3 Software Engineering with Control Structures

ur C# programs so far have been simple. All we have learned to do is execute one statement after another in order. We have not had any choices. If we lived 'life like that, we would get up every day, get dressed, and have breakfast no matter how we felt. In reality, we make decisions among alternatives. If we are very sick we might stay in bed and not get dressed. (If we are very lucky, someone might bring us breakfast in bed.) We might not be hungry one morning, so we would get up, get dressed, but skip breakfast. Here is a description of our morning, with decisions:

```
if (I feel ill)
     stay in bed;
else  {
     get up;
     get dressed;
     if (I feel hungry)
         eat breakfast;
}
```

In this "program," what I do depends on whether "I feel ill" is true or false. We will see in this chapter how to write C# expressions that are either true or false, and how to write C# statements that allow us to choose among alternatives based on the truth or falsity of a test expression.

Making choices gives us more flexibility, but we need even more control. For example, if I am thirsty, I might drink a glass of water, but one glass of water might not be enough. What I really want to do is to keep drinking water as long as I am still thirsty. I need to be able to repeat an action. The kind of program I want is

```
while (I feel thirsty)
     drink a glass of water;
```

We will see in this chapter how to write C# statements that allow us to repeat steps in our program.

We think of C# as flowing from one statement to the next as it executes our program. The if and while statements allow us to specify how C# should flow through our program as it executes its statements.

Controlling the flow of execution gives us flexibility as to which statements we execute, but we also need some choices about the type of data we use. So far, we have declared variables only of type int, representing whole numbers. In this chapter we will introduce the type double to represent decimal numbers.

With the if-else and while statements and the type double, we have the language support to create more complex programs,[1] but how do we use these tools to solve problems? In this chapter we introduce a systematic process we can use to develop problem solutions.

OBJECTIVES

- Use relational operators and expressions
- Learn the basic sequence, selection, and repetition statements necessary for a generalpurpose programming language
- Design solutions to problems
- Introduce simple debugging techniques, including a debugger
- Use the double type

3.1 RELATIONAL OPERATORS AND EXPRESSIONS

Arithmetic operators take numeric operands and give numeric results. For example, the value of 3+4 is an integer, 7. By contrast, an expression such as 3<4, stating that 3 is less than 4, gives the value true, and the expression 7<2 gives the value false. Type bool, named for the British mathematician and logician, George Boole (1815-1864), provides two values, true and false, which we use to express the value of relational and logical expressions.

C# provides relational and equality operators, listed in Figure 3.1, which take two operands of a primitive type and produce a bool result

Symbol	Meaning	Example	
<	less than	31 < 25	false
<=	less than or equal	464 <= 7213	true
>	greater than	-98 > -12	false
>=	greater than or equal	9 >= 99	false
==	equal	9 == 12 + 12	false
!=	not equal	292 != 377	true

FIGURE 3.1 C# relational and equality operators

The operators <=, >=, ==, and != are two-character operators which must be together, without any spaces between the two characters. The expression 3 <= 4 is fine, but 3 < = 4 will give an error. (The compiler thinks we want the '<' operator and cannot figure out why we did not give a correct right-hand operand.) ■

We can mix arithmetic operators and relational operators in the same expression, as in

```
643 < 350 + 450
```

which evaluates to true. We can omit parentheses because C# uses precedence rules, as we saw in Section 2.3. Arithmetic operators all have higher precedence than relational operators, so C# adds 350 + 450 giving 800, and then determines that 643 is less than 800. We could have written the expression using parentheses, as in `643 < (350 + 450)`

but in this case we can omit the parentheses and let C# use the precedence rules to evaluate the expression.[2] Some programmers prefer to include parentheses for clarity, even when they are not necessary.

We can use variables in relational expressions, and can declare variables of type bool . For example, if x is an integer variable, the expression

```
x < 3
```

is true if the value of x is less than 3, and false otherwise. The expression

```
x == 3
```

evaluates to true if x equals 3, and to false otherwise

Be careful not to confuse the equality operator, ==, with the assignment operator, =. If x has the value 12, then x == 3 evaluates to false, but x = 3 assigns

the value 3 to x, changing it from 12. ■

EXAMPLE 3.1 ■ Relational.cs

```
/*  Use relational expressions and boolean variables
 */

using System;
public class Relational {
  public static void Main( ) {
    int i = 3;
    bool result;                                        // Note 1

    result = (32 > 87);                                 // Note 2
    Console.WriteLine(" (32 > 87) is {0}",  result);
    result = (-20 == -20);                             // Note 3
    Console.WriteLine(" (-20 == -20) is {0}", result);
    result = -20 == -20;                                // Note 4
    Console.WriteLine(" -20 == -20 is {0}', result);
    result = -20 == -10 - 10;                           // Note 5
    Console.WriteLine(" -20 == -10 - 10 is {0}", result);
    Console.WriteLine(" 16 <= 54 is {0}', 16 <= 54);    // Note 6
    Console.WriteLine(" i != 3 is {0}", i != 3);        // Note 7
  }
}
```

Output

```
(32 > 87) is False
(-20 == -20) is True
-20 == -20 is True
-20 == -10 - 10 is True
16 <= 54 is True
i != 3 is False
```

■

Note 1:bool result;

We can declare variables of type bool, which will have the values True or False.

Note 2:result = (32 > 87);

For clarity we use parentheses, but we could have omitted them because the greater than operator, >, has higher precedence than the assignment operator, =. The value of the bool variable result is false, a literal of the bool type

Note 3:result = (-20 == -20);

We could omit the parentheses because the equality operator, ==, has higher precedence than the assignment operator, =

Note 4: result = -20 == -20;

We do not need parentheses, because == has higher precedence than =.

Note 5:result = -20 == -10 - 10;

This expression uses the equality operator, the subtraction operator, and the negation operator. Again, we do not need parentheses

Note 6: Console.WriteLine(" 16 <= 54 is {0}ℍ, 16 <= 54);

We can use a relational expression in a WriteLine statement without assigning it to a variable. C# will evaluate the expression and display its value. Here we do not need to enclose 16<=54 in parentheses

Note 7: Console.WriteLine(" i != 3 is {0}", i != 3);

Here we used a variable, i, in a relational expression, i != 3. Because i has the value 3, the value of this expression is false

The BIG Picture

The relational operators <, >, <=, >=, ==, and ! = return bool values. The last four are two-character operators that we must type without any space between the two characters. They have lower precedence than the arithmetic operators, but higher precedence than assignment

☑ Test Your Understanding

1. Write a relational expression in C# for each of the following:

a. 234 less than 52

b. 435 not equal to 87

c. -12 equal to -12

d. 6 greater than or equal to 54

2. Evaluate the following relational expressions:

a. 23 < 45

b. 49 >= 4 + 9

c. 95 != 100 - 5

3. What is wrong with the expression (3 < 4) < 5 in C#?

4. If x has the value 7, and y is 12, evaluate each of the following:

a. y == x + 5

b. x >= y - 7

c. 2 * x < y

d. y + 3 != x

5. Explain the difference between x = 5 and x == 5

6. Explain why the expression x > = 3 is not a correct C# expression to state that x is greater than or equal to 3.

3.2 if AND if-else STATEMENTS[3]

We are now ready to make choices about which statements to execute. Three C# statements permit us to make choices. We cover the if and if-else statements in this section. We cover the switch statement, which allows a choice among multiple alternatives, in the next chapter

The if Statement

The if statement is essential because

- It allows us to make choices
- It allows us to solve more complex problems

The if statement has the pattern

```
if (condition)
    if_true_statement
```

as in the example

```
if (x > 2)
    y = x + 17;
```

The condition is an expression such as x > 2 that evaluates to true or false. The if_true_statement is a C# statement such as y = x + 17. If the condition is true, then execute the if_true_statement, but if the condition is false, skip the if_true_statement and go on to the next line of the program. In this example, if x happened to have the value 5, we would assign y the value 22, but if x had the value 1, we would skip the statement y = x + 17.

EXAMPLE 3.2 ▪ Overtime.cs

```
/*  Uses the if statement */

using System;
public class Overtime {
  public static void Main( ) {
    Console.Write('Enter the hours worked this week: ');
    int hours = int.Parse(Console.ReadLine( ));
    if (hours > 40)                                  // Note 1
      Console.WriteLine('You worked overtime this week');
    Console.WriteLine
                ('You worked {0} hours', hours);    // Note 2
  }
}
```

First run

Enter the hours worked this week: 76 You worked overtime this week You worked 76 hours

Second run

Enter the hours worked this week: 8 You worked 8 hours

Note 1:if (hours > 40) Console.WriteLine(“You worked overtime this week”);

The condition, hours > 40, is true if the number we enter is greater than 40, in which case C# executes the WriteLine statement to display the message, You worked overtime this week. If the number we enter is not greater than 40, then C# skips this WriteLine statement.

Note 2: Console.WriteLine (“You worked {0} hours”, hours);

No matter what number we enter, C# executes this WriteLine statement, displaying the value we entered.

Style

Indent all lines after the first to show that these lines are part of the if statement, and to make it easier to read.

Do

```
if (myItem > 10)
  Console.WriteLine("Greater than ten");
```

Do Not

```
if (myItem > 10)
Console.WriteLine('Greater than ten');
```

Control Flow

Control flow refers to the order in which the processor executes the statements in a program. For example, the processor executes the three statements

```
int item1 = 25;
int item2 = 12;
item2 = item1 + 15;
```

one after the other. These three statements are in a sequence. We call this type of control flow, executing one statement after another in sequence, the **sequence** control structure. We can visualize the sequence structure in Figure 3.2, in which we write each statement inside a box and use directed lines to show the flow of control from one statement to the next The if statement allows us to make a choice about the control flow. In Example 3.2, if the hours worked is greater than 40, we print a message about overtime, otherwise we skip this message. We use a diamond shape to represent a decision based on the truth or falsity of a condition. One arrow, called the true branch, shows what comes next if the condition is true. Another arrow, called the false branch, shows **FIGURE 3.2** The sequence control flow

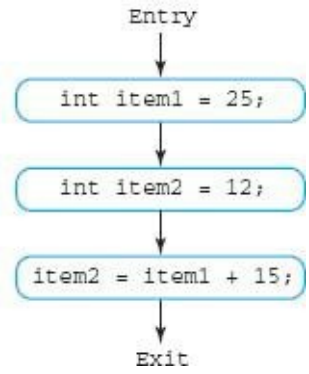
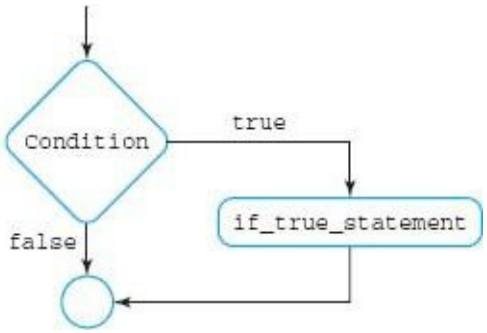


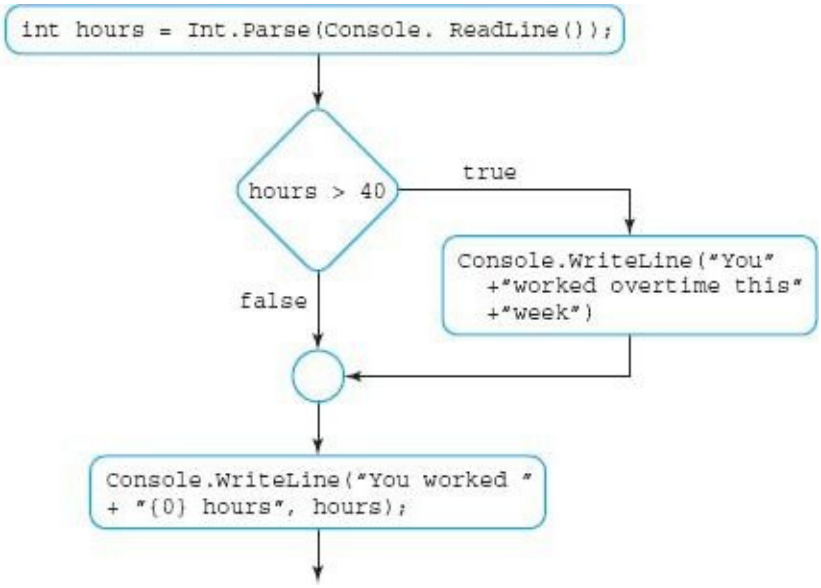
FIGURE 3.3 Control flow for the if statement



what comes next if the condition is false. Figure 3.3 shows the control flow for an if statement. When the condition is true, C# will execute an additional statement. Figure 3.4 shows the control flow for the program of Example 3.2

The if-else Statement

The if statement allows us to choose to execute a statement or not to execute it depending on the value of a test expression. With the if-else statement we can choose between two alternatives, executing one when the test condition is true and the other when the test condition is false **FIGURE 3.4** Control flow for Example 3.2



The if-else statement has the form

```
if  (condition)
     if_true_statement
else
     if_false_statement
```

For example,

```
if (x <= 20)
   x += 5;
else
   x += 2;
```

If x is less than or equal to 20, then we add 5 to it, otherwise we add 2 to it. The if-else statement gives us a choice between two alternatives. We choose if_true_statement if the condition is true and if_false_statement if the condition is false

EXAMPLE 3.3 RentalCost.cs

```
 /* Computes the cost of a car rental.
 */

using System;
public class RentalCost  {

    /* Cost is $30 per day for the first three days
     * and $20 for each additional day.
     * Input: number of days
     * Output: cost of rental
     */
  public static int Cost(int days) {
    int pay;
    if (days <= 3)
      pay = 30*days;                                  // Note 1
    else
      pay = 90 + 20*(days - 3);                       // Note 2
    return pay;
  }

       // Enters number of days. Calls the cost method.
  public static void Main()  {
    Console.Write("Enter the number of rental days: ');
    int days = int.Parse(Console.ReadLine( ));
    Console.WriteLine("The rental cost is {0:C}", Cost(days));
  }
}
```

First run

Enter the number of rental days: 7 The rental cost is $170

Second run

Enter the number of rental days: 2 The rental cost is $60

Note 1:if (days <= 3) pay = 30*days;

If we rent for up to three days, the cost is $30 times the number of days

Note 2:pay = 90 + 20*(days - 3);

If we rent for more than three days, the cost is $90 for the first three days plus $20 for each additional day

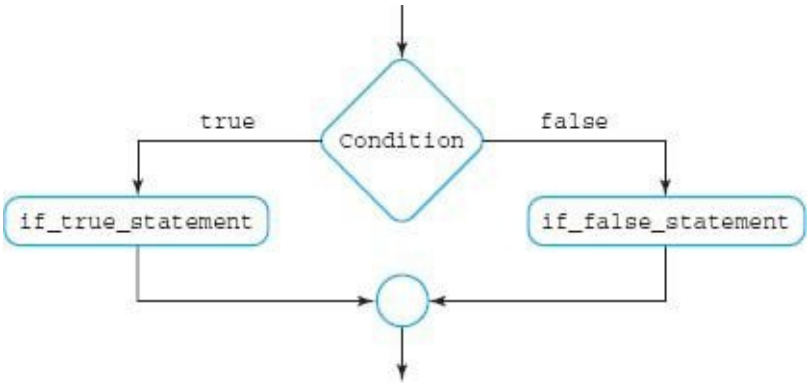
Figure 3.5 shows the flow chart for the if-else statement

Blocks

We can group a sequence of statements inside curly braces to form a **block**, as in

```
{
  x = 5;
  y = -8;
  z = x * y;
 }
```

FIGURE 3.5 Flow chart for the if-else statement



We can use a block as a statement in an if or an if-else statement, as in

```
if (y > 5) {
   x = 5;
   y = -8;
   z = x * y;
}
```

By using a block, we can perform more than one action if the test condition is true. In this example, if y is greater than 5, we want to set x, y, and z to new values

Do not forget to enclose in curly braces the statements that you want to execute if a condition is true. Just indenting them, as in

```
if (y > 5)
  x = 5;
  y = -8;
  z = x * y;
```

will not group the three statements together. We indent to make the program easier to read; indenting does not affect the meaning of the program. Without the braces, C# will interpret the code as

```
if (y > 5)
  x = 5;
y = -8;
z = x * y;
```

If y is greater than 5, then C# will set x to 5. Whether or not y is greater than 5, C# will always set y to -8, and z to x*y. This is quite a different result than we would get if we grouped the three statements in a block, and changed the values of x, y, and z only if the condition is true.

Style

Use a consistent style for blocks to help you match the opening brace, { , with the closing brace , } . One choice is to put the left brace on the same line as the if or else, and to align the right brace with the if or else, as in

```
if (x < 10){
  y = 5;
  z = 8;
}
else  {
  y = 9;
  z = -2;
}
```

Using this style, we can match the keyword if with the closing brace, } , to keep our code neatly organized. Another choice is to align the left brace with the if or else, as in

```
if (x < 10)
{
  y = 5;
  z = 8;
}
else
{
  y = 9;
  z = -2;
}
```

Either of these styles allows us to add or delete lines within a block without having to change the braces. The latter style makes it easier to match opening with closing braces, but uses an extra line to separate the opening brace from the code. We could make the code more compact by putting the braces on the same line as the code, but this is harder to read and modify, and is not recommended

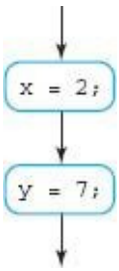
---

A Little Extra : Flow Charts for if Statements with Blocks

Figure 3.3 shows the flow chart for the if statement. If the condition in the diamond is true, then the control flows to the statement in the box on the right. Remember that this statement can be a block, such as

```
{
  x = 2;
  y = 7;
}
```

which has the flow chart



When the statement in the box is a block, we replace the box by the flow chart for that block. We find the flow chart for the if statement

```
if (z <= 10) {
   x = 2;
   y = 7;
}
```

FIGURE 3.6 Flow chart for if statement with block, step 1

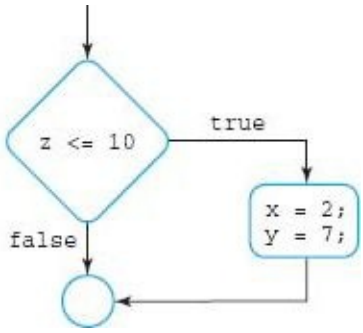
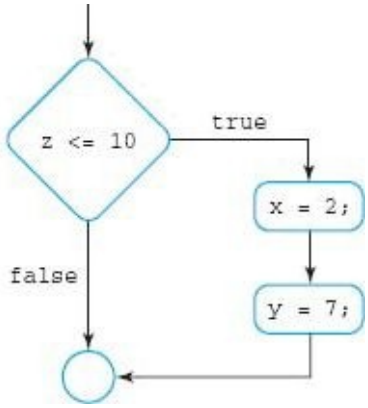


FIGURE 3.7 Flow chart for if statement with block, step 2



in two steps. First, Figure 3.6 applies the pattern of Figure 3.3 to this example. Second, in Figure 3.7 we replace the block statement by its flow chart.

The BIG Picture

The if statement allows us to make a choice to execute a statement or not. The if-else statement lets us choose between two alternatives. Each alternative may be a simple statement or a block, which uses curly brackets to enclose C# code. Flow charts show the flow of control in diagram form

✓Test Your Understanding

7. Correct the error in each of the following:

a. if {x == 12} y += 7;

b. if (x=12) y += 7;

c. if (x == 12) then y += 7;

8. Correct the error in each of the following:

```
a. if (y > 5)        b. if  y > 5        c. if (y > 5)
     z = 7;               z = 3;              z= 3;
     x = 5;             else                else  (
   else                   x = y + 2;            s = y + 7;
     w = 4;                                     z = s - 2;
                                               );
```

9. How would you improve the style in each of the following?

```
a. if (y <= 6)    b. if (x != 0)
     z += 5;            y+=5;
                        else
                        z = y + 9;
```

10. Draw the flow chart for each of the following if statements:

```
a. if (x == 17)    b. if (s > 12)
     y += 6;             z = 52;
```

11. Draw the flow chart for each of the following if-else statements.

```
a. if (z <= 10)    b. if (x == 0)
     y -= 11;             z = 3*x;
   else               else
     y += 10;             y = 4 + x;
```

A Little Extra

12. Draw the flow chart for each of the following if statements with blocks.

```
a. if (s > 4) {    b. if (r != 0) {
     y = s - 30;          x = 4*r + 3;
     z = 2;               w = 3;
   }                      y = x + w;
                        }
```

3.3 THE TYPE double[4]

In this section, we introduce the type double for decimal values. In the next chapter we will cover other numeric types

Scientific Notation

In many applications, we need to use decimal numbers such as 2.54, the number of centimeters in an inch. Small decimal numbers like .00000003937, the

number of inches in a nanometer, are harder to read, but we can use floating-point notation, 3.937E-8, to simplify them. The number following the E, -8 in this example, tells us that we need to move the decimal point eight places to the left to get the value .00000003937. We can write 5,880,000,000,000, the number of miles in a light-year, more conveniently as 5.88E12. The positive exponent, 12, in 5.88E12 informs us that we need to shift the decimal point twelve places to the right to get the number 5880000000000.0.

The number following the E, called the exponent, tells us how many places to shift the decimal point. To write very large or very small numbers in a way that is easier to read, we float the point to a more convenient location, to the right of the first nonzero digit, and adjust the exponent to keep the value of the number unchanged.

We often call this notation using exponents *scientific notation*, because of its usefulness in expressing the varied sizes of numbers used in scientific calculations. Options in scientific notation are to use a lowercase e, as in 5.88e12, and to use a plus sign with a positive exponent, as in 5.88E+12. Of course, we can use both options, as in 5.88e+12.

We can express the same number in many ways using scientific notation. For example, each of the following expresses the value .00000003937

```
3.937E-8      393.7E-10     .03937E-6     .00000000003937E+3
```

The usual choice is to float the decimal point just to the right of the first nonzero digit, as in 3.937E-8.

For display purposes, we write numbers that are not too small or too large without exponents, as in 2.54 or 2755.323 or .01345. Very small or very large numbers are easier to read in scientific notation, as in 5.88E+12

☑ Test Your Understanding

13. Express the following without using exponents:

a. 345.22E-2

b. —2.98765e4

c. .000098765E+8

d. 435e-2

14. Express the following in scientific notation with the decimal point to the right of the first nonzero digit:

a. 893.454

b. .000345722

c. 98761234
d. .090909

15. Which of the following express the same value?
a. .6789E+2
b. 6.789e-2
c. 67.89
d. 678.9e-2
e.0.006789

double Values

Now that we have seen how to write decimal numbers using scientific notation, we can introduce the type double, which C# provides for decimal numbers. Numbers of type double provide 15 decimal digits accurately. If you are using scientific notation, the exponents for double values can range from -324 to 308. A literal of type double may have a decimal point or an exponent or both, and it must have at least one digit. Some valid values of type double are:

```
22.7
 4.123E-2
36e2
 3.
 0.54296
  .1234
```

When we write a decimal literal such as 2.54, C# treats it as a value of type double . We can declare variables of type double as, for example

double length;

double height;

and we can initialize variables in the declaration as, for example

Expression			Value
6.2	+	5.93	12.13
72.34	-	2.97	69.37
32.3	*	654.18	21130.014
3.0	/	2.0	1.5

We can use the arithmetic operators, +, -, *, and /, with double operands. For example,

When dividing numbers of type double, be sure to include the decimal points. The expression 3/2 will use integer division, which will truncate the value to 1. Writing 3.0/2.0 gives the decimal value 1.5.

Output

If we output values of type double using the WriteLine method, we have several choices for the format specification. Example 3.4 illustrates some of the possibilities. We illustrate the E, F, and G formats,

```
E   Scientific
F   Fixed-point
G   General
```

We can use these formats alone or with a precision specifier. For example, F3 specifies a fixed-point format with three places after the decimal point

EXAMPLE 3.4 DoubleOut.cs

```
/* Illustrates output formatting for type double.
 */

using System;
public class DoubleOut {

    /* Creates double values using formats to see
     * how C# WriteLine statements display them.
     */
  public static void Main( ) {
    double five3rds = 5.0/3.0;                          // Note 1
    Console.WriteLine
      ("Default precision     {0}", five3rds);          // Note 2
    double threeHalves = 3.0/2.0;
    Console.WriteLine
      ("No trailing zeros     {0}", threeHalves);       // Note 3
    Console.WriteLine
      ("Default               {0}", 1234567890.987);    // Note 4
    Console.WriteLine
      ("Fixed, two places     {0:F2}", 1234567890.987);// Note 5
    Console.WriteLine
      ("Exponent              {0:E}", 1234567890.987); // Note 6
    Console.WriteLine
      ("Exponent, two places  {0:E2}", 1234567890.987);// Note 7
    Console.WriteLine
      ("General               {0:G}", 1234567890.987); // Note 8
    Console.WriteLine
      ("Changes to fixed      {0}", 1234567.890987E2); // Note 9
    Console.WriteLine
      ("Changes to scientific {0}", .0000123456789);  // Note 10
  }
}
```


```
Default precision     1.66666666666667
No trailing zeros     1.5
Default               1234567890.987
Fixed, two places     1234567890.99
Exponent              1.234568E+009
Exponent, two places  1.23E+009
General               1234567890.987
Changes to fixed      123456789.0987
Changes to scientific 1.23456789E-05
```

Note 1:double five3rds = 5.0/3.0;

C# treats decimal literals such as 5.0 as type double.

Note 2: Console.WriteLine (“Default precision {0}”, five3rds);

A double value is accurate to 15 decimal digits. C# converts the internal binary number to the closest decimal equivalent.

Note 3:double threeHalves = 3.0/2.0; Console.WriteLine (“No trailing zeros {0}”, threeHalves);

C# does not print zeroes at the end of a number. Every double value has 15 digits, but C# does not display the trailing zeroes.

Note 4: Console.WriteLine (“Default {0}”, 1234567890.987);

C# uses the General (G) format when we do not include any format specifier. Thus {0} has the same effect as {0:G}. The G (or default) specifier will use the scientific format if the exponent the number would have when written in scientific notation is less than –4. Writing such a number in fixed-point format would require a number of leading zeroes. It will also use scientific notation if the exponent is greater than or equal to the number of significant digits in the number. The G specifier will use a fixed-point format if neither of these two conditions applies, which is the case here.

Note 5: Console.WriteLine (“Fixed, two places {0:F2}”, 1234567890.987);

F2 specifies fixed-point notation with two places after the decimal point.

Note 6: Console.WriteLine (“Exponent {0:E}”, 1234567890.987);

The E format uses scientific notation. It uses six places after the decimal point when no precision is specified. The exponent uses three places, filled with zeroes if necessary.

Note 7: Console.WriteLine (“Exponent, two places {0:E2}”, 1234567890.987);

The E2 format specifies two places after the decimal point using scientific notation.

Note 8: Console.WriteLine (“General {0:G}”, 1234567890.987);

The G format is the default if no format is specified.

Note 9: Console.WriteLine (“Changes to fixed {0}”, 1234567.890987E2);

Here the exponent is less than the number of significant digits, so the display is in fixed-point.

Note 10: Console.WriteLine (“Changes to scientific {0}”, .0000123456789);

Here the exponent is less than –4, so the display is in scientific notation.

Input

In C#, we can use the double.Parse method to convert the input String to a double value. Example 3.5 inputs hot and cold Celsius temperatures and converts them to Fahrenheit

EXAMPLE 3.5 Temperature.cs

```
/* Converts degrees Celsius to degrees Fahrenheit.
 */

using System;
public class Temperature {
  public static void Main() {
    double  hotC, coldC;
    double  hotF, coldF;
    Console.Write("Enter a hot temperature in Celsius: ");
    String input = Console.ReadLine();
    hotC = double.Parse(input);
    hotF = 9.0*hotC/5.0 + 32.0;                          // Note 1
    Console.WriteLine
       ("The Fahrenheit temperature is {0:F1}", hotF);
    Console.Write("Enter a cold temperature in Celsius: ");
    input = Console.ReadLine();
    coldC = double.Parse(input);                         // Note 2
    coldF = 9.0*coldC/5.0 + 32.0;


   Console.WriteLine
      ("The Fahrenheit temperature is {0:F1}", coldF);
  }
}
```

Run

Enter a hot temperature in Celsius: 56 The Fahrenheit temperature is 132.8
Enter a cold temperature in Celsius: -19.33 The Fahrenheit temperature is -2.8

Note 1: hotF = 9.0*hotC/5.0 + 32.0;

To convert degrees Centigrade to degrees Fahrenheit, we use the formula

F = 9C/5 + 32.

We write the constants with decimal points to show that they have type double. In the next subsection, we will discuss mixed-type in which

the constants might be integers.

Note 2: coldC = double.Parse(input);

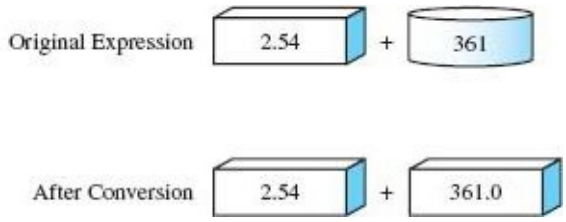
Type double is a shorthand for System.Double, so we could have written this method as Double.Parse.

Mixed-type Expressions

Usually a numeric expression uses all variables and literals of type int, or all type double. For example, 2.54 + 3.61 is a double addition and 254 + 361 is an int addition. The addition operator, +, looks the same in both cases, but int addition is quite different from double addition because, in the computer memory, int and double values are stored differently. For convenience, we use the same + symbol to represent these two kinds of addition When we mix types in an expression, as in 2.54 + 361, which type of addition does C# use, int or double? As they say, it is like adding apples and oranges. We cannot just add a double to an int because they are different types. We cannot convert real apples to real oranges, or oranges to apples, but in the numeric case we have better luck. We cannot convert a double such as 2.54 to an int without losing information. Rounding it to 3 probably is not a good choice. But we can convert an int to a double without losing information, for example, by changing 361 to 361.0

Although we do not lose any information by converting 361 to 361.0, it does require a change inside memory, because 361 has an internal representation that is a lot different from that for 361.0. Because we can always convert an int to a double without losing information, C# will do it for us automatically. If we write 2.54 + 3 61, C# will convert 361 to 361.0, and use type double addition to get the result 363.54. Figure 3.8 illustrates this conversion process.

FIGURE 3.8 Conversion of a mixed-mode expression



We could have used this automatic conversion in Example 3.5, where we used the expression

```
9.0*c/5.0 + 32.0
```

in which we wrote all the literals as double values, 9.0, 5.0, and 32.0. Letting C#

do the conversion from int to double, we could instead have written

```
9*c/5 + 32
```

where c has type double

C# converts from int to double only in a mixed-mode expression where one operand has type int and the other has type double. The expression

```
9*c/5 + 32
```

will not give the result we expect. The division 9/5 has both operands, 9 and 5, as integers, so C# uses integer division, obtaining the integer quotient of 1, not the value 1.8 that we want. Writing the expression as

```
9*c/5 + 32
```

where c has type double, works because 9*c is a mixed-mode expression in which one operand, 9, has type int and the other, c, has type double. The result, 9*c, is a double value, so (9*c)/5 is a mixed-mode expression and C# will convert the 5 from int to double. Finally, (9*c)/5 has type double, so C# will convert 32 from an int to a double before doing the addition.

C# will also convert from int to double in an assignment statement. For example, in

double d = 4;

C# will assign the double value 4.0 to the variable d. However, C# will not automatically convert a double to an int to assign it to an integer variable, because a double value may be out of the range of values an int variable can hold.

The general rule C# follows for these implicit conversions of one primitive type to another is that we can assign any numeric value to any numeric variable whose type supports a larger range of values

A Little Extra ; Type Casts

As we saw, C# will do an implicit conversion from int to double in a mixed-type expression. We could explicitly cast the type from int to double by putting the

desired type, double, in parentheses to the left of the int literal or variable we wish to convert. For example, in the expression `2.54 + (double)361`

C# converts the value 361 to type double before adding it to 2.54

By using an explicit cast, we show that we really want C# to convert from one type to another. If we always use explicit type casts, then when checking our code we would recognize an implicit mixed-type expression, such as 2.54 + 361, as an error, say, of omission of a decimal point in 361. We might have meant to write 2.54 + 3.61, but instead wrote a mixed-type expression. Because C# does not treat such a mixed-type expression as an error, we would have to inspect our

results carefully to see that they are not correct

Just because your program compiles and runs does not mean that the results are correct. Always check that your results are reasonable. Make a prediction before you run your program. If you expect the result to be positive, do not accept a negative value without further investigation. If you expect the result to be about 10, do not accept a value of 17,234 without more checking.

The BIG Picture

Floating-point notation allows us to write large and small numbers conveniently. The type double provides 15-place accuracy. By default, C# will use the G format for output. We can use the E format to specify scientific notation or the F for fixed-point. Each format allows a precision specification for the number of decimal places. C# lets us use some forms of mixed-type expressions.

Test Your Understanding

16. What will be the result of each division?

a. 5.0 / 2.0

b. 5 / 2

c. 12 / 5

d. 12.0 / 5.0

17. Suppose that the double variable, x, has the indicated value. Show the result of Console.WriteLine(“{0}”,x)

a. 3456.789

b. .0000023456

c. .09876543

d. 1234567890.987

e. _234567.765432

18. Suppose that the double variable, x, has the indicated value. Show the result of Console.WriteLine("(0:E2}",x)

a. 3456.789

b. .0000023456

c. .09876543

d. 1234567890.987

e. –234567.765432

Try It Yourself

19. In Example 3.5, change the formula

```
9.0*c/5.0 + 32.0
```

to

```
9*c/5 + 32
```

Do you think the program will still work correctly? Rerun Example 3.5 with this change to verify that your answer is correct

Try It Yourself

20. In Example 3.5, change the formula

```
9.0*c/5.0 + 32.0
```

to

```
9/5*c + 32
```

Do you think the program will still work correctly? Rerun Example 3.5 with this change to verify that your answer is correct

A Little Extra

21. Rewrite each of the following to use explicit type casts:

a. 72 + 37.5

b. 23.28 / 7

c. double d = 874;

3.4 PROGRAM DESIGN WITH THE while LOOP[5]

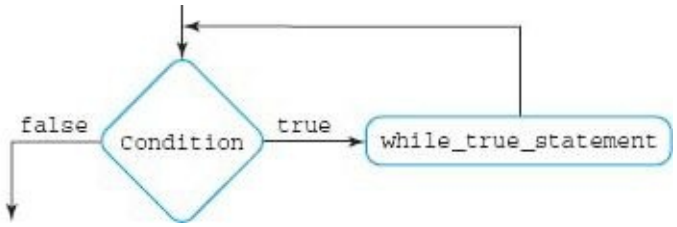
The if and if-else statements give us the ability to make choices. In this section we will see how the while statement enables us to repeat steps

Repetition

The while statement follows the pattern

```
while (condition)
  while_true_statement
```

FIGURE 3.9 Flow chart for the while loop



with the flow chart shown in Figure 3.9, where the condition evaluates to true or false, and the while_true_statement can be any C# statement including a code block. If the condition is true, C# executes the while_true_statement and goes back to check the condition again. If the condition is still true, C# executes the while_true_statement and goes back to check the condition again, and so on. This process repeats until the condition is false For example, suppose that the variable x has the value 7 just before C# starts to execute the while statement

```
while (x < 10)
   x += 2;
```

Because 7 < 10 is true, C# executes the statement x += 2, which changes the value of x to 9. Remember, this is a while statement, so C# again checks the condition, x < 10. Because 9 < 10 is still true C# again executes, x += 2, giving x the value 11. Now checking the condition x < 10, C# finds that 11 < 10 is false, so the execution of the while statement is finished The while statement is a type of **loop**, so called because execution keeps looping back to check the condition after every execution of the while_true_statement, which we call the **body** of the loop. The body of the loop could be a block, in which case C# executes every statement in the block while the condition is true. For example, if x had the value

```
while (x < 10) {
   Console.WriteLine('x is now {0}', x);
   x += 2;
 }
```

would output

```
x is now 5
x is now 7
x is now 9
```

5 before the loop, the while statement

The condition in a while statement may evaluate to false on the first entry to the loop, in which case C# never executes the body of the loop. For example, if x has the value 3 before executing the loop

```
while (x >= 5)
  x -= 4;
```

then the condition, 3 >= 5, is false, and the loop body, x -= 4, is never executed

Planning Programs Using Pseudocode

With the while statement we can solve more interesting problems. However, as problems get more complex, we need to plan our solutions carefully. In this section, we develop a program, Example 3.6, which uses a while loop to find and display the sum of test scores inputted by the user Before coding, we need to plan our approach. The user will keep entering test scores, which are nonnegative. To signal to the program that no more scores are available the user can enter a negative number, which cannot be a valid score. We call this value a **sentinel.** The program checks each value the user inputs. If it is the sentinel, the program exits the while loop and outputs the total of all the scores. If it is not the sentinel the program adds the score to the total. We can use a while loop to code these repetitive steps. Informally, we could write the loop as

```
Read the first score;
while (the score is not the sentinel) {
  Add the score to the total so far;
  Read the next score;
}
Output the total of all the scores.
```

We call this informal way of writing the program **pseudocode.** Pseudocode helps us design the C# program. It is easier to understand than the very specific programming language statements, so we can correct logical errors before they get into the actual program We can use a flow chart in the same way as pseudocode to show the structure of a program as we develop the detailed solution. For small programs both techniques work well, but for larger programs

it becomes cumbersome to manage the flow charts, making pseudocode a better choice Pseudocode makes sense. We can execute it mentally with some test data as a further check. Suppose the scores are 56, 84, and 75, and that the user follows them with the sentinel, -1. Let us number each line and trace, line by line, how the pseudocode would execute

Pseudocode

```
1       Read the first score;
2       while ( the score is not the sentinel) {
3            Add the score to the total so far;
4            Read the next score;
5       }
6       Output the total of all the scores.
```

Trace of the pseudocode:

Line Number	**Action**
1	Read the first score: **56**
2	>56 is not the sentinel.
3	Add score to total: total is 56
4	Read the next score: **84**
2	84 is not the sentinel.
3	Add score to total: total is 140
4	Read the next score: **75**
2	75 is not the sentinel.
3	Add score to total: total is 215
4	Read the next score: **–1**
2	–1 is the sentinel.
6	Output the total of 215.

This trace looks correct. We should also check our pseudocode when the user has no scores to enter

Line Number	**Action**
1	Read the next score: **–1**
2	–1 is the sentinel.
6	Output the total of ???.

Here we get to line 6 without ever adding any scores to the total. The pseudocode always displays the total, whether or not the user entered any scores. We need to make sure that whenever we get to line 6, we have a total to display.

If the user has no scores to enter, we could display zero. To set this initial value for the total, we could add a line at the beginning of the pseudocode, say line 0,

```
0.Initialize the total to zero.
```

that sets the total to zero. Then, if we do not read any scores, we will still have a total to display. Actually, we implicitly use the initial value of total in line 3 after we read the very first score. Line 3 states add the score to the total so far, but before reading any scores the total does not have a value unless we give it an initial value in line 0.

If we want to be meticulous, we might want to distinguish between the case when the total is zero because no scores were entered and when it is zero because a bunch of zero scores were entered. More broadly, we might want to keep track of the number of scores entered. Should we do these things or not? In this example, the problem does not explicitly ask to keep track of the number of scores, and for simplicity we will not add this feature now. The problem does not say what to do if no scores are available, but we assume that displaying zero is sensible, and to be sure, we check with the proposer of the problem, who agrees.

We have taken a lot of trouble to design a solution to a simple problem. Because we have been careful, we should be able to convert the pseudocode to a C# program that solves the problem. In this chapter we use a procedural approach. When we start object-oriented programming in Chapter 5, we shall see how to design a solution using an object.

We want to abstract operations that model our understanding of the problem. Looking at the pseudocode, we identify three operations

```
Input a score from the user.
Add the score to the total.
Output the total.
```

If we define a method to implement each of these operations, we can easily translate our pseudocode into an understandable C# program.

We could use any valid identifiers, not keywords, for our method and variable names. To make our program as readable as possible, we choose meaningful names, GetScore and UpdateTotal for our method names. The GetScore method will have a String prompt as a parameter. It will return the value the user enters. We will pass the new score and the current total to the UpdateTotal method. It will return the updated total

EXAMPLE 3.6 ■ ScoreSum.cs

```
/*  Uses a while loop to compute the sum of
 *  test scores.
 */

using System;
public class ScoreSum  {

     // returns a score entered by the user
  public static int GetScore(String prompt) {
    Console.Write(prompt);
    return int.Parse(Console.ReadLine());
  }

     /* Input:    The new score and the total so far
      * Output:  The updated total
      */
  public static int UpdateTotal(int newValue, int subTotal) {
                                                          // Note 1
    subTotal += newValue;
    return subTotal;
  }

     // loops until the user enters -1
  public static void Main(String [] args) {
    int total = 0;                       // Sum of scores
                                                          // Note 2
    int score = GetScore('Enter the first score, -1 to quit: ");
    while (score >= 0) {
      total = UpdateTotal(score, total);
      score =  GetScore("Enter the next score, -1 to quit: ');
    }
    Console.WriteLine("The total is {0}", total);
  }
}
```

Run

```
Enter the first score, -1 to quit:
Enter the next score, -1 to quit: 45
Enter the next score, -1 to quit: 56
Enter the next score, -1 to quit: -1
The total is 135
```

Note 1:

```
Note 1:  public static int UpdateTotal(int newValue, int
         subTotal){
```

We choose variable names, newValue and subTotal, that remind us of the role these quantities play. Our program is easier to understand than if we had used variable names x and y for these quantities.

Note 2:

```
Note 2:  int total = 0;
```

We initialize total to 0, representing the sum of scores before the user

has entered any scores. If the user enters the sentinel, _1, before entering any scores, the output will be 0. The C# code closely follows the steps of the pseudocode.

Having completed this program nicely, we might ask about alternative solutions in case another approach might allow us to improve our program. Instead of using a sentinel to signify the end of the input, the user might specify how many scores are available. Then we can have our loop condition check to see if all scores have been entered. If not, the body of the loop reads the next score and adds it to the total. The pseudocode for this approach is

```
Read the quantity of scores to read;
while (number of scores read so far < quantity to read) {
  Read the next score;
  Add the score to the total so far
}
Display the total;
```

Each time around the loop we need to check the condition

```
number of scores read so far < quantity to read.
```

We know how many scores the user will enter. If we have not read that many, there are more scores; otherwise, we have read them all. But how many scores have we read? Let us keep track as we read.We could include another variable, count, to hold the number of scores read so far.We initialize count to 0, and each time we read a score, we increase count by 1. The revised pseudocode is shown in Figure 3.10.

We leave the next steps in this solution process, which include tracing the execution of the pseudocode with sample data and converting it to a C# program, to the exercises. Note that to solve the problem using this approach, we had to know the quantity of test scores, so we added that to the output to make it more informative.

```
Read the quantity of scores;
while (count < quantity) {
     Read the next score;
     Add the score to the total so far;
     Increment the count of scores;
}
Display the quantity and the total;
```

FIGURE 3.10 Pseudocode for the sum of test scores problem

Think carefully when writing the condition (often called the test) for the while loop. Getting it correct can be tricky. In the preceding example we might easily have written count <= quantity, which is incorrect because when count equals quantity, then we have already read all the scores and should terminate the loop rather than ask for another score.

A Little Extra: Input Validation

In Example 3.6, we assumed that the user would enter correct data. Entering a value such as 34.5 or hat will cause the program to abort. Later we will introduce exception handling, which we can use to recover from such input errors. Even if the user enters a correct int value, that value could be 312017 or - 52, neither of which is likely to be a test score. We could have prompted the user to enter a value from 0 to 100, as in `Enter a score between 0 and 100`

Even then a careless user could enter an invalid value.[6]

Style

Do declare each variable on a separate line and use a comment to specify the use of that variable in the program.

Do use meaningful variable names that remind us of the purpose of that variable.

Do precede the program with a comment that describes the intent of the program, and the input and output. This comment is a good place to put the programmer's name and the date. Precede each method with a descriptive comment.

Do include comments to explain any parts of the code that might be difficult to understand. (In our examples, we use Notes at the end of the code for this purpose.)

Loop Termination

Each time the loop condition is true, C# executes the loop body. In order for the loop to terminate, something must change that causes the condition to fail. In the

loop

```
while ( x < 10)
     x += 2;
```

we add 2 to x each time we execute the body. Eventually x will become greater than or equal to 10, so the condition x < 10 will fail and the loop will terminate

If we were just to display the value of x, and not increase it, then the loop might never stop. For example, if x has the value 5 when C# executes the loop

```
while (x < 10)
     Console.WriteLine('x is now {0}', x);
```

the result will be an unending sequence

```
x is now 5
x is now 5
x is now 5
...
```

and so on until someone aborts the program. (Holding the *Control* key down and pressing the *C* key will interrupt the program on Windows systems.)

The last example, repeatedly displaying x is now 5, made it easy to spot that something was very wrong. Sometimes an unending loop (often called an **infinite loop)** may have the opposite behavior, showing nothing at all to the user while it goes on computing. For example, if x has the value 5, the loop

```
while (x < 10)
  x -= 2;
```

keeps subtracting 2 from x, never terminating. The value of x is 5,3,1, -1, -3, -5, and so on. The condition, x < 10, is always true, so the loop keeps on executing, but displays nothing to the user. The programmer's first response is to blame the computer for being very slow, because nothing seems to be happening, but that eerie stillness could be a symptom of an infinite loop. If so, the user must interrupt, aborting the program Remember when writing a while statement that something must eventually cause the condition to be false

Beware of loops that never terminate. We use loops because we want repetition, but we must make sure that the repetition stops. Check each loop you write to make sure that it will terminate.

Finding the Maximum

We close this section by solving the following problem: "Find the largest of the prices entered. The user will input a sequence of prices, and input a negative item as a sentinel to indicate that input has ended. The program will output the maximum of the prices."

First, let us try to solve the problem informally, supposing that the user inputs 49.23, 16.78, 92.14, 32.75, and –1.00. We want to compare each new price to the maximum price entered so far, to see if the new price is larger. When we get the first price, we have no previous maximum to compare with. But because all the prices are positive numbers, we can initialize our maximum to zero, which is a value smaller than any price By initializing the maximum to zero, we can perform the same steps with each price the user inputs. We compare the first price, 49.23, to the initial maximum of 0.0. Because 49.23 is larger than 0.0, we save it as the new maximum. Looking at the second price, 16.78, we see that it is smaller than the current maximum, so we go on. The third item, 92.14, is greater than the current maximum, so we save it as the new maximum. Finding that the fourth item, 32.75, is smaller than the maximum so far, we go on and find that the fifth invoice item is negative, so we stop and output the maximum, which is 92.14

Our solution works for this example, but will it work for other inputs? Let us try the boundary case, when the user just inputs -1.00, signaling the end of the data. Our first step was to look at the first item and compare it to the initial maximum. We neglected to specify that we first need to check if the value the user input is negative, in which case we quit. Here, if we look at the first item, -1.00, we see that we should quit, perhaps displaying a message that no positive items were input.

The pseudocode is

```
Initialize the maximum to zero.
Input the first price.
while (the price is not the sentinel) {
  Update the maximum.
  Input the next price.
}
Output the maximum.
```

From the pseudocode, we identify three operations:

```
input a price
update the maximum
display the maximum
```

EXAMPLE 3.7 ■ Max.cs

```
/*  Finds the maximum of prices that the user enters.
 *  The user enters a negative value to indicate that
 *  no more data is available. The program displays the
 *  maximum value.
 */

using System;
public class Max {

     /* Input:    The new price and the current maximum price
      * Output: The updated maximum
      */
  public static double UpdateMax
    (double newPrice, double maxSoFar) {


              if (newPrice > maxSoFar)                        // Note 1
                return newPrice;
              else
                return maxSoFar;
            }

             // returns a double entered by the user
            public static double GetDouble(String prompt) {
              Console.Write(prompt);
              return double.Parse(Console.ReadLine());
            }

             /* Inputs prices until the user enters -1
              * Outputs the maximum of all prices entered.
              */
            public static void Main() {
              double maxSoFar = 0;   // Max of prices entered
                                                              // Note 2
              double price = GetDouble
               ("Enter the first price, -1 to quit: ");    // Note 3
              while (price >=0 ) {
                maxSoFar = UpdateMax(price, maxSoFar);
                price = GetDouble("Enter the next price, -1 to quit: ");
              }
              Console.WriteLine('The maximum is {0}', maxSoFar);
            }
          }
```

First run

```
Enter the first price, -1 to quit: 49.23
Enter the next price, -1 to quit: 16.78
Enter the next price, -1 to quit: 92.14
Enter the next price, -1 to quit: 32.75
Enter the next price, -1 to quit: -1
The maximum is 92.14
```

Second run

Enter the first price, -1 to quit: -1
The maximum is 0

Note 1:

```
Note 1:   if (newPrice > maxSoFar)
             return newPrice;
```

UpdatePrice returns the new price if it is greater than the maximum found so far. Otherwise, it returns the current maximum unchanged

Note 2:

```
Note 2:  double maxSoFar = 0;
```

We initialize maxSoFar to 0 because the prices are all nonnegative, so the maximum must be at least 0

Note 3:

```
double price =
GetDouble('Enter the first price, -1 to quit: ');
```

We get the first price before entering the loop because the while loop checks the price to see if it is the sentinel before trying to update the maximum. We could have simply initialized the price to zero with the declaration

```
double price = 0;
```

The first time we evaluate the loop condition, it would harmlessly check the zero value and go on to read the prices the user enters. By doing this, we would only need to call the GetDouble method once in the body of the loop.

The BIG Picture

The while statement enables us to repeat steps. We use pseudocode to plan problem solutions carefully. When writing while loops, we must make sure that the condition becomes false so the loop terminates

✓ Test Your Understanding

22. How many times will the body of each of the following while loops be executed if x has the value 5 at the start of the loop?

```
a. while (x <= 10)
      x +=3;

b. while (x == 2)
      x -= 7;

c. while (x > 1)
      x--;
```

23. Find any errors in the following while loops:

```
a. while (x != 9)
     x +=4;
c. while (x =! 7)
     x++;
```

```
b. while (x)
     x *= 2;
```

24. Trace the execution of the pseudocode in Figure 3.10, assuming test scores of:

a. 95, 46, 68, and 79
b. 14, 87, 35, 76, and 80

25. Which of the following loops terminate? Assume that x has the value 12 at the start of the loop

```
a. while (x != 5)
     x++;
c. while (x != 5)
     x = 5;
```

```
b. while (x != 5)
     x--;
```

26. Draw the flow chart for each of the following while loops.

```
a. while(y < 7)
     y += 4;
c. while (z != 0) {
     x = 4 * z;
     z++;
   }
```

```
b. while(y > 5) {
     x = y + 10;
     y --;
   }
```

27. Develop a solution in pseudocode to the problem of looking up a number in the telephone directory

28. Develop a solution in pseudocode to the problem of finding the minimum of a sequence of nonnegative numbers entered by the user, where the user enters a negative number to terminate the input

29. Trace the execution of the pseudocode preceding Example 3.7 for finding the maximum of numbers input by the user, with the input data 49.23, 16.789, 92.145, 32.7, and -1

30. Develop a solution in pseudocode to the problem of finding the maximum of a sequence of nonnegative numbers entered by the user. In this solution, before reading the numbers, ask the user to enter how many

numbers will be input

31. Develop a solution in pseudocode to the problem of finding both the maximum and the minimum of a sequence of nonnegative numbers entered by the user. In this solution, before reading the numbers, ask the user to input how many numbers will be input

32. Develop a solution in pseudocode to the problem of counting the number of negative numbers in a sequence of numbers the user inputs. In this solution, before reading the numbers, ask the user to input how many numbers will be input. (Example: For input 32,76, -12, 49, -11, and -3 the output should be that there are three negative numbers.)

3.5 DEBUGGING

Following the careful problemsolving methods described in the last section will help us to produce programs free from errors. Hasty coding before developing a careful solution is much more likely to lead to errors. In this section, we discuss some simple approaches to debugging, finding, and correcting any errors in the program.

We seek to correct errors in logic in a program that compiles but either aborts with an error message while running or produces incorrect results. Those learning a new language or with little prior programming experience make many syntax errors, writing C# statements and expressions incorrectly, as part of the learning process. The compiler catches these syntax errors and provides messages to help the programmer correct the syntax.

Our examples in this section compile but do not produce the desired results. We suggest some simple techniques for finding and correcting the observed errors. We take a simple problem to sum the squares of the integers from 1 to a high value entered by the user. Example 3.8 is an attempted solution

EXAMPLE 3.8 Mistake1.cs

```
/* Incorrect attempt to sum the squares of
 * numbers from 1 to a high value entered
 * by the user
 */

using System;
public class Mistake1 {

     // Use a loop to compute a sum of squares
  public static void Main() {
    int sum = 0;              // Current value to square and add
    int count = 1;
    Console.Write("Enter the number of squares to sum: ");
    int high = int.Parse(Console.ReadLine());
    while (count <= high)
       sum += count*count;
    Console.WriteLine("The sum of the first {0} squares is {1}",
                      high, sum);
  }
}
```

Getting Information

When we run Example 3.8, we find that there is no output. Nothing happens, and the program does not terminate. We must abort the program. Reading the code, we see that the WriteLine statement comes after the while loop. Because the WriteLine statement never is executed, it seems like the while loop is not terminating. To see more clearly what is happening we add a WriteLine statement in the body of the loop

EXAMPLE 3.9 ▪ Mistake2.cs

```
/* Adds a WriteLine statement in the body of
 * the while loop of Example 3.8
 */

using System;
public class Mistake2 {

    // Use a loop to compute a sum of squares
  public static void Main() {
    int sum = 0;
    int count = 1;     // Current value to square and add
    Console.Write("Enter the number of squares to sum: ");
    int high = int.Parse(Console.ReadLine());
    while (count <= high) {                              // Note 1
      sum += count*count;
      Console.WriteLine("Sum is {0}", sum);
    }
    Console.WriteLine("The sum of the first {0} squares is {1}",
                      high, sum);
  }
}
```

Run

Sum is 1
Sum is 2
Sum is 3
…….(nonterminating)

Note 1: while (count <= high) {

We make the while loop body a block, enclosing the two statements in curly braces. We did not need the curly braces in Example 3.8 because the while loop body was a single assignment statement

The output of Example 3.9 continues until we abort the program. It is clear that the while loop does not terminate. Looking more closely at the code, we see that we forgot to increment count after adding the next square to the sum

EXAMPLE 3.10 Mistake3.cs

```
/* Modifies Example 3.9 to increment the count.
 */

using System;
public class Mistake3 {

    // Use a loop to compute a sum of squares
  public static void Main() {
    int sum = 0;
    int count = 1;        // Current value to square and add
    Console.Write("Enter the number of squares to sum: ");

  int high = int.Parse(Console.ReadLine());
  while (count <= high) {
    sum += count*count;
    count++;
  }
  Console.WriteLine("The sum of the first {0} squares is {1}",
                    high, sum);
  }
}
```

Run

Enter the number of squares to sum: 5
The sum of the first 5 squares is 55

Testing Carefully

The output looks correct. Checking it by hand, we see that the sum of the first five squares, 1 + 4 + 9 + 16 + 2 5, is indeed 55. It is tempting to conclude that

our program is correct, but we should never make such a conclusion based on one test case. Let us do some more testing, trying larger high values such as 100, 1000, and 10,000. The output of these tests is

```
Enter the number of squares to sum:  100
The sum of the first 100 squares is 338350

Enter the number of squares to sum:  1000
The sum of the first 1000 squares is 333833500

Enter the number of squares to sum:  10000
The sum of the first 10000 squares is -1624114088
```

Surely, something is wrong here. The sum of the first 10,000 squares is certainly not negative. Remember that the int type can hold values up to 2,147,483,647. Trying to store values larger than 2,147,483,647 will give spurious results.

We leave it to the exercises to determine the largest number of squares we can sum correctly using the program of Example 3.10. The debugging techniques we illustrated in these examples are:

- Read the code carefully to determine where the error might be located. In Example 3.8 we could tell from reading the code that the error was in the while loop
- Add WriteLine statements to get more information. The added WriteLine statement in Example 3.9 clearly demonstrated the error
- Test thoroughly. Even though our first result was correct, more extensive testing of Example 3.10 found an error

Using the Visual C# Debugger

Many development environments include a debugger, which is an application used to debug a program. We illustrate with the Visual C# debugger using Example 3.8.

After opening Visual C#, we click on *File*, *New Project* choosing *Console Applicaton* as the template. We enter a project name and click *OK*. Click *Project*, *Add Existing Item* and select *Mistake.cs*. In the Solution Explorer window, right click on *Program.cs* and click *delete* to remove it.

To compile, we click on *Build*, *Build Solution.* To see in detail how the program is executing, we execute it one step at a time, by clicking on *Debug*, *Step Over*. Choosing *Step Over* executes a method call without executing each line of the method. To execute each line of the method we would use *Step Into* instead. The *Step Into* choice is useful if the error occurs inside a method call.

Hovering the mouse over a variable name in the code will pop up a box containing the current value of that variable.

Figure 3.11 shows the Visual C# display after we first press *Debug*, *Step Over*. The lower-left section shows the values of the variables. The upper-left section shows the source code with an arrow pointed to the line to be executed next.

We can set a breakpoint by clicking in the margin next to the line at which we want to interrupt execution. We can then execute until we reach the breakpoint without having to execute each line up to that point. In this example we click to the left of the statement `sum += count*count;`

FIGURE 3.11 Debugging Mistake1.cs

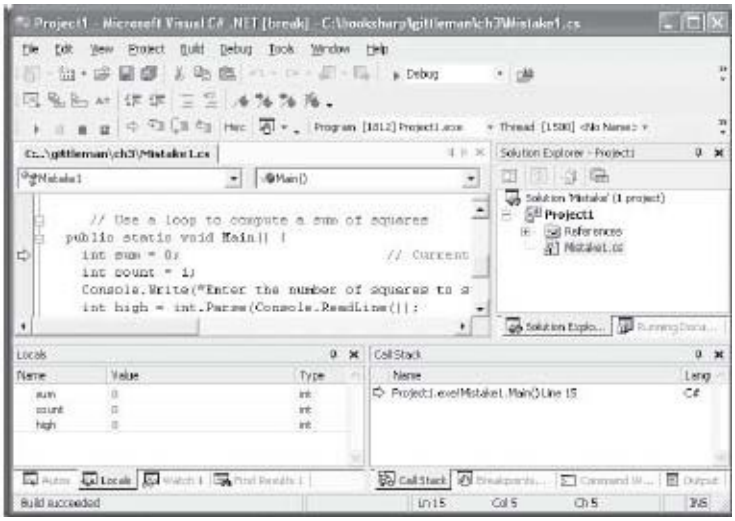
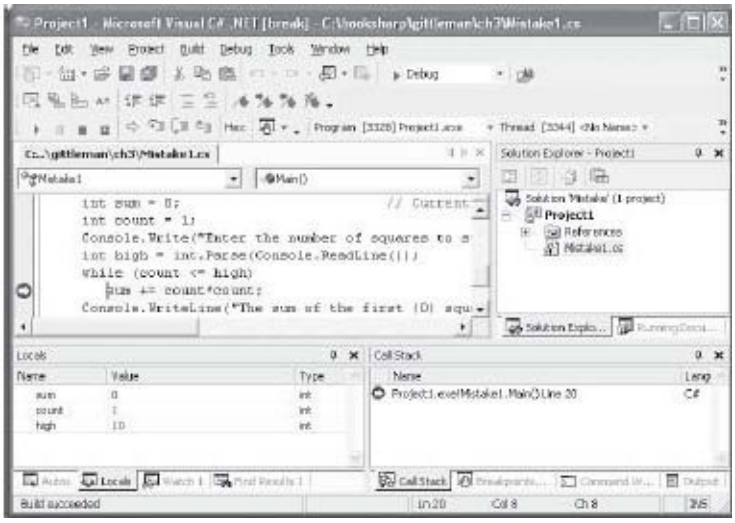


FIGURE 3.12 Corrected Mistake1.cs



Then we click on *Debug*, *Continue* to continue executing until we reach the breakpoint. A console window opens and we enter 10 as the number of squares to sum. Execution stops at the screen shown in Figure 3.12. The upper-left window shows that the next line to execute is our breakpoint. The value of sum is 0, count is 1, and high is 10

We now click on *Debug*, *Step Over* repeatedly to continue executing the program one line at a time. The first click changes sum to 1 and returns to the while test. Clicking *Debug*, *Step Over* two more times has the while test succeeding and sum changing to 2. We see that the program is not executing properly because we did not increment count. We can stop debugging and edit our program to add the line `count++;`

to the while loop. When we recompile and rerun the program we see that it now computes the sum as we expected

The BIG Picture

We debug a program to find and correct errors. Simple techniques include reading the program carefully, adding WriteLine statements to display values, and testing the program thoroughly. A debugger allows us to execute one statement at a time so we can watch changes in the values of variables. We can set a breakpoint to stop execution at a particular point

Try It Yourself ➤

Try It Yourself ➤

☑ Test Your Understanding

33. Run Example 3.10 to find the largest value, *n*, such that Example 3.10 finds the sum of the first *n* squares correctly.

34. Modify Example 3.10 to inform the user of the largest acceptable input in the prompt and to reject any input greater than that value.

SUMMARY

- In this chapter we develop the tools to solve problems using C#. To make our programs more flexible, we need to make decisions based on the value of a test condition that can be either true or false. C# provides the bool type, which has values True and False. We write our test conditions as relational or equality expressions. The relational expressions use the operators <, >, <=, and >= and produce bool values. The equality expressions use the operators == and ! = and also produce bool values. We can include arithmetic operators in our test conditions. C# will use precedence rules to help evaluate such expressions, with arithmetic expressions having higher precedence than relational expressions

- Once we know how to write a test condition, we use the if and if-else statements to make choices based on the result of a test condition. In the if statement we execute the next statement if the condition is true and skip it if the condition is false. The if-else statement gives us two alternative statements, one to execute if the test condition is true and the other to execute if the test condition is false. Each of these statements can be a simple statement or a code block enclosed by curly braces. Flow charts give a visual representation of the control flow

- The if and if-else statements give us choices in control flow. We discuss decimal numbers and the type double to add a choice of data in addition to the int type covered in Chapter 2. We can use scientific notation to express decimals, so we can write .000000645 as 6.45E-7. The type double represents decimal numbers with 15-digit accuracy. We can declare variables of type double and perform the usual arithmetic operations of +, -, *, and /

- C# displays numbers of type double using the F format for fixed-point notation and the E format for scientific notation. The G, or general, format chooses one or the other depending on the size of the number. We can specify the number of places after the decimal point

- If an operator, say +, has one int operand and one double operand, then C# will convert the int operand to a double, and add, producing a result of type double. C# will not automatically convert a double to an int, but we can use