Data Types:

Why Data Types Are Important

Data types are especially important in C# because it is a strongly typed language. This means that all operations are type-checked by the compiler for type compatibility. Illegal operations will not be compiled. Thus, strong type-checking helps prevent errors and enhances reliability. To enable strong type-checking, all variables, expressions, and values have a type. There is no concept of a “typeless” variable, for example. Furthermore, a value’s type determines what operations are allowed on it. An operation allowed on one type might not be allowed on another.

C#’s Value Types

C# contains two general categories of built-in data types: *value types* and *reference types.* The difference between the two types is what a variable contains. For a value type, a variable holds an actual value, such 3.1416 or 212. For a reference type, a variable holds a reference to the value. The most commonly used reference type is the class, and a discussion of classes and reference types is deferred until later in this book. The value types are described here.

At the core of C# are the 13 value types shown in Table 3-1. Collectively, these are referred to as the *simple types.* They are called simple types because they consist of a single value. (In other words, they are not a composite of two or more values.) They form the foundation of C#’s type system, providing the basic, low-level data elements upon which a program operates. The simple types are also sometimes referred to as *primitive types.*

35



|  |  |  |  |
| --- | --- | --- | --- |
| 36 | P a r t I : | T h e C # L a n g u a g e | |
|  |  |  |  |
|  | Type |  | Meaning |
|  |  |  |  |
|  | bool |  | Represents true/false values |
|  |  |  |  |
|  | byte |  | 8-bit unsigned integer |
|  |  |  |  |
|  | char |  | Character |
|  |  |  |  |
|  | decimal |  | Numeric type for financial calculations |
|  |  |  |  |
|  | double |  | Double-precision floating point |
|  |  |  |  |
|  | float |  | Single-precision floating point |
|  |  |  |  |
|  | int |  | Integer |
|  |  |  |  |
|  | long |  | Long integer |
|  |  |  |  |
|  | sbyte |  | 8-bit signed integer |
|  |  |  |  |
|  | short |  | Short integer |
|  |  |  |  |
|  | uint |  | An unsigned integer |
|  |  |  |  |
|  | ulong |  | An unsigned long integer |
|  |  |  |  |
|  | ushort |  | An unsigned short integer |
|  |  |  |  |



TABLE 3-1 The C# Value Types

C# strictly specifies a range and behavior for each value type. Because of portability requirements, C# is uncompromising on this account. For example, an **int** is the same in all execution environments. There is no need to rewrite code to fit a specific platform. Although strictly specifying the size of the value types may cause a small loss of performance in some environments, it is necessary in order to achieve portability.

***NOTE*** *In addition to the simple types, C# defines three other categories of value types. These areenumerations, structures, and nullable types, all of which are described later in this book.*

Integers

C# defines nine integer types: **char**, **byte**, **sbyte**, **short**, **ushort**, **int**, **uint**, **long**, and **ulong**. However, the **char** type is primarily used for representing characters, and it is discussed later in this chapter. The remaining eight integer types are used for numeric calculations. Their bit-width and ranges are shown here:

|  |  |  |
| --- | --- | --- |
| Type | Width in Bits | Range |
| byte | 8 | 0 to 255 |
| sbyte | 8 | –128 to 127 |
| short | 16 | –32,768 to 32,767 |
| ushort | 16 | 0 to 65,535 |
| int | 32 | –2,147,483,648 to 2,147,483,647 |
| uint | 32 | 0 to 4,294,967,295 |
| long | 64 | –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| ulong | 64 | 0 to 18,446,744,073,709,551,615 |
|  |  |  |



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 37

As the table shows, C# defines both signed and unsigned versions of the various integer types. The difference between signed and unsigned integers is in the way the high-order bit of the integer is interpreted. If a signed integer is specified, then the C# compiler will generate code that assumes the high-order bit of an integer is to be used as a *sign flag.* If the sign flag is 0, then the number is positive; if it is 1, then the number is negative. Negative numbers are almost always represented using the *two’s complement* approach. In this method, all bits in the negative number are reversed, and then 1 is added to this number.

Signed integers are important for a great many algorithms, but they have only half the absolute magnitude of their unsigned relatives. For example, as a **short**, here is 32,767:

0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

For a signed value, if the high-order bit were set to 1, the number would then be interpreted as –1 (assuming the two’s complement format). However, if you declared this to be a**ushort**, then when the high-order bit was set to 1, the number would become 65,535.

Probably the most commonly used integer type is **int**. Variables of type **int** are often employed to control loops, to index arrays, and for general-purpose integer math. When you need an integer that has a range greater than **int**, you have many options. If the value you want to store is unsigned, you can use **uint**. For large signed values, use **long**. For large unsigned values, use **ulong**. For example, here is a program that computes the distance from the Earth to the sun, in inches. Because this value is so large, the program uses a **long** variable to hold it.

* Compute the distance from the Earth to the sun, in inches. using System;

class Inches {

static void Main() {

long inches;

long miles;

miles = 93000000; // 93,000,000 miles to the sun

* 5,280 feet in a mile, 12 inches in a foot. inches = miles \* 5280 \* 12;

Console.WriteLine("Distance to the sun: " + inches + " inches.");

}

}

Here is the output from the program:

Distance to the sun: 5892480000000 inches.

Clearly, the result could not have been held in an **int** or **uint** variable.

The smallest integer types are **byte** and **sbyte**. The **byte** type is an unsigned value between 0 and 255. Variables of type **byte** are especially useful when working with raw binary data, such as a byte stream produced by some device. For small signed integers, use **sbyte**. Here is an example that uses a variable of type **byte** to control a **for** loop that produces the summation of the number 100:

|  |
| --- |
| PART I |



// Use byte.



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

using System;

classUse\_byte {

static void Main() {

byte x;

int sum;

|  |  |  |
| --- | --- | --- |
| sum = | 0; | |
| for(x | = | 1; x <= 100; x++) |
| sum | = | sum + x; |

Console.WriteLine("Summation of 100 is " + sum);

}

}

The output from the program is shown here:

Summation of 100 is 5050

Since the **for** loop runs only from 0 to 100, which is well within the range of a **byte**, there is no need to use a larger type variable to control it.

When you need an integer that is larger than a **byte** or **sbyte**, but smaller than an **int** or **uint**, use **short** or**ushort**.

Floating-Point Types

The floating-point types can represent numbers that have fractional components. There are two kinds of floating-point types, **float** and **double**, which represent single- and double-precision numbers, respectively. The type **float** is 32 bits wide and has an approximate range of 1.5E–45 to 3.4E+38. The **double** type is 64 bits wide and has an approximate range of 5E–324 to 1.7E+308.

Of the two, **double** is the most commonly used. One reason for this is that many of the math functions in C#’s class library (which is the .NET Framework library) use **double** values. For example, the **Sqrt( )** method (which is defined by the library class **System.Math**) returns a **double** value that is the square root of its **double** argument. Here, **Sqrt( )** is used to compute the radius of a circle given the circle’s area:

* Find the radius of a circle given its area. using System;

classFindRadius {

static void Main() {

Double r;

Double area;

area = 10.0;

r = Math.Sqrt(area / 3.1416);



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 39

Console.WriteLine("Radius is " + r);

}

}

The output from the program is shown here:

Radius is 1.78412203012729

One other point about the preceding example. As mentioned, **Sqrt( )** is a member of the **Math** class. Notice how**Sqrt( )** is called; it is preceded by the name **Math**. This is similar tothe way **Console** precedes **WriteLine( )**. Although not all standard methods are called by specifying their class name first, several are, as the next example shows.

The following program demonstrates several of C#’s trigonometric functions, which are also part of C#’s math library. They also operate on **double** data. The program displays the sine, cosine, and tangent for the angles (measured in radians) from 0.1 to 1.0.

* Demonstrate Math.Sin(), Math.Cos(), and Math.Tan(). using System;

class Trigonometry {

static void Main() {

Double theta; // angle in radians

for(theta = 0.1; theta <= 1.0; theta = theta + 0.1) { Console.WriteLine("Sine of " + theta + " is " +

Math.Sin(theta));

Console.WriteLine("Cosine of " + theta + "is " + Math.Cos(theta));

Console.WriteLine("Tangent of " + theta + "is " + Math.Tan(theta));

Console.WriteLine();

}

}

}

Here is a portion of the program’s output:

Sine of 0.1 is 0.0998334166468282

Cosine of 0.1 is 0.995004165278026

Tangent of 0.1 is 0.100334672085451

Sine of 0.2 is 0.198669330795061

Cosine of 0.2 is 0.980066577841242

Tangent of 0.2 is 0.202710035508673

Sine of 0.3 is 0.29552020666134

Cosine of 0.3 is 0.955336489125606

Tangent of 0.3 is 0.309336249609623

To compute the sine, cosine, and tangent, the standard library methods **Math.Sin( )**, **Math.Cos( )**, and**Math.Tan( )** are used. Like**Math.Sqrt( )**, the trigonometric methodsare called with a **double** argument, and they return a **double** result. The angles must be specified in radians.

|  |
| --- |
| PART I |





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

The decimal Type

Perhaps the most interesting C# numeric type is **decimal**, which is intended for use in monetary calculations. The **decimal** type utilizes 128 bits to represent values within the range 1E–28 to 7.9E+28. As you may know, normal floating-point arithmetic is subject to a variety of rounding errors when it is applied to decimal values. The **decimal** type eliminates these errors and can accurately represent up to 28 decimal places (or 29 places in some cases). This ability to represent decimal values without rounding errors makes it especially useful for computations that involve money.

Here is a program that uses a **decimal** type in a financial calculation. The program computes the discounted price given the original price and a discount percentage.

* Use the decimal type to compute a discount. using System;

classUseDecimal {

static void Main() {

decimal price;

decimal discount;

decimaldiscounted\_price;

* Compute discounted price. price = 19.95m;

discount = 0.15m; // discount rate is 15%

discounted\_price = price - ( price \* discount);

Console.WriteLine("Discounted price: $" + discounted\_price);

}

}

The output from this program is shown here:

Discounted price: $16.9575

In the program, notice that the decimal constants are followed by the *m* suffix. This

is necessary because without the suffix, these values would be interpreted as standard floating-point constants, which are not compatible with the **decimal** data type. You can assign an integer value, such as 10, to a **decimal** variable without the use of the *m* suffix, though. (A detailed discussion of numeric constants is found later in this chapter.)

Here is another example that uses the **decimal** type. It computes the future value of an investment that has a fixed rate of return over a period of years.

/\*

Use the decimal type to compute the future value of an investment.

\*/

using System;

classFutVal {



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 41

static void Main() {

decimal amount;

decimalrate\_of\_return;

int years, i;

amount = 1000.0M;

rate\_of\_return = 0.07M;

years = 10;

Console.WriteLine("Original investment: $" + amount);

Console.WriteLine("Rate of return: " + rate\_of\_return);

Console.WriteLine("Over " + years + " years");

for(i = 0; i < years; i++)

amount = amount + (amount \* rate\_of\_return);

Console.WriteLine("Future value is $" + amount);

}

}

Here is the output:

Original investment: $1000

Rate of return: 0.07

Over 10 years

Future value is $1967.151357289565322490000

Notice that the result is accurate to several decimal places—more than you would probably want! Later in this chapter you will see how to format such output in a more appealing fashion.

Characters

In C#, characters are not 8-bit quantities like they are in many other computer languages, such as C++. Instead, C# uses a 16-bit character type called *Unicode.* Unicode defines a character set that is large enough to represent all of the characters found in all human languages. Although many languages, such as English, French, or German, use relatively small alphabets, some languages, such as Chinese, use very large character sets that cannot be represented using just 8 bits. To address this situation, in C#, **char** is an unsigned 16-bit type having a range of 0 to 65,535. The standard 8-bit ASCII character set is a subset of Unicode and ranges from 0 to 127. Thus, the ASCII characters are still valid C# characters.

A character variable can be assigned a value by enclosing the character inside single quotes. For example, this assigns X to the variable **ch**:

charch;

ch = 'X';

You can output a **char** value using a **WriteLine( )** statement. For example, this line outputs the value in **ch**:

|  |
| --- |
| PART I |



Console.WriteLine("This is ch: " + ch);



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

Although **char** is defined by C# as an integer type, it cannot be freely mixed with integers in all cases. This is because there are no automatic type conversions from integer to **char**. For example, the following fragment is invalid:

charch;

ch = 88; // error, won't work

The reason the preceding code will not work is that 10 is an integer value, and it won’t automatically convert to a **char**. If you attempt to compile this code, you will see an error message. To make the assignment legal, you would need to employ a cast, which is described later in this chapter.

The bool Type

The **bool** type represents true/false values. C# defines the values true and false using the reserved words **true** and **false**. Thus, a variable or expression of type **bool** will be one of these two values. Furthermore, there is no conversion defined between **bool** and integer values. For example, 1 does not convert to true, and 0 does not convert to false.

Here is a program that demonstrates the **bool** type:

* Demonstrate bool values. using System;

classBoolDemo {

static void Main() {

bool b;

b = false;

Console.WriteLine("b is " + b);

b = true;

Console.WriteLine("b is " + b);

* A bool value can control the if statement. if(b) Console.WriteLine("This is executed.");

b = false;

if(b) Console.WriteLine("This is not executed.");

* Outcome of a relational operator is a bool value. Console.WriteLine("10 > 9 is " + (10 > 9));

}

}

The output generated by this program is shown here:

b is False

b is True

This is executed.

10 > 9 is True

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 43

There are three interesting things to notice about this program. First, as you can see, when a **bool** value is output by **WriteLine( )**, “True” or “False” is displayed. Second, the value of a **bool** variable is sufficient, by itself, to control the **if** statement. There is no need to write an **if** statement like this:

if(b == true) ...

Third, the outcome of a relational operator, such as **<**, is a **bool** value. This is why the expression **10 > 9** displays the value “True.” Further, the extra set of parentheses around **10 > 9** is necessary because the **+** operator has a higher precedence than the**>**.

Some Output Options

Up to this point, when data has been output using a **WriteLine( )** statement, it has been displayed using the default format. However, the .NET Framework defines a sophisticated formatting mechanism that gives you detailed control over how data is displayed. Although formatted I/O is covered in detail later in this book, it is useful to introduce some formatting options at this time. Using these options, you will be able to specify the way values look when output via a **WriteLine( )** statement. Doing so enables you to produce more appealing output. Keep in mind that the formatting mechanism supports many more features than described here.

When outputting lists of data, you have been separating each part of the list with a plus sign, as shown here:

Console.WriteLine("You ordered " + 2 + " items at $" + 3 + " each.");

While very convenient, outputting numeric information in this way does not give you any control over how that information appears. For example, for a floating-point value, you can’t control the number of decimal places displayed. Consider the following statement:

Console.WriteLine("Here is 10/3: " + 10.0/3.0);

It generates this output:

Here is 10/3: 3.33333333333333

Although this might be fine for some purposes, displaying so many decimal places could be inappropriate for others. For example, in financial calculations, you will usually want to display two decimal places.

To control how numeric data is formatted, you will need to use a second form of

**WriteLine( )**, shown here, which allows you to embed formatting information:

WriteLine(“*format string*”, *arg0*, *arg1*, ... , *argN*);

In this version, the arguments to **WriteLine( )** are separated by commas and not + signs. The *format string* contains two items: regular, printing characters that are displayed as-is, andformat specifiers. Format specifiers take this general form:

{*argnum*, *width*: *fmt*}

|  |
| --- |
| PART I |





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

Here, *argnum* specifies the number of the argument (starting from zero) to display. The minimum width of the field is specified by *width,* and the format is specified by *fmt.* The *width* and*fmt*are optional.

During execution, when a format specifier is encountered in the format string, the corresponding argument, as specified by *argnum,* is substituted and displayed. Thus, the position of a format specification within the format string determines where its matching data will be displayed. Both *width* and *fmt* are optional. Therefore, in its simplest form, a format specifier simply indicates which argument to display. For example, **{0}** indicates *arg0,* **{1}**specifies *arg1,* and so on.

Let’s begin with a simple example. The statement

Console.WriteLine("February has {0} or {1} days.", 28, 29);

produces the following output:

February has 28 or 29 days.

As you can see, the value 28 is substituted for **{0}**, and 29 is substituted for **{1}**. Thus, the format specifiers identify the location at which the subsequent arguments, in this case 28 and 29, are displayed within the string. Furthermore, notice that the additional values are separated by commas, not + signs.

Here is a variation of the preceding statement that specifies minimum field widths:

Console.WriteLine("February has {0,10} or {1,5} days.", 28, 29);

It produces the following output:

February has 28 or 29 days.

As you can see, spaces have been added to fill out the unused portions of the fields. Remember, a minimum field width is just that: the *minimum* width. Output can exceed that width if needed.

Of course, the arguments associated with a format command need not be constants. For example, this program displays a table of squares and cubes. It uses format commands to output the values.

* Use format commands. using System;

classDisplayOptions {

static void Main() {

int i;

Console.WriteLine("Value\tSquared\tCubed");

for(i = 1; i < 10; i++)

Console.WriteLine("{0}\t{1}\t{2}", i, i\*i, i\*i\*i);

}

}

The output is shown here:



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 45

|  |  |  |
| --- | --- | --- |
| Value | Squared | Cubed |
| 1 | 1 | 1 |
| 2 | 4 | 8 |
| 3 | 9 | 27 |
| 4 | 16 | 64 |
| 5 | 25 | 125 |
| 6 | 36 | 216 |
| 7 | 49 | 343 |
| 8 | 64 | 512 |
| 9 | 81 | 729 |

In the preceding examples, no formatting was applied to the values themselves. Of course, the purpose of using format specifiers is to control the way the data looks. The types of data most commonly formatted are floating-point and decimal values. One of the easiest ways to specify a format is to describe a template that **WriteLine( )** will use. To do this, show an example of the format that you want, using **#**s to mark the digit positions. You can also specify the decimal point and commas. For example, here is a better way to display 10 divided by 3:

Console.WriteLine("Here is 10/3: {0:#.##}", 10.0/3.0);

The output from this statement is shown here:

Here is 10/3: 3.33

In this example, the template is **#.##**, which tells **WriteLine( )** to display two decimal places. It is important to understand, however, that **WriteLine( )** will display more than one digit to the left of the decimal point, if necessary, so as not to misrepresent the value.

Here is another example. This statement

Console.WriteLine("{0:###,###.##}", 123456.56);

generates this output:

123,456.56

If you want to display monetary values, use the **C** format specifier. For example:

decimal balance;

balance = 12323.09m;

Console.WriteLine("Current balance is {0:C}", balance);

The output from this sequence is shown here (in U.S. dollar format):

Current balance is $12,323.09

The **C** format can be used to improve the output from the price discount program shown earlier:

// Use the C format specifier to output dollars and cents.

|  |
| --- |
| PART I |



using System;



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

classUseDecimal {

static void Main() {

decimal price;

decimal discount;

decimaldiscounted\_price;

* Compute discounted price. price = 19.95m;

discount = 0.15m; // discount rate is 15%

discounted\_price = price - ( price \* discount);

Console.WriteLine("Discounted price: {0:C}", discounted\_price);

}

}

Here is the way the output now looks:

Discounted price: $16.96