VS Debugging and Troubleshooting ..... by B. Arnold 2/16/2016

Welcome to the CSIG, a Special Interest Group of the ACGNJ. This group is a "Special Interest Group" of the ACGNJ devoted to discussing programming languages in general and C, C#, and C++ for Windows programming in particular. Each month a small but hopefully useful program (complete with source code) is presented for discussion.

This month's topic will be "Troubleshooting and Debugging of Visual Studio programs with emphasis on C# and C++. We will discuss a program that I presented in November and how it has evolved do to problem solving. Over the years I have found that a significant amount of project time is often spent on debugging. It's quite common for debugging to take 50 percent of a project's estimated development time. Since managers seldom allow more that 10 percent on project estimates, this can be a source of extreme frustration. An excellent book on the subject is "The Mythical Man-Month" by Frederick Brooks, Jr. Check it out on Wikipedia along with Murphy's Law. Although the presentation will include some real-life issues on multi-thread code, attendees are encouraged to bring in examples of their own trouble shooting experiences and history.

This is a CLI DOT NET program is coded using C-sharp (C#) and is very similar to C++. It also still uses the same DOT NET libraries.

Debugging and Troubleshooting

1. Because C++ and C# are Strongly-Typed languages, you have a head-start in writing reliable software. However, be aware of the important distinction between References and Values.
2. Never use Integer or Numeric values to represent Boolean logic.

3. For C#, the Equals() Method is more reliable than "==" for comparing strings. Also, LINQ (Language Integrated Query) provides type-checking in collections of various types.
4. As you write code, let Visual Studio IDE (Integrated Development Editor) advise you of errors. It's your friend.
5. Set the Warning Level of the compiler to maximum. (Usually 4)
6. Recode to eliminate any and all compiler warnings.
7. Run the program in Debug Mode. The IDE will stop when it encounters a run time error.
8. Check the memory status of the application. Watch for programs that increase memory usage over time. Don't assume that Garbage Collection will occur at any particular time. Also, be aware that most Graphic, File, and Network Objects have no garbage collection and must be deleted (finalized) manually.
9. Verify that the program is always responsive to the user and provides status information. *A file copy operation should provide feedback so that the user knows is working properly.*

10. The use of Breakpoints, Tracepoints, and Single Step can provide debugging information without having to modify your code.
11. The simplest form of debugging procedure adds "Print Statements" in the code to display the running status of the application.
12. "Assert" statements may be added to the code to guarantee the outcome of checks and calculations.
13. "Debug" statements may be added to provide IDE debugging information only during debug operations, thus eliminating the need to recode to remove the special code.

```
using System.Diagnostics;
Debug.WriteLine("The product name is " + sProdName);
Debug.WriteLine("The available units on hand are" + iUnitQty.ToString());

Debug.Assert(dUnitCost > 1, "Message will NOT appear");
Debug.Assert(dUnitCost < 1, "Message will appear since dUnitcost < 1 is false");
```

14. Most Dot Net Library functions and classes that you call will return an error status code. Your code should respond to those codes.
15. As Library codes run, they may also trigger Exceptions which are operating system errors. In most cases you will want to trap these Exceptions in your code using "try-catch" logic.

```
try
{
int j=0;   int k = i/j ;                    // Error on this line.
}
```

```
Catch
{
Console.WriteLine("An error occurred.");
}
```

16. Exceptions that are not trapped will stop the application and issue operating system errors and messages.
17. Multi-thread applications are often difficult to code because you often can't access the screen to advise the user. Other debugging methods will apply here. For example, the RunWorkerCompleted() method provides feedback on exceptions. The following example shows this.

```
private void backgroundWorker1_DoWork(object sender, DoWorkEventArgs e)
        {
        File.Copy(SourceFilePath, AdjustedDestinationFile);
        }

private void backgroundWorker1_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)

        {
        if (e.Error != null)            // Exception has occurred!
          {
          MessageBox.Show("File copy exception reported.  ERROR: \r\n"+ e.Error.Message);
          }
        Else                            // No exception reported.  Good.
          {
          MessageBox.Show("No copy errors reported.  GOOD");
          }
        }
```

18. In the above example, instead of using the MessageBox.Show() statement, you may simply write to text file without stopping the application. Alternatively, you could use the Windows Event Viewer system:

```
        using (EventLog eventLog = new EventLog("Application"))
          {
          eventLog.Source = "Application";
          eventLog.WriteEntry("Exception in DoWork()", EventLogEntryType.Information);
          }
```