Datatypes and Variables: More Details

Principles of Computer Programming I

Spring/Fall 20XX

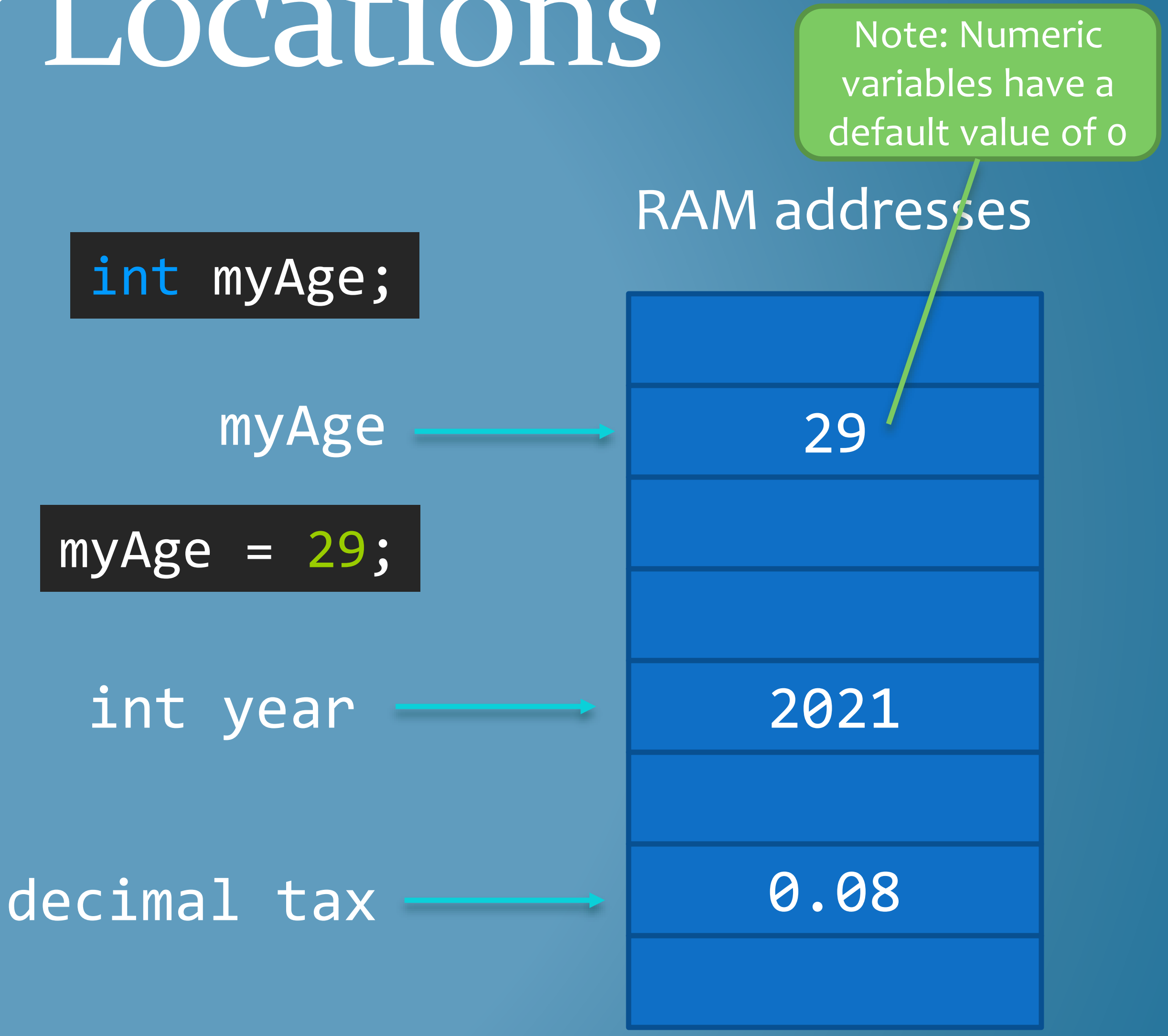
AUGUSTA UNIVERSITY

Outline

- More About C# Datatypes
    - Integer types and sizes
    - Floating-point type precision
    - Value vs. Reference Types
- Overflow and Underflow

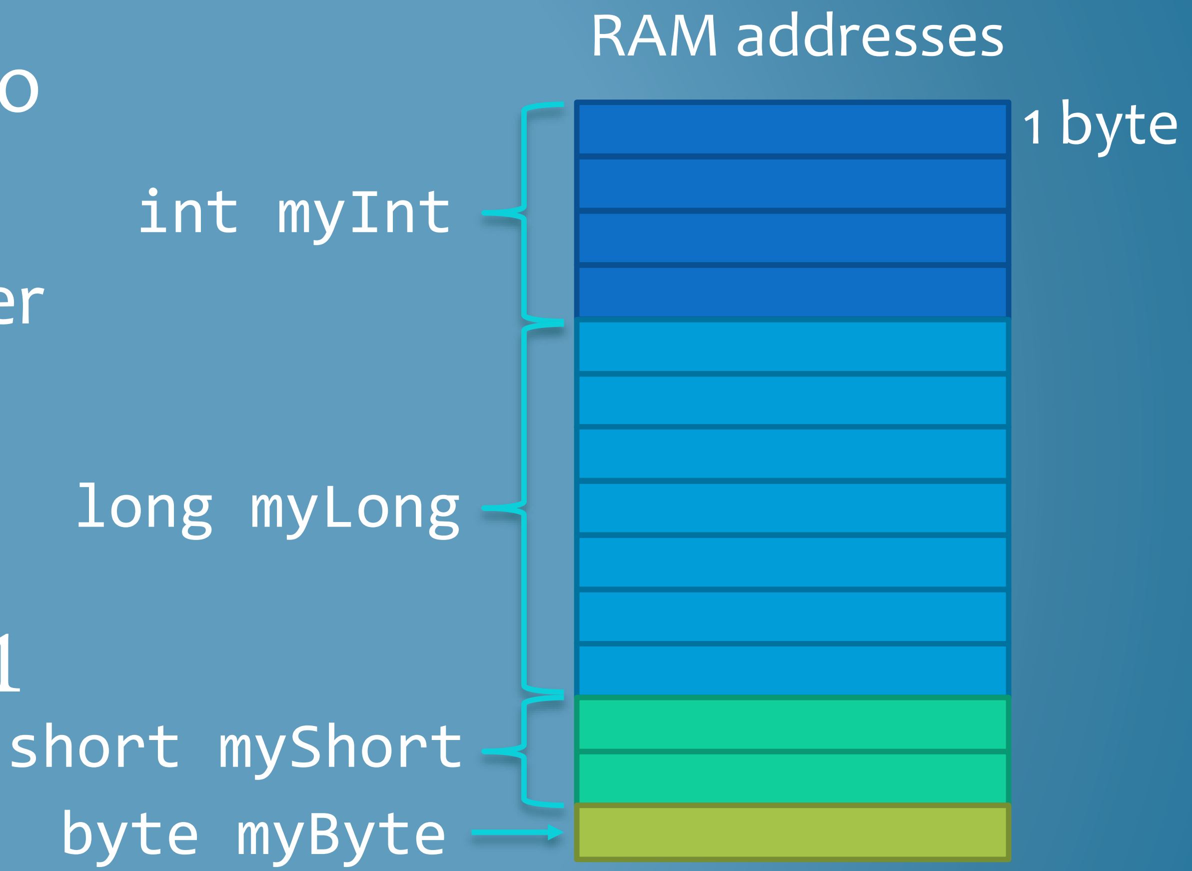
Variables: Memory Locations

- A variable names a *memory location* in which to store data
- Declaring a variable = reserving memory, giving it a name
- Assigning a variable = storing data to that address in memory



Integer Types by Size

- How much memory do you use to store each variable?
  - Depends on the size of the number
- `int` = 4 bytes = $-2^{31} \ldots 2^{31} - 1$
- `long` = 8 bytes = $-2^{63} \ldots 2^{63} - 1$
- `short` = 2 bytes = $-2^{15} \ldots 2^{15} - 1$
- `byte` = 1 byte = $0 \ldots 255$
  - byte is *unsigned* by default



AUGUSTA UNIVERSITY

Signed vs. Unsigned

- Unsigned versions of types can store larger (positive) values, in the same number of bytes
  - `ushort` = 2 bytes = $0 \ldots 2^{16} - 1$
  - `uint` = 4 bytes = $0 \ldots 2^{32} - 1$
  - `ulong` = 8 bytes = $0 \ldots 2^{64} - 1$

Power of 2 greater than `int`

- Why? Storing the plus/minus sign takes one bit (binary digit), so the number itself must be 1 bit smaller

Integer Format Details

- Computers store numbers in *binary*, i.e. base-2
  - `uint` = 4 bytes = 32 bits (digits)

2^4 2^3 2^2 2^1 2^0

`10101` = 21

The 2^{31} digit

Byte 4	Byte 3	Byte 2	Byte 1
`11101110`	`01101011`	`00101000`	`00000000`

The 2^0 digit

In decimal: 4,000,000,000

- Sign bit replaces a digit, so `int` can only store a 31-bit number

The 2^{30} digit

`0`	`1101110`	`01101011`	`00101000`	`00000000`

The 2^0 digit

In decimal: 1,852,516,352

Sign bit: 0 for +, 1 for –

Outline

- More About C# Datatypes
  - Integer types and sizes
  - **Floating-point type precision**
  - Value vs. Reference Types
- Overflow and Underflow

Floating-Point Types by Size

- How many significant figures do you care about?

Type	Size	Range of Values	Digits of Precision
`float`	4 bytes	$\pm 1.5 \cdot 10^{-45} \ldots \pm 3.4 \cdot 10^{38}$	7
`double`	8 bytes	$\pm 5.0 \cdot 10^{-324} \ldots \pm 1.7 \cdot 10^{308}$	15-16
`decimal`	16 bytes	$\pm 1.0 \cdot 10^{-28} \ldots \pm 7.9 \cdot 10^{28}$	28-29

- `float` and `double`: *approximations* of decimal values
- `decimal`: *exact* decimal values, as long as they fit in range

Approximate Decimals

- `float` and `double` use **binary** (base 2) fractions:

2^2 2^1 2^0 2^{-1} 2^{-2}

`101.01`

$= 4+1+{}^{1}\!/_{4} = 5.25$

- Why "floating point"? Actually it's binary scientific notation:

Mantissa → `10101e-10` ← Exponent

Shift "." left by 2

`101.01`

Or read it as a decimal

$$21 \cdot 2^{-2} = \frac{21}{4} = 5.25$$

Approximate Decimals

- Some decimal (base 10) fractions can't be expressed in binary
- Decimal 0.1 = $\frac{1}{10}$. Can you express $\frac{1}{10}$ as a sum of $\frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{32}, \ldots$?

$$\underbrace{.0001}_{\frac{1}{16}}\underbrace{1}_{\frac{1}{32}}00\underbrace{1}_{\frac{1}{256}}\underbrace{1}_{\frac{1}{512}}\ldots \ (\ldots 00110011\ldots)$$

=.099609375

- Truncating the infinite sequence leads to errors, just like with $\frac{1}{3}$:

$\frac{1}{3} + \frac{2}{3}$ in decimal = $.33333333 + .66666666 = .99999999 \neq 1$

Exact Decimals

- The `decimal` type uses decimal (base 10) fractions
- Finite decimals stay finite, so no unexpected errors
- But computers can only use binary, how does that work?

Decimal 5.1:

Mantissa → `110011 e`$_{10}$`-1` ← Exponent, but as a power of 10

51 10^{-1}

$$51 \cdot 10^{-1} = \frac{51}{10} = 5.1$$

Choosing a Floating-Point Type

- More precision sounds good, why not always use decimal?

Type	Size	Range of Values	Digits of Precision
`float`	4 bytes	$\pm 1.5 \cdot 10^{-45} \ldots \pm 3.4 \cdot 10^{38}$	7
`double`	8 bytes	$\pm 5.0 \cdot 10^{-324} \ldots \pm 1.7 \cdot 10^{308}$	15-16
`decimal`	16 bytes	$\pm 1.0 \cdot 10^{-28} \ldots \pm 7.9 \cdot 10^{28}$	28-29

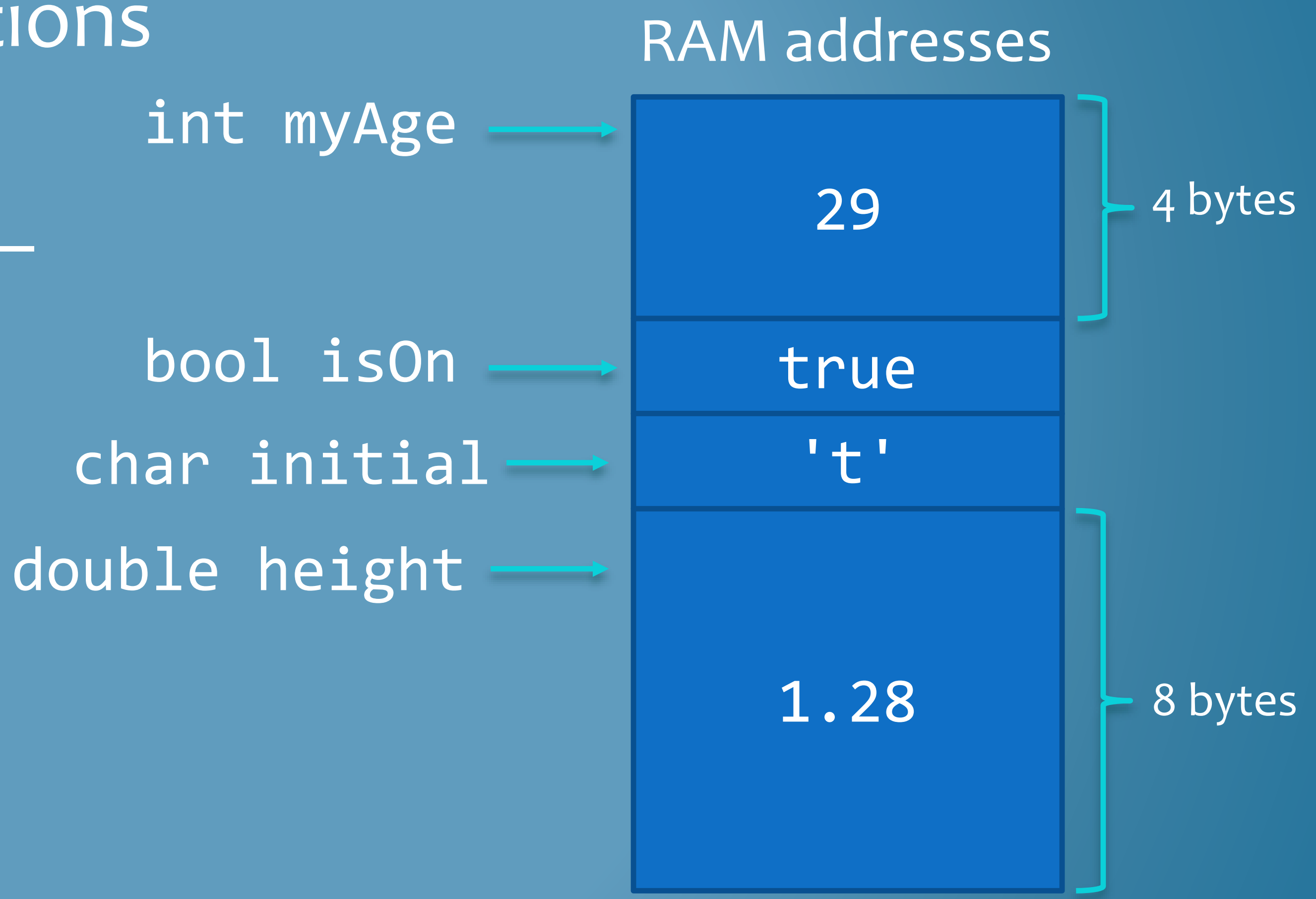
- **Much** slower to compute with than `double`
  - Base-10 exponent is hard, base-2 exponent is easy
- Only use if you really need exact values of .01, i.e. money
  - Physical measurements are not that exact

Outline

- More About C# Datatypes
  - Integer types and sizes
  - Floating-point type precision
  - **Value vs. Reference Types**
- Overflow and Underflow

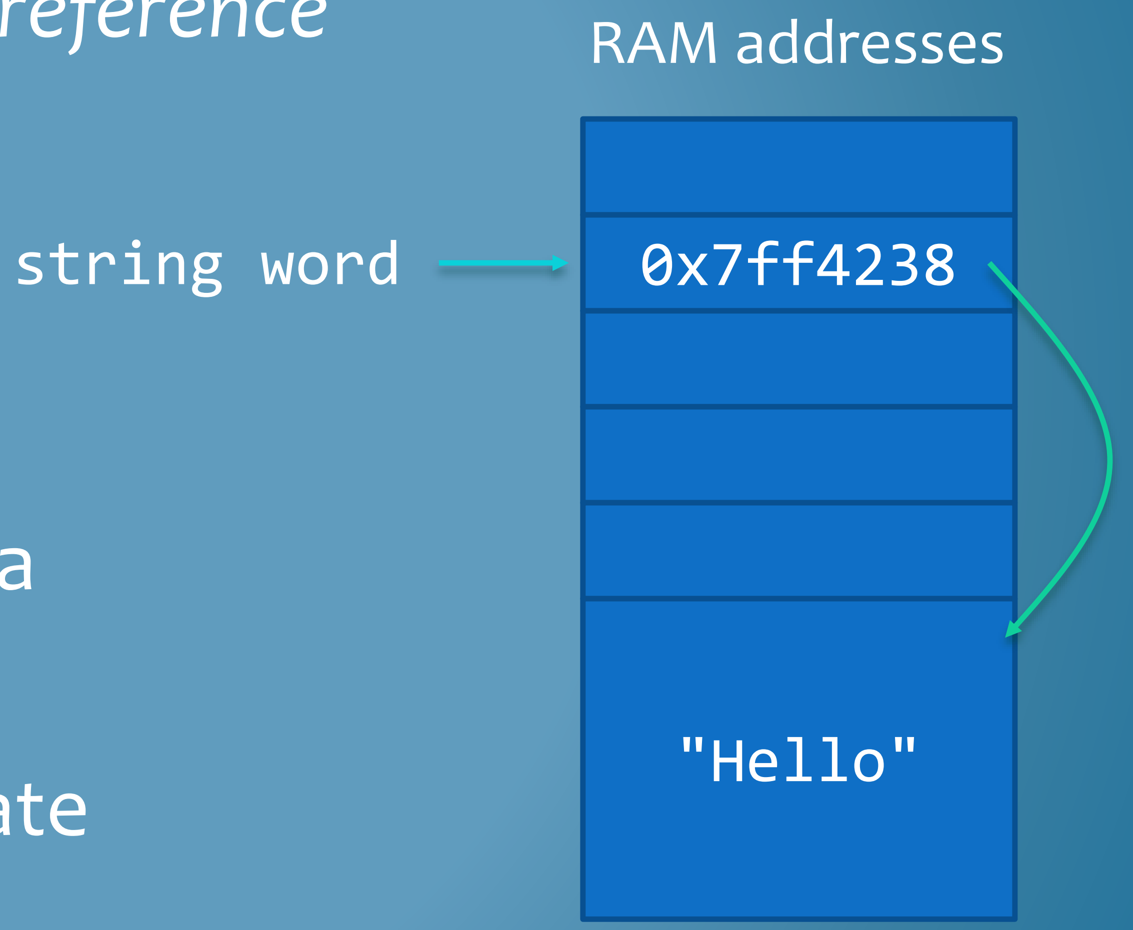
Value Types

- Variables name memory locations
- **Value Type:** Named memory location just stores the value – what you'd expect
- Numeric types (`int`, `uint`, `double`, `decimal`, etc.)
- `char`
- `bool` – logical true or false



Reference Types

- Named memory location stores a *reference* to the data, not the data itself
- Contents of the named memory location = the address of another memory location
- This location stores the actual data
- `string`
- `object` – and all objects you create

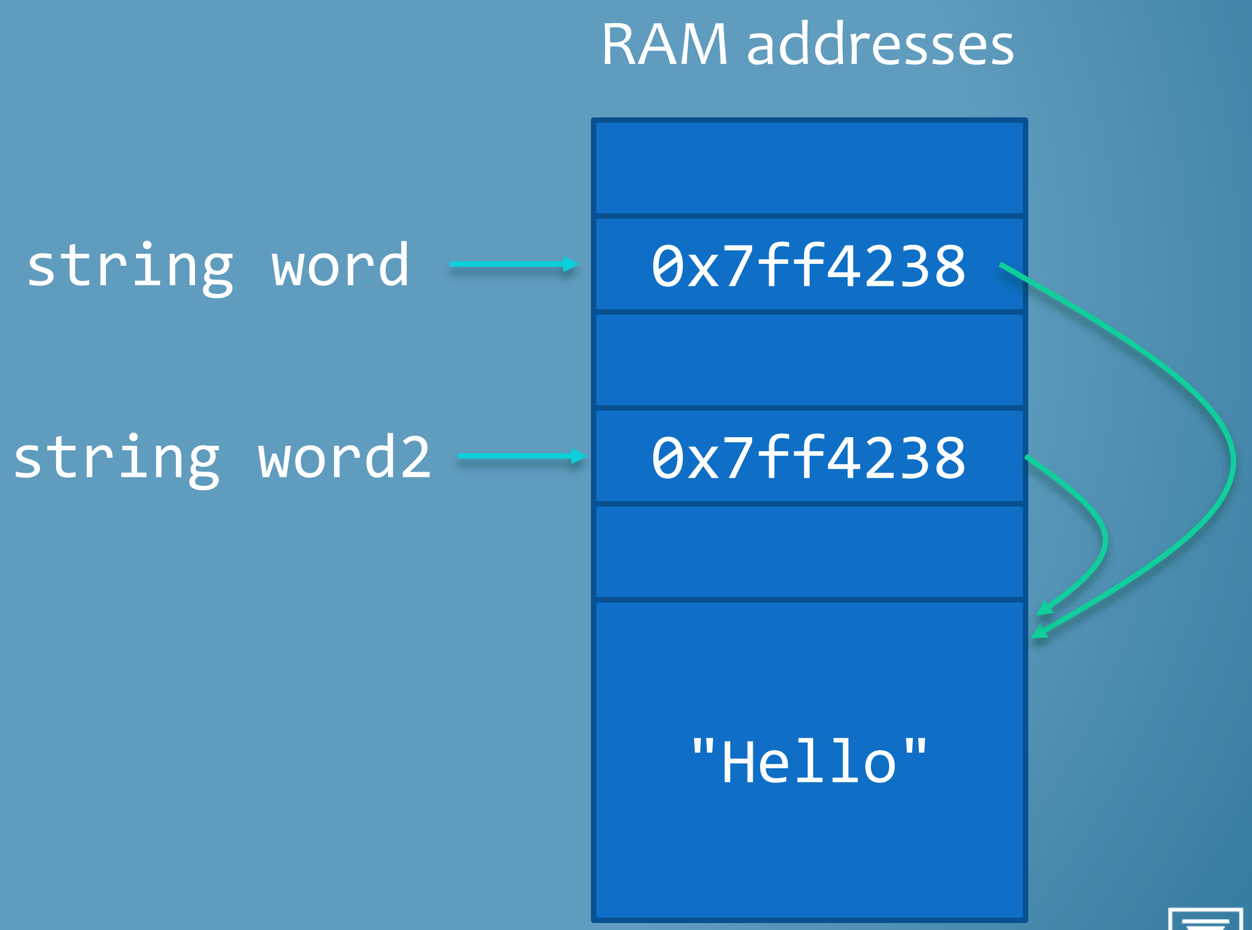


Implications of Reference Types

- Assignment copies the value, but **not** the data it refers to

```
string word = "Hello";
```

```
string word2 = word;
```



Outline

- More About C# Datatypes
  - Integer types and sizes
  - Floating-point type precision
  - Value vs. Reference Types
- **Overflow and Underflow**

The Overflow Problem

- What happens when the odometer reads 999999 and you drive 1 more mile?

- Same problem with C# integers: Fixed number of digits (bits), once they are all 1, adding 1 "rolls over"

`uint` with value 4,294,967,295:

11111111	11111111	11111111	11111111

+ 1

00000000	00000000	00000000	00000000

The Overflow Problem

- Even worse with signed integers: Adding 2 positive numbers results in a negative number!

`int` with value 2,147,483,647 → `0 1111111 11111111 11111111 11111111`

Sign bit

`+ 1`

`int` with value -2,147,483,648 → `1 0000000 00000000 00000000 00000000`

- In C#, this **does not** make your program crash

Overflow Bug Example

```
Console.WriteLine("Enter requested loan amount for first person");
uint loan1 = uint.Parse(Console.ReadLine());
Console.WriteLine("Enter requested loan amount for second person");
uint loan2 = uint.Parse(Console.ReadLine());
if(loan1 + loan2 < 10000)
{
  Console.WriteLine("Your loans are approved!");
}
else
{
  Console.WriteLine("Error: The sum of the loans exceeds "
    + " the maximum of $10,000");
}
```

4000000000 + 294967300 = 4

What if `loan1` is 4,000,000,000 and `loan2` is 294,967,300?

The Underflow Problem

- Floating-point types have range limits (limited size of exponent)
- Any number smaller than minimum value will be rounded to 0

```
float myNumber = 1E-45f;
myNumber = myNumber / 10;
Console.WriteLine(myNumber);
```

```
0
```

- This applies even to intermediate values in a calculation:

```
float myNumber = 1E-45f;
myNumber = (myNumber / 10) * 10;
Console.WriteLine(myNumber);
```

```
0
```