If a program is not running the way you intend, then you will have to debug the program. Debugging is the process of finding and correcting the errors. There are two general ways to go about debugging:

1. Add Console.WriteLine or MsgBox statements at strategic points in the program to display the values of selected variables or expressions until the error is detected.

2. Use an integrated debugger that lets you pause, view, and alter variables while the program is running. Such a tool is called a debugger.

Debugging with WriteLine

Let’s first examine the WriteLine method. Although somewhat “primitive” it is useful since it works in virtually any programming environment. Consider the following program, which intends to determine if a word is a palindrome:

```vbnet
Private Sub Button_Click(...) Handles Button.Click
    If CheckPalindrome("redivider") = True Then
        Console.WriteLine("It's a palindrome")
    Else
        Console.WriteLine("Not a palindrome")
    End If
End Sub

Function CheckPalindrome(ByVal sWord As String) As Boolean
    Dim iFront, iRear As Integer
    iFront = 1
    iRear = sWord.Length()
    While (iFront < iRear)
        If (sWord.Substring(iFront, 1) <> 
            sWord.Substring(iRear, 1)) Then
            Return False     ' Not a palindrome
        End If
        iFront += 1
        iRear -= 1
    End While
    Return True     ' A palindrome
End Function
```

This program intends to work by comparing the first letter with the last letter, the second letter with the second to last letter, and so forth, continuing until we hit the middle or find letters do not match.

First, let’s just try to run the program. When we click the button we get the error message shown below:
If we click on “Break” this will bring us into the debugger automatically. For now, click on “Continue” and the program will just exit.

Something is obviously wrong – to find out where the program is crashing, we could add some WriteLine statements in strategic places. It must be crashing somewhere in the CheckPalindrome function, because otherwise it would print something in the button click event. Let’s add some strategic WriteLine statements:

```vbnet
Function CheckPalindrome(ByVal sWord As String) As Boolean

    Dim iFront, iRear As Integer
    iFront = 1
    iRear = sWord.Length()
    Console.WriteLine("Initial iFront = " & iFront & ": iRear = " & iRear)
    While (iFront < iRear)
        Console.WriteLine("iFront = " & iFront & ": iRear = " & iRear)
        If (sWord.Substring(iFront, 1) <> sWord.Substring(iRear, 1)) Then
            Return False ' Not a palindrome
        End If
        iFront += 1
        iRear -= 1
    End While
    Return True ' A palindrome
End Function
```

When we re-run the program now, it still crashes, but if we scroll back in the Output window we will see:

```
Initial iFront = 1 iRear = 9
iFront = 1 iRear = 9
An unhandled exception of type 'System.ArgumentOutOfRangeException' occurred in mscorlib.dll
```

This means that we successfully made it into the loop, but it is crashing somewhere afterwards inside the loop. The likely culprit looks like the Substring statements. We can use more WriteLine statements to try to isolate the problem:
Function CheckPalindrome(ByVal sWord As String) As Boolean
    Dim iFront, iRear As Integer

    iFront = 1
    iRear = sWord.Length()
    While (iFront < iRear)
        Console.WriteLine("iFront = " & iFront & " iRear = " & iRear)
        Console.WriteLine("Front Char is " & sWord.substring(iFront, 1))
        Console.WriteLine("Rear Char is " & sWord.substring(iRear, 1))
        If (sWord.Substring(iFront, 1) <> sWord.Substring(iRear, 1)) Then
            Return False ' Not a palindrome
        End If
        iFront += 1
        iRear -= 1
    End While
    Return True ' A palindrome
End Function

Upon executing the new version with the WriteLine statements:

iFront = 1 iRear = 9
Front Char is e
An unhandled exception of type 'System.ArgumentOutOfRangeException' occurred in mscorlib.dll

This time the program prints the “Front Char is e” and then crashes trying to print the rear character. At this point we have narrowed down the line causing the crash to the code:

sWord.Substring(iRear, 1)

We know that iRear is 9. So there is a problem with sWord.Substring(9,1). Hopefully at this point this would be enough information to tell us that 9 is an invalid number – in this case, because it is exceeding the size of the string. Substring uses indices going from 0 to the length of the string minus one!

The last message also illuminated another error; the first character should be the letter “r” not “e”! We are really getting the second character and thinking it is the first. This is because the first character starts at position 0 in Substring.

This is enough information to fix the program, as shown below:
Function CheckPalindrome(ByVal sWord As String) As Boolean
Dim iFront, iRear As Integer

iFront = 0
iRear = sWord.Length() - 1
While (iFront < iRear)
    If (sWord.Substring(iFront, 1) <> sWord.Substring(iRear, 1)) Then
        Return False ' Not a palindrome
    End If
    iFront += 1
    iRear -= 1
End While
Return True ' A palindrome
End Function

The program will now run correctly.

Using the Integrated Debugger

While the process described above works, it is somewhat tedious to all of the WriteLine statements and them remove them. A much nicer technique is to use the built-in debugger.

VB.NET programs run in one of three modes – design mode, run mode, or break mode. The current mode is displayed in parentheses in the VB.NET title bar. Design mode is where you design the program. Run mode is when you run the program. Break mode is when you pause the program to debug it.

If we return to the original program with the bugs, one way to enter break mode is to click on the “Break” button when a fatal error occurs. VB.NET will jump straight to the line causing the fatal error:

Function CheckPalindrome(ByVal sWord As String) As Boolean
Dim iFront, iRear As Integer

iFront = 0
iRear = sWord.Length()
While (iFront < iRear)
    If (sWord.Substring(iFront, 1) <> sWord.Substring(iRear, 1)) Then
        Return False ' Not a palindrome
    End If
End While
Return True ' A palindrome
End Function

Instead of having to print out each variable, we can immediately see the contents of all the active variables by looking in the “Autos” window:
We can also click on the “Locals” tab to see all local variables in the current procedure:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Front</td>
<td>1</td>
<td>Integer</td>
</tr>
<tr>
<td>Rear</td>
<td>9</td>
<td>Integer</td>
</tr>
<tr>
<td>sWord</td>
<td>9</td>
<td>Integer</td>
</tr>
</tbody>
</table>

The values in red indicate that the variables have just been changed.

You can also look at the value of variables by hovering the mouse over the variable name.

The Watch and Immediate windows let you view the values of variables and expressions. To use the Immediate window, tab to the “Command Window” and, type a statement of the form:

```
? expression
```

to display the value of the expression. For example:

```
? sword
"redivider"
? sword.length()
9
? sword.substring(iFront,1)
"e"
? sword.substring(iRear,1)
Run-time exception thrown : System.ArgumentOutOfRangeException - Index and length must refer to a location within the string.
Parameter name: length
```

This should give us enough information to fix the bug.

Depending upon the state of the machine, we can change the contents of variables in the debugger via:

```
varName = newValue
```
We can also view the contents of variables and expressions by adding them into the “Name” column of the Watch window:

Whenever you are done debugging your program, you must make sure that the debugging session is ended before you go back to edit your code. Click the “Stop Debugging” button to exit the debugger.

Another useful way to use the debugger is to set a breakpoint. A breakpoint stops execution at a particular line of code and enters Break mode. This is useful when you know that a particular routine is faulty and want to inspect the code more closely when execution reaches that point.

To set a breakpoint, click in the border to the left of the code. A red dot will appear. Click the same dot to turn the breakpoint off. In the example below, we have fixed the Substring bug but still have iFront initialized to 1 instead of to 0:

```vba
Function CheckPalindrome(ByVal sWord As String) As Boolean
    Dim iFront, iRear As Integer

    iFront = 1
    iRear = sWord.Length() - 1
    While (iFront < iRear)
        If (sWord.Substring(iFront, 1) <> sWord.Substring(iRear, 1)) Then
            Return False ' Not a palindrome
        End If
        iFront += 1
        iRear -= 1
    End While
    Return True ' A palindrome
End Function
```
When we run the program and reach this code, the program automatically enters Break mode. We can now step through the program one line at a time using the buttons:

These buttons are used respectively to **step into** a procedure, **step over** a procedure, or **step out** of a procedure. We can use these buttons and view our watch and local variables change as we run the program.

As a shortcut, F11 steps into a procedure, and F10 steps over a procedure. These commands are the same for non-procedures (i.e. the move to the next statement).

There is much more that the debugger can do, we have only covered the very basics here. However, what we have covered is sufficient to track down the majority of bugs. In particular, setting breakpoints, tracing through the program, and viewing the contents of variables is the most common task that will aid you in tracking down bugs. Feel free to check out the help topics within the program to learn more. Once you have learned to use the debugger you will have a powerful tool in your arsenal against bugs. Although there is a little bit of a learning curve to use the debugger, it is time well spent in tracking down future errors more quickly.