Chapter 3 – Variables and Expressions

To use C# effectively, it’s important to understand what you’re actually doing when you create a computer program. The most basic description of a computer program is that it is a series of operations that manipulate data.

Consider the display unit of your computer. What you see onscreen is often so familiar that it is difficult to imagine it as anything other than a “moving picture.” In fact, what you see is only a representation of some data, which in its raw form is merely a stream of 0s and 1s stashed away somewhere in the computer’s memory. Any onscreen action – moving a mouse pointer, clicking on an icon, typing text into a word processor – results in the shunting around of data in memory.

Therefore, if computer programs are fundamentally performing operations on data, this implies that you need a way to store that data, and some methods to manipulate it. These two functions are provided by **variables** and **expressions**, respectively.

## Basic C# Syntax

Unlike the compilers of some other languages such as Python, C# compilers ignore additional spacing in code, whether it results from spaces, carriage returns, or tab characters (collectively known as whitespace characters). This means that you have a lot of freedom in the way that you format your code, although conforming to certain rules to make code easier to read.

C# code is made up of a series of statements, each of which is terminated with a semi-colon. Because whitespace is ignored, multiple statements can appear on one line, although for readability it is usual to add carriage returns after semicolons - to avoid multiple statements on one line. It is perfectly acceptable however to use statements that span several lines of code.

C# is a **block-structured language,** meaning statements are part of a block of code. These blocks, which are delimited with curly braces ({ and } ), may contain any number of statements, or none at all. **\*\*Note – that curly bracket characters do not need accompanying semicolons.\*\***

**For example, a simple C# code block could take the following form:**

{

<code line 1, statement 1>;  
<code line 2, statement 2>  
 <code line 3, statement 2>;

}

In this case, the second and third lines of code are part of the same statement, because there is no semicolon after the second line. Indenting the third line makes it easier to recognise that it is a continuation of line two.

**\*\*Check Tools > Options > Text Editor > C# to see the rules that Visual Studio uses for formatting your code.\*\***

**Comments –** All comments are ignored by the compiler. There are two methods:

**Method One**: Start with /\* …(insert comments)… \*/ End with

**For example:**

/\*

This is a comment

\*/

**Method Two**: //...(insert comments)… - can write whatever you like as long as it is on one line.

**For example:**

// This is a comment

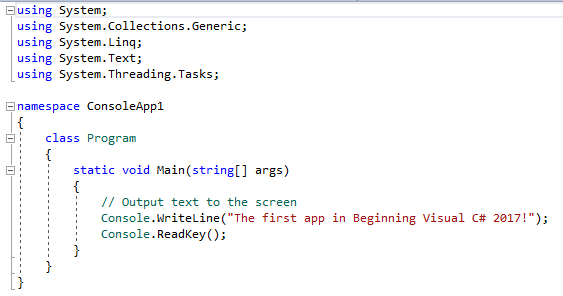
**\*\*Special third method\*\***

/// This is a comment

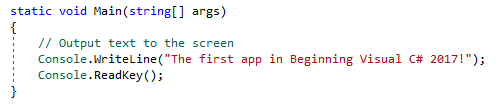
You can configure Visual Studio to extract the text after these comments to create a specially formatted text file when a project is compiled. This can then be used to create documentation. Must follow XML documentation rules.

C# Code is **CASE SENSITIVE**.

From the Console application written previously we find ourselves with the following code to run the console app.

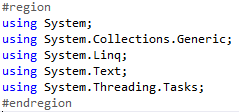


The most important section of this code is the following:



When the console application is run the code contained within the curly braces is executed. The comment line doesn’t do anything and the other two code lines output some text to the console window and wait for a response, respectively.

You can use the **#region** and **#endregion** keywords to define the start and end of a region of code that can be expanded and collapsed.





This enables you to collapse this code into a single line and expand it again later if needed.

**NOTE** Any keyword that starts with a # is actually a pre-processor directive and not, strictly speaking, a C# keyword. Other than the two described here, **#region** and **#endregion,** these can be quite complicated and have very specialised uses.

## Variables

Variables are concerned with the storage of data. Essentially, you can think of variables in computer memory as boxes sitting on a shelf. You can put things in boxes and take them out again, or you can just look inside a box to see if anything is there. The same goes for variables; you place data in them and can take it out or look at it, as required.

Variables come in different flavours known as **types.** Using the box analogy again, boxes come in different shapes and sizes, so some items fit only certain boxes. The reasoning behind the type system is that different types of data may require different methods of manipulation, and by restricting variables to individual types you can avoid mixing them up. For example, it wouldn’t make much sense to treat the series of 1s and 0s that make up a picture as an audio file.

To use variables, you must declare them. This means you have to assign them a **name** and a **type.** After you have declared variables, you can use them as storage units for the type of data that you declared them to hold.

C# syntax for declaring variables merely specifies the type and variable name:

*<type> <name>;*

If you try to use a variable that hasn’t been declared your code will not compile but the error log will notify you of this.

### Simple Types

Simple Types include types such as **numbers** and **Boolean** values that make up the fundamental building blocks for your application. Unlike complex types, simple types cannot have children or attributes. Most simple types are **numeric**.

The reason that there are a number of integer types is because of the mechanics of storing numbers as a series of 0s and 1s in memory. For integer values, you simply take a number of bits and represent your number in binary format. A variable storing N bits enables you to represent any number between 0 and (2n – 1). Any number above this value are too big to fit into this variable.

For example, suppose you have a variable that can store two bits. The mapping between integers and the bits representing those integers is therefore as follows:

* **0 = 00**
* **1 = 01**
* **2 = 10**
* **3 = 11**

The inevitable result of this system is that you will need an infinite number of bits to be able to store every imaginable number, which won’t fit into a PC. Instead, to combat this a number of different integer types can be used to stores various ranges of numbers, that take up different amounts of memory.

|  |  |  |
| --- | --- | --- |
| **TYPE** | **ALIAS FOR** | **ALLOWED VALUES** |
| sbyte | System.SByte | Integer between –128 and 127 |
| byte | System.Byte | Integer between 0 and 255 |
| short | System.Int16 | Integer between –32768 and 32767 |
| ushort | System.UInt16 | Integer between 0 and 65535 |
| int | System.Int32 | Integer between –2147483648 and 2147483647 |
| uint | System.UInt32 | Integer between 0 and 4294967295 |
| long | System.Int64 | Integer between –9223372036854775808 and 9223372036854775807 |
| ulong | System.UInt64 | Integer between 0 and 18446744073709551615 |

The **u** characters before some of the variable names are shorthand for ***unsigned.*** This means that you can’t store negative numbers in variables of those types.

There is also a need to store ***floating-point*** values, those that are not whole numbers. There are three floating-point variable types: **float, double,** and **decimal**.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **TYPE** | **ALIAS FOR** | **MIN M** | **MAX M** | **MIN E** | **MAX E** | **APPROX MIN VALUE** | **APPROX MAX VALUE** |
| float | System.Single | 0 | 224 | -149 | 104 | 1.5 x 10**-**45 | 3.4 x 1038 |
| double | System.Double | 0 | 253 | -1075 | 970 | 5.0 x 10**-**3²4 | 1.7 x 10308 |
| decimal | System.Decimal | 0 | 296 | -28 | 0 | 1.0 x 10**-**²8 | 7.9 x 10²8 |

In addition to numeric types, three other simple types are available: **char, bool,** and **string**.

|  |  |  |
| --- | --- | --- |
| **TYPE** | **ALIAS FOR** | **ALLOWED VALUES** |
| char | System.Char | single Unicode character, stored as an integer between 0 and 65535. |
| bool | System.Boolean | Boolean value, true or false. |
| string | System.String | A sequence of characters. |

There is no upper limit on the number of characters making up a **string**, because it can use varying amounts of memory.

The **Boolean** type **bool** is one of the most commonly used variable types in C#. Having a variable that can be either true or false has important ramifications when it comes to the flow of logic in an application.

**TRY IT OUT – Using Simple Type Variables**

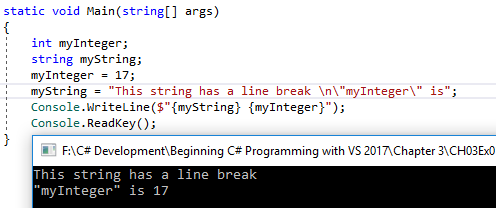
In this short example we will be declaring two variable, assigning them values, and then outputting these values.



**What this code does is:**

1. Declare two variables – string & int.
2. Assigns values to the two variables.
   1. Here we assign two fixed values known as ***literal values*** to our variables using the = assignment operator.
   2. When you assign ***string literal values*** in this way, double quotation marks are required to enclose the string. This means that certain characters may cause problems if included in the string itself. E.g. double quotation marks themselves. To get around this we must use an escape sequence which in this case is \”.
3. Outputs the values of the two variables to the console.

**NOTE** **:** Assigning string literals is another situation in which we must be careful of line breaks. The C# compiler will reject string literals that span more than one line. To add a line break you must use the escape sequence for a new line **\n.** For example:



**ALL** escape sequences consist of the backslash symbol followed by one of a small set of characters. As this symbol is used for this purpose, these is also an escape sequence for the backslash symbol itself, which is simply two consecutive backslashes (\ \).

***String Interpolation***

**Console.WriteLine($"{myString} {myInteger}");**

new feature of C# 6 and will be discussed later on.

## Literal Values

In the previous example we seen two examples of literal values : integer (17) and string ("This string has a line break \n\"myInteger\" is").

|  |  |  |  |
| --- | --- | --- | --- |
| **TYPE(S)** | **CATEGORY** | **SUFFIX** | **EXAMPLE/ALLOWED VALUES** |
| bool | Boolean | **None** | True or False |
| int, uint, long, ulong | Integer | **None** | 100 |
| uint, ulong | Integer | **u or U** | 100U |
| long, ulong | Integer | **l or L** | 100L |
| ulong | Integer | **ul, uL, Ul, UL, lu, lU, Lu, or LU** | 100UL |
| float | Real | **f or F** | 1.5F |
| double | Real | **None, d, or D** | 1.5 |
| decimal | Real | **m or M** | 1.5M |
| char | Character | **None** | ‘a’, or escape sequence |
| string | String | **None** | “a. . .a”, may include escape sequences |

### Binary Literals and Digit Separators

No matter how sophisticated or complex programming syntax becomes, computers function in only 2 states, 0 and 1, also known as binary (base 2). Without going deep into the historical understandings of bits, bytes, characters etc. It is enough to know that you can use binary literals as an elegant way to store values as constants for pattern matching and comparison, as well as for implementing bit masks.

In the binary vs hex example below, you can see that the binary numbers are rotated by a single bit from right to left. The hex values have no pattern.

**int[] binaryPhases = [0b00110001, 0b01100010, 0b11000100, 0b10001001];**

**int[] hexPhases = [0x31, 0x62, 0xC4, 0x89];**

### String Literals

|  |  |  |
| --- | --- | --- |
| **ESCAPE SEQUENCE** | **CHARACTER PRODUCED** | **UNICODE (HEX) VALUE OF CHARACTER** |
| \’ | Single quotation mark | 0x0027 |
| \” | Double quotation mark | 0x0022 |
| \\ | Backslash | 0x005C |
| \0 | Null | 0x0000 |
| \a | Alert (causes a beep) | 0x0007 |
| \b | Backspace | 0x0008 |
| \f | Form feed | 0x000C |
| \n | New line | 0x000A |
| \r | Carriage return | 0x000D |
| \t | Horizontal tab | 0x0009 |
| \v | Vertical tab | 0x000B |

## Expressions

You can create ***expressions by*** combining operators with variables and literal values (together referred to as ***operands***). Expressions are the basic building blocks of computation. The simple operators include all the basic mathematical operations, such as the **+** operator to add two operands.

Operators can be roughly classified into three categories:

* **Unary –** Act on single operands
* **Binary –** Act on two operands
* **Ternary –** Act on three operands

Most operators fall into the binary category, with a few unary ones, and a single ternary one call the ***conditional operator.***

### Mathematical Operators

|  |  |  |  |
| --- | --- | --- | --- |
| **OPERATOR** | **CATEGORY** | **EXAMPLE EXPRESSION** | **RESULT** |
| **+** | Binary | var1 = var2 + var3; | **var1** is assigned the value that is the sum of **var2** & **var3**. |
| **-** | Binary | var1 = var2 - var3; | **var1** is assigned the value of **var2** minus **var3**. |
| **\*** | Binary | var1 = var2 \* var3; | **var1** is assigned the value of **var2** multiplied by **var3**. |
| **/** | Binary | var1 = var2 / var3; | **var1** is assigned the value of dividing **var2** by **var3** |
| **%** | Binary | var1 = var2 % var3; | **var1** is assigned the value that is the remainder when **var2** is divided by **var3**. |
| **+** | Unary | var1 = +var2; | **var1** is assigned the value of **var2**. |
| **-** | Unary | var1 = -var2; | **var1** is assigned the value of **var2** multiplied by –1. |

If you try to add two bool variables together the compiler will complain and nothing will happen (or if you use any mathematical operator) with bool variables.

When adding char variables this will result in a number (of type int) as a char variable is actually a number. This is an example of ***implicit conversion****.*

The binary + operator **does** make sense when used with string type variables. ***See table below.***

#### String Concatenation Operator

|  |  |  |  |
| --- | --- | --- | --- |
| **OPERATOR** | **CATEGORY** | **EXAMPLE EXPRESSION** | **RESULT** |
| + | Binary | var1 = var2 + var3; | **var1** is assigned the value that is the concatenation of the two strings stored in **var2** & **var3**. |

**\*\*NONE of the other mathematical operators work with strings\*\***

The other two operators you should look at here are the **increment** and **decrement** operators, both of which are **unary operators.** These operators can be used in two ways: either immediately before or immediately after the operand.

#### Increment and Decrement Operators

|  |  |  |  |
| --- | --- | --- | --- |
| **OPERATOR** | **CATEGORY** | **EXAMPLE EXPRESSION** | **RESULT** |
| ++ | Unary | var1 = ++var2; | **var1** is assigned the value of **var2 + 1.var2** is incremented by 1 |
| -- | Unary | var1 = --var2; | **var1** is assigned the value of **var2 - 1.var2** is decremented by 1 |
| ++ | Unary | var1 = var2++; | **var1** is assigned the value of **var2 + 1.var2** is incremented by 1 |
| -- | Unary | var1 = var2--; | **var1** is assigned the value of **var2 - 1.var2** is decremented by 1 |

**++ always results in its operand being incremented by one.**

**-- always results in its operand being decremented by one.**

The differences between the results stored in var1 are a consequence of a fact that the placement of the operator determines when it takes effect. Placing the operator before the operation means that the operand is affected before any other computation takes place. Placing it after means that the operand will be affected after all other computation of the expression is completed.

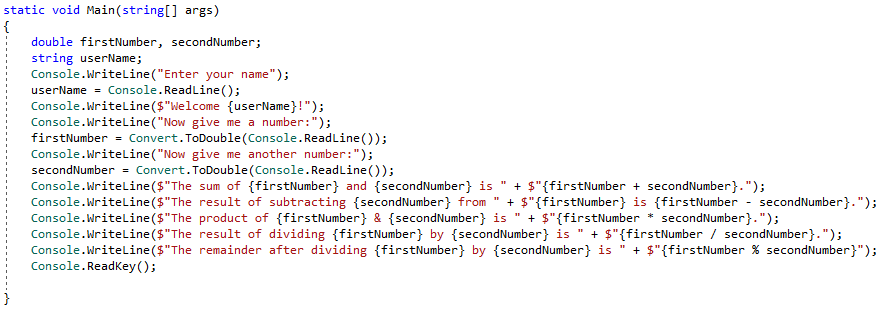
int var1, var2 = 5, var3 = 6;  
var1 = var2++ \* --var3;

What value will be assigned to var1? Before the expression is evaluated, the -- operator preceding var3 takes effect, changing its value from 6 to 5 (decrementing it by 1). We can ignore the ++operator that follows var2, as t won’t take effect until after the calculation is completed. Therefore, var1 will be the product of 5 \* 5 = 25.

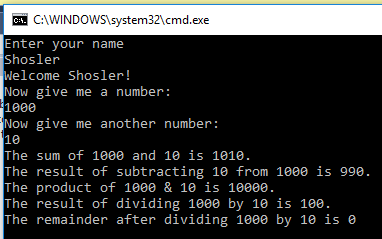
This type of expression has many uses, particularly where ***looping*** is concerned.

**TRY IT OUT – Manipulating Variables with Mathematical Operators**

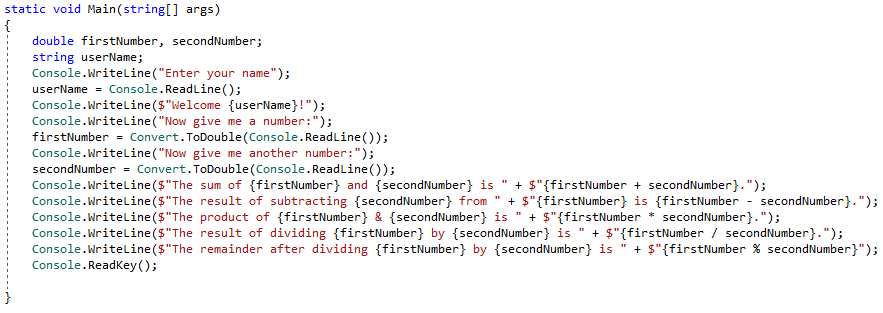
1. Create a new console application
2. Enter in the following code



1. Enter your name and provide a number (pressing the enter key to move on).
2. Result:



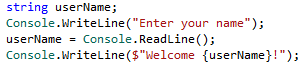
Can also write the code like this:



***How it works***  
This code introduces two important concepts:

* User input
* Type conversion

User input uses a syntax similar to the Console.WriteLine() command we have already seen – you use Console.ReadLine(). **This command prompts the user for input, which is stored in a string variable.**



The code writes the contents of the assigned variable, userName, straight to the screen.

You also read in two numbers in this example. This is slightly more involved, because the Console.ReadLine()command generates a string, but you want a number. Here we introduce the topic of ***type conversion.***

First, we declare the variables in which we want to store the number input:



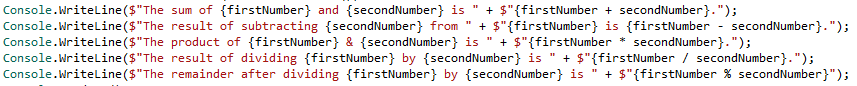
Next, we supply a prompt and use the command Convert.ToDouble()on a string obtained by Console.ReadLine()to convert the string into a double type. We then assign this number to the firstNumber variable:



The remainder of the code obtains a second number in the same way:



Next, we output the results of adding, subtracting, multiplying, dividing the two numbers, and the remainder after division.



We are supplying the expressions, firstNumber + secondNumber and so on, as a parameter to the Console.WriteLine() statement, without using an intermediate variable:

This kind of syntax makes your code very readable, and reduces the number of lines of code you need to write.

### Assignment Operators

So far, we have just been using the simple = assignment operator.

|  |  |  |  |
| --- | --- | --- | --- |
| **OPERATOR** | **CATEGORY** | **EXAMPLE EXPRESSION** | **RESULT** |
| = | Binary | var1 = var2; | **var1** is assigned the value of **var2** |
| += | Binary | var1 += var2; | **var1 i**s assigned the value that is the sum of **var1** and **var2**. |
| -= | Binary | var1 -= var2; | **var1** is assigned the value that is the value of **var2** subtracted from the value of **var1.** |
| \*= | Binary | var1 \*= var2; | **var1** is assigned the value that is the product of **var1** and **var2**. |
| /= | Binary | var1 /= var2; | **var1** is assigned the value that is the result of dividing **var1** and **var2**. |
| %= | Binary | var1 %= var2; | **var1** is assigned the value that is the remainder when **var1** is divided by **var2**. |