CHAPTER 12

Understanding Different Kinds of Numbers

In this chapter, you will study several more advanced topics concerning numbers and calculations, such as more numeric types, memory consumption, and overflow. If you do not need this much detail at this time, you can safely skip this chapter or just skim it.

More Numeric Types

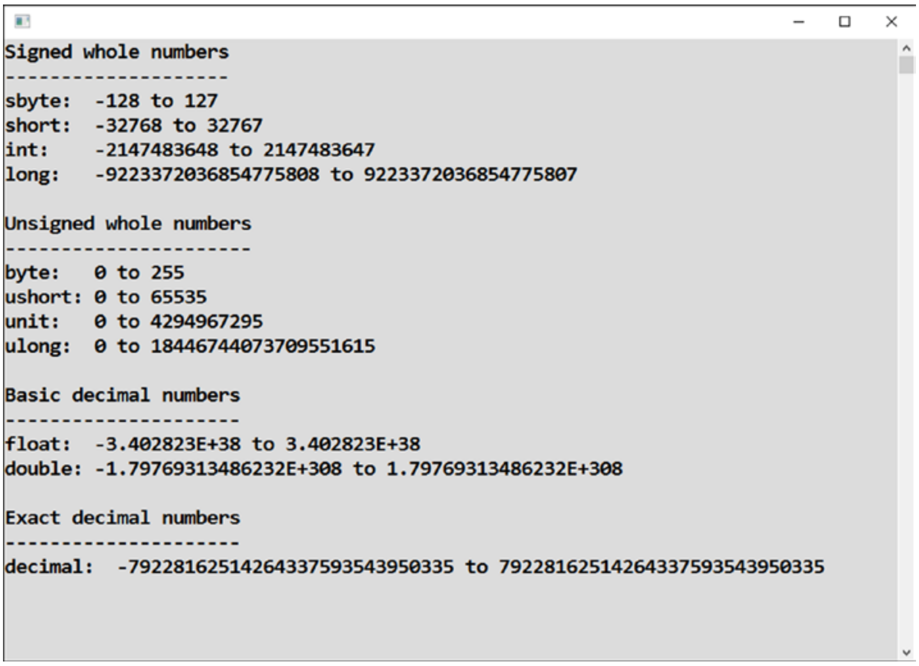
You already know that there is a distinction between whole numbers and decimal numbers in computing. You use the `int` type for whole numbers, and you use the `double` type for decimal numbers.

But there are other numeric data types in C#. Although many of them exist mainly for historical reasons and you will probably never use them, it is good to know about them at least.

© Radek Vystavěl 2017
R. Vystavěl, *C# Programming for Absolute Beginners*, https://doi.org/10.1007/978-1-4842-3318-4_12

Task

You will write a program that displays an overview of all the C# numeric data types. For each type, its range of possible values will be printed (see Figure 12-1).



```
Signed whole numbers
--------------------
sbyte:  -128 to 127
short:  -32768 to 32767
int:    -2147483648 to 2147483647
long:   -9223372036854775808 to 9223372036854775807

Unsigned whole numbers
----------------------
byte:   0 to 255
ushort: 0 to 65535
unit:   0 to 4294967295
ulong:  0 to 18446744073709551615

Basic decimal numbers
---------------------
float:  -3.402823E+38 to 3.402823E+38
double: -1.79769313486232E+308 to 1.79769313486232E+308

Exact decimal numbers
---------------------
decimal:  -79228162514264337593543950335 to 79228162514264337593543950335
```

Figure 12-1. *Printing all numeric data types*

Solution

Here is the code:

```
static void Main(string[] args)
{
    // Immediately outputs
    Console.WriteLine("Signed whole numbers");
    Console.WriteLine("--------------------");
    Console.WriteLine("sbyte:  " + sbyte.MinValue + " to " +
    sbyte.MaxValue);
    Console.WriteLine("short:  " + short.MinValue + " to " +
    short.MaxValue);
    Console.WriteLine("int:    " + int.MinValue + " to " + int.MaxValue);
    Console.WriteLine("long:   " + long.MinValue + " to " + long.MaxValue);
    Console.WriteLine();

    Console.WriteLine("Unsigned whole numbers");
    Console.WriteLine("----------------------");
    Console.WriteLine("byte:   " + byte.MinValue + " to " + byte.MaxValue);
    Console.WriteLine("ushort: " + ushort.MinValue + " to " + ushort.
    MaxValue);
    Console.WriteLine("unit:   " + uint.MinValue + " to " + uint.MaxValue);
    Console.WriteLine("ulong:  " + ulong.MinValue + " to " + ulong.
    MaxValue);
    Console.WriteLine();

    Console.WriteLine("Basic decimal numbers");
    Console.WriteLine("---------------------");
    Console.WriteLine("float:  " + float.MinValue + " to " + float.MaxValue);
    Console.WriteLine("double: " + double.MinValue + " to " + double.
    MaxValue);
    Console.WriteLine();

    Console.WriteLine("Exact decimal numbers");
    Console.WriteLine("---------------------");
    Console.WriteLine("decimal:  " + decimal.MinValue + " to " + decimal.
    MaxValue);

    // Waiting for Enter
    Console.ReadLine();
}
```

Note

To display the ranges, I have used the `MinValue` and `MaxValue` properties of all the numeric data types.

Discussion

The following sections discuss this program.

Unsigned Numbers

The results printed by the program show that some data types do not allow the storage of negative numbers! However, these *unsigned numbers* are rarely used, with the exception of the `byte` type, which you use when reading binary data from a file, a database, or a web service.

Contrary to their signed counterparts, unsigned numbers usually begin with a *u*, meaning "unsigned." Similarly, the signed type `sbyte` starts with an *s*, meaning the "signed" variant of the much more important `byte`.

Decimal Numbers

Decimal type ranges are displayed in scientific notation (also called *exponential notation*). For example, the greatest `float` number is displayed as 3.4E+38, which means 3.4 times 10 to the 38th power. This is a really big number, isn't it?

Decimal types differ also in their precision. While the `float` type stores a decimal value with approximately 7 significant digits, the `double` type offers a precision of about 15 significant digits, and the `decimal` type offers 28 digits.

Special Type decimal

The `decimal` data type is somewhat special. Because of the following reasons, it is preferably used when working with currency:

- It stores cent values exactly. For example, the amount of 12.80 will be stored precisely as 12.80 rather than something like 12.7999999999, which might happen using other types.
- Because of a large number of significant digits, the `decimal` data type allows you to represent large amounts of money and still keep the cent precision.

However, both of these reasons are not as convincing as they might seem. If you perform rounding correctly, you can store cents exactly with the `double` type. And frankly speaking, you usually need to solve other problems than that of whether `double` 15 digits are enough for money!

Moreover, many things are easier with the `double` type, which is why I use preferably `double` for decimals in this book.

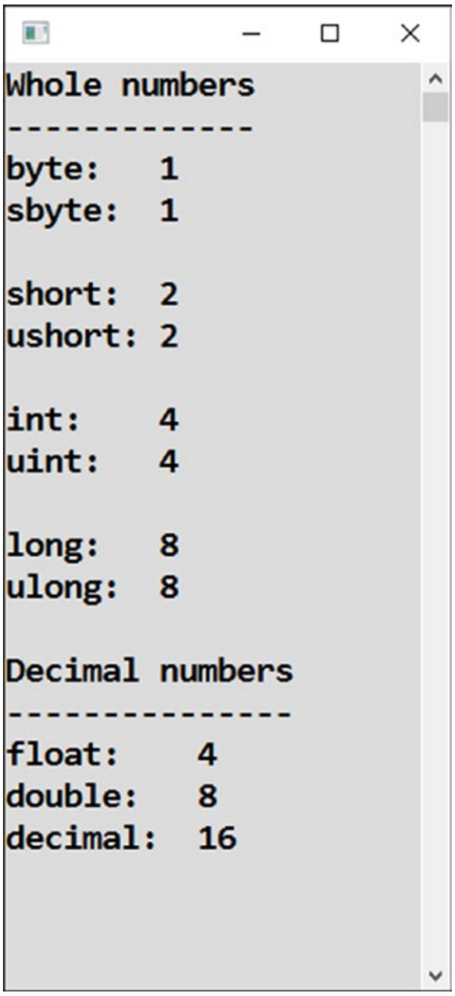
One last note: calculations with the `decimal` type are *much* slower (in fact, hundreds of times slower) than the same calculations with the `double` type. This does not matter if you crunch just a few numbers, but the difference can be significant in large data sets.

Memory Consumption

If you know something about bits and bytes, it may have occurred to you that the type ranges differ because of the memory space that is available to the corresponding types. This is exactly right, and you will learn more about it in this section.

Task

In this section, you will write a program that tells you how many bytes of memory each type uses (see Figure 12-2).



```
Whole numbers
-------------
byte:   1
sbyte:  1

short:  2
ushort: 2

int:    4
uint:   4

long:   8
ulong:  8

Decimal numbers
---------------
float:    4
double:   8
decimal:  16
```

Figure 12-2. *Displaying the number of bytes each type uses*

Solution

Here is the code:

```
static void Main(string[] args)
{
    // Outputs
    Console.WriteLine("Whole numbers");
    Console.WriteLine("-------------");
    Console.WriteLine("byte:   " + sizeof(byte));
    Console.WriteLine("sbyte:  " + sizeof(sbyte));
    Console.WriteLine();
    Console.WriteLine("short:  " + sizeof(short));
    Console.WriteLine("ushort: " + sizeof(ushort));
    Console.WriteLine();
    Console.WriteLine("int:    " + sizeof(int));
    Console.WriteLine("uint:   " + sizeof(uint));
    Console.WriteLine();
    Console.WriteLine("long:   " + sizeof(long));
    Console.WriteLine("ulong:  " + sizeof(ulong));
    Console.WriteLine();
    Console.WriteLine("Decimal numbers");
    Console.WriteLine("---------------");
    Console.WriteLine("float:    " + sizeof(float));
    Console.WriteLine("double:   " + sizeof(double));
    Console.WriteLine("decimal:  " + sizeof(decimal));
    Console.WriteLine();

    // Waiting for Enter
    Console.ReadLine();
}
```

Connections

It is possible to connect the results of the current and previous programs. For example, let's discuss the important `int` type. It uses 4 bytes, or 32 bits of memory. This means 2 to the 32nd power of possible values, which is more than 4 billion. `int` is a signed type, so you have 2 billion for positive numbers and 2 billion for negative numbers. Its unsigned counterpart `uint` has all 4 billion values for positive numbers (and, of course, zero).

Discussion

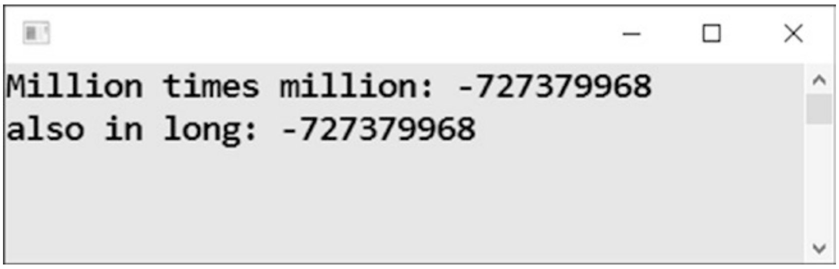
You may feel confused about the variety of numeric data types. To help you understand them, here is a summary of when you should use each one:

- `int`: For regular work with values that are intrinsically integers (for example, counts of something).
- `double`: For regular work with values that may be decimal (for example, measured values) or values you do math with. Money amounts are also mostly OK.
- `byte`: For work with binary data.
- `long`: For big integer values such as file sizes, payment identifications (for example, ten digits may be required), or multiplication results of regular (whole) values.
- `decimal`: A common choice for money amounts.

The other types are not used that often.

Overflow

When program calculates a value that does not "fit" into an appropriate type's range, what happens is called *overflow*. The behavior of your program can be very strange, as shown in Figure 12-3.



```
Million times million: -727379968
also in long: -727379968
```

Figure 12-3. *Overflow*

Overflow can occur especially when multiplying because multiplying results in large numbers.

Task

In this section, you will write a program that tries to calculate a million times a million.

Solution

Here is the code:

```
static void Main(string[] args)
{
    // Multiplying million by million
    int million = 1000000;
    int result = million * million;
    long resultInLong = million * million;

    // Outputs
    Console.WriteLine("Million times million: " + result);
    Console.WriteLine("also in long: " + resultInLong);

    // Waiting for Enter
    Console.ReadLine();
}
```

Discussion

What the program does is totally unexpected. You need to be aware of this kind of anomaly.

What is actually happening? The program multiplies a million by a million. The result is too big to fit into the positive or negative two-billion range of the 32-bit signed `int` type. So, the computer simply throws away the upper bits, resulting in complete nonsense.

Please note that you get the same nonsense even when you store the result in a `long`-typed variable. That nonsense, which throws away the bits greater than 32, arises during calculation. According to C# rules, `int` times `int` is simply `int` regardless of where you store the result.

Dealing with Overflow

The previous program displayed an incorrect result. Now you will see what can be done about it.

Task

Here are two possibilities of how to handle overflow problems:

- If you do not expect a big value and it appears anyway, the program should at least crash or let you know about the problem. Displaying a nonsense value is the worst alternative. Users trust their computers and can make wrong decisions based upon believing incorrect results.
- If you have an idea that `int` might be insufficient, you can make the calculation correctly with the following solution.

Solution

The new project source code follows:

```
static void Main(string[] args)
{
    // 0. Preparation
    int million = 1000000;

    // 1. Crash at least, we do not
    //    definitely want a nonsense
    Console.WriteLine("1. calculation");
    try
    {
        long result = million * million;
        Console.WriteLine("Million times million:" + result);
    }
    catch (Exception)
    {
        Console.WriteLine("I cannot calculate this.");
    }

    // 2. Correct calculation of a big value
    Console.WriteLine("2. calculation");
    long millionInLong = million;
    long correctResult = millionInLong * millionInLong;
    Console.WriteLine("Million times million: " + correctResult.
    ToString("N0"));

    // 3. Alternative calculation of a big valule
    Console.WriteLine("3. calculation");
    long correctResultAlternatively = (long)million * (long)million;
    Console.WriteLine("Million times million: " +
    correctResultAlternatively.ToString("N0"));

    // Waiting for Enter
    Console.ReadLine();
}
```

Note

However, this code does not solve everything. When you immediately launch the program, the first calculation is still going to be wrong. People sometimes take `try-catch` as a kind of panacea, but it is definitely not. You need something else, as discussed next.

Settings in Visual Studio

You need to set up your project in Visual Studio so that it reports overflow out of the program instead of sweeping it under the rug.

From the Visual Studio menu, choose Project and then <Project name> Properties (see Figure 12-4).

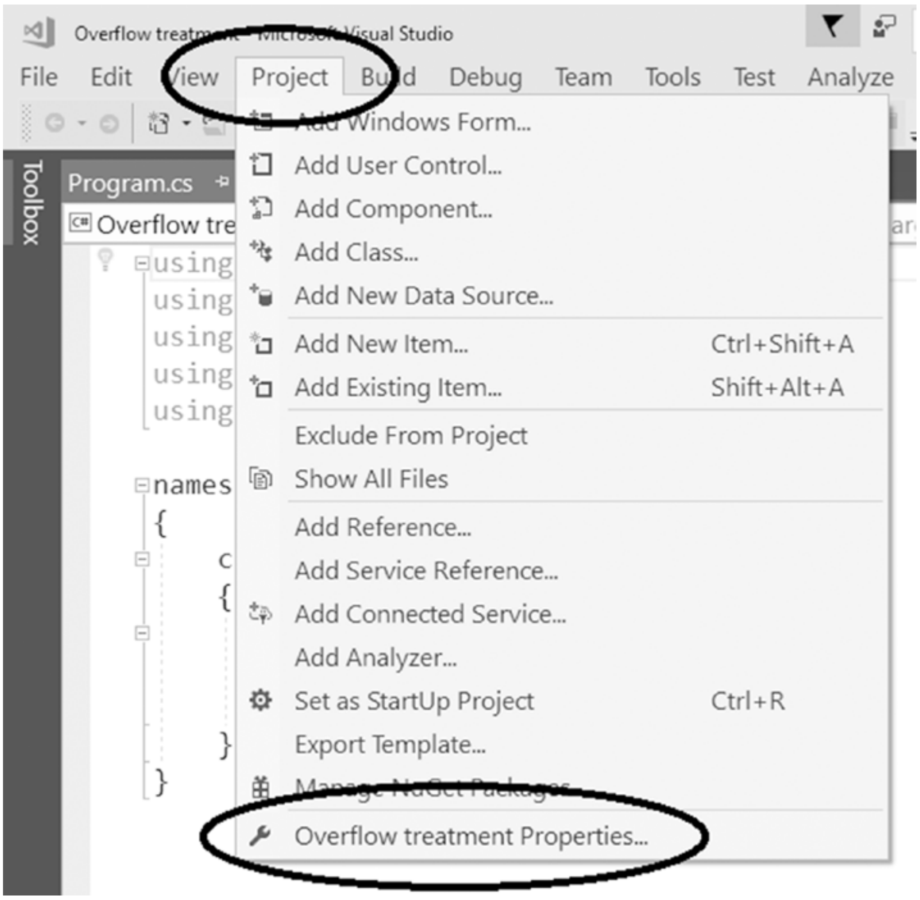
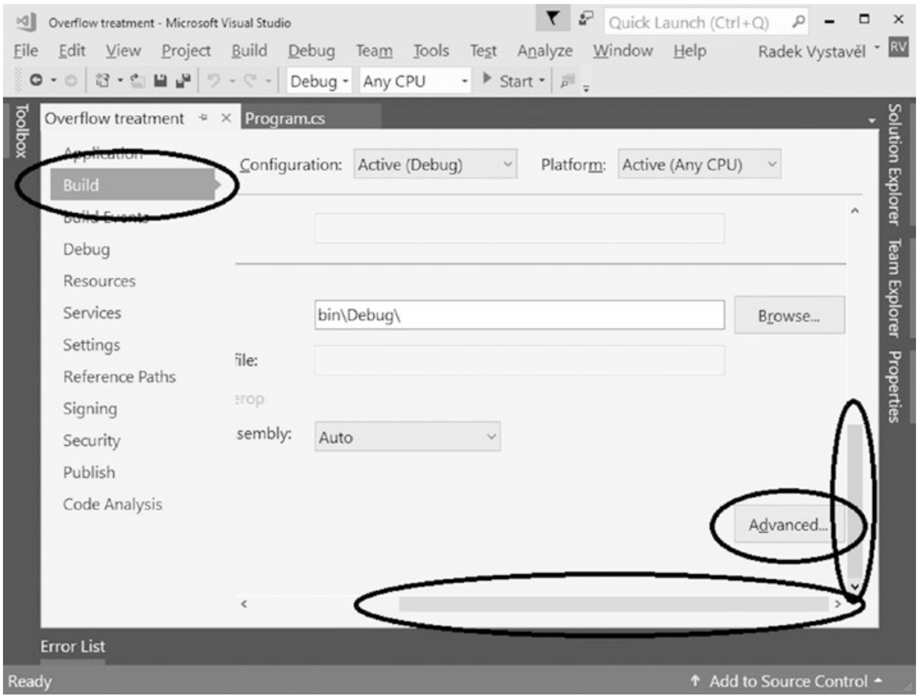


Figure 12-4. Opening the properties

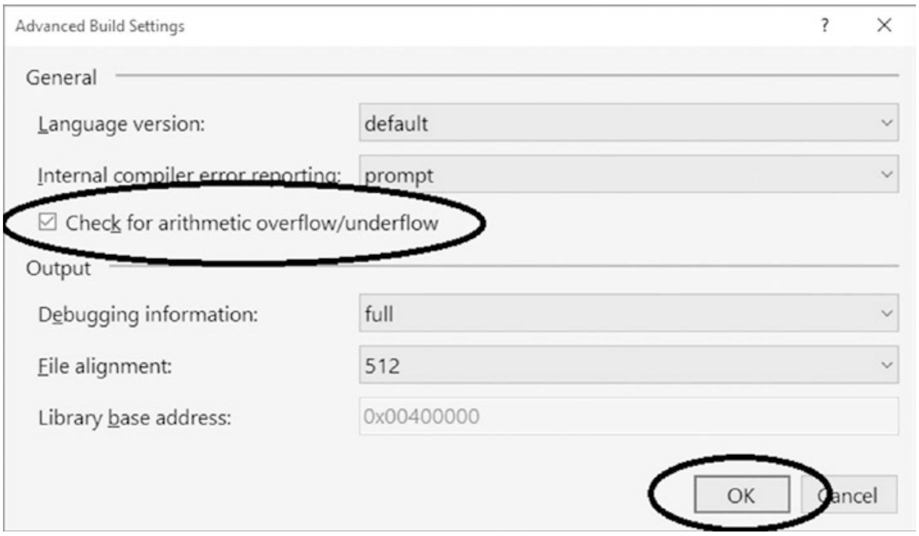
Choose the Build tab next, scroll vertically (and maybe also horizontally) so that you can see the Advanced button (it is really hidden!), and then click that button (see Figure 12-5).



Configuration: Active (Debug)
Platform: Active (Any CPU)
bin\Debug\
sembly: Auto

Figure 12-5. *Build tab*

In the dialog that appears, select the "Check for arithmetic overflow/underflow" check box and confirm by clicking the OK button (see Figure 12-6).



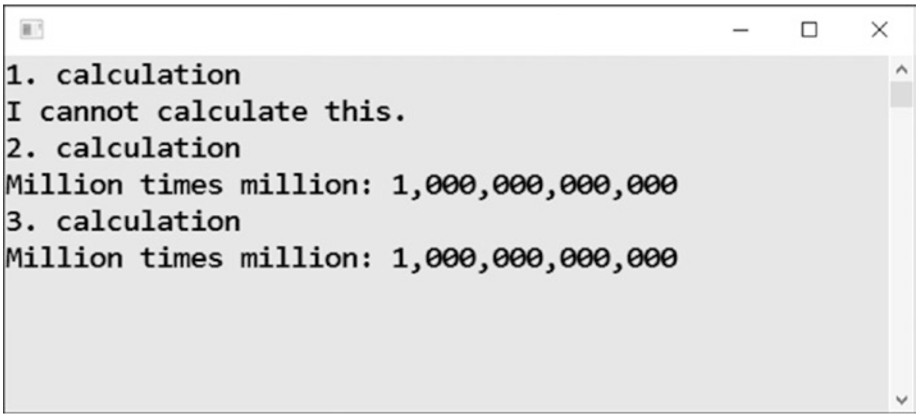
```
Advanced Build Settings
General
  Language version:                     default
  Internal compiler error reporting:    prompt
  ☑ Check for arithmetic overflow/underflow
Output
  Debugging information:                full
  File alignment:                       512
  Library base address:                 0x00400000
```

Figure 12-6. *"Check for arithmetic overflow/underflow" check box*

Your project is finally ready to run now.

Results

Now the program behaves according to expectations, as shown in Figure 12-7.



```
1. calculation
I cannot calculate this.
2. calculation
Million times million: 1,000,000,000,000
3. calculation
Million times million: 1,000,000,000,000
```

Figure 12-7. *Multiplying a million by a million*

First Alternative

The first calculation correctly reports a problem. Omitting `try-catch` would cause a runtime error, but at least it does not display an incorrect result.

Other Alternatives

A correct calculation converts the million into a `long` type *before* the calculation starts. Here are two ways to perform this conversion:

- Assigning the million to a variable of type `long`
- Using an explicit *type cast* with `(long)million`

If you do not require precise integer arithmetic, you might also calculate in the `double` type. Unless you solve some very exotic math, `double` does not have a chance to overflow.

Summary

In this chapter, you studied advanced number calculations. You got to know all the numeric data types that are available in C#. The types differ in whether they allow integers or decimals, and they differ also in the ranges of allowed values. Types for decimals mutually differ also in the precision with which the number is stored.

At the beginner level, knowledge of `int` and `double` is enough; you can always work using them only. When you become more experienced, you might also use the following:

- The `long` type for big integers such as file sizes, ten-digit payment numbers, or multiplication results of moderately sized numbers
- The `decimal` type for working with currency
- The `byte` type for working with binary data

You also studied the question of overflow. When a calculated value is too big to fit into the range of a particular data type, nonsense results. The default behavior of Visual Studio is to continue as normally. However, now you know how to change the settings to cause a runtime error at least, because continuing with the incorrect result is the worst alternative.

The best alternative is to avoid the overflow completely by choosing a data type with an appropriate range. However, keep in mind that changing the type of variable used to store the result may not be enough. For example, `int` multiplied by `int` is always `int` with a maximum value of about 2 billion, regardless of where you store it. It may be suitable to convert the number into a `long` type before the calculation.