A Closer Look at Variables

Variables are declared using this form of statement:

*type var-name*;

where *type* is the data type of the variable and *var-name* is its name. You can declare a variable of any valid type, including the value types just described. It is important to understand that a variable’s capabilities are determined by its type. For example, a variable of type **bool** cannot be used to store floating-point values. Furthermore, the type of a variable cannot change during its lifetime. An **int** variable cannot turn into a **char** variable, for example.

All variables in C# must be declared prior to their use. This is necessary because the compiler must know what type of data a variable contains before it can properly compile any statement that uses the variable. It also enables C# to perform strict type-checking.

C# defines several different kinds of variables. The kind that we have been using are called *local variables* because they are declared within a method.

|  |
| --- |
| PART I |





50 P a r t I : T h e C # L a n g u a g e

Initializing a Variable

One way to give a variable a value is through an assignment statement, as you have already seen. Another way is by giving it an initial value when it is declared. To do this, follow the variable’s name with an equal sign and the value being assigned. The general form of initialization is shown here:

*type var-name* = *value*;

Here, *value* is the value that is given to the variable when it is created. The value must be compatible with the specified type.

Here are some examples:

int count = 10; // give count an initial value of 10 char ch = 'X'; // initialize ch with the letter X float f = 1.2F; // f is initialized with 1.2

When declaring two or more variables of the same type using a comma-separated list, you can give one or more of those variables an initial value. For example:

int a, b = 8, c = 19, d; // b and c have initializations

In this case, only **b** and **c** are initialized.

Dynamic Initialization

Although the preceding examples have used only constants as initializers, C# allows variables to be initialized dynamically, using any expression valid at the point at which the variable is declared. For example, here is a short program that computes the hypotenuse of a right triangle given the lengths of its two opposing sides.

* Demonstrate dynamic initialization. using System;

class DynInit {

static void Main() {

* Length of sides. double s1 = 4.0; double s2 = 5.0;
* Dynamically initialize hypot.

double hypot = Math.Sqrt( (s1 \* s1) + (s2 \* s2) );

Console.Write("Hypotenuse of triangle with sides " + s1 + " by " + s2 + " is ");

Console.WriteLine("{0:#.###}.", hypot);

}}

Here is the output:

Hypotenuse of triangle with sides 4 by 5 is 6.403.



C h a p t e r 3 : D a t a T y p e s , L i t e r a l s , a n d V a r i a b l e s 51

Here, three local variables—**s1**, **s2**, and **hypot**—are declared. The first two, **s1** and **s2**, are initialized by constants. However, **hypot** is initialized dynamically to the length of the hypotenuse. Notice that the initialization involves calling **Math.Sqrt( )**. As explained, you can use any expression that is valid at the point of the initialization. Since a call to **Math.Sqrt( )** (or any other library method) is valid at this point, it can be used in the initialization of **hypot**. The key point here is that the initialization expression can use any element validat the time of the initialization, including calls to methods, other variables, or literals.

Implicitly Typed Variables

As explained, in C# all variables must be declared. Normally, a declaration includes the type of the variable, such as **int** or **bool**, followed by the name of the variable. However, beginning with C# 3.0, it is possible to let the compiler determine the type of a local variable based on the value used to initialize it. This is called an *implicitly typed variable.*

An implicitly typed variable is declared using the keyword **var**, and it must be initialized. The compiler uses the type of the initializer to determine the type of the variable. Here is an example:

var e = 2.7183;

Because **e** is initialized with a floating-point literal (whose type is **double** by default), the type of **e** is **double**. Had **e** been declared like this:

var e = 2.7183F;

then **e** would have the type **float**, instead.

The following program demonstrates implicitly typed variables. It reworks the program shown in the preceding section so that all variables are implicitly typed.

* Demonstrate implicitly typed variables. using System;

class ImplicitlyTypedVar {

static void Main() {

* These are now implicitly typed variables. They
* are of type double because their initializing
* expressions are of type double.

var s1 = 4.0;

var s2 = 5.0;

* Now, hypot is implicitly typed. Its type is double
* because the return type of Sqrt() is double.

var hypot = Math.Sqrt( (s1 \* s1) + (s2 \* s2) );

Console.Write("Hypotenuse of triangle with sides " + s1 + " by " + s2 + " is ");

Console.WriteLine("{0:#.###}.", hypot);

* The following statement will not compile because
* s1 is a double and cannot be assigned a decimal value.

|  |
| --- |
| PART I |





52 P a r t I : T h e C # L a n g u a g e

* s1 = 12.2M; // Error!

}

}

The output is the same as before.

It is important to emphasize that an implicitly typed variable is still a strongly typed variable. Notice this commented-out line in the program:

* s1 = 12.2M; // Error!

This assignment is invalid because **s1** is of type **double**. Thus, it cannot be assigned a **decimal** value. The only difference between an implicitly typed variable and a “normal”explicitly typed variable is how the type is determined. Once that type has been determined, the variable has a type, and this type is fixed throughout the lifetime of the variable. Thus, the type of **s1** cannot be changed during execution of the program.

Implicitly typed variables were not added to C# to replace “normal” variable declarations. Instead, implicitly typed variables are designed to handle some special-case situations, the most important of which relate to Language-Integrated Query (LINQ), which is described in Chapter 19. Therefore, for most variable declarations, you should continue to use explicitly typed variables because they make your code easier to read and easier to understand.

One last point: Only one implicitly typed variable can be declared at any one time. Therefore, the following declaration,

var s1 = 4.0, s2 = 5.0; // Error!

is wrong and won’t compile because it attempts to declare both **s1** and **s2** at the same time.

The Scope and Lifetime of Variables

So far, all of the variables that we have been using are declared at the start of the **Main( )** method. However, C# allows a local variable to be declared within any block. As explained in Chapter 1, a block begins with an opening curly brace and ends with a closing curly brace. A block defines a *scope.* Thus, each time you start a new block, you are creating a new scope. A scope determines what names are visible to other parts of your program without qualification. It also determines the lifetime of local variables.

The most important scopes in C# are those defined by a class and those defined by a method. A discussion of class scope (and variables declared within it) is deferred until later in this book, when classes are described. For now, we will examine only the scopes defined by or within a method.

The scope defined by a method begins with its opening curly brace and ends with its closing curly brace. However, if that method has parameters, they too are included within the scope defined by the method.

As a general rule, local variables declared inside a scope are not visible to code that is defined outside that scope. Thus, when you declare a variable within a scope, you are protecting it from access or modification from outside the scope. Indeed, the scope rules provide the foundation for encapsulation.

Scopes can be nested. For example, each time you create a block of code, you are creating a new, nested scope. When this occurs, the outer scope encloses the inner scope. This means that local variables declared in the outer scope will be visible to code within the inner scope.

C h a p t e r 3 : D a t a T y p e s , L i t e r a l s , a n d V a r i a b l e s 53

However, the reverse is not true. Local variables declared within the inner scope will not be visible outside it.

To understand the effect of nested scopes, consider the following program:

* Demonstrate block scope. using System;

class ScopeDemo {

static void Main() {

int x; // known to all code within Main()

x = 10;

if(x == 10) { // start new scope

int y = 20; // known only to this block

* + x and y both known here. Console.WriteLine("x and y: " + x + " " + y); x = y \* 2;

}

* y = 100; // Error! y not known here.
* x is still known here. Console.WriteLine("x is " + x);

}

}

As the comments indicate, the variable **x** is declared at the start of **Main( )**’s scope and is accessible to all subsequent code within **Main( )**. Within the **if** block, **y** is declared. Since a block defines a scope, **y** is visible only to other code within its block. This is why outside of its block, the line **y = 100;** is commented out. If you remove the leading comment symbol, a compile-time error will occur because **y** is not visible outside of its block. Within the **if** block, **x** can be used because code within a block (that is, a nested scope) has access to variables declared by an enclosing scope.

Within a block, variables can be declared at any point, but are valid only after they are declared. Thus, if you define a variable at the start of a method, it is available to all of the code within that method. Conversely, if you declare a variable at the end of a block, it is effectively useless, because no code will have access to it.

If a variable declaration includes an initializer, then that variable will be reinitialized each time the block in which it is declared is entered. For example, consider this program:

* Demonstrate lifetime of a variable. using System;

class VarInitDemo {

static void Main() {

int x;

for(x = 0; x < 3; x++) {

int y = -1; // y is initialized each time block is entered Console.WriteLine("y is: " + y); // this always prints -1

|  |
| --- |
| PART I |





54 P a r t I : T h e C # L a n g u a g e

y = 100;

Console.WriteLine("y is now: " + y);

}

}

}

The output generated by this program is shown here:

y is: -1

y is now: 100

y is: -1

y is now: 100

y is: -1

y is now: 100

As you can see, **y** is always reinitialized to –1 each time the inner **for** loop is entered. Even though it is subsequently assigned the value 100, this value is lost.

There is one quirk to C#’s scope rules that may surprise you: Although blocks can be nested, no variable declared within an inner scope can have the same name as a variable declared by an enclosing scope. For example, the following program, which tries to declare two separate variables with the same name, will not compile.

/\*

This program attempts to declare a variable in an inner scope with the same name as one defined in an outer scope.

\*\*\* This program will not compile. \*\*\*

\*/

using System;

class NestVar {

static void Main() {

int count;

for(count = 0; count < 10; count = count+1) { Console.WriteLine("This is count: " + count);

int count; // illegal!!!

for(count = 0; count < 2; count++) Console.WriteLine("This program is in error!");

}

}

}

If you come from a C/C++ background, then you know that there is no restriction on the names you give variables declared in an inner scope. Thus, in C/C++ the declaration of **count** within the block of the outer **for** loop is completely valid. However, in C/C++, sucha declaration hides the outer variable. The designers of C# felt that this type of *name hiding* could easily lead to programming errors and disallowed it.