Contents

Overflow and Underflow

This lab serves multiple goals:

- To introduce you to the concept of *overflow* and *underflow*,
- To give an example of the `MaxValue` and `MinValue` constants,
- To exemplify the care required when performing mathematical calculations with programs,
- (Optional) To illustrate the `String.Format` method.

Overflow

Warm Up

For a general introduction of overflow, please read the relevant section[1]. Do execute the code shared in this section:

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

Make sure you have a general understanding of what overflowing means before proceeding. If you are unsure, reading about integer overflow on wikipedia[2] can help.

We will now enter a surprising world where

- a number can be equal to itself plus one,
- a number plus one can be *less* than the number itself, and
- a number multiplied by two and then divided by two is the same as the number multiplied by two.

Now, download[3], execute the "Overflow" solution, and answer the following questions:

- What is the maximum value that can be stored in an `int`?

[1] https://csci-1301.github.io/book.html#overflow
[2] https://www.wikiwand.com/en/Integer_overflow
[3] ./code/projects/Overflow.zip

- What *should* be the result of adding one to the maximum value that can be stored in an `int`?
- What is the result actually displayed by C#?

Then, answer the same questions for the `float` and `double` datatypes.

C#'s Checks

We will now study some of the safeguards against overflowing that are implemented in C#. **Note that some of those checks have been deactivated by the `unchecked` command at the beginning of our project. Make sure to complete the following section *outside* of `unchecked`'s scope (i.e., after it).**

1. Enter the following statements in your program:

   ```
   int int_max_value_plus_one_bis = int.MaxValue + 1;
   ```

   You should receive an error message. Reads it and try to understand what C## is warning you against.

2. Isn't that weird that C## let go

   ```
   int int_max_value_ter = int.MaxValue;
   int int_max_value_plus_one_ter = int_max_value_ter + 1;
   ```

   (even outside an `unchecked` environment, you can try it) not the previous statement? This is because, by default, arithmetic operations on `int` are "unchecked[4]". To check it, use

   ```
   int int_max_value = int.MaxValue;
   int int_max_value_plus_one = checked(int_max_value + 1); // Note the "checked(…
   ```

   What happens when you try to compile and run this program?

3. Note that our program does not give the result of adding one to the maximum value that can get assigned to a `decimal`. Try to display on the screen the result of adding one to `decimal.MaxValue` (both inside and outside the `unchecked` environment).

Strange Mathematical Properties

Circling back to our prompt in the warm-up section, enter in the following table the value(s) and datatype(s) for `x`, `y`, and `z`:

Description	Value(s)	Datatype(s)
x is equal to x+1		

[4] https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/checked-and-unchecked

Description	Value(s)	Datatype(s)
y +1 is less than y		
z * 2 / 2 is equal to z times 2		

Note that `int.MinValue` can similarly be used to produce strange mathematical properties of the same kind.

As funny or interesting as that strange behavior may seem, overflow errors actually caused death and millions of dollars of losses repeatedly, as you can read for instance in this blog post[5].

Underflow

Most of what we wrote about overflow is also true of *under* flow, and you can read about it in the lecture notes[6]. In a nutshell, you can witness it by executing a statement such as

```
Console.WriteLine(0.00000000000000001f * 0.00000000000000001f);
```

Which should display `1e-34` but actually displays `9.999999E-35`. As you can see, a rounding error took place because C## did not have enough "room" to store all the information. Another interesting example is given by the following loop:

```
float x = int.MaxValue;
while (x > 0)
{
    Console.Write(x + " ");
    x = x / 2;
}
Console.WriteLine(x);
```

Note that this loop should *never* exit: no matter how large `x` is, dividing it by two repeatedly should never make it 0! After how many iterations would this loop terminate?

(Optional) String Formatting

As you may have noticed, our program uses the `String.Format` method to nicely display the information. You can read about this method in the official documentation[7].

[5] https://medium.com/@jollyfish/integer-overflow-underflow-and-floating-point-imprecision-6ba869a99033#73a3

[6] https://csci-1301.github.io/book.html#underflow

[7] https://learn.microsoft.com/en-us/dotnet/api/system.string.format?view=net-7.0#the-format-method-in-brief

Compile and execute the following code:

```
string value = String.Format("|{0,-20:C}|{0,20:C}|{0,10:C}|{0,20:P}|{1}", 126347.
Console.WriteLine(value);
```

Explain the role of:

- the first digit (either `0` or `1`) between the braces,
- the second number (`-20`, `20`, `10`, …) between the braces,
- the last character (`C`, `P`) between the braces.

Note that the previous statement could have been replaced by

```
Console.WriteLine("|{0,-20:C}|{0,20:C}|{0,10:C}|{0,20:P}|{1}", 126347.89m, "test"
```

as `Console.WriteLine` can process "composite format strings".