Variables

Introduction to variables

A variable is a name for a container in memory that can be used to store data. C# is a **strongly typed language** so that the data type of all variables must be declared before using the variable. Declaring a variable's type provides the compiler with the information it needs to figure out how the variable should be stored in memory.

The first step in using any variable is to **declare its type**. For example, to declare variable "n" as an integer, one uses the following command:

```
int n;
```

The "`int`" in the command indicates that the variable with name "n" is an integer and the ";" indicated the end of the declaration. The second step in the use of a variable is to **assign a value to it**. This would be done by a command such as:

```
n = 95;
```

The "=" assigns the integer "95" to the variable n.

The process of declaration and assignment can be done in one step through a command such as:

```
int n = 95;
```

Once a value is assigned to a variable, the variable can be used in calculations and assignment to another variable or to itself. For example, consider the following sequence of commands:

```
int n = 95;
int m = n+5;
n = m+n;
```

In this sequence, we have declared n and m as integers, assigned 95 to n, added five to n and assigned it to m (m=100), and then assigned m+n to n (n=100+95=195). As can be seen, one variable n can even be used to calculate a new integer that is to be assigned back to n.

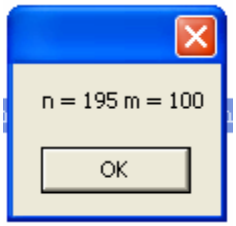
To try this out in practice, develop a windows application in Visual Studio and add a button to it. Double click on the button to get the method attached to the Click event of the button. Type the following code in this method

```
int n = 95;
int m = n+5;
n = m+n;
```



```
MessageBox.Show("n = "+n.ToString()+" m = "+m.ToString());
```

Build the application and run it. Press the button, you should get a box looking like



The box is generated by the "MessageBox.Show" method shown. Keep this application open since we will use the MessageBox and this application to try out variable declarations in the remainder of the chapter.

Numeric data types in C#

The following table summarizes the **numeric data types** in C#:

Data type name	Memory storage	Range	Suffix
sbyte	8-bit signed integer	-128 to 127	
short	16-bit signed integer	-32,768 to 32,767	
int	32-bit signed integer	-2,147,483,648 to 2,148,483,647	
long	64-bit signed integer	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	L or l
byte	8-bit unsigned integer	0 to 255	
ushort	16-bit unsigned integer	0 to 65,535	
uint	32-bit unsigned integer	0 to 4,294,967,295	U or u
ulong	64-bit unsigned integer	0 to 18,446,744,073,709,551,615	UL, LU
float	32-bit single precision floating point (7 significant figures)	$\pm 1.5\times10^{-45}$ to $\pm 3.4\times10^{38}$	F or f
double	64-bit double precision floating point (15/16 significant figures)	$\pm 5.0\times10^{-324}$ to $\pm 1.7\times10^{308}$	D or d
decimal	128-bit high precision decimal notation (28 significant figures)	$\pm 1.0\times10^{-28}$ to $\pm 7.9\times10^{28}$	M or m

The suffixes provided allow one to declare a number of the a specific type. This can be useful when there might be an ambiguity about the type of a number. For example, the number "128f" is a 32-bit single precision floating point and "128d" is a 64-bit double

precision floating point number. In most cases this is not necessary, since the compiler can figure out the type of a number by the context it is used in.

Text data types in C#

There are only two data types that can contain text. These are character and string types. The character type name is "**char**" which contains one character of text. The string type name is "**string**" and is used to store a variable length string of characters.

An example of declaring and assigning a value to a character would be:

```
char myChar;
myChar = 'p';
```

The second line assigns a number associated with the character "p" to myChar. Note the **single quotation marks** used in this assignment.

An example of declaring and assigning a value to a string variable would be:

```
string myString;
myString = "This is a string.";
```

The second line assigns the text "This is a string." to myString. Note the double quotation marks used around a string to define the start and end of the string.

As before, the declaration and assignment can be done in one step so that we could write:

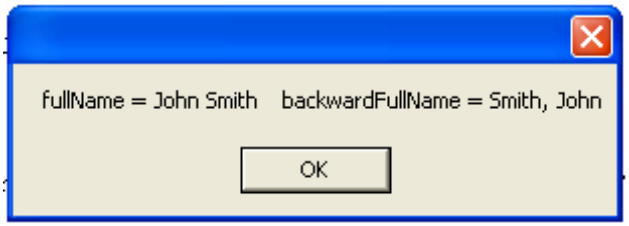
```
char myChar = 'p';
string myString = "This is a string.";
```

Strings can be added together to make longer strings. For example we could do the following to get the forward and backward full name of a person:

```
string firstName = "John";
string lastName = "Smith";
string fullName = firstName + " " + lastName;
string backwardFullName = lastName + ", "+ firstName;
```

The result of this would be that "John Smith" would be assigned to fullName and "Smith, John" would be assigned to vackwardFullName. If you add the MessageBox Command (all on one line) you should see the output that follows

```
MessageBox.Show("fullName = "+fullName.ToString()+"    backwardFullName
= "+backwardFullName.ToString());
```

An **escape sequence** is a command that allows us to provide some formatting in the string, such as starting a new line, or inserting a quotation, which would otherwise be confused with the ending of the string. A backslash ("\") is used to declare an escape sequence so it cannot be used in a string. If there is a need to use a backslash in a string, as is the case when you need to provide an address to a directory, then one must type a double backslash in such "\\" to get the single backslash. The escape sequences and their usage is given in the following table:

Escape sequence	Function
\'	Single quote
\"	Double quote
\\	Single backslash
\0	Null
\a	Alert
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Tab character
\v	Vertical tab

For example to assign the value "c:\CSClass\Notes" to myNotesAddress we need to type the command:

```
string myNotesAddress ="c:\\CSClass\\Notes";
```

Since the address declaration is so common, C# provides an alternate way to do this:

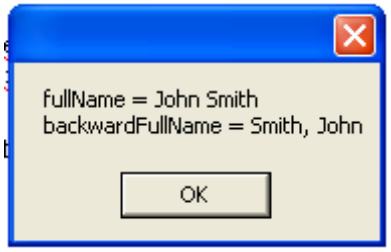
```
string myNotesAddress = @"c:\CSClass\Notes";
```

The "@" character when added at the start makes the "\" appear as they are typed without the need to use "\\" to get the single backslash.

To demonstrate the use of the escape sequence, in the fullName example of the last section add the new line escape sequence \n in the MessageBox code as follows:

```
MessageBox.Show("fullName = "+fullName.ToString()+"\nbackwardFullName =
"+backwardFullName.ToString());
```

The result of pressing your button will now look like:



Boolean type

The **Boolean type** variable only takes one of two values: true or false. The name "bool" is used to declare a Boolean variable. An example of the declaration and assignment of a Boolean would be:

```
bool myBoolean;
myBoolean = false;
```

As before these two can be done in one step as

```
bool myBoolean = false;
```

Object type

The **object type** is the base type for all data types in C#. As the base type, all data types can be represented by an object type. An object is declared with the command

```
object myObject;
```

Any data type can be assigned to the object type variable as in the following case:

```
bool myBoolean = false;
object myObject;
myObject = myBoolean;
```

In this case, to retrieve the Boolean back we need to **cast** the object to Boolean. This can be done for this case by the command

```
bool myOtherBoolean = (bool)myObject;
```

The "(bool)" tells the compiler that the object myObject is a Boolean so that it can be assigned to the Boolean variable myOtherBoolean. The process of assigning a data type to an object type and then retrieving it is known as **boxing and unboxing**.

Reference and Value type variables

A variable is either value type or reference type. The difference between the two is in how the variable is stored and how it is accessed and manipulated. There are two areas in memory, the **stack** and the **heap**. A value type data is entirely stored in the stack. Since the stack is designed to store fixed length data, all value type data must be of fixed length. For reference type data, only the reference to the location of the data in the heap is stored in the stack, while the data itself is stored in the heap. The structure of the reference variable type storage best suites variable length data types.

From the data types that we have studied, the numeric data types, the character data types and the Boolean data types are fixed length and are value type. On the other hand, strings and objects are of variable length and are reference type.

There is a big difference between the value and reference type variables in how they are assigned and manipulated. When a value type variable is assigned to another value type variable, the content of the first is copied into the second so you get two separate copies. In such a case, manipulation on one variable does not affect the other variable since each is distinct. On the other hand, if one reference type variable is assigned to another, only the reference to the location on the heap is stored for the new variable, without making a new copy of the variable on the heap. As such, whenever either is manipulated, since both point to the same memory on the heap, it is possible that changes to one will result in changes to the other (this is not always the case, as will be shown below). For example, consider the following commands:

```
int n = 3;
int m = n;
n = 4;
```

First n is assigned the value 3, then n is assigned to m, and, finally, n is assigned the value 4. Since the integer type is a value type, the end result is that m is assigned the value 3 and n is assigned the value 4. Note that the last assignment of “n = 4” dose not effect m since m and n are distinct (i.e., they each have their own memory in the stack). Now add the following commands to your application:

```
string myString1 = "Hi!";
string myString2 = myString1;
myString1 = "Bye!";

MessageBox.Show("myString1 = "+myString1+"\nmyString2 =
"+myString2);
```

The result will be

Since strings are reference type, when we assigned myString2 the value of myString1, we simply stored for myString2 a reference in the stack to the memory in the heap that contained myString1. So, we would expect that at the end of the process "Bye!" would be the value of both myString1 and myString2, but this is not the case since strings are manipulated in a different way. For strings, when you change one of two or more variables that point to the same location on the heap, a new memory location on the heap is made for the string that has its value changed. In the example, this feature mans that the value of myString2 does not get affected when we give myString1 a new value. As a result myString2 still retains the value "Hi!" even after myString1 has been assigned the value "Bye!".

If we do this same operation using an object, the result would be the same since both string and object are handled in the same way. **On the other hand more complex reference type objects such as classes and arrays are not handled in this way and care must be taken not to assume that when you assign one object to the other it will create two independent objects that can be manipulated independently**.

Data conversion

Not all data types can be converted to one and other. At a minimum, to convert one data type to another, the data type receiving the value needs to have sufficient memory to store the other data type. C# provides two types of data conversions: implicit and explicit. Implicit data conversions are those that the compiler makes based on the context of the operations or assignments. Explicit conversions are those which we explicitly ask the compiler to make.

An implicit conversion would be

Explicit conversions can be done through the **Convert command**. The C# compiler provides a large number of methods for such conversions. For example, to convert a value to int type we could write the following

```
string myString = "56";
int myInteger = Convert.ToInt32(myString);

MessageBox.Show("myInteger = "+myInteger.ToString());
```


The result is that the string value is converted to integer so that myInteger would be assigned the value 56. Note that “Int32” is the actual data type for “int” in C#.Visual Studio can be used to check all the available conversions. There is one method for each simple data type. Running your application would result in

Another method of conversion is using **casts**. One can cast a variable of one type into a variable of another type by a command such as

```
float myFloat = 56;
int myInteger = (int)myFloat;
```

On the right hand side of the second line we have cast the single precision floating point variable myFloat as a 32-bit integer. Obviously, if we had selected to cast the myFloat into a data type which was not compatible, there would have been a problem. Examine the following code

```
float myFloat = 56.6f;
int myInteger = (int)myFloat;

MessageBox.Show("myInteger = "+myInteger.ToString());
```

The result of this is as follows

As can be seen, still the value of 56 is assigned to myInteger since there is no way to convert 56.6 to integer. Also note that we needed to use “f” at the end of 56.6 to let the compiler know that we mean it to be single precision floating point. The compiler by default assumes that such numbers are double precision floating point, which cannot implicitly be converted to single precision do to the smaller storage size of the single precision.

Literals

Literals are specifically written out values. We have already used literals several times. The right-hand-side of all the following commands are literals.

```
float myFloat = 56.6f;
int myInteger = 16;
string myName = "John";
double myDouble = 106.0d;
```

