Bit Manipulation

Objectives

- To understand the concept of bit manipulation.
- To be able to use bitwise operators.
- To be able to use class **`BitArray`** to perform bit manipulation.

Outline

O.1 Introduction

In this appendix, we present an extensive discussion of bit manipulation and the *bitwise operators* that enable it. We also discuss class **`BitArray`**, from which we create objects useful for manipulating sets of bits.

O.2 Bit Manipulation and the Bitwise Operators

C# provides extensive bit-manipulation capabilities for programmers who must work at the "bits-and-bytes" level. Operating systems, test-equipment software, networking software and many other kinds of software require that programmers communicate "directly with the hardware." In this section and the next, we discuss C#'s bit-manipulation capabilities. After introducing C#'s bitwise operators, we demonstrate the use of the operators in live-code examples.

Computers represent data internally as sequences of bits. Arithmetic Logic Units (ALUs), Central Processing Units (CPUs) and other pieces of hardware in a computer process data as bits or groups of bits. Each bit can assume either the value **`0`** or the value **`1`**. On all systems, a sequence of 8 bits forms a *byte*—the standard storage unit for a variable of type **`byte`**. Other data types require larger numbers of bytes for storage. The bitwise operators manipulate the bits of integral operands (i.e., **`sbyte`**, **`byte`**, **`char`**, **`short`**, **`ushort`**, **`int`**, **`uint`**, **`long`** and **`ulong`**).

Note that the discussion of bitwise operators in this section illustrates the binary representations of the integer operands. For a detailed explanation of the binary (also called base-2) number system, see Appendix B, Number Systems.

The operators *bitwise AND* (**`&`**), *bitwise inclusive OR* (**`|`**) and *bitwise exclusive OR* (**`^`**) operate similarly to their logical counterparts, except that the bitwise versions operate on the level of bits. The bitwise AND operator sets each bit in the result to 1 if the corresponding bits in both operands are 1 (Fig. O.2). The bitwise inclusive OR operator sets each bit in the result to 1 if the corresponding bits in either (or both) operand(s) are 1 (Fig. O.3). The bitwise exclusive OR operator sets each bit in the result to 1 if the corresponding bit in exactly one operand is 1 (Fig. O.4). Exclusive OR is also known as *XOR*.

The *left-shift* (**`<<`**) operator shifts the bits of its left operand to the left by the number of bits specified in its right operand. The *right-shift* (**`>>`**) operator shifts the bits in its left operand to the right by the number of bits specified in its right operand. If the left operand is negative, **`1`**s are shifted in from the left, whereas, if the left operand is positive **`0`**s are shifted in from the left. The bitwise *complement* (**`~`**) operator sets all **`0`** bits in its operand to **`1`** and all **`1`** bits to **`0`** in the result; this process sometimes is referred to as "taking the *one's complement* of the value." A detailed discussion of each bitwise operator appears in the examples that follow. The bitwise operators and their functions are summarized in Fig. O.1.

Operator	Name	Description
`&`	bitwise AND	Each bit in the result is set to `1` if the corresponding bits in the two operands are both `1`. Otherwise, the bit is set to `0`.
`|`	bitwise inclusive OR	Each bit in the result is set to `1` if at least one of the corresponding bits in the two operands is `1`. Otherwise, the bit is set to `0`.
`^`	bitwise exclusive OR	Each bit in the result is set to `1` if exactly one of the corresponding bits in the two operands is `1`. Otherwise, the bit is set to `0`.
`<<`	left shift	Shifts the bits of the first operand to the left by the number of bits specified by the second operand; fill from the right with `0` bits.
`>>`	right shift	Shifts the bits of the first operand to the right by the number of bits specified by the second operand. If the first operand is negative, `1`s are shifted in from the left; otherwise, `0`s are shifted in from the left.
`~`	complement	All `0` bits are set to `1`, and all `1` bits are set to `0`.

Fig. O.1 Bitwise operators.

Bit 1	Bit 2	Bit 1 & Bit 2
0	0	0
1	0	0
0	1	0
1	1	1

Fig. O.2 Results of combining two bits with the bitwise AND operator (`&`).

Bit 1	Bit 2	Bit 1 | Bit 2
0	0	0
1	0	1
0	1	1
1	1	1

Fig. O.3 Results of combining two bits with the bitwise inclusive OR operator (`|`).

Bit 1	Bit 2	Bit 1 ^ Bit 2
0	0	0
1	0	1
0	1	1
1	1	0

Fig. O.4 Results of combining two bits with the bitwise exclusive OR operator (**^**).

When using the bitwise operators, it is useful to display values in their binary representations to illustrate the effects of these operators. In Fig. O.5, integers are displayed in their binary representations as groups of eight bits each. Method **GetBits** (lines 67–91) of class **PrintBits** uses the bitwise AND operator (line 79) to combine variable **number** with variable **displayMask**. Often, the bitwise AND operator is used with a *mask* operand—an integer value with specific bits set to **1**. Masks hide some bits in a value and select other bits. **GetBits** assigns mask variable **displayMask** the value **1 << 31** (**10000000 00000000 00000000 00000000**). The left-shift operator shifts the value **1** from the low-order (rightmost) bit to the high-order (leftmost) bit in **displayMask** and fills in **0** bits from the right. Because the second operand is 31, 31 bits (each is **0**) are filled in from the right. The word "fill" in this context, means that we add a bit to the right end, and delete one from the left end. Every time we add a **0** to the right end, we remove the bit at the left end.

The statement on line 79 determines whether a **1** or a **0** should be appended to **StringBuilder output** for the leftmost bit of variable **number**. For this example, assume that **number** contains **11111** (**00000000 00000000 00101011 01100111**). When **number** and **displayMask** are combined using **&**, all the bits except the high-order bit in variable **number** are "masked off" (hidden), because any bit "ANDed" with **0** yields **0**. If the leftmost bit is **0**, **number & displayMask** evaluates to **0**, and **0** is appended; otherwise, **1** is appended. Line 83 then left shifts variable **val** one bit with the expression **number <<= 1**. (This is equivalent to **number = number << 1**.) These steps are repeated for each bit in variable **number**. At the end of method **GetBits**, line 89 converts the **StringBuilder** to a **string** and returns it from the method.

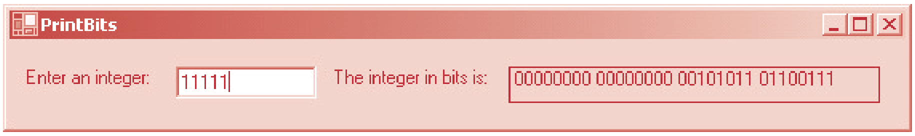
```
1   // Fig O.5: PrintBits.cs
2   // Printing the bits that constitute an integer.
3
4   using System;
5   using System.Drawing;
6   using System.Collections;
7   using System.ComponentModel;
8   using System.Windows.Forms;
9   using System.Data;
10  using System.Text;
11
12
```

Fig. O.5 Displaying the bit representation of an integer. (Part 1 of 3.)

```
// displays bit representation of user input
public class PrintBits : System.Windows.Forms.Form
{
   private System.Windows.Forms.Label promptLabel;
   private System.Windows.Forms.Label viewLabel;

   // for user input
   private System.Windows.Forms.TextBox inputTextBox;

   // bit representation displayed here
   private System.Windows.Forms.Label displayLabel;

   private System.ComponentModel.Container components = null;

   // default constructor
   public PrintBits()
   {
      InitializeComponent();
   }

   // Visual Studio .NET generated code

   [STAThread]
   static void Main()
   {
      Application.Run( new PrintBits() );
   }

   // process integer when user presses Enter
   private void inputTextBox_KeyDown(
      object sender, System.Windows.Forms.KeyEventArgs e )
   {
      // if user pressed Enter
      if ( e.KeyCode == Keys.Enter )
      {
         // test whether user enetered an integer
         try
         {
            displayLabel.Text = GetBits(
               Convert.ToInt32( inputTextBox.Text ) );
         }

         // if value is not integer, exception is thrown
         catch ( FormatException )
         {
            MessageBox.Show( "Please Enter an Integer",
               "Error", MessageBoxButtons.OK,
               MessageBoxIcon.Error );
         }
      }

   } // end method inputTextBox_KeyDown

```

Fig. O.5 Displaying the bit representation of an integer. (Part 2 of 3.)

```
    // convert integer to its bit representation
    public string GetBits( int number )
    {
       int displayMask = 1 << 31;

       StringBuilder output = new StringBuilder();

       // get each bit, add space every 8 bits
       // for display formatting
       for ( int c = 1; c <= 32; c++ )
       {
          // append 0 or 1 depending on result of masking
          output.Append(
             ( number & displayMask ) == 0 ? "0" : "1" );

          // shift left so that mask will find bit of
          // next digit during next iteration of loop
          number <<= 1;

          if ( c % 8 == 0 )
             output.Append( " " );
       }

       return output.ToString();

    } // end method GetBits

 } // end class PrintBits
```



```
11111
00000000 00000000 00101011 01100111
```

Fig. O.5 Displaying the bit representation of an integer. (Part 3 of 3.)

Common Programming Error 24.1

*Using the logical AND operator (**&&**) in place of the bitwise AND operator (**&**) is a common programming error.*

Common Programming Error 24.2

*Using the logical OR operator (**||**) in place of the bitwise inclusive OR operator (**|**) is a common programming error.*

The program in Fig. O.6 demonstrates the bitwise AND operator, the bitwise inclusive OR operator, the bitwise exclusive OR operator and the bitwise complement operator. The program uses method **GetBits**, which returns a **string** which contains the bit representation of its integer argument. Users enter values into **TextBox**es and press the button corresponding to the operation they would like to test. The program displays the result in both integer and bit representations.

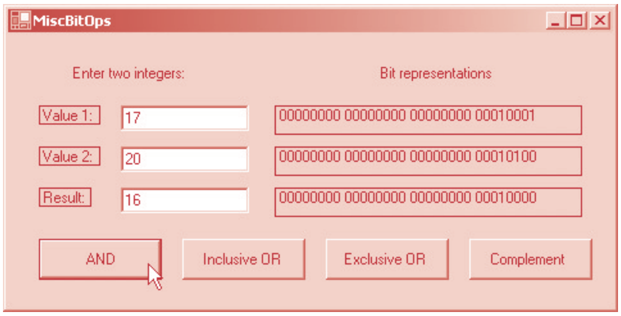
```
// Fig. O.6: BitOperations.cs
// A class that demonstrates miscellaneous bit operations

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Text;

// allows user to test bit operators
public class BitOperations : System.Windows.Forms.Form
{
   private System.Windows.Forms.Label promptLabel;
   private System.Windows.Forms.Label representationLabel;
   private System.Windows.Forms.Label value1Label;
   private System.Windows.Forms.Label value2Label;
   private System.Windows.Forms.Label resultLabel;

   // display bit reprentations
   private System.Windows.Forms.Label bit1Label;
   private System.Windows.Forms.Label bit2Label;
   private System.Windows.Forms.Label resultBitLabel;

   // allow user to perform bit operations
   private System.Windows.Forms.Button andButton;
   private System.Windows.Forms.Button inclusiveOrButton;
   private System.Windows.Forms.Button exclusiveOrButton;
   private System.Windows.Forms.Button complementButton;

   // user inputs two integers
   private System.Windows.Forms.TextBox bit1TextBox;
   private System.Windows.Forms.TextBox bit2TextBox;

   private System.Windows.Forms.TextBox resultTextBox;

   private int value1, value2;

   private System.ComponentModel.Container components = null;

   // default constructor
   public BitOperations()
   {
      InitializeComponent();
   }

   // Visual Studio .NET generated code

   [STAThread]
   static void Main()
   {
```

Fig. O.6 Demonstrating the bitwise AND, bitwise inclusive OR, bitwise exclusive OR and bitwise complement operators. (Part 1 of 4.)

```
         Application.Run( new BitOperations() );
      }

      // AND
      private void andButton_Click(
         object sender, System.EventArgs e )
      {
         SetFields();

         // update resultTextBox
         resultTextBox.Text =
            string.Format( "{0}", value1 & value2 );

         resultBitLabel.Text = GetBits( value1 & value2 );
      }

      // inclusive OR
      private void inclusiveOrButton_Click(
         object sender, System.EventArgs e )
      {
         SetFields();

         // update resultTextBox
         resultTextBox.Text =
            string.Format( "{0}", value1 | value2 );
         resultBitLabel.Text = GetBits( value1 | value2 );
      }

      // exclusive OR
      private void exclusiveOrButton_Click(
         object sender, System.EventArgs e )
      {
         SetFields();

         // update resultTextBox
         resultTextBox.Text =
            string.Format( "{0}", value1 ^ value2 );
         resultBitLabel.Text = GetBits( value1 ^ value2 );
      }

      // complement of first integer
      private void complementButton_Click(
         object sender, System.EventArgs e )
      {
         value1 = Convert.ToInt32( bit1TextBox.Text );
         bit1Label.Text = GetBits( value1 );

         // update resultTextBox
         resultTextBox.Text = string.Format( "{0}", ~value1 );
         resultBitLabel.Text = GetBits( ~value1 );
      }

```

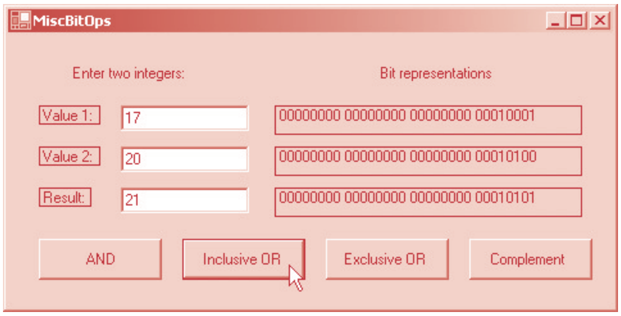
Fig. O.6 Demonstrating the bitwise AND, bitwise inclusive OR, bitwise exclusive OR and bitwise complement operators. (Part 2 of 4.)

```
    // convert integer to its bit representation
    private string GetBits( int number )
    {
       int displayMask = 1 << 31;

       StringBuilder output = new StringBuilder();

       // get each bit, add space every 8 bits
       // for display formatting
       for ( int c = 1; c <= 32; c++ )
       {
          // append 0 or 1 depending on the result of masking
          output.Append(
             ( number & displayMask ) == 0 ? "0" : "1" );

          // shift left so that mask will find bit of
          // next digit in the next iteration of loop
          number <<= 1;

          if ( c % 8 == 0 )
             output.Append( " " );
       }

       return output.ToString();

    } // end method GetBits

    // set fields of Form
    private void SetFields()
    {
       // retrieve input values
       value1 = Convert.ToInt32( bit1TextBox.Text );
       value2 = Convert.ToInt32( bit2TextBox.Text );

       // set labels to display bit representations of integers
       bit1Label.Text = GetBits( value1 );
       bit2Label.Text = GetBits( value2 );
    }

 } // end class BitOperations
```



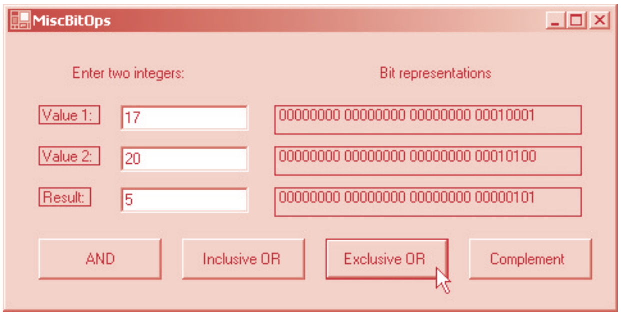
```
Value 1:  17   00000000 00000000 00000000 00010001
Value 2:  20   00000000 00000000 00000000 00010100
Result:   16   00000000 00000000 00000000 00010000
```

Fig. O.6 Demonstrating the bitwise AND, bitwise inclusive OR, bitwise exclusive OR and bitwise complement operators. (Part 3 of 4.)



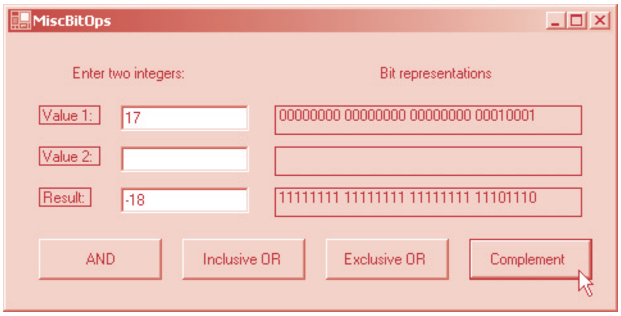
Enter two integers: Bit representations

Value 1:	17	00000000 00000000 00000000 00010001
Value 2:	20	00000000 00000000 00000000 00010100
Result:	21	00000000 00000000 00000000 00010101



Enter two integers: Bit representations

Value 1:	17	00000000 00000000 00000000 00010001
Value 2:	20	00000000 00000000 00000000 00010100
Result:	5	00000000 00000000 00000000 00000101



Enter two integers: Bit representations

Value 1:	17	00000000 00000000 00000000 00010001
Value 2:		
Result:	-18	11111111 11111111 11111111 11101110

Fig. O.6 Demonstrating the bitwise AND, bitwise inclusive OR, bitwise exclusive OR and bitwise complement operators. (Part 4 of 4.)

The first output window of Fig. O.6 shows the results of combining the value **`17`** and the value **`20`** using the bitwise AND operator (**`&`**); the result is **`16`**. The second output window shows the results of combining the value **`17`** and the value **`20`** using the bitwise OR operator; the result is **`21`**. The third output shows the results of combining the value **`17`** and the value **`20`** using the exclusive OR operator; the result is **`5`**. The fourth output window shows the results of taking the one's complement of the value **`17`**. The result is **`-18`**.

The program in Fig. O.7 demonstrates the use of the left-shift operator (**`<<`**) and the right-shift operator (**`>>`**). Method **`GetBits`** returns a **`string`** containing the bit representation of an integer value passed to it as an argument. When users enter an integer in a **`TextBox`** and press *Enter*, the program displays the bit representation of the specified integer in a **`Label`**.

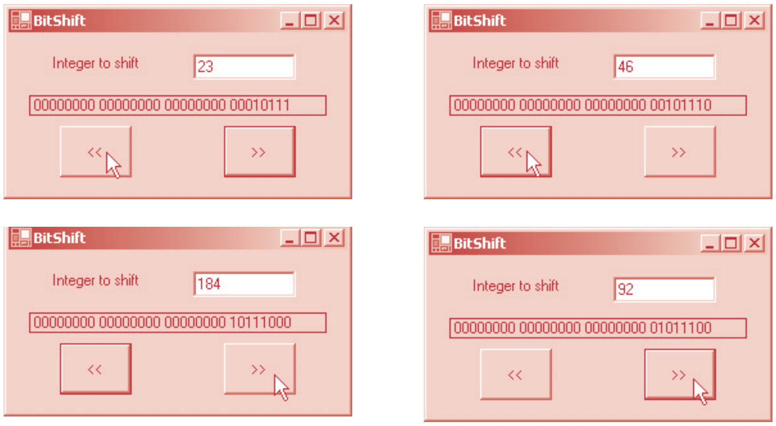
```
// Fig O.7: BitShift.cs
// Demonstrates bitshift operators.

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Text;

// shifts bits to the right or left
public class BitShift : System.Windows.Forms.Form
{
   private System.Windows.Forms.Label inputLabel;

   // accepts user input
   private System.Windows.Forms.TextBox inputTextBox;

   // displays integer in bits
   private System.Windows.Forms.Label displayLabel;
   private System.Windows.Forms.Button rightButton;
   private System.Windows.Forms.Button leftButton;

   private System.ComponentModel.Container components = null;

   // default constructor
   public BitShift()
   {
      InitializeComponent();
   }

   // Visual Studio .NET generated code

   [STAThread]
   static void Main()
   {
      Application.Run( new BitShift() );
   }

   // process user input
   private void inputTextBox_KeyDown(
      object sender, System.Windows.Forms.KeyEventArgs e )
   {
      if ( e.KeyCode == Keys.Enter )
         displayLabel.Text =
            GetBits( Convert.ToInt32( inputTextBox.Text ) );
   }

   // do left shift
   private void leftButton_Click(
      object sender, System.EventArgs e )
   {
```

Fig. O.7 Using the bitshift operators. (Part 1 of 3.)

```
         // retrieve user input
         int number = Convert.ToInt32( inputTextBox.Text );

         // do left shift operation
         number <<= 1;

         // convert to integer and display in textbox
         inputTextBox.Text = number.ToString();

         // display bits in label
         displayLabel.Text = GetBits( number );
      }

      // do right shift
      private void rightButton_Click(
         object sender, System.EventArgs e )
      {
         // retrieve user input
         int number = Convert.ToInt32( inputTextBox.Text );

         // do right shift operation
         number >>= 1;

         // convert to integer and display in textbox
         inputTextBox.Text = number.ToString();

         // display bits in label
         displayLabel.Text = GetBits( number );
      }

      // convert integer to its bit representation
      private string GetBits( int number )
      {
         int displayMask = 1 << 31;

         StringBuilder output = new StringBuilder();

         // get each bit, add space every 8 bits
         // for display formatting
         for ( int c = 1; c <= 32; c++ )
         {
            // append a 0 or 1 depending on the result of masking
            output.Append(
               ( number & displayMask ) == 0 ? "0" : "1" );

            // shift left so that mask will find bit of
            // next digit during next iteration of loop
            number <<= 1;

            if ( c % 8 == 0 )
               output.Append( " " );
         }

```

Fig. O.7 Using the bitshift operators. (Part 2 of 3.)

```
         return output.ToString();

      } // end method GetBits

   } // end class BitShift
```



BitShift
Integer to shift 23
00000000 00000000 00000000 00010111

BitShift
Integer to shift 46
00000000 00000000 00000000 00101110

BitShift
Integer to shift 184
00000000 00000000 00000000 10111000

BitShift
Integer to shift 92
00000000 00000000 00000000 01011100

Fig. O.7 Using the bitshift operators. (Part 3 of 3.)

Each shift operator has its own button on the application's GUI. As a user clicks each button, the bits in the integer shift left or right by one bit. The **`TextBox`** and **`Label`** display the new integer value and new bit representation, respectively.

The left-shift operator (**`<<`**) shifts the bits of its left operand to the left by the number of bits specified in its right operand. The rightmost bits are replaced with **`0`**s; **`1`**s shifted off the left are lost. The first two output windows in Fig. O.7 demonstrate the left-shift operator. To produce the output, the user entered the value 23 and clicked the left-shift button, resulting in the value **`46`**.

The right-shift operator (**`>>`**) shifts the bits of its left operand to the right by the number of bits specified in its right operand. **`0`**s replace vacated bits on the left side if the number is positive, and **`1`**s replace the vacated bits if the number is negative. Any **`1`**s shifted off the right are lost. The third and fourth output windows depict the result of shifting **`184`** to the right once.

Each bitwise operator (except the bitwise complement operator) has a corresponding assignment operator. Figure O.8 describes these *bitwise assignment operators*, which are used in a manner similar to the arithmetic assignment operators introduced in Chapter 3.

Bitwise assignment operators	
`&=`	Bitwise AND assignment operator.
`|=`	Bitwise inclusive OR assignment operator.
`^=`	Bitwise exclusive OR assignment operator.

Fig. O.8 Bitwise assignment operators. (Part 1 of 2.)

Bitwise assignment operators	
`<<=`	Left-shift assignment operator.
`>>=`	Right-shift assignment operator.

Fig. O.8 Bitwise assignment operators. (Part 2 of 2.)

O.3 Class `BitArray`

Class ***`BitArray`*** facilitates the creation and manipulation of *bit sets*, which programmers often use to represent a set of *boolean flags.* A boolean flag is a variable that keeps track of a certain boolean decision. **`BitArray`**s are resizable dynamically—more bits can be added once a **`BitArray`** object is created, causing the object to grow to accommodate the additional bits.

Class **`BitArray`** provides several constructors, one of which accepts an **`int`** as an argument. The **`int`** specifies the number of bits that the **`BitArray`** represents, all of which are initially set to **`false`**.

Method ***`Set`*** of **`BitArray`** can change the value of an individual bit; it accepts the index of the bit to change and its new **`bool`** value. Class **`BitArray`** also includes an indexer that allows us to get and set individual bit values. The indexer returns **`true`** if the specified bit is on (i.e., the bit has value 1) and returns **`false`** otherwise (i.e., the bit has value 0 or "off").

Class **`BitArray`** method ***`And`*** performs a bitwise AND between two **`BitArray`**s and returns the **`BitArray`** result of the operation. Methods ***`Or`*** and ***`Xor`*** perform bitwise inclusive OR and bitwise exclusive OR operations, respectively. Class **`BitArray`** also provides a ***`Length`*** property, which returns the number of elements in the **`BitArray`**.

Figure O.9 implements the *Sieve of Eratosthenes*, which is a technique for finding prime numbers. A prime number is an integer that is divisible evenly only by itself and one. The Sieve of Eratosthenes operates as follows:

a) Create an array with all elements initialized to 1 (true). Array elements with prime subscripts remain 1. All other array elements eventually are set to 0.
b) Starting with array subscript 2 (subscript 1 must not be prime), every time an array element is found with a value of 1, loop through the remainder of the array and set to 0 every element whose subscript is a multiple of the subscript for the element with value 1. For example, for array subscript 2, all elements after 2 in the array that are multiples of 2 are set to 0 (subscripts 4, 6, 8, 10, etc.); for array subscript 3, all elements after 3 in the array that are multiples of 3 are set to 0 (subscripts 6, 9, 12, 15, etc.); and so on.

At the end of this process, the subscripts of the array elements that are one are prime numbers. The list of prime numbers can then be displayed by locating and printing these subscripts.

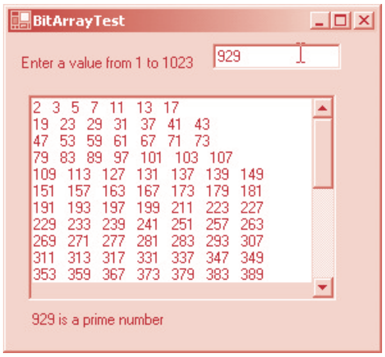
```
// Fig O.9: BitArrayTest.cs
// Demonstrates BitArray class.

```

Fig. O.9 Sieve of Eratosthenes. (Part 1 of 3.)

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

// implements Sieve of Eratosthenes
public class BitArrayTest : System.Windows.Forms.Form
{
   private System.Windows.Forms.Label promptLabel;

   // user inputs integer
   private System.Windows.Forms.TextBox inputTextBox;

   // display prime numbers
   private System.Windows.Forms.TextBox outputTextBox;

   // displays whether input integer is prime
   private System.Windows.Forms.Label displayLabel;

   private BitArray sieve;

   private System.ComponentModel.Container components = null;

   // default constructor
   public BitArrayTest()
   {
      InitializeComponent();

      // create BitArray and set all bits to true
      sieve = new BitArray( 1024 );
      sieve.SetAll( true );

      int finalBit = ( int ) Math.Sqrt( sieve.Length );

      // perform sieve operation
      for ( int i = 2; i < finalBit; i++ )
         if ( sieve.Get( i ) )
            for ( int j = 2 * i; j < sieve.Length; j += i )
               sieve.Set( j, false );

      int counter = 0;

      // display prime numbers
      for ( int i = 2; i < sieve.Length; i++ )
         if ( sieve.Get( i ) )
            outputTextBox.Text += i +
               ( ++counter % 7 == 0 ? "\r\n" : "   " );
   }

   // Visual Studio .NET generated code

```

Fig. O.9 Sieve of Eratosthenes. (Part 2 of 3.)

```
57       [STAThread]
58       static void Main()
59       {
60          Application.Run( new BitArrayTest() );
61       }
62
63       private void inputTextBox_KeyDown(
64          object sender, System.Windows.Forms.KeyEventArgs e )
65       {
66          // if user pressed Enter
67          if ( e.KeyCode == Keys.Enter )
68          {
69             int number = Convert.ToInt32( inputTextBox.Text );
70
71             // if sieve is true at index of integer
72             // input by user, then number is prime
73             if ( sieve.Get( number ) )
74                displayLabel.Text = number + " is a prime number";
75             else
76                displayLabel.Text =
77                   number + " is not a prime number";
78          }
79       } // end method inputTextBox_KeyDown
80
81    } // end class BitArrayTest
```



```
2  3  5  7  11  13  17
19  23  29  31  37  41  43
47  53  59  61  67  71  73
79  83  89  97  101  103  107
109  113  127  131  137  139  149
151  157  163  167  173  179  181
191  193  197  199  211  223  227
229  233  239  241  251  257  263
269  271  277  281  283  293  307
311  313  317  331  337  347  349
353  359  367  373  379  383  389
```

929 is a prime number

Fig. O.9 Sieve of Eratosthenes. (Part 3 of 3.)

We use a **BitArray** to implement the algorithm. The program displays the prime numbers in the range 1–1023 in a **TextBox**. The program also provides a **TextBox** in which users can type any number from 1–1023 to determine whether that number is prime. (In which case, it displays a message indicating that the number is prime.)

The statement on line 35 creates a **BitArray** of **1024** bits. **BitArray** method ***SetAll*** sets all the bits to **true** on line 36; then, lines 41–44 determine all prime numbers occurring between 1 and 1023. The integer **finalBit** determines when the algorithm is complete.

When the user inputs a number and presses *Enter*, line 73 tests whether the input number is prime. This line uses the ***Get*** method of class **BitArray**, which takes a number and returns the value of that bit in the array. Lines 74 and 76 print an appropriate response.

SUMMARY

- Computers represent data internally as sequences of bits. Each bit can assume the value **0** or the value **1**.
- On all systems, a sequence of 8 bits forms a byte—the standard storage unit for a variable of type **byte**. Other data types require larger numbers of bytes for storage.
- The bitwise AND operator sets each bit in the result to 1 if the corresponding bits in both operands are 1.
- The bitwise inclusive OR operator sets each bits in the result to 1 if the corresponding bit in either (or both) operand(s) are 1.
- The bitwise exclusive OR operator sets each bit in the result to 1 if the corresponding bit in exactly one operand is 1. Exclusive OR is also known as XOR.
- The left-shift (**<<**) operator shifts the bits of its left operand to the left by the number of bits specified in its right operand.
- The right-shift (**>>**) operator shifts the bits in its left operand to the right by the number of bits specified in its right operand. If the left operand is negative, **1**s are shifted in from the left, whereas, if the left operand is positive, **0**s are shifted in from the left.
- The bitwise complement (**~**) operator sets all **0** bits in its operand to **1** in the result and sets all **1** bits to **0** in the result; this process is sometimes referred to as "taking the one's complement of the value."
- Often, the bitwise AND operator is used with a mask operand—an integer value with specific bits set to **1**. Masks hide some bits in a value and select other bits.
- Each bitwise operator (except the bitwise complement operator) has a corresponding assignment operator.
- Class **BitArray** facilitates the creation and manipulation of bit sets, which programmers often use to represent a set of boolean flags. A boolean flag is a variable that keeps track of a certain boolean decision.
- **BitArray**s are resizable dynamically—more bits can be added once a **BitArray** object is created, causing the object to grow to accommodate the additional bits.
- Method **Set** of **BitArray** can change the value of an individual bit—it accepts the index of the bit to change and the **bool** value to which the bit should change.
- **BitArray** method **And** performs a bitwise AND between two **BitArray**s. It returns the **BitArray** that is the result of performing this operation. Methods **Or** and **Xor**, perform bitwise inclusive OR and bitwise exclusive OR, respectively.
- **BitArray** method **SetAll** sets all the bits in the **BitArray** to **true**.

TERMINOLOGY

& (bitwise AND)
&= (bitwise AND assignment operator)
<< (left-shift operator)
<<= (left-shift assignment operator)
>> (right-shift operator)
>>= (right-shift assignment operator)
^ (bitwise exclusive OR)
^= (bitwise exclusive OR assignment operator)
| (bitwise inclusive OR)
|= (bitwise inclusive OR assignment operator)
~ (bitwise complement operator)
ALU (Arithmetic Logic Unit)
And method of class **BitArray**
Arithmetic Logic Unit (ALU)
base-2 number system
binary number system
binary representation
bit
bit manipulation
bit mask

bit representation of an integer
bit set
BitArray class
BitArray indexer
bit representation of an integer
bit set
BitArray class
BitArray indexer
BitArrayTest.cs
BitOperations.cs
bits-and-bytes level
BitShift.cs
bitwise AND (**&**)
bitwise AND assignment operator (**&=**)
bitwise assignment operators
bitwise complement operator (**~**)
bitwise exclusive OR (**^**)
bitwise exclusive OR assignment operator (**^=**)
bitwise inclusive OR (**|**)
bitwise inclusive OR assignment operator (**|=**)
bitwise operators
boolean decision
boolean flag
byte
byte primitive data type
Central Processing Unit (CPU)
Get method of class **BitArray**
high-order bit
left-shift assignment operator (**<<=**)
left-shift operator (**<<**)
Length property of class **BitArray**
low-order bit
one's complement (**~**)
Or method of class **BitArray**
prime number
right-shift assignment operator (**>>=**)
right-shift operator (**>>**)
Set method of class **BitArray**
SetAll method of **BitArray** class
Sieve of Eratosthenes
XOR (exclusive OR)
Xor method of class **BitArray**