# Shortcuts

Notes:

* Black: Same as VS Code
* Brown: Different from VS Code

|  |  |  |
| --- | --- | --- |
| **Groups of Functions** | **Functions** | **Shortcuts** |
| **Edit** | Indent | Tab |
| Re-indent | Shift + Tab |
| Toggle single comment | Ctrl + K + C, Ctrl + K + U |
| Toggle block comment |  |
| Set multiple cursors | **Each time**: Hold Ctrl + Alt while clicking at different positions to add different cursors simultaneously  **All-in-one**: Shift + Alt + Arrow |
| Delete a whole word | Ctrl + Delete / Backspace |
| Delete line(s) | Ctrl + Shift + L |
| Duplicate line(s) | Ctrl + D or Ctrl + E + V |
| Copy / cut line(s) | Ctrl + C / X |
| Move line(s) up / down | Alt + ↑ / ↓ |
| Show completion | Ctrl + Space |
| **Navigation** | Quick jump | **Jump to file**: Ctrl + Shift + T, or Ctrl + T then type 'f '  **Jump to recent file(s)**: Ctrl + T then type 'r '  **Jump to line**: Ctrl + G, or Ctrl + T then type ': '  **Jump to member**: Ctrl + T then type 'm '  **Jump to symbol**: Ctrl + T then type '# '  **Jump back**: Ctrl + -  **Jump forward**: Ctrl + Shift + -  **Jump to last edit**: Ctrl + Shift + Backspace  **Jump to bracket**: Ctrl + ]  **Jump to Definition**: F12  **Jump to Declaration**: Ctrl + F12  **Open Peek Definition**: Alt + F12  **Open Peek Declaration**: Ctrl + Alt + F12 |
| Select occurrences of current word | **Each time**: Shift + Alt + .  **All-in-one**: Shift + Alt + ;  **Redo**: |
| Go to:   * Beginning of a line * Ending of a line * Beginning of a file * Ending of a file | * Home * End * Ctrl + Home * Ctrl + End |
| Switch between files | Ctrl + Tab |
| **View** |  |  |
| **Others** | Save all | Ctrl + Shift + S |
| Show Preference / Setting |  |

<http://visualstudioshortcuts.com/2017/>

# Debugging

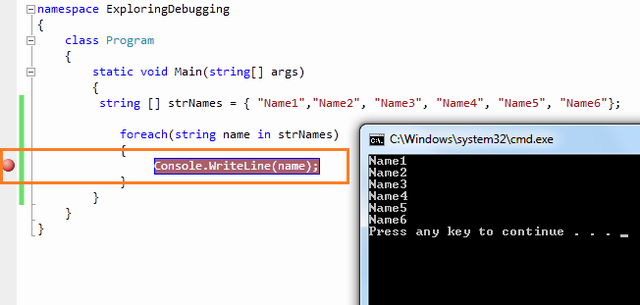
In the software development process, testing and bug fixing take more time than actually code writing. When you have some bugs in your code, first of all you need to **identify the root cause of the bug, which is called debugging**. Bug fixing comes after the debugging.

But how to debug an application? Visual Studio IDE gives us many tools to debug our application in a better way. In this article, we will discuss all the important features of VS IDE for debugging.

## Breakpoints

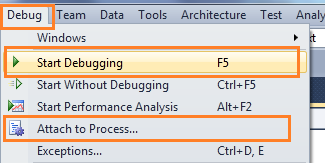
We generally start debugging any application by putting breakpoint(s) on code where we think the problem may occur. So, what is a breakpoint?

Breakpoint is used to **notify debugger where and when to pause the execution of program**. You can put a breakpoint in code by clicking on the side bar of code. So **before keeping a breakpoint, you should know (or guess) WHERE your code is going wrong and has to be stopped**. When the program reaches the breakpoint, execution will automatically pause. And the debugger will check out what's actually going wrong within the code.



### Debugging with Breakpoints

You have already set a breakpoint in your code where you want to pause the execution. Now start debugging from the **Debug Menu [F5]**:



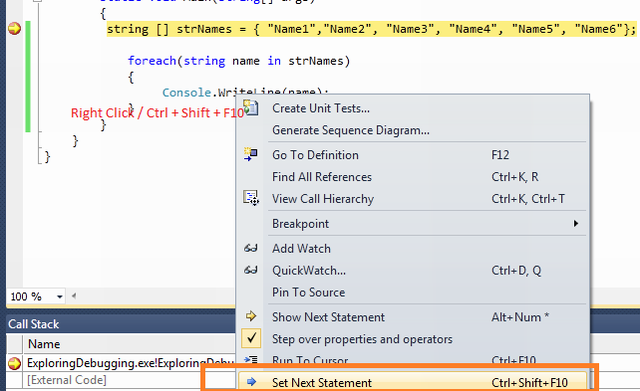
Now you have several commands available in break toolbar:

https://www.codeproject.com/KB/cs/MasteringInDebugging/debug4.png

#### Step Over [F10]

It is used to execute the code **line by line** after debugger hits the breakpoint. This will execute the currently highlighted line and then pause. Step Over will execute the entire method at a time.

After hitting the breakpoint, a **yellow arrow** icon will be created to indicate that this is the line which will be executed next.



#### Step Into [F11]

It is similar to Step Over. The only difference is, if the current highlighted section is a method call, the debugger does NOT execute the entire method at a time, but it will pause and **show us what happens inside the method**.

#### Step Out [Shift + F11]

It is related when you are using Step Into. The execution will complete the execution of the entire method, then **go out of the method**, pause and show us what happens at the next statement from where the method is called.

#### Continue [F5]

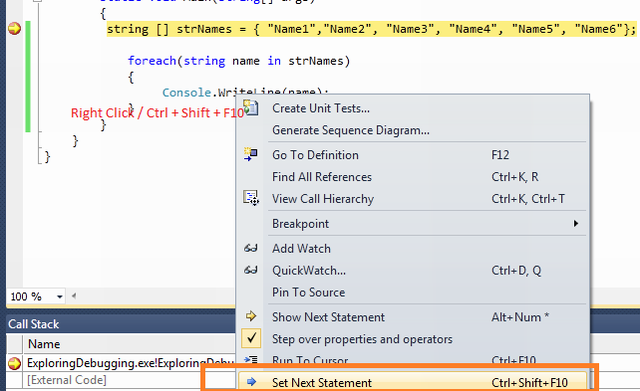
It will **continue the program flow unless reaching the next breakpoint**. If you place only one breakpoint or you go to the last breakpoint, using Continue is like running your application again.

#### Show Next Statement [Alt + \*]

It redirects you to the line of code where the yellow arrow is pointing to. So, it’s extremely useful when you are somewhere in the project and want to come back to the next executed statement.

#### Set Next Statement [Ctrl + Shift + F10]

Set Next Statement allows you tochange the path of execution of program while debugging. If your program paused in a particular line and you want to change the execution path,go to the particular line, right click on it and select "Set Next Statement" from the context menu. You will see, **the yellow arrow comes to that line without EXECUTING the previous lines of code**.



This is quite **useful when you found a line of code causing breaking your application** and you don’t want to break at that time. Instead of setting a different breakpoint and restarting debugging, simply use Set Next Statement.

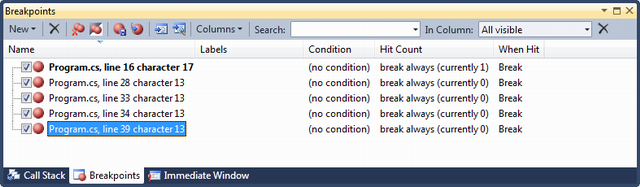
#### Run To Cursor

This feature is used when you want to **quickly jump to another piece of code** without having to step through your code line by line until reaching that point. In other word, “Run to Cursor” is like another breakpoint; you press F5 and the cursor jumps right to where the breakpoint is set.

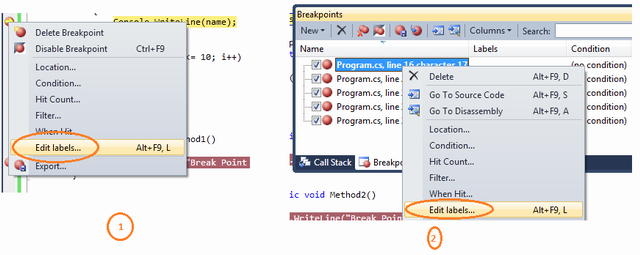
### Breakpoint Labeling

This feature enables us to better group and filter breakpoints. If you are having different types of breakpoints which are related with a particular functionality, you can **give their name** and **enable, disable, filter** based on the requirements.

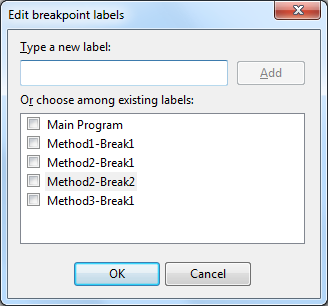
Assume you have a list of breakpoints:



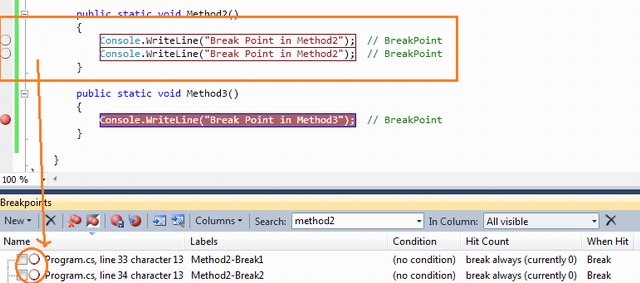
To set label for any breakpoint, you just need to right click on the breakpoint symbol on the particular line or you can set it directly from breakpoint window:



As per the sample code, I have given very simple understandable names for all the breakpoints.



Let's have a look at how this labeling helps us during debugging. At this time, all the break points are enabled. Now if you don’t want to debug the “Method1-Break1”, in a general case you need to go to the particular method and need to disable the breakpoints one by one, here you can filter them by label and disable them by selecting them together.

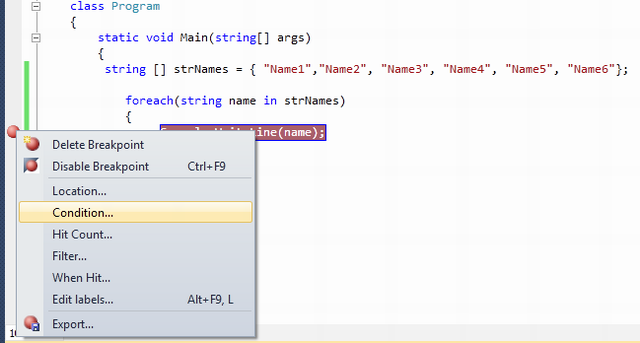


This is all about the Breakpoint Labeling. It is very much useful when you have huge lines of code, multiple projects, etc.

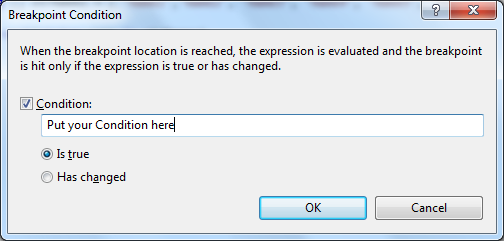
### Conditional Breakpoints

Suppose you are iterating through a large amount of data and you want to debug a few of them. It means you want to pause your program on some specific condition. VS allows you to put conditional breakpoint. If and only if that condition is satisfied, the debugger will pause the execution.

To do this, first of all you need to put the breakpoint on a particular line where you want to pause execution. Then right click on the breakpoint icon and click on "Condition":



This will open the Breakpoint Condition dialog:



Let's assume that you have the following code block:

class Program

{

static void Main(string[] args)

{

string [] strNames = { "Name1","Name2", "Name3", "Name4", "Name5", "Name6"};

foreach(string name in strNames)

{

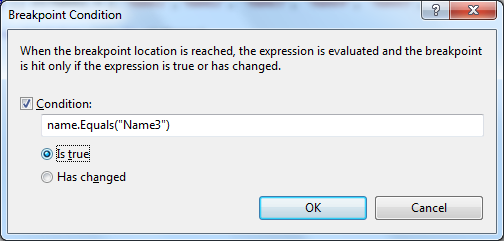
Console.WriteLine(name); *// Breakpoint is here*

}

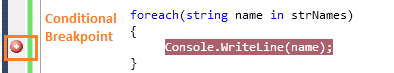
}

}

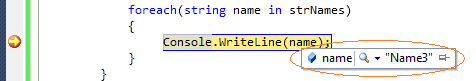
You have a breakpoint on Console.WriteLine() statement. On running of the program, execution will stop every time inside that for-each statement. Now if you want your code to break only when name="Name3", you just need to give the condition likename.Equals("Name3").



Check the Breakpoint Symbol. It should look like a plus (+) symbol inside the breakpoint circle which indicates the conditional breakpoints.

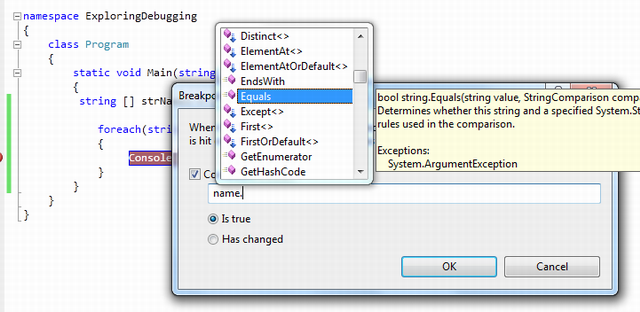


After setup of the condition of your breakpoint, if you run the application to debug it, you will see execution of program is only paused when it satisfied the given condition with breakpoint. In this case when name="Name3".



**IntelliSense In Condition Text Box**

The breakpoint condition which I have demonstrated here is very simple and can be written easily inside condition textbox. Sometimes, you may need to specify complex conditions. VS provides the IntelliSense within the condition textbox. Have a look into the below picture:



I have almost covered all about the conditional breakpoints except one thing. In condition window you have seen that there are two options available:

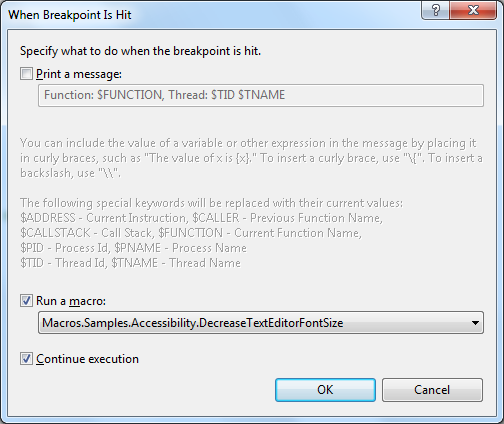
**"Is True" vs "Has Changed"**

We have already seen what the use of **"Is True"** option is. **"Has changed"** is used when you want to break the code if some value has changed for some particular value.

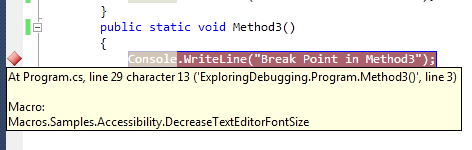
### "When Hit" (or "Action")

This feature is used when you want to **NOT pause the execution of program when breakpoint is hit**, like print a message or run some macros. You can open this window by right clicking on the breakpoint icon and choose "When Hit" (in later versions of VS, this menu item is changed to "Action").

Note: This kind of breakpoint, which prints a message to the Output window, is also called **tracepoint**.

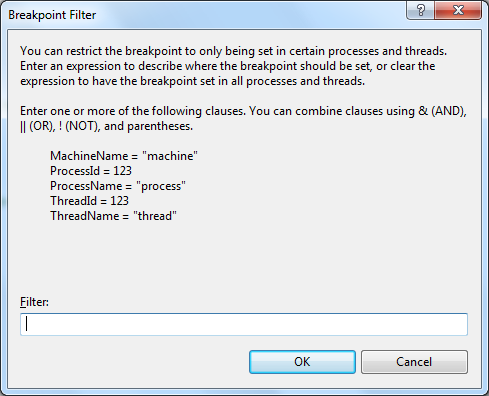


The breakpoint icon now changed to a **diamond**. You can check out the message you entered via the tool tip message box or Output window.



### Breakpoint Filter

You can restrict the breakpoint to hit for certain processes or threads. This is extremely helpful while you are dealing with multithreading program. To open the filter window, you need to right click on the breakpoint and select **"Filter"**.



In the filter criteria, you can set the process name, ID, Machine name, Thread ID, etc. I have described it in detailed uses in [Multithreading debugging section](#_4i7ojhp).

### Exception Breakpoints

In Visual Studio 2015 or newer, you can set brekapoints for exceptions at *Main Menu > Debug > Windows > Exception Settings*.

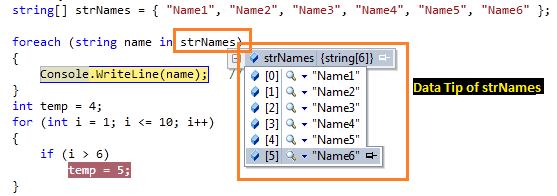
Whenever something doesn’t work for an unexplained reason, there’s a good chance an exception was thrown. **When the debugger breaks, it shows you *where* the exception was thrown**.

For how to use: <https://docs.microsoft.com/en-us/visualstudio/debugger/managing-exceptions-with-the-debugger?view=vs-2019>

## Data Tips

Data tip is kind of an advanced tool tip message, used to inspect objects or variable during debugging of the application.

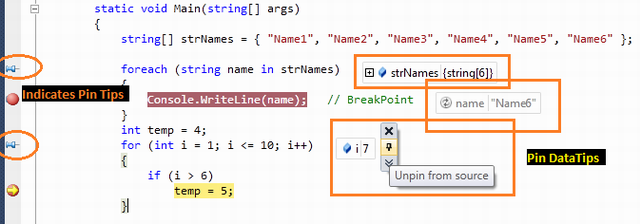
When debugger hits the breakpoint, if you **mouse over** to any of the objects or variables, you can see their current values displayed in a message box. To get the details of some complex objects like dataset, datatable, etc., you can expand their child objects or variables using**"+"** sign.



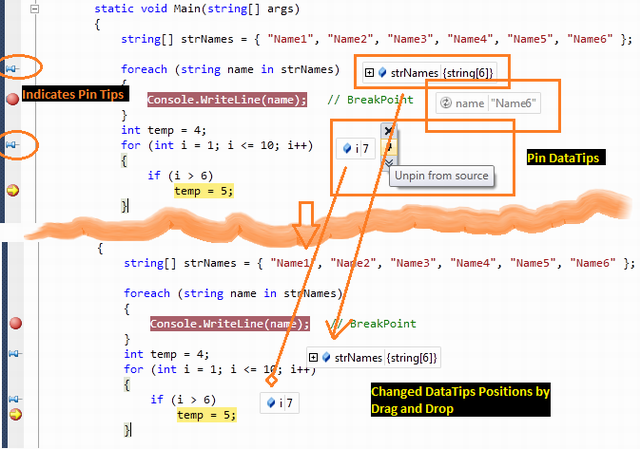
Few months ago, I published one article on [VS 2010 DataTip Debugging Tips](http://www.codeproject.com/KB/debug/VS2010Debugging.aspx).

Here are a few cool features that you can use during debugging of your application.

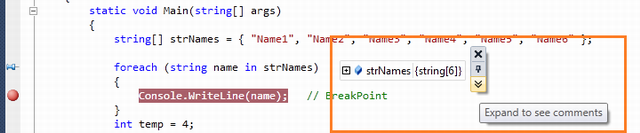
**Pin Data Tips**: While debugging in Visual Studio, we generally used mouse over on the object or variable to inspect the current value. But this is for a limited time, as long as the mouse is pointed to that object those value will be available. In Visual Studio 2010, there is a great feature to pin and unpin this inspected value. Take a look at the below picture:

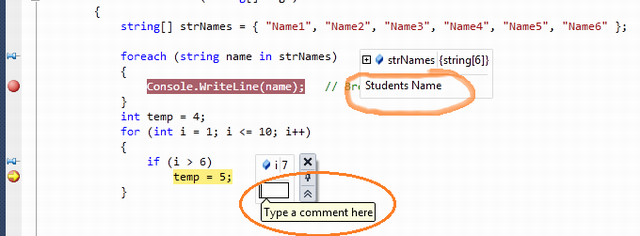


**Move Pinned Data Tips**: You can drag and drop those pinned data tips to change their position on the screen. Take a look at the below picture:

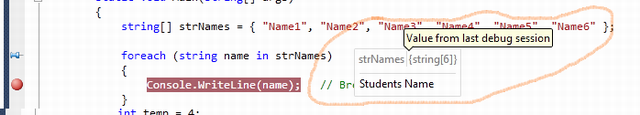


**Adding Comments:** You can add comments on the pinned data tip. Click on “**Expand to see the comments**” button.

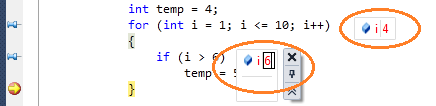




**Last Session Debugging Value:** When value of the data tip is pinned, it will remain stored in a session. In normal mode of coding, if you mouse over the pin icon, it will show the details of the last debugging session value as shown in the below picture:



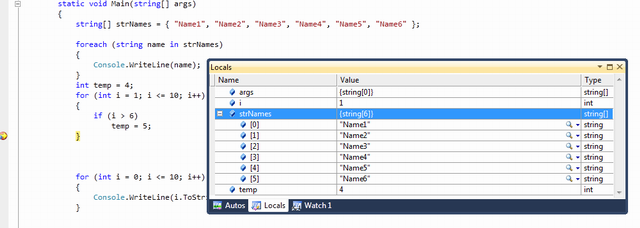
**Change Value of Pinned Objects**: You can change the value of pinned objects while debugging (like a watch window) to see the impact on the program.



## Windows

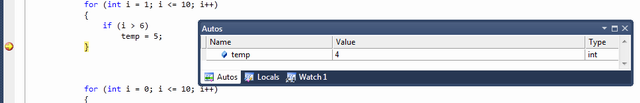
### Local Window

It automatically displays all the variables within the scope of current methods. If your debugger currently hits a particular breakpoint and if you open the "Autos" window, it will show you the current scope object variable along with the value.



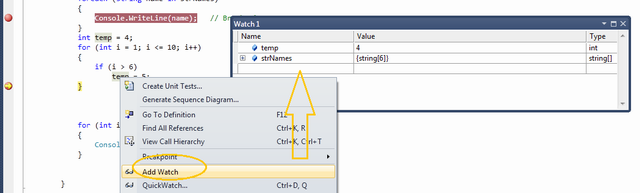
### Auto Window

The VS debugger can determine which objects or variables are important for the current code statement during the debugging. Based on that, it lists down a list of "Autos" variables.



### Watch Window

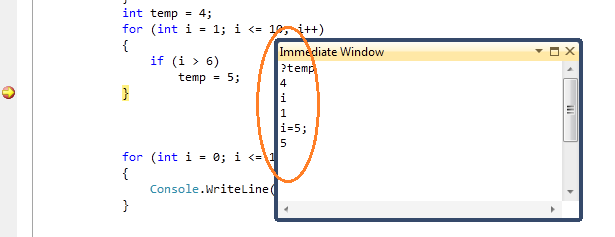
If you have many local variables and want to filter just some of them for better observation, you can do that using the "Watch" windows. To add variables as per requirement, simply right click on variable and select **"Add To Watch"**. Or you can type the variable name on the **"Name"** column of the Watch window.



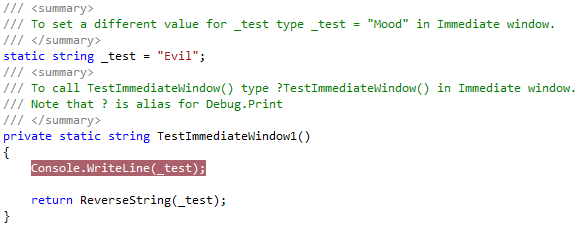
### Immediate Window

Immediate window is helpful in debug mode of the application if you want to **change the variable values or execute some statements without impacting your current debugging steps**.

You can open the "Immediate" window from menu *Debug > Window > Immediate Window*. It has a set of commands which can be executed any time during debugging. It also supports IntelliSense. During Debug mode, you can execute any command or execute any code statement from here.



Another situation where you might find the Immediate window extremely useful is, calling function directly. For example:



We can call this function from Immediate window directly as below:

Advanced_Debugging/image041.png

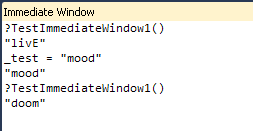
Upon hitting enter in Immediate Window, the breakpoint in the TestImmediateWindow1() function is hit without you having to debug the entire application.

Advanced_Debugging/image043.png

On proceeding, you get the output in the Immediate window too as below:

Advanced_Debugging/image045.png

You can play around with the \_test variable by changing its values and testing the reverse output:



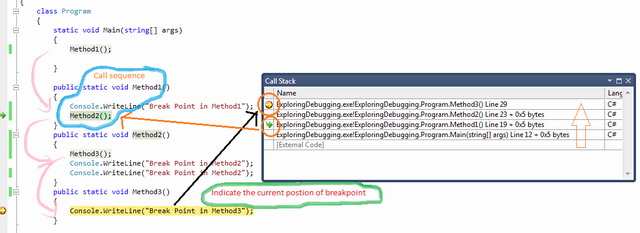
For how to use: <http://www.blackwasp.co.uk/VSImmediate.aspx>

Immediate Window vs **Command Window**: <https://stackoverflow.com/a/9000703>

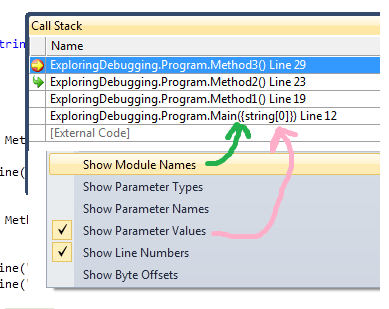
### Call Stack Window

This feature shows you an **overview of all method calls** that are currently on the stack, **as well as their calling flow**. So, it is extremely useful when you have multiple method calling or nested calling all over your application and you want to check from where this method has called.

Note: The most latest method calling is displayed on the top of the Call Stack window.



In Call Stack window, **if you click on any of the rows, it will point you to the actual line of code where the method is called**. You can also customize the Call Stack window view by right clicking on it, then select / deselect options from the context menu.



## Multithreaded Debugging

As of now, what I have discussed is all about fundamentals of debugging, knowing debugging tools and their uses. Now let's have a look into the multithreaded scenarios. Here you will see how to work with multithreaded program debugging, where is your current thread, what is the thread execution sequence, what is the state of thread. Before continuing with the demo, let's consider you have the following piece of code which you want to debug.

class ThreadTest

{

static void Main()

{

Thread t = new Thread(new ThreadStart(Go));

t.Name = "Thread 1";

Thread t1 = new Thread(new ThreadStart(Go));

t1.Name = "Thread 2";

t.Start();

t1.Start();

Go();

}

static void Go()

{

Console.WriteLine("hello!");

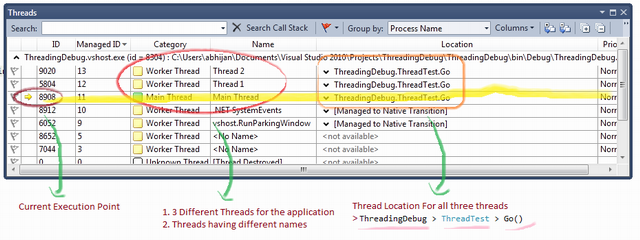
}

}

In the sample code, you have three different threads - Main Thread, Thread 1, Thread 2. I have given a thread name to make you understand better. Now set a breakpoint inside **"Go()"** and run the application. When debugger hits the breakpoint, Press **Ctrl+D,T** or Navigate through **Debug** > **Window** > **Threads**. Threads window will appeared on the screen.

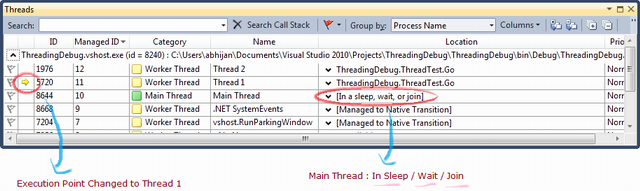
### Exploring Threads Window

After selecting the thread window from debug menu, the following screen will come:



***Figure: Detail view of Thread window***

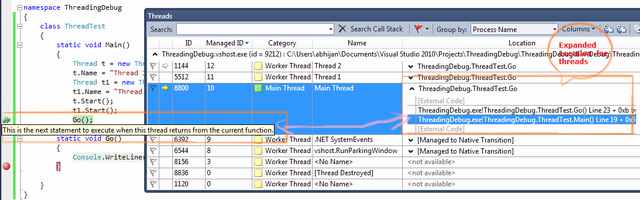
By default thread window having ID, Managed ID, Category, Name, Location and Priority column. At the start, execution pauses at "Main Thread". "Yellow Arrow" indicates the current executable thread. Category column indicates the category of threads, like main thread or worker thread. If you check the thread location, it is nothing but **Namespace** > **Class** > **Method** name. In the diagram, it is showing that the **Main Thread** will be executed next. Now to explore the next step by just pressing **"F5"** and see what are the changes in thread window.



***Figure: Detail view of Thread window - For Next Steps***

So after pressing F5, it jumped to the next step to thread 1. you can also check the current location for Main Thread. It says "**Sleep/ Wait / Join**" , means waiting for something to complete. Similarly the next step will move you to thread 2. From the Thread window, you can understand how easy it is to monitor your threads using this debugger tool.

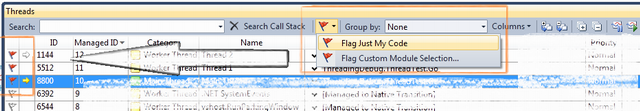
There is another great feature available within the thread window. You can **expand/collapse** the *Thread Location*and can see what is next. For example, if you expand the location for **"Main Thread"**, it will look like the diagram given below:



***Figure: Expanded Location View For Threads***

### Flag Just My Code

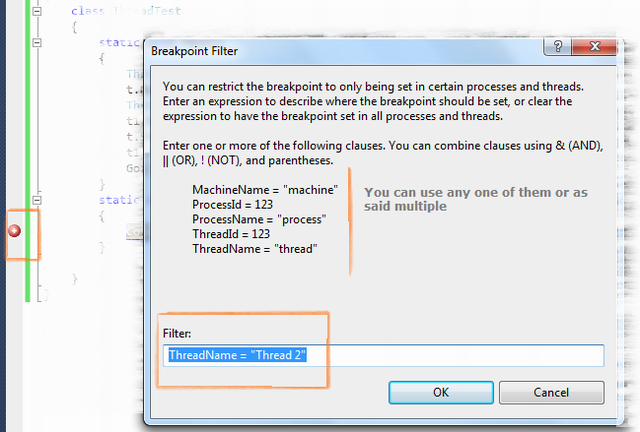
The sample code which I have explained for the thread debugging is very simple. What will happen if you have a huge code block with multiple number of threads. Then it will be very difficult for you to identify which thread is part of your code or which ones are not related. Thread window gives you very easy features to set the "**Flag**" for all the threads which are part of your code. For that, you need to just flag your thread by option "**Flag Just My Code**".



***Figure: Flag Just My Code***

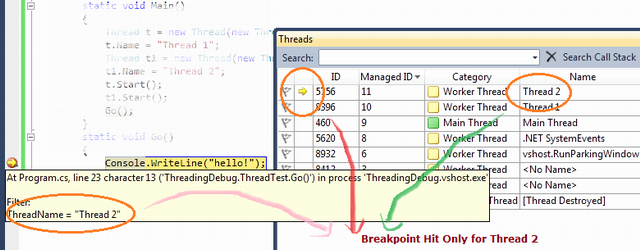
### Break Point Filter - Multithread Debugging

While discussing about breakpoint filter in breakpoint section, I said that breakpoint filter is very much helpful for Multithreaded debugging mode. Now this is the time to explore it. In our current example, we have three threads Main Thread, Thread1 and Thread 2. Now what if you want **breakpoint to hit only for "Thread 2"**. What will you do ? Here is the use of breakpoint filter. Right click on the breakpoint, select "**Filter**" from the context menu. Now in breakpoint filter window, you need to fill the filter criteria. As per your requirement, you need to specify "ThreadName="Thread 2" .



***Figure: Breakpoint Filter - Multithreaded Debugging***

Here ThreadName was one of the criteria by which you can filter, but you can filter on multiple clauses like ThreadID, ProcessName, ProcessID, etc. After setting the breakpoint filter, run the application and open the "Threads" window.



***Figure: Breakpoint Filter - Multithreaded Debugging***

You will find your program execution has only paused during the execution of **"Thread 2"** .

This is all about the debugging with multithreaded application. Hope you have learned something from it. Let's start with another most important topic "Parallel Debugging".

## Parallel Debugging

This is another great feature added to Visual Studio 2010 to debug parallel program. Parallel programming is the new feature coming with .NET 4.0. If you want to learn more about parallel programming, please check [here](http://msdn.microsoft.com/en-us/library/dd460693.aspx).

Now Debugging the parallel program is also a big topic. Here I will give you a basic overview to know about the debugging of parallel program. To discuss about it, let's consider you have the following piece of code:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading;

using System.Threading.Tasks;

namespace ParalleTaskDebugging

{

class Program

{

static void Main(string[] args)

{

var task\_a = Task.Factory.StartNew(() => DoSomeWork(10000));

var task\_b = Task.Factory.StartNew(() => DoSomeWork(5000));

var task\_c = Task.Factory.StartNew(() => DoSomeWork(1000));

Task.WaitAll(task\_a, task\_b, task\_c);

}

static void DoSomeWork(int time)

{

Thread.Sleep(time);

}

}

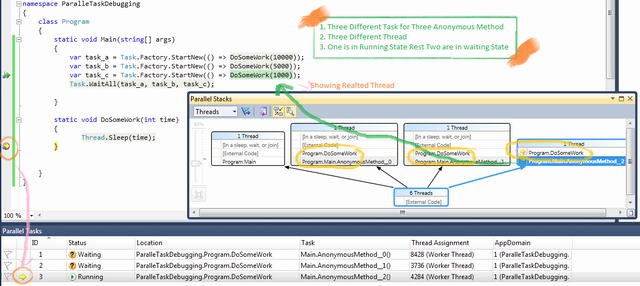
}

To understand the parallel program debugging, we need to be aware about two window options:

* **Parallel Tasks**
* **Parallel Stacks**

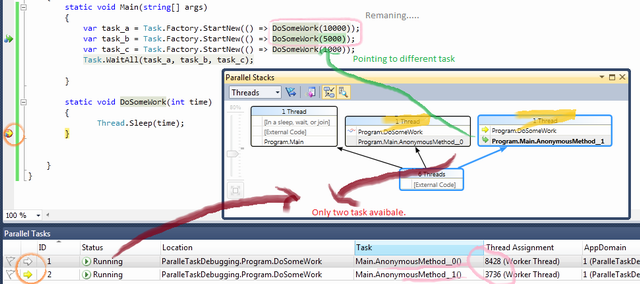
### Parallel Task and Parallel Stacks

Before continuing with parallel tasks and parallel stacks, you have to know about Threads Window which I have already covered. In the given code, you have three different tasks which are doing something and after sometime, all the tasks are put on hold. This is done intentionally to check the status of each task. To test, put a breakpoint on DoSomeWork() method and run the application. You will see your program execution paused on the breakpoint. After the program break, you can go to **Debug** > **Window** > Open **Parallel Tasks** and **Parallel Stacks** window. I asked to open both at the same time only because you can visualize what is going on.



***Figure: Breakpoint Filter - Multithreaded Debugging***

**Parallel Task** window will show you what are the different tasks that have been created for the program and what is their current status. On the other hand, Parallel Stacks will show you the graphical view of all thread creation, containing tasks, how they are related. If you click on the thread from the Parallel Stacks, it will show you the code line related with the thread (as shown in the picture with a Green Arrow). To move ahead, press F5. Let's see what comes next.



***Figure: Parallel Program - Debugging***

In the above diagram, you can find one of the tasks has been executed and the other two are remaining. Current execution point is set to AsyncMethod\_1, so if you continue, this method will execute first and next time the others. When you are working with parallel programming, there are many scenarios which will come like Deadlock, Dependency problem, etc. These topics is very interesting and long to discuss.

**Further study:**

<https://msdn.microsoft.com/en-us/magazine/ee410778.aspx>

## Debugging with IntelliTrace

IntelliTrace might be not available:

<https://blogs.msdn.microsoft.com/habibh/2009/10/22/intellitrace-is-not-available-why/>

This is another great feature of **Visual Studio 2010**. IntelliTrace Debugging is sometimes called as Historical Debugging. It operates in the background, records what you are doing during debugging. When you want the information of previous events, you can easily get it from IntelliTrace information, a past state of your application. In this mode, you can navigate to various recorded events. In this section, I will give you a basic overview of how to use IntelliTrace.

Here I am using one sample program by which I will show you what IntelliTrace does.

class Program

{

static void Main(string[] args)

{

Console.WriteLine("IntelliTrackerTest");

CallTestMethod(5);

}

public static void CallTestMethod(int TestValue)

{

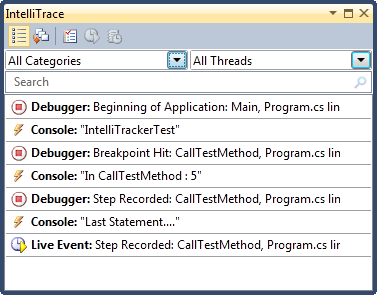
Console.WriteLine("In CallTestMethod : " + TestValue.ToString());

Console.WriteLine("Last Statement....");

}

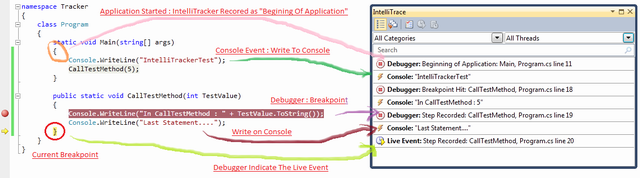
}

Run the program and Open **IntelliTrace window** from Debug menu, you will find the below screen added in the right-hand side of Visual Studio.



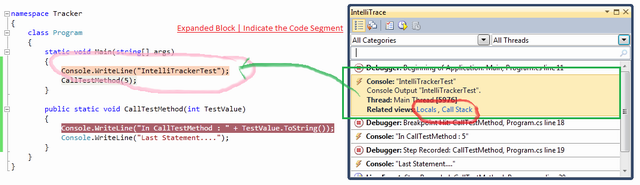
### Mapping with IntelliTrace

Did you find any relationship between your code and the **IntelliTracker List view**? Let's have a look into the below picture:

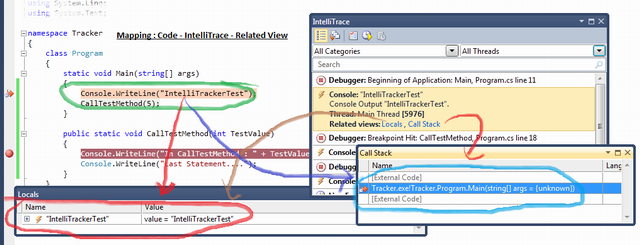


From the picture, I am sure you have got the idea what IntelliTrace is doing. Yes, exactly its **records/capture**what your code is doing. If you call a method, it will capture. If code fired an event, it will trap. At a single statement, I will capture each and everything.

Now from the IntelliTracewindow, you can navigate your code with any part and see what’s happened internally. If you can get the call stack, **Locals** variables information are recorded. To navigate, just click any of the events that you want to explore, that block will automatically expand.

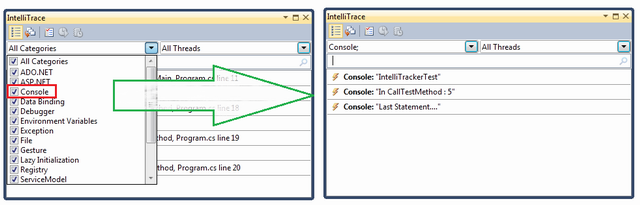


When you will select any block, it will expand automatically and that line will be highlighted. As per the given picture, I have highlighted the first console section and you can see the corresponding code block has also been highlight. You can now easily map them. In the expanded block (**Light Orange**), there are two Related View Sections **Locals**and **Stack Call**. I have already explained Locals and Stack Calls. But can you imagine what is the use of Locals and Call Stack over here. Yes, you guessed it correctly, it will show you the Locals and Call Stack status for that time period when the selected block has been executed. What a nice feature this is.



### Filter IntelliTrace Data

If you have a lot of recorded information in the IntelliTrace window, you can easily filter them out. You can filter data based on the **Categories**or **Thread**. As for example suppose you want to see only the data that is related with Console Related, you just need to check the "**Console**" From **categories**list.



**Note**: By default, Visual Studio stored IntelliTrace information in "*\Microsoft Visual Studio\10.0\TraceDebugging*" Location in a **iTrace** file. You may change the location.

Debugging with IntelliTrace is itself a big topic. So, it is very much difficult for me to cover them up within this article. I just give you the basic overview so that you can at least explore it by yourself now.

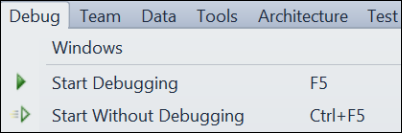
**Further study:**

<https://docs.microsoft.com/vi-vn/visualstudio/debugger/intellitrace?view=vs-2015>

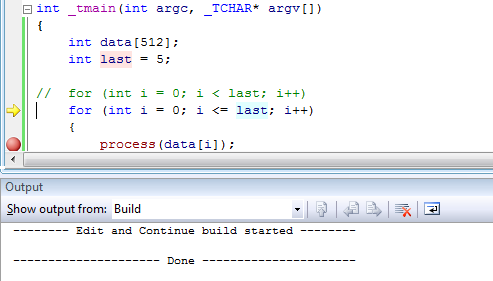
## Tips

### Start Without Debugging (Ctrl + F5)

Suppose you already set some breakpoints on your code, but you’re too lazy to disable these breakpoints, so you want to **start debugging without pausing at any of these breakpoints**. That’s when the Start Without Debugging comes to handy.



### Edit and Continue



Debugging a complex program, or a plugin? Found an error, but don't want to lose time stopping, rebuilding and restarting again and the function is called too often to use the previous trick each time? No problem, just fix the bug in-place and continue stepping. Visual Studio will modify your program and continue debugging with no need to restart.

However, edit-and-continue has a bunch of known limitations:

* First, it won't work for 64-bit code. If it refuses to work for your C# app, go to project settings, **Build** page, then select "x86" as **Platform Target**. Don't worry, the **Platform Target** for the Release configuration is separate from the debug one and can still be "Any CPU".
* Second, the edit-and-continue changes should be local, i.e., within one method. If you change the method signature, add new methods or classes, you'll have to restart the app, or undo the changes to continue. Changing methods containing lambda expressions implies modifying the auto-generated delegate classes and thus prevents continuing.

## Diagnostic Tools

[Diagnostic Tool Tutorial by MS](https://docs.microsoft.com/en-us/visualstudio/profiling/profiling-feature-tour?view=vs-2019)

## Useful Shortcut Key for Debugging

|  |  |
| --- | --- |
| **Shortcut Keys** | **Descriptions** |
| Ctrl + Alt + V, A | Displays the Auto window |
| Ctrl + Alt + B | Displays the Breakpoints dialog |
| Ctrl + Alt + C | Displays the Call Stack |
| Ctrl + Shift + F9 | Clears all of the breakpoints in the project |
| Ctrl + F9 | Enables or disables the breakpoint on the current line of code |
| Ctrl + Alt + E | Displays the Exceptions dialog |
| Ctrl + Alt + I | Displays the Immediate window |
| Ctrl + Alt + V, L | Displays the Locals window |
| Ctrl + Alt + Q | Displays the Quick Watch dialog |
| Ctrl + Shift + F5 | Terminates the current debugging session, rebuilds if necessary, and starts a new debugging session. |
| Ctrl + F10 | Starts or resumes execution of your code and then halts execution when it reaches the selected statement. |
| Ctrl + Shift + F10 | Sets the execution point to the line of code you choose |
| Alt + NUM \* | Highlights the next statement |
| F5 | If not currently debugging, this runs the startup project or projects and attaches the debugger. |
| Ctrl + F5 | Runs the code without invoking the debugger |
| F11 | Step Into |
| Shift + F11 | Executes the remaining lines out from procedure |
| F10 | Executes the next line of code but does not step into any function calls |
| Shift + F5 | Available in break and run modes, this terminates the debugging session |
| Ctrl + Alt + H | Displays the Threads window to view all of the threads for the current process |
| F9 | Sets or removes a breakpoint at the current line |
| Ctrl + Alt + W, 1 | Displays the Watch 1 window to view the values of variables or watch expressions |
| Ctrl + Alt + P | Displays the Processes dialog, which allows you to attach or detach the debugger to one or more running processes |
| Ctrl + D, V | IntelliTrace Event |

# Configuration Files

## File .vcxproj

This file contains information about:

* Version of Visual Studio IDE that generated the file
* Platforms
* Configurations
* Project features selected with the Application Wizard

This file is stored in an XML format.

General structure: <https://docs.microsoft.com/en-us/cpp/ide/vcxproj-file-structure?view=vs-2017>

### <ClCompile>

**<AdditionalIncludeDirectories>** (2 places)

C/C++ → General → Additional Include Directories

### <Link>

**<AdditionalLibraryDirectories>** (2 places)

Linker → General → Additional Library Directories

**<OutputFile>** (2 places)

Linker → General → Output File

**<AdditionalDependencies>** (2 places)

Linker → Input → Additional Dependencies

### <PostBuildEvent>

**<Command>**

Build Events → Post-Build Event → Command Line

### Others

**<OutDir>** (2 places)

Configuration Properties → General → Output Directory

**<IntDir>** (2 places)

Configuration Properties → General → Intermediate Directory

## File .vcxproj.user

**<LocalDebuggerCommand>**

Configuration Properties → Debugging → Command

**<LocalDebuggerWorkingDirectory>**

Configuration Properties → Debugging → Working Directory

## File .vcxproj.filter

This file contains information about the association between files in the project. This association is used in the IDE to show grouping of files with similar extensions under a specific node (e.g. "cpp" files are associated with the "Source Files" filter).

# Visual Leak Detector for Visual C++

<https://archive.codeplex.com/?p=vld>