Debugging in Visual Studio

Introduction

The term *debugging* can mean a lot of different things, but most literally, it means removing bugs from your code.

A debugger is a very specialized developer tool that attaches to your running app and allows you to inspect your code. In the debugging documentation for Visual Studio, this is typically what we mean when we say "debugging".

When to use a debugger

The debugger is an essential tool to find and fix bugs in your apps. However, context is king, and it is important to leverage all the tools at your disposable to help you quickly eliminate bugs or errors. Sometimes, the right "tool" might be a better coding practice.

Debugging is used to solve this types of errors:

1. Syntax Error
2. Logical Error
3. Run-time Error or Exception

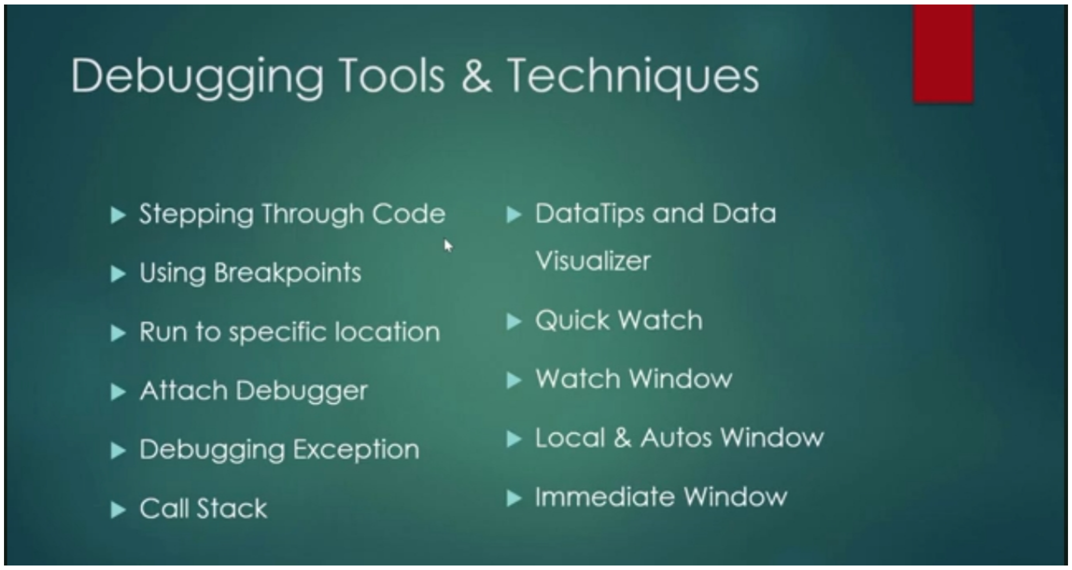
Debugging is help to find logical errors and understand the flow of execution.

To start the debugger, select F5, or choose the Debug Target button in the Standard toolbar, or choose the Start Debugging button in the Debug toolbar, or choose Debug > Start Debugging from the menu bar.

To stop the debugger, select Shift+F5, or choose the Stop Debugging button in the Debug toolbar, or choose Debug > Stop Debugging from the menu bar.

To step out from the executed function use shift + f11.

For debug the code we use technique like this:

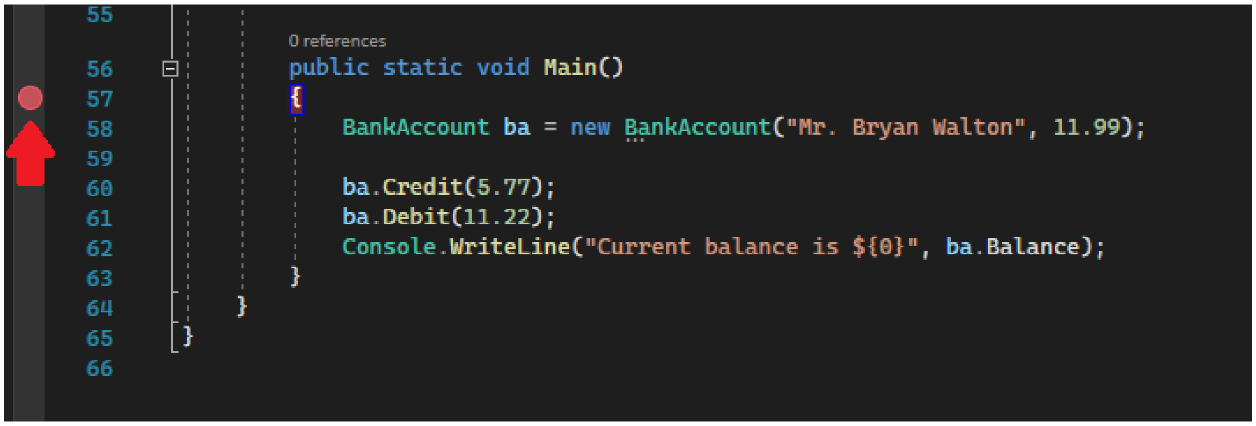


Debugging Tools & Techniques

- Stepping Through Code
- Using Breakpoints
- Run to specific location
- Attach Debugger
- Debugging Exception
- Call Stack
- DataTips and Data Visualizer
- Quick Watch
- Watch Window
- Local & Autos Window
- Immediate Window

Debug points

- Breakpoints are an essential feature of reliable debugging. You can set breakpoints where you want Visual Studio to pause your running code so you can look at the values of variables or the behavior of memory, or know whether or not a branch of code is getting run.
- A red circle appears where you set the breakpoint.



```
            0 references
            public static void Main()
            {
                BankAccount ba = new BankAccount("Mr. Bryan Walton", 11.99);

                ba.Credit(5.77);
                ba.Debit(11.22);
                Console.WriteLine("Current balance is ${0}", ba.Balance);
            }
        }
    }
```

- **Types of debug points:**

  - **Conditional breakpoint**
    A conditional breakpoint is a breakpoint that is triggered only when a specified condition is met during program execution. Instead of breaking every time the breakpoint is encountered, it allows you to define a condition based on variables or expressions. The program will only halt if the specified condition evaluates to true. This is helpful for focusing on specific scenarios or problematic code paths.

  - **Trace Point**
    A trace point is a type of breakpoint that doesn't interrupt the normal flow of the program but logs information when it's hit. When a trace point is encountered, the debugger logs a message to the output window, allowing you to trace the execution flow without pausing the program. This is useful for gathering information about the program's behavior at specific points without the need for manual intervention.

- **Temporary breakpoint**

  A temporary breakpoint is a breakpoint that is automatically removed after being hit. Unlike a regular breakpoint, which remains set until explicitly removed, a temporary breakpoint is useful when you want to interrupt the program flow once at a specific point without affecting future runs of the program. It's a quick way to inspect a specific section of code during debugging without leaving breakpoints behind.

- **Dependent breakpoint**

  A dependent breakpoint is a breakpoint that depends on another breakpoint. When the primary breakpoint is hit, it triggers the dependent breakpoint. This is helpful when you have a complex debugging scenario where you want to break at different points depending on the context. It allows you to create a hierarchy of breakpoints, making the debugging process more flexible and customizable.

Different Debug windows

- **Locals Window:**
  - Displays local variables and their current values within the scope of the current execution point.
- **Watch Window:**
  - Allows you to add variables and expressions to monitor their values during debugging. It provides a way to keep track of specific values of interest.
- **Autos Window:**
  - Automatically displays variables that are likely to be of interest based on the current context and execution point.
- **Call Stack Window:**
  - Shows the call hierarchy, displaying the sequence of method calls that led to the current point in the code.
- **Breakpoints Window:**
  - Lists all breakpoints set in your code, enabling you to manage, enable, disable, or delete breakpoints.
- **Output Window:**
  - Provides additional information about the debugging process, including program output, build information, and other status messages.
- **Immediate Window:**
  - Allows you to interactively execute code and evaluate expressions during debugging. It's a quick way to test code snippets without modifying your source.
- **Memory Window:**
  - Lets you inspect the memory of your program, view memory addresses, and explore the contents of variables in a specific memory location.
- **Modules Window:**
  - Displays a list of loaded modules and assemblies, along with information about their symbols and addresses.
- **Threads Window:**
  - Shows information about the threads currently running in your program, including their call stacks and states.
- **Exceptions Window:**
  - Provides control over how exceptions are handled during debugging, allowing you to break on specific exceptions or configure other actions.
- **Task List Window:**
  - Lists tasks and comments marked with specific tokens in your code, helping you keep track of things to do.

- **Watch 2, Watch 3, ...:**
    - Additional Watch windows that allow you to monitor multiple variables and expressions simultaneously.
- **Parallel Stacks Window:**
    - Available in multi-threaded applications, this window helps visualize and navigate the call stacks of multiple threads.

★ List of different debug windows are as follows :

Window	**Hotkey**	**See topic**
Breakpoints	CTRL+ALT+B	Use Breakpoints
Exception Settings	CTRL+ALT+E	Manage Exceptions with the Debugger
Output	CTRL+ALT+O	Output Window
Watch	CTRL+ALT+W, (1, 2, 3, 4)	Watch and QuickWatch Windows
QuickWatch	SHIFT+F9	Watch and QuickWatch Windows
Autos	CTRL+ALT+V, A	Autos and Locals Windows
Locals	CTRL+ALT+V, L	Autos and Locals Windows
Call Stacks	CTRL+ALT+C	How to: Use the Call Stack Window
Immediate	CTRL+ALT+I	Immediate Window
Parallel Stacks	CTR:+SHIFT+D, S	Using the Parallel Stacks Window

Parallel Watch	CTR:+SHIFT+D, (1, 2, 3, 4)	Get started Debugging Multithreaded Applications
Threads	CTRL+ALT+H	Debug using the Threads Window
Modules	CTRL+ALT+U	How to: Use the Modules Window
GPU Threads	-	How to: Use the GPU Threads Window
Tasks	CTR:+SHIFT+D, K	Using the Tasks Window
Python Debug Interactive	SHIFT+ALT+I	Python Interactive REPL
Live Visual Tree	-	Inspect XAML properties while debugging
Live Property Explorer	-	Inspect XAML properties while debugging
Processes	CTRL+ALT+Z	Debug Threads and Processes
Memory	CTRL+ALT+M, (1, 2, 3, 4)	Memory Windows
Disassembly	CTRL+ALT+D	How to: Use the Disassembly Window
Registers	CTRL+ALT+G	How to: Use the Registers Window

Editing

- With Hot Reload, or Edit and Continue for C#, you can make changes to your code in break or run mode while debugging. The changes can be applied without having to stop and restart the debugging session.

- To enable or disable Hot Reload:
    - If you're in a debugging session, stop debugging (Debug > Stop Debugging or Shift+F5).
    - Open Tools > Options > Debugging > .NET/C++ Hot Reload, select or clear the Enable Hot Reload and Edit and Continue when debugging check box.
- To use the classic Edit and Continue experience:
    - While debugging, in break mode, make a change to your source code.
    - From the Debug menu, click Continue, Step, or Set Next Statement. Debugging continues with the new, compiled code.
- Some types of code changes are not supported by Edit and Continue.

Conditional breakpoints

- You can control when and where a breakpoint executes by setting conditions. The condition can be any valid expression that the debugger recognizes.
- To set a breakpoint condition:
    - Right-click the breakpoint symbol and select Conditions (or press **Alt + F9, C**). Or hover over the breakpoint symbol, select the Settings icon, and then select Conditions in the Breakpoint Settings window.
    - In the dropdown, select Conditional Expression, Hit Count, or Filter, and set the value accordingly.
    - Select Close or press **Ctrl+Enter** to close the Breakpoint Settings window. Or, from the Breakpoints window, select OK to close the dialog.
- When you select Conditional Expression, you can choose between two conditions: Is true or When changed. Choose Is true to break when the expression is satisfied, or When changed to break when the value of the expression has changed.
- If you suspect that a loop in your code starts misbehaving after a certain number of iterations, you can set a breakpoint to stop execution after that number of hits, rather than having to repeatedly press **F5** to reach that iteration.
- Under Conditions in the Breakpoint Settings window, select Hit Count, and then specify the number of iterations.

Data inspector

Data inspection can be done in following ways :

- Debugger windows
- Inspect exceptions
- Inspect variables
- Set watch on variables
- View data value in data tip
- Disassembly
- Registers

- **Locals and Watch Windows:**
  - Visual Studio includes windows like Locals and Watch that allow developers to inspect the values of variables , objects ,expressions and collections during debugging. These windows display the state of variables within the current scope, helping developers understand the data flow in their code. You can change and alter values and method call from there.
- **Immediate Window:**
  - The Immediate Window in Visual Studio is another tool for data inspection. It allows developers to execute code snippets and evaluate expressions in the context of the current debugging session. This can be helpful for on-the-fly testing and data exploration.It is one type of CLI.
- **Object Browser:**
  - The Object Browser is a feature in Visual Studio that allows developers to explore the structure of assemblies, namespaces, types, and their members. While it may not be directly related to debugging, it provides insights into the structure of data types.
- **Data Tips:**
  - During debugging, hovering over variables in the code editor provides data tips, showing the current values of those variables. This is a quick way to inspect data without navigating to separate windows. Datatips are editable.

Conditional compilation

Conditional compilation is a feature in programming languages that allows certain sections of code to be included or excluded during the compilation process based on specified conditions. This feature is typically controlled by preprocessor directives, and it enables developers to create different builds for different scenarios without changing the source code.

Syntax:

```
#if DEBUG
   // This code will be included only in DEBUG builds
   Console.WriteLine("Debugging is enabled");
#else
   // This code will be included in Release builds
   Console.WriteLine("This is a Release build");
#endif
```

Key Directives:

- **#if, #else, #endif:**
  - **#if** is used to start a conditional directive.
  - **#else** specifies the code to be included if the condition is false.
  - **#endif** marks the end of the conditional block.
- **#define and #undef:**
  - **#define** is used to create a symbol that can be used in conditional directives.
  - **#undef** is used to undefine a symbol.

Debug vs. Release Builds:

- Developers might include additional logging, debugging information, or assertions in DEBUG builds but exclude them in RELEASE builds for performance reasons.

```
#if DEBUG
   // Debugging-specific code
   Console.WriteLine("Debug information");
#endif
```

Platform-Specific Code:

- Include or exclude code based on the target platform.

```
#if WINDOWS
   // Windows-specific code
#elif LINUX
   // Linux-specific code
#endif
```

Usage Considerations:

- Conditional compilation should be used judiciously to maintain code clarity and readability.
- Overuse of conditional compilation can lead to code that is difficult to understand and maintain.
- It is often used for scenarios where maintaining separate code branches would be impractical.