Contents

Datatypes and Variables

Datatype Basics

- Recall the basic structure of a program
  - Program receives input from some source, uses input to make decisions, produces output for the outside world to see
  - In other words, the program reads some data, manipulates data, and writes out new data
  - In C#, data is stored in objects during the program's execution, and manipulated using the methods of those objects
- This data has **types**
  - Numbers (the number 2) are different from text (the word "two")
  - Text data is called "strings" because each letter is a **character** and a word is a *string of characters*
  - Within "numeric data," there are different types of numbers
    - Natural numbers ($\mathbb{N}$): 0, 1, 2, ...
    - Integers ($\mathbb{Z}$): ... -2, -1, 0, 1, 2, ...
    - Real numbers ($\mathbb{R}$): 0.5, 1.333333..., -1.4, etc.
- Basic Datatypes in C#
  - C# uses keywords to name the types of data
  - Text data:
    - `string`: a string of characters, like `"Hello world!"`
    - `char`: a single character, like `'e'` or `'t'`
  - Numeric data:

* `int`: An integer, as defined previously
* `uint`: An *unsigned* integer, in other words, a natural number (positive integers only)
* `float`: A "floating-point" number, which is a real number with a fractional part, such as 3.85
* `double`: A floating-point number with "double precision" – also a real number, but capable of storing more significant figures
* `decimal`: An "exact decimal" number – also a real number, but has fewer rounding errors than `float` and `double` (we will explore the difference later) [1]

Literals and Variables

Literals and their types

- A **literal** is a data value written in the code
- A form of "input" provided by the programmer rather than the user; its value is fixed throughout the program's execution
- Literal data must have a type, indicated by syntax:
  - `string` literal: text in double quotes, like `"hello"`
  - `char` literal: a character in single quotes, like `'a'`
  - `int` literal: a number without a decimal point, with or without a minus sign (e.g. `52`)
  - `long` literal: just like an `int` literal but with the suffix `l` or `L`, e.g. `4L`
  - `double` literal: a number with a decimal point, with or without a minus sign (e.g. `-4.5`)
  - `float` literal: just like a `double` literal but with the suffix `f` or `F` (for "float"), e.g. `4.5f`
  - `decimal` literal: just like a `double` literal but with the suffix `m` or `M`(for "deciMal"), e.g. `6.01m`

Variables overview

- Variables store data that can *vary* (change) during the program's execution
- They have a type, just like literals, and also a name

[1] At this point, you may wonder "why don't we always use the most precise datatype instead of using imprecise ones?". There are three dimensions to consider to answer this question: first, using `decimal` takes more memory, hence more time, than the other numerical datatypes. Second, they are a bit more cumbersome to manipulate, as we will see later on. Last, you generally don't need to be *that* precise: for example, it would not make sense to use a floating-point number to account for human beings or other indivisible units. Even decimal may be an overkill for floating-point values sometimes: for instance, the NASA uses `3.141592653589793` as an approximation of pi[2] for their calculations. A `double` can hold such a value[3], so there is no need to be more precise.

- You can use literals to write data that gets stored in variables
- Sample program with variables:

```
using System;

class MyFirstVariables
{
  static void Main()
  {
    // Declaration
    int myAge;
    string myName;
    // Assignment
    myAge = 29;
    myName = "Edward";
    // Displaying
    Console.WriteLine(
      $"My name is {myName} and I am {myAge} years
↪   old."
    );
  }
}
```

This program shows three major operations you can do with variables.

  - First it **declares** two variables, an `int`-type variable named "myAge" and a `string`-type variable named "myName"
  - Then, it **assigns** values to each of those variables, using literals of the same type. `myAge` is assigned the value 29, using the `int` literal `29`, and `myName` is assigned the value "Edward", using the `string` literal `"Edward"`
  - Finally, it **displays** the current value of each variable by using the `Console.WriteLine` method and **string interpolation**, in which the values of variables are inserted into a string by writing their names with some special syntax (a $ character at the beginning of the string, and braces around the variable names)

Variable Operations

Declaration

- This is when you specify the *name* of a variable and its *type*
- The syntax is the type keyword, a space, the name of the variable, then a semi-colon.
- Examples: `int myAge;`, `string myName;`, `double winChance;`.

- A variable name is an identifier, so it should follow the rules and conventions
  - Can only contain letters and numbers
  - Must be unique among all variable, method, and class names
  - Should use CamelCase if it contains multiple words
- Note that the variable's type is not part of its name: two variables cannot have the same name *even if* they are different types
- Multiple variables can be declared in the same statement: `string myFirstName, myLastName;` would declare *two* strings called respectively `myFirstName` and `myLastName`

Assignment

- The act of changing the value of a variable
- Uses the symbol =, which is the *assignment operator*, not a statement of equality – it does not mean "equals"
- Direction of assignment is **right to left**: the variable goes on the left side of the = symbol, and its new value goes on the right
- Syntax: `variable_name = value;`
- Example: `myAge = 29;`
- Value *must* match the type of the variable. If `myAge` was declared as an `int`-type variable, you cannot write `myAge = "29";` because `"29"` is a `string`

Initialization (Declaration + Assignment)

- Initialization statement combines declaration and assignment in one single statement (it is just a shortcut, a.k.a. some "syntactical sugar"[4], and not something new)
- Creates a new variable and also gives it an initial value
- The syntax is the datatype of the variable, the name of the variable, the = sign, the value we want to store, and a semi-colon
- Example: `string myName = "Edward";`
- Can only be used once per variable, since you can only declare a variable once

Assignment Details

- Assignment replaces the "old" value of the variable with a "new" one; it is how variables *vary*
  - If you initialize a variable with `int myAge = 29;` and then write `myAge = 30;`, the variable `myAge` now stores the value 30

[4] https://www.wikiwand.com/en/Syntactic_sugar

- You can assign a variable to another variable: just write a variable name on both sides of the = operator
  - This will take a "snapshot" of the current value of the variable on the right side, and store it into the variable on the left side
  - For example, in this code:

    ```
    int a = 12;
    int b = a;
    a = -5;
    ```

    the variable b gets the value 12, because that's the value that a had when the statement `int b = a` was executed. Even though a was then changed to -5 afterward, b is still 12.

Displaying

- Only text (strings) can be displayed in the console
- When we want to print a mixture of text and variables with `Console.WriteLine`, we need to convert all of them to a string
- **String interpolation**: a mechanism for converting a variable's value to a `string` and inserting it into the main string
  - Syntax: `$"text {variable} text"` – begin with a $ symbol, then put variable's name inside brackets within the string
  - Example: `$"I am {myAge} years old"`
  - When this line of code is executed, it reads the variable's current value, converts it to a string (`29` becomes `"29"`), and inserts it into the surrounding string
  - Displayed: `I am 29 years old`
- If the argument to `Console.WriteLine` is the name of a variable, it will automatically convert that variable to a `string` before displaying it
- For example, `Console.WriteLine(myAge);` will display "29" in the console, as if we had written `Console.WriteLine($"{myAge}");`
- When string interpolation converts a variable to a string, it must call a "string conversion" method supplied with the data type (`int`, `double`, etc.). All built-in C# datatypes come with string conversion methods, but when you write your own data types (classes), you'll need to write your own string conversions – string interpolation will not magically "know" how to convert `MyClass` variables to `string`s

On a final note, observe that you can write statements mixing multiple declarations and assignments, as in `int myAge = 10, yourAge, ageDifference;` that declares three variables of type `int` and set the value of the first one. It is generally recommended to separate those instructions in different statements as you begin, to ease debugging and have a better understanding of the "atomic steps" your program should perform.

Format Specifiers

- Formats for displaying numbers
  - There are lots of possible ways to display a number, especially a fraction (how many decimal places to use?)
  - String interpolation has a default way to format numbers, but it might not always be the best
  - For example, consider this program:

    ```
    decimal price = 19.99m;
    decimal discount = 0.25m;
    decimal salePrice = price - discount * price;
    Console.WriteLine($"{price} with a discount of " +
        $"{discount} is {salePrice}");
    ```

    It will display this output:

    ```
    19.99 with a discount of 0.25 is 14.9925
    ```

    But this isn't the best way to display prices and discounts. Obviously, the prices should have dollar signs, but also, it does not make sense to show a price with fractional cents (14.9925) – it should be rounded to two decimal places. You might also prefer to display the discount as 25% instead of 0.25, since people usually think of discounts as percentages.
- Improving interpolation with format specifiers
  - You can change how numbers are displayed by adding a format specifier to a variable's name in string interpolation
  - **Format specifier**: A special letter indicating how a numeric value should be converted to a string
  - General format is `{[variable]:[format specifier]}`, e.g. `{numVar:N}`
  - Common format specifiers:

Format specifier	Description
N or n	Adds a thousands separator, displays 2 decimal places (by default)
E or e	Uses scientific notation, displays 6 decimal places (by default)
C or c	Formats as currency: Adds a currency symbol, adds thousands separator, displays 2 decimal places (by default)
P or p	Formats as percentage with 2 decimal places (by default)

  - Example usage with our "discount" program:

    ```
    decimal price = 19.99m;
    decimal discount = 0.25m;
    ```

```
decimal salePrice = price - discount * price;
Console.WriteLine($"{price:C} with a discount of
↪   " +
    $"{discount:P} is {salePrice:C}");
```

will display

```
$19.99 with a discount of 25.00% is $14.99
```

- Format specifiers with custom rounding
  - Each format specifier uses a default number of decimal places, but you can change this with a precision specifier
  - **Precision specifier**: A number added after a format specifier indicating how many digits past the decimal point to display
  - Format is `{[variable]:[format specifier][precision specifier]}`, e.g. `{numVar:N3}`. Note there is no space or other symbol between the format specifier and the precision specifier, and the number can be more than one digit (`{numVar:N12}` is valid)
  - Examples:
    - Given the declarations

      ```
      double bigNumber = 1537963.666;
      decimal discount = 0.1337m;
      ```

Statement	Display
`Console.WriteLine($"{bigNumber:N}");`	`1,537,963.67`
`Console.WriteLine($"{bigNumber:N3}");`	`1,537,963.666`
`Console.WriteLine($"{bigNumber:N1}");`	`1,537,963.7`
`Console.WriteLine($"{discount:P1}");`	`13.4%`
`Console.WriteLine($"{discount:P4}");`	`13.3700%`
`Console.WriteLine($"{bigNumber:E}");`	`1.537964E+006`
`Console.WriteLine($"{bigNumber:E2}");`	`1.54E+006`

Variables in Memory

- A variable names a memory location
- Data is stored in memory (RAM), so a variable "stores data" by storing it in memory
- Declaring a variable reserves a memory location (address) and gives it a name
- Assigning to a variable stores data to the memory location (address) named by that variable

Sizes of Numeric Datatypes

- Numeric datatypes have different sizes

- Amount of memory used/reserved by each variable depends on the variable's type
- Amount of memory needed for an integer data type depends on the size of the number
  - `int` uses 4 bytes of memory, can store numbers in the range $[-2^{31}, 2^{31} - 1]$
  - `long` uses 8 bytes of memory can store numbers in the range $[-2^{63}, 2^{63} - 1]$
  - `short` uses 2 bytes of memory, can store numbers in the range $[-2^{15}, 2^{15} - 1]$
  - `sbyte` uses only 1 bytes of memory, can store numbers in the range $[-128, 127]$
- Unsigned versions of the integer types use the same amount of memory, but can store larger positive numbers
  - `byte` uses 1 byte of memory, can store numbers in the range $[0, 255]$
  - `ushort` uses 2 bytes of memory, can store numbers in the range $[0, 2^{16} - 1]$
  - `uint` uses 4 bytes of memory, can store numbers in the range $[0, 2^{32} - 1]$
  - `ulong` uses 8 bytes of memory, can store numbers in the range $[0, 2^{64} - 1]$
  - This is because in a signed integer, one bit (digit) of the binary number is needed to represent the sign (+ or -). This means the actual number stored must be 1 bit smaller than the size of the memory (e.g. 31 bits out of the 32 bits in 4 bytes). In an unsigned integer, there is no "sign bit", so all the bits can be used for the number.
- Amount of memory needed for a floating-point data type depends on the precision (significant figures) of the number
  - `float` uses 4 bytes of memory, can store positive or negative numbers in a range of approximately $[10^{-45}, 10^{38}]$, with 7 significant figures of precision
  - `double` uses 8 bytes of memory, and has both a wider range (10^{-324} to 10^{308}) and more significant figures (15 or 16)
  - `decimal` uses 16 bytes of memory, and has 28 or 29 significant figures of precision, but it actually has the smallest range (10^{-28} to 10^{28}) because it stores decimal fractions exactly
- Difference between binary fractions and decimal fractions
  - `float` and `double` store their data as binary (base 2) fractions, where each digit represents a power of 2
    - The binary number 101.01 represents $4 + 1 + 1/4$, or 5.25 in base 10
  - More specifically, they use binary scientific notation: A man-

tissa (a binary integer), followed by an exponent assumed to be a power of 2, which is applied to the mantissa
    - 10101e-10 means a mantissa of 10101 (i.e. 21 in base 10) with an exponent of -10 (i.e. 2^{-2} in base 10), which also produces the value 101.01 or 5.25 in base 10
- Binary fractions cannot represent all base-10 fractions, because they can only represent fractions that are negative powers of 2. $1/10$ is not a negative power of 2 and cannot be represented as a sum of $1/16, 1/32, 1/64$, etc.
- This means some base-10 fractions will get "rounded" to the nearest finite binary fraction, and this will cause errors when they are used in arithmetic
- On the other hand, `decimal` stores data as a base-10 fraction, using base-10 scientific notation
- This is slower for the computer to calculate with (since computers work only in binary) but has no "rounding errors" with fractions that include 0.1
- Use `decimal` when working with money (since money uses a lot of 0.1 and 0.01 fractions), `double` when working with non-money fractions

Summary of numeric data types and sizes:

Type	Size	Range of Values	Precision
`sbyte`	1 bytes	$-128...127$	N/A
`byte`	1 bytes	$0...255$	N/A
`short`	2 bytes	$-2^{15}...2^{15}-1$	N/A
`ushort`	2 bytes	$0...2^{16}-1$	N/A
`int`	4 bytes	$-2^{31}...2^{31}-1$	N/A
`uint`	4 bytes	$0...2^{32}-1$	N/A
`long`	8 bytes	$-2^{63}...2^{63}-1$	N/A
`ulong`	8 bytes	$0...2^{64}-1$	N/A
`float`	4 bytes	$\pm 1.5 \cdot 10^{-45}... \pm 3.4 \cdot 10^{38}$	7 digits
`double`	8 bytes	$\pm 5.0 \cdot 10^{-324}... \pm 1.7 \cdot 10^{308}$	15-16 digits
`decimal`	16 bytes	$\pm 1.0 \cdot 10^{-28}... \pm 7.9 \cdot 10^{28}$	28-29 digits

Value and Reference types

- Value and reference types are different ways of storing data in memory

- Variables name memory locations, but the data that gets stored at the named location is different for each type
- For a **value type** variable, the named memory location stores the exact data value held by the variable (just what you'd expect)
- Value types: all the numeric types (`int`, `float`, `double`, `decimal`, etc.), `char`, and `bool`
- For a **reference type** variable, the named memory location stores a *reference* to the data, not the data itself
  - The contents of the memory location named by the variable are the address of another memory location
  - The *other* memory location is where the variable's data is stored
  - To get to the data, the computer first reads the location named by the variable, then uses that information (the memory address) to find and read the other memory location where the data is stored
- Reference types: `string`, `object`, and all objects you create from your own classes
- Assignment works differently for reference types
  - Assignment always copies the value in the variable's named memory location - but in the case of a reference type that's just a memory address, not the data
  - Assigning one reference-type variable to another copies the memory address, so now both variables "refer to" the same data
  - Example:

    ```
    string word = "Hello";
    string word2 = word;
    ```

    Both `word` and `word2` contain the same memory address, pointing to the same memory location, which contains the string "Hello". There is only one copy of the string "Hello"; `word2` does not get its own copy.