have. The reason can be explained as follows:
- `uint` is a 32-bit data type.
- The binary representation of 2 and 4,294,967,295 are `00000000000000000000000000000010` and `11111111111111111111111111111111`.
- Therefore, the sum of these numbers should be `100000000000000000000000000000001`, which needs 33 bits.
- Nevertheless, there are only 32 bits available for the result, and the extra bits will be dropped, so the result will be `00000000000000000000000000000001`. This is less than 10,000, so the program will conclude that the sum of the loan values is less than 10,000.

Underflow

- Sometimes, the result of arithmetic operations over floating-point numbers is smaller than the minimum value that can be stored in the corresponding data type. This problem is known as the **underflow** problem.
- In C#, in case of an underflow, the result will be zero.
- For example, the smallest value that can be stored in a `float` variable is $1.5 \cdot 10^{-45}$. If we attempt to divide this value by 10, the variable will get the value 0, not $1.5 \cdot 10^{-46}$:

```
!include code/snippets/underflowExample.cs
```

- An underflow error can result in "losing" data in the middle of a series of operations: even if you immediately multiply by 10 again, the intermediate result was less than $1.5 \cdot 10^{-45}$, so the final result is still 0.