### Variables

In this chapter you look at a subject that is core to handling data of any kind in programming — variables. Variables are the cornerstone of handling and passing data in C# and other programming languages. Whenever there’s any data being handled or processed, variables are never far away!

##### What are Variables?

There are a number of ways to describe what a variable is. In cold computer terms, a variable is a storage location for data. Rather than having to mess around addressing memory locations directly (something that can lead even the best programmer into the tar pits), variables are referenced by the name attached to them.

You can think of variables as storage boxes in memory. Each box is given a name and can hold specific kinds of data, called values.

Every variable has a type. This type determines what values can be stored in the variable. C# is a type-safe language, and the compiler ensures that values stored in a variable are of the right type.

Not all Variables Are Created Equally

Not all variables are created in the same way. In fact, two kinds of variables can be created:

* + **Initially assigned.** Here are a few simple examples:

int myInt = 3;

string myString = “Hello”; char myChar = “x”;

* **Initially unassigned.** Here are a few simple examples:

int myInt; string myString; char myChar;

The difference between an initially assigned and an initially unassigned variable is that when an initially unassigned variable is created, it is created without an initial value, whereas an initially assigned vari- able has a well-defined initial value.

*A value has to be assigned to a variable before a value can be obtained from it (more on this later in this chapter).*

##### Categories of Variables

There are seven distinct categories of variables:

* Static variables
* Instance variables
* Array elements
* Value parameters
* Reference parameters
* Output parameters
* Local variables

All these variables will be discussed over the course of this chapter. All seven types of variables are shown in the following code snippet:

class VarEx

{

public static int StaticVar; int InstanceVar;

void F(int[] ArrayEl,

int ValueParam, ref int RefParam,

out int OutputParam) { int LocalVar = 1;

OutputVar = ValueParam + RefParam++;

}

}

Below is a list of the variable names used, along with the type of variable each name represents:

* + StaticVar — This is a static variable.
  + ArrayEl — This is an array element.
  + InstanceVar — This is an instance variable.
  + ValueParam — This is a value parameter.
  + RefParam — This is a reference parameter.
  + OutputParam — This is an output parameter.
  + LocalVar — This is a local variable.

Let’s take a look at each of these variable categories in turn.

Static Variables

Static variables are initially assigned variables.

Any field declared with a static modifier is called a static variable.

These variables come into being before the execution of a static constructor for the containing type. The variable disappears when the application domain it is associated with no longer exists.

The initial value of the static variable is the default value of the type of the variable.

class VarEx

{

public static int StaticVar; int InstanceVar;

void F(int[] ArrayEl,

int ValueParam, ref int RefParam,

out int OutputParam) { int LocalVar = 1;

OutputVar = ValueParam + RefParam++;

}

}

Array Elements

Array elements are initially assigned.

The elements of an array appear when the array instance is created and disappears when there is no longer any reference to that array instance.

The initial value of each array element is the default value of the type of the element.

class VarEx

{

public static int StaticVar; int InstanceVar;

void F(int[] ArrayEl, int ValueParam,

ref int RefParam,

out int OutputParam) { int LocalVar = 1;

OutputVar = ValueParam + RefParam++;

}

}

Instance Variables

Any field declared without the static modifier is known as an instance variable. Instance variables can be used in the following:

* Classes
* Structs

class VarEx

{

public static int StaticVar;

int InstanceVar;

void F(int[] ArrayEl,

int ValueParam, ref int RefParam,

out int OutputParam) { int LocalVar = 1;

OutputVar = ValueParam + RefParam++;

}

}

Using Instance Variables in Classes

Instance variables used in classes are initially assigned variables.

An instance variable of a class comes into being when a new instance of that class is created. The vari- able disappears when there are no longer any references to that instance (and any finalizers executed).

The initial value of any instance variable of a class is the default value of the variable type.

Using Instance Variables in Structs

Instance variables used in structs are initially assigned variables if the struct variable is assigned and are unassigned if the struct variable is unassigned.

Instance variables of structs have the same lifecycle as that of the struct itself. That is, they are created when the struct is created and disappear when the struct ends.

Value Parameter

Value parameters are initially assigned.

A value parameter is declared without a ref or out modifier.

The lifecycle of a value parameter starts when the function member (instance constructor, accessor, method, or operator) to which the parameter belongs is invoked. Value parameters are initialized with the value of the argument given during invocation.

Value parameters end on return of the function member (except where the parameter is captured by an anonymous method or the function member body is an iterator block).

class VarEx

{

public static int StaticVar; int InstanceVar;

void F(int[] ArrayEl,

int ValueParam, ref int RefParam,

out int OutputParam) { int LocalVar = 1;

OutputVar = ValueParam + RefParam++;

}

}

Reference Parameters

When within function members, reference parameters are initially assigned.

A parameter that has been declared with a ref modifier is called a reference parameter.

It is important to note that reference parameters don’t themselves create new storage locations in mem- ory. Instead, they are a representation of an existing storage location. This means that the value of a ref- erence parameter is always the same as that of the underlying variable.

class VarEx

{

public static int StaticVar; int InstanceVar;

void F(int[] ArrayEl,

int ValueParam,

ref int RefParam, out int OutputParam) {

int LocalVar = 1;

OutputVar = ValueParam + RefParam++;

}

}

Output Parameters

A parameter declared with an out modifier is called an output parameter.

As with reference parameters, output parameters do not create any new storage locations on memory. Output parameters reference the same storage location as the variable given as the argument in the func- tion member invocation.

Definite assignment rules are applicable to output parameters:

* No variable needs to be definitely assigned before it can be passed as an output parameter in a member invocation function.
* Within a function member, output parameters are initially unassigned.
* Output parameters of a function member have to be definitely assigned before the function member returns normally.

class VarEx

{

public static int StaticVar; int InstanceVar;

void F(int[] ArrayEl,

int ValueParam, ref int RefParam,

out int OutputParam) { int LocalVar = 1;

OutputVar = ValueParam + RefParam++;

}

}

Local Variables

Local variables are declared by:

* local-variable-declaration — The variable will be initially assigned.
* foreach-statement — Here the local variable is an exception variable.
  + specific-catch-clause of a try-statement — The variable will be initially assigned.

class VarEx

{

public static int StaticVar; int InstanceVar;

void F(int[] ArrayEl,

int ValueParam, ref int RefParam,

out int OutputParam) {

int LocalVar = 1; OutputVar = ValueParam + RefParam++;

}

}

##### Default Values

Variables belonging to the following categories are initialized to their default values automatically:

* + Static variables
  + Instance variables (of class instances)
  + Array elements

The default value of a variable depends on the type of the variable:

* + For a variable of a value-type, the default value will be the same as the value computed by the

value-type’s default constructor.

* + For reference-type, the default value is null.

##### Definite Assignment

If the compiler can prove that a variable has been automatically initialized or has been the target of one or more assignment, that variable is said to be definitely assigned.

There are a handful of rules for definite assignment:

* + Initially assigned variables are always considered to be definitely assigned.
  + Initially unassigned variables are considered to be definitely assigned if all the execution paths contain one of the following:
    - An invocation expression that passes the variable as an output parameter
    - An object-creation expression that passes the variable as an output parameter
* A simple assignment where the variable is a left operand
* A local variable declaration that includes a variable initializer (local variables only) Separate rules apply to struct-type variables and their instance variables:
* An instance variable is definitely assigned if the containing struct-type variable is definitely assigned.
* A struct-type variable is definitely assigned if each of the instance variables is also definitely assigned.

Initially Assigned Variables

The following variable categories are classified as initially assigned:

* Static variables
* Array elements
* Value parameters
* Reference parameters
* Instance variables of class instances
* Instance variables of initially assigned struct variables
* Variables declared by:
  + A using statement
  + A foreach statement
  + A catch clause

Initially Unassigned Variables

The following variable categories are initially unassigned:

* Instance variables of initially unassigned struct variables
* Local variables (except those declared in a foreach statement, a catch clause, or a using

statement)

* Output parameters

Rules for Determining Definite Assignment

The compiler uses specific rules to check whether a variable is definitely assigned or not.

To check, the compiler processes the body of each function that contains one (or more) unassigned vari- ables. For each such variable (v) encountered, the compiler defines the assignment state for the variable at the following spots:

* At the beginning of every statement
* At the end of every statement
  + A the point where control is transferred to another statement
  + At the beginning of every expressions
  + At the end of every expression

What follows are rules that control how the state of a variable is determined.

***General Rules for Statements***

* + v is not definitely assigned at the start of a function member body.
  + v is definitely assigned at the start of an unreachable statement.
  + The definite assignment state of v at the start of any other statement can be determined by checking the definite assignment state of v on all control-flow transfers that target the beginning of that statement.
  + The definite assignment state of v at the end of a block (checked, unchecked, if, while, do, for, foreach, lock, using, or switch statement) is determined by the compiler by checking the definite assignment state of v on all control-flow transfers that target the end of that statement.

***Rules for Block Statements, Checked, and Unchecked Statements***

* + The definite assignment state of v on the control transfer to the first statement of the statement list in the block will be the same as the definite assignment statement of v before the block, checked, or unchecked statement.

***Rules for Expression Statements***

The following rules apply for an expression statement stmt that consists of the expression expr:

* + v has the same assignment state at the beginning of expr as it does at the beginning of stmt.
  + When v is definitely assigned at the end of expr, it is definitely assigned at the end point of stmt.

***Rules for Declaration Statements***

* + If stmt is a declaration statement that does not have initializers, v will have the same definite assignment state at the end point of stmt as at the beginning of stmt.
  + If stmt is a declaration statement that does have initializers, the definite assignment state for v is determined as if stmt were a statement list, with one assignment statement for each declara- tion with an initializer.

***Rules for If Statements***

Let’s take a look at an if statement called stmt with the following form:

if ( expr ) then-stmt else else-stmt

* + v has the same definite assignment state at the beginning of expr as at the beginning of stmt.
  + If v is definitely assigned at the end of expr, it is also definitely assigned during the control-flow transfer to then-stmt and to either else-stmt or to the end of stmt if there is no else clause.
* If v is definitely assigned after an expression that returns a true at the end of expr, it is defi- nitely assigned during the control-flow transfer to then-stmt and not definitely assigned on the control-flow transfer to either else-stmt or to the end of stmt if there is no else clause.
* If v is definitely assigned after an expression that returns a false at the end of expr, it is defi- nitely assigned on the control-flow transfer to else-stmt and not definitely assigned on the control-flow transfer to then-stmt. It is definitely assigned at the end of stmt if and only if it is definitely assigned at the end-point of then-stmt.
* If none of the rules apply, v is not definitely assigned on the control-flow transfer to either the

then-stmt or else-stmt or to the end of stmt in the event that there is no else clause.

***Rules for Switch Statements***

In a switch statement, stmt that has the controlling expression expr:

* The definite assignment state of v at the beginning of expr is the same as the state of v at the beginning of stmt.
* The definite assignment state of v at control flow transfer to a switch block statement list is the same as the definite assignment state of v at the end of expr.

***Rules for While Statements***

Let’s take a while statement stmt of the form:

while ( expr ) while-body

* v has the same definite assignment state at the beginning of expr as it does at the beginning of

stmt.

* If v is definitely assigned at the end of expr, it is definitely assigned on the control-flow transfer to while-body and until the end of stmt.
* If v is definitely assigned after an expression that returns a true at the end of expr, it is defi- nitely assigned at the point of control-flow transfer to while-body but not definitely assigned at the end of stmt.
* If v is definitely assigned after an expression that returns a false at the end of expr, it is also def- initely assigned at the point of control-flow transfer to the end point of stmt but not definitely assigned on the control-flow transfer to while-body.

***Rules for Do Statements***

Let’s take a do statement stmt of the form:

do do-body while ( expr ) ;

* v has the same definite assignment state on the control-flow transfer from the beginning of stmt

to do-body as at the beginning of stmt.

* v has the same definite assignment state at the beginning of expr as it does at the end of do-body.
* If v is definitely assigned at the end of expr, it is definitely assigned on control-flow transfer to the end point of stmt.
  + If v is definitely assigned after an expression that returns a false at the end of expr, it is also definitely assigned on the control-flow transfer to the end point of stmt but is not definitely assigned on the control-flow transfer to do-body.

***Rules for Break, Continue, and Goto Statements***

* + The definite assignment state of v on the control-flow transfer caused by a break, continue, or

goto statement is the same as the definite assignment state of v at the beginning of the statement.

***Rules for Throw Statements***

Take a statement stmt of the form:

throw expr ;

* + The definite assignment state of v at the beginning of expr is the same as the definite assign- ment state of v at the beginning of stmt.

***Rules for Return Statements***

The rules for return statements depend on the form that the statement takes: For a statement stmt of the form:

return expr ;

* + The definite assignment state of v at the beginning of expr is the same as the definite assignment state of v at the beginning of stmt.
  + If v is an output parameter, it will be definitely assigned either:
    - After expr
    - At the end of the finally block of a try-finally or try-catch-finally that encloses the

return statement

If the statement stmt has the following form:

return ;

* + If v is an output parameter, it will be definitely assigned either:
    - Before stmt
    - At the end of the finally block of a try-finally or try-catch-finally that encloses the return statement

***Rules for Try-Catch Statements***

For a try-catch statement stmt of the form:

try try-block

catch ( ... ) catch-block-1

...

catch ( ... ) catch-block-n 93

* The definite assignment state of v at the beginning of try-block will be the same as the definite assignment state of v at the beginning of stmt.
* The definite assignment state of v at the beginning of catch-block-i is the same as the defi- nite assignment state of v at the beginning of stmt.
* The definite assignment state of v at the end-point of stmt is definitely assigned if v is definitely assigned at the end of try-block and every catch-block-i.

***Rules for Try-Finally Statements***

Let’s examine a try statement stmt of the form:

try try-block finally finally-block

* The definite assignment state of v at the beginning of try-block is the same as the definite assignment state of v at the beginning of stmt.
* The definite assignment state of v at the beginning of finally-block is the same as the defi- nite assignment state of v at the beginning of stmt.
* The definite assignment state of v at the end of stmt is definitely assigned if either:
  + v is definitely assigned at the end-point of try-block.
  + v is definitely assigned at the end-point of finally-block.

***Rules for Foreach Statements***

Let’s look at a foreach statement stmt of the form:

foreach ( type identifier in expr ) embedded-statement

* The definite assignment state of v at the beginning of expr is the same as the state of v at the beginning of stmt.
* The definite assignment state of v on the control-flow transfer to embedded-statement or to the end point of stmt will be the same as the state of v at the end of expr.

***Rules for Using Statements***

Let’s next take a look at a using statement stmt of the form:

using ( resource-acquisition ) embedded-statement

* The definite assignment state of v at the beginning of resource-acquisition is the same as the state of v at the beginning of stmt.
* The definite assignment state of v during the control-flow transfer to embedded-statement is the same as the state of v at the end of resource-acquisition.

***Rules for Lock Statements***

Next, a lock statement stmt of the form:

lock ( expr ) embedded-statement

* + The definite assignment state of v at the beginning of expr will be the same as the state of v at the beginning of stmt.
  + The definite assignment state of v during the control-flow transfer to embedded-statement is the same as the state of v at the end of expr.

***Rules for Simple Expressions***

The rules regarding simple expressions apply to the following expressions:

* + Literals
  + Simple names
  + Member access expressions
  + Nonindexed base access expressions
  + Typeof expressions

The definite assignment state of v at the end of the expression is the same as the definite assignment state of v at the beginning of the expression

The following rules:

* + The definite assignment state of v at the beginning of expr1 is the same as the definite assign- ment state at the beginning of expr.
  + The definite assignment state of v at the beginning of expri (where i is greater than one) is the same as the definite assignment state at the end of expri-1.
  + The definite assignment state of v at the end of expr is the same as the definite assignment state at the end of exprn.

Apply to these expressions:

* + Parenthesized expressions
  + Element access expressions
  + Base access expressions (with indexing)
  + Increment expressions
  + Decrement expressions
  + Cast expressions
  + unary +
  + -
  + ~
  + \* expressions
  + binary +
  + -
  + \*
* /
* %
* <<
* >>
* <
* <=
* >
* >=

❑ ==

* !=
* is
* as
* &
* |
* ^ expressions
* Compound assignment expressions
* Checked expressions
* Unchecked expressions
* Array
* Delegate creation expressions

***Rules for && Expressions***

Next, we’ll look at an expression expr of the form:

expr-first && expr-second

* The definite assignment state of v before expr-first will be the same as the definite assign- ment state of v before expr.
* The definite assignment state of v before expr-second will be definitely assigned if the state of v after expr-first is either definitely assigned or definitely assigned after a true expression. Otherwise, it will not be definitely assigned.
* The definite assignment state of v after expr is determined by:
  + If the state of v after expr-first is definitely assigned, the state of v after expr is also definitely assigned.
  + Otherwise, if the state of v after expr-second is definitely assigned and the state of v after expr-first is definitely assigned after false expression, the state of v after expr is defi- nitely assigned.
  + Otherwise, if the state of v after expr-second is definitely assigned or definitely assigned after a true expression, the state of v after expr is definitely assigned after true expression.
    - Otherwise, if the state of v after expr-first is definitely assigned after false expression and the state of v after expr-second is definitely assigned after false expression, the state of v after expr is definitely assigned after a false expression.
    - Otherwise, the state of v after expr is not definitely assigned.

***Rules for || Expressions***

Next, we’ll look at an expression expr of the form:

expr-first || expr-second

* The definite assignment state of v before expr-first will be the same as the definite assign- ment state of v before expr.
* The definite assignment state of v before expr-second will be definitely assigned if the state of v after expr-first is either definitely assigned or definitely assigned after a false expression. Otherwise, it will not be definitely assigned.
* The definite assignment state of v after expr is determined by:
  + If the state of v after expr-first is definitely assigned, the state of v after expr is also definitely assigned.
  + Otherwise, if the state of v after expr-second is definitely assigned and the state of v after expr-first is definitely assigned after a false expression, the state of v after expr is defi- nitely assigned.
  + Otherwise, if the state of v after expr-second is definitely assigned or definitely assigned after true expression, the state of v after expr is definitely assigned after a false expression.
  + Otherwise, if the state of v after expr-first is definitely assigned after a true expression and the state of v after expr-second is definitely assigned after a true expression, the state of v after expr is definitely assigned after a false expression.
  + Otherwise, the state of v after expr is not definitely assigned.

***Rules for ! Expressions***

For an expression expr of the form:

! expr-operand

* The definite assignment state of v before expr-operand is identical to the definite assignment state of v before expr.
* The definite assignment state of v after expr is determined by:
  + If the state of v after expr-operand is definitely assigned, the state of v after expr is definitely assigned.
  + If the state of v after expr-operand is not definitely assigned, the state of v after expr is also not definitely assigned.
  + If the state of v after expr-operand is definitely assigned after a false expression, the state of v after expr is definitely assigned after a true expression.
  + If the state of v after expr-operand is definitely assigned after a true expression, the state of v after expr is definitely assigned after a false expression.

***Rules for ?: Expressions***

For an expression expr of the form:

expr-cond ? expr-true : expr-false

* The definite assignment state of v before expr-cond will be the same as the state of v before expr.
* The definite assignment state of v before expr-true is definitely assigned if the state of v after

expr-cond is definitely assigned or definitely assigned after a true expression.

* The definite assignment state of v before expr-false is definitely assigned if the state of v after

expr-cond is definitely assigned or definitely assigned after a false expression.

* The definite assignment state of v after expr is determined by:
  + If expr-cond is a constant expression with a value true, the state of v after expr is the same as the state of v after expr-true.
  + Otherwise, if expr-cond is a constant expression with a value false, the state of v after

expr is the same as the state of v after expr-false.

* + Otherwise, if the state of v after expr-true is definitely assigned and the state of v after

expr-false is definitely assigned, the state of v after expr is definitely assigned.

* + Otherwise, the state of v after expr is not definitely assigned.

***Rules for Yield Statements***

Finally, let’s take a look at a yield return statement stmt of the form:

yield return expr ;

* A variable v has the same definite assignment state at the beginning of expr as at the beginning of stmt.
* If a variable v is definitely assigned at the end of expr, it is definitely assigned at the end of

stmt. Otherwise, it is not definitely assigned at the end of stmt.

##### Summary

In this chapter you looked at one of the most important elements related to programming — variables. You learned about assigned and unassigned variables, along with the seven categories of variables.

After that you examined default values and definite assignment before looking in detail at the rules for definite assignment.

In Chapter 8, you look at conversions in C#.