CS109
Loops and Bitmap Graphics

Last time we looked at how to use if-then statements to control the flow of a program. In this section we will look at different ways to repeat blocks of statements. Such repetitions are called loops and are a powerful way to perform some task over and over again that would typically be too much work to do by hand. There are several ways to construct loops. We will examine the while and for loop constructs here.

Conceptually, we have two categories of loops. Pre-test loops tests to see if some condition is true before executing the loop statement. This is also called an entrance-controlled loop. The **Do-While** and **For** loop structures are pretest loops.

In a posttest loop, the condition is evaluated at the end of the repeating section of code. This is also called an exit-controlled loop. The **Do/Loop Until** construct is a posttest loop.

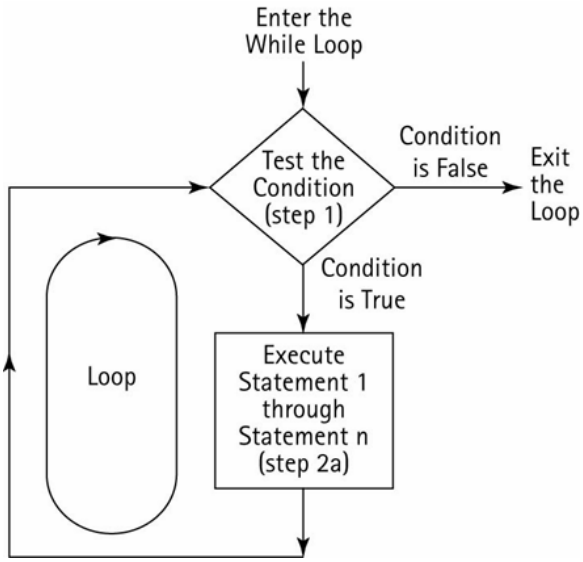
Do While Loop

The while loop allows you to direct the computer to execute the statement in the body of the loop as long as the expression within the parentheses evaluates to true. The format for the while loop is:

```
Do While (boolean_expression)
        statement1;
        …
        statement N;
Loop
```

As long as the Boolean expression evaluates to true, statements 1 through N will continue to be executed. Generally one of these statements will eventually make the Boolean expression become false, and the loop will exit.

In terms of a flowchart, the while loop behaves as follows:



An alternate way to write a while loop is as While loop. The syntax is:

```
While (boolean_condition)
        Statement 1
        …
        Statement N
End While
```

Both formats are equivalent. The latter format more closely matches other programming languages.

Here is an example of a while loop that prints the numbers from 1 to 10:

```
Dim i As Integer = 0
Do While (i <= 10)
      Console.WriteLine(i)
      i=i+1
Loop
```

If we wanted to print out 1,000,000 numbers we could easily do so by changing the loop! Without the while loop, we would need 1,000,000 different WriteLine statements, certainly an unpleasant task for a programmer. Similarly, you might recall an earlier example where we scored a quiz. If there were hundreds of questions in the quiz, it would be much better to score everything using a loop.

There are two types of while loops that we can construct. The first is a *count-based* loop, like the one we just used above. The loop continues incrementing a counter each time, until the counter reaches some maximum number. The second is an event-based loop, where the loop continues indefinitely until some event happens that makes the loop stop. Here is an example of an event-based loop:

```
Dim i As Integer = 0
Dim intSum As Integer = 0
While (i <> -9999)
    i = CInt(InputBox("Enter an integer, -9999 to stop"))
    If (i <> -9999) Then
        intSum = intSum + i
    End If
End While
Console.WriteLine("The total is " & intSum)
```

This loop will input a number and add it to sum as long as the number entered is not -9999. Once –9999 is entered, the loop will exit and the sum will be printed. This is an event-based loop because the loop does not terminate until some event happens – in this case, the special value of –9999 is entered. This value is called a *sentinel* because it signals the end of input. Note that it becomes possible to enter the sentinel value as data, so we have to make sure we check for this if we don't want it to be added to the sum.

What is wrong with the following code? Hint: It results in what is called an infinite loop.

```
Dim x as Integer = 1
Dim y as Integer = 1

Do While (x<=10)
      Console.WriteLine(y)
      y=y+1
Loop
```

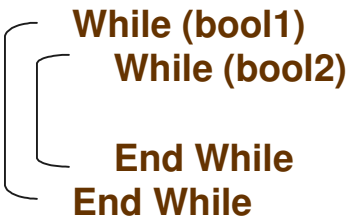
Exercise: Write a program that outputs all 99 stanzas of the "99 bottles of beer on the wall" song.

For example, the song will initially start as:

> 99 bottles of beer on the wall, 99 bottles of beer,
> take one down, pass it around,
> 98 bottles of beer on the wall.

Write a loop so that we can output the entire song, starting from ninety-nine and counting down to zero.

It is also possible to put a loop inside a loop. You really have no restrictions about the type of statements that can go in a loop! This type of construct is called a nested loop. The inner loop must be fully contained inside the outer loop:



Example: What is the output of this code?

```
i = 0
Do While i < 6
    j = 0
    Do While j < i
        Console.Write("*")
        j = j + 1
    Loop
    Console.WriteLine()
    i = i + 1
Loop
```

Nested loops are quite common, especially for processing tables of data.

Loop Until

It turns out that we can do all of the looping we need with the do while loop. However, there are a number of other looping constructs that make it easier to write certain kinds of loops than others. Consider the loop-until loop, which has the following format:

```
Do
        statement1
        …
        statement N
Loop Until (Boolean_condition)
```

The Loop Until executes all of the statements, 1-N, first. Then, if the Boolean condition is true, the loop ends and the program continues to execute whatever comes after the loop. However, if the Boolean condition is false, the loop will be executed again starting at the beginning. With the Loop Until, the computer **always** executes the body of the loop at least once before it checks the Boolean condition. In the while-do loop, the Boolean condition is checked **first**. If it is false then the while loop's body is never executed.

For example, we could rewrite the following While Loop as a Loop Until:

```
Do While (Boolean_Condition)
        Statements
Loop
```

Into:

```
If (Boolean_condition)
        Do
                Statements
        Loop Until (Not (Boolean_condition))
End If
```

We could rewrite the following Loop Until as a While Loop:

```
Do
        Statements
Loop Until (Boolean_condition)
```

Into:

```
Statements
Do While (Not (Boolean_condition))
        Statements
Loop
```

As an example, let's convert the while loop we wrote to input numbers into a loop-until.

```
Dim i As Integer = 0
Dim intSum As Integer = 0
Do
    i = CInt(InputBox("Enter an integer, -9999 to stop"))
    If (i <> -9999) Then
        intSum += i
    End If
Loop Until (i = -9999)
Console.WriteLine("The total is " & intSum)
```

Note that in the while loop we continue while i<>-9999. In this case, we write the loop as continuing until i=-9999, which is the opposite of i<>-9999. The special value -9999 is called a *sentinel*.

Another place where a do-until loop is useful is to print menus and check for valid input:

```
Dim i As Integer
Do
    i = CInt(InputBox("Enter 1 for task 1, and 2 for task 2", _
          "Main Menu"))
Loop Until ((i = 1) Or (i = 2))
```

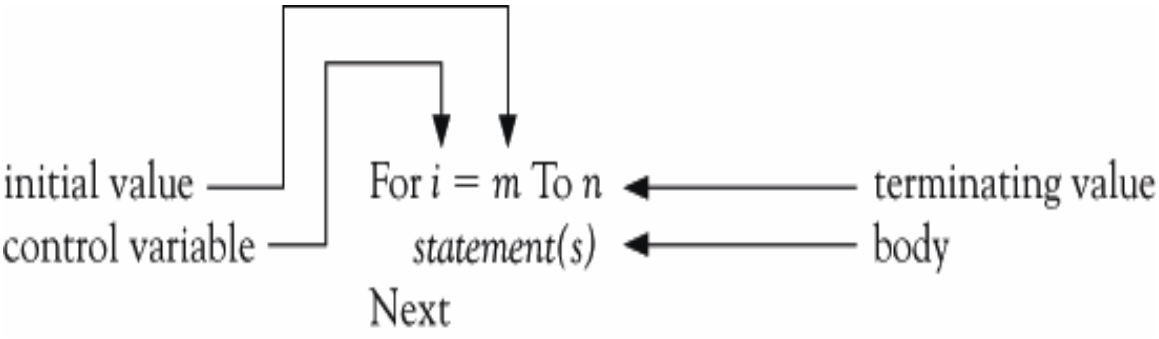
This loop will continue as long as the user types in something that is neither '1' nor '2'.

VB.NET allows for the use of either the While keyword or the Until keyword at the top or the bottom of a loop. As we have seen above, when using a While we continue to loop as long as the Boolean condition is true. When using a Until we continue to loop as long as the Boolean condition is false.

We will only use While at the top and Until at the bottom as this is a fairly standard convention in Visual Basic.

The For Loop

The for loop is a compact way to initialize variables, execute the body of a loop, and change the contents of variables. It is typically used when we know how many times we want the loop to execute – i.e. a counter controlled loop. The syntax is shown below:



This code is equivalent to the following While loop:

```
i = m
Do While (i <= n)
      Statement(s)
      i = i + 1
Loop
```

The basic for loop counts over the loop control variable, i, starting at value m and ending at value n.

Here is our loop to print ten numbers as a for loop:

```
Dim i As Integer
For i = 1 To 10
    Console.WriteLine(i)
Next
```

Suppose the Anchorage population is 300,000 in the year 2002 and is growing at the rate of 3 percent per year. The following for loop shows the population each year until 2006:

```
Dim pop As Integer = 300000
Dim yr As Integer
For yr = 2002 to 2006
      Console.WriteLine(yr & "  pop=" & pop)
      Pop += 0.03 * pop
Next
```

Optionally, we can add the keyword **Step** followed by a value at the end of the For line. This specifies the value that the index variable should be changed each loop iteration. If this is left off, we have seen that the loop is incremented by 1. Here is the new format:

For i = m to n **Step s**
    Statement(s)
Next

Instead of setting **i = i+1** at the end of the for loop, this sets **i = i + s** at the end of the loop. We can use this construct to count backwards or forwards in amounts not equal to 1.

The following prints out the numbers from 10 down to 1:

```
Dim i As Integer
For i = 10 To 1 Step -1
    Console.WriteLine(i)
Next
```

The following shows one way to reverse a string:

```
Dim sOriginal, sReverse As String
Dim j As Integer
sOriginal = "Kenrick"
sReverse = ""
For j = sOriginal.Length() - 1 To 0 Step -1
    sReverse &= sOriginal.Substring(j, 1)
Next
Console.WriteLine(sReverse)
```

This sums the odd integers between 1 and 10:

```
Dim i As Integer
Dim s As Integer = 0
For i = 1 To 10 Step 2
    s += i
Next
Console.WriteLine(s)
```

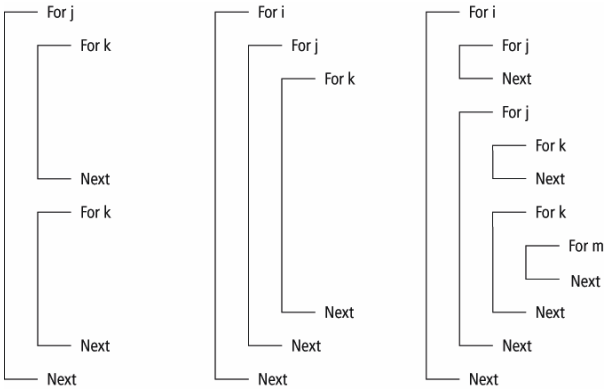
Note that the program above outputs 25; this is 1+3+5+7+9. However, on the last iteration, i set to 11. The loop stops since i is greater than 10; this is pointed out since i is not equal to 10.

For any for loop of the form:

```
For i = m To n Step s
```

The loop will be executed exactly once if m equals n no matter what value s has.
The loop will not be executed at all if m is greater than n and s is positive, or if m is less than n and s is negative. Each for must also be paired with a Next.

Just as we constructed nested loops using the While statement, we can also make nested loops using for statements. Just as with the while loops, nested for loops must be completely contained inside the outer loop:



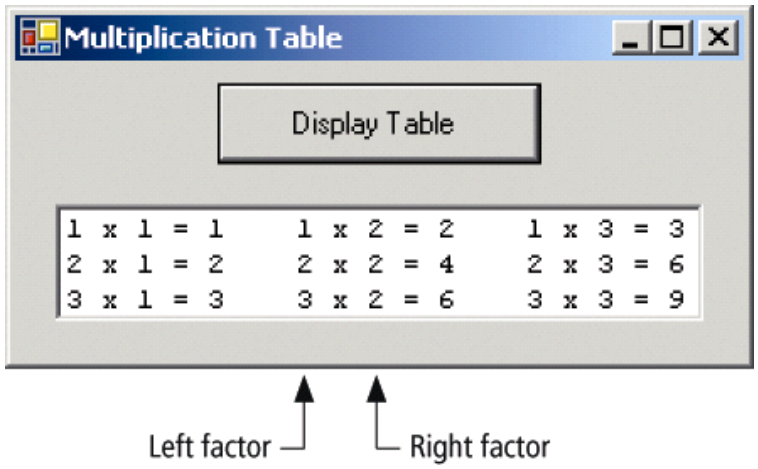
Here is an example to create a multiplication table:

```
For j = 1 To 3
   row = ""
   For k = 1 To 3
      entry = j & " x " & k & " = " & (j * k)
      row &= entry & " "
   Next
   lstTable.Items.Add(row)
Next
```

Outer loop

Inner loop

The resulting output is:



In-Class Exercise: Write a program that inputs from the user how many numbers she would like to enter. The program should then input that many numbers and computes the average of the numbers. All input should be via InputBox.

Working with Images

Images provide a nice visual way to see loops in action. We'll use nested loops to process images to do things you might normally run in a paint program.

First, create a project and add a PictureBox control and a Button to it. Set the text of the button to "Test". Load an image into the PictureBox. In this example I picked the following image:

Let's show how we can access individual colors of the image. Add the following code to the Button Click event of the "Test" button:

```
Private Sub btnTest_Click(. . .) Handles btnTest.Click
    ' Get bitmap of the image
    Dim bmp As Bitmap = New Bitmap(PictureBox1.Image)
    Dim c As Color
    Dim x As Integer

    ' Get color values for first line
    For x = 0 To bmp.Width - 1
        ' Get color of pixel at coordinate (x, 0)
        c = bmp.GetPixel(x, 0)
        Console.WriteLine("Red=" & c.R & " Green=" & c.G & _
                          " Blue=" & c.B)
    Next
End Sub
```

This code will output the Red, Green, and Blue values of each pixel on the first horizontal line of the image when we click the button. The output will show up in the Debug Output window, if in debug mode. Here is a sample of the output:

```
Red=202 Green=196 Blue=172
Red=141 Green=136 Blue=114
...
```

We can also set the color of pixels if we like. Consider the following code:

```
Private Sub btnTest_Click(. . .) Handles btnTest.Click
    ' Get bitmap of the image
    Dim bmp As Bitmap = New Bitmap(PictureBox1.Image)
    Dim x As Integer

    For x = 0 To bmp.Width - 1
        ' Set color of pixel at coordinate (x, 0) to Red
        bmp.SetPixel(x, 0, Color.FromArgb(255, 0, 0))
    Next
    PictureBox1.Image = bmp
End Sub
```

This code loops through each pixel on the top row and sets its color to red (255 red, 0 green, 0 blue). Note that we must reset the Image property to our bitmap at the end for the changes to take effect. This is shown below (the top line is turned to red).

Image Brightness

If we wanted to set every pixel to red, we would just need a nested loop so that we process every row in addition to the columns. An example is shown below. However, it doesn't turn every pixel to red – can you guess what it will do?

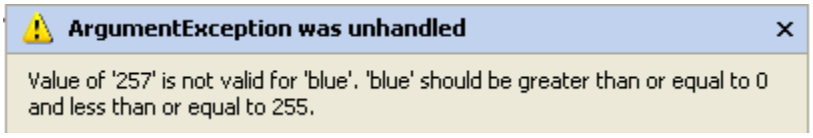
```
Private Sub btnTest_Click(. . .) Handles btnTest.Click
    Dim bmp As Bitmap = New Bitmap(PictureBox1.Image)
    Dim c As Color
    Dim x, y As Integer

    For x = 0 To bmp.Width - 1
        For y = 0 To bmp.Height - 1
            c = bmp.GetPixel(x, y)

            Dim intRed As Integer = CInt(c.R)
            Dim intGreen As Integer = CInt(c.G)
            Dim intBlue As Integer = CInt(c.B)

            intRed = CInt(intRed / 1.2)
            intGreen = CInt(intGreen / 1.2)
            intBlue = CInt(intBlue / 1.2)

            c = Color.FromArgb(intRed, intGreen, intBlue)
            bmp.SetPixel(x, y, c)
        Next
    Next
    PictureBox1.Image = bmp
End Sub
```

In this case we decrease the red, green, and blue components by 1.2 every time we click the button. This darkens the entire image until it becomes black.

If we wanted to brighten the image, we might try changing the code so we multiply 1.2 instead of dividing by 1.2:

```
            intRed = CInt(intRed * 1.2)
            intGreen = CInt(intGreen * 1.2)
            intBlue = CInt(intBlue * 1.2)
```

However, this results in an error message:



A color value cannot be larger than 255. We can compensate for this by limiting the maximum value of a color to 255:

```
intRed = CInt(intRed * 1.2)
If (intRed > 255) Then
    intRed = 255
End If
intGreen = CInt(intGreen * 1.2)
If (intGreen > 255) Then
    intGreen = 255
End If
intBlue = CInt(intBlue * 1.2)
If (intBlue > 255) Then
    intBlue = 255
End If
```

Every time the button is clicked the image will get brighter, until everything is washed out and eventually becomes white (except for values that started at 0, in which case multiplying by 1.2 still results in 0).

Changing Color Values

We don't have to always change the red, green, and blue colors by the same amount. Consider a form with the following image in PictureBox2:

Let's try to change the colors to simulate a sunset. When the sun sets, things turn red, so we could try increasing the amount of red in every pixel but leave the green and blue alone:

```
Dim bmp As Bitmap = New Bitmap(PictureBox2.Image)
Dim c As Color
Dim x, y As Integer

For x = 0 To bmp.Width - 1
    For y = 0 To bmp.Height - 1
        c = bmp.GetPixel(x, y)

        Dim intRed As Integer = CInt(c.R)
        Dim intGreen As Integer = CInt(c.G)
        Dim intBlue As Integer = CInt(c.B)

        intRed = CInt(intRed * 1.4)
        If (intRed > 255) Then
            intRed = 255
        End If

        c = Color.FromArgb(intRed, intGreen, intBlue)
        bmp.SetPixel(x, y, c)
    Next
Next
PictureBox2.Image = bmp
```

This sort of works at first, but repeated applications of the code brings out too much red in things like the grass and the simulated sunset makes the image brighter! It looks like a scene from Mars:

Is that how a sunset really works? Instead, maybe less blue and green is visible as the sun sets and this makes things look more red. We can try this by lowering the amount of blue and green:

```
'intRed = CInt(intRed * 1.4)
'If (intRed > 255) Then
'   intRed = 255
'End If
intGreen = CInt(intGreen * 0.7)
intBlue = CInt(intBlue * 0.7)
```

The result looks reasonably good:

Grayscale and Sepia

We can also use our basic nested loop to easily convert an image to grayscale. A color of gray is one in which the red = green = blue. Large values are white and small values are black. An easy way to make an grayscale image out of color is to set each color value to the average of all three:

Gray = (Red + Green + Blue) / 3
Red = Gray
Green = Gray
Blue = Gray

Here is an example:

```
Dim bmp As Bitmap = New Bitmap(PictureBox1.Image)
Dim c As Color
Dim x, y As Integer

For x = 0 To bmp.Width - 1
    For y = 0 To bmp.Height - 1
        c = bmp.GetPixel(x, y)

        Dim intRed As Integer = CInt(c.R)
        Dim intGreen As Integer = CInt(c.G)
        Dim intBlue As Integer = CInt(c.B)

        Dim intGray As Integer=(intRed + intGreen + intBlue) \3

        intRed = intGray
        intGreen = intGray
        intBlue = intGray

        c = Color.FromArgb(intRed, intGreen, intBlue)
        bmp.SetPixel(x, y, c)
    Next
Next
PictureBox1.Image = bmp
```

The image with the kids becomes this:

Once we have a grayscale image, it is relatively easy to make a sepia-toned image. Sepia tones are pictures with a brownish/yellowish tint that are often seen with old photographs.

Pictures that are sepia-toned have a brownish tint to them that we associate with older photographs. In the early days of photography, sepia prints were produced by adding a pigment made from the sepia cuttlefish to the positive print of a photograph.

This tone is a little trickier to produce because there is not a simple one-to-one correspondence between the RGB and the sepia intensity. Instead, we must scale the sepia based on different ranges of the grayscale:

```
For x = 0 To bmp.Width - 1
    For y = 0 To bmp.Height - 1
        c = bmp.GetPixel(x, y)

        Dim intRed As Integer = CInt(c.R)
        Dim intGreen As Integer = CInt(c.G)
        Dim intBlue As Integer = CInt(c.B)

        ' First, make it Gray
        Dim intGray As Integer =(intRed + intGreen + intBlue)\3
        intRed = intGray
        intGreen = intGray
        intBlue = intGray

        ' Tint shadows
        If (intRed < 63) Then
            intRed = CInt(intRed * 1.15)
            intBlue = CInt(intBlue * 0.9)
        End If

        ' Tint midtones
        If (intRed > 62 And intRed < 192) Then
            intRed = CInt(intRed * 1.2)
            intBlue = CInt(intBlue * 0.85)
        End If

        ' tint highlights
        If (intRed > 191) Then
            intRed = CInt(intRed * 1.08)
            If (intRed > 255) Then
                intRed = 255
                intBlue = CInt(intBlue * 0.93)
            End If
        End If

        c = Color.FromArgb(intRed, intGreen, intBlue)
        bmp.SetPixel(x, y, c)
    Next
Next
PictureBox1.Image = bmp
```

End result:

Selective Color Changes

Let's say that we would like to turn the color of the power ranger on the t-shirt from red to blue.

Using a paint program it is possible to see that the red pixels are very red – their color is typically something like (Red=140, Green=15, Blue=15). To turn the power ranger to blue, we could find all pixels where the (Red – Green) > 90, and (Red – Blue) > 90. Then we can swap the Red and Blue components. So (Red=140, Green=15, Blue=15) would become (Red = 15, Green = 15, Blue = 140).

```
Dim bmp As Bitmap = New Bitmap(PictureBox1.Image)
Dim c As Color
Dim x, y As Integer

For x = 0 To bmp.Width - 1
    For y = 0 To bmp.Height - 1
        c = bmp.GetPixel(x, y)

        Dim intRed As Integer = CInt(c.R)
        Dim intGreen As Integer = CInt(c.G)
        Dim intBlue As Integer = CInt(c.B)

        If (intRed - intGreen > 90) And _
           (intRed - intBlue > 90) Then
            Dim temp As Integer = intRed
            intRed = intBlue
            intBlue = temp
        End If

        c = Color.FromArgb(intRed, intGreen, intBlue)
        bmp.SetPixel(x, y, c)
    Next
Next
PictureBox1.Image = bmp
```

The result is close, but not quite there. We change the color on most of the power ranger, but some of the flesh tones are inadvertently changed as well. We could increase our threshold from 90 to 100 and get less flesh tones, but then fewer pixels on the power ranger would be changed.

One way out of this problem would be to only apply our loop to a small area instead of the entire image. For example we could change our loop boundaries to only include the pixels that surround the power ranger:

```
For x = 104 To 144
      For y = 183 To 272
```

We get most of the power ranger and miss the skin:

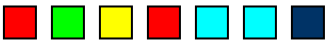
To get the rest of the power ranger we could write a separate nested loop that only changes pixels around the red portion we would like to convert.

A very similar process is done when performing red-eye reduction on an image in a photo editing program. The user typically selects the eye region (so the program knows what area to look for) and changes any reddish pixels in that area to dark pixels.

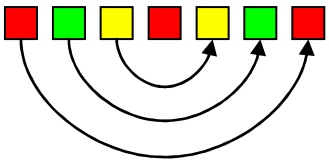
Copying Pixels - Mirroring

In addition to changing the color of pixels we can also "move" pixels on the image. In this case we are not literally moving the pixel like we would move an object, but we can copy a pixel's color values to another location to get the effect of moving the pixel.

The mirroring effect is fairly easy to create. Imagine a mirror placed in the middle of the image, so the left half is reflected onto the right half. We can achieve this effect by copying the pixels from the left onto the right. If we have the following row of pixels:



Then we reflect the left side to the right across the midpoint:



Once we calculate the midpoint of the image, we copy the leftmost pixel to the rightmost. Then the second to left pixel is copied to the second to right, or rightmost – 1. This continues until we reach the midpoint.

If xSrc is the coordinate of the source pixel to copy, then xDest = RightmostPixelIndex – xSrc. In our case, we can get the RightmostPixelIndex by getting bmp.Width – 1. We have to subtract 1 since the index starts at position 0, not position 1.

Here is the code. Note that the y loop is the outer loop, so we can process the image by row:

```
Dim bmp As Bitmap = New Bitmap(PictureBox1.Image)
Dim colSrc, colDest As Color
Dim x, y As Integer
Dim midpoint As Integer = bmp.Width \ 2

For y = 0 To bmp.Height - 1
    For x = 0 To midpoint
        colSrc = bmp.GetPixel(x, y)

        Dim intRed As Integer = CInt(colSrc.R)
        Dim intGreen As Integer = CInt(colSrc.G)
        Dim intBlue As Integer = CInt(colSrc.B)

        colDest = Color.FromArgb(intRed, intGreen, intBlue)
        bmp.SetPixel(bmp.Width - x - 1, y, colDest)
    Next
Next
PictureBox1.Image = bmp
```

And a resulting image giving identical twins:

It should be straightforward to modify the program to mirror the image vertically instead of horizontally. Switch the x/y and use the bitmap height instead of width.

Copying Pixels - Chromakey

Finally, we can also copy pixels between images. A common form of copying between pixels is chromakey. This is the technique used by the weatherperson so that he or she can be seen standing in front of a weather map. In reality, they are standing in front of a green or blue screen. The map is then copied over everywhere there is green or blue.

As an example, let's say that we have the following two images. The first is a model photographed in front of a green screen. The second is the Parthenon:

First, let's just copy the model image onto the Parthenon image. If the model is loaded into a picturebox named pboxGreen and the Parthenon is in a picturebox named pboxParthenon, then the following will copy the upper left hand corner of the model image to the Parthenon beginning at coordinate 257, 45:

```
Private Sub btnTest_Click(. . .) Handles btnTest.Click
    Dim bmpSrc As Bitmap = New Bitmap(pboxGreen.Image)
    Dim bmpDest As Bitmap = New Bitmap(pboxParthenon.Image)
    Dim c As Color
    Dim x, y As Integer
    Dim destX, destY As Integer

    For x = 0 To bmpSrc.Width - 1
        For y = 0 To bmpSrc.Height - 1
            c = bmpSrc.GetPixel(x, y)

            destX = 257 + x
            destY = 45 + y
            bmpDest.SetPixel(destX, destY, c)
        Next
    Next
    pboxParthenon.Image = bmpDest
End Sub
```

The result is:

We can get rid of the green by copying only those pixels that are “not mostly green”. In this case, I will define “not mostly green” as pixels whose (green – red) < 50 or (green – blue) < 50. In other words, green is not the dominant color component compared to red or blue.

```
Private Sub btnTest_Click(. . .) Handles btnTest.Click
    Dim bmpSrc As Bitmap = New Bitmap(pboxGreen.Image)
    Dim bmpDest As Bitmap = New Bitmap(pboxParthenon.Image)
    Dim c As Color
    Dim x, y As Integer
    Dim destX, destY As Integer

    For x = 0 To bmpSrc.Width - 1
        For y = 0 To bmpSrc.Height - 1
            c = bmpSrc.GetPixel(x, y)

            Dim intRed As Integer = CInt(c.R)
            Dim intGreen As Integer = CInt(c.G)
            Dim intBlue As Integer = CInt(c.B)

            If (intGreen - intRed < 50) Or _
               (intGreen - intBlue < 50) Then
                destX = 257 + x
                destY = 45 + y
                bmpDest.SetPixel(destX, destY, c)
            End If
        Next
    Next
    pboxParthenon.Image = bmpDest
End Sub
```

The end result looks like this:

If the model was wearing something green, then you might end up with undesirable (or desirable?) effects where you could see through parts of the person.

Lastly, we can get a sort of ghostly effect by averaging together the source and destination colors:

```
If (intGreen - intRed < 50) Or _
   (intGreen - intBlue < 50) Then
    destX = 257 + x
    destY = 45 + y
    Dim cDest As Color
    cDest = bmpDest.GetPixel(destX, destY)
    intRed = CInt((intRed + cDest.R) / 2)
    intGreen = CInt((intGreen + cDest.G) / 2)
    intBlue = CInt((intBlue + cDest.B) / 2)
    bmpDest.SetPixel(destX, destY, _
        Color.FromArgb(intRed, intGreen, intBlue))
End If
```