Contents

Conversions

We now discuss implicit and explicit conversions between datatypes: how C# can (or not!) convert a value from one datatype to another, and how we can "force" this conversion if C# does not do it automatically.

Assignments from different types

- The "proper" way to initialize a variable is to assign it a literal of the same type:

```
int myAge = 29;
double myHeight = 1.77;
float radius = 2.3f;
```

Note that 1.77 is a double literal, while 2.3f is a float literal

- If the literal is not the same type as the variable, you will sometimes get an error – for example, `float radius = 2.3` will result in a compile error – but sometimes, it appears to work fine: for example `float radius = 2;` compiles and executes without error even though 2 is an int value.

- In fact, the value being assigned to the variable **must** be the same type as the variable, but some types can be **implicitly converted** to others

Implicit conversions

- Implicit conversion allows variables to be assigned from literals of the "wrong" type: the literal value is first implicitly converted to the right type

- In the statement `float radius = 2;`, the int value 2 is implicitly converted to an equivalent float value, 2.0f. Then the computer assigns 2.0f to the radius variable.

- Implicit conversion also allows variables to be assigned from other variables that have a different type:

```
int length = 2;
float radius = length;
```

When the computer executes the second line of this code, it reads the variable `length` to get an `int` value 2. It then implicitly converts that value to `2.0f`, and then assigns `2.0f` to the `float`-type variable `radius`.

- Implicit conversion only works between *some* data types: a value will only be implicitly converted if it is "safe" to do so without losing data

- Summary of possible implicit conversions:

Type	Possible Implicit Conversions
short	int, long, float, double, decimal
int	long, float, double, decimal
long	float, double, decimal
ushort	uint, int, ulong, long, decimal, float, double
uint	ulong, long, decimal, float, double
ulong	decimal, float, double
float	double

- In general, a data type can only be implicitly converted to one with a *larger range* of possible values

- Since an `int` can store any integer between -2^{31} and $2^{31}-1$, but a `float` can store any integer between -3.4×10^{38} and 3.4×10^{38} (as well as fractional values), it is always safe to store an `int` value in a `float`

- You *cannot* implicitly convert a `float` to an `int` because an `int` stores fewer values than a `float` – it cannot store fractions – so converting a `float` to an `int` will **lose data**

- Note that all integer data types can be implicitly converted to `float` or `double`

- Each integer data type can be implicitly converted to a larger integer type: `short` $\rightarrow$ `int` $\rightarrow$ `long`

- Unsigned integer data types can be implicitly converted to a *larger* signed integer type, but not the *same* signed integer type: `uint` $\rightarrow$ `long`, but **not** `uint` $\rightarrow$ `int`

- This is because of the "sign bit": a `uint` can store larger values than an `int` because it does not use a sign bit, so converting a large `uint` to an `int` might lose data

Explicit conversions

- Any conversion that is "unsafe" because it might lose data will not happen automatically: you get a compile error if you assign a `double` variable to a `float` variable

- If you want to do an unsafe conversion anyway, you must perform an **explicit conversion** with the **cast operator**

- Cast operator syntax: `([type name]) [variable or value]` – the cast is "right-associative", so it applies to the variable to the right of the type name

- Example: `(float) 2.8` or `(int) radius`

- Explicit conversions are often used when you (the programmer) know the conversion is actually "safe" – data will not actually be lost

- For example, in this code, we know that 2.886 is within the range of a `float`, so it is safe to convert it to a `float`:

```
float radius = (float) 2.886;
```

The variable `radius` will be assigned the value `2.886f`.

- For example, in this code, we know that 2.0 is safe to convert to an `int` because it has no fractional part:

```
double length = 2.0;
int height = (int) length;
```

The variable `height` will be assigned the value `2`.

- Explicit conversions only work if there exists code to perform the conversion, usually in the standard library. The cast operator isn't "magic" – it just calls a method that is defined to convert one type of data (e.g. `double`) to another (e.g. `int`).

- All the C# numeric types have explicit conversions to each other defined

- `string` does not have explicit conversions defined, so you cannot write `int myAge = (int) "29";`

- If the explicit conversion is truly unsafe (will lose data), data is lost in a specific way

- Casting from floating-point (e.g. `double`) types to integer types: fractional part of number is *truncated* (ignored/dropped)

- In `int length = (int) 2.886;`, the value 2.886 is truncated to 2 by the cast to `int`, so the variable `length` gets the value 2.

- Casting from more-precise to less-precise floating point type: number is *rounded* to nearest value that fits in less-precise type:

```
decimal myDecimal = 123456789.999999918m;
double myDouble = (double) myDecimal;
float myFloat = (float) myDouble;
```

In this code, `myDouble` gets the value 123456789.99999993, while `myFloat` gets the value `123456790.0f`, as the original `decimal` value is rounded to fit types with fewer significant figures of precision.

- Casting from a larger integer to a smaller integer: the most significant *bits* are truncated – remember that numbers are stored in binary format

- This can cause weird results, since the least-significant *bits* of a number in binary do not correspond to the least significant *digits* of the equivalent base-10 number

- Casting from another floating point type to `decimal`: Either value is stored precisely (no rounding), or *program crashes* with `System.OverflowException` if value is larger than `decimal`'s maximum value:

```
decimal fromSmall = (decimal) 42.76875;
double bigDouble = 2.65e35;
decimal fromBig = (decimal) bigDouble;
```

In this code, `fromSmall` will get the value `42.76875m`, but the program will crash when attempting to cast `bigDouble` to a `decimal` because 2.65×10^{35} is larger than `decimal`'s maximum value of 7.9×10^{28}

- `decimal` is more precise than the other two floating-point types (thus does not need to round), but has a smaller range (only 10^{28}, vs. 10^{308})

Summary of implicit and explicit conversions for the numeric datatypes:

Refer to the "Result Type of Operations" chart from the cheatsheet[1] for more detail.

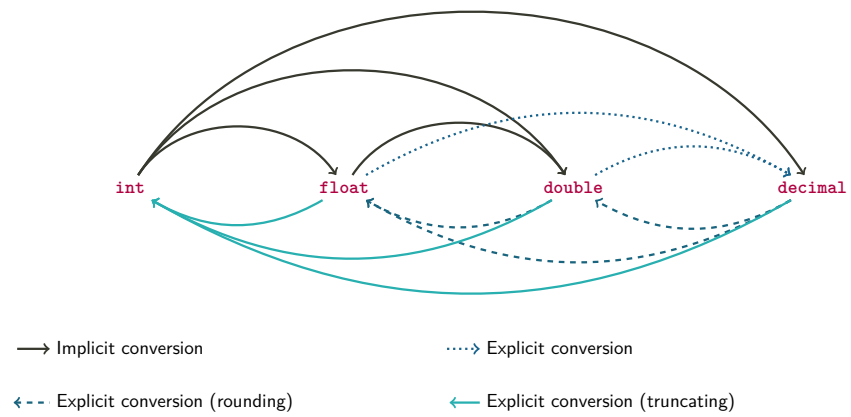[1] https:/princomp.github.io/docs/programming_and_computer_usage/datatypes_in _csharp

Figure 1: "Implicit and Explicit Conversion Between Datatypes"